Panduan Pengembang Web Modern

Dr. Budi Raharjo, S.Kom, M.Kom, MM.

Panduan Pengembang Web Modern

Dr. Budi Raharjo, S.Kom, M.Kom, MM.

YAYASAN PRIMA AGUS TEKNIK

PENERBIT :
YAYASAN PRIMA AGUS TEKNIK
Jl. Majapahit No. 605 Semarang
Telp. (024) 6723456. Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

ISBN 978-623-8642-67-0 (PDF)
9 786238 642670

Panduan Pengembang Web Modern

Penulis :
Dr. Budi Raharjo, S.Kom, M.Kom, MM.

ISBN : 978-623-8642-62-0

Editor :
Dr. Mars Caroline Wibowo. S.T., M.Mm.Tech

Penyunting :
Dr. Joseph Teguh Santoso, M.Kom.

Desain Sampul dan Tata Letak :
Irdha Yunianto, S.Ds., M.Kom.

Penebit :
Yayasan Prima Agus Teknik Bekerja sama dengan
Universitas Sains & Teknologi Komputer (Universitas STEKOM)

Anggota IKAPI No: 279 / ALB / JTE / 2023

Redaksi :
Jl. Majapahit no 605 Semarang
Telp. (024) 6723456
Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

Distributor Tunggal :
Universitas STEKOM
Jl. Majapahit no 605 Semarang
Telp. (024) 6723456
Fax. 024-6710144
Email : info@stekom.ac.id

KATA PENGANTAR

Puji syukur kami panjatkan kepada Tuhan Yang Maha Esa, atas segala rahmat dan karunia-Nya, sehingga kami dapat menyelesaikan buku ini dengan judul **"Panduan Pengembangan Web Modern"**. Karya ini disusun untuk memberikan pemahaman yang lebih mendalam mengenai perkembangan teknologi web terkini dan bagaimana penerapan berbagai konsep dalam pengembangan web dapat meningkatkan efisiensi, interaktivitas, dan pengalaman pengguna. Fokus utama dari karya ini adalah untuk menggali berbagai aspek dalam pengembangan web modern, termasuk penerapan teknologi terbaru, metodologi pengembangan, dan tantangan yang dihadapi dalam menciptakan aplikasi web yang responsif, aman, dan dapat diakses di berbagai platform.

Dalam beberapa tahun terakhir, teknologi web berkembang pesat, membuka banyak peluang di dunia digital. Pengembangan web modern tidak hanya fokus pada fungsionalitas situs, tetapi juga pada pengalaman pengguna, kecepatan, dan keberlanjutan aplikasi. Topik "Pengembangan Web Modern" mencakup penggunaan framework dan library terbaru, desain UI/UX, pengembangan berbasis API, serta prinsip pengembangan perangkat lunak yang efisien dan skalabel. Selain itu, pengembangan web modern juga menyoroti pentingnya keamanan, integrasi dengan cloud, dan optimasi untuk perangkat mobile.

BAB 1 membahas perkembangan teknologi web dalam bisnis digital, mulai dari bangkitnya web hingga munculnya web seluler, serta dampaknya terhadap desain aplikasi modern yang responsif di berbagai platform. BAB 2 menjelaskan penerapan metodologi agile dalam organisasi digital, termasuk cara mengidentifikasi persyaratan, mengelola bug, serta mengestimasi dan melacak pekerjaan untuk proyek pengembangan perangkat lunak yang efisien. BAB 3 menekankan pentingnya pengalaman pengguna (UX) dalam pengembangan perangkat lunak, membahas arsitektur informasi, pola-pola untuk optimasi UX, serta cara menerapkan prinsip UX dalam pengembangan aplikasi. BAB 4 mengulas tentang desain sistem end-to-end dalam pengembangan aplikasi, dengan fokus pada arsitektur sistem, identifikasi interaksi pengguna, desain modul, dan pengelolaan persyaratan lintas fungsional.

BAB 5 menyelidiki aspek etika dalam rekayasa perangkat lunak, termasuk privasi, beban kognitif, dan pentingnya membangun kepercayaan, serta implikasi etis dalam praktik pengkodean perangkat lunak. BAB 6 membahas komponen utama dalam pengembangan front-end aplikasi web, seperti HTML, CSS, desain responsif, mobile-first, dan optimisasi mesin pencari untuk meningkatkan pengalaman pengguna. BAB 7 menjelaskan berbagai pendekatan pengujian perangkat lunak, termasuk pengujian berbasis perilaku, manual, visual, dan lintas fungsional, untuk memastikan kualitas dan kinerja aplikasi. BAB 8 mengulas perkembangan JavaScript dari sisi server hingga peramban, termasuk callback, promises, async/await, serta penggunaan modul JavaScript dan JSON dalam pengembangan aplikasi web.

BAB 9 membahas prinsip-prinsip aksesibilitas dalam pengembangan web, cara bekerja dengan teknologi pendukung, serta pengujian aksesibilitas untuk memastikan aplikasi dapat diakses oleh semua pengguna. BAB 10 mengulas perbandingan antara API SOAP dan REST, cara mendesain dan mengamankan API, serta penerapan API berbasis peristiwa dalam pengembangan aplikasi web. BAB 11 membahas perbandingan basis data relasional dan NoSQL, serta cara mengelola dan melindungi data dalam sistem pengelolaan data yang

kompleks. BAB 12 memberikan panduan untuk membangun sistem aman, termasuk kepercayaan, rahasia, ekspresi reguler, serta daftar periksa keamanan dalam pengembangan perangkat lunak.

BAB 13 mengulas penerapan prinsip pengembangan perangkat lunak seperti aplikasi dua belas faktor, pengelolaan infrastruktur, serta penerapan pengiriman dan penerapan berkelanjutan (CI/CD). BAB 14 membahas pengintegrasian pengembang dan operasi (DevOps) untuk meningkatkan kolaborasi, serta tantangan faktor manusia dalam pengelolaan rotasi on-call dalam produksi perangkat lunak. BAB 15 mengajarkan pentingnya eksperimen, analisis hasil, dan pengembangan berbasis hipotesis untuk meningkatkan keterampilan pengembang perangkat lunak secara berkelanjutan.

Demikian buku ini dibuat, harapan penulis semoga Pembahasan dalam buku ini dapat memberikan wawasan tentang penerapan tren dan teknologi terbaru untuk menciptakan aplikasi web yang handal dan inovatif. Terima Kasih.

Semarang, Januari 2025
Penulis

Dr. Budi Raharjo, S.Kom, M.Kom, MM.

DAFTAR ISI

BAB 1
WEB MODERN

1.1 PENGEMBANGAN FULL STACK DALAM BISNIS DIGITAL

Komputer elektronik, Internet, dan World Wide Web adalah penemuan terbesar di zaman kita, dan kita beruntung dapat bekerja di puncak gelombang dampak penemuan ini terhadap masyarakat. Pengembang biasanya bekerja di perusahaan teknologi, atau di dalam departemen teknologi organisasi lain, tetapi karena pengembang awal kini telah pensiun, generasi terbaru mendapatkan pekerjaan di banyak bidang yang berbeda. Istilah "industri digital" sering digunakan untuk menggambarkan bisnis yang lebih tradisional yang menyadari potensi yang telah dibuka oleh para teknolog.

Bagi organisasi-organisasi ini, teknologi bukan hanya pusat biaya dan utilitas, tetapi bagian inti dari bisnis apa pun yang mereka geluti, bahkan jika layanan atau produk yang mereka sediakan tidak bersifat teknologi. Dengan munculnya bisnis digital inilah sifat pengembangan telah berubah. Jika produk dan layanan yang Anda sediakan tidak bersifat teknologi, tidak masuk akal lagi untuk menyelaraskan tim pengembangan Anda dengan teknologi, tetapi sebagai gantinya meminta tim pengembangan dan pengiriman Anda bekerja bersama non-teknisi dalam bisnis digital ini.

Bagi sebagian orang, "digital" mungkin hanya sekadar kata kunci, tetapi bagi mereka yang menggunakannya, perubahan yang disiratkannya sangat nyata. Bagi pengembang yang bekerja di tim digital ini, pengetahuan teknis yang mendalam kurang penting dibandingkan pengetahuan tentang bisnis dan organisasi, dan kecepatan penyampaian menjadi kunci. Meluncurkan sesuatu tiga minggu lebih awal dapat memberi organisasi digital keunggulan atas pesaingnya, dan dengan komunikasi sebagai salah satu beban terbesar dalam pengembangan, menjadi menguntungkan bagi tim pengembangan untuk dapat mengimplementasikan semua bagian dari fitur baru, atau memperbaiki bug yang memengaruhi banyak komponen, tanpa harus bernegosiasi dengan tim lain untuk melakukannya.

Pengembang yang bekerja di tim ini terkadang disebut berbentuk T, karena keluasan pengetahuan mereka sama pentingnya dengan kedalaman spesialisasi teknis yang mungkin mereka miliki. Demikian pula, ketika perangkat lunak gagal, memiliki tim yang mengoperasikan perangkat lunak yang terpisah dari tim yang membangunnya meningkatkan beban komunikasi. Tim digital menghilangkan beban komunikasi tersebut dengan mengadopsi cara kerja dan budaya yang dikenal sebagai DevOps, yang memadukan garis antara tanggung jawab pengembangan dan operasional.

Semua faktor ini telah memunculkan pengembang "tumpukan penuh". Bagi sebagian orang, istilah tumpukan penuh adalah istilah yang keliru. Jarang sekali melihat pengembang yang merasa nyaman menulis assembler x86 seperti mereka menggunakan Sass, tetapi bagi organisasi tempat para pengembang ini bekerja, tumpukan yang mereka pedulikan adalah yang memberi mereka nilai. Segala sesuatu di bawah ini dari bahasa pemrograman hingga OS hingga perangkat keras fisik adalah utilitas. Tim pengembangan dalam organisasi semacam ini

harus mampu menggunakan setiap bagian tumpukan yang penting.

Bagi sebagian organisasi, tumpukan memang meluas lebih jauh, tetapi itu hanya karena pada skala mereka, ada nilai lebih daripada menggunakan solusi siap pakai untuk tingkat yang lebih rendah, atau ada karena alasan warisan. Banyak organisasi masih menjalankan pusat data mereka sendiri, misalnya, karena investasi telah dilakukan, atau mereka memiliki persyaratan khusus (peraturan hukum, keamanan yang lebih tinggi, atau penyimpanan yang sangat besar), tetapi bagi banyak organisasi lain, menjalankan bagian tumpukan itu sendiri tidak menawarkan nilai lebih daripada membelinya sebagai utilitas.

Bagi organisasi digital ini, tim pengembangan tidak lagi dipenuhi oleh programmer. Output yang penting bagi tim ini bukanlah kode; melainkan solusi untuk masalah. Banyak dari masalah ini cenderung melibatkan interaksi dengan pengguna, yang telah menyebabkan munculnya Pengalaman Pengguna sebagai sebuah disiplin ilmu, atau mungkin hanya masalah proses, di mana analis bisnis yang berdedikasi dapat memberikan nilai tambah. Peran tradisional pengembang, QA, dan manajer proyek masih ada, tetapi sekarang kita memiliki banyak orang lain untuk membantu kita.

Luasnya pengembangan web modern sangat mengejutkan, dan akan sulit bagi semua pengembang dalam satu tim untuk sama-sama ahli di semua bagiannya. Kenyataannya, tidak ada yang namanya pengembang tumpukan penuh, tetapi tim pengembangan tumpukan penuh. Anda dapat membuat tim pengembangan tumpukan penuh dari spesialis individu, tetapi tidak akan pernah seefektif tim yang diisi oleh "generalis yang mengkhususkan diri". Pekerjaan datang dalam bentuk naik dan turun, dan tidak pernah didistribusikan secara merata ke semua bagian tumpukan. Oleh karena itu, tim pengembangan tumpukan penuh harus lebih besar daripada jumlah bagian-bagiannya, dan memiliki ciri-ciri yang sama dengan konsep Tim Berkinerja Tinggi dalam teori manajemen.

Tim spesialis juga kurang mudah beradaptasi dengan perubahan. LAMP tidak lagi sekeren dulu, MEAN adalah hal baru, dan bahasa-bahasa baru pada JVM menghidupkan kembali popularitas tumpukan Java, tetapi generalis khusus yang bekerja berdasarkan prinsip dan fondasi daripada detail dapat beradaptasi dengan perubahan ini lebih mudah daripada mereka yang memiliki pengetahuan mendalam tentang satu hal. Bisnis digital modern melihat tech stack sebagai keputusan taktis, bukan strategis. Mereka ingin dapat memanfaatkan utilitas terbaik untuk membantu mereka mencapai tujuan tersebut, daripada membangun dan mengandalkan platform teknis yang sulit diubah.

Bahasa dan teknik baru terus bermunculan, dan tim full stack yang baik harus diberdayakan untuk memilih alat terbaik untuk pekerjaan tersebut, dan harus dapat mengadopsi alat baru seiring dengan perkembangannya. Jadi, untuk siapa buku ini? Buku ini ditujukan untuk pengembang junior dan lulusan yang ingin memahami pengembangan web modern dalam bisnis digital. Buku ini ditujukan untuk teknisi dalam organisasi berstruktur tradisional yang sedang bertransisi ke dunia digital baru ini. Buku ini ditujukan untuk para pemimpin tim pengembangan yang ingin lebih memahami pekerjaan yang mereka pimpin. Buku ini ditujukan untuk para pengembang yang sudah berada di tim full stack ini yang ingin menyempurnakan keterampilan mereka.

Buku ini bukan untuk orang-orang yang tidak tahu cara membuat kode, atau yang baru memulai perjalanan pengembangan web mereka. Buku ini mengasumsikan pemahaman dasar tentang teknik pengembangan web (misalnya, Anda mungkin telah membangun setidaknya satu halaman web, atau API), tetapi tidak akan membahas detail implementasinya. Keindahan pengembangan tumpukan penuh adalah Anda dapat memilih sendiri bagian tumpukan, berdasarkan alat terbaik untuk pekerjaan tersebut, dan dalam lingkungan yang berubah dengan cepat, setiap rekomendasi konkret kemungkinan akan segera kedaluwarsa. Yang ingin diajarkannya adalah teknik yang berlaku untuk pengembangan modern, dan jebakan yang harus dihindari saat membangun aplikasi di web masa kini.

1.2 BANGKITNYA WEB

Minat terhadap Web tidak pernah setinggi ini. Kerangka kerja baru, perkakas yang lebih baik, dan evolusi standar muncul dari mana-mana, dan sulit untuk mengikutinya. Web modern terus berkembang, lebih banyak perangkat daripada sebelumnya yang dapat mengaksesnya dan berinteraksi dengannya, dan teknik terus berkembang untuk mengatasi hal ini. Penambahan kueri media ke CSS memungkinkan penataan halaman web untuk menargetkan layar dengan karakteristik tertentu, daripada membuat halaman terlihat sama di setiap perangkat.

Penyesuaian kecil ini secara mendasar mengubah cara situs dirancang, melahirkan serangkaian teknik baru untuk membangun situs web yang dikenal sebagai Desain Web Responsif, di mana desain satu halaman web harus merespons kemampuan perangkat yang digunakan untuk menampilkannya. Namun, CSS masih kekurangan beberapa fitur yang dibutuhkan untuk menjadikan desain web responsif berhasil satu-satunya informasi yang dapat kita gunakan dengan andal adalah ukuran layar dan kepadatan piksel, sedangkan ada banyak karakteristik lain yang akan memengaruhi desain jika tersedia, seperti metode interaksi (mouse, layar sentuh, ucapan, kendali jarak jauh lima titik, dll).

Jadi, Web masih merupakan format yang dibatasi. Terlepas dari batasan ini, metode distribusi Web telah menggantikan aplikasi desktop tradisional dalam banyak kasus. Pada masa-masa awal komputasi, pengguna berinteraksi dengan aplikasi di server melalui terminal, atau thin client. Ini sangat hebat, karena itu berarti aplikasi hanya berjalan di satu tempat, dan semua orang menggunakan tempat yang sama. Meningkatnya komputasi personal dan daya yang lebih besar pada desktop individual memungkinkan perpindahan ke klien besar, di mana semakin banyak logika bisnis berjalan dalam program yang didistribusikan ke desktop pengguna akhir.

Hal ini menyebabkan masalah dalam mendistribusikan aplikasi tersebut ke semua orang yang membutuhkannya serta mempertahankan versi yang berbeda pada perangkat individual. Itu adalah cara yang lambat untuk melakukan sesuatu. Ada banyak manfaat dari komputer yang menjadi lebih kecil dan lebih portabel, yang memungkinkannya muncul di rumah orang dan tidak hanya di fasilitas besar yang terhubung, tetapi dunia yang selalu terhubung masih jauh. Dengan munculnya Web, kita sekarang dapat memperoleh manfaat dari kedua dunia: server mendistribusikan perangkat lunak ke klien pada titik permintaan,

daripada perangkat lunak yang dimuat sebelumnya ke mesin.

Awalnya, kita kehilangan kemampuan untuk menggunakan aplikasi tanpa koneksi ke server, tetapi peningkatan konektivitas seluler membuat ini tidak menjadi masalah lagi, biasanya dengan mempertimbangkan koneksi internet sebagai status utama, kemudian menggunakan berbagai mode kegagalan atau penurunan yang terkontrol untuk menangani pemutusan sambungan (seperti pesan kesalahan, atau offline sementara). Baru-baru ini, Web telah mengadopsi dukungan yang lebih baik untuk bekerja secara offline, menggunakan fitur-fitur seperti pekerja layanan yang memungkinkan aplikasi untuk terus bekerja saat terputus.

Untuk waktu yang lama, standar Web dan kinerja browser jauh tertinggal dari aplikasi asli. Microsoft dan yang lainnya memperkenalkan alat-alat seperti ActiveX, applet Java, dan Flash untuk mengatasi hal ini, tetapi masing-masing pendekatan ini memiliki tantangannya sendiri. Microsoft-lah yang memulai kebangkitan dari apa yang pada akhirnya memungkinkan standar web digunakan untuk menghasilkan aplikasi yang akan menggantikan aplikasi desktop tersebut, dengan diperkenalkannya XMLHttpRequest API (XHR) ke JavaScript.

Kini, standar web telah matang hingga mampu memenuhi berbagai kebutuhan aplikasi yang kaya, dan pengembangan Web berlangsung dengan sangat cepat sehingga fungsionalitas yang hilang akan segera terlihat. Web telah menjadi alat untuk mendistribusikan aplikasi ini, dan teknologi web bahkan menggantikan bagian-bagian aplikasi asli, dengan aplikasi web yang dikemas untuk mekanisme distribusi asli.

1.3 WEB SELULER

Sama seperti aplikasi web yang menggantikan banyak aplikasi desktop, aplikasi asli pada platform seluler telah mengalami hal sebaliknya. Sebagian besar platform seluler utama kini memiliki mekanisme distribusi aplikasi bawaan. Distribusi Linux desktop juga telah memilikinya selama bertahun-tahun, tetapi Linux pada desktop tidak pernah mengalami terobosan yang diharapkan banyak penggemar. Di sisi lain, Apple dan Microsoft kini menghadirkan toko aplikasi ke OS desktop mereka sebagai perluasan dari toko seluler mereka. Mekanisme distribusi "toko aplikasi" ini memiliki dua keunggulan utama dibandingkan Web yang tidak pernah dimiliki aplikasi desktop: pasar yang efektif dengan mekanisme untuk mengenakan biaya untuk mendapatkan aplikasi, dan kemampuan untuk tetap berada di peluncur perangkat tersebut untuk mendapatkan ruang bagi pengguna.

Mekanisme mengenakan biaya juga merupakan salah satu kelemahan model distribusi, karena pemilik toko dapat menyertakan persyaratan yang tidak adil atau membatasi aplikasi tertentu dari toko, yang telah menyebabkan beberapa penerbit aplikasi mengabaikan toko tersebut. Situs web dan aplikasi web juga dapat ditambahkan ke peluncur, tetapi proses ini sering kali kikuk jika dibandingkan dan jarang digunakan. Tentu saja, aplikasi seluler, seperti aplikasi desktop sebelumnya, masih memiliki beberapa peningkatan kinerja saat dijalankan, serta tingkat akses yang lebih tinggi ke perangkat dan perangkat kerasnya. Namun, pasar seluler lebih terfragmentasi daripada pasar desktop, dan untuk memaksimalkan jangkauan, perangkat "lintas platform" sering digunakan untuk membangun aplikasi asli guna menargetkan beberapa perangkat.

Banyak contoh terkini dari hal ini sebenarnya hanyalah pembungkus untuk teknologi web. Bagi banyak organisasi, web tetap menjadi mekanisme distribusi default, kecuali jika ada persyaratan khusus yang hanya dapat dipenuhi oleh aplikasi asli dan pasar aplikasi, atau persepsi bahwa aplikasi asli tampaknya lebih unggul daripada aplikasi web terlalu menggoda untuk diabaikan. Keadaan HTML HTML adalah bahasa Web. Untuk membangun situs web, pada titik tertentu HTML akan terlibat. Standar HTML tidak hanya mencakup tag yang Anda tulis di halaman (markup), tetapi juga menentukan Model Objek Dokumen (cara Anda memanipulasi halaman web dari JavaScript) dan CSS juga.

Dan HTML sedikit rumit saat ini. Konsorsium World Wide Web (W3C) awalnya menetapkan standar HTML, dan merasa puas dengan standar HTML4. Fokus kemudian beralih ke XHTML, bentuk HTML yang sedikit berbeda berdasarkan standar XML. XML, Extensible Markup Language, menawarkan tingkat kekuatan yang besar misalnya, dokumen XML lainnya dapat disematkan di dalam dokumen HTML, tetapi karena XHTML terkenal tidak pernah didukung oleh Internet Explorer, kekuatan ini tidak pernah terwujud. Ada juga masalah lain dengan XHTML: parser XML yang sesuai standar menolak untuk merender apa pun jika ada masalah dengan XML, yang menghambat migrasi dari standar HTML4 yang jauh lebih longgar.

Pembuat peramban, yang paling terkenal adalah Opera dan Mozilla, tidak setuju dengan perpindahan ke XML ini, dan merasa frustrasi dengan pendekatan badan standar yang lambat, sehingga mereka mengambil tindakan sendiri dan membentuk WHATWG (Web Hypertext Application Technology Working Group) yang mengembangkan variasi standar HTML-nya sendiri. Mereka mengusulkan standar ini —HTML5— ke W3C, dan standar ini diadopsi, tetapi definisi HTML5 W3C dan WHATWG telah bergeser. WHATWG mendefinisikan HTML5 sebagai standar yang berlaku, yang berarti tidak akan pernah ada spesifikasi definitif HTML5 untuk diterapkan. Kedengarannya seperti mimpi buruk, bukan?

Untungnya, sebagian besar pembuat peramban kini telah bereaksi terhadap standar yang berlaku ini dan beralih ke model rilis yang jauh lebih sering, dan pengguna pun mengikutinya, sehingga waktu antara fitur baru dalam HTML yang diterima dan tersedia di peramban semakin singkat. Pengecualian terbesar adalah Apple, yang masih menautkan versi Safari dan Mobile Safari ke rilis OS X dan iOS (ini khususnya buruk pada iOS, di mana pengguna dibatasi untuk memasang runtime alternatif), dan beberapa vendor Android yang menggabungkan peramban mereka sendiri, bukan Google Chrome. Untungnya, ada banyak perkakas untuk membantu Anda di sini.

Sering kali, Anda dapat menulis HTML, CSS, atau JavaScript menggunakan teknik terbaru, dan ada perkakas di front end yang akan mengubah kode Anda dalam versi yang kompatibel dengan versi sebelumnya yang dapat menangani peramban lama untuk Anda. Untungnya, dengan markup HTML, sebagian besar peramban akan memperlakukan tag yang tidak dikenal sebagai tag generik, dan tag tersebut masih dapat ditata seperti itu (meskipun beberapa peramban lama memerlukan polyfill, seperti html5shiv.js yang terkenal). Selain itu, ada banyak pustaka JavaScript, yang dikenal sebagai "polyfill," yang ada untuk menyediakan versi JavaScript murni dari fitur baru apa pun dari bahasa JavaScript, yang memungkinkannya untuk digunakan pada peramban lama.

Ini dibahas secara terperinci dalam bab Front End. Dengan dua standar yang berbeda, sulit juga untuk mengetahui dokumentasi mana yang harus digunakan. Kenyataannya, yang terpenting adalah apa yang sebenarnya dilakukan oleh browser. Untungnya, masa perang browser di mana vendor yang berbeda akan menerapkan ide yang sama dengan cara yang berbeda sudah berakhir, dan ada lebih banyak kolaborasi antara pengembang browser untuk membuat standar berfungsi. Browser juga menunjukkan fitur nonstandar dengan "awalan vendor", di mana API JavaScript atau atribut CSS diawali dengan nama pembuat browser untuk menunjukkan bahwa itu adalah fitur khusus browser (fitur tersebut harus digunakan dengan hati-hati sebagai bagian dari pendekatan peningkatan progresif, atau dihindari sama sekali).

Jaringan Pengembang Mozilla adalah sumber daya yang sangat menyeluruh, dan situs web caniuse.com yang brilian akan memberi tahu Anda seberapa baik dukungan fitur tertentu, dan keanehan apa pun yang mungkin dimilikinya. Banyak situs yang dibangun saat ini harus berjalan di sejumlah besar browser, semuanya dengan versi dan tingkat dukungan yang berbeda, tetapi ada juga banyak yang tidak harus demikian. Jika Anda sedang membangun aplikasi internal untuk sebuah perusahaan, dan organisasi tersebut menggunakan Internet Explorer di Windows, sangat menggoda untuk membangun dan menguji hanya di browser tersebut, tetapi keadaan bisa tiba-tiba berubah misalnya, pengenalan tablet secara tiba-tiba ke dalam sebuah organisasi.

Menargetkan runtime yang dikenal dapat mempermudah hidup Anda, karena Anda akan memiliki lebih sedikit hal untuk diuji, dan Anda tidak perlu mendukung perangkat yang sangat lama/rusak, tetapi jangan terbuai oleh rasa aman yang salah tulislah sesuai standar, dan verifikasi ini dengan mengujinya di browser yang berbeda. Melakukan hal ini di awal dapat menyelamatkan Anda dan organisasi Anda dari banyak kesulitan di kemudian hari. Terkadang, Anda akan menemukan fitur yang ingin Anda gunakan yang tidak didukung oleh setiap browser yang ingin Anda targetkan.

Dalam kasus seperti ini, teknik yang dikenal sebagai peningkatan progresif sangat berguna. Peningkatan progresif adalah teknik di mana Anda memberikan versi dasar dari beberapa fungsi ke perangkat, lalu menguji apakah perangkat dapat mendukung alternatif yang lebih canggih, lalu mengaktifkan alternatif yang disempurnakan jika pengujian berhasil. Peningkatan progresif merupakan teknik yang berguna untuk memecahkan banyak masalah berbeda, dan dibahas lebih rinci dalam bab Front End.

Aplikasi vs. Situs Web

Dengan maraknya penggunaan Web sebagai platform pengiriman untuk apa yang dulunya merupakan aplikasi desktop, batasan antara apa yang dulunya disebut "situs web" dan apa yang sekarang disebut "aplikasi web" menjadi kabur. Situs web pada umumnya merupakan situs berbasis konten situs web dibuat untuk mengomunikasikan informasi, sering kali melibatkan beberapa interaktivitas, tetapi pada intinya, situs web adalah tentang informasi. Banyak kerangka kerja yang muncul untuk membantu membangun aplikasi web, dan kerangka kerja ini sering digunakan untuk membangun situs web juga, meskipun bukan untuk itu kerangka kerja tersebut paling cocok.

Angular dan Backbone adalah contoh dari jenis kerangka kerja ini, dan dioptimalkan

untuk saat data perlu dimodifikasi oleh pengguna aplikasi ini dengan menyediakan abstraksi untuk operasi ini. Abstraksi ini sering kali tidak membantu untuk situs web berbasis konten. Sebelum memulai proyek, Anda harus memutuskan apakah Anda akan membuat situs web atau aplikasi web. Banyak proyek yang cenderung menyertakan keduanya misalnya, katalog yang dijelajahi pengguna berfungsi paling baik sebagai situs, tetapi alat manajemen inventaris berfungsi paling baik sebagai aplikasi.

Anda dapat menganggap situs web sebagai bagian dari fungsionalitas aplikasi web yang lengkap, tetapi dengan mengidentifikasinya sebagai situs web murni, akan lebih mudah menggunakan serangkaian alat yang lebih terbatas yang sering kali akan memberi Anda hasil yang lebih baik, serta memungkinkan Anda mencapai hasil tersebut dalam waktu yang lebih singkat. Perangkat dan kerangka kerja aplikasi web mungkin tampak hebat, tetapi sering kali melanggar dasar penting Web - bahwa Web terdiri dari serangkaian halaman yang ditautkan.

Jika konten di situs Anda perlu ditautkan langsung dari URL, baik melalui berbagi URL di media sosial, atau dari mesin pencari, maka menganggapnya sebagai situs web, dan menghindari perangkat aplikasi web akan memungkinkan Anda memperoleh manfaat dari banyak fitur dasar Web (seperti URL yang dapat dibagikan, serta penyimpanan sementara dan pengarsipan).

Mengikuti Perkembangan

Pada saat buku ini diterbitkan, saya yakin beberapa contoh spesifik yang digunakan sudah ketinggalan zaman, tetapi teknik yang dibahas, sebagian besar, bukanlah hal baru, dan akan tetap berlaku untuk sistem selama bertahun-tahun mendatang. Yang akan terus berubah adalah cara penerapan teknik-teknik ini, dan alat, bahasa, serta kerangka kerja yang digunakan untuk menerapkannya. Saat kerangka kerja atau alat baru muncul, penting untuk mengevaluasinya dalam konteks apa yang sudah Anda ketahui. Sebagian besar akan menerapkan teknik yang sudah ada dengan cara baru (yang mungkin lebih baik), dan jika Anda dapat menarik persamaan dengan konsep yang sudah Anda kenal, akan lebih mudah untuk melihat bagaimana alat baru itu berguna (atau tidak).

Banyak alat baru yang tidak berguna! Sering kali, perangkat yang sudah matang dan sudah dikenal akan jauh lebih produktif daripada yang baru. Namun, penting untuk menyadari hal-hal baru yang mengilap itu, karena tiba-tiba akan muncul perangkat yang telah diadopsi dan matang secara signifikan, dan yang ingin Anda terapkan. Seperti biasa, penting untuk mencapai keseimbangan, dan hanya Anda dan tim Anda yang dapat membuat keputusan. Tim pengembangan dalam organisasi digital harus bereaksi cepat untuk mengikuti laju perubahan, dan masuk akal untuk mengadopsi alat baru jika alat tersebut menawarkan manfaat yang signifikan dibandingkan metode yang sudah ada untuk membantu Anda mencapai tujuan.

Penting juga untuk memeriksa apakah manfaat tersebut tidak langsung hilang dalam waktu dan energi yang terlibat dalam melakukan perubahan sejak awal. Menjaga hal-hal tetap kecil dapat mempermudah penerapan teknik-teknik baru ini, karena Anda dapat mengulangi perubahan dengan cepat untuk melihat apakah berhasil, dan bahkan mungkin memutuskan untuk tidak melakukan perubahan tanpa terlalu berkomitmen ini dikenal sebagai gagal dengan cepat. Konsep-konsep ini dibahas secara mendalam dalam bab Mendesain Sistem.

Hadiri konferensi, baca blog dan agregator, beli buku, ikuti orang-orang yang Anda hormati di media sosial, kunjungi grup pengguna lokal atau acara kumpul-kumpul, dan berkolaborasilah dengan pikiran terbuka. Seperti kebanyakan profesi yang membutuhkan keterampilan, ini bukanlah pekerjaan dari jam sembilan sampai jam lima. Itu tidak berarti Anda boleh terus-menerus bekerja lembur hanya untuk membuat produk; hanya saja Anda perlu menginvestasikan waktu Anda sendiri untuk karier Anda juga, dan "pengembangan profesional berkelanjutan" ini merupakan bagian penting dari itu. Jika atasan Anda tidak memberi Anda waktu atau ruang untuk melakukan ini, Anda mungkin bekerja di lingkungan yang tidak sehat.

1.4 KESIMPULAN

Munculnya pengembang tumpukan penuh tidak terjadi begitu saja. Organisasi bergerak menuju digital, di mana teknologi tertanam sepenuhnya di seluruh organisasi, daripada berada di departemen TI, dan pengembang tumpukan penuh dibutuhkan untuk memberi organisasi ini kelincahan dan keahlian yang mereka butuhkan. Oleh karena itu, pengembang tumpukan penuh harus menjadi generalis spesialis yang dapat menangani sejumlah tugas berbeda, dan bekerja dengan keterampilan dari disiplin ilmu lain.

Web telah menjadi tempat alami untuk mengembangkan aplikasi. Dari awalnya yang sederhana sebagai cara membaca dokumen, hingga ledakan alat dan teknologi browser yang tidak menunjukkan tanda-tanda melambat, Web memungkinkan Anda untuk menghadirkan aplikasi yang semakin kompleks ke banyak perangkat berbeda. Penting untuk memahami sejarah Web agar Anda dapat memilih alat yang tepat untuk aplikasi yang kompleks dan kaya yang dikenal sebagai aplikasi web, yang mungkin tidak sama untuk situs web berbasis konten, yang mengikuti pola interaksi yang berbeda.

BAB 2
MERENCANAKAN PEKERJAAN ANDA

2.1 AGILE DAN PERANANNYA DALAM ORGANISASI DIGITAL

Pengembang yang baru memulai sering kali percaya bahwa pekerjaan mereka adalah membangun apa yang diperintahkan oleh manajer atau klien mereka. Semakin berpengalaman seorang pengembang, semakin mereka menyadari bahwa membangun hal yang benar lebih penting daripada membangun hal yang benar. Peretasan cepat yang melakukan apa yang dibutuhkan pengguna jauh lebih baik daripada basis kode yang dibuat dengan indah yang tidak menyelesaikan masalah pengguna, meskipun sistem yang berfungsi tetapi diretas akan dengan cepat menghambat kemajuan.

Memiliki cara yang efektif untuk mengembangkan rencana untuk pekerjaan yang akan Anda lakukan adalah cara terbaik untuk memastikan bahwa Anda mendekati pekerjaan tersebut sehingga benar-benar memenuhi kebutuhan pengguna. Merupakan kesalahpahaman umum bahwa pengembang ada untuk menulis kode, tetapi tim pengembangan tanpa proses perencanaan yang efektif akan sering menghabiskan waktu menulis kode yang salah, atau terjebak dan tidak dapat melanjutkan dengan kode yang mereka tulis, tersandung rintangan dan menjadi korban risiko yang gagal mereka atasi. Proses perencanaan yang efektif, yang melibatkan seluruh tim, dapat meminimalkan atau menghindari masalah-masalah ini secara keseluruhan.

Kekhawatiran lain yang muncul dari proses perencanaan adalah bahwa proses ini sering kali juga digunakan untuk mengembangkan estimasi dan mengomunikasikan kepada pemangku kepentingan eksternal seberapa baik kinerja tim. Kekhawatiran ini sering kali dapat membuat proses perencanaan menjadi tidak efektif, dan membuat perencanaan secara keseluruhan memiliki reputasi yang buruk. Namun, adalah mungkin untuk menerapkan proses perencanaan yang menghindari jebakan-jebakan ini sekaligus membuat orang-orang di luar tim Anda tetap senang. Satu perbedaan utama antara organisasi digital dan organisasi tradisional adalah hubungan antara tim pengembangan dan bisnis.

Dalam organisasi tradisional, pengembang tinggal di tim mereka sendiri, sehingga tidak selalu memiliki pengetahuan domain seputar area bisnis tempat mereka mengembangkan perangkat lunak. Mengomunikasikan pengetahuan domain tersebut kepada pengembang itu sulit, jadi orang-orang yang benar-benar memiliki pengetahuan untuk menentukan kebutuhan perangkat lunak organisasi ditugaskan untuk mengomunikasikan informasi ini kepada tim pengembangan melalui dokumen dengan nama seperti "dokumen spesifikasi" atau "dokumen persyaratan fungsional". Karena membangun perangkat lunak bisa mahal, diyakini bahwa mendapatkan spesifikasi yang tepat sejak awal akan meminimalkan biaya pengerjaan ulang yang berpotensi mahal, jadi sejumlah besar upaya dilakukan untuk menyiapkan dokumen-dokumen ini. Masalahnya adalah dokumen-dokumen ini menjadi pengganti komunikasi antar anggota tim.

Lebih jauh, karena biaya overhead untuk menyiapkan dokumen-dokumen ini, dan

mendapatkan persetujuan, cukup besar, lebih masuk akal untuk memiliki proyek yang lebih sedikit dan lebih besar. Dokumen-dokumen tersebut selengkap dan setepat kode yang ditulis untuk mengimplementasikannya, tetapi memiliki kekurangan karena ditulis menggunakan ketidaktepatan bahasa manusia, dan proses membuatnya sulit diubah. Satu istilah yang belum saya gunakan sejauh ini dalam buku ini adalah "agile," tetapi pembaca yang jeli akan melihat esensi gerakan agile yang merayap ke bab sebelumnya.

Gerakan agile telah memiliki salah satu efek terbesar pada pertumbuhan bisnis digital dan perubahan struktur tim pengembangan. Proses yang dijelaskan di atas sering disebut model "waterfall", yang dicirikan sebagai kebalikan dari agile. Pada kenyataannya, tidak ada organisasi yang benar-benar bekerja dengan cara waterfall yang ketat. Dan bahkan tim agile terbaik pun akan memiliki semacam air terjun mini, meskipun air terjun mini dapat memberi masukan bagi dirinya sendiri; Anda tidak dapat menguji sesuatu hingga dibangun, dan Anda tidak dapat membangunnya hingga didefinisikan, tetapi Anda dapat mendefinisikan pengujian tersebut sebelum dibangun, dan membangun bagian dari sistem dapat menginformasikan bagian lain.

Inti dari gerakan agile adalah manifesto yang mendefinisikan agile sebagai upaya membawa orang kembali ke dalam proses membangun perangkat lunak, mendobrak hambatan komunikasi, dan merangkul perubahan. Kami menemukan cara yang lebih baik untuk mengembangkan perangkat lunak dengan melakukannya dan membantu orang lain melakukannya. Melalui pekerjaan ini, kami menghargai:

- Individu dan interaksi daripada proses dan alat
- Perangkat lunak yang berfungsi daripada dokumentasi yang komprehensif
- Kolaborasi pelanggan daripada negosiasi kontrak
- Merespons perubahan daripada mengikuti rencana
- Artinya, meskipun ada nilai pada item di sebelah kanan, kami lebih menghargai item di sebelah kiri.

Saat Anda membaca manifesto agile, penting untuk memahami apa artinya sebenarnya. “Individu dan interaksi daripada proses dan alat” tidak berarti “jangan gunakan Jira.” Artinya, jangan hanya menggunakan Jira, atau biarkan Jira menghalangi komunikasi. Paragraf ketiga dari manifesto agile sering terlupakan; ini bukan pilihan biner antara konsep kiri dan kanan, tetapi preferensi untuk konsep kanan. Jadi, miliki beberapa proses, dan gunakan alat, tetapi jangan biarkan hal itu menghalangi individu dan interaksi.

Dokumentasi dapat membantu komunikasi, jadi gunakan itu, tetapi jangan dokumentasikan setiap bagian dari apa yang Anda bangun sebelum Anda membangunnya. Kontrak itu penting itulah cara Anda dibayar tetapi membangun sesuatu yang tidak diinginkan pelanggan hanya karena kontrak mengatakan demikian adalah pemborosan waktu semua orang. Dan memiliki rencana membuat memulai menjadi sangat mudah, tetapi Anda harus memastikan rencana tersebut dapat diubah.Organisasi digital (berbeda dengan organisasi tradisional) bersifat agile karena mereka telah mendobrak hambatan komunikasi dengan mencampur tim bisnis dengan tim pengembangan.

Tanpa hambatan yang sewenang-wenang, proses yang lebih ringan untuk menentukan

apa yang perlu dilakukan dapat muncul. Ide-ide yang lebih baik juga muncul, karena komunikasi sekarang dapat mengalir bebas antara orang-orang dengan pengetahuan domain dan pengetahuan teknis, dan silo-silo ini pun runtuh. Dengan proses yang lebih ringan dengan overhead yang lebih rendah, setiap perubahan bisa lebih kecil, yang meminimalkan waktu antara mendefinisikan proyek dan benar-benar menyelesaikannya. Meminimalkan waktu ini berarti bahwa perubahan besar di tengah-tengah pembangunan untuk menanggapi beberapa perubahan eksternal atau penemuan baru tidak lagi diperlukan, karena ada lebih sedikit waktu untuk hal-hal untuk berubah sejak awal.

Ketika suatu perubahan benar-benar terjadi, dampaknya sendiri kemudian diminimalkan, karena itu memerlukan perencanaan ulang pekerjaan yang akan datang, atau dengan mengubah pekerjaan yang sedang berlangsung. Pekerjaan yang sedang berlangsung harus dibatasi agar kecil, sehingga dampak dari perubahan tersebut tidak besar. Agile bukanlah serangkaian proses, juga bukan serangkaian "ritual" seperti standup harian, cerita pengguna, atau retrospektif. Agile adalah cara berpikir, dan harus berkembang melampaui tim pengembangan dan ke seluruh organisasi. Tim yang "agile" dalam organisasi tradisional mungkin, paling banter, memiliki pendekatan pengiriman yang efektif dan memuaskan pemangku kepentingan internal, tetapi kecuali seluruh bisnis dapat beroperasi dengan cara yang agile dengan menerapkan prinsip-prinsip di atas, Anda akan dibatasi dalam seberapa cepat Anda dapat menyediakan perangkat lunak ke tangan orang-orang, alih-alih benar-benar membuat seluruh organisasi menanggapi perubahan, yang merupakan hal yang perlu dilakukan oleh bisnis modern.

Paling buruk, tim agile dalam organisasi tradisional akan terus-menerus berselisih dengan seluruh organisasi. Misalnya, organisasi mungkin menginginkan stabilitas dan visibilitas jangka panjang dari sistem TI mereka yang tidak dapat disediakan oleh agile; sifat dasar dari respons cepat terhadap perubahan berarti Anda tidak dapat menetapkan peta jalan enam bulan, kecuali Anda setuju untuk tidak mengubahnya, yang tidak lagi agile. Dalam organisasi tradisional, jika tim pengembangan tidak dapat beriterasi dengan bisnis, maka mereka tidak dapat menanggapi perubahan yang ditanggapi oleh seluruh bisnis. Lebih buruknya lagi, jika seluruh organisasi terkunci dalam proses perencanaan dan stabilitas jangka panjang ini, maka organisasi tersebut tidak dapat merespons perubahan dari luar organisasi, sehingga semakin sulit bagi setiap orang untuk mencapai tujuan mereka.

Menerapkan teknik agile dalam lingkungan tradisional seperti ini memang bisa dilakukan, tetapi jika tim diminta untuk membangun hal-hal besar sekaligus, proyek tersebut menjadi seperti proyek "air terjun", di mana potongan-potongan besar didefinisikan di awal, dan potongan terkecil yang bersedia diluncurkan oleh bisnis adalah keseluruhan produk. Pengembangan dan QA berlangsung cepat terhadap tumpukan pekerjaan ini, tetapi tumpukan pekerjaan itu sendiri tidak dapat berubah sebagai respons terhadap keadaan eksternal. Proses agile harus dimulai dengan ide awal untuk sebuah proyek atau fitur, bukan hanya pada saat pengembang mengambilnya.

2.2 MENGIDENTIFIKASI PERSYARATAN

Peran pengembang dalam mengidentifikasi persyaratan dapat sangat bervariasi di antara tim. Pada tingkat yang paling sederhana, mengidentifikasi persyaratan hal yang akan dibangun sering kali merupakan masalah meminta seluruh organisasi Anda untuk memberikannya kepada Anda, dan meskipun ini dapat bekerja dengan baik di organisasi yang matang, atau di mana ada hubungan pemasok/klien, Anda tidak dapat selalu berasumsi bahwa organisasi mengetahui apa yang mereka inginkan. Apa yang mereka minta belum tentu sama dengan apa yang sebenarnya mereka inginkan.

Banyak tim, terutama tim yang lebih besar, menunjuk seorang individu yang sebagian besar bertanggung jawab untuk ini. Terkadang, peran ini disebut analis bisnis, tetapi meskipun tidak ada orang yang memegang jabatan tersebut, peran tersebut akan ada di suatu tempat dalam tim Anda mungkin sebagai pemilik produk, atau manajer proyek, atau bahkan pimpinan teknologi. Pekerja lepas yang bekerja sendiri biasanya juga menempati peran ini. Lebih baik mengisi peran tersebut di dalam tim Anda daripada di luarnya, tetapi penting juga agar individu tersebut tidak menjadi silo atau jembatan bagi satu orang. Penting bagi pengembang untuk bekerja sama dengan mereka guna benar-benar memahami apa yang mereka buat, dan juga memberikan masukan untuk proses tersebut.

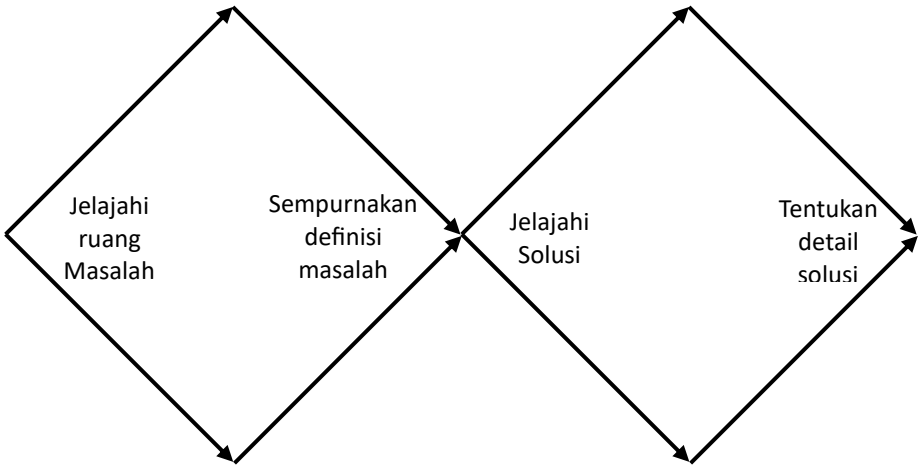
Jika Anda memahami apa yang ingin dicapai organisasi, Anda akan mengurangi waktu siklus masukan, sehingga Anda dapat menyelesaikan pekerjaan dengan lebih cepat. Jika Anda memiliki pertanyaan, daripada harus menunggu jawaban, Anda dapat membuat tebakan yang matang dan melanjutkan pekerjaan. Dalam kasus di mana pertanyaan tersebut lebih mendasar bagi apa yang sedang dibangun, maka berhenti dan bertanya tetap penting, tetapi dengan lebih memahami persyaratannya, Anda akan mengetahui pertanyaan mana yang benar-benar penting, dan mana yang tidak. Hal ini terutama penting bagi pengembang dan pimpinan senior, atau mereka yang ingin berkembang dalam peran tersebut.

Pengembang senior sering kali diharapkan menghadiri rapat dengan kelompok pemangku kepentingan yang lebih luas untuk membantu mendefinisikan produk dan memperoleh pemahaman tentang apa yang diinginkan para pemangku kepentingan. Awalnya mungkin tampak bahwa waktu yang dihabiskan untuk tidak membuat kode tidak seproduktif waktu yang dihabiskan untuk membuat kode, tetapi bagi pengembang tumpukan penuh, memperluas jangkauan di luar kode sebenarnya memungkinkan mereka untuk menghasilkan produk akhir yang lebih baik, dan membantu orang-orang di sekitar mereka bekerja lebih baik. Hari-hari persyaratan yang dilemparkan ke dinding bagi pengembang, dan kerugian yang menyertainya, sudah berakhir.

Lokakarya adalah salah satu cara yang paling umum untuk memperoleh persyaratan. Lokakarya melibatkan sekelompok orang bersama-sama dan menjalankan kegiatan, dengan harapan mendapatkan beberapa informasi yang berguna dari mereka. Orang-orang dari luar tim Anda yang menghadiri lokakarya sering disebut sebagai pemangku kepentingan; mereka adalah orang-orang yang memiliki minat terhadap apa yang Anda bangun. Untuk situs web berita, ini mungkin termasuk orang-orang seperti editor, atau untuk toko daring, ini mungkin termasuk tim pemasaran dan pembeli. Ini juga dapat mencakup orang-orang dari luar

organisasi Anda; dalam contoh e-commerce, sistem backend mungkin termasuk pemasok yang bekerja sama dengan Anda. Ini mungkin juga termasuk pengguna akhir Anda.

Format lokakarya ini bervariasi tergantung pada siapa pemangku kepentingan dan fase proyek yang sedang Anda jalani. Di awal, biasanya dimulai fase penemuan untuk memahami masalah yang ingin Anda selesaikan, daripada langsung mencari solusi. Sebagian orang mungkin menganggap ini anti-agile, tetapi penting untuk melakukan secukupnya agar tidak memulai perjalanan Anda ke arah yang salah. Anda tidak perlu memahami sepenuhnya ruang masalah dan sepenuhnya menentukan solusi Anda sebelum mulai membangun. Lokakarya dalam fase penemuan mungkin melibatkan sekelompok besar pemangku kepentingan, mengumpulkan semua orang di satu ruangan untuk menentukan masalah yang perlu dipecahkan. Para pemangku kepentingan mungkin tidak setuju tentang apa masalahnya, tetapi dengan meminta semua pemikiran mereka, Anda seharusnya dapat menarik beberapa tema umum yang akan memungkinkan tim pelaksana mengidentifikasi tantangan, dan menentukan serangkaian langkah untuk mengatasinya.



Gambar 2.1 Model Berlian Ganda, Yang Menunjukkan Cara Melakukan Divergensi Untuk Menghasilkan Keluasan Ide, Dan Kemudian Konvergensi Untuk Menghasilkan Kedalaman Pada Masalah Dan Solusi

Dalam bentuk model berlian ganda ini, Anda mengumpulkan semua orang untuk mengeksplorasi keseluruhan masalah, dan kemudian tim yang bertanggung jawab untuk merancang solusi menyempurnakannya menjadi pekerjaan aktual yang perlu dilakukan untuk memecahkan masalah pada tingkat tinggi. Pertama, Anda mengidentifikasi bentuk umum ruang masalah dengan terlebih dahulu mengeksplorasi keluasannya, dan kemudian menelusuri detail dan pertanyaan yang belum terjawab. Kemudian, berlian konvergen-divergensi kedua diikuti, dengan terlebih dahulu mengidentifikasi solusi yang memungkinkan, dan kemudian menghilangkan yang tidak sesuai dan mendefinisikan spesifikasi dan detail aktual untuk setiap bagian dari solusi.

Ingatlah bahwa tugas Anda adalah untuk benar-benar menemukan solusi; pemangku kepentingan mungkin paling memahami masalah mereka, tetapi belum tentu cara terbaik

teknologi dapat membantu mereka. Di sinilah model berlian ganda dapat membantu, karena melibatkan pemangku kepentingan untuk mendapatkan informasi yang relevan dari mereka guna merancang solusi yang tepat, tetapi kelompok inti (sering kali hanya anggota tim yang paling senior) kemudian menyempurnakannya menjadi apa yang benar-benar dibangun. Ini mencegah hasil desain-oleh-komite yang ditakuti, dan juga situasi yang paling terkenal yang dijelaskan oleh Henry Ford: "Jika saya bertanya kepada orang-orang apa yang mereka inginkan, mereka akan mengatakan kuda yang lebih cepat." Ada banyak teknik lain yang memanfaatkan model berlian ganda, dan banyak model dan cara lain untuk menjalankan lokakarya guna mengumpulkan persyaratan, tetapi ini adalah salah satu contohnya.

Bidang Pengalaman Pengguna (UX) telah berkembang dari campuran bidang akademis interaksi manusia-komputer dengan peran tradisional desain visual dan interaksi, dan banyak tim UX mengikuti pendekatan desain yang berpusat pada pengguna, yang menempatkan pengguna di jantung proses pengumpulan persyaratan. Seorang praktisi pengalaman pengguna adalah sekutu yang sangat kuat selama proses penggalian persyaratan. Banyak organisasi paling sukses di luar sana mengutamakan kebutuhan pengguna dan memiliki tim yang terdiri dari praktisi teknik dan praktisi UX yang bekerja berdampingan. Sering kali batasan peran tersebut kabur, dan beberapa praktisi UX berkontribusi pada teknik, atau pengembang mengambil bagian dalam proses UX, meningkatkan efektivitas tim melalui pengurangan biaya komunikasi.

Bentuk pengumpulan persyaratan ini dikenal sebagai penelitian kualitatif yaitu, penelitian ini berfokus pada fakta dan informasi abstrak serta perasaan manusia. Cara lain untuk memperoleh persyaratan berasal dari metode yang disebut penelitian kuantitatif, di mana data mentah dianalisis untuk menemukan informasi. Pengumpulan data kuantitatif paling efektif saat produk Anda berada di depan pengguna, dan dapat memberikan banyak kenyamanan bagi para pemangku kepentingan dibandingkan dengan penelitian kualitatif, yang sering kali didorong oleh "perasaan hati" atau informasi subjektif lainnya. Di sisi lain, manusia bukanlah makhluk rasional, dan banyak yang akan lebih mempercayai perasaan mereka daripada statistik!

Ada ilmu pasti di balik analisis data kuantitatif, dan penting untuk tidak meremehkannya. Statistik adalah subjek yang rumit, dan mudah untuk menerapkan bentuk analisis data yang tidak tepat, atau salah menafsirkan hasilnya, yang mengarah pada keputusan yang buruk. Saat bekerja dengan pengguna, data kuantitatif jauh lebih mudah dikumpulkan daripada data kualitatif, sehingga data kuantitatif dapat menjadi lebih bermanfaat di kemudian hari dalam pengembangan produk saat ada banyak pengguna aplikasi Anda di dunia nyata. Pengiriman berkelanjutan adalah teknik yang mendorong rilis lebih awal dan sering, dan untuk tim yang mempraktikkan pengiriman berkelanjutan, penggunaan data kuantitatif dapat menjadi jauh lebih efektif karena memungkinkan pengumpulan data saat bagian dari fitur baru dikembangkan. Hal ini dibahas lebih lanjut di bab-bab selanjutnya.

Bentuk data kuantitatif yang paling umum berasal dari penggunaan sistem analitik, yang melaporkan bagaimana pengguna berinteraksi dengan produk Anda, tetapi data tersebut juga dapat berasal dari tempat lain di organisasi Anda. Misalnya, jika membangun alat untuk

digunakan di pusat panggilan, melihat data seperti waktu tunggu rata-rata dapat terbukti mencerahkan saat menemukan persyaratan apa yang mungkin perlu Anda pertimbangkan dalam produk Anda.

2.3 MENDEFINISIKAN PEKERJAAN

Setelah apa yang perlu dilakukan suatu produk telah ditemukan, penting untuk mendefinisikan pekerjaan tersebut. Definisi ini berfungsi sebagai metode untuk mengomunikasikan kepada tim yang lebih luas apa yang perlu dilakukan dan sebagai cara untuk memastikan bahwa masalah benar-benar dipahami hingga titik di mana solusi dapat dinyatakan dalam kode. Di banyak tim, kisah pengguna telah menjadi hampir identik dengan agile, meskipun ada banyak cara lain untuk mendokumentasikan persyaratan, dan kisah pengguna juga dapat digunakan dalam pengaturan non-agile.

Kisah pengguna membingkai fitur baru dari sudut pandang pengguna, dan umumnya diungkapkan dalam bentuk:

a. Sebagai seseorang
b. Saya ingin melakukan sesuatu
c. Agar saya dapat mencapai sesuatu

Teknik populer lainnya termasuk pengembangan berbasis hipotesis, dokumen persyaratan fungsional yang terkenal, atau bahkan pendekatan yang lebih ad-hoc seperti teks bebas di bagian belakang kartu indeks. Sangat mudah untuk menulis kisah pengguna yang buruk. Ambil contoh:

- Sebagai pengguna
- Saya ingin mendaftar akun
- Agar saya dapat menyelesaikan transaksi lebih cepat di lain waktu.

Jenis persyaratan ini sering kali datang dari seseorang di dalam bisnis, bukan pengguna, dan kemudian diterapkan pada pengguna. Apakah pengguna benar-benar peduli untuk menyelesaikan transaksi lebih cepat? Mungkin untuk situs yang sering mereka gunakan, tetapi tidak untuk situs yang jarang diakses. Waspadalah terhadap persyaratan yang telah diterapkan pada pengguna; persyaratan tersebut sering kali tidak masuk akal, dan dapat mengakibatkan usaha yang sia-sia.

Membingkai kisah pengguna dengan benar dapat mengubah pendekatan Anda, dan membantu memenuhi tujuan yang diinginkan dengan lebih baik. Kisah pengguna di atas mungkin lebih masuk akal diungkapkan sebagai:

- Sebagai bisnis
- Saya ingin pengguna mendaftar akun
- Agar saya bisa mendapatkan lebih banyak wawasan tentang pelanggan saya, dan penelitian menunjukkan bahwa pelanggan yang memiliki akun lebih cenderung kembali ke situs.

Perombakan sering terjadi saat sulit menulis bagian "agar" dari kisah pengguna. Jika sulit menulis baris itu, nilainya mungkin tidak jelas dan perlu lebih didefinisikan. Kisah pengguna kedua di sini dapat lebih mudah ditantang, dan menjadi subjek diskusi yang lebih luas dengan

seluruh organisasi. Misalnya, apakah nilai tambah di sini lebih besar daripada biaya penambahan gesekan pada proses pembayaran? Selain itu, definisi data apa yang akan diambil berbeda antara kedua kisah tersebut.

Pertanda lain dari kisah pengguna yang tidak didefinisikan dengan baik adalah kisah yang hanya mengatakan, "Sebagai pengguna." Sebagian besar pengguna memiliki kebutuhan yang berbeda, dan dalam kisah pertama di atas, pernyataan tersebut tidak berlaku untuk semua pengguna hanya pengguna yang cenderung kembali ke situs. Saat membangun untuk pengguna, penting untuk memiliki pemahaman yang baik tentang siapa pengguna Anda dan perbedaan di antara mereka. Sangat sedikit sistem yang hanya memiliki satu jenis pengguna. Merupakan praktik umum untuk mengembangkan "persona" yang mewakili pengguna umum dan memberi mereka identitas agar perilaku mereka lebih mudah dipahami.

Terkadang, kisah pengguna sulit didefinisikan karena mencakup berbagai teknologi dan perjalanan pengguna terkait. Jenis kisah pengguna ini sering kali dapat diubah menjadi "epik", yaitu kisah pengguna yang telah dipecah menjadi (atau sedang dipecah menjadi) sejumlah kisah pengguna yang lebih kecil dan lebih terperinci. Epik sering kali muncul sangat awal, saat sebuah ide pertama kali muncul. Misalnya, jika Anda sedang membangun situs web e-niaga baru, sebuah epik mungkin ada dalam bentuk ini:

- Sebagai pelanggan,
- Saya ingin membeli barang dari situs web,
- Agar barang itu dapat dikirimkan kepada saya dan saya dapat menikmatinya.

Ini kemudian dapat dipecah menjadi kisah pengguna yang lebih kecil berdasarkan mekanisme pembelian sesuatu yang berbeda. Misalnya, kisah pengguna mungkin muncul seputar pencarian katalog untuk menemukan barang yang akan dibeli, atau berbagai cara membayar barang. Kisah pengguna individual ini dapat lebih mudah untuk dipikirkan dan diprioritaskan. Misalnya, jika Anda berpotensi memiliki pengguna di luar negara Anda, Anda akan sering perlu membuat pencari alamat, atau memiliki kalkulator biaya pengiriman untuk negara-negara tersebut, tetapi fungsi tersebut dapat dikesampingkan untuk fokus pada pasar lokal terlebih dahulu dan kemudian dilanjutkan di kemudian hari.

Mencoba menangani cerita pengguna yang terlalu besar dapat menimbulkan masalah, karena lebih sulit untuk melihat apa yang terlewat, dan dapat menyebabkan pekerjaan yang tidak perlu karena membawa serta fitur-fitur yang tidak perlu. Bill Wake mengusulkan mnemonik INVEST untuk mengingat karakteristik penting dari cerita pengguna yang baik.

- ❖ Independen
- ❖ Dapat dinegosiasikan
- ❖ Berharga
- ❖ Dapat diperkirakan
- ❖ Kecil
- ❖ Dapat diuji

Kisah pengguna yang independen adalah kisah yang dapat dicapai tanpa harus mengerjakan kisah pengguna lain terlebih dahulu. Ini berarti Anda dapat memprioritaskan ulang kisah Anda kapan saja, dengan memindahkannya ke atas atau ke bawah daftar tugas sebagai respons

terhadap perubahan apa pun dalam organisasi Anda. Dapat dinegosiasikan berarti bahwa sebuah kisah dapat diubah, ditulis ulang, atau dibuang kapan saja hingga pekerjaan dimulai.

Misalnya, jika sebuah kisah yang diajukan memiliki implikasi teknis yang signifikan, pengembang harus dapat menantang kisah tersebut dan menentukan apakah ada cara yang lebih sederhana untuk mengimplementasikannya yang masih memenuhi kebutuhan bisnis. Kisah pengguna harus bernilai agar masuk akal. Kisah yang memerlukan pekerjaan yang sibuk, atau item teknis yang menyenangkan (misalnya, mencoba teknologi baru yang canggih) tetapi sebenarnya tidak memberikan nilai bagi organisasi tidak memiliki nilai sebagai kisah, jadi tidak boleh ada di daftar tugas.

Beberapa tim melangkah lebih jauh dari sekadar mengharuskan sebuah kisah bernilai dan menetapkan "poin nilai" ke sebuah item, yang merupakan konsep abstrak yang mirip dengan poin kisah yang dapat membantu memprioritaskan item. Item bernilai tinggi/biaya rendah cenderung menjadi prioritas tertinggi, sedangkan item bernilai rendah/biaya tinggi dapat dibuang. Inilah sebabnya mengapa cerita pengguna juga harus dapat diperkirakan. Jika cerita pengguna tidak dapat diperkirakan, sulit untuk memprioritaskannya. Hal ini juga menunjukkan adanya masalah mendasar pada cerita tersebut. Jika sesuatu tidak dapat diperkirakan, mungkin karena cerita tersebut tidak cukup jelas atau terdefinisi.

Ini juga biasanya berarti cerita tersebut tidak cukup jelas atau terdefinisi bagi tim untuk dapat menerapkannya. Kecil adalah karakteristik penting lainnya dari cerita pengguna. Banyak tim akan menentukan aturan praktis tentang seberapa besar cerita tersebut. Tim yang berlari dalam sprint sering kali akan menyatakan bahwa cerita harus cukup kecil untuk diselesaikan dalam satu sprint, dan yang lain akan menggunakan aturan praktis yang berbeda berdasarkan pada apa yang mereka rasa nyaman. Ketika cerita besar, lebih mudah untuk melewatkan detail halus di dalamnya, yang dapat menyebabkan usaha yang sia-sia, dan juga menjadi lebih sulit untuk menyadari bagian-bagian cerita yang mungkin memiliki nilai lebih cepat, karena seluruh cerita harus didefinisikan. Secara anekdot, cerita besar menyebabkan masalah terbesar dalam tim pengiriman.

Persyaratan terakhir INVEST adalah bahwa cerita pengguna harus dapat diuji. Jika sebuah cerita tidak dapat diuji, itu berarti tidak ada cara untuk mengetahui apakah yang telah diterapkan oleh tim itu benar, atau apakah itu memiliki dampak yang tepat. Testabilitas di sini sering merujuk pada pekerjaan yang dilakukan oleh QA atau pengujian otomatis dalam sebuah tim, tetapi menjadi lebih banyak digunakan untuk merujuk pada cara mengukur dampak dari pengiriman fitur tersebut pada sebuah organisasi (dan dengan demikian memeriksa apakah cerita tersebut memiliki nilai). Ketika sebuah persyaratan tidak dapat diuji, itu sering kali disebabkan oleh kurangnya kejelasan tentang apa yang seharusnya dilakukan dan bagaimana seharusnya berperilaku. Kurangnya testabilitas juga dapat berarti bahwa nilainya tidak terdefinisi dengan baik.

Persyaratan ini terkadang dapat saling bertentangan. Sering kali, untuk membuat cerita pengguna menjadi independen, Anda harus membuatnya cukup besar. Hal ini terutama berlaku saat Anda mulai membangun fondasi suatu sistem. Item pertama yang Anda bangun mungkin mengharuskan Anda menyiapkan perkakas pembangunan, atau memiliki

ketergantungan pada sistem lain misalnya, sistem login. Beberapa tim memilih untuk mengatasi hal ini dengan mengomunikasikan bahwa akan ada "overhead cerita pertama" pada item pertama yang mereka bangun, atau dengan mengadakan "sprint pondasi" untuk menyiapkan berbagai hal. Ada pula tantangan di luar tahap pondasi awal ini. Misalnya, jika sekelompok cerita yang terkait tetapi independen muncul, biasanya cerita pertama dalam kelompok ini memerlukan serangkaian tugas teknis yang lebih besar, yang kemudian membuat penerapan cerita selanjutnya menjadi lebih mudah.

Hal ini dapat menantang estimasi cerita, karena urutan pengerjaannya akan mengubah estimasi tersebut. Contoh lain terjadi saat pembangunan produk minimum yang layak (MVP) sedang berlangsung. MVP adalah hal minimum yang harus dilakukan suatu produk agar bermanfaat atau bernilai, jadi menurut definisinya, produk tidak memiliki nilai hingga fase MVP ditulis. Oleh karena itu, semua cerita dalam MVP menjadi saling bergantung satu sama lain, dan suatu item dengan sendirinya mungkin tidak dianggap bernilai (misalnya, mengapa harus memiliki kemampuan menelusuri katalog hingga Anda memiliki cara untuk memeriksanya?).

Pada tahap awal ini, memahami mnemonik secara harfiah sulit, meskipun seiring produk berlanjut melampaui MVP ke pengembangan berkelanjutan atau BAU (bisnis seperti biasa), menjadi jauh lebih mudah untuk berpegang pada prinsip-prinsip tersebut, dan melakukannya menjadi pendorong utama untuk pengiriman berkelanjutan. Ketika cerita baru diajukan, cerita tersebut tidak selalu memenuhi persyaratan INVEST dengan segera. Cerita baru sering kali menjadi penanda untuk menunjukkan lebih banyak pekerjaan yang diperlukan untuk mengidentifikasi nilai, atau menemukan informasi yang cukup untuk membuatnya dapat diperkirakan. Merupakan hal yang umum untuk memiliki fase "ide" atau "penemuan" dalam proses perencanaan Anda, tempat cerita-cerita ini disempurnakan sebelum beralih menjadi bagian penuh dari backlog Anda.

Setelah definisi tingkat tinggi dari suatu persyaratan selesai dan muncul di daftar tunggu, selanjutnya menjadi penting untuk mendefinisikan sisa pekerjaan yang diperlukan agar siap untuk diimplementasikan oleh tim. Jenis definisi yang diperlukan akan bervariasi di antara tim. Tim yang relatif belum matang akan memerlukan definisi yang lebih banyak sebelum beralih ke fase berikutnya, sedangkan tim yang lebih matang dapat bekerja dengan lebih sedikit detail. Mendapatkan tingkat detail yang tepat itu sulit! Jika terlalu banyak, Anda akan membuang-buang waktu dan tenaga dalam menentukan spesifikasi yang berlebihan; jika terlalu sedikit, Anda berisiko mengerjakan ulang atau membuat fitur yang salah.

Saat pekerjaan berlanjut dari fase ide ke tahap produksi, diperlukan tingkat detail yang berbeda-beda. Sebagai pengembang, Anda mungkin memiliki ekspektasi tertentu tentang apa yang diberikan kepada Anda, artinya sering kali ada tahapan sebelum tahap pengembangan (seperti desain visual) yang harus diselesaikan. Karena definisi ini terjadi di dalam tim Anda, penting untuk memiliki komunikasi yang baik dan ekspektasi yang jelas tentang apa yang terjadi selama setiap fase. Pengembang senior atau utama dapat dilibatkan lebih awal untuk membantu anggota tim lainnya (seperti praktisi UX) memahami batasan teknis apa pun pada proses tersebut, serta mengidentifikasi cara alternatif untuk memecahkan masalah.

Sebagai pengembang, tingkat definisi yang paling umum diperlukan adalah serangkaian kriteria penerimaan, dan sering kali beberapa wireframe atau aset visual jika ada komponen UI yang terlibat. Namun, tidaklah membantu jika hal-hal ini muncul tiba-tiba dalam bentuk yang telah dipoles dengan sempurna; akan berguna bagi tim pengembangan untuk memiliki visibilitas pada pekerjaan yang akan mereka lakukan, sebelum pekerjaan tersebut didefinisikan sepenuhnya.Metode mengomunikasikan definisi ini kepada tim pengembangan juga penting, dan sangat bergantung pada cara tim tersebut bekerja. Salah satu metode efektif untuk melakukan ini disebut pengembangan berbasis perilaku, yang mengandalkan campuran percakapan dan catatan tertulis untuk menetapkan kriteria penerimaan ini.

Pengembangan berbasis perilaku dibahas lebih rinci dalam bab Pengujian. Komunikasi verbal adalah metode yang jauh lebih kaya bagi manusia untuk berkomunikasi daripada komunikasi tertulis, dan meskipun definisi sering kali datang kepada pengembang dari peran lain dalam tim, itu bukanlah satu-satunya arah komunikasi yang dapat mengalir. Mengajukan pertanyaan dan, untuk peran lain dalam tim, siap menjawab pertanyaan tersebut—sama pentingnya dengan menghasilkan definisi yang baik sejak awal! Ada banyak tim efektif yang menggunakan komunikasi verbal daripada catatan tertulis, dan meskipun ini sering menyebabkan masalah dengan skala, ini dapat bekerja dengan sangat baik. Ingat manifesto tangkas: "Orang-orang lebih penting daripada proses."

2.4 MELACAK PEKERJAAN

Bahkan tim satu orang pun akan sering merasa perlu melacak pekerjaan yang sedang dilakukan. Melacak pekerjaan sering kali menjadi ladang ranjau, dengan banyak orang yang berbeda ingin pekerjaan dilacak karena berbagai alasan. Tim mungkin ingin pekerjaan dilacak sehingga mereka dapat memiliki visibilitas yang lebih baik tentang apa yang dilakukan setiap orang; Manajer eksternal mungkin ingin melacak efektivitas satu tim relatif terhadap tim lain; pemangku kepentingan mungkin ingin tahu kapan fitur yang mereka pedulikan selesai; dan manajer proyek mungkin ingin tahu berapa lama orang menghabiskan waktu pada setiap fitur sehingga mereka dapat menagih pelanggan dengan tepat.

Namun, membuat laporan dan tampilan alternatif untuk memenuhi semua kebutuhan yang berbeda ini dapat menjadi beban yang cukup besar bagi sebuah tim, sehingga banyak yang menggunakan alat untuk mengotomatiskan sebanyak mungkin pekerjaan ini. Meskipun banyak klaim, tidak ada "satu alat yang benar-benar tepat" untuk melacak pekerjaan. Sebuah tim yang menggunakan Jira mungkin menganggap Trello sama sekali tidak cocok untuk apa yang ingin mereka capai, dan tim yang menggunakan dinding fisik dengan kartu indeks mungkin tidak akan pernah dapat menerapkan proses mereka ke dalam tim yang merangkul kerja jarak jauh.

Apa pun alat yang Anda gunakan, Anda tidak mungkin menemukan sesuatu yang sempurna untuk semua orang, dan Anda mungkin harus membuat beberapa kompromi untuk mendapatkan yang paling cocok untuk semua orang. Metode yang digunakan untuk melacak pekerjaan harus ditinjau ulang untuk memastikannya berfungsi untuk semua orang, dan tidak menyebabkan pekerjaan yang tidak perlu. Selain alat visual yang dapat Anda gunakan untuk

mengelola pekerjaan Anda, ada alat lain yang dapat digunakan untuk mengelola aliran pekerjaan ke tim Anda. Kadang-kadang disebut metodologi, ada banyak yang dapat dipilih satu opsi populer dalam komunitas agile adalah Scrum. Beberapa metodologi atau kerangka kerja yang lebih luas, seperti PRINCE2, mencakup berbagai teknik lain (seperti pengumpulan dan pengujian persyaratan), selain proses menjalankan proyek.

Scrum

Scrum telah menjadi sangat menonjol di bidang pengembangan perangkat lunak, hingga Scrum menjadi sinonim dengan agile di beberapa komunitas. Namun, Scrum bukanlah peluru ajaib, dan tidak sempurna. Gerakan agile secara umum, dan Scrum secara khusus, tumbuh dari dunia Extreme Programming (XP), dan salah satu kritik terhadap Scrum adalah bahwa dengan membuang aspek teknis XP, hal itu mengakibatkan proses yang lebih lemah. Yang lain adalah bahwa fokus pada "ritual" dalam Scrum (seperti sprint berdurasi tetap, standup harian, perencanaan, dan retrospektif) telah memunculkan gerakan kultus kargo, di mana organisasi "mengadopsi" Scrum dengan melakukan ritual ini, tetapi masih berada dalam gerakan waterfall yang lebih besar.

Hal ini diperkuat oleh menjamurnya pelatih dan sertifikasi Scrum yang mengajarkan Anda ritual dan prinsip Scrum, tetapi tidak mengajarkan cara menerapkannya dalam konteks organisasi Anda. Hal ini dapat menyebabkan hal terburuk dari kedua dunia dan kekecewaan terhadap agile secara umum. Saat menerapkan Scrum, penting untuk diingat bahwa Anda dapat menyesuaikan kerangka kerja dengan keadaan Anda, dan Anda tidak perlu mengikuti definisi Scrum yang baku. Inti dari Scrum adalah beberapa artefak, peran, dan ritual utama. Hal terpenting adalah backlog, yang merupakan daftar prioritas hal-hal yang harus dikerjakan.

Dalam Scrum, tim pengembangan bekerja dalam sprint dengan durasi tetap (sering kali antara satu dan empat minggu, tetapi durasinya tidak berubah, dan durasi sprint disetujui di awal), dan menyetujui di awal sprint berapa banyak item yang akan diterima dari backlog ke sprint. Pada akhir setiap sprint, tim pengembangan harus menghasilkan "peningkatan yang berpotensi dapat dikirim" dari produk, sehingga nilai ditambahkan secara berkala. Tim Scrum terdiri dari pemilik produk, yang harus diberdayakan untuk membuat keputusan seputar produk yang sedang dibangun dan bertanggung jawab untuk membangun backlog, dan harus dapat diakses oleh tim pengembangan; seorang Scrum Master, yang bertanggung jawab untuk memfasilitasi Scrum dalam tim dan bekerja dengan pemilik produk dan pemangku kepentingan untuk menghasilkan backlog, dan yang dapat memfasilitasi penghapusan hambatan atau "penghambat" yang menghentikan pekerjaan dalam sprint; dan tim pengembangan itu sendiri.

Tim pengembangan harus lintas fungsi dan mampu melakukan semua pekerjaan untuk mengirimkan item backlog dalam sprint. Sprint dimulai dengan sesi perencanaan. Pada paruh pertama sesi perencanaan sprint, pemilik produk dan tim pengembangan menegosiasikan item backlog mana yang harus diterima dalam sprint—idenya adalah bahwa tim hanya menerima item yang menurutnya dapat diselesaikan dalam sprint. "Definisi selesai"—kesepakatan tentang apa arti sebenarnya dari item yang diselesaikan—ditentukan oleh tim Scrum. Umumnya, "definisi siap" juga ada, yang mendefinisikan apa artinya item backlog

memiliki definisi yang cukup bagi tim untuk menerimanya.
Agar suatu fitur dianggap siap untuk dikembangkan, hal-hal berikut harus lengkap:

- Pemilik produk telah memprioritaskannya.
- UX, produk, dan pengembangan telah mengembangkan dan menyetujui kriteria penerimaan untuknya.
- Tim pengembangan telah memperkirakannya (atau memperkirakan ulang, jika cakupan item telah berubah). • Pimpinan teknis senang dengan pendekatan yang diusulkan, jika pendekatan tersebut berdampak pada arsitektur teknis.
- Jika fitur tersebut melibatkan perubahan apa pun pada alur kerja administrator, administrator telah diajak berkonsultasi.
- Jika fitur tersebut memerlukan perubahan apa pun pada skema data, arsitek data telah meninjaunya.
- Ketergantungan apa pun pada API eksternal telah diidentifikasi.
- Jika ada aset UI baru yang diperlukan, aset tersebut telah diidentifikasi.

Agar suatu fitur dianggap selesai, hal-hal berikut harus lengkap:

- Kode harus sesuai dengan panduan gaya pengodean.
- Semua kode baru harus memiliki pengujian unit yang ditulis terhadapnya, dan semua pengujian unit harus lulus.
- Pimpinan teknis harus telah meninjau dampak pada persyaratan lintas fungsi produk.
- Kode harus telah ditulis oleh pasangan, atau ditinjau oleh rekan sejawat.
- Desainer UX dan pemilik produk harus telah melihat fitur tersebut di lingkungan QA.
- Penguji harus telah menjalankan fungsionalitas dan menguji produk secara menyeluruh di lingkungan QA.
- Pengujian otomatisasi berbasis browser harus lulus di semua browser yang didukung.
- Semua tiket bug yang diidentifikasi oleh QA manual seharusnya sudah diselesaikan.

Teknik estimasi yang dikenal sebagai "story pointing" dan "velocity tracking" digunakan untuk membantu menentukan seberapa banyak pekerjaan yang dapat diterima tim dalam sprint ini dibahas lebih rinci di bagian Prioritization. Di paruh kedua sesi perencanaan sprint, pengembang memecah cerita individual menjadi tugas-tugas yang masuk akal bagi tim tersebut. Untuk tim yang benar-benar lintas fungsi, tugas-tugas ini sering kali dapat berupa hal-hal seperti "membuat kriteria penerimaan," "menghasilkan wireframe," atau "menambahkan titik akhir API baru ke backend."

Salah satu kelemahan Scrum adalah ia mengharapkan seluruh tim untuk dapat mengerjakan tugas-tugas ini, tetapi pada kenyataannya, ada spesialisasi dalam sebuah tim, sehingga tugas-tugas wireframe mungkin mengharuskan seorang desainer untuk mengerjakannya sebelum pengembang dapat mengambilnya. Sementara sprint berlangsung, setiap hari harus dimulai dengan standup harian, di mana pemilik produk, Scrum Master, dan tim pengembangan berkumpul di sekitar papan tugas dan membahas apa yang mereka lakukan hari sebelumnya, apa yang mereka rencanakan untuk dilakukan hari itu, dan apakah ada sesuatu yang menghentikan mereka (penghambat).

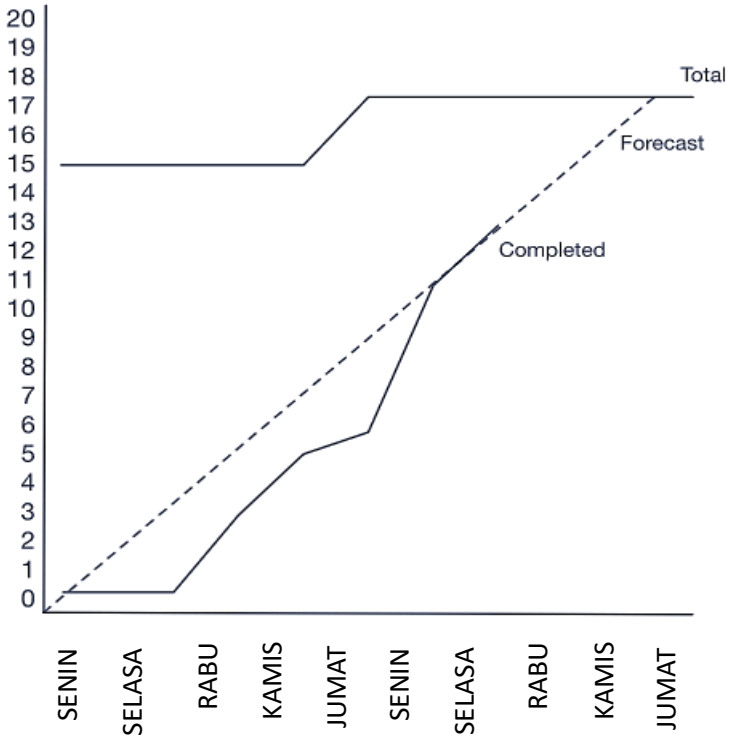
Setiap orang harus dibatasi satu menit, dan hanya satu orang yang boleh berbicara

pada satu waktu. Pendekatan alternatif adalah dengan "berjalan di papan", jadi alih-alih individu memberikan pembaruan, diskusi difokuskan pada perkembangan tugas individu atau item yang tertunda. Standup disebut demikian karena berdiri (bagi mereka yang bisa), dibandingkan dengan duduk, membuat rapat tetap singkat. Standup tidak dimaksudkan untuk berfungsi sebagai pembaruan bagi manajemen, dan hanya mereka yang merupakan bagian dari tim inti yang boleh berbicara, meskipun yang lain diizinkan untuk mengamati.

Dahulu kala ada seekor babi dan seekor ayam. Ayam menoleh ke babi dan berkata, "Kita harus membuka restoran. Kita bisa menamainya 'Ham & Eggs.'" Babi tidak setuju, berkata, "Kamu akan terlibat, tetapi aku akan berkomitmen penuh". Terkadang anggota tim inti Scrum di standup disebut babi, dan pengamat serta pemangku kepentingan disebut ayam. Hanya babi yang boleh bicara. Jika ada hambatan yang diidentifikasi dalam standup harian, hambatan tersebut tidak boleh diselesaikan dalam rapat kecuali benar-benar diperlukan. Sebaliknya, Scrum Master harus mencatatnya dan menandai item terkait sebagai hambatan. Setelah standup, anggota tim dapat membahas sebagai kelompok yang lebih kecil tindakan apa pun yang mungkin diperlukan dan mengalokasikannya dengan tepat untuk membuka hambatan tugas.

Di akhir sprint, tinjauan sprint dilakukan. Seperti halnya perencanaan sprint, hal ini terjadi dalam dua bagian. Bagian pertama adalah demo, yang merupakan kesempatan bagi tim untuk menunjukkan pekerjaan yang telah diselesaikan dan peningkatan yang dapat dikirim kepada pemangku kepentingan mana pun, dan bagian kedua adalah retrospeksi sprint. Retrospeksi sprint adalah kesempatan bagi tim untuk melihat kembali bagaimana sprint berjalan, dan untuk mengidentifikasi cara apa pun yang dapat dilakukan untuk meningkatkan proses mereka agar sprint berikutnya lebih berhasil. Retrospeksi harus seterbuka dan sejujur mungkin, dan hanya dihadiri oleh anggota tim pemangku kepentingan lain jarang diundang. Bagian Peningkatan Berkelanjutan di bab ini membahas retrospektif secara lebih rinci.

Dalam sprint, biasanya ada bagan burn-up (atau burn-down), yang melacak kemajuan tim dalam sprint, dan untuk menunjukkan apakah semua item backlog kemungkinan akan diselesaikan atau tidak. Backlog secara keseluruhan dapat dilacak dengan cara ini, di seluruh sprint, untuk melacak kemajuan menuju tujuan yang lebih besar, atau untuk membantu membuat estimasi. Salah satu contoh bagan burn-up ditunjukkan pada Gambar 2-2. Bagan tersebut menunjukkan jumlah total poin cerita yang dikomit dalam sprint (Anda dapat melihat peningkatan cakupan di tengah sprint), serta berapa banyak yang telah diselesaikan. Ini dapat digunakan untuk memprediksi apakah item dalam sprint akan selesai tepat waktu atau tidak.



Gambar 2.2 Contoh Bagan Burn-Up Untuk Sprint

Tersedia banyak literatur tentang bentuk manajemen proyek ini, dan banyaknya gaya pelacakan dan kemungkinan perluasannya (terutama jika Anda mempertimbangkan untuk melibatkan beberapa tim yang saling terhubung dalam mengerjakannya), meskipun ringkasan ini seharusnya cukup bagi pengembang untuk memiliki pemahaman dasar tentang konsep tersebut.

Kanban

Alternatif Scrum dalam tim tangkas adalah Kanban. Kanban tumbuh dari gerakan lean manufacturing, dan menggambarkan analogi antara gerakan tersebut, yang merevolusi manufaktur mobil, dan rekayasa perangkat lunak. Kanban sering digunakan sebagai bagian dari serangkaian proses yang dikenal sebagai pengembangan perangkat lunak "lean". Pada akhir tahun 1940-an, Toyota mulai menganalisis rantai pasokan dan produksinya dan membandingkannya dengan cara supermarket mengelola inventaris mereka untuk menyimpan cukup setiap barang dalam stok yang dijual, tetapi untuk meminimalkan biaya penyimpanan dan waktu penyimpanan saat makanan dapat membusuk.

Pada tahun 1990-an, ini telah berkembang menjadi gerakan yang lebih umum yang disebut "lean manufacturing," yang didasarkan pada gagasan bahwa memperlancar aliran kerja dapat membuat produksi lebih efisien. Jika stasiun pada jalur produksi memproduksi komponen terlalu cepat, hal itu menyebabkan pemborosan karena kemungkinan kerusakan atau memperkenalkan persyaratan penyimpanan tambahan; tetapi jika satu stasiun terlalu lambat untuk tahap berikutnya dalam prosesnya, hal ini menyebabkan kemacetan. Menetapkan batasan minimum dan maksimum pada ukuran antrean antar stasiun dalam jalur

produksi memungkinkan Anda mengidentifikasi di mana masalah mungkin muncul dan memaksimalkan efisiensi pabrik Anda.

Meskipun pada tingkat permukaan konsep-konsep ini tampak cukup terputus, prinsip-prinsip yang mendasarinya bekerja dengan sangat baik ketika diterapkan pada perangkat lunak. Alih-alih jalur produksi fisik, kami memiliki ide-ide, yang berkembang melalui sistem pada tingkat definisi yang meningkat hingga mencapai tingkat definisi tertinggi, yang merupakan ekspresi dalam kode. Meskipun ada perbedaan yang jelas dalam detail langkah-langkah tersebut, mengelola cara kerja mengalir di antara langkah-langkah tersebut sangat mirip.

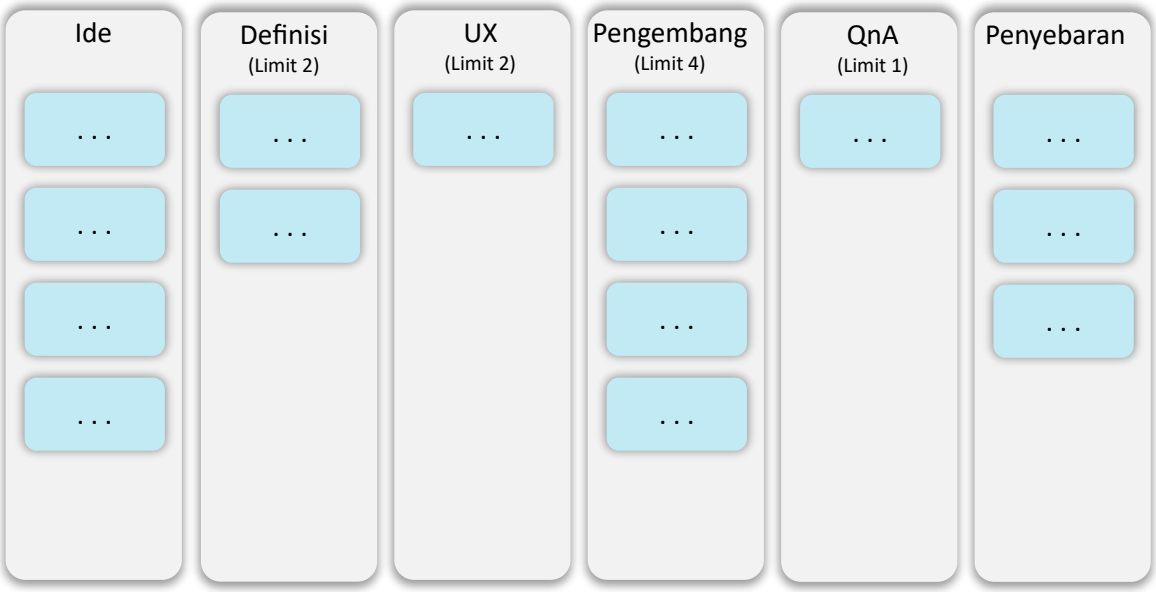
Kanban lebih merupakan seperangkat prinsip umum daripada praktik Scrum yang didefinisikan secara lebih kaku. Tidak ada kualifikasi Certified Kanban Master yang harus dibayar. Prinsip-prinsip Kanban cukup mudah dipahami (meskipun ada banyak variasi berbeda di dalamnya):

- Jadikan semua pekerjaan terlihat.
- Batasi pekerjaan yang sedang berlangsung.
- Berlatih perbaikan diri secara terus-menerus.

Karena Kanban lebih fleksibel, tampaknya lebih sulit untuk diadopsi. Namun, dengan memulai dengan prinsip-prinsip ini, dan memeriksa beberapa skenario umum di mana Kanban digunakan, Anda seharusnya dapat mengidentifikasi versi yang sesuai untuk organisasi Anda.
Sebuah tim di dalam organisasi digital modern seharusnya diberdayakan untuk mengatur pekerjaan mereka sesuai keinginan mereka, sesuai dengan apa yang terbaik bagi tim mereka. Meskipun terkadang sulit untuk meyakinkan orang yang skeptis, jika tim Anda tidak diberdayakan untuk mengelola pekerjaan sesuai keinginan Anda, maka kendala tersebut dapat membatasi hasil kerja Anda.

Bagi tim yang baru mengenal cara kerja agile, Scrum dapat menjadi langkah awal yang menenangkan, dengan aturan dan prosedur yang mirip dengan metode waterfall atau proses pengendalian perubahan. Jika Scrum tidak berhasil untuk Anda, itu belum tentu karena Anda "melakukan Scrum dengan salah"; mungkin Scrum tidak cocok untuk tim Anda. Singkirkan batasan yang menahan Anda dan jangan tambahkan nilai, dan ubah prosesnya hingga berhasil untuk Anda. Tim yang menggunakan Kanban sejak awal dapat terlihat mirip dengan tim yang menggunakan Scrum. Standup harian sering kali ditampilkan sebagai acara rutin, dan masih ada papan tugas, yang akan terlihat familier bagi mereka yang berasal dari latar belakang Scrum.

Gambar 2.3 memberikan contoh tata letak, tetapi setelah diperiksa lebih dekat, Anda akan melihat ada perbedaan antara papan Kanban dan papan Scrum.



Gambar 2.3 Papan Kanban Dengan Item Di Setiap Kolom

Satu perbedaan penting adalah Kanban tidak bekerja dalam sprint. Sebaliknya, setelah status tugas berubah, Anda menariknya dari satu kolom ke kolom berikutnya. Selain itu, batasan "pekerjaan yang sedang berlangsung" diterapkan sebelum sesuatu ditarik ke tahap pengembangan. Tim mengerjakan satu item pada satu waktu hingga selesai, daripada mengerjakan beberapa tugas sekaligus. Ketika satu item selesai, pekerjaan dapat dipindahkan ke prioritas berikutnya. Hal ini mendorong alur yang teratur melalui tim, alih-alih fokus Scrum pada peningkatan yang berpotensi dapat dikirim setiap dua minggu.

Hal ini juga memungkinkan respons cepat terhadap perubahan alih-alih harus menunggu sprint baru untuk mengerjakan tugas baru, atau harus mengubah cakupan sprint di tengah jalan, item berprioritas tinggi dapat dipindahkan ke bagian atas backlog dan diambil seperti pekerjaan normal. Batasan pekerjaan yang sedang berlangsung dapat terasa seperti kendala bagi tim, tetapi batasan tersebut dirancang untuk menjaga efisiensi tim. Batasan pekerjaan yang sedang berlangsung adalah batasan jumlah item yang ada di kolom tertentu pada satu waktu. Misalnya, tim dengan tiga pengembang dapat memilih batas pekerjaan yang sedang berlangsung sebanyak tiga. Mengerjakan lebih dari satu tiket sekaligus tidak dianjurkan, karena peralihan konteks dapat menyebabkan produktivitas yang lebih rendah. n+1 juga merupakan batas pekerjaan yang sedang berlangsung yang umum (misalnya, empat item untuk tiga pengembang).

Hal ini memperhitungkan fakta bahwa terkadang pemblokir eksternal dapat terjadi, yang berarti tiket harus dijeda untuk sementara waktu hingga pemblokir eksternal tersebut teratasi dengan sendirinya. Idealnya, pemblokir ini harus diidentifikasi sebelum memulai pengembangan, karena semakin lama kode tidak digunakan, semakin besar kemungkinan persyaratan untuk kode tersebut akan berubah, serta menambah overhead jika cara kode tersebut terintegrasi ke dalam sistem berubah, tetapi kode tersebut tidak memberikan nilai pada saat itu, hanya overhead. Jika banyak penghambat diidentifikasi sekaligus (misalnya, jika tiket merupakan pekerjaan yang jauh lebih banyak daripada yang diasumsikan sebelumnya),

dan batas pekerjaan yang sedang berlangsung tercapai, maka tim dapat mengerumuni suatu masalah, sehingga banyak orang bekerja untuk menyelesaikannya.

Terkadang hal ini tidak memungkinkan, dalam hal ini batas pekerjaan yang sedang berlangsung dapat ditembus, tetapi hanya dengan dukungan dari seluruh tim. Ada perbedaan lain antara Scrum dan Kanban. Orang yang menggunakan Kanban sering kali merasa berdaya untuk mengonfigurasi kolom pada papan mereka sesuai keinginan mereka, dan tim didorong untuk mulai melacak pekerjaan di seluruh siklus, bukan hanya tugas pengembangan. Banyak papan dimulai dengan kolom pertama "ide" dan kolom terakhir "siap untuk diterapkan" atau "dalam produksi." Jika tiket ada di papan, maka ada pekerjaan yang harus dilakukan. Sekadar "siap untuk diterapkan" tidaklah cukup, karena masih ada langkah untuk menerapkan pekerjaan sebelum manfaatnya dapat direalisasikan.

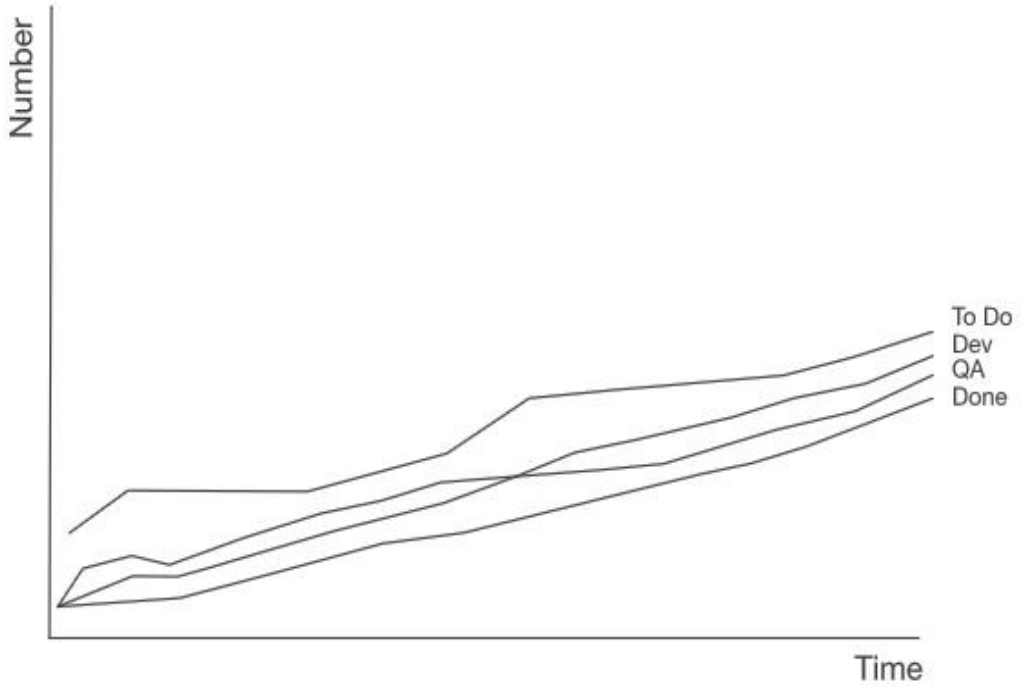
Bagi tim yang menjalankan pengujian A/B atau menggunakan pengembangan berbasis hipotesis, item yang "dalam produksi" tidak akan lengkap hingga informasi yang berguna telah diidentifikasi atau hingga eksperimen selesai berjalan. Kolom-kolom ini sering kali memiliki kriteria masuk/keluar, yang berperilaku seperti "definisi siap" dan "definisi selesai" dalam Scrum, tetapi alih-alih menentukan apakah cerita diterima atau tidak dalam sprint, kriteria ini menentukan transisi antarkolom. Kriteria ini berfungsi sebagai daftar periksa untuk memastikan cerita, item, atau pekerjaan benar-benar siap, terutama antara kolom yang ditangani oleh orang yang berbeda dalam satu tim.

Meskipun Kanban tidak menggunakan sprint, merupakan hal yang umum bagi tim (terutama tim yang telah beralih ke Kanban dari Scrum) untuk mengadakan rapat rutin yang terjadi pada frekuensi yang sama seperti dalam sprint. Misalnya, mengumpulkan tim untuk membuat rencana tidak harus menghasilkan hasil formal Scrum, tetapi dapat menjadi cara yang berguna untuk memperkirakan pekerjaan dan memastikan seluruh tim mengetahui setiap bagian pekerjaan yang akan datang, dan memungkinkan mereka untuk memberikan kontribusi dalam fase perencanaan atau desain tersebut (misalnya, menunjukkan ketergantungan teknis). Demikian pula, retrospektif sering kali berjalan dengan kecepatan yang teratur, mengikuti format yang mirip dengan Scrum, sebagai cara untuk memenuhi prinsip Kanban tentang peningkatan diri yang berkelanjutan.

Tim lain tidak merasa perlu melakukan ini (terutama tim kecil tempat orang bekerja secara lintas fungsi). Misalnya, item pekerjaan dapat didiskusikan secara ad-hoc saat muncul, meniadakan perlunya sesi perencanaan formal, atau retrospektif dapat dijalankan saat diperlukan. Beberapa tim memiliki "area retrospektif" di papan, tempat setiap anggota tim dapat menambahkan catatan tempel atau kartu indeks dengan masalah di atasnya, dan ini didiskusikan dan diselesaikan setelah stand-up. Fleksibilitas Kanban memungkinkan Anda menemukan apa yang cocok untuk Anda.

Teknik yang dikenal sebagai Scrumban juga ada, yang memiliki banyak varian. Satu varian Scrumban menggabungkan iterasi sprint yang dibatasi waktu dalam Scrum dengan aliran Kanban. Dalam metode ini, Kanban digunakan untuk mengelola semua aktivitas pra-sprint dan pasca-sprint misalnya, pengumpulan persyaratan, pekerjaan UX, atau analisis pasca-penerapan KPI tetapi tim pengembangan inti bekerja sementara itu dengan mengambil

sejumlah tiket dan mengerjakannya dalam sprint. Ini menyeimbangkan stabilitas pekerjaan yang akan datang untuk tim pengembangan sekaligus memberi lebih banyak struktur pada aktivitas seputar pekerjaan pengembang. Bentuk Scrumban ini sering kali diimplementasikan sebagai pendahulu penghapusan sprint dan Kanban penuh.



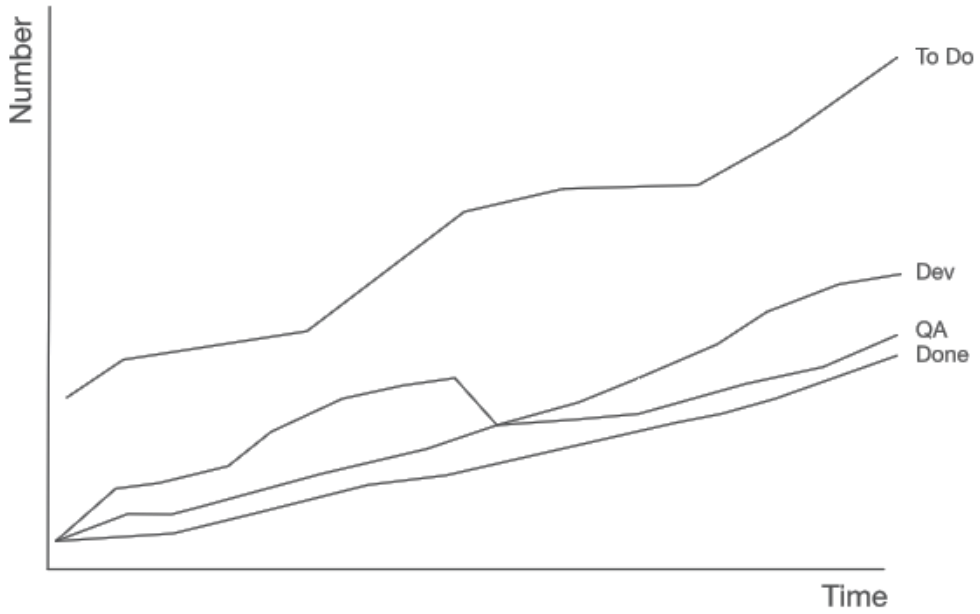
Gambar 2.4. Diagram Aliran Kontinu Kanban Yang Menunjukkan Aliran Yang Sebagian Besar Lancer

Pilihan alternatif adalah di mana kotak waktu tidak digunakan, tetapi kumpulan item dikerjakan secara sprint sebagai satu kelompok. Banyak organisasi juga telah mengembangkan varian mereka sendiri, sebagai cara untuk memadukan praktik Scrum yang ada dengan manfaat Kanban, atau untuk beralih ke Kanban penuh. "Scrumban" adalah istilah yang sangat longgar. Sementara dalam Scrum Anda menggunakan bagan burn-up/burn-down untuk melacak kemajuan pekerjaan dalam sprint menuju suatu tujuan, Kanban juga menawarkan alat yang dapat digunakan untuk memeriksa tim. Alat yang paling sering digunakan adalah diagram alir kontinu (CFD).

Dalam CFD, Anda membuat bagan garis yang menunjukkan berapa banyak item dalam setiap kolom dalam suatu sistem pada titik waktu tertentu. Dalam sistem yang ideal, garis-garis tersebut harus tetap datar, dengan pengecualian "selesai," yang harus bertambah. Ketika Kanban digunakan terhadap backlog proyek yang tetap, garis "yang harus dilakukan" akan berkurang, meskipun untuk banyak produk berkelanjutan (di mana Kanban bekerja paling baik), pekerjaan harus datang pada tingkat yang hampir sama dengan penyelesaiannya. Gambar 2.4 menunjukkan CFD umum yang sebagian besar berjalan lancar melalui sistem.

Contoh ini menunjukkan sedikit hambatan di pertengahan jalan, di mana terdapat tonjolan dalam QA, tetapi hal ini diatasi dengan memfokuskan kembali upaya pengembangan

ke QA (sehingga garis pengembangan menjadi nol dan menyentuh garis QA). Jika salah satu garis tersebut tidak datar, seperti yang terjadi pada Gambar 2.4, ada hambatan dalam sistem Anda yang perlu diatasi untuk menghindari hilangnya aliran kontinu dan orang-orang yang kewalahan. Gambar 2.5 menunjukkan tim yang diminta untuk melakukan terlalu banyak pekerjaan; pekerjaan bertambah lebih cepat daripada yang selesai, dan ada hambatan dalam pengembangan.



Gambar 2.5 Diagram Alir Berkelanjutan Yang Menunjukkan Aliran Yang Tidak Merata Melalui Pengembangan Dan Tumpukan Pekerjaan Yang Terus Bertambah

Peningkatan Berkelanjutan

Cara tim Anda merencanakan dan melacak pekerjaan tidak boleh statis. Cara tersebut harus terus berkembang sebagai respons terhadap pertumbuhan dan perubahan tim Anda. Dalam Scrum, retrospektif akhir sprint (atau "retro") diperkenalkan sebagai cara bagi tim untuk merefleksikan pekerjaan mereka dan bagaimana mereka melakukan peningkatan. Metodologi lain, seperti Kanban, tidak secara eksplisit menyerukan rapat seperti ini, tetapi merupakan hal yang umum bagi tim untuk terlibat dalam retrospektif secara berkala.

Ada banyak format untuk menjalankan retrospektif, tetapi yang paling umum adalah melihat kembali untuk mengidentifikasi apa yang salah, dan kemudian menghasilkan tindakan untuk mengatasi kekurangan tersebut. Ada banyak buku dan blog yang ditulis tentang subjek retrospektif, dan ini hanyalah salah satu cara untuk menjalankannya. Terlepas dari bagaimana Anda menjalankan retrospektif, ada beberapa prinsip yang perlu diperhatikan. Salah satunya dikenal sebagai "arahan utama" retrospektif: apa pun yang kami temukan, kami memahami dan benar-benar percaya bahwa setiap orang telah melakukan pekerjaan terbaik yang mereka bisa, mengingat apa yang mereka ketahui saat itu, keterampilan dan kemampuan mereka, sumber daya yang tersedia, dan situasi yang ada.

Bagian penting lainnya adalah bahwa tim harus merasa bahwa retrospektif adalah

tempat yang aman bagi mereka untuk terlibat dalam diskusi yang konstruktif dan menyarankan perbaikan. Substansi diskusi juga harus tersedia bagi para pemangku kepentingan yang tidak hadir, sehingga setiap perbaikan potensial dapat dibagikan, dan umpan balik yang lebih luas diakui. Perlu diingat bahwa kedua komponen terakhir ini dapat saling bertentangan, dalam hal ini prinsip-prinsip dapat dikesampingkan sementara masalah tersebut ditangani. Misalnya, jika menambahkan transparansi berarti orang tidak lagi merasa dapat berbicara dengan bebas, itu adalah masalah yang harus ditangani secara langsung dengan pembinaan sehingga tabir dapat diangkat perlahan-lahan, daripada mengorbankan kemampuan tim untuk berbicara secara terbuka.

Praktik penting lainnya adalah menjaga agar retrospektif tetap fokus pada hal-hal yang berada dalam kendali tim. Jika sprint berjalan buruk karena faktor eksternal (misalnya, pemutusan hubungan kerja di seluruh perusahaan, atau DDoS di situs web), meskipun mungkin melegakan untuk terlibat dalam sesi pengaduan, hal itu tidak terlalu produktif untuk dilakukan. Salah satu format yang mungkin untuk retrospektif adalah memulai dengan merefleksikan tindakan yang direncanakan dari rapat terakhir, melihat apakah ada yang telah diselesaikan, dan menentukan tindakan mana yang harus dilanjutkan, jika ada.

Kemudian sering kali berguna untuk membuat pengamatan sederhana tentang bagaimana anggota tim berpikir sprint terakhir berjalan, dan melihat apakah ada konsensus dan apakah umpan baliknya positif atau tidak. Ini dapat menandai masalah untuk segera dibicarakan; jika tidak, maka ada baiknya untuk merefleksikan kembali faktor-faktor individual dari sprint sebelumnya yang memengaruhi apa yang berjalan dengan baik dan apa yang berjalan buruk. Setelah peristiwa-peristiwa spesifik ini diidentifikasi, Anda dapat mengelompokkannya di sekitar tema dan membahas apa yang menyebabkannya terjadi, sebelum melanjutkan dan mengidentifikasi cara untuk memastikannya terus berlanjut (jika itu hal yang baik), atau menghindarinya terjadi lagi (jika itu hal yang buruk).

Sepanjang keseluruhan proses, penting untuk memastikan Anda merenungkan hal-hal baik yang layak dirayakan dan dilanjutkan, serta hal-hal yang perlu ditingkatkan. Retromat1 adalah alat yang dapat menghasilkan aktivitas dalam kerangka kerja ini, meskipun Anda harus memastikan untuk mengadaptasinya untuk situasi khusus tim Anda. Retrospeksi adalah bagian penting dari pekerjaan tim Anda, tetapi penting bahwa itu bukan satu-satunya waktu Anda merenungkan dan mempraktikkan peningkatan diri. Beberapa tim memiliki area di papan tugas mereka untuk catatan tempel berdasarkan pengamatan saat itu terjadi, dan jika beberapa peristiwa besar terjadi, tidak apa-apa untuk mengatasinya saat itu juga daripada menundanya ke retrospeksi.

Apa pun yang Anda putuskan untuk melakukan refleksi diri dan peningkatan berkelanjutan pada proses, Anda harus memastikannya berhasil untuk tim Anda. Retrospektif adalah salah satu metode perbaikan diri, tetapi bukan satu-satunya. Mengatasi utang teknis adalah cara lain, dan menjadi perhatian utama bagi tim mana pun. Sering kali tergoda untuk menempatkan utang teknis pada daftar tugas Anda sebagai hal yang harus diselesaikan, tetapi sering kali tidak pernah, karena sulit untuk menjelaskan nilai bisnisnya kepada pemangku kepentingan Anda, dan hal itu membuatnya sulit untuk diprioritaskan. Ada dua cara untuk

mengatasinya. Yang pertama adalah memastikan Anda mengungkapkan item teknis apa pun dalam bentuk yang sama seperti cerita pengguna lainnya.

Misalnya, jika pengujian Anda membutuhkan waktu lama untuk dijalankan, dan Anda ingin bereksperimen dengan paralelisasi, mungkin tergoda untuk menambahkan item daftar tugas yang mengatakan, "aktifkan pengujian paralel." Sebaliknya, Anda dapat mengatakan: Sebagai pengembang, Saya ingin pengujian saya selesai dalam waktu kurang dari 60 detik, Sehingga saya dapat melakukan penerapan lebih cepat. Hal ini dinyatakan dengan jelas dalam bahasa yang dapat dipahami oleh pemangku kepentingan lain, dan mudah-mudahan dapat diprioritaskan oleh pemilik produk Anda seperti pekerjaan lainnya. Pendekatan lain untuk mengatasi utang teknis adalah dengan menerapkan aturan Pramuka: selalu tinggalkan tempat perkemahan lebih bersih daripada saat Anda menemukannya.

Kali ini, tempat perkemahan kita adalah basis kode kita. Setiap kali Anda mengerjakan fitur baru, jika Anda menemukan utang teknis, atasi segera daripada menundanya nanti. Atau, jika Anda tahu ada utang teknis yang perlu diselesaikan, dan ada fitur mendatang yang akan diuntungkan dengan penyelesaian utang tersebut, maka masukkan saja ke dalam perkiraan fitur tersebut, karena keduanya saling terkait. Beberapa tim juga memiliki "waktu untuk menggunakan alat." Jika ada alat umum yang digunakan oleh beberapa tim, tetapi organisasi Anda tidak cukup besar untuk tim platform khusus, maka biasanya satu pengembang per minggu akan ditugaskan untuk tugas pemeliharaan umum, seperti memutakhirkan Jenkins, atau perbaikan kecil lainnya.

Jika ada tim yang berbeda dalam satu organisasi, ini bisa menjadi kerja lintas tim, sehingga semua tim diuntungkan. Sisi sebaliknya adalah bahwa hal itu akan menyulitkan tim individu untuk menentukan prioritas, dalam hal ini setiap tim menyumbangkan satu orang untuk jangka waktu tertentu guna melakukan perbaikan yang dapat dimanfaatkan semua orang.

2.5 PRIORITAS & ESTIMASI

Terlepas dari cara Anda mengelola pekerjaan, ada satu pertanyaan yang akan Anda tanyakan setiap kali menyelesaikan sesuatu. "Apa yang harus saya kerjakan selanjutnya?" Terkadang tidak perlu banyak berpikir. Jika sistem mati dan server bermasalah, mungkin Anda harus memperbaiki bug kritis yang memicu masalah. Di lain waktu, tekanan berkurang dan tidak ada yang benar-benar kritis. Yang lebih buruk lagi adalah saat ada sejumlah tenggat waktu yang ketat, dan banyak pekerjaan yang harus diselesaikan sebagai persiapan.

Backlog pada dasarnya adalah daftar tugas yang besar dan berurutan. Konsep backlog dimulai di Scrum, tetapi secara sederhana, backlog diurutkan sedemikian rupa sehingga hal terpenting berada di atas, dan Anda secara otomatis memilih hal terpenting tersebut. Anda kemudian dapat terus mengerjakannya hingga selesai, lalu memulai yang berikutnya, atau jika ada hal yang lebih penting, Anda berhenti dan mengerjakan hal terpenting yang baru. Skenario terakhir dapat menyebabkan banyak pekerjaan setengah jadi menumpuk, yang dapat meningkatkan pemborosan, tetapi skenario pertama dapat berarti pekerjaan penting tertunda hingga terlambat. Meminimalkan ukuran setiap cerita atau item pada daftar pekerjaan Anda

mengurangi kemungkinan Anda terganggu di tengah pekerjaan atau hal baru yang lebih penting tertunda terlalu lama.

Ada banyak cara untuk memprioritaskan backlog Anda, tetapi sering kali dalam sebuah tim, lebih baik jika ada satu orang yang bertanggung jawab untuk itu, karena sering kali akan ada banyak ide berbeda tentang apa yang menjadi prioritas tertinggi seperti jumlah orang dalam tim. Cara paling sederhana untuk mengatasi hal ini adalah dengan bertanya kepada para pemangku kepentingan tentang tindakan yang menurut mereka paling penting, tetapi dengan sekelompok pemangku kepentingan yang beragam, Anda akan sering kali mendapatkan pemahaman yang sangat berbeda tentang prioritas dari masing-masing pemangku kepentingan. Namun, ini dapat berguna untuk mengidentifikasi item mana yang berdampak pada banyak orang.

Pilihan lain adalah dengan mempertimbangkan biaya penerapan ide tertentu terhadap dampak yang akan ditimbulkan oleh pelaksanaannya. Menolak dan memangkas backlog juga dapat diterima. Jika sebuah ide bernilai rendah dan berbiaya tinggi, maka ide tersebut mungkin tidak akan pernah layak untuk dikerjakan. Sebaliknya, jika ide tersebut telah lama berada di backlog, maka ada baiknya untuk menanyakan apakah ide tersebut masih valid; mungkin kebutuhan awalnya telah hilang atau berubah sedemikian rupa sehingga tidak lagi berguna.

Dalam hal menentukan biaya sebuah ide, ini sering dilakukan dengan menggunakan proses yang disebut estimasi. Estimasi sering kali menjadi salah satu bagian tersulit dalam pelacakan pekerjaan, karena estimasi jarang akurat, meskipun orang-orang menginginkannya. Jarang bagi tim pengembangan untuk mengerjakan sesuatu yang persis sama dengan yang telah dikerjakan sebelumnya, dan jika memang demikian, cenderung menjadi tugas kecil untuk dilakukan, jadi estimasi akan selalu menyertakan unsur ketidakpastian. Metode awal estimasi berkisar pada prediksi waktu aktual yang dibutuhkan untuk suatu tugas.

Ini menghasilkan angka konkret yang dapat dikomunikasikan kepada para pemangku kepentingan, tetapi tingkat keyakinan seputar angka-angka tersebut sering kali tidak jelas, dan tenggat waktu sering terlewat, yang menyebabkan kurangnya kepercayaan pada tim pengembangan. Konsep story point yang umum digunakan dengan Scrum. Story point adalah alat yang berguna untuk memisahkan tanggal pasti dari estimasi, tetapi memiliki sisi negatif karena mempersulit pengomunikasikan makna sebenarnya kepada para pemangku kepentingan. Story point adalah angka tanpa satuan yang seharusnya menunjukkan ukuran suatu cerita, relatif terhadap cerita lain dalam daftar yang sama. Tidak ada pedoman mutlak mengenai arti angka-angka tersebut, dan bahkan dalam satu organisasi, angka-angka tersebut tidak dapat dibandingkan secara langsung antartim.

Setiap item pada backlog diberi nilai poin cerita, dan kemudian Anda dapat menjumlahkannya untuk menentukan seberapa besar backlog tersebut. Kecepatan adalah jumlah poin cerita yang diselesaikan dalam sprint rata-rata (sering kali dihitung dengan mengambil rata-rata dari total poin cerita yang diselesaikan dalam tiga atau empat sprint terakhir), dan dengan kecepatan, Anda dapat memperkirakan berapa banyak sprint yang diperlukan untuk menyelesaikan backlog atau mencapai item tertentu di dalamnya. Nilai yang mungkin untuk poin cerita biasanya tidak linear, untuk mengenali bahwa tugas yang lebih

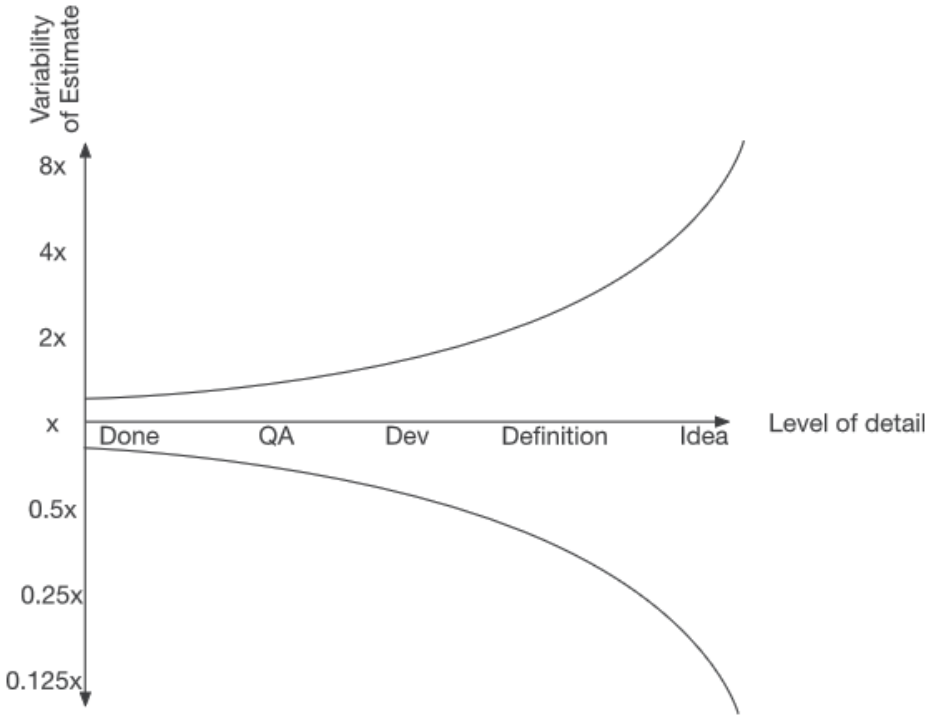
besar sering kali lebih sulit diperkirakan daripada yang lebih kecil. Sebaliknya, deret Fibonacci, atau variannya, digunakan.

Sebuah cerita dapat berupa 1, 2, 3, 5, 8, 13, atau 21 poin (terkadang lebih besar, tetapi sering kali ketika sebuah cerita terlalu besar, ada baiknya untuk memecahnya menjadi cerita yang lebih kecil). Aturan praktis yang baik adalah membayangkan hal paling sederhana yang dapat diselesaikan oleh sebuah tim untuk aplikasi web, ini mungkin sesuatu seperti desain ulang kecil sebuah tombol lalu tetapkan itu sebagai "1." Anda kemudian dapat memperkirakan kompleksitas dan ukuran relatif suatu item dibandingkan dengan itu. Dan di sinilah menjadi jelas bahwa poin cerita tidak dapat dibandingkan secara langsung antar tim. Sebuah tim dengan proyek yang relatif baru yang memiliki alat pengembangan yang baik dan rangkaian pengujian otomatis yang efektif mungkin dapat menyelesaikan "1" jauh lebih mudah daripada tim dengan basis kode lama dengan sedikit pengujian.

Bisa juga bahwa sebuah tim hanya memiliki ide yang berbeda tentang hal-hal paling sederhana yang harus diselesaikan. Poin cerita juga dapat digunakan dalam metodologi selain Scrum. Misalnya, dalam Kanban, alih-alih menggunakan sprint sebagai kerangka waktu untuk menentukan kecepatan, Anda dapat memilih periode waktu tetap dan menghitungnya menggunakan periode tersebut. Saat Anda mulai menggunakan poin cerita dan kecepatan untuk memprediksi proyek yang lebih jauh, penting untuk menyadari bahwa cara kecepatan dihitung (biasanya sebagai rata-rata) dapat menyembunyikan kompleksitas yang mendasarinya. Hal ini terkadang dikenal sebagai kerucut ketidakpastian hal-hal yang dekat dapat diprediksi dengan tingkat keyakinan yang lebih tinggi daripada yang lebih jauh.

Gambar 2-6 menunjukkan contoh kerucut ketidakpastian ini. Kerucut ketidakpastian dapat digunakan untuk memetakan kemungkinan nilai sebenarnya dari sebuah cerita dalam poin berada di antara dua ekstrem, tergantung pada seberapa jauh cerita tersebut dari penyelesaian. Poin cerita dapat menjadi alat internal yang efektif, tetapi sering kali berbahaya untuk dikomunikasikan di luar tim. Pemangku kepentingan yang tidak sepenuhnya memahami poin cerita dapat salah menafsirkannya, terutama jika ada tim lain yang menggunakannya. Salah satu kesalahan umum adalah manajemen melihatnya dan kemudian menetapkan target kinerja kepada tim untuk meningkatkan kecepatan mereka.

Velocity adalah ukuran yang digunakan untuk membantu perencanaan, bukan ukuran kecepatan atau kinerja tim. Cara termudah untuk meningkatkan velocity adalah dengan mulai memperkirakan angka yang lebih tinggi, yang dapat baik-baik saja selama angka-angka tersebut tetap relatif satu sama lain, tetapi karena cerita lama tidak tunduk pada inflasi poin ini, itu berarti sprint historis tidak dapat lagi digunakan untuk menentukan velocity.



Gambar 2.6. Diagram Kerucut Ketidakpastian, Menunjukkan Bahwa Semakin Jauh Suatu Hal Dari Yang Seharusnya Dilakukan, Semakin Besar Ketidakpastian Dalam Nilai Estimasi.

Teknik yang lebih berguna adalah memberikan tanggal kepada pemangku kepentingan. Membagi velocity dengan jumlah poin cerita yang ada di depan item tertentu pada backlog dapat memberikan perkiraan kapan sesuatu akan selesai, tetapi ini tidak memperhitungkan kerucut ketidakpastian. Sebaliknya, Anda dapat menerapkan teknik statistik, seperti mengambil deviasi standar dari perhitungan velocity untuk menentukan perkiraan "tinggi" dan "rendah" dari berapa banyak sprint yang diperlukan hingga item tertentu selesai, dan karenanya diterjemahkan ke dalam rentang tanggal. Ini masih dapat membuat pemangku kepentingan marah, karena mereka cenderung menyukai hal yang konkret.

Dalam skenario ini, perkiraan kasus terburuk dari keduanya harus digunakan, dan kemudian ada peluang yang wajar untuk menyelesaikannya lebih awal, yang akan membuat mereka senang. Ketika poin cerita digunakan, penting untuk memperkirakan ulang saat tugas bergerak melalui alur kerja. Ketika cerita dipahami dengan lebih baik, apa yang awalnya tampak sangat rumit dapat menjadi lebih sederhana, dan apa yang awalnya tampak sederhana dapat mengembangkan banyak kerumitan tersembunyi. Beberapa tim melangkah lebih jauh dan menggunakan "ukuran kaus" untuk cerita mereka yang paling tidak terdefinisi. Dengan ukuran kaus, cerita dinilai kecil, sedang, besar, atau ekstra besar, yang sesuai dengan poin cerita untuk tujuan estimasi.

Beberapa tim telah merangkul gerakan yang dikenal sebagai "tanpa estimasi," dan, lebih dari itu, "tanpa tumpukan pekerjaan." Tim tanpa estimasi tidak memberikan jaminan kepada pemangku kepentingan dan berjanji untuk bekerja secepat mungkin, hanya mengambil apa pun yang menjadi prioritas tertinggi pada waktu tertentu. Ini bekerja dengan baik ketika

tumpukan pekerjaan pendek. Tim tanpa tumpukan pekerjaan melangkah lebih jauh; tim tidak menerima pekerjaan kecuali mereka segera siap untuk mengatasinya (ini terkadang diterapkan sebagai batas WiP pada tumpukan pekerjaan). Pendekatan ini jarang terjadi, karena dapat menyebabkan tugas-tugas penting terlewatkan karena muncul pada waktu yang tidak tepat, dan dapat mempersulit manajemen pemangku kepentingan.

Untuk menentukan estimasi poin cerita ini, digunakan teknik ampuh yang dikenal sebagai Planning Poker. Dalam sesi Planning Poker, setiap orang diberikan kartu remi (atau aplikasi) dengan urutan poin cerita di dalamnya. Untuk setiap item, setiap individu dalam tim memilih kartu yang mencerminkan poin cerita yang akan mereka tetapkan untuk item tersebut, terkadang setelah beberapa diskusi tentang cakupan dan tujuan pasti dari sebuah cerita. Kartu-kartu tersebut kemudian diungkapkan secara bersamaan—ini dilakukan untuk mengurangi pengaruh di antara anggota tim, sehingga sampel yang benar-benar representatif dari seluruh tim diambil. Jika ada konsensus, maka nilai tersebut digunakan sebagai nilai poin cerita.

Jika tidak, tim dapat mendiskusikan mengapa mereka mungkin tidak setuju pada nilai apa pun (mungkin seseorang lupa beberapa tantangan teknis yang perlu diimplementasikan, atau mereka terlalu memperumit tugas di kepala mereka) dan kemudian putaran pemungutan suara lebih lanjut terjadi hingga konsensus tercapai. Sedangkan Scrum menggunakan proses poin cerita dengan kecepatan, Kanban menggunakan waktu sebagai ukuran kerja melalui sistem Anda. Oleh karena itu, akan sangat membantu jika Anda menggunakan Kanban dengan INVEST untuk mencoba dan membuat semua item dalam proses Anda memiliki "ukuran" yang sama dalam hal pekerjaan yang perlu dilakukan, dan setiap varians dirata-ratakan.

Untuk menggunakan estimasi dengan Kanban, Anda harus mengukur berapa lama waktu yang dibutuhkan suatu item antara saat dibuat, pekerjaan dimulai, dan kemudian diselesaikan. Selisih antara pembuatan dan penyelesaian disebut waktu tunggu, dan selisih antara pekerjaan yang dimulai pada item tersebut dan penyelesaiannya disebut waktu siklus. Waktu siklus adalah waktu minimum absolut yang diperlukan untuk mengirimkan tiket, jika pekerjaan segera dimulai, dan sering kali dapat diminimalkan oleh tim melalui perbaikan proses. Namun, waktu tunggu adalah hal yang diperhatikan oleh pemangku kepentingan eksternal, karena itulah waktu yang dibutuhkan untuk menyampaikan ide mereka, selain dari waktu yang sebenarnya dibutuhkan untuk menyelesaikan pekerjaan, tetapi juga termasuk waktu tunggu untuk pekerjaan yang akan diambil.

Jika suatu tim memiliki banyak pekerjaan yang tertunda, maka itu dapat berarti bahwa waktu tunggunya panjang, yang menunjukkan bahwa tim tersebut memiliki terlalu banyak pekerjaan yang harus dilakukan. Jika waktu tunggu meningkat, itu juga menunjukkan bahwa para pemangku kepentingan meminta terlalu banyak dari tim (diagram aliran kumulatif seharusnya menunjukkan hal ini juga).

2.6 MENGELOLA BUG

Bug (kadang-kadang dikenal sebagai cacat) merupakan hasil dari pekerjaan yang belum selesai, dan meskipun selalu ada bug yang belum ditemukan di sistem Anda, terlalu banyak bug yang ditemukan dapat melumpuhkan tim. Banyak tim melacak bug bersamaan dengan pekerjaan lain yang perlu diselesaikan, dan ini berfungsi dengan baik bagi mereka. Yang lain memiliki pelacak bug terpisah, yang dapat menyebabkan masalah ketika tim harus mencari di banyak tempat untuk menemukan bug. Seperti pekerjaan lain yang harus diselesaikan, bug perlu didefinisikan. Sering kali ini dilakukan oleh analis atau penguji, dengan format yang telah dicoba dan diuji:

a. Langkah-langkah untuk mereproduksi
b. Perilaku yang diharapkan
c. Perilaku actual

Ketika Anda sedang dalam krisis waktu, sangat menggoda untuk mengabaikan bug demi memenuhi tenggat waktu fitur, tetapi melakukan hal itu akan meningkatkan jumlah pekerjaan yang sedang berlangsung. Bug merupakan sesuatu yang terlewatkan (baik oleh pengembang, atau pada tahap awal proses definisi) dalam suatu pekerjaan, dan tidak adil untuk menyebut pekerjaan itu selesai jika masih ada bug.

Bug regresi bahkan lebih buruk, karena bug tersebut mengambil sesuatu dari pengguna yang selama ini diandalkan, dan alih-alih menghentikan kita menyelesaikan pekerjaan yang kita kira telah selesai, bug tersebut menambahkan pekerjaan tambahan kembali ke proyek yang sedang berlangsung.

Bagi pengembang web tumpukan penuh, banyak bug terjadi sebagai akibat dari ketidakcocokan browser atau bergantung pada perangkat. Ini adalah bug yang paling menyebalkan untuk diperbaiki, karena secara teknis bug tersebut tidak ada dalam kode Anda (jika Anda telah menulis kode sesuai standar). Karena menambahkan solusi untuk memperbaiki bug ini memerlukan biaya, sering kali lebih efektif untuk melakukan triase bug, tempat bug didiskusikan dan diprioritaskan.

Beberapa bug yang muncul sama sekali bukan bug. Alih-alih menjadi masalah yang tidak kami putuskan untuk diperbaiki, masalah tersebut adalah hal-hal yang telah diidentifikasi sebagai bug karena perbedaan pemahaman antara orang yang melakukan pengujian dan orang yang menerapkannya. Jenis bug ini juga dapat didiskusikan dalam sesi triase bug, di mana maksudnya dapat diklarifikasi.

Triase bug seharusnya bukan pertemuan formal, melainkan pertemuan santai bagi orang-orang dalam tim untuk membahas bug. Penjadwalan triase sering kali dapat menunda perbaikan (misalnya, harus menunggu seharian penuh untuk menerapkan perbaikan selama 30 menit), tetapi bagi sebagian tim, perlu untuk memesan waktu dengan pemangku kepentingan seperti pemilik produk untuk membuat keputusan tersebut. Bagi tim lain, triase bug dapat sesederhana menandai seseorang dalam komentar, atau mengirim IM.

Bug yang jelas terjadi ketika penguji mengidentifikasi sesuatu yang menurut mereka terlewatkan oleh pengembang, dan pengembang setuju bahwa mereka memang telah melewatkan sesuatu. Ketika semua orang setuju, tidak perlu melalui sesi triase formal untuk

ini; pengembang dapat memperbaikinya dan penguji dapat memverifikasinya dengan cukup cepat. Triase tidak boleh menjadi pengganti saluran komunikasi ad-hoc dalam tim.

Bahaya utama bug adalah ketika bug terbengkalai dalam waktu lama di suatu tempat dalam daftar tunggu, jadi hal ini harus dihindari. Ingatlah bahwa meminimalkan pekerjaan yang sedang berlangsung itu penting, dan membiarkan pekerjaan terbuka menyebabkan beban kognitif dan juga dapat memengaruhi kualitas produk kita. Teknik yang sangat efektif untuk mengelola bug adalah dengan segera memperbaikinya (sehingga bug selalu berada di urutan teratas daftar tunggu), atau memutuskan bahwa bug tidak layak untuk diperbaiki. Cara yang terakhir bisa jadi tidak nyaman bagi beberapa tim, tetapi jika bug tidak cukup penting untuk diperbaiki sekarang, lalu mengapa itu bisa berubah?

Mungkin Anda dapat membayangkan masa depan di mana tidak ada pekerjaan proyek berprioritas tinggi, yang dalam hal ini bug ini dapat ditemukan, tetapi jika bug dapat dibiarkan, maka itu sering kali karena dampaknya sangat kecil atau tidak signifikan, dan pada kenyataannya, pengembangan fitur baru tidak mungkin memiliki dampak yang lebih rendah dari itu. Lebih baik menghilangkan bug ini daripada membiarkannya tetap ada di daftar tunggu Anda. Jika bug cukup penting untuk diperbaiki, sebaiknya Anda segera memperbaikinya, daripada menundanya. Bug tersebut menyebabkan masalah pada situs web atau aplikasi Anda saat ini, dan jelas perlu diperbaiki.

Pengiriman Berkelanjutan

Seperti yang disinggung di atas, data kualitatif dapat menjadi metode yang sangat efektif untuk menemukan informasi tentang produk yang Anda buat. Namun, sangat sulit untuk mengumpulkan data kualitatif tanpa produk yang ada di dunia nyata. Pengiriman berkelanjutan (CD) adalah salah satu dari banyak cara untuk mendekati pengiriman ide hingga ke tahap produksi, dan disusun berdasarkan gagasan untuk meminimalkan waktu yang dibutuhkan untuk memasukkan proyek ke tahap produksi.

Pengiriman berkelanjutan bekerja seperti jalur produksi. Salah satu ujung jalur produksi adalah ide atau masalah, dan ujung lainnya adalah perangkat lunak yang sedang berjalan. Ini adalah metafora yang tidak sempurna, karena Anda sering kali ingin menggunakan perangkat lunak yang dirilis untuk menghasilkan wawasan tentang pengguna Anda dan kinerja fitur tersebut untuk menginformasikan fitur baru. Satu perbedaan utama dari pendekatan lain adalah bahwa perangkat lunak selalu bergerak maju melalui jalur produksi.

Alih-alih melakukan rollback, dengan pengiriman berkelanjutan Anda "melakukan rollback ke depan," jadi jika rilisnya buruk, versi dengan perbaikan atau dengan kode yang salah dikembalikan akan didorong ke depan, daripada versi lama dari kode yang sama. Pada intinya, pengiriman berkelanjutan adalah tentang meminimalkan waktu dan gesekan yang diperlukan untuk membuat perubahan pada aplikasi. Tim yang mempraktikkan pengiriman berkelanjutan harus secara formal mendefinisikan alur pengiriman produk dari ide ke pengiriman, dan mengartikulasikan bagaimana sesuatu bergerak dari satu fase ke fase berikutnya. Alur tersebut dapat terdiri dari beberapa fase, seperti:

- Penemuan, di mana penelitian dilakukan pada sebuah ide untuk mendapatkan pemahaman tingkat tinggi tentang ruang masalah dan solusi apa yang mungkin

diperlukan;

- Definisi, di mana kriteria penerimaan formal dan pekerjaan desain menghasilkan informasi yang diperlukan agar pekerjaan pengembangan dapat dilanjutkan;
- Pengembangan, di mana produk dibangun dan diuji secara fungsional;
- Penerimaan, di mana pengujian penerimaan tingkat tinggi dilakukan dan persetujuan pemangku kepentingan yang diperlukan diperoleh;
- Penerapan, di mana perubahan dimasukkan ke dalam produksi;
- Verifikasi, di mana setiap analitik diselidiki untuk memastikan bahwa perubahan tersebut memiliki dampak yang diinginkan.

Daftar periksa di setiap fase menentukan apa yang dibutuhkan agar perubahan dapat terus berlanjut. Terkadang fase tingkat tinggi ini dipecah lebih jauh lagi—misalnya, fase pengembangan mungkin memiliki fase arsitektur teknis, pembuatan, dan peninjauan kode tambahan. Alur seperti ini mungkin tampak mirip dengan proses waterfall, tetapi masing-masing item harus mengalir melalui alur ini, bukan kumpulan besar item yang berbeda-beda di mana definisi seluruh kumpulan dilakukan di awal.

Berbagai tahap alur ini dapat melibatkan banyak orang dengan keterampilan yang berbeda-beda, sehingga tugas yang berbeda dapat dikerjakan secara paralel di berbagai tahap, tetapi setiap tugas bersifat independen dan dikerjakan oleh orang yang berbeda. Dengan membiarkan tugas-tugas individual mengalir melalui alur ini, anggota tim didorong untuk terlibat dalam aktivitas yang sesuai lebih sering, berdasarkan gagasan bahwa melakukan sesuatu lebih sering berarti Anda menjadi lebih baik dalam melakukannya. Setiap tugas hafalan, seperti beberapa jenis pengujian regresi, atau kemampuan penyebaran, atau analisis data, kemudian didorong untuk diotomatisasi guna meningkatkan efisiensi alur. Menerapkan teknik yang dibahas sebelumnya dalam bab ini, seperti INVEST, juga dapat memberi Anda item dalam bentuk yang sesuai untuk dibawa melalui alur CD.

Menurut pengalaman saya, pengiriman berkelanjutan bekerja dengan baik untuk tim yang mengorganisasikan diri mereka sendiri menggunakan Kanban. Meskipun Scrum tidak selalu membatasi Anda untuk hanya merilis di akhir sprint, ini sering kali tampak sebagai tempat yang wajar untuk melakukannya. Dengan Kanban, dengan membatasi pekerjaan yang sedang berlangsung, dan mengerjakan satu item hingga selesai, lalu beralih ke item berikutnya, tim terbebas dari siklus sewenang-wenang yang diamanatkan oleh Scrum, dan menjadi aliran ide yang lebih teratur ke produksi.

Jika backlog Anda terdiri dari cerita pengguna yang memenuhi mnemonik INVEST, Anda kemungkinan akan melihat keberhasilan dengan pengiriman berkelanjutan. Dengan membuat setiap cerita pengguna kecil dan independen, Anda dapat menyebarkannya ke produksi saat selesai, setelah mengujinya untuk memastikan bahwa cerita tersebut memenuhi persyaratan. Membuat banyak perubahan kecil mengurangi risiko setiap perubahan individu dan meminimalkan waktu antara pekerjaan yang dilakukan dan kemampuan untuk memanfaatkan pekerjaan tersebut.

Sering kali mengejutkan untuk menyadari bahwa ada hubungan yang kuat antara cara Anda merencanakan dan mengerjakan pekerjaan Anda dan cara Anda mengelola kode sumber

Anda dalam repositori seperti Git. Ada dua pendekatan dominan untuk mengelola kode sumber di dunia rekayasa perangkat lunak. Salah satunya sering disebut sebagai percabangan fitur, dan yang lainnya sebagai integrasi berkelanjutan (CI), atau pengembangan berbasis trunk. Integrasi berkelanjutan sering kali disalahartikan dengan serangkaian alat yang mendukung praktik tersebut, tetapi alat-alat ini juga dapat membantu dalam pendekatan pengembangan berbasis cabang fitur, jadi penting untuk fokus pada prinsip-prinsip CI, bukan hanya penggunaan alat-alat CI.

Di dunia tanpa CI, setiap kali fitur baru dikembangkan, cabang baru dibuat dalam sistem kontrol sumber dan semua pekerjaan untuk fitur tersebut dilakukan di cabang tersebut, tanpa perlu khawatir tentang ketidakstabilan jalur utama atau pekerjaan yang setengah selesai diterapkan. Terkadang ada cabang untuk setiap pengembang. Setelah pekerjaan selesai, atau rilis jatuh tempo, yang bisa jadi beberapa hari atau bahkan minggu kemudian, cabang tersebut kemudian digabungkan ke jalur utama. Ada sisi negatif dari pendekatan ini, karena dapat berarti sejumlah besar perubahan terjadi sekaligus, dan jika ada banyak cabang fitur, maka cabang Anda mungkin telah menyimpang cukup jauh dari jalur utama sehingga penggabungan otomatis tidak dapat terjadi sampai Anda melakukan beberapa pengerjaan ulang.

Ini terkadang disebut "neraka penggabungan." Jika suatu fitur besar, mungkin ada manfaat tambahan selama pengembangan fitur tersebut bagi pengguna akhir yang tidak terungkap sampai setelah penggabungan, meningkatkan waktu antara melakukan pekerjaan dan nilai yang direalisasikan. CI dibuat sebagai metode alternatif untuk mengatasi sisi negatif ini. Setiap pengembang berkomitmen pada cabang utama secara berkala, biasanya setidaknya sekali sehari. Dalam aplikasi integrasi berkelanjutan murni, pengembang selalu melakukan komitmen langsung ke master, tetapi sudah menjadi hal yang umum untuk menggunakan varian tempat cabang fitur berumur pendek digunakan sebagai mekanisme untuk peninjauan kode. Ini memungkinkan peninjauan kode sebelum mendorong ke master, tetapi setiap cabang berumur sangat pendek (seringkali hanya beberapa jam) untuk tetap memungkinkan sebagian besar manfaat integrasi berkelanjutan.

Salah satu keuntungan dari percabangan fitur sebelum adopsi CI secara luas adalah bahwa jalur utama selalu stabil dan lengkap, karena fitur tidak digabungkan hingga selesai dan diuji. Alat integrasi berkelanjutan, seperti Cruise dan Jenkins, diciptakan untuk menjalankan pengujian otomatis pada setiap komitmen ke jalur utama, untuk menghadirkan tingkat keyakinan yang sama seputar stabilitas bagi tim yang menggunakan integrasi berkelanjutan. Namun, tim cabang fitur juga melihat nilai dari alat ini, dan masih mengaturnya untuk dijalankan setelah setiap penggabungan, atau bahkan pada setiap cabang secara terpisah. Berhati-hatilah agar tidak membingungkan konsep integrasi berkelanjutan dengan penggunaan alat integrasi berkelanjutan!

Tim yang melakukan integrasi berkelanjutan sering mengalami masalah saat mereka ingin melakukan komitmen kode yang stabil tetapi tidak selalu mewakili fitur yang lengkap. Merupakan hal yang umum bagi tim-tim ini untuk melakukan commit pada kode yang sudah "mati" (yaitu, hanya menggunakan pengujian yang menguji apakah kode tersebut berfungsi atau tidak, tetapi tidak diaktifkan dalam aplikasi yang sebenarnya), atau di balik "bendera

fitur".

Bendera fitur (kadang-kadang disebut sakelar fitur) adalah pengaturan konfigurasi aplikasi yang menunjukkan apakah fitur tertentu tersedia bagi pengguna atau tidak. Dalam aplikasi web, ini dapat mengaktifkan/menonaktifkan berbagai rute URL di backend, atau menyembunyikan widget UI yang berarti kode tidak akan pernah dapat dipicu. Ini memungkinkan Anda untuk mengembangkan di lingkungan praproduksi, tetapi tidak mengaktifkannya di lingkungan produksi hingga seluruh fitur selesai. Ini berarti bahwa kode diaktifkan secara bertahap, bukan sekaligus, yang dapat menjadi cara yang efektif untuk mengelola risiko, meskipun dengan biaya tambahan untuk mengelolanya.

Untuk refaktor yang lebih besar, misalnya, jika Anda mengganti satu fitur dengan fitur baru lainnya, merupakan hal yang umum untuk memiliki kedua versi kode tersebut, dan kemudian sakelar fitur yang beralih di antara keduanya. Melakukan hal-hal dengan cara ini memudahkan Anda untuk melakukan pengujian lain dengan kode Anda, seperti hanya mengaktifkannya untuk persentase tertentu dari basis pengguna Anda untuk mengujinya saat meluncurkannya, atau untuk melakukan pengujian A/B, di mana Anda memiliki dua versi yang aktif sekaligus dan melihat mana yang berkinerja lebih baik dengan melihat statistik. Hal ini dibahas dalam bab Pembelajaran Konstan. Konsep lain yang terkait dengan pengiriman berkelanjutan adalah penerapan berkelanjutan.

Penerapan berkelanjutan membawa pengiriman berkelanjutan ke tingkat berikutnya. Dalam penerapan berkelanjutan, setiap komitmen (yang lulus pengujian) sebenarnya diterapkan ke lingkungan produksi, sedangkan dalam pengiriman berkelanjutan, setiap komitmen mampu diterapkan ke lingkungan produksi, tetapi penerapan yang sebenarnya masih terjadi saat tim mendefinisikannya sebagai mungkin. Penerapan berkelanjutan memerlukan tingkat otomatisasi yang lengkap dari komitmen hingga produksi, karena tidak ada ruang untuk berhenti untuk fase QA manual. Tim yang mempraktikkan penerapan berkelanjutan sering kali masih melakukan QA manual, tetapi sering kali di lingkungan produksi itu sendiri, bukan sebelumnya di lingkungan QA. Penerapan berkelanjutan dibahas lebih mendalam dalam bab Penerapan.

Ringkasan

Merencanakan pekerjaan secara efektif sebelum mengerjakannya akan meningkatkan efektivitas Anda, karena hal itu memastikan Anda benar-benar memahami masalah yang ingin Anda selesaikan. Secara historis, perencanaan ini dilakukan secara formal dengan banyak definisi di awal, tetapi dalam banyak situasi hal itu telah digantikan oleh metodologi agile. Metodologi agile berfokus pada perencanaan hanya apa yang benar-benar dibutuhkan saat ini, sehingga Anda bebas merencanakan atau merencanakan ulang bagian lain dari proyek Anda dengan cara yang bereaksi terhadap perubahan.

Perencanaan dimulai dengan mendefinisikan backlog daftar prioritas tentang apa yang akan dibangun oleh tim dan item-item ini harus disusun sedemikian rupa sehingga tidak sepenuhnya teknis agar dapat diprioritaskan sesuai dengan tujuan organisasi. Item pekerjaan kemudian bergerak melalui alur kerja tempat detail tambahan dapat ditambahkan, mungkin oleh pemangku kepentingan dalam disiplin ilmu lain seperti UX, menambahkan elemen seperti

desain visual, hingga akhirnya disebarkan ke pengguna akhir Anda. Dua teknik agile utama adalah Scrum dan Kanban, yang mengambil pendekatan berbeda untuk melacak dan mengelola pekerjaan. Baik Scrum maupun Kanban mendorong tim untuk merefleksikan alur kerja mereka dan memperbaruinya secara berkala untuk memastikan bahwa alur kerja tersebut memenuhi kebutuhan unik tim tersebut.

Perbedaannya terletak pada cara mereka memungkinkan prediksi kapan pekerjaan mendatang dapat diselesaikan, dengan Scrum menggunakan story point untuk menentukan kecepatan, dan Kanban menggunakan waktu rata-rata penyelesaian fitur sebelumnya. Selain merencanakan pekerjaan baru, Anda juga harus mengelola bug yang terjadi selama pengembangan, mungkin saat fitur sedang dikembangkan, atau mungkin disembunyikan hingga nanti. Bug ini masih dapat diperlakukan sebagai item backlog, tetapi menangkap persyaratannya dengan cara yang berbeda.

Penyerahan berkelanjutan dan integrasi berkelanjutan adalah dua pendekatan yang berbeda tetapi terkait untuk mengelola pekerjaan rekayasa guna memungkinkan proses penyelesaian fitur. Penyerahan berkelanjutan menerapkan otomatisasi untuk memudahkan item backlog bertransisi melalui berbagai tahap dari ide hingga sistem yang direalisasikan dan diterapkan dan integrasi berkelanjutan adalah cara pengembang mengoordinasikan perubahan ke satu basis kode bersama.

BAB 3
PENGALAMAN PENGGUNA

3.1 PERAN UX DALAM PENGEMBANGAN PERANGKAT LUNAK

Salah satu karakteristik pengembangan perangkat lunak dalam organisasi digital modern adalah menempatkan kebutuhan pengguna sebagai inti tujuan tim, alih-alih hanya berfokus pada tujuan rekayasa yang lebih terpisah. Disiplin pengalaman pengguna (UX) bekerja sama dengan gaya pengembangan baru ini untuk membantu membangun perangkat lunak yang benar-benar memenuhi kebutuhan pengguna. Tidak ada definisi yang jelas tentang UX; disiplin ini relatif baru, dan berkembang pesat. UX telah berevolusi dari sejumlah bidang yang terpisah, tetapi terkait, menjadi satu yang menyatukan semuanya.

Ini menggabungkan disiplin yang lebih akademis tentang interaksi manusia-komputer (HCI) dengan peran "desainer" tradisional dalam desain visual, desain interaksi, dan desain produk, tetapi juga mencakup proses dan layanan perancangan, yang sebelumnya mungkin dilakukan oleh analis bisnis. Sebelum UX, banyak desainer digital berasal dari latar belakang cetak tradisional. Untuk banyak situs web pemasaran atau berbasis konten, ini dirancang dengan cara yang sama seperti buletin atau kampanye poster. Ini sering kali meninggalkan kesenjangan besar dalam keterampilan antara desainer dan pengembang; File Photoshop dilempar ke dinding dan diharapkan dapat diimplementasikan dengan sempurna.

Meningkatnya desain responsive membuat hal ini semakin menantang, karena reaksi awalnya adalah membuat desain yang pas di iPhone, tetapi gagal beradaptasi dengan ukuran layar yang berubah cepat yang berevolusi dari produsen lain. Hal ini memaksa desainer dan pengembang untuk bekerja sama secara erat satu sama lain, karena desain ini tidak dapat lagi ditentukan dengan sempurna di Photoshop, tetapi harus ditentukan dengan cara yang lebih abstrak yang sesuai dengan maksud desainer. Pengembang kemudian harus memahami maksud desain untuk mengekspresikannya dalam kode, daripada hanya membuat replika desain.

Untuk jenis aplikasi lain, terutama yang digerakkan oleh proses, banyak proyek digital awal mengambil proses yang digerakkan oleh kertas yang ada dan mengimplementasikannya secara elektronik. Secara tradisional, jenis formulir ini dirancang oleh analis bisnis, yang mengembangkan proses yang memenuhi kebutuhan bisnis, dan ketika tiba saatnya untuk mengimplementasikan formulir ini secara elektronik, analis bisnis sering kali akan menentukan interaksi ini dalam bentuk yang sama dengan sistem berbasis kertas. Terkadang seorang desainer akan mengambil desain tersebut dan menatanya agar terlihat menarik secara estetika, tetapi hal ini mengabaikan cakupan yang lebih luas untuk melihat bagaimana proses tersebut secara keseluruhan memenuhi kebutuhan pengguna, bukan hanya kebutuhan bisnis.

Studi tentang interaksi manusia-komputer muncul sebagai bidang dalam ilmu komputer akademis pada tahun 1970-an, tetapi seiring berkembangnya desain web dari dunia desain cetak, kedua bidang ini terpisah hingga menyebarnya proses yang dikenal sebagai desain yang berpusat pada pengguna. Pendekatan baru ini berkembang dari desain proses dan

interaksi tradisional dengan mengambil pendekatan ilmiah yang digunakan oleh para peneliti HCI. Hal ini dilakukan untuk menguji prototipe antarmuka pengguna pada pengguna nyata saat mereka masih dalam tahap draf awal, kemudian menggunakan pengamatan dari eksperimen ini untuk mengulang dan mengembangkan desain.

Menggabungkan desain proses ini dengan desain UI dan menerapkan pendekatan ilmiah untuk membawa pengguna ke pusat proses tersebut merupakan apa yang sekarang dikenal sebagai bidang UX. UX suatu produk lebih dari sekadar UI-nya; UX juga merupakan prinsip di baliknya dan cara kerjanya sebagai sistem secara keseluruhan. Ada banyak sub-bidang dalam UX, meskipun hanya organisasi terbesar yang cukup beruntung untuk dapat mempekerjakan spesialis untuk setiap bidang. Sebaliknya, sebagian besar praktisi UX melakukan sedikit dari semuanya. Seperti dalam rekayasa perangkat lunak, jabatan dan peran tidak universal. Beberapa orang dalam UX menggunakan "desainer" sebagai istilah umum, sementara yang lain menggunakannya untuk merujuk pada seseorang yang lebih berfokus pada desain visual, tetapi ada tiga jenis peran utama yang mungkin Anda temui.

Desainer UX mirip dengan desainer visual tradisional, yang dapat menghasilkan desain dan aset berkualitas tinggi untuk diterapkan oleh pengembang, dan terkadang memiliki beberapa keterampilan pengembangan front-end sendiri. Peneliti, atau penguji pengguna, biasanya berfokus pada perancangan dan menjalankan studi dengan calon pengguna untuk menguji kesesuaian desain tertentu dan pengalaman yang diusulkan. Terakhir, seorang arsitek UX (atau arsitek informasi) berpikir tentang bagaimana sistem tersebut terstruktur secara keseluruhan dan di mana informasi disajikan kepada pengguna berdasarkan kebutuhan mereka.

Sesuatu yang mungkin dihasilkan oleh seorang arsitek informasi untuk situs web tradisional adalah peta situs yang menunjukkan bagaimana pengguna dapat menavigasi melalui halaman, tetapi pada akhirnya tujuannya adalah agar pengguna menemukan informasi yang ingin mereka temukan di tempat yang mereka harapkan, dan memastikan bahwa informasi tersebut berada pada tingkat detail yang mereka harapkan tidak terlalu detail, tetapi juga tidak terlalu umum sehingga pengguna tidak memenuhi kebutuhan awal mereka. Ada tumpang tindih antara peran-peran ini. Situs web yang sarat konten mungkin memiliki peran yang menggabungkan peran arsitek dengan peran copywriter. Output di sini bukanlah desain visual, tetapi konten yang sesuai dengan situs web dan arsitektur yang menampung konten tersebut.

Konten ditulis sebagai respons langsung terhadap kebutuhan pengguna, dan proses penemuan dan desain UX yang sama masih berlaku untuk mengembangkan output ini. Semakin umum bagi desainer UX untuk juga memiliki keterampilan pengembangan front-end, sehingga mereka dapat mengimplementasikan desain mereka secara langsung dalam kode, serta keterampilan yang terkait dengan proses memahami dan menguji persyaratan pengguna. Sebaliknya, pengembang tumpukan penuh akan sering mempelajari keterampilan yang sebelumnya berada dalam domain spesialis UX, yang selanjutnya menyadari manfaat pengembangan tumpukan penuh. Pengalaman pengguna telah menjadi begitu lazim sehingga jika suatu organisasi tidak merangkulnya, mereka akan dengan cepat dikalahkan oleh pesaing

yang melakukannya.

Terkadang tim terkecil tidak memiliki orang yang didedikasikan untuk aktivitas UX ini, tetapi penting bagi seluruh tim untuk tetap fokus dalam memenuhi kebutuhan pelanggan yang menggunakan produk. Setiap antarmuka pengguna akan memiliki beberapa desain di dalamnya. Bahkan jika desain itu belum sepenuhnya dikembangkan oleh seorang desainer, desain itu akan muncul dari kode yang diimplementasikan. Sebagai seorang pengembang, memahami sepenuhnya UX dari apa yang Anda bangun dari sudut pandang mereka yang membangunnya akan membantu Anda membuat produk yang lebih baik, dan Anda mungkin akan bekerja dengan praktisi UX dalam upaya ini.

3.2 ARSITEKTUR INFORMASI

Jika tujuan akhir perancang UX adalah tampilan dan nuansa yang menarik secara estetika yang membuat setiap fitur mudah dan jelas digunakan, maka tujuan akhir seorang arsitek informasi adalah memastikan setiap komponen individual pada halaman berada di tempat yang diharapkan pengguna, dengan ambiguitas minimal. Namun, ini melampaui satu halaman, dan berlaku untuk struktur seluruh situs, memastikan bahwa pengguna dapat menavigasi situs atau aplikasi untuk menemukan informasi yang mereka inginkan. Pada tim yang memiliki silo antara UX dan pengembang, perancang UX mungkin sering menyerahkan spesifikasi visual untuk komponen, dan arsitek UX menyerahkan peta situs dan wireframe yang menunjukkan bagaimana halaman menampilkan komponennya.

Dalam dunia tim tumpukan penuh, tim-tim ini bekerja sama. Faktanya, estetika sebenarnya adalah perhatian yang paling tidak penting dari tim UX. Jika pengguna Anda tidak dapat menemukan informasi yang mereka cari, maka tidak masalah seberapa bagus tampilannya. Demikian pula, setelah mereka sampai di tempat yang tepat, informasi yang mereka cari atau fitur yang mereka coba gunakan harus memiliki "affordance," yang berarti fitur tersebut harus berperilaku dengan cara yang jelas dan tidak mengejutkan. Bersama-sama, konsep-konsep ini dianggap sebagai kegunaan, dan tujuan utama tim UX adalah untuk menghasilkan situs web yang dapat digunakan, lalu menerapkan branding dan estetika di atasnya sebagai "hal yang lebih penting”.

Contoh klasik dari desain versus estetika adalah Craigslist. Desain visual Craigslist sangat mendasar, tetapi situs tersebut menawarkan pengalaman pengguna yang kuat. Informasi terstruktur dengan cara yang logis, dan jelas bagaimana cara menggunakan setiap fitur situs tersebut. Affordance adalah konsep penting dalam HCI, dan karenanya dalam pengalaman pengguna. Hal ini berlaku untuk desain secara umum, bukan hanya desain digital. Affordance suatu objek dikatakan sebagai cara bentuknya menunjukkan bagaimana objek tersebut dapat digunakan misalnya, pegangan pada teko, atau pegangan di satu sisi pintu versus panel dorong di sisi lainnya.

Antarmuka pengguna komputer biasanya memiliki affordance nonfisik. Biasanya pola umum digunakan untuk menunjukkan cara berinteraksi dengan sesuatu (misalnya, tombol pada halaman web mungkin memiliki bantalan dan dikelilingi kotak, serta memiliki beberapa status melayang), atau untuk menyarankan affordance dengan menggunakan skeuomorfisme

meminjam desain dari objek dunia nyata. Konsep affordance yang dipersepsikan menempatkan affordance apa pun yang mungkin dimiliki objek ke dalam konteks. Misalnya, affordance kunci pintu toilet sering kali berwarna hijau untuk menunjukkan bahwa pintu tersedia dan merah jika tidak tersedia, tetapi affordance ini bukan affordance yang dipersepsikan bagi orang buta warna.

Demikian pula, dalam desain UX, alasan orang mungkin datang ke situs web atau menggunakan aplikasi web sering kali akan menempatkan mereka dalam kondisi mental tertentu atau memberi mereka harapan tentang cara kerja suatu hal. Desainer perlu mempertimbangkan affordance yang dipersepsikan ini. Affordance sendiri sering kali tidak cukup, terutama ketika pengguna melakukan tindakan yang sangat rumit. Dalam kasus ini, penting untuk memastikan pengguna mengetahui model logis yang mendasari operasi yang ingin dicapai dan memandu mereka melalui tindakan tersebut.

Desainer juga harus mencoba untuk mencegah dan membatasi tindakan apa pun yang mungkin diambil karena pemahaman yang salah tentang model yang mendasarinya. Namun, hal ini tidak boleh digunakan sebagai alasan untuk desain yang buruk. Pola "tur berpemandu" yang digunakan banyak aplikasi web saat ini sering kali dijalankan secara tidak tepat dan digunakan saat model yang mendasarinya terlalu rumit. Meskipun pengguna dapat menavigasi desain yang buruk (terutama saat tidak diberi pilihan untuk melakukannya, seperti dengan alat internal), hal itu tetap harus dihindari, dan memperbaikinya adalah hal yang etis untuk dilakukan karena akan mengurangi stres, meningkatkan efisiensi, dan mengurangi kesalahan bagi suatu organisasi.

Namun, jika diberi pilihan, banyak pengguna akan menghindari situs yang dirancang dengan buruk sama sekali. Sudah umum untuk melihat lowongan pekerjaan untuk "desainer konten," terutama di sektor jasa keuangan, dan sering kali ada beberapa tumpang tindih antara peran arsitek informasi dan desainer konten. Seorang desainer konten melangkah lebih jauh dari seorang copywriter (meskipun banyak desainer konten yang akhirnya membuat konten juga), yaitu membantu menentukan konten yang perlu disampaikan. Mereka dapat membantu menentukan struktur tingkat tinggi konten ini dalam konteks keseluruhan situs, serta struktur konten itu sendiri, dan hubungan di antara keduanya.

Satu area yang tumpang tindih antara arsitek informasi dan desainer konten adalah proses penamaan sesuatu. Seperti lelucon lama, penamaan sesuatu (bersama pembatalan cache dan kesalahan selisih satu) adalah salah satu dari dua tugas tersulit dalam ilmu komputer, tetapi penamaan yang konsisten dan jelas dapat mempermudah pengguna untuk menemukan apa yang mereka cari. Nama harus dipilih dari sudut pandang pengguna, bukan dari sudut pandang bisnis. Misalnya, operator kereta api mungkin mengumumkan bahwa mereka mengalami "kegagalan kendaraan karena masalah pantograf," tetapi ini dapat dianggap sebagai jargon yang tidak berarti bagi banyak anggota masyarakat. Sebaliknya, mengumumkan bahwa "kereta dibatalkan karena kesalahan pada kabel udara" lebih jelas dan tetap menyampaikan informasi yang sama.

Salah satu tanggung jawab penting seorang arsitek informasi adalah memastikan struktur situs mencerminkan apa yang sebenarnya diinginkan pengguna. Dalam organisasi

yang tidak memiliki arsitek informasi, biasanya bagian-bagian yang berbeda dari suatu situs, atau situs-situs yang berbeda untuk organisasi yang sama, dijalankan oleh tim internal yang berbeda, dan akibatnya struktur situs web mencerminkan hierarki manajemen organisasi, yang berarti bahwa beberapa informasi mungkin tidak berada di tempat yang diharapkan pengguna.

Ini adalah contoh dari "Hukum Conway," yang menyatakan bahwa apa pun yang dirancang oleh suatu organisasi ditakdirkan untuk mencerminkan struktur korporat internal organisasi tersebut. Ketika suatu organisasi menganggap serius arsitektur informasi, arsitek informasinya harus meruntuhkan dinding-dinding yang terisolasi ini. Seorang arsitek informasi mungkin termasuk dalam satu tim tetapi harus bekerja dengan tim lain (atau bekerja dengan arsitek informasi lain di seluruh organisasi) untuk menghasilkan sesuatu yang bekerja secara holistik dari perspektif pengguna di luar organisasi.

Satu catatan terakhir tentang arsitek informasi: ada jabatan lain yang kedengarannya mirip yaitu arsitek data. Arsitek data bertanggung jawab untuk mendefinisikan struktur dan hubungan, tetapi seperti namanya, mereka bekerja dengan data mentah yang mendasari suatu organisasi. Arsitek data biasanya juga berfokus pada kebutuhan bisnis, bukan kebutuhan pengguna, dan sering kali membahas berbagai masalah seperti menghilangkan duplikasi, keamanan, dan keakuratan data. Pada akhirnya, data ini digunakan untuk menyampaikan informasi di situs web, sehingga arsitektur informasi berdampak pada arsitektur data, tetapi keterampilannya sangat berbeda.

Mendapatkan Pengalaman Pengguna yang Tepat

Pada proyek yang melibatkan tim pengalaman pengguna dan tim pengembangan yang terpisah, desainer mungkin akan membangun desain berdasarkan pengalaman mereka sendiri dan persyaratan fitur tertentu, lalu menyerahkannya kepada pengembang untuk diimplementasikan. Sering kali, estetika menjadi salah satu kekuatan pendorong desain. Mengikuti prinsip UX, saat desainer membangun desain, atau arsitek membuat struktur, mereka memikirkan asumsi yang telah mereka buat. Jika asumsi yang berbeda dapat dibuat secara wajar dan tidak jelas mana yang benar, maka desain yang berbeda dibangun dengan masing-masing berfokus pada asumsi yang berbeda.

Desain ini kemudian diuji dengan pengguna nyata untuk memeriksa apakah asumsi tersebut benar, dan memastikan tidak ada masalah tersembunyi dalam desain dan bahwa desain tersebut berperilaku seperti yang diharapkan pengguna. Salah satu indikator terbesar bahwa tim bekerja dengan cara modern adalah seberapa sering mereka melakukan pengujian pengguna, dan bagaimana mereka melakukannya. Tim yang terus-menerus melakukan pengujian pengguna mungkin baru dalam hal ini dan mencoba menemukan keseimbangan antara cara kerja baru ini dan yang lama, sedangkan tim yang tidak pernah melakukan pengujian pengguna mungkin terjebak dalam pola pikir tradisional. Pengujian pengguna pada intinya tampak sederhana.

Anda cukup mengundang pengguna untuk berinteraksi dengan situs web Anda dan memberi mereka tugas untuk diselesaikan, sambil memperhatikan mereka saat melakukannya. Penting untuk tidak hanya memperhatikan apa yang mereka lakukan dan

katakan, tetapi juga bahasa tubuh mereka, karena ini dapat menunjukkan rasa frustrasi atau masalah lain yang mungkin tidak terlihat dari apa yang mereka katakan atau lakukan. Namun, merancang pengujian Anda dengan benar (tugas yang Anda minta dilakukan pengguna) dan memilih pengguna yang tepat untuk diuji bisa jadi cukup rumit, di sinilah peran peneliti yang berdedikasi berperan.

Sering kali memiliki latar belakang psikologi, para peneliti ini dapat menggunakan pelatihan dan pengalaman formal mereka untuk membantu mengembangkan tugas yang berfokus pada sifat sistem yang diuji, menghindari pertanyaan yang mengarahkan, dan mengajukan pertanyaan yang tepat untuk mendapatkan wawasan yang baik. Dalam hal jumlah pengguna yang ingin Anda uji, satu aliran pemikiran yang dipromosikan oleh Jakob Nielsen adalah menjalankan banyak pengujian, di mana setiap pengujian hanya berfokus pada satu hal, dan memiliki lima peserta. Aliran pemikiran alternatif adalah menggunakan kelompok yang lebih besar (mungkin selusin peserta), tetapi melihat banyak bagian dari suatu aplikasi sekaligus. Hal terakhir yang perlu dipertimbangkan adalah apa yang akan diuji.

Tentu saja, Anda tidak benar-benar menguji pengguna, melainkan aplikasi Anda (atau sebagian darinya) untuk melihat apakah aplikasi tersebut dapat digunakan (dalam hal manfaat yang dirasakan dari komponen UI dan kemampuan menemukan tujuan dalam arsitektur informasi). Dalam beberapa kasus, mungkin dapat diterima untuk menguji produk yang sudah jadi mungkin jika itu adalah situs lama yang sedang Anda coba tingkatkan tetapi sering kali menunggu hingga akhir pengembangan sebelum pengujian pengguna dapat menjadi mahal jika ternyata ada masalah signifikan yang perlu diselesaikan. Akibatnya, pengujian pengguna paling sering dilakukan pada beberapa tingkat prototipe.

Di ujung lain spektrum pengujian dengan produk akhir adalah pengujian dengan tiruan kertas. Meskipun beberapa detail akan hilang, jenis pembuatan prototipe ini sering kali cepat dibuat dan berguna untuk memeriksa konsep tingkat tinggi apa pun yang mendasari seluruh situs, dan khususnya berguna jika Anda membuat asumsi tentang model mental yang mungkin dimiliki pengguna tentang tindakan yang akan mereka lakukan. Di antara kedua ekstrem ini, kolaborasi yang baik antara pengembang dan desainer dapat menjadi efektif. Prototipe dapat dibangun dalam teknologi web aktual yang sesuai untuk pengujian pengguna, sering kali tanpa tingkat ketahanan penuh yang akan digunakan dalam kode produksi.

Bergantung pada apa yang sedang diuji, masalah seperti responsivitas, kompatibilitas lintas-peramban, aksesibilitas, dan penanganan kesalahan dapat diabaikan. Merupakan hal yang umum bagi komponen ujung depan prototipe untuk dikembangkan secara terpisah, menggunakan respons kalengan dan tanpa terhubung ke ujung belakang mana pun. Prototipe ini kemudian dibuang, bahkan jika desainnya berhasil, karena perbaikan masalah aksesibilitas, responsivitas, dll., biasanya lebih sulit daripada membangunnya kembali dari awal dan mempertimbangkan kualitas-kualitas ini sejak awal. Kecepatan, bukan kualitas, lebih penting di sini.

Anda mungkin tergoda untuk bertanya langsung kepada pengguna apa yang mereka inginkan, tetapi ini sering kali tidak berjalan sebaik yang Anda harapkan. Tidak mudah bagi pengguna untuk memvisualisasikan secara realistis cara mereka menginginkan sesuatu

bekerja, dan mereka sering kali tidak menghasilkan ide-ide terbaik dalam praktiknya. Melakukan pengujian pengguna yang lebih formal ini memungkinkan Anda untuk memeriksa perasaan "naluri" pengguna. Namun, beberapa UXer akan melakukan lokakarya "desain bersama", di mana para pemangku kepentingan (termasuk pengguna) akan berkumpul dan merancang sesuatu secara kolaboratif, daripada hanya disajikan dengan desain yang sudah jadi pada sesi pengujian pengguna, dan ini dapat menjadi cara yang berharga untuk bekerja untuk menghasilkan desain juga.

Namun, ada sisi buruk dari menjalankan sesi dan eksperimen pengujian pengguna formal, yaitu biayanya. Jika fitur baru di situs tidak terlalu inovatif dalam hal persyaratan atau interaktivitas, atau sebagian besar mirip dengan fitur lain di tempat lain, mungkin lebih efektif untuk sekadar menerapkan pola yang diketahui pada fitur tersebut alih-alih menguji semuanya, lalu memeriksanya dengan menggunakan analitik setelah fitur tersebut sepenuhnya dibangun, jika risiko kesalahan desainnya kecil. Kompromi ini bisa berbahaya, karena pola web umum yang tampaknya efektif ternyata tidak mencapai tujuannya dengan baik setelah diteliti lebih lanjut. Ini termasuk pola seperti "menu hamburger", ikon tanpa label di sebelahnya, dan korsel, yang telah diadopsi secara luas.

Penting untuk diingat bahwa pengujian pengguna tidak perlu terlalu formal. Ada cara berbiaya rendah untuk menjalankan pengujian pengguna tanpa harus menyewa spesialis atau menyewa fasilitas dengan kaca satu arah atau perangkat perekam. Meskipun cara ini tidak memberi Anda wawasan yang terperinci atau akurat seperti sesi yang lebih formal, cara ini dapat membantu Anda mengidentifikasi masalah yang paling jelas dengan aplikasi Anda. Sesi berbiaya rendah ini disebut sebagai pengujian pengguna "gerilya". Sesi pengujian gerilya yang umum dapat dilakukan di kedai kopi, di mana (dengan izin dari pemiliknya) Anda dapat menawarkan untuk membelikan seseorang kopi dengan imbalan 10 menit waktu mereka, di mana Anda mengamati mereka menggunakan situs Anda atau melakukan beberapa tugas umum.

Jika Anda mengembangkan alat internal, maka ini menjadi lebih mudah; Anda hanya perlu bertanya kepada calon pengguna apakah Anda dapat mengamati mereka menggunakan desain Anda di tempat, dan banyak yang akan dengan senang hati melakukannya. Seperti yang akan kita bahas dalam bab Etika, pengujian pengguna adalah bentuk eksperimen pada manusia. Dalam hal ini, kita harus berperilaku etis. Untungnya, ada banyak hal yang dapat kita pelajari dari disiplin ilmu lain (terutama ilmu sosial), karena mereka telah mengasah pemahaman yang baik tentang cara melakukan eksperimen semacam ini secara etis. Aturan umum yang harus diperhatikan adalah persetujuan yang diinformasikan: pengguna harus menyadari apa yang diminta untuk mereka lakukan dan setuju untuk ikut serta, dengan kemampuan untuk menarik diri dari persetujuan kapan saja.

Pengujian pengguna tidak boleh bertujuan untuk menyesatkan atau menipu pengguna, dan harus memperlakukan mereka dengan hormat. Hasilnya juga harus dianonimkan dengan tepat. Meskipun pengujian pengguna telah menjadi contoh utama dari proses UX, itu bukan satu-satunya teknik yang dapat Anda gunakan untuk memeriksa kegunaan situs Anda. Aplikasi web atau situs dengan analitik yang baik seharusnya memungkinkan Anda untuk

menginterpretasikan data untuk menjawab pertanyaan tertentu tentang bagaimana kinerja situs Anda dalam produksi. Di tengah-tengah antara pengujian pengguna secara langsung dan hanya berfokus pada analitik setelah kejadian adalah jenis pengujian yang dikenal sebagai pengujian A/B.

Pengujian A/B melibatkan penyajian sejumlah varian (seringkali dua) kepada pengguna bersama dengan kasus "kontrol" dari situs saat ini tanpa perubahan. Sebagian pengguna di situs Anda dipilih secara acak untuk mengambil bagian dalam uji coba, lalu diberikan satu dari setiap varian. Analisis kemudian digunakan untuk menentukan berapa banyak pengguna yang menyelesaikan aktivitas menggunakan setiap varian dan kontrol, dan angka ini menentukan mana yang paling berhasil (atau tidak; tidak diasumsikan bahwa kedua varian akan berkinerja lebih baik daripada kontrol) dengan menerapkan uji statistik pada data yang dikumpulkan. Terlepas dari apa yang mungkin coba dijual oleh banyak perusahaan, pengujian A/B tidak selalu merupakan alat yang tepat untuk digunakan.

Pendekatan naif terhadap pengujian A/B mungkin dengan langsung membandingkan jumlah orang yang mampu menyelesaikan tindakan yang Anda uji. Namun, teori statistik memberi tahu kita bahwa angka-angka ini sebenarnya sedikit kabur di baliknya. Signifikansi statistik dari hasil perlu ditentukan untuk mengevaluasi apakah angka-angka ini relevan. Konsep ini berasal dari gagasan "pengambilan sampel". Dalam hal ini, sampel adalah bagian dari basis pengguna Anda yang melakukan percobaan. Tidak mungkin untuk memilih bagian yang benar-benar representatif, jadi tingkat ketidakjelasan diterapkan pada hasil untuk menentukan apakah hasil yang berbeda merupakan hasil dari bagian yang berbeda yang berperilaku dengan cara yang sedikit berbeda, atau apakah sebenarnya ada perubahan nyata di sana.

Ketika analisis menemukan bahwa dua angka secara sah mewakili perbedaan dalam kinerja dua versi, ini dikatakan "signifikan secara statistik". Dengan sedikit peserta, ketidakjelasan angka lebih intens, jadi lebih sulit untuk menentukan apakah satu angka akurat atau tidak, jadi sejumlah besar peserta diperlukan untuk menentukan apakah perbedaan benar-benar signifikan. Oleh karena itu, pengujian A/B tidak sesuai pada area situs web dengan lalu lintas rendah, meskipun dengan menjalankan pengujian untuk jangka waktu yang lama (mungkin selama beberapa minggu) Anda dapat meningkatkan jumlah peserta dan karenanya mendapatkan hasil yang lebih akurat.

Pada bagian situs web yang sangat sibuk, mungkin tampak bahwa Anda hanya perlu menjalankan pengujian untuk jangka waktu yang singkat, tetapi ini juga bisa bias. Misalnya, menjalankan pengujian selama satu jam pada Jumat sore hanya akan memberi Anda data untuk jangka waktu tersebut, dan pengguna mungkin berperilaku berbeda di pagi hari dibandingkan malam hari. Menjalankan pengujian Anda setidaknya selama satu hari penuh, dan sering kali seminggu, direkomendasikan.

Pengujian A/B juga menjadi rumit jika Anda ingin menjalankan beberapa pengujian sekaligus. Pengujian ini dapat saling mengganggu dan membatalkan hasil satu sama lain. Meskipun dimungkinkan untuk menjalankan beberapa pengujian di situs yang sama, pengujian tersebut harus menguji aktivitas berbeda yang mungkin ingin dilakukan pengguna.

Demikian pula, saat Anda melakukan pengujian A/B sesuatu, perubahan yang dibuat harus cukup kecil. Misalnya, jika Anda bereksperimen dengan perubahan pada proses pembayaran, jika Anda mendesain ulang alur formulir serta elemen visual, akan sulit untuk menentukan apakah perubahan pengguna disebabkan oleh alur yang ditingkatkan atau elemen visual. Dengan pengujian A/B, Anda juga perlu menerapkan versi lengkap dari semua varian yang diuji; prototipe saja tidak cukup baik.

Etika pengujian A/B masih diperdebatkan. Tidak mungkin untuk mendapatkan persetujuan berdasarkan informasi dalam beberapa kasus, sementara dalam kasus lain, meminta pengguna untuk memilih mode "beta" mungkin sudah cukup. Sebagian besar diskusi seputar pengujian A/B difokuskan pada apakah hasil pengujian dapat menyebabkan kerugian atau tidak. Contoh yang terkenal adalah pengujian A/B yang dilakukan Facebook yang melibatkan menampilkan posting dengan sentimen positif dan negatif di umpan pengguna, lalu melihat apakah hal itu memengaruhi sentimen posting yang dibuat pengguna tersebut. Banyak yang mengecam hal ini karena dianggap membawa kerugian karena membuat orang semakin sedih.

Jenis pengujian A/B umum lainnya (terutama di perusahaan rintisan) adalah dengan bereksperimen dengan struktur harga yang berbeda. Jika pengguna mendaftar dengan harga yang lebih tinggi, yang tetap mereka bayarkan, meskipun mereka secara acak tidak diizinkan mengetahui ada harga yang lebih rendah, apakah ini merugikan mereka? Beberapa organisasi mengatasi hal ini dengan selalu memberikan harga yang lebih rendah di akhir proses terlepas dari alur mana yang dipilih pengguna, tetapi jawaban untuk pertanyaan ini tidak jelas. Kelemahan penggunaan analitik dan metode "kuantitatif" semacam ini adalah sering kali kurang mendalam. Kita dapat dengan mudah melewatkan "alasan" orang berperilaku seperti itu, yang dapat ditemukan oleh metode "kualitatif" seperti pengujian pengguna.

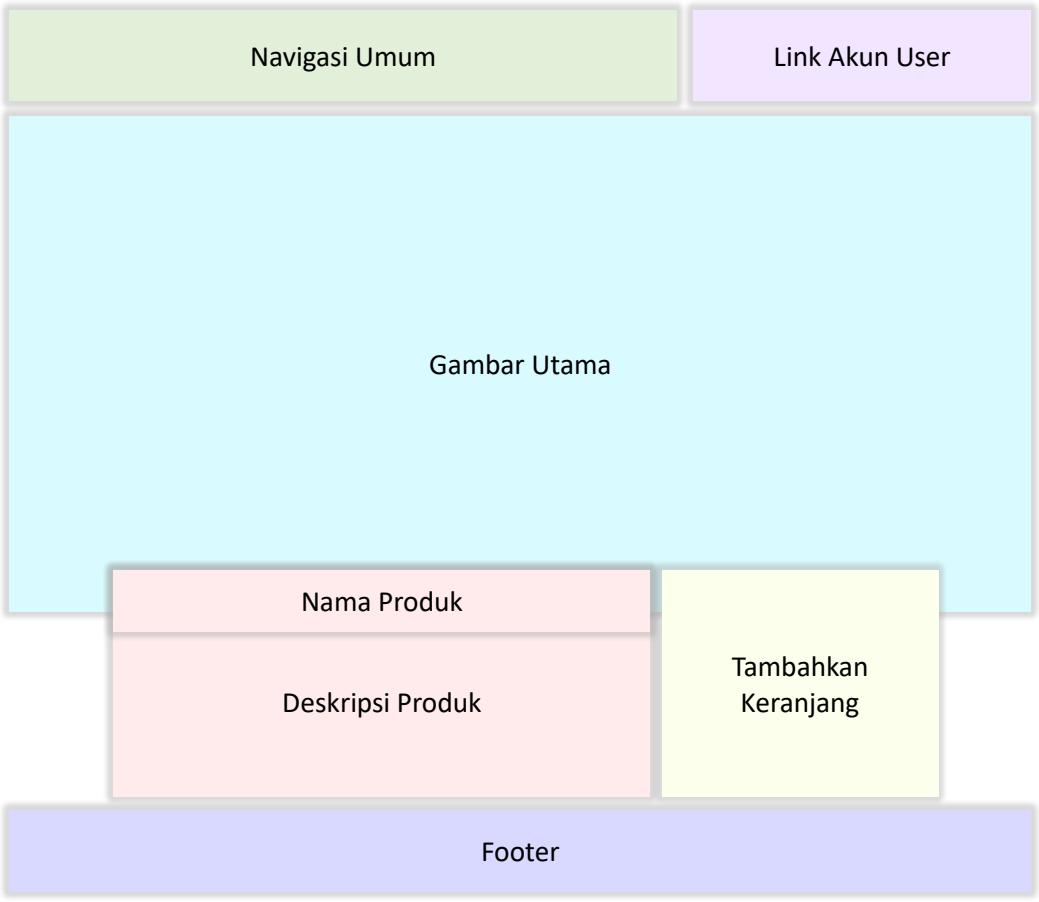
Misalnya, Google menggunakan pengujian A/B di awal keberadaannya untuk membandingkan dua pengalaman pengguna, dan kemudian menemukan bahwa salah satunya berkinerja buruk karena memiliki waktu muat yang lebih lama karena alasan teknis, bukan karena alasan kegunaan. Sebaliknya, metode "kualitatif" sering kali mahal untuk dijalankan dalam skala besar, dan mungkin sulit untuk menghasilkan kumpulan data yang cukup besar dan dapat dibandingkan secara langsung. Tim yang paling hebat akan menggunakan keduanya, dengan pengujian pengguna yang memberikan kedalaman dan analitik yang lebih memvalidasi asumsi tersebut dalam skala besar, atau menyarankan di mana pekerjaan lebih lanjut perlu dilakukan.

3.3 OPTIMALISASI PENGALAMAN PENGGUNA DENGAN POLA DAN KERANGKA KERJA

Karena pola dan kerangka kerja berguna bagi pengembang saat mereka mengimplementasikan kode, pola berguna saat mendesain pengalaman pengguna. Bahkan saat mendesain pengalaman yang lebih kompleks atau baru yang sedang diuji pengguna, memiliki serangkaian pola untuk diambil dapat mempercepat proses ini. "Prinsip kejutan paling sedikit" berlaku dalam pengalaman pengguna; sebuah fitur harus bekerja seperti yang diharapkan pengguna, dan konsistensi di seluruh aplikasi Anda, dan Web secara keseluruhan,

adalah cara yang efektif untuk membantu mencapainya.

Ada persyaratan non-fungsional yang juga masuk ke dalam pengalaman pengguna. Hal-hal seperti penggunaan gambar dan warna, dan suara teks, dapat menciptakan suasana bagi pengguna. Bagi organisasi yang juga memiliki kehadiran fisik, seperti pengecer fisik, penting untuk membuat situs web terasa seperti bagian dari rantai toko fisik yang sama, karena hal ini dapat memperkuat pesan pemasaran yang disampaikan dan menetapkan ekspektasi untuk pengalaman berdasarkan atmosfer tersebut.

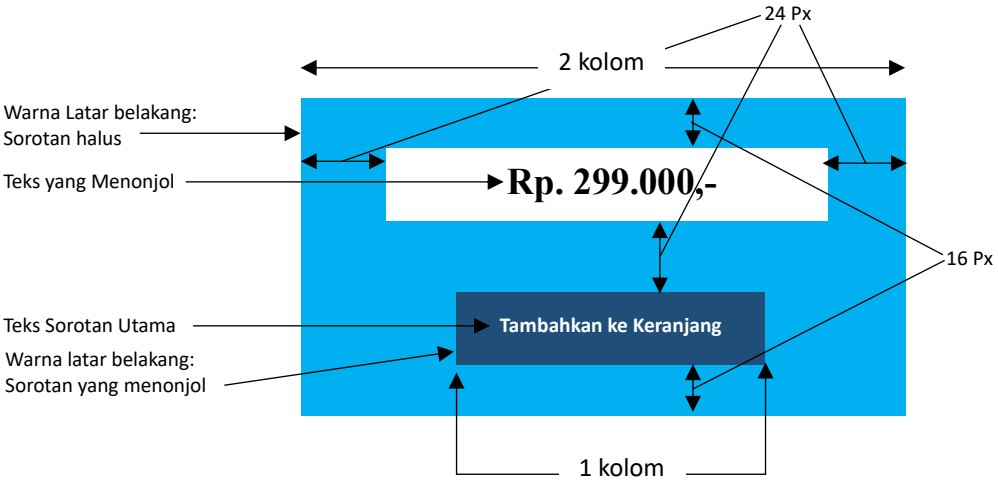


Gambar 3.1 Kerangka Kerja Untuk Halaman Arahan Produk

Buku merek sering kali menjadi dasar bagi apa yang disampaikan oleh tim UX. Buku merek tidak sering berubah dan menyediakan serangkaian dasar untuk menerapkan desain: warna, tipografi, logo, dan aturan tentang bagaimana merek harus diproyeksikan. Misalnya, perusahaan jasa keuangan mungkin mencoba memproyeksikan citra yang dapat diandalkan dan dapat dipercaya dalam mengelola uang. Menulis pesan kesalahan dalam format yang terlalu ramah, atau mengilustrasikan fitur dengan gambar kucing kartun, mungkin tidak sesuai dengan tujuan ini, dan aturan semacam ini harus dinyatakan dalam buku merek. Primitif sederhana ini, seperti aturan warna dan tipografi, sangat efektif dalam membangun konsistensi. Terutama di situs web besar atau kehadiran digital yang memiliki berbagai layanan mikro untuk berbagai bagian frontend Anda, ketidakkonsistenan apa pun di antara primitif ini dapat mengungkap jahitan yang mendasari aplikasi Anda, yang dapat mengganggu bagi

pengguna yang menganggap Anda sebagai satu situs terlepas dari struktur aplikasi yang mendasarinya.

Saat mengimplementasikan fitur baru, atau seluruh halaman atau layar baru di situs web, Anda dapat menggunakan berbagai tingkat fidelitas untuk memberikannya kepada pengembang. Tingkat terendah mungkin sekadar rangka kerja, seperti pada Gambar 3.1, berupa sketsa longgar yang memperlihatkan bagaimana halaman disusun dan arsitektur informasi tingkat tinggi dari halaman tersebut. Di ujung spektrum yang lain mungkin terdapat spesifikasi desain yang sangat terperinci dan diberi anotasi yang merepresentasikan setiap elemen pada halaman dalam piksel, seperti pada Gambar 3.2.



Gambar 3.2 Spesifikasi Desain Beranotasi

Namun seperti yang disebutkan sebelumnya, halaman baru sering kali menggunakan pola yang sudah ada sebelumnya, dan jika halaman menggunakan kembali pola, sering kali tidak perlu menentukan ulang detail ini. Terkadang pola ini hanya menggunakan kembali hal-hal mendasar, seperti merujuk pada warna yang dirujuk dalam merek (misalnya, alih-alih mengekspresikan kode heksadesimal dalam desain, referensi ke warna seperti "warna primer," yang didefinisikan dalam buku merek).

Menentukan detail semacam ini dapat mempermudah kami untuk menerapkannya sebagai pengembang, karena hal ini memperjelas maksud yang mendasarinya sehingga kami dapat mengoptimalkan dan menggunakan kembali implementasi, daripada berpotensi menerapkan ulang kode yang sama karena tidak jelas bahwa itu adalah penggunaan ulang pola desain. Menentukan tipografi dalam hal primitif umum juga sangat ampuh. Dalam hal menentukan tata letak, penyelarasan item pada halaman dapat memberikan efek estetika yang menyenangkan. Hal ini terkadang disebut sebagai ritme, dan halaman yang tidak memiliki ritme dapat terlihat berantakan dan mengganggu.

Atribut seperti jarak vertikal yang merata dan penyelarasan item secara horizontal berkontribusi pada estetika dan nuansa ritme. Alat umum untuk membantu menegakkan ritme pada halaman adalah dengan menerapkan "kotak" pada halaman. Kotak terdiri dari kolom, dan menentukan lebar dan selokan kolom (ruang kosong untuk memberi ruang bagi

elemen untuk bernapas). Semua elemen pada halaman kemudian sejajar dengan batas kolom, dan saat menentukan tata letak halaman, desainer cukup menentukan jumlah kolom yang harus ditempati item. Kotak juga dapat membantu dalam mendesain situs web dengan lebar variabel dan responsif, seperti pada ponsel, karena kolom-kolom ini dapat ditentukan sebagai persentase lebar layar yang tersedia, tetapi tetap mempertahankan properti "ritme" yang diinginkan.

Banyak desainer, terutama untuk aplikasi yang lebih kompleks, melangkah lebih jauh, dan mengembangkan pustaka komponen yang dapat digunakan kembali. Dalam kasus ini, wireframe dapat merujuk pada komponen yang mungkin sudah ada di pustaka tersebut (ini terkadang disebut panduan gaya, pustaka komponen, atau pustaka pola). Jika ada komponen baru yang terlibat, komponen ini sering kali ditetapkan terlebih dahulu sebagai item yang berdiri sendiri, bukan dalam konteks halaman tempat komponen tersebut digunakan, sehingga memudahkan penerapannya untuk dapat digunakan kembali.

3.4 MENERAPKAN PENGALAMAN PENGGUNA

Setelah pengalaman pengguna dirancang, pengalaman tersebut harus diterapkan. Ini adalah area lain di mana kolaborasi antara praktisi UX dan teknisi perangkat lunak dapat menjadi sangat berharga. Sebagian besar desainer UX akan bekerja dengan seperangkat aturan dasar, terutama seputar penyelarasan dan ritme penempatan elemen pada halaman, dan pemahaman tentang aturan-aturan ini akan memudahkan pembuatan implementasi yang kuat dari desain yang telah dikembangkan oleh tim UX.

Banyak pengembang yang telah bekerja dengan kerangka kerja CSS seperti Bootstrap akan terbiasa dengan konsep kisi, di mana elemen-elemen disejajarkan secara vertikal dan horizontal di seluruh halaman. Gambar 3.3 menunjukkan kisi 12 kolom yang dihamparkan ke desain situs web. Setiap kolom memiliki selokan yang memisahkannya, yang memungkinkan item dalam kolom memiliki ruang bernapas, tetapi juga memungkinkan item individual menyebar ke beberapa kolom. Desainer UX sering kali akan menghasilkan desain yang selaras dengan konsep grid mereka sendiri, jadi penting untuk memastikan bahwa grid apa pun yang Anda gunakan dalam CSS Anda dikonfigurasi dengan cara yang sama seperti grid desainer Anda.

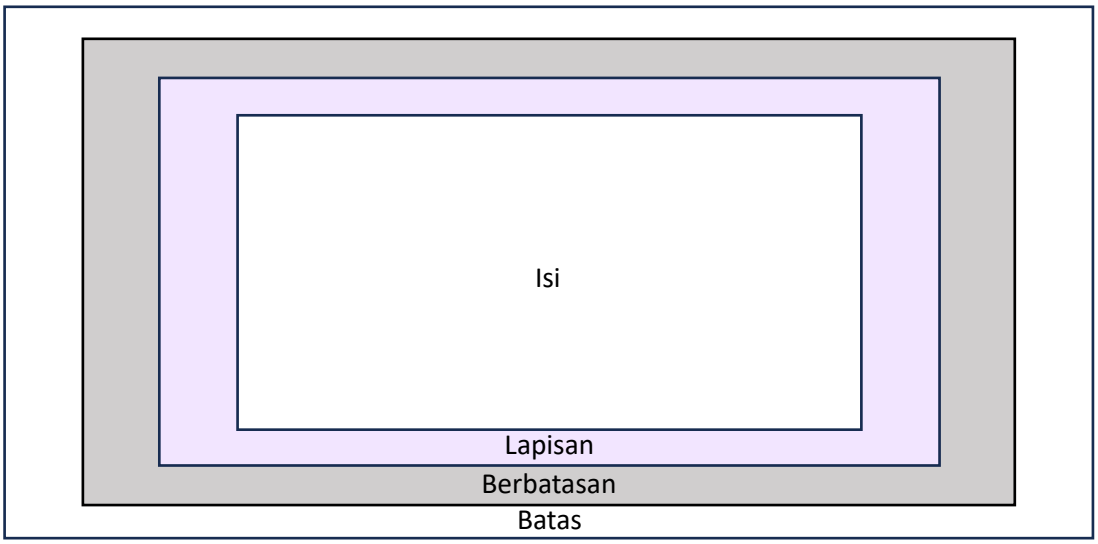
Yang juga umum adalah penggunaan padding dan margin dalam desain untuk memberikan elemen ritme yang konsisten (melalui jarak elemen yang sama). Memahami bagaimana desainer UX menggunakan konsep ini berarti Anda juga dapat mulai menggunakan variabel untuk menentukan jenis elemen ini, serta elemen lain yang akhirnya digunakan kembali, seperti palet warna.

Gambar 3.3 Kisi 12 Kolom Yang Ditata Di Atas Halaman Arahan

Anda perlu berhati-hati saat berkoordinasi dengan desainer, karena terkadang kata-kata yang sama belum tentu berarti hal yang sama bagi Anda berdua. Misalnya, cara desainer menggunakan kata "padding" dan "margin" mungkin tidak persis sama dengan cara model kotak CSS mendefinisikannya; Seorang desainer mungkin menentukan padding 12px antara teks dan batas kotak tempatnya berada. Anda mungkin berpikir padding sederhana: 12px sudah cukup di sini, tetapi jika ada atribut line-height yang diterapkan pada teks yang membuat tinggi garis lebih tinggi daripada teks di dalamnya, maka itu dapat memengaruhi apa yang dilihat desainer sebagai padding (mereka mungkin mendefinisikannya sebagai bagian atas teks ke tepi batas), dalam hal ini Anda mungkin perlu mengurangi padding atas dan bawah yang dijelaskan dalam CSS agar sesuai dengan padding yang diharapkan desainer.

Mempelajari perbedaan antara domain desain dan implementasi, melalui pembelajaran atau sekadar komunikasi terbuka, dapat membantu memperjelas maksud dan menghindari kesalahpahaman semacam ini. Ini berlaku dua arah, tidak hanya bagi pengembang untuk memahami bahasa desain, tetapi juga bagi desainer untuk memahami bahasa implementasi, karena keduanya dapat berkembang untuk mengaburkan batasan dan bekerja lebih erat bersama. Model kotak CSS merupakan konsep penting untuk dipahami, termasuk perbedaannya dari cara desainer mengekspresikan margin, padding, dan ukuran. Gambar 3.4 menunjukkan bagaimana ukuran berbagai bagian kotak disusun bersama menjadi ukuran keseluruhan.

Model kotak CSS digunakan untuk menentukan bagaimana spesifikasi ukuran komponen ditafsirkan saat ditampilkan di halaman. Model kotak asli pertama kali ditetapkan pada tahun 1996 oleh W3C dengan mengambil lebar seperti yang ditentukan dalam CSS lalu menambahkan padding, border, dan margin untuk menentukan lebar seluruh elemen. Namun, hingga Internet Explorer 6, Microsoft menerapkan interpretasinya sendiri terhadap model kotak, di mana lebar menentukan semuanya hingga border, dan hanya margin yang ditambahkan.



Gambar 3.4 Model Kotak CSS

Microsoft akhirnya mengganti implementasinya dengan yang terstandar, tetapi banyak yang merasa bahwa definisi Microsoft lebih berguna, dan dalam CSS3, opsi untuk mengubah model kotak CSS diperkenalkan dengan perintah box-sizing. Gaya W3C default dikenal sebagai content-box, dan alternatif Internet Explorer didefinisikan sebagai border-box. Untuk menghasilkan tampilan dan nuansa yang konsisten di seluruh situs, desainer sering kali menggunakan kembali komponen. Ada berbagai pola umum yang digunakan untuk melakukan ini, tetapi semuanya berasal dari prinsip yang sama yaitu memiliki hierarki komponen yang disusun untuk menghasilkan hasil akhir.

Sistem populer untuk melakukan ini didefinisikan oleh Brad Frost dan dikenal sebagai desain atom. Dalam desain atom, ada beberapa tingkat hierarki:

- Atom
- Molekul
- Organisme
- Templat
- Halaman

Atom adalah unit desain terkecil, dan sering kali memiliki pemetaan satu-ke-satu dengan elemen HTML misalnya, tombol generik atau kolom teks. Ini sering kali mencakup semua kemungkinan status—misalnya, tombol dalam status nonaktif atau melayang, atau kolom teks tempat validasi gagal. Atom digabungkan menjadi molekul, yang merupakan komponen yang lebih berguna karena mewakili beberapa fungsi yang berbeda. Molekul mungkin merupakan pencari alamat atau kotak pencarian, sesuatu yang mewakili bagian yang berbeda dari fungsi pengguna tetapi tidak terlalu banyak sehingga menjadi tahap perjalanan dengan sendirinya.

Lapisan di atas molekul dikenal sebagai organisme. Organisme adalah elemen khas halaman yang secara teori dapat digunakan kembali, tetapi sering kali hanya digunakan sekali. Header dan footer yang ada di mana-mana dari halaman web mungkin merupakan organisme yang digunakan kembali, tetapi organisme "konfirmasi pengiriman", yang terdiri dari atom dan

molekul untuk membuat formulir dan menangkap bit informasi lainnya, hanya akan digunakan sekali dalam proses pembayaran. Templat menempatkan organisme ke dalam struktur (tanpa konten) yang pada dasarnya merupakan tampilan keseluruhan bagi pengguna. Halaman adalah templat yang telah diisi dengan konten, baik secara dinamis, jika berupa halaman informasi produk atau sejenisnya, atau secara statis, jika lebih mirip halaman "Tentang Kami".

Hal yang baik tentang struktur semacam ini adalah ia memetakan dengan baik cara penerapan spesifikasi desain menggunakan HTML dan CSS. Atom dapat diimplementasikan sebagai kelas CSS, yang diterapkan pada elemen HTML biasa. Molekul dan organisme sering diimplementasikan baik sebagai bagian templat atau sebagai komponen dalam pustaka DOM virtual seperti React, lalu disusun bersama menjadi templat yang diisi dengan data nyata untuk membuat halaman. Aspek terpenting lainnya untuk diterjemahkan dari desain menjadi implementasi adalah tipografi—tampilan dan nuansa teks pada halaman. Tipografi adalah salah satu bagian desain yang paling banyak dipelajari, sejak awal mula huruf cetak di Asia Timur pada awal milenium kedua.

Banyak penganut tipografi yang mengecam kurangnya kontrol yang diberikan teknologi web kepada desainer dan pengembang atas rendering teks, tetapi banyak yang telah mengatasi tantangan tersebut, dan dengan kontrol yang tepat, tipografi yang baik dapat dicapai. Seperti halnya komponen halaman individual dalam desain atom, sebaiknya desain serangkaian atom tipografi yang kemudian disertakan dalam komponen individual, daripada terus-menerus menerapkan ulang aturan tipografi per komponen. Ini dapat memberi Anda tingkat konsistensi yang tinggi dalam tampilan dan nuansa teks di seluruh aplikasi Anda.

Anda mungkin tergoda untuk langsung menerapkan aturan tipografi ke tajuk dan tag paragraf, tetapi ini dapat mengurangi fleksibilitas. Misalnya, jika struktur visual halaman Anda mengharuskan Anda untuk melewati satu tingkat tajuk, maka hal itu akan menimbulkan masalah aksesibilitas ke aplikasi Anda. Selain itu, Anda mungkin ingin memberi gaya pada elemen seperti label formulir di berbagai tingkatan tergantung pada konteksnya, jadi memiliki serangkaian kelas CSS atau campuran SCSS adalah pendekatan yang lebih fleksibel.

Sekolah populer menganjurkan pemberian nama generik pada atom tipografi Anda yang tidak menautkannya ke elemen halaman tertentu, yang dapat memberi desainer dan pengembang lebih banyak kebebasan untuk menggunakan kelas yang tepat, daripada merasa canggung saat menerapkan kelas yang disebut "tajuk" ke tag <p> jika itu adalah hal yang tepat untuk dilakukan. Global Experience Language milik BBC, misalnya, menggunakan nama seperti “Primer,” “Canon,” dan “Trafalgar” sebagai pengenal tipografi.

Tidak mengherankan bahwa transisi desain digital dari yang analog dengan desain cetak menjadi merangkul metode yang berfokus pada UX yang kita lihat saat ini telah terjadi seiring dengan kebutuhan dan peningkatan desain responsif. Desain responsif adalah tentang membuat situs atau aplikasi Anda berfungsi di semua ukuran layar. Bagi banyak desainer, ini adalah perubahan pola pikir yang signifikan, karena sebelumnya desain "sempurna piksel" diproduksi dan diharapkan untuk diimplementasikan.

Dalam desain responsif, konsep breakpoint diperkenalkan. Breakpoint biasanya didasarkan pada lebar perangkat (tetapi dapat juga didasarkan pada atribut seperti tinggi

perangkat atau rasio aspek), dan desainer akan sering mendesain setiap komponen atau halaman dengan mode untuk kedua sisi tata letak, mungkin mengubah atribut tipografi, spasi, atau tata letak. Merupakan hal yang umum untuk menghasilkan spesifikasi desain yang mewakili setiap breakpoint, tetapi tidak mungkin untuk mencakup setiap kasus, terutama untuk layar yang lebih kecil di mana lebar variabel paling umum.

Menerapkan desain responsif sekali lagi menunjukkan bahwa hubungan kerja yang erat antara desainer dan pengembang itu penting. Pengembang harus memahami maksud perancang, karena ini bukan lagi kasus sederhana untuk sekadar mereplikasi desain piksel demi piksel, karena ada celah antara setiap struktur breakpoint representatif yang mungkin telah dirancang oleh perancang. Demikian pula, pengembang mungkin menemukan celah dalam desain lebar layar tertentu yang asumsinya (umumnya tentang seberapa banyak teks dapat masuk ke area tertentu) gagal, dalam hal ini perancang harus memikirkan ulang dan memperbaikinya.

Perbedaan lain antara dunia perancang visual dan pengembang adalah dalam definisi piksel. Anda mungkin berpikir bahwa piksel adalah sedikit cahaya pada monitor komputer, tetapi dalam CSS, ini tidak benar. Implementasi piksel yang mendasarinya disebut "piksel tampilan", tetapi piksel CSS dikenal sebagai "piksel referensi". Piksel referensi didefinisikan sebagai piksel tunggal pada layar 96dpi yang berjarak cukup jauh dari pemirsa. Ini mungkin tampak rumit, tetapi ini memungkinkan Anda untuk mendesain situs yang akan tampak berukuran kira-kira sama terlepas dari kepadatan piksel layar dan jarak pemirsa darinya.

Ponsel sering kali memiliki ukuran piksel fisik yang lebih kecil, tetapi karena ponsel dipegang lebih dekat ke mata, rasionya tetap terjaga, dan perangkat dengan layar "retina" atau DPI tinggi akan memetakan satu piksel CSS ke beberapa piksel perangkat. Penting bagi desainer untuk memahami hal ini, karena ini berarti jika seorang desainer mengerjakan desain responsif dengan mempertimbangkan ukuran layar tertentu, piksel CSS-lah yang penting, bukan piksel tampilan. Misalnya, ketika iPhone 4 (iPhone "retina" pertama) keluar, Anda masih menentukan dimensi seolah-olah lebar perangkat adalah 320 piksel, meskipun layar sebenarnya berukuran 640 piksel fisik.

Anda dapat memanfaatkan fakta bahwa ada beberapa piksel perangkat yang mendasari piksel CSS, karena teks dan grafik vektor akan ditampilkan pada resolusi yang lebih tinggi dengan peningkatan skala yang ditangani oleh perender browser yang mendasarinya. Demikian pula, Anda dapat memuat grafik bitmap yang mungkin tampak diperkecil saat ukurannya dinyatakan dalam piksel CSS, tetapi akan ditampilkan pada resolusi aslinya jika piksel perangkat mengizinkannya. Hal terakhir yang perlu dipertimbangkan adalah bahwa mungkin ada kasus di mana keterampilan desainer khusus tidak tersedia untuk Anda. Hal ini sering terjadi pada tampilan seperti panel admin atau alat pengembangan lainnya. Untungnya, dengan bekerja sama dengan desainer pada komponen lain, dan melalui desainer yang menggunakan sistem seperti desain atom, Anda seharusnya memiliki pengetahuan yang cukup untuk melanjutkan apa pun yang terjadi.

Dengan memilih dan memilah komponen dari pustaka komponen, dan menggunakan palet warna yang ditetapkan dalam buku merek, serta menggunakan komponen yang memiliki

beberapa kegunaan yang terbukti, Anda seharusnya dapat menerapkan desain yang terasa konsisten dengan tampilan dan nuansa produk lainnya. Tidak ada salahnya melakukan pengujian gerilya ad-hoc sendiri (terutama jika pengguna adalah rekan Anda, dalam kasus alat pengembangan) untuk membantu menyempurnakan bagian depan juga. Arsitektur informasi juga penting dalam hal penerapan desain.

Memahami hierarki informasi yang mendasarinya dapat membantu Anda menghindari kesalahan dalam memilih elemen HTML seperti elemen tajuk atau tag <aside>. Meskipun mungkin tergoda untuk merasa puas dengan hierarki yang diekspresikan secara visual, menggunakan semantik yang benar untuk HTML yang mendasarinya penting bagi pengguna teknologi yang dapat diakses untuk memahami desain Anda juga.

3.5 KESIMPULAN

Pengalaman pengguna adalah disiplin ilmu yang relatif modern yang muncul dari beberapa bidang yang saling berhubungan, dan paling sering dikaitkan dengan prinsip desain yang berpusat pada pengguna, di mana kebutuhan pengguna ditempatkan pertama dan terutama, dengan pertimbangan desain estetika menjadi hal yang sekunder. Garis UX menjadi kabur antara analis bisnis dan pengembang front-end, dan sebagai pengembang tumpukan penuh, Anda diharapkan untuk memahami prinsip-prinsip UX tersebut juga, bahkan jika Anda memiliki spesialis UX khusus di tim Anda.

Arsitektur informasi adalah salah satu aspek UX yang berhubungan dengan bagaimana konten diatur di situs web atau aplikasi, khususnya agar berada di tempat yang diinginkan pengguna. Aspek lain dari UX adalah pengujian, di mana asumsi apa pun dalam desain fitur atau situs dapat diuji untuk memvalidasi bahwa asumsi tersebut benar dan meningkatkan kegunaan situs. Pengujian pengguna ini dapat berkisar dari pengamatan skala kecil untuk menghasilkan wawasan yang mendalam (data kualitatif) hingga menggunakan analitik untuk menghasilkan wawasan yang dangkal tetapi luas tentang kinerja dengan seluruh basis pengguna Anda (data kuantitatif).

Mendesain dan mengimplementasikan pengalaman pengguna perlu dilakukan secara bersamaan dan menggunakan bahasa yang sama untuk meminimalkan hambatan. Menggunakan teknik untuk memecah desain menjadi beberapa komponen adalah salah satu cara untuk melakukannya, dan juga memungkinkan penggunaan ulang kode yang diimplementasikan dan meningkatkan kegunaan situs dengan mengimplementasikan pola yang sama. Mendokumentasikan dasar-dasar desain juga membantu menjaga konsistensi.

BAB 4
MERANCANG SISTEM

4.1 PERANCANGAN SISTEM DALAM PENGEMBANGAN APLIKASI END-TO-END

Saat ini, jarang sekali aplikasi web dibangun secara terpisah. Sebagai pengembang tumpukan penuh, Anda mungkin harus bekerja pada seluruh sistem, bukan hanya satu komponen dalam suatu sistem, dan melakukan ini secara efektif berarti mampu berpikir pada tingkat abstraksi yang berbeda saat memecahkan masalah. Dalam organisasi modern, sistem akan terus tumbuh dan berubah, meskipun masing-masing komponen mungkin tetap stabil. Tugas Anda adalah merancang komponen-komponen tersebut dengan cara yang memaksimalkan kelincahan yaitu, ketika sesuatu berubah di masa mendatang, transisi tidak akan sulit dilakukan.

Komponen-komponen ini dapat berada pada tingkat abstraksi apa pun, dari tingkat arsitektur organisasi yang sangat tinggi hingga kelas dan modul individual dalam basis kode, dan prinsip-prinsip ini juga dapat diterapkan pada tingkat apa pun. Jenis sistem yang akan ditemui oleh pengembang tumpukan penuh dalam karier mereka akan berkisar dari yang sederhana (seperti situs brosur) hingga yang sangat rumit (misalnya, situs manajemen akun untuk utilitas, yang memiliki banyak fungsi dan harus terintegrasi dengan komponen penagihan dan CRM pusat dari arsitektur perusahaan besar), dan penting untuk mengetahui cara bekerja dalam semua jenis lingkungan ini.

Bahkan apa yang pada awalnya tampak seperti komponen yang sederhana dan mandiri dapat dengan cepat terjerat dalam sistem lain tanpa pemikiran yang cermat, atau dapat menjadi silo yang dapat menduplikasi fungsionalitas atau mengisolasi data yang sudah ada di dalam organisasi. Sistem yang dirancang dengan baik dapat dengan cepat menjadi lebih besar daripada jumlah bagian-bagiannya melalui efek jaringan. Beberapa organisasi memiliki peran "arsitek" (atau beberapa) yang bertanggung jawab penuh untuk merancang sistem ini dan interaksi di antara mereka, biasanya pada tingkat aplikasi. Bahkan ketika ada arsitek khusus, penting untuk memastikan bahwa Anda juga memikirkan sistem secara keseluruhan, dan bukan hanya komponen tunggal di dalamnya.

4.2 ARSITEKTUR SISTEM

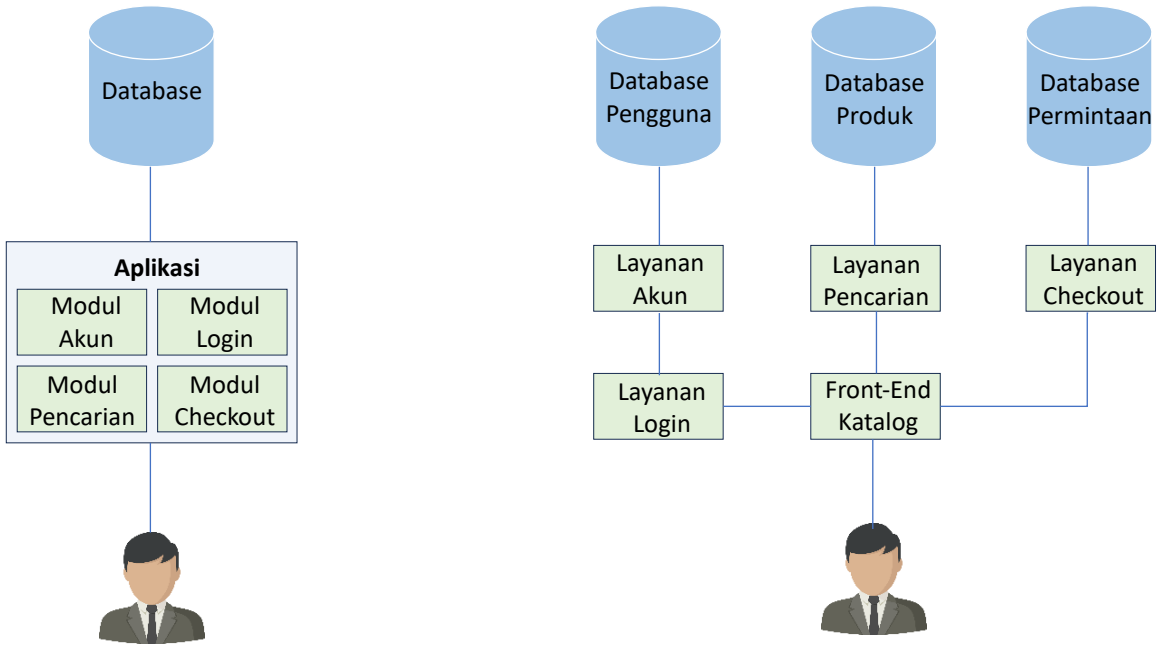
Dua jenis arsitektur sistem yang umum Anda temui adalah "monolitik" dan "layanan mikro", tetapi pada kenyataannya, sebagian besar sistem akan berada di antara kedua ekstrem tersebut. Ini umumnya mengacu pada pengorganisasian aplikasi individual dalam sistem, dan bagaimana mereka berinteraksi, bukan pada tingkat yang lebih kecil.

Kedua gaya tersebut memiliki kelebihan dan kekurangan: monolit dapat lebih mudah dibangun pada awalnya, tetapi memiliki kekurangan karena menjadi besar dan sulit diubah jika beberapa tim bekerja pada basis kode yang sama. Layanan mikro memerlukan platform yang kuat untuk berkembang, tetapi dapat lebih mudah diadaptasi dan diubah di masa mendatang.

Apa Itu Layanan Mikro?

Layanan mikro adalah layanan kecil yang bertanggung jawab atas satu bagian sistem. Layanan ini dapat digunakan dan ditingkatkan skalanya secara independen dari bagian sistem lainnya, dan berkomunikasi melalui API yang terdefinisi dengan baik. Layanan mikro dapat dianggap sebagai penerapan prinsip tanggung jawab tunggal dari SOLID (dibahas lebih rinci nanti dalam bab ini) pada tingkat sistem, bukan tingkat kelas. Oleh karena itu, arsitektur layanan mikro terdiri dari sejumlah layanan mikro yang digunakan dan diatur sebagai sistem yang lebih besar dari komponen-komponen individual. Sebaliknya, monolit adalah basis kode tunggal dengan satu aplikasi yang menjalankan semua fungsi sistem tersebut.

Monolit mungkin dapat ditingkatkan skalanya, tetapi peningkatan skala dicapai dengan menduplikasi seluruh sistem pada mesin lain, bukan komponen-komponen individualnya. Meskipun dapat mengekspos API agar sistem eksternal dapat terintegrasi dengannya, komponen internal sistem berkomunikasi melalui pemanggilan metode atau fungsi. Gambar 4.1 menunjukkan kontras antara kedua arsitektur ini. Meskipun sistem layanan mikro tampak lebih rumit, hal ini terjadi karena sistem tersebut memaksa Anda membuat interaksi antar layanan menjadi eksplisit, bukan model interaksi tersembunyi yang berpotensi rumit dalam satu aplikasi.



Gambar 4.1 Diagram Sistem Monolit Versus Yang Setara Dengan Layanan Mikro

Saat bekerja dengan sistem besar, arsitektur layanan mikro yang dirancang dengan baik akan memungkinkan Anda membatasi perubahan dan pertumbuhan tersebut ke bagian-bagian individual sistem, yang berarti dapat bergerak lebih cepat karena lebih sedikit bagian yang bergerak yang terlibat. Namun, hal ini dapat sulit dirancang. Untuk sistem kecil, biaya pengelolaan layanan mikro dapat terlalu besar untuk ditanggung, dan dapat memperlambat pengembangan awal. Layanan mikro lambat untuk dimulai, tetapi sering kali memungkinkan Anda mempertahankan kecepatan pengembangan yang baik yang dapat menjadi rumit

dengan desain monolitik.

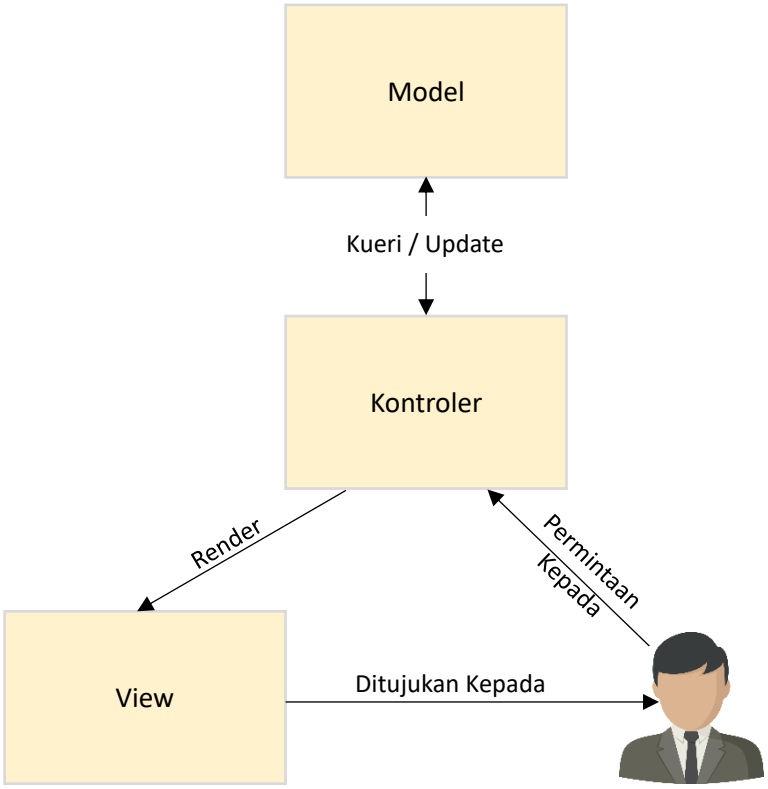
Bahkan saat merancang sistem dengan mempertimbangkan pertumbuhan, akan sangat membantu untuk memulai dengan satu aplikasi dan basis kode, lalu memecahnya menjadi beberapa layanan mikro saat aplikasi tunggal tersebut mulai menjadi terlalu besar. Trik untuk merancang sistem pada tingkat aplikasi adalah membuat setiap komponen sekecil yang dibutuhkan, tetapi tidak lebih kecil. Ini lebih mudah diucapkan daripada dilakukan, dan tidak ada ilmu pasti atau aturan yang harus diterapkan. Desain sistem lebih merupakan seni, dan sangat bergantung pada konteks, jadi akan berguna bagi tim untuk mendesain secara kolaboratif. Mengikuti pola untuk arsitektur perangkat lunak yang mungkin sudah ada di dalam organisasi Anda adalah awal yang baik, karena dapat mempermudah integrasi terhadap ketergantungan. Arsitektur perangkat lunak bukanlah daftar teknologi yang digunakan untuk membangun sistem Anda.

Pada titik arsitektur dirancang, Anda seharusnya hanya peduli tentang menangkap konsep dan interaksinya, lalu menentukan di mana mereka berada dalam sistem. Setelah Anda mengidentifikasi ini, Anda dapat menggunakan persyaratan non-fungsional Anda untuk menentukan properti yang harus dimiliki setiap komponen dalam sistem Anda, lalu menemukan teknologi terbaik yang cocok untuk masing-masing komponen tersebut. Mencoba memaksakan pilihan teknologi ke dalam arsitektur sebelum konsep, interaksi, dan sistem diidentifikasi dapat menyebabkan keputusan yang mengorbankan tujuan pembuatan arsitektur perangkat lunak yang baik.

4.3 MENGIDENTIFIKASI KONSEP

Hierarki aplikasi berjenjang merupakan pendekatan umum untuk mendesain sistem, baik itu arsitektur Model-View-Controller (ditunjukkan pada Gambar 4.2) yang ada di sebagian besar aplikasi web (atau variasinya, seperti Model-View-View Model) atau model tiga tingkatan dari arsitektur perusahaan. Untuk menyesuaikan sistem Anda ke dalam model-model ini, pertama-tama Anda perlu mengidentifikasi dua hal: konsep domain yang Anda pedulikan (misalnya, ini bisa berupa pelanggan, inventaris stok, atau artikel berita), dan cara pengguna berinteraksi dengan konsep-konsep tersebut.

Konsep domain akan membentuk lapisan model Anda dalam aplikasi Model-View-Controller (MVC), atau komponen logika bisnis/persistensi Anda dalam arsitektur tiga tingkat. Metode interaksi akan membentuk tampilan dan pengontrol (atau model tampilan) dari aplikasi MVC, atau lapisan presentasi dalam arsitektur tiga tingkat. Dalam MVC, tampilan berkaitan dengan penyajian detail yang sebenarnya kepada pengguna (dalam kerangka web, ini sering kali berupa templat HTML), sedangkan pengontrol berkaitan dengan perolehan semua data dan nilai yang diperlukan untuk membuat templat, dan melakukan tindakan apa pun yang mungkin diminta oleh permintaan.



Gambar 4.2 Arsitektur Model-View-Controller

Mungkin Anda tergoda untuk menambahkan logika yang berkaitan dengan model ke dalam pengontrol dalam aplikasi MVC (misalnya, membangun model), tetapi pengontrol terlalu terikat pada tampilan tertentu, dan sebagian besar logika sebenarnya milik konsep itu sendiri, sehingga model menjadi tempat yang lebih tepat untuknya. Pendekatan "model gemuk, pengontrol tipis" dapat membuat pemfaktoran ulang arsitektur menjadi jauh lebih mudah di kemudian hari, karena pendekatan ini memungkinkan Anda untuk mengekstrak konsep ke dalam API-nya sendiri, yang dibagikan ke beberapa komponen.

Dalam dunia berorientasi objek, sisi buruk pendekatan model gemuk adalah pendekatan ini dapat membuat kelas Anda membengkak secara signifikan. Daripada menambahkan banyak metode ke kelas, Anda dapat mengimplementasikan kelas berguna lainnya yang membantu Anda memanipulasi model. Jika Anda menggunakan bahasa yang memadukan paradigma yang berbeda, seperti JavaScript atau Python, maka mengimplementasikannya sebagai fungsi juga dapat membantu, dengan hanya menyisakan satu status dalam model. Ada banyak jenis pembantu yang dapat Anda pisahkan dari model.

Misalnya, pembangun berguna untuk membantu membangun model (dari formulir, misalnya), dan logika validasi juga dapat diekstraksi saat diperlukan. Pola Model-View-View Model (MVVM) berbeda dari MVC karena pengontrol menghilang dan Model Tampilan terikat secara khusus ke tampilan. Dalam pengontrol MVC yang umum, Anda mungkin memiliki pengontrol dengan metode untuk menangani GET dan satu untuk menangani POST. Untuk kasus GET, pengontrol akan mengambil model yang sesuai dan merender tampilan dengan meneruskan nilai. Dalam kasus POST, pengontrol akan membaca permintaan, melakukan validasi yang sesuai, lalu menerapkan perubahan tersebut ke model.

Dalam MVVM, model tampilan malah "terikat" ke tampilan, sehingga saat tampilan dirender, pengambil pada model tampilan menyediakan nilai dalam templat dengan tepat. Untuk menangani kasus POST, penyetel digunakan untuk mengembalikan perubahan. Selain variasi MVC seperti MVVM, MVC telah berkembang melampaui fungsi awalnya. Meskipun banyak kerangka kerja web sisi server, seperti Laravel milik PHP, Rails milik Ruby, atau Django milik Python menggunakan MVC (meskipun Django secara membingungkan menggunakan istilah Model, View, dan Template untuk Model, Controller, dan View, secara berurutan), tidak jarang melihat sedikit adaptasi pada arsitektur MVC, terutama dalam hal-hal seperti memulai tindakan yang berjalan lama.

Garis-garis MVC cenderung menjadi kabur dalam kode sisi klien. Karena JavaScript juga menjadi lebih umum di server, tidak jarang menemukan aplikasi NodeJS yang tidak secara ketat mengikuti pola MVC. Hal ini sering kali terjadi karena kecenderungan aplikasi JS dibuat dengan menyusun pustaka bersama-sama, daripada memilih kerangka kerja. Namun, banyak aplikasi (kadang-kadang berdasarkan desain, kadang-kadang melalui kecenderungan alami, karena merupakan cara yang efektif untuk mengatur masalah) yang menggunakan NodeJS akan memiliki struktur seperti MVC, meskipun tidak secara eksplisit.

Dalam kombinasi React+Redux yang populer, biasanya aplikasi Anda disusun sedemikian rupa sehingga Redux store menjadi model Anda, lalu komponen React menjadi tampilan. Fungsi Redux connect() kemudian memetakan status di store dan tindakan yang dapat didistribusikan ke properti, dan pemetaan ini menyediakan peran pengontrol. Namun, tampilan juga dapat menumpuk komponen lain, yang mungkin dibungkus dalam connect(), sehingga tampilan dapat menyertakan pengontrol lain. Metode hierarkis ini mungkin tampak seperti MVC dari dekat, tetapi dari jauh terlihat jelas bahwa itu bukan, meskipun pemisahan masalah tetap berguna.

MVC sering kali gagal saat ada interaksi UI yang kaya. Misalnya, saat memanipulasi UI, tidak setiap tindakan perlu dipertahankan dalam model, tetapi ada beberapa status sementara yang disimpan di suatu tempat. Jika Anda ingin mengimplementasikan antarmuka drag-and-drop, maka dalam MVC murni, Anda harus mempertahankan setiap gerakan ke store yang melalui pengontrol. Sebaliknya, biasanya "View" menyimpan statusnya sendiri, hanya melalui pengontrol saat item dijatuhkan, sehingga model diperbarui dengan benar. MVC merupakan titik awal yang baik, tetapi Anda tidak boleh membatasi diri secara artifisial hanya pada pola tersebut pola tersebut memiliki keterbatasan jika pola tersebut juga boleh dipecah.

Mirip dengan cara aplikasi React+Redux yang tersusun dari banyak lapisan hal yang secara longgar dianalogikan sebagai lapisan tampilan dan pengontrol, Anda juga akan sering mendapati diri Anda melakukan hal yang sama di aplikasi Anda. Kode sisi server Anda akan sering terstruktur dalam MVC, bersama dengan kode UI yang kaya, meskipun keduanya mungkin tidak selalu terhubung. Meskipun beberapa model mungkin sama, kemungkinan kode sisi server dan sisi klien Anda akan sedikit berbeda, dan perhatian pengontrol akan berbeda. Kode sisi server sering kali lebih memperhatikan validasi dan keamanan, sedangkan kode sisi klien mungkin kurang memperhatikan faktor-faktor ini.

Saat mengidentifikasi konsep yang membentuk model Anda, penting juga untuk

memahami konteks saat Anda membuat identifikasi tersebut. Desain berbasis domain (DDD) adalah metodologi dalam rekayasa perangkat lunak yang membantu mengelola masalah ini. Inti dari DDD adalah konsep domain: hal-hal yang diketahui dan dilakukan organisasi Anda yang relevan dengan suatu proyek. Domain didasarkan pada realitas organisasi Anda, dan melalui realitas itulah kita dapat mengidentifikasi model-modelnya. Dengan memulai pada level apa yang saat ini dilakukan orang-orang di organisasi Anda dan bagaimana mereka merujuk pada berbagai hal, dan menggunakan ini sebagai dasar untuk membangun model-model Anda, akan menjadi lebih mudah untuk membangun perangkat lunak yang sesuai dengan tindakan yang sebenarnya ingin dilakukan orang-orang.

Terkadang, kumpulan model ini dikenal sebagai ontology yaitu, struktur yang mendefinisikan domain Anda. Biasanya akan membantu jika menggunakan terminologi yang konsisten di seluruh sistem. Misalnya, toko e-commerce mungkin merujuk pada "item stok" di area logistik, tetapi "produk" di bagian depan, tetapi keduanya mungkin memiliki arti yang sama, jadi menggunakan satu istilah di seluruh sistem Anda dapat menyederhanakan banyak hal. Meskipun sangat menggoda untuk mengandalkan definisi model tunggal yang dapat digunakan di seluruh sistem, penting juga untuk menyadari bahwa model tersebut juga digunakan dalam konteks yang berbeda. Seperti yang saya singgung sebelumnya, konteks item dalam kode sisi klien mungkin berbeda dengan konteks penyimpanan data, jadi menggunakan variasi model yang tepat dalam konteks tersebut masuk akal.

Model-model ini mungkin merupakan implementasi yang berbeda dari konsep yang sama, tetapi mereka perlu diterjemahkan ke dalam konsep yang sama dalam konteks lain. Misalnya, berpindah dari konteks sisi klien ke sisi server dapat melibatkan POSTing formulir ke server, dan mungkin menggoda untuk langsung menggunakan kembali data POST. Akan tetapi, yang sebaiknya Anda lakukan adalah menerjemahkan antara dua konteks, menggunakan metode yang menerima data formulir model tersebut, dan mengembalikan objek baru yang masuk akal dalam konteks kode sisi server. Penerjemahan ini sering kali cukup sederhana, tetapi juga merupakan tempat yang baik untuk melakukan hal-hal seperti pemeriksaan otorisasi atau validasi.

Melakukan penerjemahan ini saat berbagai hal berpindah di antara konteks Anda dapat mengurangi bug, karena setiap konteks memiliki serangkaian asumsinya sendiri, dan memindahkan data di antara konteks tanpa menerjemahkannya dengan tepat dapat berarti asumsi yang berbeda pada data tersebut yang tidak lagi benar. Perangkap umum di sini adalah menggabungkan konsep yang serupa di domain Anda, atau bekerja pada tingkat abstraksi yang terlalu tinggi. Salah satu contohnya mungkin dalam sistem perusahaan besar, yang harus berurusan dengan konsep pelanggan dan karyawan. Mungkin menggoda untuk mencoba menyatukan ini ke dalam semacam konsep "orang", tetapi di sebagian besar sistem, konsep karyawan dan pelanggan cukup berbeda sehingga lebih baik jika tetap dipisahkan, dan kekhawatiran tentang "duplikasi" data (misalnya, jika seorang karyawan juga muncul di basis data pelanggan) tidak pernah terwujud.

Banyak konsep inti yang diidentifikasi akan memiliki status yang perlu dikelola, yang berarti perlu ada penyimpanan data dan logika bisnis untuk meminta dan memanipulasi

penyimpanan tersebut. Pengecualian di sini adalah saat komponen dan konsep untuk interaksi pengguna digunakan, yang tidak memiliki persistensi, atau hanya persistensi yang berkaitan dengan data sementara atau sesi, bukan konsep inti. Saat arsitektur terwujud, persistensi dan logika bisnis ini dapat dipecah menjadi komponen terpisah—satu untuk basis data Anda, lalu API yang berinteraksi dengan basis data tersebut dan mengimplementasikan logika bisnis Anda. Saat memodelkan arsitektur sistem Anda, ada baiknya untuk menganggapnya sebagai satu konsep yang terikat bersama.

Dalam mewujudkan arsitektur, penting untuk memilih penyimpanan data yang tepat bagi konsep-konsep ini guna memungkinkan persistensi. Penyimpanan data NoSQL populer, tetapi masih ada tempat untuk data relasional, terutama ketika integritas sangat penting, atau modelnya sangat terdefinisi dengan baik dan stabil. Hal ini dibahas lebih lanjut dalam bab Menyimpan Data. Berurusan dengan logika bisnis dari konsep domain inti dengan penyimpanan data sebagai komponen tingkat tinggi tunggal menyediakan tingkat abstraksi yang baik bagi komponen lain untuk menangani konsep-konsep tersebut.

Semua manipulasi konsep-konsep ini dalam lapisan persistensi Anda kemudian menjadi detail implementasi API, alih-alih lapisan persistensi menjadi titik integrasi itu sendiri. Beberapa aplikasi yang mengakses database yang sama secara langsung dapat menyebabkan masalah bagi integritas dan penerapan data, terutama jika ada perubahan skema yang terlibat, sehingga API yang mengimplementasikan logika bisnis membungkus database dan menjadi satu titik interaksi untuk komponen-komponen lainnya. Ingatlah bahwa tujuan di sini adalah agar setiap komponen dapat tumbuh dan bergerak secara independen satu sama lain, dan enkapsulasi ini memungkinkan hal itu.

4.4 MENGIDENTIFIKASI INTERAKSI PENGGUNA

Perjalanan dan cerita pengguna (atau epik) dapat membantu mengidentifikasi interaksi pengguna. Setelah Anda mengidentifikasi perjalanan yang ingin dilakukan pengguna, perjalanan tersebut dapat disatukan menjadi satu komponen front-end. Sangat menggoda untuk mengelompokkan interaksi pengguna berdasarkan konsep, tetapi hal ini sering kali akan menghasilkan banyak kode yang tidak terkait dalam komponen yang sama, dan idealnya Anda ingin setiap rangkaian hal terkait bergerak dengan kecepatannya sendiri. Memisahkan berdasarkan pengguna sering kali dapat berdampak besar.

Misalnya, di situs e-katalog, staf mungkin bertanggung jawab untuk memperbarui dan memelihara katalog yang kemudian dijelajahi pengguna, tetapi perjalanan pengguna untuk menjelajahi katalog sangat berbeda dari memeliharanya, jadi keduanya harus menjadi komponen yang berbeda. Bagi mereka yang telah bekerja dengan kerangka kerja yang ahli dalam membuat aplikasi CRUD (Buat-Baca-Perbarui-Hapus), hal ini mungkin awalnya tampak berlawanan dengan intuisi biasanya dalam kerangka kerja ini untuk menetapkan satu pengontrol per konsep, dengan tindakan untuk hal-hal yang terjadi pada pengontrol tersebut. Sering kali, kode yang Anda perlukan untuk menjalankan ini dibuat secara otomatis untuk Anda!

Namun, salah satu tujuan utama yang ingin kami capai adalah kemampuan untuk

mengembangkan setiap bagian sistem secara independen, tanpa menimbulkan risiko pada bagian lain sistem, sehingga kami dapat mempertahankan laju perubahan. Kisah pengguna dan interaksi dalam sistem manajemen sering kali akan berkembang ke arah yang sangat berbeda dengan yang berkaitan dengan komponen penelusuran, dan dengan menghubungkan keduanya, Anda mungkin akan merasa terbebani karena harus melakukan lebih banyak perubahan daripada jika Anda memisahkannya.

Menangani Kesamaan

Setelah mengidentifikasi masing-masing komponen dalam sistem, Anda mungkin akan memiliki pemahaman yang baik tentang batasan komponen tersebut, tetapi kemungkinan besar Anda juga telah mengidentifikasi beberapa dependensi umum dalam sistem (kemungkinan ini adalah elemen inti organisasi Anda, seperti konten, pelanggan, atau inventaris). Jika dependensi umum ini adalah konsep domain, dependensi tersebut dapat diabstraksikan bersama di balik API, dan jika merupakan interaksi, pustaka atau komponen bersama antara aplikasi front-end Anda adalah tempat terbaik untuk itu.

Bekerja dengan Dependensi Lama dan Eksternal

Di banyak organisasi, kemungkinan ada sistem di luar cakupan tanggung jawab Anda. Ini bisa berupa sistem siap pakai atau SaaS yang dibeli oleh organisasi Anda atau proyek yang dikelola oleh tim lain. Anda perlu menangkap dependensi ini sehingga Anda tahu bagaimana sistem Anda berinteraksi dengannya. Penting untuk diingat bahwa meskipun Anda mungkin memiliki pengaruh yang lebih kecil terhadap dependensi eksternal ini, dependensi tersebut tidak bersifat mutlak, dan Anda dapat meminta orang yang memilikinya untuk melakukan perubahan. Jika Anda harus meminta banyak perubahan, mungkin komponen ini seharusnya menjadi milik tim Anda, untuk mengurangi risiko memperkenalkan pemblokir. Jika sistem ini merupakan warisan, atau merupakan ketergantungan bagi sejumlah komponen dalam sistem Anda, akan sangat membantu jika Anda menyisipkan fasad atau lapisan abstraksi di antara sistem dan Anda.

Terutama jika suatu sistem merupakan warisan, memperkenalkan antarmuka baru dapat membantu dalam penghentian operasionalnya, karena antarmuka ini dapat diubah untuk mengabstraksi layanan pengganti tanpa harus memperbarui semua klien yang bergantung pada komponen warisan tersebut. Ini dikenal sebagai pola pencekik. Saat mendesain sistem Anda, penting untuk memahami komponen nondigital apa pun yang mungkin ada di dalamnya, dan bagaimana interaksi yang mereka miliki misalnya, mencetak label pengiriman, atau mengirimkan produk fisik. Anda mungkin tidak bertanggung jawab untuk membangunnya, tetapi penting untuk memastikan bahwa desain proses fisik apa pun sesuai dengan arsitektur Anda untuk menghindari integrasi yang menyakitkan di kemudian hari.

Pada akhirnya, Anda harus dapat melacak setiap interaksi dengan pengguna, data, dan respons terhadap tindakan yang dihasilkan melalui seluruh sistem, termasuk komponen dunia nyata. Memperlakukan ketergantungan nondigital ini seperti ketergantungan eksternal akan sering membantu.

4.5 INTERAKSI KOMPONEN

Penting untuk memahami hubungan antara komponen-komponen ini, dan cara mereka berkomunikasi. Cara melakukannya adalah dengan merenungkan berbagai skenario yang mungkin dilalui pengguna untuk menggunakan sistem, dan jalur serta alur kerja yang diambil pengguna untuk mencapai tujuan mereka. Ini disebut perjalanan pengguna, dan berguna untuk mengidentifikasi komponen front-end mana yang menjadi bagian dari setiap perjalanan, mempertimbangkan konsep mana yang perlu berinteraksi dengannya, dan menentukan sifat interaksi tersebut.

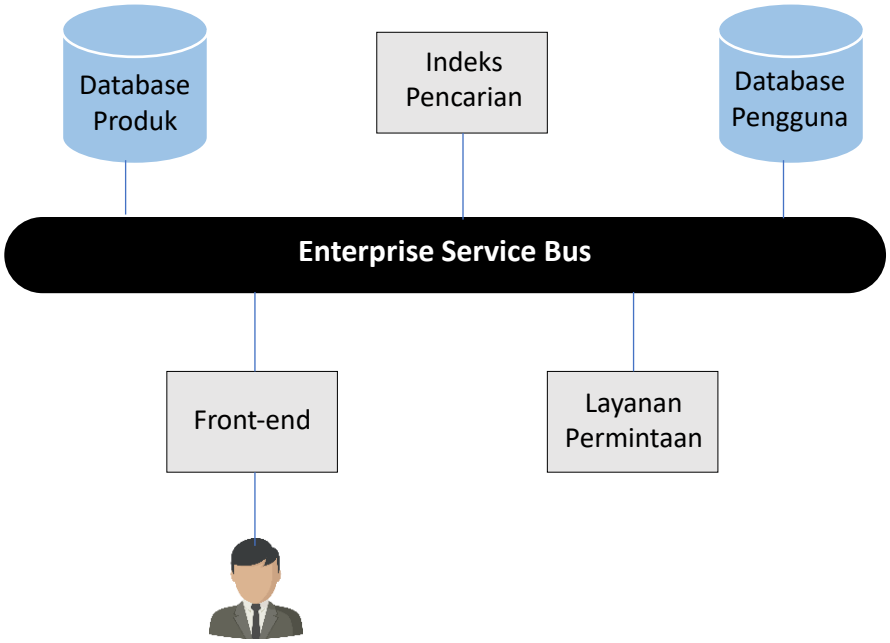
Dalam arsitektur web yang umum, ada tiga cara bagi komponen untuk berinteraksi: meminta data; mengambil tindakan secara sinkron, seperti ketika hasil tindakan, atau mengetahui kapan tindakan itu selesai, penting; atau mengambil tindakan secara asinkron, ketika hasilnya tidak penting, atau waktu yang dibutuhkan untuk melakukan tindakan ini begitu lama sehingga tidak masuk akal untuk menunggu hingga selesai sebelum menunjukkannya kepada pengguna. Kasus terakhir mencakup tindakan yang sepenuhnya digital dan memakan waktu lama, seperti transcoding video, atau melibatkan aktivitas di dunia nyata, seperti mengirimkan kartu keanggotaan atau mengirimkan produk.

Untuk jenis tindakan asinkron ini, yang perlu diketahui pengguna hanyalah apakah permintaan untuk melakukan tindakan tersebut berhasil atau tidak, sehingga asinkronisitas dapat disembunyikan di balik layanan sinkron. Ada dua jenis komunikasi umum antara komponen dalam suatu sistem: permintaan menggunakan HTTP dan antrean pesan. Untuk meminta data atau membuat perubahan sinkron, HTTP adalah pilihan yang tepat, dan jenis interaksi HTTP tertentu yang dikenal sebagai REST adalah pilihan yang popular saya akan membahasnya nanti. HTTP GET dapat digunakan untuk meminta data, dan HTTP POST atau PUT harus digunakan untuk membuat perubahan.

Untuk tindakan asinkron, di mana pengguna tidak peduli apakah tindakan tersebut berhasil atau tidak (ini bisa berupa perekaman analitik), antrean pesan sangat cocok. Namun, jika kode tersebut adalah kode front-end, sering kali mudah untuk membuat panggilan XHR fire-and-forget (XMLHttpRequest, API browser untuk membuat permintaan HTTP, dan bagian penting dari teknik JavaScript dan XML-AJAX asinkron). Dengan jenis panggilan tembak dan lupakan ini, hasil pemrosesan mungkin tidak perlu ditampilkan kepada pengguna mereka hanya perlu tahu bahwa permintaan telah dibuat meskipun bagi yang lain, pelacakan kemajuan mungkin diperlukan, di mana kueri RESTful dapat digunakan untuk berinteraksi dengan pekerjaan pada antrean pesan.

Setelah semua hal ini teridentifikasi, penting untuk menilai ulang batasan komponen-komponen ini, dan juga pada level apa komponen-komponen ini berada. Jika ada dua komponen yang tampaknya sering berkomunikasi satu sama lain, ini dapat menunjukkan bahwa keduanya sebenarnya termasuk dalam komponen yang sama. Di sisi lain, jika ada komponen utama yang tampaknya saling berkomunikasi, ini dapat menunjukkan bahwa komponen ini melakukan terlalu banyak hal, dan mungkin ada elemen-elemen yang lebih kecil yang harus diekstraksi, atau mungkin komponen ini menangkap konsep pada level abstraksi yang terlalu tinggi.

Saat mendefinisikan interaksi di atas, kita juga menentukan tindakan yang perlu dilakukan oleh suatu konsep. Beberapa tindakan ini, terutama jika tindakan tersebut besar atau tidak sinkron, mungkin termasuk dalam komponennya sendiri. Dengan mengingat hal ini, penting untuk mengidentifikasi komponen-komponen baru ini dan gaya komunikasinya. Saat beberapa peristiwa dipicu oleh satu tindakan, atau saat suatu peristiwa terjadi sebagai respons terhadap beberapa tindakan, akan sangat membantu untuk memperkenalkan antrean pesan berbasis topik. Dalam pendekatan ini, hal yang menyebabkan suatu peristiwa tidak perlu mengetahui semua tindakan yang akan disebabkan oleh peristiwa tersebut. Ia hanya mengirim pesan ke antrean topik, dan semua yang mendengarkan antrean itu akan berfungsi sebagaimana mestinya.



Gambar 4.3 Modul Yang Terhubung Ke Bus Layanan Perusahaan

Enterprise Service Bus (ESB) adalah alat populer yang, di permukaan, tampak mirip dengan antrean pesan, tetapi menawarkan serangkaian fitur yang jauh lebih mendalam. Dalam ESB, Anda menempatkan pesan ke bus, lalu ESB mendistribusikan pesan itu ke semua komponen yang membutuhkannya. ESB menjadi satu titik integrasi untuk semua komponen, seperti yang ditunjukkan pada Gambar 4.3, yang merupakan nilai jual utama sekaligus kelemahan terbesarnya. ESB sering kali mengendalikan bagaimana pesan dirutekan (alih-alih aplikasi yang menentukan topik mana yang akan dilanggani), dan ESB dapat melakukan tindakan tambahan pada pesan yang sedang dikirim, seperti mengubah mekanisme transportasi atau mengubah pesan itu sendiri, serta audit dan keamanan.

Mereka pada dasarnya menjadi satu titik kegagalan dan satu titik integrasi untuk ditingkatkan. Kasus penggunaan ESB yang paling umum adalah mengintegrasikan banyak komponen berbeda bersama-sama, karena ESB dapat menyembunyikan kompleksitas integrasi, tetapi Anda sebaiknya mempertimbangkan untuk memiliki API yang terdefinisi dengan baik pada layanan ini, karena ini dapat memungkinkan penskalaan sistem yang lebih

independen. Sangat menggoda untuk memperkenalkan ESB guna menangani interaksi antara komponen backend, terutama jika Anda beralih dari sistem lama yang sudah menggunakan ESB.

Namun, Anda harus menyadari bahwa banyak ESB dirancang untuk disambungkan ke komponen yang sudah ada (sering kali dijual oleh vendor yang sama dengan ESB) dan berfungsi sebagai tempat untuk mengonfigurasi komponen tersebut. Dari sinilah kebutuhan akan banyak fitur ESB muncul. Dengan memanfaatkan fitur-fitur ini, Anda berisiko logika halus berakhir di ESB Anda, sehingga perubahan akan tersebar di beberapa aplikasi, dan pengelolaan perubahan pada satu komponen bersama ini menjadi terkontrol ketat untuk menghindari risiko. Pendekatan layanan mikro tempat setiap komponen mempertahankan tingkat independensi yang lebih tinggi bertujuan untuk menghindari hal ini.

Menggunakan pendekatan standar untuk komunikasi (seperti antrean pesan dan REST) umumnya akan memberi Anda lebih banyak fleksibilitas daripada yang dapat diberikan ESB, dan biasanya dengan biaya yang jauh lebih rendah. Menghindari keterikatan vendor semacam ini juga dapat memberi organisasi Anda lebih banyak kelincahan; tidak perlu terikat pada kontrak yang mahal. Salah satu alasan utama untuk menggunakan ESB adalah karena ia menawarkan abstraksi terhadap banyak komponen lama yang berbeda. Untuk mengatasinya, pola umum yang dikenal sebagai pola facade dapat digunakan. Jika suatu komponen memiliki API yang aneh, atau suatu tindakan sebenarnya tersebar di banyak layanan backend yang berbeda, facade dapat diimplementasikan. Aplikasi yang menggunakan pola facade sebenarnya tidak melakukan banyak hal dengan sendirinya; ia hanya ada untuk menyajikan API umum di atas layanan lain dan menerjemahkan antara API baru tersebut dan API asli.

Meskipun ada fitur yang diinginkan pada ESB, seperti pencatatan, manfaat ini dapat diimplementasikan dengan cara alternatif. Misalnya, pustaka HTTP umum dapat digunakan di seluruh organisasi Anda dengan default untuk pencatatan, atau alat seperti gateway API dapat menyediakannya (tetapi berhati-hatilah, karena banyak gateway API menyediakan jenis fungsionalitas yang sama seperti ESB, dan harus dihindari). Jika tidak, menyediakan pustaka yang mudah digunakan atau fungsionalitas lain untuk tim berarti mereka dapat diintegrasikan dengan cara yang masuk akal untuk komponen tersebut.

Peringatan untuk menggunakan antrean pesan: pastikan Anda memahami karakteristik antrean pesan yang Anda gunakan. Misalnya, jaminan tentang pengiriman bervariasi di antara implementasi. Anda juga harus menjaga logika komunikasi aktual antrean pesan di tepi aplikasi, jika Anda perlu pindah ke implementasi antrean pesan yang berbeda (misalnya, jika Anda pindah dari cloud AWS ke cloud lain). Hal ini mengurangi risiko vendor lock-in, di mana beberapa teknologi mungkin tidak lagi masuk akal bagi bisnis untuk digunakan oleh suatu organisasi, tetapi penggabungan teknologi mencegah organisasi membuat pilihan tersebut.

Aplikasi vs. Modul

Beberapa komponen ini bisa jadi hanya modul di dalam aplikasi yang lebih besar, atau bisa jadi aplikasi itu sendiri. Aturan praktis yang baik untuk diikuti adalah "prinsip tanggung jawab tunggal," di mana setiap aplikasi hanya melakukan satu hal, tetapi melakukannya dengan baik. Ini terkadang dikenal sebagai "cara UNIX," berdasarkan prinsip desain yang

digunakan untuk membangun alat baris perintah UNIX asli. Namun, dunia nyata melibatkan lebih banyak trade-off, dan untuk platform mana pun yang Anda gunakan untuk menyebarkan, akan selalu ada overhead untuk setiap aplikasi, dan penting untuk menyeimbangkan overhead untuk setiap aplikasi dengan manfaat menjaga setiap komponen tetap kecil.

Platform yang ideal akan membuat overhead tersebut sekecil mungkin: menjadi layanan mandiri, dengan tingkat otomatisasi yang tinggi, sekaligus menyediakan fungsi-fungsi umum. Hal ini dibahas lebih lanjut dalam bab Penerapan. Sering kali juga ada implikasi biaya karena memiliki beberapa aplikasi. Sangat umum di cloud untuk menerapkan satu aplikasi per mesin (virtual), sehingga lebih banyak aplikasi memerlukan lebih banyak mesin, meskipun teknik yang dikenal sebagai kontainerisasi dapat mengatasi hal ini dengan risiko memperkenalkan kompleksitas tambahan dan tingkat isolasi yang lebih rendah pada platform Anda.

4.6 PERSYARATAN LINTAS FUNGSIONAL

Persyaratan lintas fungsi sistem Anda juga akan memengaruhi arsitektur, dan cara Anda memutuskan apakah komponen yang berbeda termasuk sebagai aplikasi yang berbeda atau hanya sebagai modul di dalam layanan yang lebih besar. Persyaratan lintas fungsi (sering disebut persyaratan non-fungsional) adalah persyaratan yang mencakup setiap fitur sistem Anda, bukan fitur yang dibangun sekali secara terpisah. Persyaratan lintas fungsi yang paling penting adalah persyaratan yang menentukan kinerja dan skala sistem yang diinginkan, serta atribut seperti keamanan.

Memenuhi persyaratan kinerja bisa jadi sulit dalam sistem terdistribusi. Umumnya, semakin banyak koneksi yang ada di antara komponen Anda untuk menyelesaikan satu perjalanan, semakin lambat komponen tersebut akan berjalan kecuali Anda mengoptimalkannya (caching adalah salah satu cara umum untuk melakukannya). Melakukan panggilan RESTful melalui antarmuka HTTP akan selalu jauh lebih lambat daripada memanggil fungsi di modul lain dalam aplikasi yang sama. Alih-alih memperkenalkan caching, Anda mungkin tergoda untuk membuat pustaka bersama yang didistribusikan dengan setiap aplikasi di sistem Anda, tetapi ini bisa menjadi anti-pola.

Jika logika bisnis Anda berubah, kini Anda perlu memperbarui pustaka tersebut di setiap komponen yang menggunakannya (yang berarti Anda harus melacak di mana pun pustaka tersebut digunakan) dan menerapkan ulang komponen tersebut. Jika perubahannya besar, berbagai bagian sistem Anda mungkin memiliki versi logika bisnis yang berbeda saat semua penerapan ini terjadi. Jauh lebih baik untuk memiliki satu tempat di mana logika bisnis Anda ditentukan, yang dapat dikelola tanpa berdampak pada sejumlah besar sistem saat berubah. Berhati-hatilah terhadap pustaka bersama yang kecil sekalipun, jika memiliki versi pustaka yang berbeda dalam produksi akan menyebabkan masalah.

Salah satu contoh kasus penggunaan pustaka bersama yang berhasil adalah pustaka HTTP. Pustaka ini membungkus pustaka permintaan HTTP, tetapi menambahkan pencatatan tambahan untuk membantu mendiagnosis masalah dan menangani skema otorisasi HTTP kami secara transparan. Memiliki beberapa salinan dalam produksi tidak menimbulkan masalah

misalnya, memperbaiki bug tidak mengharuskan versi baru pustaka tersebut segera diluncurkan, karena bug tersebut hanya memengaruhi satu layanan yang menggunakan fitur tertentu. Pustaka bersama dipisahkan dari logika backend yang dikomunikasikannya untuk memungkinkan hal ini.

Dalam hal masalah keamanan, Anda dapat mengidentifikasi tindakan mana dalam sistem Anda yang "tepercaya" dan yang tidak untuk memastikan bahwa titik akhir dilindungi dengan tepat. Aplikasi dapat dipisahkan berdasarkan garis keamanan. Misalnya, komponen yang dapat mendaftarkan pengguna (tindakan yang dapat dilakukan siapa saja) mungkin ingin berada dalam aplikasi terpisah dari aplikasi yang dapat mencari basis data pengguna (tindakan yang mungkin terbatas pada staf pusat panggilan). Hal ini memungkinkan "keamanan yang mendalam", dengan asumsi platform dan panggilan API Anda memiliki sistem keamanan yang baik, yang memungkinkan Anda menempatkan aplikasi pencarian di balik keamanan tingkat platform daripada mengandalkan keamanan tingkat aplikasi.

Keamanan tingkat platform biasanya diaudit dan ditinjau lebih ketat, dan ini mengurangi risiko bug tingkat aplikasi. Apa pun arsitektur yang Anda pilih, sering kali sangat efektif untuk membangun setiap aplikasi agar tidak memiliki status; misalnya, tidak menyimpan sesi di sisi server, atau dalam basis data yang berada di luar aplikasi Anda. Hal ini memungkinkan Anda untuk membuat penerapan dengan mengganti aplikasi di tingkat mesin dan menskalakannya secara horizontal dengan beberapa kotak di belakang penyeimbang beban, yang dapat sangat membantu dalam mengatasi skala. Teknik yang lebih terperinci untuk mencapai hal ini dibahas nanti dalam buku ini.

Caching

Saat mempertimbangkan persyaratan kinerja atau beban, satu aspek HTTP yang hebat adalah dukungan bawaannya untuk caching. Mungkin tergoda untuk menempatkan cache di mana-mana, terutama di sekitar panggilan HTTP GET Anda, tetapi ini berdampak pada pengalaman pengguna aplikasi Anda, karena informasi yang dilihat pengguna mungkin sudah kedaluwarsa. Ini khususnya bermasalah ketika pengguna telah memperbarui sesuatu sendiri (seperti menyimpan alamat baru dan kemudian masih melihat yang lama). Cache juga bersifat kumulatif, jadi jika ada beberapa lapisan caching di antara pengguna, cache dapat memakan waktu lama yang tidak disengaja untuk kedaluwarsa.

Cache juga secara efektif merupakan penyimpanan data tambahan, karena menyimpan banyak salinan data, dan ini dapat menjadi kedaluwarsa dengan kebenaran. Dalam beberapa konteks, ini tidak menjadi masalah; Misalnya, produk baru yang tidak muncul dalam hasil pencarian selama lima menit mungkin tidak bermasalah, tetapi situs berita yang menerbitkan berita terkini yang harus menunggu 30 menit untuk memperbarui indeks halaman depan adalah masalah. Penting untuk memahami dampak penundaan ini pada pengalaman pengguna Anda.

Ketika cache digunakan, akan berguna untuk menggabungkannya dengan komunikasi berbasis peristiwa sehingga cache tidak valid sebagai respons terhadap peristiwa tertentu. Ada kemungkinan juga untuk melangkah lebih jauh dari ini dan menggunakan peristiwa saja untuk membangun tampilan data yang hanya diandalkan oleh aplikasi tertentu, daripada

menggunakan API inti untuk satu titik kebenaran. Ini adalah teknik tingkat lanjut!

Mendesain untuk Kegagalan

Hal terakhir yang perlu dipertimbangkan adalah mode kegagalan sistem ini, seperti apa yang terjadi ketika satu komponen gagal atau tidak tersedia lagi. Untuk pendekatan berbasis antrean pesan (baik sebagai tindakan asinkron, atau dalam sistem berbasis peristiwa), penting untuk memahami apa yang terjadi jika pesan diterima lebih dari satu kali, dalam urutan yang salah, atau tidak sama sekali, dan untuk memilih platform yang mendasarinya berdasarkan persyaratan ini. Misalnya, saat menambahkan komponen baru ke sistem yang merespons peristiwa, pertimbangkan apakah komponen tersebut perlu menerima semua peristiwa sebelumnya yang telah terjadi dalam sistem agar mutakhir, atau apakah komponen tersebut akan merespons dengan tepat hanya untuk peristiwa yang baru.

Sebagian besar respons terhadap mode kegagalan ini dapat ditangani dalam aplikasi individual itu sendiri, dan ada teknik yang baik untuk melakukannya. Untuk komunikasi RESTful, pola pemutus sirkuit berguna untuk melindungi dari kegagalan layanan yang mendasarinya, di samping lapisan caching (terutama memanfaatkan fungsionalitas seperti basi-pada-kesalahan). Jika memungkinkan, membuat komunikasi sinkron menjadi asinkron juga efektif, karena pesan tetap berada di antrean hingga diproses.

4.7 POLA PEMUTUS SIRKUIT

Dalam kehidupan nyata, pemutus sirkuit adalah sesuatu yang aktif saat mendeteksi kegagalan. Pemutus sirkuit perangkat lunak cukup mirip. Saat pemutus sirkuit "tertutup", semuanya beroperasi secara normal, tetapi saat kegagalan terdeteksi, pemutus sirkuit "terbuka", menghentikan permintaan apa pun ke layanan backend tersebut. Setelah jangka waktu tertentu, pemutus sirkuit mengizinkan satu permintaan untuk melihat apakah backend telah pulih. Jika permintaan tersebut berhasil, pemutus sirkuit menutup dan semuanya kembali normal jika tidak, ia menunggu lagi. Penantian tersebut sering kali berupa semacam kemunduran eksponensial dengan penundaan acak, untuk menghindari membanjiri backend secara tidak sengaja saat ia pulih.

Metode untuk menentukan kegagalan dapat bervariasi. Metode yang umum adalah menambah penghitung setiap kali terjadi kesalahan, dan menyetel ulang saat berhasil, dengan kegagalan terdeteksi saat penghitung melewati angka tertentu. Hal ini memiliki sisi buruk karena membiarkan layanan yang rusak sebagian berjalan tanpa perlindungan, jadi alternatifnya adalah mengukur persentase permintaan yang gagal menjadi berhasil selama periode waktu tertentu, dan menentukan kegagalan berdasarkan persentase permintaan yang gagal selama periode waktu tertentu.

Namun, ingatlah untuk membedakan antara kegagalan yang diharapkan dan yang tidak diharapkan! 404 bisa menjadi respons yang valid dari layanan back-end jika Anda meminta sumber daya yang tidak Anda yakini keberadaannya, dan tidak boleh dihitung sebagai kegagalan. Hal ini sangat berguna jika kegagalan layanan back-end terkait dengan beban, dan memerlukan waktu untuk pulih. Selain itu, jika layanan back-end gagal karena kehabisan waktu, maka tidak repot-repot melakukan permintaan dapat mempercepat render halaman,

terutama jika itu adalah bagian aplikasi yang tidak penting yang gagal.

Terakhir, penting juga untuk memahami dampak kegagalan ini pada cerita pengguna Anda, dan saat merancang pengalaman pengguna. Misalnya, jika situs Anda adalah situs e-commerce, tetapi komponen yang menentukan apakah suatu barang tersedia atau tidak sedang tidak tersedia, apa yang harus disajikan kepada pengguna? Ada sejumlah opsi, seperti mengizinkan pesanan dan kemudian mengembalikan uang jika barang tersebut ternyata tidak tersedia, atau sekadar menolak menerima pesanan. Jawaban yang benar bergantung pada konteks organisasi Anda.

Kebutuhan untuk menangani banyak persyaratan ini mungkin tampak akan hilang jika monolit dibangun, bukan layanan mikro, tetapi pada kenyataannya sistem itu akan lebih rapuh, karena kegagalan dapat menyebar ke sistem yang tidak terkait dalam monolit yang sama, sehingga penanganan kegagalan tidak dapat dihindari, meskipun kasus yang harus ditangani berbeda. Manfaat memiliki arsitektur sistem yang fleksibel, yang terdiri dari modul-modul kecil dengan tanggung jawab tunggal dengan antarmuka yang terdefinisi dengan baik, jauh lebih besar daripada kerugian karena harus mengelola mode kegagalan antara modul-modul tersebut, dan Anda akan berakhir dengan sistem yang lebih tangguh sebagai hasilnya.

4.8 MENDESAIN MODUL

Selain mendesain arsitektur seluruh sistem, Anda juga harus mempertimbangkan struktur komponen dan modul individual dalam sistem tersebut. Perbedaan utama antara desain komponen individual dan desain sistem berasal dari overhead yang berbeda yang berkaitan dengan masing-masing bagian penyusunnya. Dalam suatu sistem, membuat komponen terlalu kecil akan menimbulkan overhead untuk mengoordinasikan komponen tersebut, tetapi membuatnya terlalu besar akan meningkatkan fleksibilitas untuk penskalaan atau membangun ketahanan dalam sistem Anda.

Namun, di dalam satu komponen, memecahnya menjadi banyak bagian yang lebih kecil akan menimbulkan sedikit overhead, sehingga Anda memperoleh fleksibilitas dan kesederhanaan karena memiliki banyak bagian yang melakukan satu hal dengan baik, tanpa kekurangan koordinasi. Aspek lain yang perlu dipertimbangkan adalah, di dalam suatu komponen, menjadi mudah untuk melakukan refaktor lintas batas modul (selama batas tersebut tidak menyimpang ke bagian lain sistem), yang berarti Anda dapat meminimalkan perencanaan arsitektur di awal di dalam komponen individual, karena biaya untuk memperbaiki kesalahan lebih kecil. Namun, biayanya tidak nol, jadi memikirkan struktur aplikasi akan bermanfaat.

Salah satu keputusan awal yang penting tentang komponen individual adalah teknologi mana yang akan digunakan: bahasa pemrograman, kerangka kerja, dan pustaka. Dalam dunia yang sempurna, memilih teknologi dan bahasa terbaik untuk mengimplementasikan persyaratan komponen tertentu akan menjadi jawaban yang jelas, tetapi komponen selalu ada sebagai bagian dari sistem yang lebih besar. Menggunakan bahasa dan kerangka kerja yang sudah digunakan di komponen lain dapat memaksimalkan efisiensi tim karena mereka tidak perlu beralih konteks. Dalam pengembangan greenfield, ini bisa menjadi pilihan yang berisiko.

Dalam pengembangan web, dua pendekatan yang dominan adalah menggunakan kerangka kerja seperti Angular, Rails, atau Django, atau membangun sesuatu dengan menggabungkan pustaka bersama-sama (mungkin dengan kerangka kerja yang relatif kecil yang memfasilitasi ini).

Kerangka kerja bisa sangat kuat, tetapi tidak fleksibel. Misalnya, Rails memungkinkan Anda membuat aplikasi CRUD monolitik dengan sangat mudah, tetapi hanya jika Anda bekerja dengan cara yang diharapkannya. Jika tidak, Anda mungkin menghadapi hambatan, dan itu mungkin sebenarnya lebih lambat daripada tidak menggunakan kerangka kerja sama sekali. Di sisi lain, penggunaan pustaka bisa lebih lambat untuk dimulai, dan melibatkan penulisan lebih banyak kode untuk menyatukan pustaka, tetapi jika Anda perlu membangun jenis aplikasi yang sangat spesifik domain, ini memberi Anda fleksibilitas untuk mengonfigurasi aplikasi sesuai keinginan.

Kerangka kerja juga lebih sulit untuk ditinggalkan, karena hampir semua kode harus dibangun dengan gaya kerangka kerja tersebut, sedangkan pustaka lebih mudah diganti dalam aplikasi karena hanya bagian yang berinteraksi dengan pustaka tersebut yang bergantung padanya. Saat memilih kerangka kerja atau memutuskan pustaka mana yang akan digunakan, pastikan Anda benar-benar memahami kemampuan dan kasus penggunaan kerangka kerja atau pustaka yang Anda pilih, dan bahwa mereka benar-benar memenuhi kebutuhan Anda. Merancang komponen individual jauh lebih mudah dipahami daripada merancang jenis sistem terdistribusi yang rumit yang sekarang digunakan banyak organisasi. Konsep layanan mikro baru muncul pada tahun 2011, sedangkan pola desain untuk aplikasi pertama kali dibahas pada tahun 1980-an, dan buku Pola Desain yang terkenal diterbitkan pada tahun 1994.

Tentu saja, sejak saat itu, bidang pengembangan aplikasi terus berkembang dan disempurnakan lebih lanjut. Pola desain adalah pendekatan yang direkomendasikan (dan telah dicoba dan diuji) untuk memecahkan jenis masalah tertentu dalam desain aplikasi. Saat menggunakan kerangka kerja, Anda sering kali dipaksa untuk menggunakan pola tertentu, tetapi saat menggunakan pustaka, Anda sering kali perlu menerapkan pola tersebut sendiri. Ada banyak pola desain, tetapi ada seluk-beluk konteks tempat Anda harus menerapkannya. Banyak pola desain dalam buku eponim tersebut dikembangkan dalam konteks bahasa C++, dan yang lainnya dalam versi awal Java. Dalam bahasa lain, fitur yang tidak ada dalam C++ atau versi Java yang lebih lama dapat membuat pola desain menjadi berlebihan.

Pola Desain juga lebih banyak berkaitan dengan pola untuk pemrograman berorientasi objek, yang telah menjadi paradigma pemrograman dominan dalam beberapa dekade terakhir namun, itu bukan satu-satunya. Bahasa seperti JavaScript dan Python memungkinkan Anda untuk memadukan gaya prosedural dan fungsional di samping OOP, tetapi memahami prinsip desain berorientasi objek memungkinkan Anda untuk menerapkan konsep yang sama pada paradigma ini juga.

Selain pola desain, banyak prinsip telah berkembang yang membantu mendorong desain dan rekayasa yang baik. Banyak dari prinsip-prinsip ini telah mengadopsi akronim yang menarik seperti KISS, DRY, dan SOLID. "Keep it simple stupid" (KISS) mendorong pengembang untuk menulis kode yang mudah dipahami oleh pengembang rata-rata untuk membantu

pemeliharaan di masa mendatang, daripada menambahkan lapisan abstraksi tambahan yang mungkin elegan tetapi tidak diperlukan untuk menyelesaikan masalah yang dihadapi. "Don't repeat yourself" (DRY) mendorong Anda untuk hanya mengimplementasikan fungsionalitas atau konsep yang dibutuhkan satu kali, lalu menggunakannya di beberapa modul jika sesuai. Penerapan DRY yang benar dibahas nanti dalam bab ini. SOLID mengacu pada lima konsep berbeda:

- Prinsip tanggung jawab tunggal
- Prinsip terbuka/tertutup
- Prinsip substitusi Liskov
- Prinsip pemisahan antarmuka
- Prinsip inversi ketergantungan

Prinsip-prinsip SOLID dapat ditingkatkan atau diturunkan tergantung pada skala yang Anda gunakan. Komponen-komponen yang lebih besar dari sistem yang dirancang dengan baik dapat dikatakan sebagai SOLID, begitu pula kelas atau fungsi individual dalam setiap komponen. Prinsip tanggung jawab tunggal, misalnya, merupakan salah satu faktor pendorong di balik layanan mikro. Prinsip tersebut menyatakan bahwa setiap kelas individual dalam sistem Anda seharusnya hanya bertanggung jawab atas satu hal.

Akibatnya, untuk setiap tanggung jawab yang diambil alih sistem Anda, harus ada satu bagian dari sistem Anda yang bertanggung jawab untuk menanganinya. Misalnya, jika suatu bagian dari sistem Anda menghitung biaya PPN, maka jika tarif PPN berubah, maka Anda hanya perlu memperbarui satu bagian dari sistem Anda, dan kelas yang sama tersebut juga tidak boleh menangani tugas-tugas lain, seperti menghitung biaya pengiriman. Menurut prinsip terbuka/tertutup, bagian kode tertentu harus terbuka untuk perluasan tetapi tertutup untuk modifikasi. Hal ini sebagian besar diterapkan pada bagian fungsionalitas yang dapat digunakan kembali.

Prinsip ini menyatakan bahwa setiap kelas lain yang menggunakan kelas Anda harus dapat memperluas fungsionalitasnya melampaui apa yang disediakan untuk memenuhi kasus penggunaannya, tanpa harus memodifikasi modul yang diwarisinya. Dalam pemrograman berorientasi objek, hal ini sering diartikan sebagai pewarisan, tetapi dapat juga berarti bahwa setiap dependensi yang ditetapkan dilakukan dengan menggunakan antarmuka, sehingga implementasi alternatif dapat digunakan.

Barbara Liskov awalnya menggambarkan prinsip substitusi Liskov dalam makalahnya tahun 1994 sebagai "Misalkan $\phi(x)$ menjadi properti yang dapat dibuktikan tentang objek x bertipe T. Maka $\phi(y)$ harus benar untuk objek y bertipe S di mana S adalah subtipe dari T." (Barbara Liskov dan Jeannette Wing, “Sebuah gagasan perilaku tentang subtipe,” Transaksi ACM tentang Bahasa Pemrograman dan Sistem (TOPLAS), vol. 16, #6, 1994). Meskipun deskripsi matematis ini mungkin tampak menakutkan, ini berarti bahwa jika Anda memperluas modul, atau menyediakan modul dengan antarmuka yang sama dan bermaksud agar modul tersebut dapat digunakan secara bergantian, maka asumsi yang dimiliki pengguna kelas tersebut tentang cara berinteraksi dan menggunakannya harus berlaku di antara semua hal yang mengimplementasikan atau memperluas antarmuka tersebut.

Misalnya, jika Anda memiliki modul yang berbeda untuk menghitung biaya pengiriman yang mengimplementasikan antarmuka yang sama, tetapi salah satunya mengharuskan Anda memasukkan berat total pengiriman, dan yang lainnya tidak, ini dapat dikatakan melanggar prinsip tersebut. Semua modul juga harus mengharuskan berat total diketahui, bahkan jika tidak ada yang dilakukan dengannya, untuk menjaga antarmuka dan asumsinya tetap sama. Jika Anda tidak menggunakan ini, maka setiap kali kelas-kelas ini digunakan dalam basis kode Anda, Anda harus menyadari adanya kasus tepi, bukannya komponen yang benar-benar dapat digunakan kembali.

Beberapa bahasa pemrograman memungkinkan Anda untuk secara formal menentukan kasus-kasus ini dalam bentuk prasyarat atau pascakondisi, dan untuk mengujinya. Prakondisi adalah pernyataan yang harus benar agar Anda dapat menggunakan antarmuka dengan benar, dan pascakondisi adalah sesuatu yang dapat Anda nyatakan tentang keluaran suatu fungsi atau metode, jika prasyaratnya terpenuhi. Misalnya, dalam kasus kalkulator PPN kita, kita dapat menyatakan prasyarat bahwa masukan harus berupa daftar objek yang terdiri dari harga, yang merupakan non-negatif, dan apakah itu bebas PPN atau tidak, yang merupakan boolean. Pascakondisi adalah bahwa ia akan memberi Anda bilangan bulat non-negatif.

Beberapa teori ilmu komputer melangkah lebih jauh dari ini untuk sepenuhnya menentukan prakondisi dan pascakondisi dalam bentuk matematika untuk membuat sistem yang secara matematis dapat dibuktikan sebagai benar, tetapi ini adalah sesuatu yang tidak mungkin Anda temukan dalam kehidupan sehari-hari. Huruf 'I' dalam SOLID adalah prinsip pemisahan antarmuka, yang menyatakan bahwa pengguna antarmuka tidak boleh diharuskan bergantung pada lebih banyak metode daripada yang sebenarnya dibutuhkan untuk menyelesaikan tugasnya. Hal ini terkait dengan prinsip tanggung jawab tunggal untuk menjaga kelas tetap kecil dan fokus pada satu hal. Jika kelas menjadi terlalu besar, atau antarmuka melakukan terlalu banyak hal, maka sebaiknya dipecah menjadi kelas yang lebih kecil dengan antarmuka yang cakupannya lebih ketat.

Misalnya, keranjang belanja dapat ditambahkan atau dibaca, tetapi selama proses pembayaran, keranjang tersebut hanya perlu dibaca. Jika antarmuka untuk keranjang belanja Anda menjadi terlalu besar, Anda dapat membaginya menjadi dua satu yang menangani penambahan dan satu lagi menangani pembacaan lalu memiliki kelas yang mengimplementasikan kedua antarmuka tersebut. Inversi dependensi adalah salah satu komponen SOLID yang paling terkenal dan, khususnya dalam kerangka kerja untuk bahasa berorientasi objek yang kuat seperti Java, komponen yang harus Anda tangani secara langsung, khususnya jika Anda ingin melakukan pengujian unit apa pun.

Pendekatan untuk membangun sistem yang berjalan dari komponen yang mengimplementasikan inversi dependensi disebut injeksi dependensi. Injeksi dependensi juga dapat menjadi masalah bagi pengembang baru, mereka yang tidak terbiasa dengan lingkungan perusahaan, dan mereka yang berasal dari bahasa OO yang lebih lemah, tetapi hal ini lebih disebabkan oleh kerangka kerja injeksi dependensi yang terlalu besar dan memberatkan daripada konsep itu sendiri. Dengan inversi dependensi, Anda tidak bertanggung jawab untuk

menemukan kelas atau modul yang Anda perlukan untuk melakukan tugas Anda; sebaliknya, Anda diberikan kelas atau modul tersebut. Terkadang hal ini dilakukan dengan menggunakan kerangka kerja injeksi ketergantungan seperti Spring, tetapi Anda juga dapat menulis kode yang menyuntikkan metode yang sesuai secara langsung (dikenal sebagai kode pengkabelan), tanpa menggunakan kerangka kerja.

Sebagai hasil dari inversi ketergantungan, Anda tidak bergantung pada contoh konkret dari ketergantungan Anda, tetapi pada antarmuka. Misalnya, kelas yang berbicara dengan layanan backend, daripada membuat contoh klien HTTP-nya sendiri, dapat diberikan klien yang mengimplementasikan antarmuka yang sama. Dalam mode pengembangan, ini bisa menjadi klien yang mengimplementasikan kebutuhan untuk bekerja melalui proxy perusahaan, tetapi dalam mode produksi, yang dapat digunakan untuk AWS, bisa jadi klien yang tidak menggunakan proxy, tetapi mungkin mengimplementasikan lapisan caching sebagai gantinya.

Meskipun prinsip SOLID diformulasikan di sekitar pemrograman berorientasi objek, prinsip tersebut juga dapat digunakan kembali untuk paradigma lain. Misalnya, dalam injeksi ketergantungan, ketergantungan sering kali diteruskan ke konstruktor, tetapi dalam gaya pemrograman fungsional, prinsip inversi ketergantungan dapat diimplementasikan menggunakan penutupan. Ambil contoh berikut dalam JavaScript. Pendekatan injeksi non-ketergantungan yang naif mungkin adalah sebagai berikut:

```
import { getProduct } from ' .. /db/products';

export default (request, response) => {
  getProduct(request.params.productId)
    .then((product) => {

      if (product === null) {
        response.status(404).end();
      } else {
        response. json({
          name: product.name,
          description: product.description
        });
      }
    })
    .catch((err) => {
      response.status(500).end();
    });
};
```

Namun, paling tidak, hal ini bisa jadi sulit untuk diuji unitnya. Dalam bentuk ini, kita mungkin harus benar-benar membuat basis data nyata dan mengisinya dengan konten pengujian. Kita ingin mengganti impor dengan cara memilih `getProduct` mana yang akan digunakan. Kita dapat melakukannya sebagai berikut:

```
export default (getProduct) => (request, response) => {
```

```
  getProduct(request.params.productId)
    .then((product) => {
      if (product === null) {
        response.status(404).end();
      } else {
        response. json({
          name: product.name,
          description: product.description
        });
      }
    })
    .catch((err) => {
      response.status(500).end();
    });
};
```

Sebaliknya, fungsi yang diekspor adalah pabrik yang mengembalikan fungsi pengontrol yang sebenarnya. Tingkat abstraksi tambahan ini dapat tampak membingungkan, dan Anda sekarang perlu menghubungkan pengontrol Anda ke dependensinya pada titik Anda membuat instance-nya (biasanya pada titik Anda menambahkan rute), yang memberi Anda tingkat fleksibilitas yang tinggi. Namun, Anda tidak selalu perlu menyuntikkan semuanya.

Misalnya, jika Anda mengandalkan fungsi murni yang telah diabstraksikan untuk dapat digunakan kembali, Anda dapat mengimpornya secara langsung daripada mengharapkannya untuk disuntikkan. Pengecualian umum lainnya adalah logger. Logger cenderung universal, jadi hampir di mana pun dalam kode Anda memerlukannya, jadi daripada menyuntikkannya, Anda dapat mengimpornya dari file (ini dikenal sebagai singleton) untuk menjaga kode Anda tetap rapi.

Refactoring

Jika menyangkut arsitektur di seluruh sistem, stabilitas dan beberapa desain di muka penting, tetapi dalam modul individual, Anda harus merasa bebas untuk mengubah struktur internal dan implementasi tanpa khawatir akan memengaruhi antarmuka. Proses ini disebut refactoring, dan berlaku untuk apa pun mulai dari level fungsi/metode individual, kelas, hingga seluruh modul. Refactoring sering dilakukan di akhir pekerjaan untuk membersihkan sampah yang terkumpul, atau sebelum pekerjaan baru untuk membuat kode menjadi lebih mudah diubah.

Refactoring bekerja dengan baik dengan pengembangan berbasis pengujian, di mana setiap pengujian yang berhasil memberikan kesempatan untuk merapikan kode, dan memiliki cakupan pengujian yang baik memungkinkan Anda untuk melakukan refactor dengan percaya diri, karena Anda tahu Anda tidak akan merusak sesuatu secara tidak sengaja. Refactoring biasanya tidak mengubah antarmuka, dan seharusnya tidak mengubah fungsionalitas. Buku Robert C. Martin Clean Code mendefinisikan seperti apa seharusnya kode yang bersih dan rapi, dan layak dibaca oleh setiap insinyur perangkat lunak.

Apa pun itu, tujuan refactoring saat Anda melakukannya adalah untuk memastikan bahwa ketika Anda, atau seorang kolega, kembali ke kode yang ditulis beberapa minggu, bulan,

atau tahun sebelumnya, Anda atau mereka dapat dengan mudah mengikuti alur dan maksud kode tersebut. Menghapus duplikasi adalah alasan umum untuk melakukan refaktor kode. Jika Anda menduplikasi hal yang sama di beberapa tempat, ada baiknya mengekstraknya ke tempat bersama, sehingga di masa mendatang hanya ada satu tempat Anda perlu mengubah kode.

Konsep ini dikenal sebagai "jangan ulangi diri Anda sendiri," yang disingkat menjadi DRY (dan kode yang mengurangi pengulangan disebut sebagai "dry"). Kesalahpahaman tentang DRY adalah bahwa hal yang tidak boleh diulang adalah kodenya. Faktanya, mengulang kode tidak apa-apa—konsep yang tidak boleh diduplikasi. Jika Anda melakukan refaktor dua konsep yang serupa, tetapi terpisah, bersama-sama karena kodenya serupa, Anda mungkin membuatnya lebih sulit untuk diikuti dan dibongkar nanti jika satu konsep berubah tetapi yang lain tidak. Demikian pula, jika Anda telah menggandakan kode hingga sulit untuk mengikuti logikanya, itu sering kali berarti Anda telah membuat kode Anda terlalu DRY.

Alat

Setelah Anda merancang arsitektur sistem, Anda harus membangunnya. Meskipun arsitektur sistem mungkin merupakan fondasinya, fondasi tetap harus dibangun, dan cara Anda membangun sesuatu terkait dengan apa yang ingin Anda bangun. Setiap organisasi memerlukan semacam platform pengembangan. Ini mungkin berupa rangkaian lengkap alat yang dihosting di cloud untuk kontrol sumber, integrasi dan penerapan berkelanjutan, atau IDE yang dikonfigurasi dengan baik pada satu mesin pengembang. Kecuali jika Anda mendesain ulang semuanya dari awal, kemungkinan besar sudah ada sesuatu yang siap pakai. Namun, menerapkan teknik arsitektur sistem ke platform pengembangan Anda juga penting.

Platform yang dikembangkan secara ad-hoc sering kali menjadi bagian yang lemah dari arsitektur yang kuat; server CI yang kurang aman telah menjadi titik masuk ke arsitektur yang aman dari peretasan besar di masa lalu. Paling tidak, platform pengembangan harus memiliki beberapa cara untuk mengelola kode sumber, membangun paket yang dapat digunakan, dan menyebarkannya ke lingkungan pengembangan, pementasan, dan produksi dengan cara yang aman dan mudah digunakan. Namun, yang tidak boleh Anda lakukan adalah merancang arsitektur dengan mempertimbangkan implementasi tertentu.

Setelah Anda mengonseptualisasikan cara kerja sistem, Anda dapat menemukan alat yang tepat untuk digunakan guna mewujudkannya. Arsitektur sistem, misalnya, tidak boleh mendikte teknologi basis data atau bahasa pemrograman tertentu. Sebaliknya, karakteristik komponen yang diinginkan harus diungkapkan, lalu alat dan kerangka kerja yang sesuai yang memenuhi karakteristik tersebut diidentifikasi. Hal ini harus selalu dipertimbangkan dalam konteks tim, dan setiap detail implementasi harus digunakan sebagai masukan bagi arsitektur.

Misalnya, jika tim pengembangan perangkat lunak sangat familier dengan teknologi Microsoft, tetapi satu-satunya basis data yang memenuhi persyaratan adalah Oracle, maka sebaiknya sesuaikan arsitektur untuk menemukan sesuatu yang lebih dapat direalisasikan. Demikian pula, terkadang opsi "terbaik" mungkin adalah bahasa pemrograman tertentu yang tidak dikenal siapa pun, dalam hal ini kompromi dapat ditemukan yang masih memenuhi persyaratan ringkasan.

Arsitektur juga harus dapat direalisasikan dalam konteks organisasi yang

membangunnya jadi, meskipun arsitektur sistem tidak boleh terikat pada teknologi tertentu, akan tiba saatnya detail implementasi tersebut harus tercermin kembali padanya. Hal ini terutama berlaku saat merancang arsitektur sistem yang mungkin menjadi bagian dari sejumlah sistem yang saling terhubung, dalam hal ini konsistensi cukup berguna.

Mengubah Arsitektur Anda

Arsitektur sistem lebih sulit diubah daripada potongan kode individual, karena merupakan fondasi untuk semua komponen sistem. Namun, mustahil bagi Anda untuk menentukan masa depan sistem Anda, jadi tidak dapat dihindari bahwa beberapa tingkat perubahan akan diperlukan. Perubahan pada arsitektur Anda juga perlu dilakukan saat sistem Anda aktif, yang memerlukan manajemen yang cermat. Beberapa perubahan ini sederhana dan mudah, seperti menambahkan komponen baru untuk mendukung fungsionalitas baru. Hampir semua sistem harus memungkinkan penambahan, tetapi sering kali menambahkan hal-hal baru ke sistem dapat menyoroti masalah atau asumsi yang buruk dalam desain sebelumnya.

Mungkin pemisahan masalah berada di tempat yang salah, atau ada masalah yang sebelumnya tidak teridentifikasi terkait penskalaan komponen sistem. Dalam kasus ini, Anda tidak hanya perlu menambahkan komponen baru, tetapi juga mengubah struktur dari apa yang sudah ada. Ini adalah salah satu keuntungan dari sistem monolitik. Karena sistem diterapkan sebagai satu unit, Anda dapat dengan mudah melakukan refaktor monolit untuk mengubah asumsi tersebut dan semua komponen internal sekaligus, dalam penerapan yang besar. Dengan layanan mikro, diperlukan pendekatan berulang terhadap jenis perubahan ini.

Salah satu cara untuk mencapainya adalah dengan menjalankan komponen lama dan baru secara paralel selama jangka waktu tertentu, lalu melakukan refaktor layanan yang menggunakan komponen yang tidak digunakan lagi untuk berhenti menggunakannya, dan akhirnya menghapus komponen yang diganti. Kita juga dapat menggunakan pola yang dibahas di atas saat beralih dari sistem lama, dengan sekarang menangani bit yang perlu diubah sebagai sistem lama dan menempatkan fasad di depannya saat beralih ke versi yang diubah, dan akhirnya menonaktifkan yang lama.

Rincian sebenarnya tentang cara perubahan dilakukan tentu saja bergantung pada perubahannya. Namun saat menggunakan layanan mikro, langkah-langkah kecil yang konstan seharusnya dapat dilakukan, yang pada akhirnya mengarah pada perubahan besar yang Anda perlukan.

4.9 KESIMPULAN

Arsitektur sistem adalah tentang merancang struktur yang sesuai untuk sistem tertentu, dan bagaimana sistem tersebut dapat terintegrasi sebagai subsistem. Sasaran arsitektur sistem adalah bertindak sebagai cara yang paling efektif untuk memecahkan masalah, sekaligus tetap cukup fleksibel untuk menanggapi perubahan di masa mendatang. Mengubah arsitektur sistem yang sedang berjalan jauh lebih sulit daripada mengubah detail masing-masing komponen dalam sistem tersebut.

Dua ekstrem arsitektur sistem adalah monolit satu aplikasi yang melakukan segalanya

dan layanan mikro, tempat serangkaian layanan yang lebih kecil bekerja sama untuk mengimplementasikan suatu tujuan. Monolit bagus untuk sistem kecil, tetapi dapat menjadi tidak praktis saat sistem tersebut berkembang, sedangkan layanan mikro menawarkan banyak fleksibilitas sebagai ganti overhead koordinasi dan distribusi yang lebih tinggi. Arsitektur sistem dimulai dengan mengidentifikasi fungsi inti yang akan ditangani oleh sistem, lalu tindakan yang mungkin dilakukan oleh pengguna sistem.

Kita kemudian dapat menggunakan informasi ini untuk memperoleh batasan masing-masing komponen sistem, dan sifat komunikasi antar-modul di antara komponen-komponen tersebut. Sebagian besar komunikasi modul dapat diklasifikasikan sebagai permintaan-respons atau berbasis asinkron/antrean. Arsitektur juga harus mempertimbangkan persyaratan lintas fungsi seperti keamanan, kinerja, dan ketahanan. Pola seperti caching atau circuit breaker dapat dimanfaatkan untuk membantu mewujudkan kebutuhan ini. Arsitektur juga harus dapat direalisasikan, dengan alat yang sesuai seperti database yang diidentifikasi berdasarkan persyaratan suatu komponen. Prinsip SOLID adalah lima konsep penting yang harus dipatuhi setiap komponen dalam sistem Anda pada skala tingkat tinggi dan tingkat rendah, karena pada akhirnya menghasilkan komponen yang dapat digunakan kembali dan terhubung secara longgar.

BAB 5
ETIKA

5.1 ETIKA DALAM REKAYASA PERANGKAT LUNAK

Rekayasa perangkat lunak sebagai sebuah profesi memiliki pemahaman yang sangat tidak matang tentang etika rekayasa, dan dampak yang kita buat pada dunia. Dalam disiplin ilmu rekayasa lainnya, etika dan kode etik tertanam dalam badan profesional yang mewakili para praktisi. Rekayasa perangkat lunak sebagai sebuah disiplin ilmu sebagian besar telah mengesampingkan badan profesional, lebih memilih praktik terbaik dan koordinasi yang muncul secara alami, daripada dari badan profesional. Sangat jarang menemukan seorang insinyur berizin di sebuah organisasi digital seperti halnya menemukan seseorang yang tidak (atau tidak berusaha) memiliki kualifikasi di bidang teknik sipil.

Pro dan kontra dari badan profesional ini dapat diperdebatkan panjang lebar, meskipun konsensusnya tampaknya adalah bahwa mereka tidak memberikan banyak nilai tambah, dan kualifikasi (tentu saja untuk pengembang) tidak terlalu penting bagi para pemberi kerja. Di beberapa negara, "insinyur" adalah istilah yang dilindungi, dan hanya individu yang berkualifikasi yang dapat menggunakannya mirip dengan dokter medis, atau pengacara. Ini untuk melindungi citra seorang insinyur, dokter, atau pengacara sebagai seseorang yang dapat dipercaya. Di negara-negara ini, insinyur perangkat lunak sering disebut sebagai pengembang perangkat lunak.

Ciri khas profesi adalah bahwa seorang profesional haruslah seseorang yang dapat dipercaya, dan ini dicapai baik dengan memasukkannya ke dalam pelatihan profesional tersebut (yang paling terkenal adalah sumpah Hipokrates untuk profesional medis) atau dengan registrasi dan sertifikasi yang dipimpin oleh legislatif atau industri (di Inggris, misalnya, tukang gas terdaftar CORGI menyiratkan tingkat kepercayaan yang tinggi, yang khususnya penting ketika berhadapan dengan sesuatu yang berbahaya seperti gas). Satu hal yang sama dari pendekatan etika ini adalah bahwa orang-orang yang menjalankan peran dalam profesi ini bertujuan untuk tidak melakukan hal yang merugikan.

Industri pengembangan perangkat lunak memiliki jalur masuk yang jauh lebih sedikit formal daripada banyak profesi lain, dan tidak ada skema sertifikasi yang diakui secara luas. Dikombinasikan dengan kurangnya badan profesional (yang memiliki kode etik mereka sendiri), ini adalah situasi yang berpotensi berbahaya, karena kita juga tidak mengatur diri sendiri secara efektif. Ada beberapa upaya nyata untuk mengatasi kekurangan ini, yang paling nyata adalah Geek Feminism dan gerakan keberagaman dalam teknologi, tetapi ini hanya difokuskan pada beberapa aspek perilaku etis. Gerakan Code Craftsmanship juga berfokus pada serangkaian etika seputar pendekatan yang diambil untuk pengodean yang sebenarnya, tetapi tidak membahas keseluruhannya.

Etika merupakan komponen penting dari masyarakat mana pun, dan umumnya paling efektif ketika muncul sebagai praktik melalui konsensus daripada melalui undang-undang. Jadi, seperti apa kode etik bagi pengembang perangkat lunak? Pada intinya, ini sama dengan etika

yang harus kita praktikkan sebagai individu dalam masyarakat, dan kode yang lebih formal dari profesi lain: kita harus berusaha untuk tidak melakukan hal yang merugikan, atau setidaknya meminimalkan kerugian.

Meskipun di permukaan mungkin tampak mudah untuk mencapainya bagaimanapun juga, jika aplikasi belanja mogok karena pendekatan rekayasa yang ceroboh, dampaknya tidak akan sama dengan jembatan yang runtuh (meskipun tentu saja, ada banyak sistem perangkat lunak yang penting untuk keselamatan juga) tetapi dampak aplikasi kita terhadap masyarakat sering kali lebih abstrak. Hal ini tidak membuatnya kurang penting untuk dipertimbangkan.

5.2 PRIVASI

Salah satu ketegangan utama yang ditimbulkan oleh komputasi personal berkaitan dengan privasi. Komputer, dan khususnya ponsel, telah menjadi perangkat yang sangat personal, dan komputer serta layanan yang berinteraksi dengannya menyimpan banyak data yang berkaitan dengan pengguna, yang mungkin tidak ingin membagikannya kepada semua orang. Sebagai seorang profesional, Anda diharapkan untuk menghormati privasi pengguna Anda, tetapi pada saat yang sama, data ini dapat berharga bagi suatu organisasi dan menginformasikan keputusan Anda untuk membangun produk yang lebih baik atau meningkatkan efektivitas strategi seperti periklanan.

Sebuah analogi yang sering muncul terkait dengan data pengguna adalah minyak, yang sering kali berfokus pada nilai data yang dirasakan dan bagaimana data tersebut diinginkan untuk "ditambang". Ketika dimurnikan, minyak sangat berguna dan berharga, tetapi sisi sebaliknya adalah bahwa dalam bentuk mentahnya, minyak tersebut beracun, dan jika tumpah, dapat menghancurkan seluruh ekosistem. Pengumpulan dan pemrosesan data harus dianggap berisiko, dan cara termudah untuk mengurangi risiko tersebut adalah dengan mengurangi jumlah data yang Anda kumpulkan.

Hal ini menjadi rumit terutama jika beberapa tingkat data pribadi diperlukan secara teknis misalnya, alamat IP dapat dianggap sebagai informasi pribadi, tetapi alamat IP diperlukan agar pengguna dapat membuat koneksi ke server. Hanya karena Anda dapat mengumpulkan data tidak selalu berarti Anda harus melakukannya. Kegilaan "big data" telah mengabadikan gagasan untuk mengumpulkan semua data yang mungkin Anda bisa dan kemudian melihat apakah ada tren yang muncul darinya. Namun, proses untuk menganalisis semua data itu rumit, dan bagi banyak organisasi, impian ini tidak pernah sepenuhnya terwujud. Sebaliknya, mengumpulkan data yang lebih terfokus dan bermakna lebih mudah dianalisis dan lebih mudah dipraktikkan secara etis.

Legislator juga semakin bereaksi terhadap apa yang dianggap sebagai kegagalan industri untuk mengatur diri sendiri, dan menerapkan undang-undang baru untuk melindungi warga negara dari pengumpulan data yang berlebihan dan penggunaan data mereka yang dipertanyakan. Peraturan Perlindungan Data Umum (GDPR) Uni Eropa telah mempertanyakan praktik bisnis banyak penyedia dan organisasi periklanan daring yang mengandalkan iklan untuk pendapatan, dan memaksakan perubahan skala besar pada praktik tersebut. Namun, banyak konsumen terkejut melihat seberapa luas data mereka dijual dan dibagikan, dan

kurangnya pengawasan terhadap praktik tersebut.

Hal ini semakin dipertegas oleh skandal Cambridge Analytica, yang secara diam-diam mengumpulkan data profil pengguna Facebook, yang kemudian digunakan untuk menargetkan iklan sebagai bagian dari kampanye politik, termasuk pemilihan presiden Trump dan pemungutan suara Brexit. Saat menentukan perlunya pengumpulan data, Anda juga harus berpikir seperti penyerang. Apa yang akan terjadi jika data yang Anda kumpulkan bocor, atau perusahaan Anda diakuisisi oleh pihak lain? Alasan Anda menggunakan data tersebut sekarang mungkin dapat diterima secara etis, tetapi dalam konteks lain, mungkin secara moral meragukan.

Karena alasan ini, penting untuk tidak hanya mendapatkan persetujuan dari pengguna untuk mengumpulkan data pribadi mereka, tetapi juga persetujuan untuk memprosesnya dengan cara tertentu, dan untuk mencatat persetujuan ini secara eksplisit di samping data. Memastikan bahwa persetujuan ini memiliki cakupan yang ketat dapat melindungi Anda dari perubahan dalam misi atau kebijakan organisasi Anda, meskipun penyerang tidak akan menghormati persetujuan ini untuk pemrosesan. Anda mungkin memutuskan bahwa, dalam kasus tersebut, menyimpan data sama sekali terlalu berisiko. Misalnya, lembaga amal yang menangani pengungsi dapat memutuskan untuk membatasi data yang mereka kumpulkan jika pemerintah yang bermusuhan memanggil mereka untuk mendeportasi para pengungsi tersebut.

Bahkan ketika menganonimkan data, kehati-hatian harus dilakukan, karena masih mungkin, jika data cukup, untuk tetap menghubungkan beberapa data ke orang tertentu. Salah satu metode untuk menganonimkan data adalah dengan memberikan ID acak kepada pengguna yang tidak terkait dengan identitas mereka yang sebenarnya, dan kemudian mengaitkan data pribadi dengan itu. Metode ini lebih tepat digambarkan sebagai nama samaran, karena orang-orang masih memiliki identitas dalam sistem Anda hanya saja sekarang identitas tersebut dikaitkan dengan ID acak tersebut, bukan yang terkait dengan identitas individu mereka. Jika data cukup, mungkin saja hal ini dapat dikaitkan kembali dengan individu. Salah satu contoh terkenal dari hal ini adalah pada tahun 2006, ketika AOL merilis log pencarian pengguna mesin pencari mereka, yang dikaitkan dengan ID anonim.

Namun, melihat istilah pencarian terkadang cukup untuk menghubungkan ID anonim tersebut dengan seseorang, karena pengguna terkadang akan mencari nama mereka sendiri, atau bisnis yang secara geografis dekat dengan mereka, sekolah tempat anak-anak mereka bersekolah, dll. Sejumlah pengguna dapat diidentifikasi secara positif oleh jurnalis untuk keperluan pers. Dilema ini dapat menjadi sulit bagi organisasi besar, terutama di mana orang dapat berinteraksi dengan mereka dengan berbagai cara, karena hal ini menghasilkan lebih banyak data di seluruh organisasi yang, jika dikorelasikan, dapat mengidentifikasi pengguna. Pengembang di tim produk mungkin kesulitan untuk dengan yakin mengatakan apakah data pengguna mereka digunakan secara bertanggung jawab di seluruh organisasi, tetapi harus dapat mengajukan pertanyaan penting.

Anonimitas sejati berarti tidak ada titik data dalam sistem Anda yang terikat pada pengenal apa pun. Hal ini memungkinkan Anda untuk tetap mengembangkan agregat dan rata-

rata, tetapi bukan korelasi. Misalnya, jika Anda meneliti kota-kota tempat orang-orang mengirim barang, maka Anda mungkin dapat mengetahui bahwa orang-orang dari Slough membuat lebih banyak pesanan dari situs web Anda daripada orang-orang dari Ashford, tetapi Anda tidak akan dapat mengetahui bahwa orang-orang dari Ashford sebenarnya menghabiskan lebih banyak secara total, karena Anda tidak dapat menghubungkannya di berbagai titik data.

Harapan privasi berubah seiring waktu dan budaya. Internet awal memiliki harapan privasi yang jauh lebih rendah, dan orang-orang yang menggunakannya membiarkan lebih banyak informasi terbuka ke dunia luar daripada yang mereka lakukan saat ini. Dalam beberapa budaya, berbagi umpan foto diri Anda yang sedang bepergian secara terus-menerus dapat diterima dan didorong, sedangkan budaya lain akan terkejut dengan perilaku tersebut, karena akan mengungkapkan bahwa rumah Anda kosong, yang membuat Anda berisiko dirampok. Memahami sepenuhnya semua budaya tempat Anda beroperasi dan harapan mereka, serta konsekuensi hukumnya, merupakan suatu keharusan di dunia modern.

Bagi banyak orang, privasi diharapkan setiap saat, tetapi bagi yang lain, ada beberapa kasus di mana kurangnya privasi dapat diterima. Hubungan budaya antara negara dan warga negaranya adalah salah satu contohnya, yang disorot oleh perbedaan stereotip antara partai politik dan antara tempat-tempat seperti AS dan Eropa. Di satu sisi, banyak yang menganggap peran negara sebagai tangan yang membantu yang peduli terhadap warga negara, dan memiliki data yang baik memungkinkannya untuk melakukannya dengan lebih baik. Yang lain percaya bahwa negara harus diminimalkan, dan kebebasan individu mengalahkan segalanya, jadi negara yang mengumpulkan data dalam skala besar merupakan pelanggaran privasi.

Contohnya adalah Layanan Kesehatan Nasional (NHS) Inggris, di mana sebuah program yang dikenal sebagai "care.data," yang bertujuan untuk menghubungkan interaksi dan catatan kesehatan warga negara, diluncurkan. Aktivis privasi berkampanye menentang ini, dan banyak individu menentangnya, karena mereka melihat privasi sebagai prinsip inti yang harus ditegakkan. NHS yakin bahwa sistem ini tidak hanya akan meningkatkan perawatan individual (misalnya, catatan pasien akan tersedia di rumah sakit mana pun), tetapi juga memungkinkan data dibagikan dengan para peneliti untuk memantau tren kesehatan di seluruh negara, dan mungkin mendeteksi korelasi baru antara gaya hidup dan kondisi kesehatan yang berbeda.

Keputusan biaya-manfaat di sini sulit, tetapi pada akhirnya privasi menang dan NHS menutup skema tersebut, meskipun banyak peneliti kesehatan menyesali hasilnya. Ketegangan ini, seperti banyak ketegangan lainnya dalam etika, tidak memiliki jawaban yang jelas, tetapi memerlukan refleksi yang cermat seiring dengan pertumbuhan karier Anda. Ada baiknya meluangkan waktu untuk merenungkan contoh NHS care.data yang diberikan di atas untuk menentukan argumen mana yang menurut Anda lebih meyakinkan dan bagaimana Anda akan menanggapinya jika data pribadi Anda sendiri yang dipertaruhkan, mungkin dalam skala yang lebih kecil di dalam organisasi Anda sendiri.

5.3 BEBAN KOGNITIF

"Aplikasi Anda membuat saya gemuk" adalah klaim yang dibuat dalam posting blog terkenal oleh Kathy Sierra, yang menyoroti sebuah studi akademis yang menunjukkan bahwa orang yang harus melakukan tugas yang lebih sulit (dengan demikian, memiliki beban kognitif yang lebih tinggi) menunjukkan kemauan yang lebih rendah (misalnya, kemampuan untuk menolak makanan yang tidak sehat) daripada mereka yang melakukan tugas yang lebih sederhana. Pengamatan sederhana ini sangat penting dalam kaitannya dengan peran kita dalam mengembangkan solusi yang digunakan orang.

Kita harus bertujuan untuk mengurangi beban kognitif bagi pengguna kita guna meminimalkan konsekuensi negatif dari penggunaan aplikasi kita. Kegunaan yang baik juga bertujuan untuk mencapai hal ini, tetapi terkadang dapat bertentangan dengan tujuan bisnis. Terkadang, Anda mungkin diminta untuk mendesain antarmuka pengguna yang mengutamakan kebutuhan bisnis di atas kebutuhan pengguna. Ini terkadang disebut pola gelap, dan mencakup hal-hal seperti mempersulit penghapusan akun, atau menyembunyikan opsi untuk menolak asuransi perjalanan saat membeli tiket pesawat. Sebagai profesional, kita memiliki tanggung jawab untuk menolak fitur-fitur yang memiliki konsekuensi negatif bagi orang-orang yang menggunakan apa yang kita buat.

Demikian pula, kita juga harus memikirkan biaya manusia dari produk kita. Jika kita membuat aplikasi untuk karyawan, maka mereka mungkin akan menghabiskan sebagian besar waktu mereka untuk bekerja dengan aplikasi Anda, dan Anda dapat memberikan dampak yang besar pada kehidupan mereka. Hal ini melampaui keputusan teknis dan mencakup seluruh keputusan produk, terutama dalam kasus aplikasi "ekonomi pertunjukan", di mana kita memiliki tanggung jawab profesional untuk memperlakukan pengguna aplikasi (untuk aplikasi seperti Uber, ini adalah penumpang sekaligus pengemudi) dengan hormat, dan untuk menghindari menyesatkan mereka atau membuat hidup mereka lebih sulit. Pada akhirnya, pengguna ini tetaplah individu, dan kategori produk ini dapat memberikan dampak yang signifikan pada kehidupan mereka.

Ada seluruh kelas produk yang mungkin harus dinilai secara etis oleh pengembang. Misalnya, potensi untuk bekerja pada sistem militer mungkin memerlukan pertimbangan yang cermat tentang apakah akan menerima pekerjaan atau tidak, tetapi proyek yang mungkin tampak lebih jinak, seperti jejaring sosial, dapat digunakan untuk menyebabkan kerugian jika digunakan secara tidak benar oleh pemiliknya. Konsekuensi etisnya juga bisa tidak kentara. Misalnya, aplikasi pengenalan wajah dapat membantu pengguna jejaring sosial menandai foto dengan cepat dan akurat, tetapi jika digunakan oleh negara totaliter, aplikasi tersebut dapat digunakan untuk menganiaya warga negara.

Tidak ada jawaban yang tepat untuk pertanyaan-pertanyaan ini, tetapi ini adalah masalah yang harus kita pertimbangkan sebagai profesional, dan pertimbangkan kembali jika konteks pekerjaan kita berubah. Kita tidak hanya memiliki hak, tetapi juga tanggung jawab, untuk mengatakan "tidak" agar pekerjaan kita digunakan dengan cara yang tidak etis.

Penggunaan Energi

Desain blockchain Bitcoin secara matematis sangat brilian, dan tentu saja telah menarik

banyak perhatian, tetapi secara teknis desainnya cacat. Karena blockchain terdesentralisasi, transaksi harus disebarkan melalui blockchain agar dapat tercatat dalam catatan kebenaran. Pada saat artikel ini ditulis, setiap transaksi menggunakan kWh listrik yang sama dengan yang digunakan rumah pada umumnya dalam sehari. Pengeluaran energi, dan sumber daya fisik lainnya, tentu saja merupakan bagian yang terlihat dari dampak TI terhadap dunia, tetapi juga terasa sangat abstrak dari perspektif pengembang individu, terutama dengan perpindahan ke cloud. Ketika organisasi menjalankan pusat data mereka sendiri, penggunaan energi setidaknya terlihat oleh organisasi tersebut, sering kali sebagai biaya yang harus diminimalkan, selain dampak lingkungan apa pun.

Di cloud, sulit untuk melihat dampak ini, apalagi mempertimbangkannya, tetapi penting untuk melakukannya. Tentu saja, ada alasan teknis dan bisnis yang bagus untuk meminimalkan penggunaan energi juga; meningkatkan kinerja/efisiensi dan mengurangi biaya juga dapat menjadi pendorong besar. Mengevaluasi apakah beban dan sumber daya yang tidak perlu sedang digunakan, dan apakah kapasitasnya mencukupi.

5.4 KEPERCAYAAN

Orang-orang yang bekerja dengan kami, baik itu klien, kolega, atau pengguna, menaruh kepercayaan yang tinggi kepada kami untuk melakukan pekerjaan kami. Mereka percaya bahwa kami akan melakukan apa yang kami katakan, biasanya dengan harga yang disepakati, dan bahwa kami akan melakukannya dengan kemampuan terbaik kami. Mereka juga percaya bahwa kami tidak akan memalsukan diri kami sendiri, dan akan berperilaku jujur. Menjual diri Anda sebagai pakar AWS setelah meluncurkan satu instans EC2 pada tingkatan gratis, atau sebagai spesialis Ruby dengan pengalaman beberapa tahun padahal sebenarnya yang Anda lakukan hanyalah mengikuti tutorial Rails, akan merusak kepercayaan tersebut.

Kita tidak perlu takut untuk mengakuinya jika kita benar-benar yakin bahwa kita tidak memenuhi harapan mereka yang meminta kita melakukan sesuatu, meskipun, pada saat yang sama, melangkah keluar dari zona nyaman kita untuk mencoba sesuatu yang baru tidak apa-apa selama harapan tersebut dijelaskan dengan jelas. Kita juga harus memercayai kolega kita, dan dipercaya oleh mereka. Merendahkan, bermain politik, atau menindas sama sekali tidak dapat diterima tetapi terlalu umum dalam profesi apa pun. Ketika sebuah tim melakukan retrospeksi, arahan utama adalah prinsip yang baik untuk diikuti dalam sebagian besar interaksi dengan orang-orang: "kami berasumsi bahwa setiap orang melakukan pekerjaan terbaik yang mereka bisa dengan pengetahuan yang mereka miliki saat itu”.

Keberagaman juga menjadi faktor di sini. Teknologi tidak sepenuhnya mengingatkan kita pada adegan dari The Wolf of Wall Street, tetapi masih terlalu banyak bidang di mana "kesesuaian budaya" (yaitu, terlihat seperti orang-orang yang sudah ada di tim) dinilai lebih tinggi daripada keahlian. Ini masih terjadi secara eksplisit, tetapi bias bawah sadar bahkan lebih berbahaya. Penelitian telah menunjukkan bahwa tim yang lebih beragam berkinerja lebih baik daripada tim yang monokultur. Banyak organisasi memanfaatkan hal ini dengan mencampur fungsi dalam tim digital, tetapi merupakan tanggung jawab kita juga untuk menjadikan teknologi sebagai industri yang ramah bagi semua yang ingin berkontribusi, dan untuk

membina tim di mana semua anggota dapat berkontribusi dengan kemampuan terbaik mereka. Insinyur perangkat lunak juga memiliki gaya interaksi yang cukup unik dengan rekan-rekan kami melalui mekanisme sumber terbuka.

Sumber terbuka sering kali merupakan hasil dari waktu sukarela, dan hasil kerja ini sering kali tersedia di bawah lisensi yang cukup permisif. Beberapa di antaranya, seperti lisensi MIT atau BSD yang terkenal, sepenuhnya permisif, sedangkan yang lain, seperti GPL, lebih bernuansa, dan mengharuskan setiap perubahan atau karya yang dibangun di atas karya mereka juga tersedia dengan cara "share-alike" jika didistribusikan. Apa yang diberlakukan GPL dalam lisensinya sering kali menjadi harapan masyarakat terhadap orang-orang yang menggunakan lisensi yang lebih permisif. Misalnya, mengambil komponen sumber terbuka dan menjualnya untuk mendapatkan keuntungan adalah kejadian yang cukup langka, karena hal itu akan membahayakan seluruh sifat sumber terbuka.

Menggunakan komponen sumber terbuka sebagai pustaka dalam produk komersial sering kali dianggap dapat diterima, tetapi setiap perbaikan atau perubahan yang dilakukan pada pustaka itu sendiri diharapkan dapat disumbangkan kembali ke proyek utama, sehingga orang lain dapat memperoleh manfaat darinya. Pengembangan web mengaburkan batasan, karena kode tidak dikirimkan langsung kepada pengguna sebagai produk, tetapi tanggung jawab yang sama tetap ada: jika Anda melakukan perbaikan atau memperbaiki bug di pustaka sumber terbuka, harapan masyarakat adalah Anda akan menyumbangkannya kembali.

Bersama-sama sebagai sebuah industri, kita dapat saling meningkatkan kualitas, dan menghindari pengkhianatan atas kepercayaan yang diberikan oleh mereka yang menyediakan pustaka ini, yang kemudian mungkin beralih ke metode alternatif seperti manajemen hak digital, atau tidak menyediakan karya mereka sama sekali. Perlu diingat juga bahwa sumber terbuka sering kali dibangun atas upaya para relawan, dan bahwa kita juga harus memperlakukan pengelola dan kontributor proyek sumber terbuka dengan aturan praktis yang sama yang berlaku bagi rekan kerja kita, seperti yang dibahas sebelumnya.

5.5 KESIMPULAN

Etika merupakan bagian penting dari sebagian besar disiplin ilmu teknik, tetapi tidak cukup dibahas dalam pengembangan perangkat lunak. Bab ini membahas empat masalah etika yang mungkin Anda temui: privasi, yang semakin banyak diatur undang-undang karena kegagalan etika profesi yang dirasakan; beban kognitif, di mana kita secara tidak sengaja dapat menimbulkan efek samping negatif pada basis pengguna kita ketika kita hanya berfokus pada tujuan lain; efisiensi energi, yang terkadang tampak abstrak dalam pekerjaan yang kita lakukan, tetapi memiliki efek langsung di dunia nyata; dan kepercayaan, dalam profesi kita dan kepada mereka yang bekerja untuk kita.

Etika hidup di dunia di mana tidak ada jawaban yang benar; sebaliknya, etika dapat bersifat pribadi dan sangat subjektif, tergantung pada keadaan individu. Meskipun demikian, etika adalah sesuatu yang harus kita pertimbangkan di semua titik ketika membangun sistem untuk membangun dunia yang lebih baik bagi semua orang.

BAB 6
FRONT END

6.1 PEMAHAMAN FRONT-END DALAM PENGEMBANGAN APLIKASI WEB

Tim yang berbeda sering kali memiliki gagasan yang sangat berbeda tentang bagian mana dari aplikasi mereka yang merupakan front end. Bagi beberapa tim, front end secara harfiah hanyalah HTML, CSS, dan JavaScript yang menyusun aplikasi mereka. Bagi yang lain, ini dapat mencakup logika dan kode sisi server yang juga menghasilkan HTML, sedangkan "back end" hanyalah API. Jadi, untuk menghindari ambiguitas, bab ini mendefinisikan front end sebagai bagian dari server Anda yang menghasilkan HTML, serta kode apa pun yang berjalan di browser.

Ada tiga teknologi utama yang perlu diingat terkait kode yang berjalan di browser: HTML, CSS, dan JavaScript. JavaScript adalah topik yang sangat besar sehingga dibagi menjadi bab tersendiri, tetapi di bawah ini kami membahas HTML dan CSS, yang merupakan bahasa spesifikasi yang menentukan struktur dan tampilan halaman web Anda. JavaScript adalah bahasa pemrograman yang sepenuhnya ditentukan yang memungkinkan Anda menambahkan interaktivitas ke browser dan memanipulasi struktur dan tampilan.

Bagi pengembang back-end yang baru mengenal front end, semuanya mungkin tampak agak terlalu asing. Tiba-tiba, Anda menggunakan bahasa baru dan toolchain baru, dan Anda harus menargetkan beberapa runtime. Namun penting untuk diingat bahwa teknik back-end apa pun yang Anda ketahui dan sukai dapat diterapkan ke front end juga. Jangan panik; mulailah menyalin cuplikan JQuery dari Stack Overflow ke dalam satu file .js, karena Anda menulis kode lama dari awal. Bagi pengembang front-end, ini mungkin terasa seperti wilayah asal, tetapi ada banyak hal yang dapat dipelajari dari teknik yang mungkin secara tradisional ada di domain pengembang back-end.

6.2 HTML

Hypertext Markup Language adalah bahasa inti web. Bahasa ini tidak selalu merupakan bahasa pemrograman untuk membangun aplikasi, tetapi sebagai gantinya bahasa untuk mendeskripsikan dokumen. Ada dua pendekatan mendasar untuk mengekspresikan dokumen terstruktur dalam rekayasa perangkat lunak: markup dan standoff. Dalam bahasa markup, seperti HTML, struktur dan anotasi sejalan dengan teks dan konten dokumen. HTML memberi anotasi pada bagian teks dengan tag, yang dilambangkan dengan tanda kurung siku yang memberikan makna pada bagian yang bersangkutan, atau sebagai cara untuk menyematkan elemen nonteks di dalam halaman.

Mekanisme anotasi standoff membiarkan dokumen apa adanya, lalu memiliki berkas kedua yang menjelaskan strukturnya. Misalnya, Anda mungkin memiliki berkas teks biasa yang berisi teks yang akan dijelaskan, lalu berkas anotasi kedua yang menjelaskan hal-hal seperti "karakter antara posisi 112-118 dicetak tebal." Manfaat markup dibandingkan standoff adalah menjaga dokumen dan anotasi tetap sinkron jauh lebih mudah, karena keduanya dapat

dimanipulasi sebagai satu kesatuan, tetapi sisi negatifnya adalah jika Anda ingin memiliki teks di dokumen Anda yang kebetulan tampak seperti tag, Anda harus mengodekan (atau melepaskan) teks tersebut untuk menunjukkan bahwa teks tersebut tidak boleh diperlakukan sebagai tag.

Hiperteks adalah bentuk khusus dari teknologi yang disebut hipermedia, yang sangat menarik bagi para peneliti pada tahun 1990-an saat HTML dikembangkan. Hypermedia merujuk pada media non-linier apa pun misalnya, video yang dapat Anda navigasikan dalam urutan apa pun yang Anda inginkan, alih-alih hanya mengikuti satu jalur. Navigasi terjadi dengan mengikuti hyperlink, yang merupakan referensi ke dokumen atau bagian media lainnya. Tim Berners-Lee menganggap ini sebagai jaringan dokumen, oleh karena itu muncul istilah World Wide Web.

Buku ini tidak akan mengajarkan Anda HTML, tetapi ada banyak sumber daya bagus lainnya di luar sana, dan penting untuk mengetahui prinsip-prinsip dasar saat membuat halaman HTML. Tag HTML digunakan untuk mengekspresikan struktur dokumen, dan meskipun tag tersebut juga menyiratkan bentuk rendering tertentu di browser, struktur dan makna ini juga digunakan dengan cara lain mesin pencari, alat aksesibilitas, dan lainnya bergantung pada struktur ini yang ada dan diterapkan secara bermakna pada dokumen untuk menafsirkan halaman web dengan benar.

Dari Server ke Browser

Inti dari setiap situs web atau aplikasi adalah beberapa cara untuk membuat HTML dan mengirimkannya kepada pengguna melalui browser web. Ini dulunya dilakukan dengan menulis HTML secara manual dan menyajikannya sebagai file dari disk, dan ini masih merupakan teknik yang valid saat ini, tetapi membatasi jumlah "dinamisme" yang tersedia, karena halaman yang sama harus disajikan kepada semua orang. Meskipun JavaScript dapat digunakan untuk menambahkan beberapa interaktivitas dan dinamisme ke browser klien, sangat umum bagi HTML aktual yang dikirimkan ke browser untuk dibuat di sisi server dalam aplikasi sebagai respons terhadap permintaan, daripada sekadar menyajikan file statis dari disk.

Bahkan ketika aplikasi dinamis tidak diperlukan, akan berguna untuk membuat HTML statis pada disk dengan menyusunnya dari templat umum pada disk, menggunakan alat yang dikenal sebagai generator situs statis (salah satu contohnya adalah Jekyll). Alat-alat ini bekerja mirip dengan kode dinamis yang dihasilkan server, tetapi sebaliknya menghasilkan semua kemungkinan keluaran di muka, daripada berdasarkan permintaan. Saat membuat halaman web, Anda akan sering menghadapi berbagai masalah untuk berbagai bagian halaman yang berbeda. Misalnya, bilah navigasi atau footer umum mungkin sama di setiap halaman, tetapi komponen lain mungkin memerlukan beberapa informasi dari basis data, atau sistem lain, untuk disisipkan.

Untuk komponen statis, seperti footer, sangat umum untuk sekadar menulis fragmen HTML yang disusun bersama oleh peramban web, tetapi untuk komponen dinamis yang dapat berubah, ada banyak sekali pustaka dan teknik untuk melakukan ini. Untuk lebih memperumit masalah, konten dinamis dalam basis data mungkin juga menyimpan data gayanya sendiri,

selain bagaimana komponen yang menyajikan data tersebut diberi gaya—ini disebut sebagai konten kaya. Pendekatan untuk membangun dokumen HTML sisi server umumnya dipecah menjadi teknik berikut: pembuatan otomatis, transformasi pohon, dan interpolasi string. Ada beberapa persilangan antara ketiga teknik ini, dan ketiganya dapat digabungkan.

Pembuatan otomatis terjadi saat pustaka dasar secara otomatis membuat HTML untuk Anda, mungkin dengan langsung membuat serial kelas. Pembuatan otomatis sering kali dapat menimbulkan masalah dan sebaiknya hanya digunakan dalam dosis yang sangat kecil misalnya, jika Anda menggunakan alat seperti formulir Django, yang merender model ke formulir HTML sederhana. Dengan pembuatan otomatis, Anda sering kali memiliki kontrol terbatas atas seperti apa tampilan HTML yang dihasilkan, dan karena HTML tersebut merupakan titik integrasi antara konten, gaya, dan logika front-end Anda, tetapi juga pengguna, penting bagi Anda untuk mengendalikannya. Metodenya cepat, dan dapat tampak berguna jika Anda bekerja dalam kerangka kerja "tumpukan penuh", tetapi seperti yang dibahas dalam bab Mendesain Sistem, sering kali lebih baik memilih beberapa alat yang lebih kecil dan menyatukannya untuk membuat sesuatu yang lebih dari sekadar jumlah bagian-bagiannya daripada menggunakan solusi yang cocok untuk semua orang.

Pembuatan otomatis dapat dianggap sebagai antipola, sesuatu yang tampak bagus pada awalnya dan akan membantu Anda bergerak cepat, tetapi seiring berjalannya waktu akan menyebabkan Anda semakin kesulitan. Satu pengecualian yang perlu disebutkan adalah React milik Facebook. Meskipun React menggunakan pendekatan pembuatan otomatis, menggunakan ekstensi JSX untuk JavaScript, ada hubungan kuat antara apa yang Anda tulis dan apa yang dihasilkan yang mengurangi sebagian besar kekurangan dan menawarkan keuntungan yang cukup besar. Memang, saat menggunakan JSX, React tampaknya memiliki lebih banyak kesamaan dengan teknik interpolasi string daripada pendekatan pembuatan otomatis tradisional.

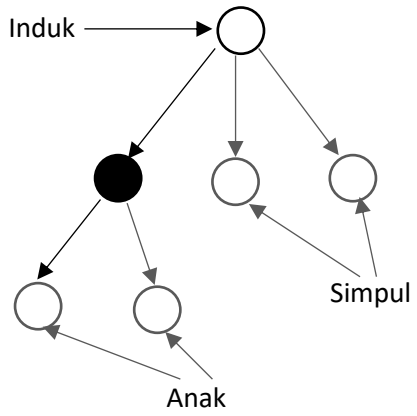
Transformasi pohon adalah teknik yang mengambil konten atau templat yang ditulis dalam bahasa non-HTML lain dan menerjemahkannya langsung ke pohon HTML, tanpa melalui tahap peralihan seperti diurai dengan kelas menjadi model. Terkadang dokumen sumber itu sendiri direpresentasikan sebagai pohon (seperti XML atau HAML), dan di lain waktu hanya berupa konten datar (seperti Markdown atau BB Code).

Apa Itu Pohon Html?

Istilah "pohon" berasal dari ilmu komputer, yang merujuk pada jenis struktur data tertentu. Pohon terdiri dari sekumpulan simpul dan sekumpulan penunjuk yang terkait dengan simpul tersebut. Akar pohon adalah simpul tunggal yang memiliki penunjuk ke simpul lain, dan penunjuk ini dikenal sebagai cabang. Akhirnya, simpul tidak memiliki penunjuk apa pun dan ini dikenal sebagai daun. Selain itu, tidak ada simpul yang dapat menunjuk ke simpul yang ditunjuk oleh simpul lain, jadi tidak ada loop, atau simpul pada beberapa cabang. Jika Anda membuat sketsa diagram ini, Anda akan mendapatkan sesuatu yang tampak seperti pohon (meskipun sering kali ilmuwan komputer memulai dengan "akar" di bagian atas, jadi Anda mendapatkan pohon yang terbalik).

Hubungan antara simpul dalam pohon dinyatakan dalam terminologi yang sama

seperti dalam leluhur manusia, seperti yang ditunjukkan pada Gambar 6.1. Simpul yang menunjuk ke simpul tertentu lainnya di pohon disebut induk, dan simpul yang ditunjuknya adalah anak. Simpul yang berbagi induk yang sama disebut sebagai saudara kandung.

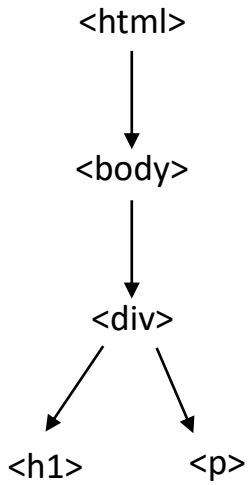


Gambar 6.1 Struktur Pohon. Dari Sudut Pandang Simpul Yang Diarsir Hitam, Induk Dan Anak Diberi Label.

HTML mengikuti struktur seperti pohon ini. Sebuah elemen HTML dapat memiliki elemen lain yang bersarang di dalamnya, yang dapat Anda sebut sebagai anak dari simpul tersebut, tetapi ini mencerminkan cabang-cabang dalam sebuah pohon. Ambil contoh HTML berikut:

```
<html>
     <body>
          <div>
               <h1>An HTML tree</h1>
               <p>Hello world</p>
          </div>
     </body>
</html>
```

Jika Anda menggambarkannya sebagai pohon, seperti pada Gambar 6.2, Anda akan mendapatkan struktur seperti ini



Gambar 6.2 Dokumen HTML Yang Terstruktur Sebagai Pohon

Pohon adalah struktur data umum, dan sintaksis lain untuk menulis HTML juga dapat direpresentasikan dengan struktur bersarang seperti ini, yang dapat diubah menjadi pohon dan kemudian ditampilkan sebagai HTML. Untuk konten dinamis yang kaya, transformasi pohon merupakan teknik yang ampuh. Sering kali tidak masuk akal untuk menyimpan konten dalam basis data sebagai HTML, karena hal ini dapat membuatnya rapuh atau terkait dengan detail implementasi tertentu, seperti URL CDN atau nama kelas CSS, yang sulit dimigrasikan.

Sebaliknya, Anda harus mempertimbangkan untuk menggunakan bahasa markup lain (seperti Markdown, BBcode, atau sintaksis wiki), yang kemudian diterjemahkan ke HTML pada waktu render. Untuk kasus sederhana, di mana sintaksis sumber sudah dikenal, ada banyak pustaka yang dapat membantu Anda melakukannya. Kelemahannya di sini adalah Anda sering kali memiliki kemampuan terbatas untuk mengaitkan atribut HTML tambahan (seperti nama kelas), yang dapat membuat penulisan pemilih CSS atau JavaScript terhadap elemen tersebut menjadi rumit.

Meskipun ada banyak pustaka yang akan melakukan penerjemahan untuk Anda, ada pustaka lain yang mengharuskan Anda untuk mengimplementasikan aturan penerjemahan sendiri. Contoh paling terkenal dari hal ini adalah XSLT, yang memiliki dukungan asli untuk menerjemahkan XML ke HTML, meskipun aturan pencocokan persis harus ditulis dalam "XSL," atau Extensible Stylesheet Language (yang merupakan XML). Meskipun XML telah mengembangkan reputasi yang buruk, umumnya karena penggunaan yang tidak tepat atau skema yang terlalu rumit, XML sangat efektif untuk tugas-tugas tertentu. XML sendiri tumbuh dari standar HTML sebagai bahasa yang lebih umum, dan bahkan ada upaya yang dibatalkan untuk menghentikan penggunaan HTML dan menggantinya dengan variasi berbasis XML yang dikenal sebagai XHTML.

XML sangat ahli dalam membuat anotasi teks sebaris, dan format lain, seperti JSON, dapat membuat prosesnya kikuk. Membuat skema XML Anda sendiri dengan tag yang memiliki makna semantik di domain Anda dan menyimpannya di basis data Anda terkadang dapat berfungsi lebih baik daripada menyimpan HTML. Dalam skenario ini, ketika konten ini secara langsung dipetakan ke keluaran HTML, XSLT harus dipertimbangkan secara serius. Daftar 6.1

menunjukkan bagaimana Anda dapat menyimpan konten sebagai XML dalam basis data, tetapi kemudian memetakannya ke beberapa keluaran HTML seperti yang ditunjukkan pada Daftar 6.2.

Daftar 6.1 Contoh XML Yang Merujuk Ke Gambar Berdasarkan ID

```
<article>
        <paragraph>This is some example content.</paragraph>
        <image id="a9be99f7" alt="Example" />
</article>
```

Daftar 6.2 Contoh Terjemahan XML Ketika Direnderer Ke HTML

```
<article>
    <p>This is some example content.</p>
    <img src="https://cdn.example.com/images/a9be99f7.jpg"
       alt="Example">
</article>
```

Anda dapat mempertimbangkan untuk menyimpan HTML secara langsung, tetapi ini dapat menjadi masalah. Misalnya, jika URL CDN Anda berubah, Anda mungkin memiliki tautan gambar yang mati di basis data Anda, atau jika Anda merujuk ke kelas CSS, akan lebih sulit untuk melakukan refaktor CSS Anda tanpa merusak konten. Ada skenario di mana konten tidak disimpan sebagai HTML, tetapi penerjemahan pohon langsung bukanlah pendekatan terbaik. Misalnya, daripada menulis sesuatu yang menerjemahkan JSON secara langsung ke HTML, akan lebih bermanfaat untuk memuatnya ke dalam model yang kemudian diteruskan ke tampilan. Kemudian, interpolasi string digunakan.

Meskipun transformasi pohon merupakan teknik yang efektif untuk menangani konten yang dinamis dan kaya, namun kurang efektif dalam hal mengelola komponen statis. Bahasa templat seperti HAML atau Jade telah menjadi umum sebagai mekanisme untuk menulis versi HTML yang diabstraksikan, yang kemudian diterjemahkan ke dalam HTML pada waktu render. Meskipun hal ini memiliki kelebihan dalam menegakkan validitas HTML yang dihasilkan, sering kali ada kekurangannya. Yang terpenting adalah hal ini menambahkan lapisan ketidakterhubungan antara kode yang Anda lihat di browser dan kode yang Anda lihat di disk, yang dapat mempersulit debugging.

Sering kali, bahasa-bahasa ini membuat penandaan sebaris (seperti memberi anotasi pada satu kata dalam kalimat) menjadi kikuk, seperti yang ditunjukkan dalam contoh berikut, selain menyembunyikan detail HTML yang penting, seperti penciutan spasi. Dalam sistem yang kompleks, sering kali perlu kembali ke HTML sebaris selain HAML, yang dapat mengacaukan kode Anda, dan gaya bahasa yang digunakan oleh teknisi yang berbeda antara dua bahasa yang berbeda yang dicampur bersama dapat menimbulkan konflik.

```
%p
  This is how you insert
  %a{:href => "foo.html"} a link
  with HAML.
```

Mengingat kelemahan ini, dan kelebihan terbatas dari penggunaan alat semacam itu, sebaiknya hindari gaya pembuatan HTML semacam ini untuk templat Anda yang sebenarnya. Jika Anda sangat khawatir tentang risiko memasukkan HTML yang tidak valid ke dalam basis kode Anda, Anda dapat memasukkan validasi HTML ke dalam pengujian otomatis Anda (dan mungkin itu adalah ide yang bagus untuk melakukannya) untuk tetap memperoleh manfaat dari lapisan transformasi ini.

Interpolasi string sejauh ini merupakan cara yang paling umum untuk membuat templat. Teknik ini bekerja dengan mengambil file HTML biasa dan menambahkan jenis tag khusus yang ditafsirkan oleh pustaka. Templat ini dirender dengan meneruskan sejumlah objek ke perender, serta nama templat, dan ini dapat dirujuk dalam tag ini—misalnya, memanggil metode pada objek untuk mengembalikan string yang digemakan ke dalam tampilan.

```
<article>
     <h2 class="title">{{ title }}</h2>
     <p class="abstract">{{ abstract }}</p>
     <a href="{{ link.href }}">{{ link.text }}</a>
</article>
```

PHP layak disebutkan secara khusus di sini, karena seluruh bahasanya didasarkan pada gagasan interpolasi string. Namun, dalam dunia MVC, mencampur logika domain dengan bahasa templating dapat membuat kode lebih sulit dipelihara karena bekerja di domain yang berbeda dalam file yang sama. Dengan demikian, banyak kerangka kerja PHP modern sebenarnya hadir dengan bahasa templating mereka sendiri, daripada menggunakan dukungan PHP untuk interpolasi ini secara langsung. Hal ini memperkenalkan batasan buatan pada seberapa banyak logika yang dapat ada dalam templat untuk menegakkan praktik baik ini.

Sebagian besar kerangka kerja web lain dalam bahasa lain juga akan hadir dengan bahasa templating mereka sendiri, seperti Ruby ERB di Rails, Python Jinja di Django dan Flask, atau Handlebars di JavaScript. Meskipun sintaksis yang tepat dapat bervariasi antara bahasa templating ini, prinsip-prinsip menyeluruhnya tetap sama. Meskipun umum untuk mendengar bahwa templat tidak boleh berisi logika apa pun, ini tidak praktis dalam sebagian besar aplikasi dunia nyata, dan sebagian besar bahasa templating menyertakan logika sederhana, seperti pernyataan if dan loop. Bahkan dengan batasan terbatas ini, mudah untuk memperkenalkan logika yang signifikan ke dalam templat, melanggar prinsip tanggung jawab tunggal dan membuat logika tersebut sulit diuji.

Ini harus difaktorkan ulang ke tempat yang lebih tepat seperti pengontrol (misalnya, menghitung nilai pernyataan if dan meneruskan hasilnya sebagai boolean ke tampilan), atau pada model atau model tampilan yang sedang ditanyakan. Fitur umum lain dari templat adalah

kemampuan untuk menyertakan templat lain. Ini adalah teknik hebat yang dapat membantu pengaturan kode dan meminimalkan penggunaan ulang.

Meskipun mungkin tidak tampak demikian, banyak teknik refaktor untuk kode logika juga berlaku di sini. Halaman yang sangat panjang dapat dipecah menjadi serangkaian templat yang lebih kecil, meskipun tidak digunakan ulang, yang dapat membuat maksud dan struktur halaman lebih mudah dilihat. Salah satu fitur penting dari beberapa bahasa templat adalah dukungan untuk pewarisan templat. Pewarisan templat agak mirip dengan pewarisan dalam kelas berorientasi objek. Templat dasar biasanya akan menentukan dokumen HTML lengkap, dengan beberapa header/footer umum, lalu akan menyertakan celah, yang dapat diisi oleh masing-masing templatdengan kontennya (biasanya badan halaman).

Ini dapat menjadi teknik yang baik untuk mengurangi duplikasi, meskipun dalam bahasa templat yang tidak mendukung ini, solusi umum adalah menyertakan file templat "header" dan "footer" di bagian atas dan bawah file untuk menyediakan tingkat fungsionalitas tersebut. Selain membuat HTML dalam aplikasi sisi server, atau sekaligus dengan generator situs statis, ada teknik yang dikenal sebagai server-side include, atau edge-side include, yang akan menyusun halaman dari komponen individual untuk dipresentasikan ke perangkat pengguna. Dalam server-side include, server yang membaca file dari disk akan menguraikannya dan secara dinamis menyertakan file lain untuk membuat komposisi baru bagi pengguna sebagai respons terhadap arahan dalam file tersebut.

Dalam edge-side include, ini dilakukan dengan menggunakan server proxy perantara. Pendekatan ini dapat menjadi ampuh jika Anda ingin memisahkan aplikasi yang menghasilkan berbagai komponen dari satu halaman web, dan juga menyertakan aturan caching yang berbeda untuk setiap komponen.

6.3 PENATAAN

HTML menentukan struktur halaman web, dan CSS (Cascading Style Sheets) menentukan penataan dan penyajiannya. Inti dari CSS adalah penyeleksi dan aturan. Penyeleksi menentukan elemen HTML mana yang akan menerapkan aturan ini, dan aturan menjelaskan nilai properti penyajian yang diterapkan ke elemen mana pun yang sesuai dengan penyeleksi.

```
.header {
  background-color: #eee;
}
```

Kode di atas adalah contoh CSS. Pemilihnya adalah `.header`, yang menunjukkan bahwa aturan dalam blok di bawah ini harus berlaku untuk objek apa pun dengan kelas header, dan harus menyetel properti background-color objek tersebut ke `#eee`, yang merupakan cara untuk mengekspresikan warna abu-abu muda. Sintaks CSS murni sangat teratur dan mengikuti format ini, dan buku ini tidak bertujuan untuk mencakup seluruh rentang properti dan nilai CSS, tetapi Mozilla Developer Network (MDN) adalah sumber yang bagus yang mencakup

seluruh rentang tersebut.

Selain kelas, pemilih CSS memungkinkan Anda menargetkan elemen berdasarkan nama tag-nya (misalnya, dengan pemilih p); elemen dengan ID (`#body` untuk elemen `<div id="body">...</div>`); atau berdasarkan atribut (`[href]` untuk semua elemen dengan atribut href). Pemilih khusus ada yang berlaku untuk semua elemen, dan pemilih atribut juga dapat menggunakan operator untuk menguji konten suatu nilai, bukan hanya keberadaan:

- `[attr="foo"]` di mana atribut sama dengan nilai foo (misalnya, `[data-languages="en"]` akan cocok dengan)
- `[attr*="foo"]` di mana atribut menyertakan substring foo (misalnya, `[data-languages*="en"]` akan cocok dengan `<span data-languages="german english">`)
- `[attr^="foo"]` di mana atribut dimulai dengan substring foo (misalnya, `[href^="https"]` akan cocok dengan tautan aman apa pun, tetapi bukan tautan yang mungkin memiliki https di tempat lain di jalur tersebut)
- `[attr$="foo"]` di mana atribut diakhiri dengan substring foo (misalnya,
- `[href$=".pdf"]` akan cocok dengan tautan apa pun ke file PDF)
- `[attr~="foo"]` jika atribut menyertakan seluruh kata foo (misalnya, `[data-languages~="en"]` akan cocok dengan tetapi tidak `<span data-languages="english">`, karena pada yang terakhir en bukan seluruh kata)
- `[attr|="foo"]` jika atributnya persis foo, atau foo diikuti tanda hubung dan teks lainnya (misalnya, `[data-language|="en"]` akan cocok dengan `<span data-language="en-GB">`)

Operator atribut apa pun juga dapat digunakan tanpa memperhatikan huruf besar/kecil dengan menambahkan i sebelum tanda kurung terakhir (misalnya, `[data-languages*="en" i]` akan cocok dengan `<span data- languages="English">`). Pemilih juga dapat disarangkan—misalnya `.body p` menyatakan berlaku untuk semua tag <p> yang muncul di bawah elemen yang memiliki kelas body, dan digabungkan, misalnya, p.author berlaku untuk tag <p> yang menentukan kelas author. Tag HTML dapat membawa beberapa kelas—misalnya, p.author akan tetap berlaku untuk `<p class="author large">`, seperti halnya p.large dan p.author.large. Penyatuan dan kombinasi dapat digunakan Bersama misalnya, `.body p.author` untuk semua `tag <p>` dengan kelas author yang memiliki leluhur di DOM di suatu tempat dengan kelas body.

Metode penyatuan juga dapat dibuat lebih spesifik. Dalam pemilih `.body p` sebelumnya, dapat ada banyak elemen yang disarangkan di antara elemen yang memiliki kelas body dan p, tetapi dengan menggunakan operator > kita dapat mengatakan `.body > p` yang berarti hanya tag p yang secara langsung merupakan anak dari tag dengan kelas body yang menjadi target. Operator lain menargetkan saudara kandung dalam pohon DOM. Ambil, misalnya, HTML berikut:

```
<div class="article">
     <p>some introductory text</p>
     <img class="figure" ... >
     <p>Some more text</p>
     <h3>A sub section</h3>
     <p>Line 1</p>
     <p>Line 2</p>
</div>
<p>Continuing text</p>
```

Jika kita memiliki selector di CSS kita, yaitu `.article img + p`, maka p yang langsung mengikuti img akan ditata sesuai dengan properti yang ditentukan dalam definisi tersebut. `<p>` lainnya tidak, berbeda dengan `.article h3 ~ p`. Tanda ~ di sini berarti menata semua saudara kandung berikutnya, jadi dua baris yang muncul setelah judul h3 akan mengambil gaya tersebut, tetapi tidak baris yang muncul setelah div, karena naik satu tingkat penyusunan berarti mereka tidak lagi menjadi saudara kandung (simpul DOM induk berbeda). CSS juga memungkinkan kita untuk menargetkan "pseudo-class," yang sesuai dengan bagaimana suatu status dapat berubah. Misalnya, `a:hover` akan berlaku saat suatu elemen sedang diarahkan kursor (misalnya, mengubah gaya tombol).

Ada banyak pseudo-class, dan sumber daring seperti MDN mencakup daftar lengkapnya. Demikian pula, ada pseudo-elemen, yang paling terkenal adalah `::after` dan `::before` (untuk alasan kompatibilitas, ini dapat ditentukan dengan satu titik dua, seperti pseudo-class). Elemen semu ini merujuk pada elemen yang tidak ada di DOM, tetapi dapat dibuat berperilaku seperti elemen tersebut—misalnya, `::before` dapat digabungkan dengan background-image untuk membuat ikon muncul sebaris yang murni didefinisikan dalam CSS. Deskripsi terperinci tentang hal ini berada di luar cakupan buku ini, tetapi ada banyak materi referensi yang bagus tentang cara menggunakannya.

"C" dari CSS-lah yang menyebabkan masalah. Beberapa lembar gaya dapat diterapkan ke dokumen, dan jika ada beberapa penyeleksi yang menargetkan contoh spesifik dari tag HTML pada halaman, maka aspek cascading CSS akan muncul dan serangkaian pembobotan digunakan untuk menentukan properti mana yang harus "menang" dan menjadi properti yang benar-benar diterapkan. Semua browser memiliki lembar gaya default, yang menyebabkan halaman setidaknya memiliki beberapa gaya (font default, ukuran berbeda untuk judul, dll.), daripada hanya merender layar kosong saat dokumen dimuat, tetapi ini akan selalu kalah dengan lembar gaya apa pun yang ditentukan oleh situs itu sendiri.

Tujuan awalnya adalah agar pengguna dapat menentukan stylesheet mereka sendiri yang akan menggantikan hal lain (misalnya, untuk menambah ukuran font bagi mereka yang memiliki gangguan penglihatan), tetapi ini tidak pernah digunakan dengan baik dalam praktik dan akhirnya menjadi terlalu rumit. Salah satu masalah utama dengan CSS adalah bahwa hal itu dapat mempersulit penentuan ruang lingkup, sehingga mudah untuk menambahkan aturan yang memiliki efek samping yang tidak diinginkan pada komponen lain, atau interaksi yang aneh. Jenis bug ini umumnya disebut sebagai bug spesifisitas, dan ada sejumlah teknik untuk

menanganinya dan meminimalkan risiko memperkenalkannya sejak awal.

Bagian lain yang membingungkan dari CSS adalah bahwa beberapa properti tidak hanya berlaku untuk elemen yang ditentukan oleh pemilih, tetapi juga elemen apa pun yang bersarang di dalamnya. Ini dapat berguna untuk menghindari duplikasi dalam jumlah besar misalnya, jika Anda memiliki judul dan paragraf di dalam div, maka Anda dapat mengatur warna font pada div dan itu akan berlaku untuk judul dan paragraf, daripada menyebabkan terlalu banyak duplikasi, tetapi ini terkadang dapat menyebabkan efek samping yang tidak terduga. CSS juga tidak memiliki kemampuan untuk mengekspresikan konsep tertentu secara langsung dengan cara yang lugas contoh paling terkenal dari hal ini adalah penyelarasan vertikal serangkaian blok (meskipun ini sudah sedikit membaik).

Konsep-konsep ini harus diekspresikan dengan membangun efek yang diinginkan dari serangkaian properti yang lebih mendasar, yang dapat mengarah pada pola kode umum (terkadang disebut "peretasan CSS") yang dapat menyulitkan untuk melihat secara pasti apa efek langsung dari setiap properti, karena efek gabunganlah yang diinginkan. Keterbatasan mendasar lainnya adalah ketidakmampuan untuk menulis pemilih dalam bentuk yang mengatakan, "berlaku untuk elemen ini, tetapi hanya jika elemen ini memiliki simpul anak yang cocok dengan pemilih ini." Artinya, Anda tidak dapat menulis pemilih seperti `.product:has-child(.price)`, yang akan memberi gaya pada elemen dengan kelas product jika memiliki kelas anak price.

Hal ini disebabkan oleh penerapan CSS secara top-down. Jika Anda dapat menata induk berdasarkan turunannya, maka perender harus melihat ke depan untuk melihat elemen mana yang muncul dalam render, yang memiliki overhead kinerja yang signifikan, atau kembali dan merender ulang induk saat turunannya sedang dirender. Solusi yang paling umum untuk ini adalah menambahkan kelas seperti `.product--has-price` dalam kode sisi server dan menatanya, daripada mencoba melakukannya dalam CSS murni. Pembatasan ini menjadi sumber frustrasi bagi banyak pengembang, tetapi kesederhanaan CSS berarti tidak banyak yang perlu dipelajari, meskipun beberapa di antaranya tidak jelas. Kesederhanaan ini telah diberlakukan secara de facto, karena sulit untuk membuat perubahan signifikan pada bahasa dengan cara yang kompatibel dengan versi lama.

Properti dan nilai baru dapat ditambahkan, karena browser akan mengabaikan properti dan nilai yang tidak dikenali, tetapi perubahan yang lebih signifikan pada sintaks bahasa lebih sulit untuk diperkenalkan. Karena kompatibilitas versi lama (mendukung browser lama) sering kali menjadi perhatian penting dalam pengembangan web, penulisan CSS sering kali memerlukan penulisan CSS "penyebut umum terendah", yang tidak memiliki kehalusan pengembang, seperti variabel dan campuran. Untuk mengatasi hal ini, konsep praprosesor CSS diperkenalkan. Ini adalah transpiler yang menerima beberapa bahasa lain dan menghasilkan CSS yang valid. Transpiler umum adalah Less dan Sass, yang semuanya mendukung penyeleksi dan aturan yang ditentukan seperti CSS normal, tetapi memiliki sintaks yang berbeda untuk deklarasi dan menambahkan fitur tambahan ke bahasa tersebut di atas CSS dasar.

SCSS adalah variasi Sass yang memungkinkan Anda menulis Sass, tetapi dalam sintaksis yang lebih dekat dengan CSS normal daripada Sass atau Less normal. Karena alasan ini, SCSS

menjadi sangat populer, dan ketika kebanyakan orang merujuk ke Sass, mereka secara khusus mengacu pada variasi SCSS. Salah satu alasan SCSS populer adalah karena CSS biasa juga merupakan SCSS yang valid, sehingga menawarkan rute pemutakhiran yang mudah bagi orang yang beralih dari front end tradisional yang hanya menggunakan CSS, dan juga familier bagi para pengembang.

Fitur awal Sass dan praprosesor lainnya yang hebat adalah kemampuan untuk mendukung variabel agar dapat digunakan kembali, meskipun CSS sendiri kini telah mengadopsi variabel. Misalnya, Anda dapat memiliki satu berkas yang menentukan palet warna Anda, lalu menggunakan kembali variabel tersebut untuk membuat beberapa tema. Daftar 6-3, 6-4, dan 6-5 menentukan 3 berkas SCSS, dan ketika dirender dengan Sass, ini akan menghasilkan dua berkas CSS, yang satu menyatakan putih di atas hitam, dan yang lainnya hitam di atas putih.

Daftar 6.3 Dark.Scss

```
$background-color: #000;
$text-color: #fff;

@import "main";
```

Daftar 6.4 Light.Scss

```
$background-color: #fff;
$text-color: #000;

@import "main";
```

Daftar 6.5 _Main.Scss

```
body {
  background-color: $background-color;
  color: $text-color;
}
```

@import adalah fitur lain yang berguna dari berkas SCSS, yang memungkinkan beberapa berkas digabungkan menjadi satu. Garis bawah dalam _main.scss menunjukkan bahwa berkas tersebut merupakan berkas parsial, sehingga hanya ada untuk disertakan oleh berkas SCSS tingkat atas, dan tidak akan dirender ke dalam berkas CSS. Fitur lain yang berguna termasuk fungsi, mixin, dan ekstensi. Misalnya, Anda dapat memiliki kelas dasar yang didefinisikan dalam SCSS sebagai:

```
%button {
  ...
}
.add-button {
```

```
    @extends %button;
    background-image: url('add.png');
  }
```

Ini menciptakan kelas yang disebut %button, tetapi tidak seperti kelas CSS normal, kelas ini tidak dapat digunakan secara langsung dalam HTML. Sebaliknya, kelas ini harus diperluas oleh selector lain, dan apa pun yang didefinisikan dalam %button juga tersedia dalam .add-button dan apa pun yang memperluasnya. Di sisi lain, Anda juga dapat menggunakan include. Keduanya mungkin tampak serupa secara fungsional, tetapi dapat mengalami masalah kinerja.

Kelas yang diperluas berkali-kali dapat memiliki selector yang sangat panjang saat direnderke CSS, yang dapat memengaruhi kinerja, sedangkan include hanya diulang setiap kali muncul di CSS final, yang dapat meningkatkan ukuran. Mana yang akan digunakan dalam keadaan apa pun sulit untuk dinilai, dan dapat memerlukan pembandingan untuk mengetahui dampak kinerja yang sebenarnya. Include dalam SCSS terlihat seperti ini:

```
  @mixin button() {
    ...
  }
  .add-button {
    @include button;
    background-image: url('add.png');
}
```

Mixin dan include juga memungkinkan perilaku seperti fungsi, karena keduanya mengambil parameter. Misalnya, Anda dapat menulis ulang contoh sebelumnya sebagai berikut:

```
  @mixin button($icon-url) {
    background-image: url($icon-url);
      ...
  }

  .add-button {
    @include button('add.png');
}
```

Parameter juga dapat mengambil parameter default, yang membuatnya sangat efektif. Untuk mengganti atribut yang diwarisi melalui @extends, Anda dapat menentukan atribut di selector yang memperluasnya, tetapi ini juga memperumit CSS akhir yang direnderdengan ekstensi dan penggantian yang direnderberdampingan. PostCSS memiliki pendekatan serupa, yang dimulai dengan CSS biasa, tetapi mendukung plugin yang menambahkan fitur tambahan ke CSS. Beberapa plugin ini mengaktifkan fitur seperti SCSS, sedangkan yang lain hanya mengaktifkan fitur bahasa CSS baru yang tidak kompatibel dengan versi lama, sehingga Anda dapat menulis CSS murni, tetapi kemudian mengeluarkannya dengan cara yang kompatibel

dengan versi lama.

Ada alternatif untuk menggunakan fitur seperti SCSS extends dan include: menggunakan CSS murni lalu mencampur kelas CSS di DOM. Untuk melakukannya diperlukan beberapa tingkat disiplin, dan teknik yang dikenal sebagai BEM adalah salah satu metode untuk mencapainya. BEM adalah singkatan dari Block Element Modifier, dan mungkin hanya dilihat sebagai skema penamaan, tetapi lebih dari itu. Dalam BEM, komponen tingkat tinggi disebut blok, dan sesuai dengan komponen yang dapat digunakan kembali atau berbeda dalam struktur halaman Anda. Blok biasanya terdiri dari elemen HTML lainnya, yang mungkin terlihat seperti ini:

```
<div class="product-card">
  <h3 class="product-card_name"> ...< /h3>
  <img class="product-card_photo" ... >
  <p class="product-card_description"> ...< /p>
  <p class="product-card_price product-card_price--discounted">...< /p>
  <a href="product-card_link button button -- primary"> ...< /a>
</div>
```

Dalam contoh sebelumnya, kita memiliki blok, `product-card`, lalu sejumlah elemen di dalamnya, name, photo, description, dan link. Link itu sendiri adalah blok bertipe button, dan tombol itu dimodifikasi dengan menjadi tombol utama. Nama-nama kelas ini menunjukkan skema penamaan BEM, yaitu "`block__element--modifier`," di mana element dan modifier bersifat opsional. Definisi CSS untuk contoh ini kemudian akan mendefinisikan product-card sebagai komponen tingkat tinggi dan setiap elemen di dalamnya juga, meskipun ada kontrak implisit yang menyatakan bahwa elemen tersebut hanya boleh digunakan di dalam blok itu, yang dapat menyederhanakan definisi Anda.

Terkadang Anda mungkin perlu memvariasikan perilaku blok, dan di sinilah pengubah berperan. Daripada memiliki beberapa kelas, seperti tombol-utama, tombol-sekunder, dll., untuk variasi pada tema, yang dapat mengakibatkan duplikasi (atau menggunakan SCSS extends/includes, yang mengakibatkan duplikasi dalam CSS yang dirender), Anda dapat mengabstraksikan fungsionalitas umum ke level blok (atau elemen) dan kemudian hanya menentukan perbedaan variasi spesifik tersebut dari basis. Hal ini ditunjukkan di atas dengan memodifikasi tombol di level blok (dalam hal ini elemen dapat mewarisi modifikasi tersebut dengan menentukan pemilih bersarang di CSS), atau di level elemen itu sendiri, seperti pada elemen harga.

Meskipun BEM dapat digunakan sendiri, BEM juga umum digunakan dengan SCSS dan alat serupa lainnya, karena merupakan cara untuk menangani masalah spesifisitas CSS dengan menghindari penumpukan yang berlebihan, dan menangani masalah namespace dengan memastikan elemen merupakan namespace dalam blok. Ada beberapa pendekatan lain untuk menangani namespace datar CSS, yang paling umum adalah dengan menambahkan namespace di awal nama. Misalnya, jika Anda membuat modul bersama yang dapat disematkan di halaman lain, seperti tombol "Bagikan", Anda dapat memulai semua kelas

dengan nama alat dan tanda hubung untuk menghindari bentrok dengan nama apa pun yang mungkin digunakan situs yang memuat sematan Anda.

Teknologi lain telah muncul yang secara dinamis mengikat CSS ke node DOM, baik dalam JavaScript atau HTML, dengan mengetahui kelas CSS mana yang digunakan oleh DOM dan kemudian menulis ulang nama tersebut untuk menyertakan pengenal unik. Alat yang berguna untuk dimiliki saat menulis CSS adalah "autoprefixer." Saat properti atau nilai CSS baru diperkenalkan, sering kali ada periode saat beberapa browser mendukung versi eksperimentalnya, yang mungkin memiliki sedikit perbedaan dari versi standar akhir. Untuk menghindari pengenalan sintaksis yang rusak untuk fitur eksperimental ini, produsen browser akan memperkenalkan versi "awalan" dari aturan nilai. Misalnya, saat metode untuk memperkenalkan sudut membulat pada kotak dibuat, Anda dapat mendukungnya di browser WebKit dengan menentukan -webkit-border-radius: 5px, hingga spesifikasi standar akhir disetujui.

Dengan autoprefixer, Anda cukup menulis versi standar, dan itu akan menghasilkan versi awalan juga untuk mendukung browser lama yang mengizinkan versi eksperimental tetapi tidak mengizinkan versi standar. Saat melakukan pekerjaan front-end dalam CSS, kemungkinan besar Anda akan menemukan kerangka kerja CSS. Saat menggunakan alat bantu seperti NPM dengan SCSS, alat bantu ini berfungsi sebagai pustaka dan utilitas yang dapat Anda impor ke dalam kode Anda, tetapi juga merupakan file CSS mandiri yang dapat Anda sematkan, yang menyediakan banyak kelas untuk Anda gunakan dalam HTML Anda. Kerangka kerja CSS dapat berkisar dalam cakupannya dari yang relatif sederhana seperti normalize.css, yang mencoba untuk "mengatur ulang" CSS ke serangkaian default yang disederhanakan yang sama di semua browser hingga yang lebih besar, yang menentukan pembantu untuk membangun tata letak kisi dan aturan tipografi, hingga yang berfitur lengkap seperti Foundation dan Bootstrap, yang juga menyertakan gaya untuk banyak elemen umum seperti formulir, menu, dll.

Banyak organisasi akan mengembangkan kerangka kerja CSS internal yang mematuhi gaya internal mereka dan menyertakan komponen dan gaya yang digunakan kembali di banyak halaman yang berbeda. CSS disebut Cascading Style Sheets karena beberapa aturan dapat diterapkan ke elemen DOM tertentu, dan proses mencari tahu nilai pasti properti yang akan diterapkan ke elemen DOM dikenal sebagai cascade. Terkadang, akan terjadi konflik saat properti yang sama didefinisikan beberapa kali, jadi cascade harus mencari tahu aturan paling spesifik yang akan diterapkan, dan ada sejumlah batasan dalam melakukan ini. Yang pertama berkaitan dengan ancestry di DOM. Jika Anda mendefinisikan font-family di selector HTML, lalu menggantinya di selector p, maka teks apa pun di dalam tag <p> akan mengambil font yang didefinisikan di selector p, karena itu lebih dekat dengannya di struktur pohon HTML.

Perhatikan bahwa tidak semua properti CSS diwarisi oleh node anak; banyak yang hanya berlaku untuk node yang menjadi target. Jika ada beberapa penyeleksi yang berlaku untuk node yang sama, dan node tersebut adalah node yang sama persis (atau pada tingkat keturunan yang sama, untuk properti yang diwarisi dari induk), maka kita harus melihat penyeleksi aktual yang mendefinisikan properti tersebut dan menghitung nilai spesifisitas.

Untuk menghitung spesifisitas aktual penyeleksi, pertama-tama Anda menghitung jumlah ID-nya, kemudian jumlah kelas, atribut, dan pseudo-kelas, dan terakhir jumlah elemen dan pseudo-elemen. Anda kemudian membandingkan, dalam urutan itu, jumlah setiap jenis, dan jumlah pertama yang lebih tinggi adalah yang paling spesifik.

Misalnya, penyeleksi dengan dua ID selalu lebih spesifik daripada penyeleksi dengan satu ID, bahkan jika yang terakhir memiliki lebih banyak kelas yang ditentukan, karena ID selalu lebih spesifik daripada kelas. Dalam contoh lain, jika tidak ada penyeleksi yang menentukan ID, atau jika jumlah ID dalam penyeleksi sama, maka kelas memang penting, dan penyeleksi dengan kelas terbanyak adalah yang paling spesifik. Selanjutnya, jika jumlah ID dan kelas dalam selector sama, maka hal tersebut diputuskan oleh elemen. Aturan terakhir untuk menentukan mana yang akan diterapkan, jika semua hal lain sama, adalah yang mana yang muncul terakhir dalam berkas definisi CSS. Jadi, jika Anda memiliki dua selector p, selector yang muncul kedua akan menggantikan nilai apa pun yang ditetapkan dalam selector pertama.

Ada kalkulator spesifisitas yang dapat digunakan untuk menentukan nilai spesifisitas selector, yang dapat membantu debugging. Namun, menghindari penggunaan aturan spesifisitas yang rumit lebih baik, karena dapat menjadi sangat rumit dengan cepat, jadi penting untuk memilih selector dengan hati-hati, memastikan bahwa selector tersebut memiliki spesifisitas terendah yang dibutuhkan untuk menyelesaikan tugas. BEM menganjurkan penggunaan CSS yang sangat datar, di mana sebagian besar selector adalah satu kelas (mungkin dengan elemen semu) untuk mengelola spesifisitas dengan cara ini. Hal ini dapat menyebabkan HTML menjadi bertele-tele, karena suatu elemen mungkin memerlukan banyak selector CSS jika menggabungkan kelas yang berbeda, tetapi pendukung BEM percaya bahwa ini adalah pilihan yang tepat.

Penggunaan ID juga umumnya tidak dianjurkan dalam CSS. Ada aturan kasus khusus dalam CSS yang dikenal sebagai `!important`, misalnya, yang menetapkan aturan seperti `font-weight:bold !important`. Tanda ! mungkin tampak seperti negasi, tetapi ini adalah bentuk deklarasi bahwa aturan ini adalah yang paling penting dan harus lebih spesifik daripada deklarasi lainnya. Penggunaan `!important` dianggap sebagai bentuk yang buruk, karena dapat menyebabkan ketidakfleksibelan saat mencoba mengerjakan basis kode yang sama di kemudian hari, dan biasanya menunjukkan bahwa ada beberapa aturan spesifisitas lain yang dikelola dengan buruk yang perlu ditangani, karena Anda mungkin memiliki basis kode yang cukup kusut.

Salah satu dari beberapa skenario yang perlu dipertimbangkan untuk menggunakan `!important` adalah saat Anda perlu mengganti stylesheet asing, tetapi dalam semua kasus lainnya, Anda harus terus menggunakan aturan spesifisitas normal sebagai gantinya. `!important` sendiri terikat oleh aturan spesifisitas; jika Anda memiliki dua aturan yang !important yang berlaku untuk elemen DOM, proses yang sama seperti di atas diterapkan, dan !important yang paling spesifik akan diterapkan. `!important` tidak memungkinkan Anda untuk mengabaikan kekhususan ia hanya membuat jenis kekhususan baru untuk dipertimbangkan. Kasus khusus terakhir yang perlu dipertimbangkan adalah gaya sebaris, misalnya, `<span style="..."></span>`. Dalam kasus ini, properti yang didefinisikan

sebaris selalu yang paling spesifik.

Komponen

Bab UX membahas pendekatan untuk mendesain pengalaman front-end, dan salah satu pendekatan yang paling umum adalah memecah satu halaman web menjadi serangkaian komponen yang dapat digunakan kembali, yang kemudian dapat muncul kembali di halaman lain. Terkadang penggunaan kembali ini juga dapat menjangkau beberapa situs web di seluruh organisasi yang sama. Hal ini masuk akal bagi seseorang yang memiliki latar belakang pengembangan, yang mungkin mencoba menyusun kode sebagai modul, kelas, dan fungsi yang dapat digunakan kembali, dan ini adalah sesuatu yang dapat kita lakukan saat membangun komponen UI juga.

Tantangan dengan komponen UI di Web adalah sulit untuk mendefinisikannya sebagai satu paket mandiri, tidak seperti di area lain seperti pengembangan seluler, tempat Anda dapat menautkan pustaka dan mengimpor kelas. Ambil contoh, situs web e-commerce, yang mungkin ingin menampilkan gambar mini item di halaman hasil pencarian dan di kotak "item serupa" di halaman hasil produk. Paling tidak, Anda perlu memasukkan beberapa HTML umum ke kedua halaman, dan ini dapat dicapai dengan menggunakan perintah "include" atau yang serupa dalam bahasa templating (di mana templat dapat diparameterisasi), tetapi Anda hampir selalu ingin menyertakan beberapa gaya dan kemungkinan JavaScript yang mendukung.

Anda dapat menyertakan skrip dan CSS sebaris dengan templat HTML, tetapi ini menyebabkan penurunan kinerja, karena dapat mengakibatkan banyak duplikasi jika templat digunakan beberapa kali, memperlambat kinerja, dan gaya CSS harus dideklarasikan di <head> halaman agar sesuai dengan spesifikasi. Beberapa kerangka kerja, seperti WordPress, memungkinkan templat ini untuk mendeklarasikan gaya dan JavaScript tambahan apa pun untuk dimuat, yang kemudian disertakan dengan benar di halaman yang dirender dan dideduplikasi, tetapi ini juga dapat menyebabkan masalah kinerja, terutama pada koneksi yang kurang dapat diandalkan seperti seluler, karena browser sekarang memiliki banyak file CSS atau JS kecil untuk diambil.

Cara yang paling umum untuk mengatasi masalah performa ini adalah dengan menggabungkan gaya dan skrip apa pun untuk semua komponen yang digunakan di situs ke dalam satu berkas untuk CSS dan JavaScript untuk seluruh situs, lalu menyajikannya kepada pengguna. Cara ini berfungsi dengan baik untuk situs bergaya kecil hingga sedang, karena meskipun kunjungan pertama ke situs mungkin memerlukan pengunduhan berkas skrip yang berisi JavaScript atau CSS yang tidak digunakan, berkas tersebut kemudian dapat di-cache dan digunakan kembali saat pengguna menavigasi situs. Ada sedikit beban tambahan pada setiap halaman, tetapi pendekatan ini biasanya menawarkan keseimbangan terbaik antara performa dan beban tambahan, selama berkas tunggal tidak menjadi terlalu besar.

Pendekatan ini mungkin tampak melanggar prinsip "jangan ulangi diri Anda sendiri", karena komponen mungkin perlu ditentukan di beberapa tempat: di HTML tempat komponen tersebut digunakan, di berkas CSS yang akan diimpor, dan di berkas JavaScript Anda. Namun, ini biasanya merupakan hal yang paling tidak rumit untuk dilakukan. Saat membuat komponen

bersama, ada dua cara utama untuk benar-benar menyusun kode, pilihan Anda dapat dipengaruhi oleh kerangka kerja yang Anda gunakan atau gaya yang disukai komunitas. Yang pertama adalah memiliki folder per jenis sumber daya, lalu file untuk setiap bagian komponen Anda. Misalnya:

- `templates/thumbnail.html.j2`
- `assets/styles/_thumbnail.scss`
- `assets/scripts/thumbnail.js`

Cara lainnya adalah mengelompokkannya bersama-sama ke dalam sebuah direktori, misalnya:

- `components/thumbnail/template.html.j2`
- `components/thumbnail/_component.scss`
- `components/thumbnail/component.js`

Terkadang mungkin ada aset lain yang perlu dipertimbangkan. Misalnya, Anda mungkin ingin membagi komponen yang sangat besar menjadi sejumlah file JavaScript atau komponen CSS, atau sub-template, tetapi Anda mungkin juga perlu menyertakan aset gambar apa pun, seperti SVG atau font khusus. Metode dasar untuk melakukan ini tetap sama, tetapi detail yang tepat akan bergantung pada alat pembuatan yang Anda gunakan. Beberapa akan secara otomatis menyalinnya ke tempat yang tepat atau mengubahnya menjadi sebaris saat disertakan, tetapi yang lain mungkin memerlukan langkah tambahan.

Kedua pendekatan di atas mengasumsikan bahwa seluruh situs Anda berada dalam satu basis kode, tetapi untuk situs yang lebih besar, atau ketika ada komponen yang dibagikan, Anda sering kali ingin membuatnya tersedia sebagai paket yang dapat diinstal menggunakan pengelola paket Anda. NPM (dan alternatif yang kompatibel, seperti Yarn) telah menjadi ekosistem terbesar untuk berbagi komponen front-end (serta untuk kode JavaScript back-end saat menggunakan Node). NPM dibahas lebih lanjut dalam bab JavaScript, tetapi merupakan repositori paket yang dipublikasikan dan alat baris perintah untuk menginstalnya dari definisi paket. NPM mendukung repositori pribadi, yang dapat menjadi cara yang efektif untuk berbagi kode Anda, baik melalui penautan langsung ke repo Git, menggunakan layanan hosting yang mereka sediakan, atau menghosting sendiri repositori Anda.

Banyak alat build JavaScript terintegrasi langsung dengan modul JavaScript yang diinstal NPM, tetapi alat lain seperti CSS atau pemuat templat mungkin mengharuskan Anda untuk menentukan jalur lengkap ke file yang akan disertakan atau menyesuaikan pengaturan untuk mencari di direktori yang tepat. Dengan menggunakan NPM, Anda dapat membagi komponen tertentu ke folder dan repositorinya sendiri, menerbitkannya ke tempat di mana proyek lain bergantung padanya.

Satu kekurangannya di sini adalah jika Anda perlu selalu memiliki versi komponen yang sama persis yang disebarkan di seluruh situs Anda, itu bisa jadi sulit karena Anda perlu memperbarui versi di setiap tempat yang digunakan saat dependensi diubah, lalu menyebarkan perubahan itu. Ini adalah kelemahan mendasar di setiap komponen yang akhirnya dibundel dalam situs web tertentu, di mana Anda tidak dapat memperbarui komponen itu secara terpisah. Jika kasus penggunaan ini penting bagi Anda, sebaiknya hindari membundelnya sama sekali, dan masukkan langsung ke halaman melalui tag `<script>` dan

`<link>`, atau menggunakan pendekatan `<iframe>`, yang dibahas di bawah ini.

Seiring dengan semakin besarnya situs, beban yang harus ditanggung untuk memiliki satu bundel untuk seluruh situs bisa sangat besar, dan teknik yang paling umum untuk mengatasi hal ini disebut "pemisahan kode". Pemecahan kode merupakan fitur dari alat pembuat kode Anda, dan saat digunakan, alat ini akan membuat beberapa bundel kode (CSS atau JavaScript) alih-alih satu. Anda kemudian dapat memilih untuk hanya menyertakan bundel yang relevan pada sekumpulan halaman tertentu, sering kali memilih bundel tingkat dasar untuk komponen umum di semua halaman (seperti navigasi), lalu bundel untuk setiap fungsionalitas khusus halaman yang dimuat.

Pendekatan naif terhadap metode ini mungkin membuat dua bundel yang berbeda, tetapi sering kali beberapa bundel akan memiliki beberapa pustaka atau gaya bersama, yang akan mengakibatkan pustaka atau komponen dibundel beberapa kali—satu di setiap bundel. Beberapa alat pembuat kode cukup cerdas untuk mengidentifikasi skenario ini dan secara otomatis akan membuat bundel lain yang berisi fungsionalitas umum. Menerapkan pemisahan kode mengharuskan Anda untuk menentukan bundel atau cara pemisahannya, dan ini bisa menjadi rumit untuk dipertahankan, jadi sebaiknya Anda hanya memperkenalkan ini jika tolok ukur kinerja benar-benar mengidentifikasi masalah signifikan dengan pendekatan bundel tunggal.

Untuk aplikasi web satu halaman, ada pendekatan lain yang dapat menyederhanakan pengalaman pengembang. Dalam aplikasi web satu halaman, templating HTML sering dilakukan di klien dengan JavaScript, menyatukan templating dan skrip ke satu tempat dan menjadi lebih dekat dengan model yang digunakan dalam sebagian besar pengembangan non-web. Pendekatan lebih lanjut, yang diklasifikasikan sebagai "CSS-in-JS," juga menghadirkan gaya ke basis kode JavaScript. Pada saat penulisan, pendekatan ini cukup belum matang, dan dapat menyebabkan masalah kinerja atau memerlukan pengikatan kode Anda secara ketat ke alat pembuatan tertentu, sehingga menambahkan ketergantungan substansial baru.

Salah satu pendekatan untuk CSS-in-JS adalah agar gaya dikelola saat dijalankan oleh JavaScript, yang dapat memiliki implikasi kinerja yang sangat negatif, terutama untuk desain responsif. Pendekatan alternatif adalah meminta alat pembuatan melihat tempat gaya telah didefinisikan dalam JavaScript dan mengekstraknya ke dalam file CSS, yang kemudian berperilaku seperti bundel tradisional di browser. Pendekatan terakhir sangat menjanjikan, meskipun alat untuk melakukannya saat ini belum matang dan adopsi (pada saat penulisan) masih terbatas. Dengan pendekatan ekstraksi CSS, Anda sering mendapatkan manfaat tambahan, seperti menulis ulang nama kelas untuk menghindari konflik jika dua kelas berbeda memiliki nama yang sama (bekerja di sekitar namespace global CSS).

Pendekatan yang sangat berbeda untuk berbagi komponen antar halaman web yang berbeda adalah dengan menggunakan IFrame. Meskipun frame telah lama dianggap sebagai teknologi yang bermasalah, kelemahan ini tidak selalu berlaku untuk IFrame. IFrame dapat diakses dan berinteraksi dengan baik dengan halaman web, dengan satu-satunya komplikasi nyata yang muncul jika Anda harus mengubah ukuran IFrame secara dinamis agar sesuai dengan kontennya. IFrame pada dasarnya menanamkan halaman web eksternal ke dalam

halaman; halaman induk hanya perlu mengetahui URL komponen yang akan disematkan, yang membuatnya sepenuhnya keluar dari proses bundling di atas, dan memungkinkan komponen untuk disebarkan dan dikelola secara independen dari halaman yang menggunakannya.

Hal ini memang menimbulkan serangkaian risiko lain jika server yang menghosting komponen IFrame mati, yang dalam hal ini akan ditampilkan sebagai kesalahan pada halaman induk, tetapi tidak akan terpengaruh. Membuat komponen IFrame mirip dengan membuat halaman web lain di situs, karena memerlukan URL yang merender HTML, tetapi URL ini biasanya tidak langsung terekspos ke pengguna karena komponen itu sendiri tidak bermakna. Komponen IFrame juga berguna jika Anda ingin berbagi konten yang akan disematkan langsung oleh situs web pihak ketiga yang mungkin memiliki banyak sistem atau gaya pembuatan yang berbeda (dalam hal ini akan sulit untuk menyediakan komponen untuk mereka), atau jika komponen Anda memerlukan akses ke cookie atau yang serupa di domain Anda yang tidak dapat diakses oleh pihak ketiga karena aturan lintas asal.

Jika suatu komponen perlu diparameterisasi, ini dapat dilakukan dengan meneruskan parameter kueri di URL dan kemudian membiarkan komponen yang merender konten IFrame bereaksi dengan tepat. API JavaScript (window.postMessage) juga memungkinkan halaman induk untuk berkomunikasi dengan IFrame yang disematkan jika interaktivitas lebih lanjut diperlukan, tetapi jika tidak, akses ke DOM induk atau anak dapat dibatasi kecuali jika keduanya dihosting di domain yang sama. Ini dapat menyebabkan beberapa masalah, jadi IFrame sering kali hanya digunakan untuk komponen yang relatif mandiri, dan bukan yang perlu berinteraksi secara signifikan dengan status halaman induk.

Tidak ada jawaban yang sempurna untuk masalah membangun UI web yang terkomponen. Komunitas web telah menyadari hal ini dan telah mulai mengerjakan serangkaian spesifikasi yang dikenal sebagai komponen web, yang memiliki cara "asli" untuk menyelesaikannya. Dengan komponen web, Anda dapat menentukan elemen kustom menggunakan JavaScript di bagian kepala dokumen Anda (atau dalam bundel), dan saat Anda ingin menyertakan komponen tersebut, Anda dapat menggunakannya seolah-olah itu adalah elemen HTML biasa, dengan meneruskan parameter apa pun sebagai atribut ke tag tersebut. Komponen web adalah sesuatu yang kompleks dan terus berkembang.

Penataan gaya ditangani menggunakan sesuatu yang dikenal sebagai shadow DOM, yang memungkinkan Anda menentukan gaya yang hanya berlaku dalam cakupan elemen HTML tersebut, tetapi juga menghentikan gaya tingkat dokumen agar tidak memengaruhi konten elemen kustom. Seiring dengan semakin matangnya komponen web dan peningkatan dukungan browser, tampaknya masuk akal untuk mengharapkan hal ini menjadi jauh lebih umum daripada saat ini.

6.4 DESAIN RESPONSIF

Desain responsif, atau desain web responsif (RWD), adalah teknik yang digunakan untuk mendesain halaman web yang akan menyesuaikan diri dengan kemampuan khusus perangkat tertentu. Teknik ini sebagian besar dimungkinkan oleh fitur CSS yang dikenal sebagai kueri media, dan sebelum desain responsif tersebar luas, merupakan hal yang umum untuk

membuat beberapa versi situs satu untuk desktop dan satu lagi untuk seluler lalu menyajikan HTML yang berbeda berdasarkan agen pengguna (string yang dikirim browser untuk menunjukkan versi) browser. Hal ini sering kali memerlukan pemeliharaan dua basis kode yang serupa tetapi terpisah, dan akibatnya banyak situs seluler hanyalah versi terbatas dari situs desktop, dan seiring pengguna semakin beralih ke seluler, hal ini menjadi tidak berkelanjutan.

Perluasan jenis layar tempat browser web dapat muncul (ponsel, tablet, laptop, jam tangan, TV, lemari es) mengakibatkan praktik pembuatan satu versi yang dapat beradaptasi dengan banyak perangkat. Namun, hal ini bukan tanpa kekurangannya. Anda sering kali harus membuat kode untuk penyebut umum terendah (ponsel tidak memiliki daya pemrosesan laptop, misalnya), yang dapat menyebabkan beberapa kompromi untuk perangkat kelas atas. Teknik yang dikenal sebagai peningkatan progresif, yang dibahas nanti dalam buku ini, dapat digunakan untuk mengatasi hal ini, tetapi ini sering kali tidak semudah memiliki satu versi biasa per situs.

Sayangnya, pilihan antara beberapa situs sederhana, yang memerlukan banyak pekerjaan untuk dikelola, versus satu situs yang rumit ini terus berlanjut hingga hari ini, tetapi konsensusnya adalah bahwa kerumitan tambahan masih lebih sedikit pekerjaan daripada membuat hanya dua versi dari satu situs. Mungkin ada keadaan di mana desain responsif tidak diperlukan misalnya, aplikasi internal yang hanya digunakan oleh pusat panggilan mungkin hanya dapat digunakan di desktop, jadi sebaiknya pertimbangkan opsi itu. Saat mendesain situs web responsif, ada baiknya mengkategorikan berbagai ukuran layar yang dibutuhkan lalu membuat desain yang berfungsi pada ukuran layar tersebut, meskipun beberapa situs hanya memiliki satu tata letak dasar dan memastikan bahwa konten berskala ke semua ukuran layar secara merata, yang merupakan solusi cepat dan mudah jika memungkinkan untuk konten dan desain Anda.

Untuk yang lain, Anda mungkin ingin memisahkan tata letak situs ke dalam kategori yang berbeda. Ini bisa sesederhana "mobile" dan "non-mobile", tetapi sering kali mencakup kategori menengah seperti "tablet" dan mungkin juga mempertimbangkan orientasi perangkat. Setiap kategori bukanlah tata letak tetap, tetapi mungkin mencakup berbagai perangkat dan ukuran layar di setiap kategori, jadi lebar variabel sering kali dipertimbangkan di sini. Sering kali, kategori terbesar memiliki lebar maksimum yang ditetapkan saat menjadi desain lebar tetap. Batasan setiap kategori dikenal sebagai breakpoint. Breakpoint adalah tempat tata letak halaman berubah dari satu kategori ke kategori lainnya. Misalnya, Anda mungkin ingin menetapkan breakpoint 400px, di mana di bawah 400px adalah kategori mobile, dan sama dengan atau di atas 400px adalah kategori tablet.

Perhatikan di sini bahwa piksel dalam CSS tidak selalu sesuai dengan piksel fisik pada layar. Ini dibahas lebih lanjut dalam bab UX, tetapi sebagai pengingat, dalam CSS, 1px sesuai dengan piksel jika layar perangkat adalah 96dpi, dan penggunaan piksel fisik disesuaikan jika lebih tinggi (atau lebih rendah). Mengubah tata letak secara drastis di setiap breakpoint dapat membingungkan jika pengguna beralih di antara keduanya (misalnya, jika mereka memutar perangkat, atau mengubah ukuran jendela browser di laptop), dan akan jauh lebih sulit untuk diterapkan. Perubahan kecil pada tata letak (mungkin berpindah dari satu kolom ke beberapa

kolom) daripada menyusun ulang seluruh halaman biasanya lebih disukai. Asumsi lain sering kali dimasukkan ke dalam berbagai ukuran layar ini.

Misalnya, seseorang mungkin berasumsi bahwa ukuran layar ponsel dan tablet akan digunakan untuk sentuhan, dan tombol atau area yang dapat diklik lainnya harus diperbesar untuk itu, tetapi laptop yang lebih kecil (atau browser web berdampingan di desktop) mungkin tidak memiliki layar sentuh, dan sebaliknya, laptop dan desktop layar sentuh dengan ukuran layar yang lebih besar semakin umum. Demikian pula, perangkat seluler sering kali diasumsikan dipegang lebih dekat ke wajah, sehingga teks dapat dibuat relatif lebih kecil. Asumsi ini dimasukkan ke dalam desain di antara setiap breakpoint, tetapi penting untuk menyadarinya, dan bahwa sebenarnya lebar/tinggi/orientasi perangkat digunakan sebagai proksi untuk asumsi yang lebih luas ini, dan mengandalkan faktor-faktor tersebut saja terkadang akan membuat Anda salah. Kueri media CSS adalah cara untuk menentukan blok CSS yang hanya berlaku saat karakteristik layar cocok dengan serangkaian kriteria tertentu. Misalnya, jika Anda ingin mengatakan bahwa teks harus berubah menjadi biru di ponsel, Anda dapat menulis yang berikut:

```
    @media (max-width: 400px) {
        p{
          color: blue;
      }
}
```

Kode sebelumnya menentukan bahwa CSS berlaku saat lebar browser kurang dari 400px. Anda dapat menggabungkan beberapa pernyataan menjadi satu kueri, sehingga aturan internal hanya berlaku saat semuanya ditetapkan. Dengan demikian, menggabungkan min-width dengan max-width memungkinkan Anda menentukan rentang:

```
    @media (min-width: 401px) and (max-width: 1024px) {
      ...
    }
```

Hal di atas hanya akan berlaku antara 401-1024 piksel. Penting untuk digarisbawahi bahwa kita mulai pada 401 piksel karena suatu alasan! Jika kita menggunakan kembali nilai 400 piksel, seperti yang kita lakukan pada kueri khusus seluler, maka jika perangkat berada tepat pada 400 piksel, maka kedua set kueri media akan berlaku. Ini akan menjadi kasus tepi yang aneh untuk ditemukan, karena lebar minimum dan lebar maksimum bersifat inklusif (setara dengan "lebih dari atau sama dengan" dan "kurang dari atau sama dengan"), sehingga keduanya saling tumpang tindih.

Selain lebar, Anda juga dapat menentukan tinggi dengan cara yang sama, serta lebar perangkat dan tinggi perangkat, yang memungkinkan Anda menentukan ukuran absolut perangkat yang Anda gunakan, daripada ukuran browser saat ini (jadi Anda dapat, misalnya, selalu merender situs desktop di desktop, meskipun situs tersebut dibuat lebih kecil), tetapi

ini belum tentu merupakan ide yang bagus. orientasi adalah properti penggunaan lainnya, dan dapat ditentukan dengan nilai potret atau lanskap untuk lebih menargetkan konfigurasi tertentu. Ada banyak properti lain, termasuk rasio aspek, dan fitur yang lebih baru seperti apakah pengguna dapat mengarahkan kursor dengan perangkat input mereka saat ini atau tidak, yang dapat digunakan untuk lebih eksplisit tentang asumsi yang dibuat tentang layar sentuh, dll.

Kueri media juga dapat dibuat menggunakan operator selain dan, termasuk atau yang berperilaku seperti yang Anda harapkan, dan bukan yang hanya dapat digunakan untuk meniadakan seluruh kueri, bukan parameter tertentu di dalamnya. Koma dapat digunakan untuk menentukan beberapa kueri berbeda yang berlaku (mirip dengan cara deklarasi CSS normal dapat menentukan beberapa kelas yang cocok). Terakhir, menarik untuk dicatat bahwa tujuan awal kueri media adalah untuk menentukan stylesheet "cetak" yang hanya berlaku saat halaman dicetak. Nilai breakpoint yang sebenarnya layak dipecah menjadi variabel saat menggunakan varian CSS yang mendukungnya, karena dapat diduplikasi dan mungkin perlu diubah selama pengembangan. Ini juga memudahkan untuk membuatnya tetap konsisten, memiliki satu set breakpoint di seluruh situs, daripada komponen yang berbeda memiliki set breakpoint mereka sendiri yang berlaku untuknya. Desain responsif dengan kueri media sering dikombinasikan dengan teknik yang dikenal sebagai “mobile first,” yang dibahas kemudian dalam bab ini.

6.5 PENINGKATAN PROGRESIF

Inti dari identitas Web adalah metafora dokumen. Setiap halaman web adalah satu dokumen, dan situs web terdiri dari kumpulan dokumen-dokumen tersebut. Setiap dokumen memiliki referensi unik global URL yang dapat digunakan untuk mencarinya, dan dokumen-dokumen tersebut dihubungkan dengan hyperlink menggunakan referensi-referensi ini. Ini benar-benar terobosan, dan selama bertahun-tahun metafora dokumen ini berlaku. Banyak standar ditulis berdasarkan gagasan bahwa halaman web hanyalah dokumen, dan kemudian muncul alat-alat yang memanfaatkan ini, seperti perangkat aksesibilitas dan mesin pencari. Kemudian, orang-orang mulai meletakkan hal-hal di web yang bukan dokumen. Situs web seperti Hotmail diluncurkan, dan tidak sepenuhnya sesuai dengan beberapa prinsip inti Web tersebut.

Saat ini, kita menyebut jenis situs ini sebagai aplikasi web. Ini dengan senang hati hidup berdampingan dengan web asli dan bekerja cukup baik dengan metafora dokumen sehingga tidak merusak terlalu banyak alat-alat tersebut. Harmoni ini dicapai dengan menggunakan teknik yang disebut peningkatan progresif. Peningkatan progresif adalah gagasan tentang pelapisan fungsionalitas di atas dokumen sederhana untuk memberikan pengalaman yang lebih kaya. Dan bukan hanya aplikasi web baru ini yang menggunakan peningkatan progresif; situs web berbasis dokumen tradisional juga mulai menggunakannya. Pada masa-masa awal web, dokumen yang dibentuk dengan baik dengan sedikit gaya merupakan norma.

Kemudian CSS muncul, yang memungkinkan orang untuk memberi gaya pada dokumen mereka, dan bukan hal yang aneh bagi orang untuk menerapkan stylesheet khusus di browser

mereka sendiri. Misalnya, pengguna yang kesulitan melihat dapat meningkatkan ukuran font default atau menambahkan kontras warna yang tinggi ke pengalaman default mereka. Ketika JavaScript muncul, skenario yang sama berlaku. Situs web berfungsi tanpa JavaScript, dan untuk browser yang mendukungnya, JavaScript meningkatkan pengalaman pengguna. Namun, pada akhir 1990-an dan awal 2000-an, Windows mulai mendominasi, bersama dengan Internet Explorer. Meskipun situs yang berbasis konten terus mematuhi standar dan teknik peningkatan progresif, aplikasi web, terutama aplikasi web intranet, berhenti melakukannya, karena lebih mudah untuk menganggap pengguna menjelajah dengan Internet Explorer.

Hal ini diperburuk oleh adopsi ekstensi milik Internet Explorer, yang memungkinkan jenis aplikasi web baru. Peningkatan progresif biasanya memerlukan disiplin teknik yang lebih tinggi daripada tidak menggunakan pendekatan tersebut, dan dalam lingkungan tersebut, hal itu tidak diperlukan. Dengan munculnya peramban seluler dan berakhirnya monopoli Microsoft atas peramban web, aplikasi web ini tiba-tiba menjadi bagian besar dari kode lama.Serupa dengan munculnya web seluler, banyak situs yang dibangun di sekitar lingkungan tertentu itu berjuang untuk beradaptasi dengan ponsel. Banyak organisasi tiba-tiba harus membangun situs seluler terpisah dan kemudian, bertahun-tahun kemudian, menggabungkannya kembali menggunakan teknik desain web responsif yang sedang naik daun.

Situs yang telah menggunakan teknik peningkatan progresif memiliki migrasi yang lebih mudah untuk ditangani. Banyak yang hanya menyajikan versi seluler dari templat CSS mereka, tetapi dapat membiarkan dokumen HTML mereka apa adanya. Hari-hari awal web seluler mirip dengan hari-hari awal Web di desktop. Puluhan perangkat yang berbeda, dengan peramban dan kemampuan yang berbeda, menjadikan peningkatan progresif satu-satunya permainan di kota. Namun, seperti konsolidasi pasar desktop ke sejumlah pemain yang lebih sedikit, pasar peramban seluler telah didominasi oleh Android dan Apple. Pada saat yang sama, kemampuan JavaScript meningkat ke titik di mana browser dapat melakukan lebih banyak hal, dan aplikasi web dapat lebih sedikit bergantung pada server. Banyak sekali kerangka kerja yang muncul, yang merangkul gagasan membangun aplikasi web semacam ini (Angular dan Meteor, di antara banyak lainnya). Ini umumnya tidak sesuai dengan gagasan peningkatan progresif, dan dalam beberapa kasus penggunaan itu tidak masalah.

Namun, alat-alat ini juga digunakan dalam banyak situasi di mana itu tidak masalah. Pengabaian peningkatan progresif dalam kasus-kasus ini tiba-tiba menimbulkan masalah ketika hal-hal yang sebelumnya dapat Anda anggap remeh di web jika Anda mematuhi standar penemuan mesin pencari, kompatibilitas dengan alat aksesibilitas tidak ada lagi. Selain itu, jenis aplikasi web ini dapat memiliki waktu mulai yang lama, menyebabkan pemuatan halaman yang lambat, dan ini terutama berlaku pada telepon pintar.

Meningkatkan Secara Progresif, atau Tidak?

Peningkatan progresif adalah tentang mengidentifikasi fungsionalitas inti situs web Anda dan membuatnya tersedia untuk semua orang, lalu menerapkan fungsionalitas tambahan atau hal-hal yang menarik sebagai lapisan di luar itu. Seiring dengan peningkatan teknologi web dan browser, pengalaman inti menjadi lebih kaya. Misalnya, memastikan bahwa

halaman Anda berfungsi tanpa CSS tidak lagi mutlak diperlukan. Di sisi lain, perangkat pada koneksi jaringan yang tidak stabil, atau masalah dengan server Anda, dapat menyebabkan halaman web dimuat tanpa lembar gaya apa pun. Anda mungkin berpikir argumen yang sama berlaku untuk JavaScript sedikit browser saat ini yang tidak mendukung JavaScript, dan kecuali ada koneksi yang tidak stabil, JavaScript harus dimuat.

Namun, tidak semua mesin pencari akan menjalankan JavaScript, atau hanya akan menjalankan sebagian kecilnya. Lebih jauh, meskipun sepenuhnya mungkin untuk membangun aplikasi web yang sepenuhnya dapat diakses dengan JavaScript, harus menerapkan semantik tersebut sendiri lebih sulit daripada menggunakan fungsionalitas bawaan browser. Selain itu, JavaScript memiliki lebih banyak mode kegagalan. Kesalahan ketik pada berkas CSS tidak akan merusak seluruh gaya Anda (meskipun mungkin merusak sebagian), tetapi kesalahan pada JavaScript Anda dapat menghambat fungsionalitas penting, atau mengubah sepenuhnya cara halaman ditampilkan.

Apa pun jenis situs web yang Anda buat, intinya adalah dokumen HTML yang Anda berikan kepada pengguna. Bagi mesin pencari dan alat aksesibilitas, ini tetap merupakan elemen terpenting. Sebagian besar akan memahami beberapa lapisan di atas, tetapi hanya sampai batas tertentu. Untuk situs web yang sarat konten, dokumen ini harus berisi esensi halaman Anda, dan harus dapat dirender sebelum JavaScript diterapkan ke dalamnya. Bagi aplikasi web, ini jauh lebih sulit dilakukan, karena bagi banyak aplikasi, esensi dari apa yang dilakukan situs tidak dapat diungkapkan hanya dengan menggunakan blok penyusun HTML yang sederhana, atau sering kali untuk melakukannya akan memerlukan upaya sisi server yang signifikan. Bagi beberapa aplikasi web, keputusan dibuat pada tingkat komponen per komponen mengenai apakah peningkatan progresif digunakan atau tidak, tetapi bagi yang lain, keputusan dibuat untuk tidak mencoba sama sekali.

Pola aplikasi satu halaman adalah salah satu contoh di mana peningkatan progresif ditinggalkan. Keputusan untuk menggunakan atau tidak menggunakan peningkatan progresif merupakan hal mendasar bagi seluruh pengalaman front-end, jadi harus dipertimbangkan dengan saksama. Bagi sebagian besar situs web konten, keputusan apa pun untuk tidak menerapkan teknik peningkatan progresif dapat menyebabkan masalah jangka panjang pada situs web. Dalam beberapa keadaan, keuntungan rekayasa jangka pendek dapat diperoleh, tetapi ini diimbangi dengan biaya yang lebih tinggi untuk menerapkan pengoptimalan mesin telusur dan menambahkan kembali kait yang sesuai untuk tujuan aksesibilitas. Peningkatan progresif telah teruji oleh waktu, dan telah menunjukkan nilai berulang kali, tetapi seperti banyak hal dalam rekayasa perangkat lunak, jawaban untuk pertanyaan "haruskah saya meningkatkan secara progresif?" adalah, "tergantung." Keputusan untuk tidak menerapkan teknik tersebut harus dibuat dengan saksama.

Tidak menggunakan peningkatan progresif, dikombinasikan dengan teknik untuk performa, seperti hanya memuat JavaScript saat halaman pertama kali dirender (untuk mempercepat waktu sebelum halaman dapat digunakan), dapat membuat halaman dalam keadaan setengah tidak nyaman, di mana tombol mungkin terlihat, tetapi mengkliknya tidak menghasilkan apa pun, karena JavaScript belum menambahkan peristiwa apa pun ke

dalamnya. Dalam konteks peningkatan progresif, ini tidak terlalu mengganggu, karena dapat kembali ke versi yang tidak ditingkatkan secara progresif (seperti mengirimkan formulir dengan cara tradisional, bukan dengan AJAX).

Namun, ketika ada seluruh fitur yang dikirimkan menggunakan JavaScript, jika ini hanya ditambahkan setelah halaman dimuat, ini dapat menyebabkan kilatan yang mengganggu saat halaman dirender ulang, atau menyebabkan konten melompat-lompat. Untuk mengatasinya, Anda dapat menggunakan tag skrip sebaris kecil untuk menambahkan kelas ke tag `<body>`, lalu menggunakan CSS untuk menyembunyikannya secara default hingga skrip tersebut berjalan. Jika skrip sebaris adalah hal pertama dalam badan, elemen akan segera terlihat saat ditampilkan pada browser yang mendukungnya, mungkin dalam status "dinonaktifkan" atau "dimuat", hingga diaktifkan sepenuhnya setelah JavaScript dimuat.

6.6 MOBILE FIRST

Pelengkap untuk peningkatan progresif adalah teknik yang dikenal sebagai "mobile first." Teknik ini dimulai dengan asumsi bahwa browser pada ponsel cerdas adalah browser yang paling tidak mampu: perangkat kerasnya kurang bertenaga, koneksi jaringan kurang dapat diandalkan, dan mekanisme interaksi paling terbatas. Teknik ini sering digunakan dalam tahap desain interaksi juga, di mana desain dikembangkan dengan mempertimbangkan lingkungan yang paling terbatas terlebih dahulu, dan saat batasan tersebut dihilangkan, lebih banyak yang dapat ditambahkan. Banyak tim merasa lebih mudah untuk menambahkan fungsionalitas ke perangkat yang didukung daripada harus memasukkan pengalaman untuk perangkat yang lebih besar atau lebih mampu ke perangkat yang lebih kecil.

Dengan mobile first, penting untuk menetapkan garis dasar. Satu dekade lalu, ponsel pintar BlackBerry atau Nokia mungkin menjadi titik awal, tetapi sekarang ini lebih cenderung berupa ponsel pintar murah, yang dapat menjalankan peramban yang sangat mumpuni tetapi dibatasi oleh seberapa ketinggalan zamannya, atau kinerja perangkat. Setelah Anda menetapkan dasar ini, Anda dapat merancang dan membangun versi yang menargetkan perangkat tersebut, lalu mencari tahu kelas perangkat lain yang lebih mumpuni yang ingin Anda targetkan. Ponsel pintar yang lebih mumpuni mungkin sangat mirip, mungkin dengan layar definisi yang lebih besar atau lebih tinggi, tetapi kemudian Anda mungkin beralih ke tablet, yang layarnya jauh lebih besar, dan cara orang memegang dan berinteraksi dengan perangkat tersebut berubah total.

Kelas perangkat terakhir yang perlu dipertimbangkan adalah perangkat desktop atau laptop, yang mana pengguna menavigasi menggunakan mouse (atau trackpad), yang memberi Anda kontrol yang jauh lebih baik atas interaksi dengan UI, serta fitur-fitur seperti hovering. Jika kita mencoba menerapkan teknik ini secara terbalik, maka nilai mobile-first menjadi jelas. Jika kita mengasumsikan desktop terlebih dahulu, lalu menggunakan gerakan melayang sebagai bagian dari UI, maka saat varian seluler dikembangkan, diperlukan solusi untuk mengaktifkan interaksi yang sama, yang mungkin tidak terlihat oleh pengguna seluler, atau pengerjaan ulang yang signifikan perlu dilakukan pada antarmuka untuk memperhitungkan fakta bahwa pengguna tidak akan selalu dapat mengarahkan kursor.

Dengan menangani desain dalam konteks yang mengutamakan seluler, kita menghilangkan penggunaan interaksi khusus desktop seperti gerakan melayang, atau hanya menggunakannya untuk "hal-hal yang menyenangkan" tambahan, bukan bagian penting dari fungsionalitas. Namun, gerakan yang mengutamakan seluler melampaui cara Anda membangun UI. Dengan menangani fakta bahwa jaringan mungkin tidak dapat diandalkan sejak awal, Anda dapat membangun dengan mempertimbangkan batasan tersebut sejak awal, daripada harus kembali dan melakukan perbaikan atau menulis ulang kode. Sebagian besar situs web saat ini juga akan merujuk ke media eksternal, dan mobile first menganjurkan Anda untuk menyediakan aset yang memiliki ukuran atau kualitas yang sesuai untuk perangkat seluler. Ambil contoh gambar.

Tidak ada gunanya mengunduh gambar 4K untuk ponsel dengan layar beresolusi rendah pada koneksi yang lambat. Karena ponsel umumnya memiliki layar yang lebih kecil daripada laptop, mungkin masuk akal untuk membuat tag <img> dengan src dengan resolusi kecil gambar yang sesuai untuk ponsel sebagai penyebut umum terendah. JavaScript kemudian dapat digunakan untuk memeriksa ukuran layar sebenarnya dan mengganti src tersebut dengan gambar berukuran lebih sesuai, tetapi memulai dengan gambar kecil memberikan pengalaman tercepat di ponsel, sekaligus menyempurnakannya untuk perangkat yang tidak terlalu dibatasi nantinya. Perilaku semacam ini semakin diabaikan dalam spesifikasi HTML. Untuk gambar, srcset sekarang melakukan ini tanpa memerlukan JavaScript apa pun, dan untuk media seperti audio atau video, format seperti MPEG-DASH juga mendukung penyesuaian ukuran media dan bitrate dengan tepat. Mobile first, bila digunakan dengan desain web responsif, biasanya melibatkan pendefinisian semua gaya seluler tanpa menggunakan kueri media apa pun, lalu mengesampingkan gaya khusus seluler apa pun dengan kueri media untuk perangkat dan ukuran layar yang lebih besar.

Deteksi Fitur

Bagi banyak orang, tingkat peningkatan progresif yang paling sederhana adalah "apakah inti pengalaman berfungsi tanpa JavaScript?" dan ini bukanlah titik awal yang buruk. Namun, seperti yang dibahas di atas, ada beberapa pengalaman, terutama di aplikasi web, di mana esensi inti tidak dapat diungkapkan hanya menggunakan HTML dan CSS. Namun, peningkatan progresif bukan tentang gagasan biner "apakah JavaScript ada atau tidak," dan dapat mencakup lebih banyak hal. Ambil contoh, geolokasi. Meskipun API JavaScript ada di browser modern, tidak ada jaminan bahwa itu benar-benar akan berfungsi (misalnya, jika perangkat tidak memiliki chip GPS, atau tidak dapat memperoleh sinyal dan GeoIP tidak meyakinkan), jadi Anda tidak boleh mengandalkan API tersebut untuk pengalaman inti situs Anda.

Misalnya, jika Anda membangun fitur "cari toko", maka Anda mungkin ingin memulai dengan formulir pencarian sederhana yang tidak bergantung pada JavaScript, dan hanya menampilkan tombol "Gunakan lokasi saat ini" jika API Geolokasi tersedia. Ini adalah teknik yang disebut deteksi fitur, di mana area fungsionalitas tertentu di situs web Anda hanya diaktifkan saat fitur terkait terdeteksi di browser pengguna. Penerapan deteksi untuk fitur tertentu bergantung pada fitur tertentu, tetapi ada pustaka yang dapat mengabstraksikan hal

ini. Teknik lain yang telah banyak digunakan adalah teknik "cuts the mustard".

Ini melampaui pendekatan sederhana "apakah JavaScript berjalan"; sebagai gantinya, pengujian dijalankan untuk menentukan apakah browser memiliki dukungan dasar untuk JavaScript modern. Menggunakan teknik semacam ini dapat membantu pengujian, karena alih-alih kombinasi besar browser dan perangkat dengan fitur berbeda untuk diuji, pengujian dapat dibagi menjadi browser dan perangkat yang "cuts the mustard" dan yang tidak. Browser yang tidak cut the mustard mendapatkan pengalaman non-JavaScript, yang lebih baik daripada memiliki situs web yang rusak secara halus, dan memberi Anda cara yang lebih mudah untuk mengonfirmasi bahwa situs Anda berfungsi untuk mesin pencari dan dalam situasi lain di mana JavaScript tidak dapat diandalkan.

Peningkatan Gaya Secara Progresif

Peningkatan CSS secara progresif lebih merupakan urusan biner. Biasanya, situs mengecilkan semua CSS menjadi satu berkas, dan memuatnya—oleh karena itu, halaman tersebut ditata sepenuhnya, atau tidak sama sekali. Bila berkas CSS belum dimuat, biasanya sangat jelas bagi pengunjung bahwa ada yang salah (dan jika mereka menggunakannya pada perangkat seluler dengan sinyal yang buruk, mereka mungkin berpikir masalahnya ada pada mereka, bukan pada situs), tetapi bahkan di sini Anda dapat memanfaatkan situasi buruk dengan peningkatan progresif. Saat membangun situs, dengan berfokus pada struktur dan konten halaman dalam HTML, jika CSS gagal dimuat, setidaknya pengguna masih dapat membaca apa pun yang telah Anda terbitkan (meskipun tidak cantik). Ini juga bagus untuk pengoptimalan mesin telusur.

Aturan praktis yang baik adalah jika halaman masih masuk akal tanpa CSS diterapkan, maka mesin telusur mungkin bisa mendapatkan beberapa data yang berarti darinya. Sayangnya, CSS tidak sesederhana proses pemuatan berkas yang serba ada atau tidak sama sekali. Seperti JavaScript, ada berbagai tingkat dukungan untuk fitur dan properti CSS di berbagai peramban, tetapi tidak seperti JavaScript, CSS tidak memiliki deteksi fitur. Untungnya, peramban akan mengabaikan pernyataan CSS yang tidak dikenali, yang dapat menghilangkan sebagian besar kebutuhan untuk peningkatan progresif. Misalnya, jika Anda perlu mendukung Internet Explorer lama, sebaiknya Anda menerima bahwa elemen tidak akan sempurna pikselnya di peramban lama tersebut. Demikian pula, beberapa peramban hanya menerima versi properti yang "diberi awalan" (terkadang dengan sintaksis yang berbeda untuk nilai daripada yang akhirnya distandarkan).

Alat seperti autoprefixer dapat berperilaku seperti polyfill JavaScript untuk memungkinkan Anda mengaksesnya. Agak lebih sulit untuk mengelola perubahan signifikan dalam CSS yang tidak dapat diurai oleh peramban. Contoh paling terkenal dari hal ini adalah untuk versi lama Internet Explorer (sebelum 9) yang tidak mendukung kueri media. Karena pengembang menggunakan peningkatan progresif yang mengutamakan perangkat seluler dan desain web responsif, hal ini memberikan pengalaman yang jauh lebih buruk bagi pengguna IE. Solusi umum adalah membuat stylesheet khusus IE, yang sesuai dengan titik henti "desktop", lalu menggunakan komentar bersyarat IE untuk menyertakannya hanya di IE. Ini jelas bukan peningkatan progresif dalam bentuk paling murni, tetapi peretasan yang

pengorbanannya masuk akal.

Saat Tidak Menggunakan Peningkatan Progresif

Untuk situs web yang belum dibangun menggunakan peningkatan progresif, sering kali ada kebutuhan untuk menyesuaikan kembali keuntungan dari peningkatan progresif. Contoh yang paling umum adalah kebutuhan untuk dapat ditemukan oleh mesin pencari, dan untuk waktu muat yang cepat. Selain itu, salah satu argumen yang lebih umum terhadap peningkatan progresif adalah bahwa hal itu dapat meningkatkan biaya rekayasa; fungsionalitas harus diimplementasikan di sisi server dan klien. Solusi untuk keduanya adalah rendering sisi server, tetapi mekanisme untuk melakukannya sangat bervariasi. Ada banyak proxy dan server di luar sana yang akan merender halaman menggunakan browser tanpa kepala dan kemudian menyajikan konten yang dihasilkan langsung kepada pengguna, tetapi ini dapat mengganggu kerangka kerja aplikasi satu halaman, yang tidak mengasumsikan adanya konten yang sudah ada.

Beberapa kerangka kerja dapat mengikatkan diri mereka ke konten yang dirender sisi server dan menggunakannya sebagai status awal, sehingga mempercepat waktu pembuatan. Terkadang proksi ini hanya akan melakukannya jika mendeteksi agen pengguna milik mesin pencari, teknik yang sering kali dikenai sanksi oleh mesin pencari jika menemukannya. Salah satu teknik yang muncul sebagai hasil dari popularitas NodeJS untuk mengurangi biaya rekayasa (membangun logika dan tampilan yang sama di sisi server dan klien) adalah "JavaScript Isomorfik" (atau "JavaScript Universal"). Dengan menggunakan JavaScript di server, ini akan merender tampilan menggunakan kode JavaScript yang sama yang digunakan di klien, dengan beberapa status awal. Modul kemudian dapat dibagikan di sisi server dan klien, tanpa harus mengimplementasikan ulang inti tampilan.

6.7 OPTIMASI MESIN PENCARI

Optimalisasi mesin pencari (SEO) sering kali dianggap sebagai seni yang gelap, tetapi merupakan suatu keharusan bagi situs web berbasis konten apa pun di zaman sekarang ini, karena mesin pencari merupakan cara utama orang menjelajahi Web. Kenyataannya, kebanyakan orang akan mengunjungi situs Anda baik melalui tautan di media sosial atau aplikasi web, atau dengan mencarinya. Memasukkan URL secara manual telah menjadi hal yang biasa bagi pengguna yang ahli. SEO pada dasarnya adalah mekanisme untuk membuat konten situs Anda mudah dipahami oleh mesin pencari sehingga mereka memberi peringkat tinggi pada situs Anda untuk istilah yang relevan, serta mengadopsi sinyal teknis positif lainnya yang digunakan mesin pencari untuk memberi peringkat. Air telah dikeruhkan oleh agensi SEO yang secara etika dipertanyakan yang menerapkan peretasan jangka pendek seperti menambahkan banyak kata kunci yang tidak perlu ke halaman, atau bahkan teknik yang lebih meragukan seperti teks tersembunyi, atau menyajikan konten yang berbeda kepada mesin pencari.

Cara penting lainnya yang digunakan mesin pencari untuk memberi peringkat konten adalah jumlah tautan masuk dari situs yang bereputasi baik. Sekali lagi, beberapa agensi yang meragukan mungkin menyebarkan URL Anda ke situs melalui komentar dalam upaya untuk

meningkatkan peringkat Anda, tetapi peretasan semacam ini paling banter hanya bersifat jangka pendek, dan sering kali merusak dalam jangka panjang. Optimasi mesin pencari jelas diperlukan untuk halaman web jenis konten, tetapi bahkan untuk halaman jenis aplikasi, beberapa dapat berguna. Meskipun banyak dari halaman ini mungkin tersembunyi di balik login dan tidak dapat dicari secara langsung, ada kemungkinan hal-hal seperti halaman arahan akan dapat dicari, jadi menerapkan teknik semacam ini akan membantu di sana. Pasar mesin pencari tidak diragukan lagi didominasi oleh Google, yang tidak mengungkapkan secara pasti bagaimana hasil diberi peringkat, tetapi telah menunjukkan bahwa beberapa faktor tertentu penting.

Etos Google dan mesin pencari lainnya adalah bahwa hal-hal yang penting bagi manusia adalah hal-hal yang penting bagi mesin. Pada akhirnya, metode teknis tidak dapat memperbaiki konten yang buruk, jadi mendapatkan konten yang tepat, termasuk tajuk berita yang efektif dan standar bahasa Inggris yang baik, adalah satu hal terpenting yang dapat Anda lakukan. Yang kedua adalah memastikan bahwa Google dapat memahami halaman Anda. Meskipun perayap mesin pencari dapat menjalankan beberapa JavaScript, tidak jelas berapa banyak bahasa yang mereka dukung. Untungnya, membuat situs mudah diurai oleh browser menggunakan teknik seperti peningkatan progresif dan memastikan situs tersebut dapat diakses berarti mesin pencari juga dapat mengurainya, jadi situs web yang dirancang dengan baik untuk pengguna manusia juga akan berfungsi dengan baik untuk tujuan SEO.

Namun, bukan hanya konten halaman itu sendiri yang dapat bermanfaat bagi mesin pencari. Ada beberapa bagian metadata dan tag terstruktur tertentu yang harus ditambahkan ke HTML Anda untuk membantu meningkatkan hasil pencarian. Tag <title> dalam HTML adalah salah satu contohnya, karena ini adalah nama dokumen seperti yang muncul di hasil pencarian. Tingkat paling sederhana di luar ini adalah apa yang disebut tag meta, yang dapat ditambahkan di dalam <head> dokumen HTML Anda untuk menunjukkan bagian data tambahan, seperti ringkasan dokumen, yang dikenal sebagai deskripsi meta. Mesin pencari dan media sosial tertentu juga mendukung tag yang lebih spesifik, seperti "rich snippet," untuk menunjukkan media atau elemen lain yang menonjol di halaman hasil, atau saat berbagi di media sosial untuk menghasilkan jendela pratinjau kecil.

Menambahkan tag semacam ini akan membuat situs Anda menonjol dibanding yang tidak, tetapi hampir semua orang menggunakannya, jadi jika tidak ada tag, konten Anda akan mendapat peringkat rendah, bukan berarti konten Anda akan mendapat peringkat tinggi. Selain konten dan metadata, ada sinyal lain yang dipertimbangkan mesin telusur saat memberi peringkat hasil. Misalnya, jika situs Anda tidak responsif terhadap ukuran layar ponsel, situs Anda tidak akan muncul setinggi itu untuk penelusuran yang dilakukan dari ponsel. Kecepatan dan keamanan juga penting. Google khususnya akan menghukum situs yang lambat ditampilkan dan dimuat, karena pengguna sering kali tidak akan menunggu lebih dari beberapa detik untuk memuat situs. Ini membuat pengelolaan kinerja situs menjadi dua kali lebih penting bagi Anda. Situs web yang bukan HTTPS juga tidak akan berkinerja sebaik situs yang disajikan melalui koneksi aman di Google.

Google sering menggunakan sinyal ini untuk mendorong web ke arah yang

diinginkannya, menuju masa depan yang lebih cepat dan lebih aman, dan tidak mengherankan jika Google mendorong lebih banyak perbaikan teknis sebagai sinyal penelusuran untuk meningkatkan kualitas Web secara umum. Rangkaian sinyal penting terakhir untuk mesin pencari, seperti yang dibahas di atas, adalah jumlah dan kualitas tautan masuk ke situs Anda. Jika Anda memiliki konten yang sama yang dibagi di banyak halaman, maka sering kali ada beberapa URL yang dapat ditautkan orang, menyederhanakan desain situs Anda sehingga hanya ada bagian konten dengan satu alamat yang ditautkan yang berarti bahwa satu tautan itu akan mendapatkan peringkat yang lebih tinggi daripada memiliki sejumlah besar tautan yang masing-masing memiliki peringkat yang lebih rendah.

Dalam hal benar-benar membuat situs eksternal tersebut menautkan ke situs Anda, ini adalah masalah yang sama sekali non-teknis! Menautkan ke situs dari media sosial dan menggunakan siaran pers untuk menautkan ke konten Anda, atau bahkan situs lain yang menautkan kembali, akan membantu. Kata terakhir tentang pengelolaan URL untuk SEO adalah bahwa jika URL Anda berubah, Anda akan sering menemukan peringkat pencarian Anda turun karena Anda kehilangan tautan masuk. Hal ini dapat diatasi dengan penggunaan pengalihan HTTP yang benar, tetapi akan selalu ada beberapa fase perantara antara penambahan pengalihan dan halaman atau URL baru yang mendapatkan peringkat yang sesuai. Dalam hal ini, penting untuk memiliki struktur URL yang baik dan masuk akal di awal.

Akan ada saatnya Anda merasa harus menghapus konten dan mengubah pengalihan, dan ketika itu terjadi, awalnya mungkin tampak bahwa Anda dapat mengatur ulang navigasi internal dan situs akan baik-baik saja, tetapi jangan lupakan tautan masuk apa pun, dan selalu atur pengalihan dengan tepat. Ini hanyalah ikhtisar singkat tentang SEO, karena ini adalah bidang yang bergerak cepat dan berubah dari hari ke hari. Buku apa pun yang mencoba membahas apa pun selain konsep tingkat tinggi akan menjadi usang dengan sangat cepat. Namun, ini bukanlah seni gelap yang hanya dapat diterapkan oleh spesialis dari agensi SEO; ini adalah sesuatu yang seharusnya dapat dilakukan oleh pengembang tumpukan penuh dengan sedikit kesulitan dengan memahami prinsip-prinsip ini.

Alat Bangun

Saat menggunakan alat seperti Sass, dan bahkan saat tidak, sering kali ada langkah-langkah yang harus kita ambil untuk mengubah kode sumber kita menjadi versi yang dikirimkan ke browser pengguna. Hal ini serupa dengan cara bahasa yang dikompilasi mungkin memerlukan semacam alat bantu untuk mengompilasi dan menyusun kelas Java ke dalam .jar untuk didistribusikan, dan memang beberapa alat bahasa tradisional tersebut telah mengembangkan beberapa kemampuan untuk menangani kode front-end. Namun, alat yang paling matang dan umum untuk membangun kode front-end adalah alat yang dirancang khusus untuk tujuan tersebut. Ada banyak alat, masing-masing dengan filosofi dan pendekatannya sendiri, dan banyak alat baru yang muncul setiap saat.

Beberapa alat ini ditargetkan untuk satu tugas misalnya, Compass bertanggung jawab untuk membangun file CSS dari input Sass tetapi yang lain lebih umum dan mengatur sejumlah tugas misalnya, Webpack mencoba memenuhi semua masalah pembangunan front-end. Ada juga alat yang menjalankan tugas yang dapat mengatur berbagai bagian perkakas

pembangunan Anda menjadi satu perintah. Beberapa alat yang populer termasuk Gulp atau Grunt, yang mungkin menggabungkan sejumlah alat yang lebih terfokus, atau alat seperti skrip NPM atau makefile, yang sering digunakan dengan alat seperti Webpack. Mengingat dunia front-end berevolusi dengan sangat cepat, saya tidak akan memberikan rekomendasi konkret di sini, karena rekomendasi tersebut niscaya akan segera menjadi usang, tetapi saya akan membahas pendekatan umum terhadap jenis aktivitas yang mungkin ingin Anda jalankan di alat build Anda.

Sering kali, Anda akan mengonfigurasi alat build Anda untuk menggunakan alur kerja, dimulai dengan file sumber dan menghasilkan build final, dengan setiap tahap mengambil output dari tahap sebelumnya. Tahap-tahap ini akan bervariasi, tetapi akan sering berkembang sebagai berikut:

- Dimulai dengan titik masuk, memprosesnya untuk menemukan semua impor, dan menggabungkannya menjadi satu keluaran yang dibangun
- Mentranspilasi setiap impor (misalnya, mengubah JavaScript menjadi varian yang kompatibel dengan versi sebelumnya, atau Sass menjadi CSS)
- Membuat peta sumber yang dapat digunakan untuk membantu men-debug versi akhir dengan memetakannya kembali ke kode asli (peta sumber menjelaskan bagaimana kode yang dikompilasi berhubungan dengan kode asli, dan didukung oleh browser utama untuk memungkinkan Anda men-debug kode asli, daripada versi yang dikompilasi)
- Meminimalkan kode atau gambar, melalui kompresi atau menghapus spasi untuk meminimalkan ukuran file, yang dapat meningkatkan kinerja bagi pengguna akhir

Alat-alat build Anda juga akan ingin menjalankan pengujian apa pun terhadap kode Anda. Ini bisa berupa pengujian unit dan integrasi, pemeriksa gaya (kadang-kadang dikenal sebagai linter yang memastikan kode Anda cocok dengan praktik terbaik dan gaya pilihan Anda), atau alat analisis lain yang mengidentifikasi bug umum. Sebagian besar alat build akan memiliki dua mode satu untuk menghasilkan build "produksi", dan yang lainnya untuk mendukung pengembangan dengan lebih baik. Menjaga perbedaan antara build produksi dan build pengembangan sekecil mungkin dapat meminimalkan bug yang hanya muncul dalam produksi, tetapi beberapa memang diperlukan.

Untuk proyek besar, minifikasi dapat memakan waktu lama, sehingga terkadang dilewati dalam mode pengembangan untuk meminimalkan waktu tunggu setelah membuat perubahan. Di sisi lain, peta sumber yang sangat berharga untuk debugging dapat berukuran besar dan memperlambat produksi, sehingga dilewati. Beberapa pustaka melangkah lebih jauh, terkadang dengan menyertakan pencatatan debug yang diperluas atau memeriksa nilai saat runtime, yang kemudian dilewati karena alasan kinerja dalam build produksi. Perbedaan umum lainnya antara pengembangan dan produksi adalah bahwa sering kali produksi akan berjalan sekali dengan output yang diunggah ke CDN, sedangkan pengembangan akan terjadi dalam mode "watch", di mana alat build akan mengawasi file sumber untuk perubahan dan kemudian secara otomatis menjalankan ulang build.

Proses ini dapat menggunakan caching agar jauh lebih cepat daripada build penuh atau

uji coba. Beberapa alat build mendukung kemampuan untuk melangkah lebih jauh, menggunakan teknik seperti pemuatan ulang langsung atau pemuatan ulang modul panas. Dalam kasus ini, alat build akan memulai server web untuk menyajikan aset yang dibangun, tetapi menyisipkan kode tambahan yang akan menyebabkan browser web Anda terhubung ke server web tersebut menggunakan soket web atau yang serupa. Server web kemudian akan memberi tahu browser untuk secara otomatis menyegarkan halaman jika kode telah berubah, atau, dalam kasus yang lebih canggih, memuat ulang satu komponen halaman untuk pengembangan yang lebih cepat.

Jangan remehkan kompleksitas atau kekuatan alat build yang disetel dengan baik. Mengonfigurasi build dengan benar dapat memberikan dampak kinerja nyata bagi pengguna akhir Anda, serta sangat meningkatkan alur kerja Anda. Lakukan riset untuk menemukan apa yang paling cocok untuk tumpukan Anda, dan luangkan waktu untuk menyiapkan kerangka yang sesuai untuk Anda dan yang dapat Anda gunakan kembali.

6.6 KESIMPULAN

Bagian depan aplikasi Anda adalah bagian yang benar-benar berinteraksi dengan pengguna Anda. Di web, HTML menentukan tata letak dokumen Anda, CSS menentukan gaya visual, dan JavaScript dapat digunakan untuk memberikan tingkat interaktivitas dengan memanipulasi struktur HTML dan aturan CSS. Semua aplikasi dimulai dengan mengirimkan HTML ke browser web, meskipun hanya sedikit untuk mem-bootstrap aplikasi web yang kaya. Untuk banyak situs, sebagian besar HTML dihasilkan di sisi server, dan ada berbagai cara untuk menghasilkannya: secara otomatis menggunakan pustaka dan kerangka kerja, yang memiliki fleksibilitas terbatas; transformasi pohon, di mana beberapa struktur data lain seperti XML atau HAML diterjemahkan langsung ke HTML; atau interpolasi string, di mana HTML dibuat menggunakan templat berbasis file atau string.

CSS terstruktur menjadi penyeleksi dan aturan. Penyeleksi menjelaskan bagian HTML mana yang harus diterapkan aturan, dan aturan kemudian menentukan efek yang tepat pada konten yang dirender. Beberapa aturan, seperti font, berlaku secara berjenjang sehingga berlaku untuk anak-anak dari elemen yang ditargetkan, dan ketika beberapa aturan dapat berlaku untuk elemen HTML tertentu, kekhususan digunakan untuk menentukan mana yang diutamakan. CSS juga dapat menargetkan elemen semu dan kelas semu, yang tidak ada dalam HTML tetapi mewakili konten virtual atau status yang berbeda.

CSS telah lama menjadi bahasa yang cukup terbatas. Bahasa lain telah diperkenalkan, seperti Sass atau Less, yang dikompilasi menjadi CSS tetapi menyediakan tambahan sintaksis, seperti variabel atau pewarisan, melalui bahasa CSS biasa. Pendekatan berbasis komponen untuk membangun front end biasanya digunakan, dan pendekatan ini cocok dengan pendekatan banyak desainer UX. Ada beberapa pendekatan untuk ini, seperti menangani templat HTML, CSS, dan JavaScript secara terpisah, meskipun dalam beberapa aplikasi yang kaya, Anda dapat menggabungkannya bersama-sama dengan beberapa biaya kinerja potensial. Komponen-komponen ini dapat digabungkan sebagai kode bersama atau dihosting dan disematkan sebagai IFrame.

Dua teknik penting untuk front end adalah desain responsif dan peningkatan progresif. Desain responsif adalah tempat gaya situs menggunakan kueri media untuk berubah berdasarkan ukuran layar pengguna, dan sering dikombinasikan dengan metodologi yang mengutamakan perangkat seluler, tempat desain dasar ramah seluler dan situs merespons saat layar diperbesar. Dengan peningkatan progresif, semua pengguna mendapatkan pengalaman dasar yang sama, dan kemudian JavaScript digunakan untuk meningkatkan pengalaman pengguna saat perangkat mendukungnya. Hal ini meningkatkan jangkauan aplikasi Anda dengan tidak bergantung pada fitur tertentu, tetapi memanfaatkannya saat fitur tersebut tersedia.

Bagian terakhir dari desain front-end yang perlu dipertimbangkan adalah pengguna non-manusia dari situs web Anda. Hal ini dapat mencakup mesin pencari, yang mengindeks situs Anda dan memungkinkannya untuk dicari, dan banyak teknik yang membuat situs dapat digunakan dan diakses dapat membuat situs Anda menarik bagi mesin pencari.

BAB 7
PENGUJIAN

Pada suatu saat selama setiap proyek, pasti akan ada dua pertanyaan yang perlu dijawab untuk setiap perubahan yang Anda buat: apakah berhasil, dan apakah ada yang rusak? Pengujian dapat digunakan untuk menjawab kedua pertanyaan ini, meskipun ada perbedaan kecil antara jenis pengujian yang digunakan untuk menyelesaikannya.

7.1 PENGEMBANGAN BERBASIS PENGUJIAN

Pengembangan berbasis pengujian (TDD) adalah teknik yang mapan untuk mengembangkan perangkat lunak, meskipun sering disalahpahami. Tujuan TDD bukanlah untuk menghasilkan cakupan pengujian yang tinggi, tetapi untuk menggunakan pengujian guna membantu mendorong desain kode Anda; TDD benar-benar tentang pengembangan, bukan hanya pengujian. TDD didasarkan pada siklus "merah-hijau-refaktor." Pertama-tama, Anda menulis pengujian untuk fungsionalitas baru yang ingin Anda tambahkan, yang seharusnya gagal (berubah menjadi merah), lalu Anda menulis sedikit fungsionalitas yang paling sederhana agar pengujian berhasil (berubah menjadi hijau). Setelah Anda melakukan ini, Anda dapat melakukan refaktor aplikasi dan kode pengujian untuk merapikannya dan menghilangkan duplikasi.

Dalam pengembangan berbasis pengujian, biasanya pengujian yang hanya ditujukan untuk satu modul dalam satu waktu adalah hal yang umum. Jenis pengujian ini dikenal sebagai pengujian unit, berbeda dengan pengujian integrasi, yang akan dibahas lebih lanjut di bab ini, dan yang menguji beberapa modul sekaligus. Beberapa orang, terutama mereka yang mempraktikkan pengembangan berbasis perilaku, akan sering memulai dengan menulis pengujian yang lebih luas dari satu modul tertentu, lalu menulis pengujian yang diperlukan dengan cakupan yang lebih kecil yang memenuhi persyaratan pengujian yang lebih besar ini adalah pendekatan luar-dalam. Yang lain lebih suka memulai dari unit terkecil lalu menulis pengujian integrasi untuk menyambungkan modul-modul setelah dibuat, yang disebut sebagai pendekatan dalam-luar.

Pendekatan mana yang Anda pilih sering kali bergantung pada gaya pribadi, meskipun para pendukung pendekatan luar-dalam mengatakan pendekatan ini membantu mereka memutuskan komponen mana yang perlu ditulis, dan para pendukung pendekatan dalam-luar menyukai fleksibilitas karena tidak perlu merancang antarmuka masing-masing modul terlebih dahulu. Jika Anda mengalami masalah saat menulis pengujian yang hanya menargetkan satu modul, hal itu mungkin menunjukkan bahwa ada beberapa modul yang sangat terkait yang mungkin lebih baik digabungkan menjadi satu, atau, jika ada beberapa jenis pengujian yang berbeda untuk satu modul tersebut, modul Anda seharusnya dipecah menjadi beberapa modul yang lebih kecil.

“Arrange-act-assert” (kadang-kadang disebut “given-when-then,” khususnya saat

digunakan dengan pengembangan berbasis perilaku, yang dibahas nanti dalam bab ini) adalah pola umum untuk mengatur kode pengujian Anda. Ambil contoh di bawah ini:

```
    it('should increase the size of the shopping cart when adding
    a new item to it', () => {
          const cart = new ShoppingCart();
          const product = fetchTestProduct();
          cart.add(product);
          expect(cart.size()).toEqual(1);
});
```

Di sini, proses arrange-act-assert ditampilkan menggunakan spasi, yang merupakan pola umum yang dapat membantu keterbacaan saat memindai file pengujian. Bagian "arrange" dari pengujian melibatkan pengaturan objek dan bit data yang ingin Anda uji ke dalam status yang tepat, lalu "menindaklanjutinya" dengan menjalankan fungsi yang sedang diuji. Terakhir, "assert" memeriksa apakah langkah act berhasil. Mungkin ada kalanya gaya ini tidak berfungsi untuk jenis pengujian yang ingin Anda tulis, tetapi berhati-hatilah pengujian yang mengikuti pola arrange-act-assert-act-assert sering kali harus dipecah menjadi dua pengujian terpisah, mungkin mengabstraksikan langkah "arrange" menjadi fungsi pembantu yang digunakan dalam beberapa pengujian.

Setiap pengujian yang Anda tulis harus menguji satu bit fungsionalitas logis: bagian kode yang Anda uji. Ini mungkin berarti ada beberapa baris pernyataan dalam pengujian, tetapi seharusnya hanya ada satu konsep yang dinyatakan. Nilai dalam hal ini adalah jika pengujian mulai gagal, Anda harus tahu dari namanya secara pasti mengapa pengujian tersebut gagal. Jika ada konsep berbeda yang diperiksa dalam satu pengujian, beberapa kegagalan dapat menutupi yang lain, jadi Anda mungkin tidak mendapatkan gambaran lengkap tentang mengapa suatu perubahan telah menyebabkan kemunduran hingga nanti.

Dalam proses menjalankan siklus red-green-refactor, Anda akan mendapatkan rangkaian pengujian otomatis yang komprehensif. Dengan rangkaian seperti ini, Anda dapat menjawab pertanyaan kedua: apakah perubahan ini merusak sesuatu? Rangkaian pengujian yang lulus seharusnya memberi Anda tingkat keyakinan yang tinggi. Orang dapat berpendapat bahwa rangkaian pengujian yang baik seharusnya memberi Anda tingkat keyakinan ini, tetapi rangkaian pengujian yang dikembangkan menggunakan TDD memungkinkan Anda membangun keyakinan ini sejak awal; melihat pengujian berubah dari merah menjadi hijau memberi tahu Anda bahwa pengujian tersebut berfungsi.

Tidak melakukan pengujian otomatis sama sekali adalah praktik yang sangat berbahaya. Melewatkan pengujian sepenuhnya sering kali merupakan risiko yang terlalu tinggi untuk diterima oleh organisasi mana pun, tetapi tanpa pengujian otomatis, QA manual diperlukan, dan terlalu lambat dan mahal untuk sifat organisasi digital yang cepat berubah. Oleh karena itu, otomatisasi pengujian, terutama untuk pemeriksaan hafalan, merupakan bagian penting dari pengiriman produk modern.

7.2 ALTERNATIF UNTUK PENGUJIAN UNIT

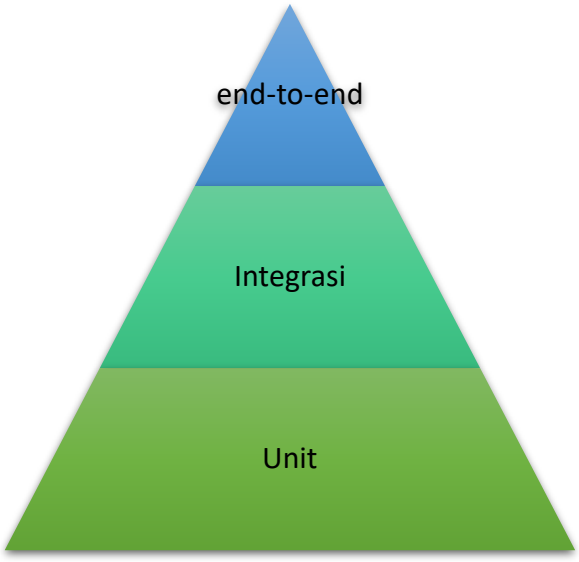
Bahasa pemrograman fungsional Haskell menggunakan pendekatan alternatif untuk pengujian unit selain metode arrange-act-assert yang dijelaskan di atas. Alih-alih menulis kode dengan contoh eksplisit untuk input dan output yang diharapkan dari fungsi tertentu, Anda malah menulis pernyataan yang seharusnya benar untuk seluruh rentang input. Pustaka pengujian (dalam kasus Haskell, ini disebut QuickCheck) kemudian secara acak menghasilkan input dari rentang yang diizinkan untuk mencoba dan menemukan kasus yang gagal.

Misalnya, saat menguji algoritme pengurutan, Anda mungkin ingin menentukan bahwa, saat input berupa array bilangan bulat, setiap item dalam array tersebut kurang dari atau sama dengan elemen berikutnya. Ini dikenal sebagai "pengujian berbasis properti" dan khususnya berguna saat mengembangkan aspek kode yang lebih algoritmik (seperti algoritme pengurutan), di mana terdapat sejumlah besar kasus tepi dalam data input.

Piramida Pengujian

Pengembangan berbasis pengujian merupakan cara efektif untuk mengembangkan rangkaian pengujian yang dapat membantu melindungi Anda dari kerusakan yang tidak disengaja, tetapi ada banyak cara berbeda yang dapat menyebabkan sistem rusak yang tidak tercakup dalam pengujian unit yang dikembangkan menggunakan TDD. Meskipun suatu kelas individual mungkin berperilaku dengan benar menurut pengujian, tidak ada jaminan bahwa kelas lain yang bergantung pada kelas ini membuat asumsi yang benar tentang bagaimana kelas tersebut berperilaku, atau jika ada perubahan yang menyebabkan hilangnya tanggung jawab. Karena itu, penting untuk menguji di berbagai tingkatan dalam aplikasi Anda, dari unit terkecil (yang dapat berupa fungsi atau kelas individual) hingga modul individual, dan bahkan seluruh sistem.

Pada akhirnya, yang menjadi perhatian organisasi hanyalah apakah sistem secara keseluruhan berfungsi atau tidak, dan mungkin sangat menggoda untuk menulis sebagian besar pengujian Anda pada level ini. Namun, ada sisi buruk dari pengujian sistem secara keseluruhan. Pengujian sistem secara keseluruhan sering kali jauh lebih lambat daripada menjalankan unit individual secara terpisah, dan ketika ada ketergantungan eksternal, pengujian dapat menjadi rapuh karena keadaan di luar kendali langsung sistem. Piramida pengujian yang ditunjukkan pada Gambar 7.1 menyoroti berbagai lapisan yang akan diuji. Ketika pengujian Anda dipertimbangkan di semua level piramida, Anda akan memiliki tingkat keyakinan yang tinggi terhadap kinerja aplikasi Anda.



Gambar 7.1 Piramida Pengujian, Dengan Tinggi Menunjukkan Tingkat Abstraksi Dan Lebar Menunjukkan Cakupan Di Setiap Tingkat

Di bagian bawah piramida terdapat pengujian unit. Pengujian ini menguji satu unit secara terpisah tetapi menguji setiap sistem secara menyeluruh. Dalam pengujian unit, penting untuk menguji hanya satu sistem secara terpisah. Namun, banyak kelas akan memiliki ketergantungan pada kelas lain, dan dalam kasus ini, teknik yang dikenal sebagai mocking digunakan. Banyak pustaka pengujian memiliki dukungan bawaan untuk mock, yang merupakan objek yang memiliki antarmuka yang sama dengan ketergantungan. Mock dapat digunakan untuk memastikan bahwa objek berinteraksi dengan cara yang benar, atau mengembalikan respons siap pakai untuk menguji dengan mudah tetapi menyeluruh bahwa suatu kelas berperilaku dengan benar dalam semua kondisi.

Dependency injection adalah teknik di mana sebuah kelas (atau fungsi) diberi dependensinya sendiri, bukan membuat instance dependensi itu sendiri. Saat menggunakan dependency injection, saat sistem sedang diuji, alih-alih instance dependensi nyata yang diberikan ke kelas (misalnya, dalam konstruktornya), tiruan diberikan sebagai gantinya. Namun, ada kalanya tidak semua dependensi dapat dibuat tiruannya misalnya, kelas yang harus membaca file dari disk, atau bertanggung jawab untuk benar-benar membuat koneksi basis data. Penting untuk membuat kelas-kelas ini sekecil mungkin, sehingga mereka tidak lebih dari sekadar pembungkus dependensi eksternal. Kemudian, untuk kelas-kelas yang membutuhkan fungsionalitas itu, tiruan dapat digunakan sebagai gantinya. Saat menguji unit-unit ini, level pengujian integrasi dapat digunakan.

Pengujian integrasi digunakan untuk memastikan bahwa sekelompok modul benar-benar berinteraksi bersama dengan cara yang diharapkan. Saat menggunakan tiruan, mudah untuk memperkenalkan asumsi ke tiruan yang tidak mencerminkan cara kerja dependensi yang sebenarnya. Ini terutama berlaku dalam bahasa yang tidak memiliki pengetikan yang kuat, seperti JavaScript, Ruby, atau Python. Pengujian integrasi memungkinkan pengujian untuk dijalankan terhadap contoh nyata dari dependensinya, seperti file nyata pada disk, basis

data nyata, atau dependensi lainnya. Namun, pengujian integrasi tidak perlu selengkap pengujian unit. Daripada menguji setiap kemungkinan konfigurasi input dan output, nilai dari pengujian integrasi berasal dari memastikan asumsi pada interaksi yang telah ditiru sudah benar, jadi hanya pengujian untuk interaksi ini yang harus diuji.

Pengujian integrasi tidak secepat pengujian unit, karena pengujian tersebut mungkin harus melakukan lebih banyak perhitungan daripada mengembalikan jawaban yang sudah disiapkan, atau harus berkomunikasi dengan basis data nyata. Terlepas dari itu, pengujian tersebut masih cukup cepat. Pengujian unit individual dapat diperkirakan memakan waktu kurang dari beberapa milidetik untuk diselesaikan, sedangkan pengujian integrasi dapat memakan waktu puluhan atau ratusan milidetik. Jenis pengujian yang paling lambat adalah pengujian menyeluruh, di mana setiap pengujian berpotensi memakan waktu beberapa detik untuk diselesaikan. Pengujian menyeluruh menawarkan nilai yang signifikan, dan karena pengujian tersebut setara dengan pengujian yang dapat dijalankan manusia, mungkin tergoda untuk menjalankannya dalam jumlah besar.

Ini adalah kesalahan umum bagi tim yang memiliki seseorang dalam peran QA tradisional yang mulai menggunakan otomatisasi pengujian. Pengujian menyeluruh berinteraksi dengan sistem dengan cara yang sama seperti yang dilakukan pengguna. Untuk API, ini dapat dilakukan dengan hanya membuat permintaan HTTP dari program terpisah, tetapi untuk situs web, ini dapat melibatkan penggunaan browser tanpa kepala (jenis browser yang dapat dikontrol secara terprogram, tetapi tidak perlu menunjukkan representasi visual halaman web kepada manusia). Namun, browser tanpa kepala tidak benar-benar mewakili cara pengguna berinteraksi dengan situs, terutama karena tidak ada browser dengan implementasi standar HTML yang sempurna. Alat populer yang disebut Selenium memungkinkan browser web nyata untuk dikontrol dari jarak jauh, dan untuk menanyakan status halaman web guna membuat pernyataan tentang status tersebut. Merupakan hal yang umum juga untuk menjalankan rangkaian pengujian yang sama terhadap browser yang berbeda untuk memastikan kompatibilitas lintas-browser.

Karena pengujian ini berinteraksi seperti pengguna sungguhan, pengujian ini memberikan tingkat keyakinan tertinggi. Namun, pengujian ini juga yang paling lambat. Anda harus menulis pengujian sesedikit mungkin pada level ini, tetapi cukup untuk mencakup bagian terpenting dari sistem, mirip dengan pendekatan yang diambil dengan pengujian integrasi versus pengujian unit. Misalnya, jika Anda memiliki formulir HTML, penting untuk menguji kasus keberhasilan dan kegagalan pada level ujung ke ujung, tetapi pengujian setiap validasi tunggal, dan setiap cara di mana validasi dapat gagal, paling baik dilakukan pada level piramida yang lebih rendah, seperti level pengujian unit. Pengujian integrasi dan unit berinteraksi dengan kelas secara langsung dan harus ditulis dalam bahasa yang sama, dan berinteraksi langsung dengan kode sistem yang diuji.

Hal yang sama tidak berlaku untuk pengujian ujung ke ujung. Tidak jarang melihat pengujian ujung ke ujung ditulis dalam bahasa yang berbeda dari aplikasi (misalnya, menggunakan Ruby dengan Capybara untuk menjalankan browser), dan pengujian ujung ke ujung dapat tumbuh dalam kompleksitas yang signifikan. Sebuah teknik yang dikenal sebagai

Page Object Model memungkinkan Anda untuk mengatur kode Anda, yang dapat menyederhanakan pengujian yang sebenarnya. Dalam proses ini, lapisan abstraksi ditempatkan di antara browser dan pengujian, dalam bentuk kelas yang mengabstraksikan interaksi antar halaman. Teknik ini sangat membantu dalam menjelaskan tujuan pengujian secara tepat, serta mengurangi duplikasi pengenal seperti pemilih CSS, yang dapat membuat pengujian menyeluruh sangat terkait dengan detail implementasi halaman yang diuji.

Dengan menggunakan Page Object Model, Anda membuat lapisan abstraksi di atas UI untuk menyembunyikan detail implementasi, lalu membuat pengujian menggunakan lapisan abstraksi tersebut. Ini memungkinkan Anda mengubah detail implementasi UI tanpa merusak sejumlah besar pengujian sekaligus, karena Anda dapat melakukan refaktor lapisan abstraksi pada saat yang sama, serta menambahkan kejelasan pada kode pengujian. Contoh di bawah ini menunjukkan cara menggunakan Page Object Model bersama Capybara di Ruby.

```
class HomePage
  include Capybara::DSL

  def visit!
    visit '/taster/' if category.nil?
  end

  def pilot_filter
    PilotFilter.new
  end
end

class PilotFilter
  include Capybara::DSL

  def visible?
    not container.nil?
  end

  def opened?
    container.find('.taster-filter_list', visible: :all).visible?
  end

  def open
    page.execute_script("$('.taster-filter').focusin()")
  end

  private

  def container
```

```
      first('.taster-filter')
    end
  end

    describe 'Pilot Filter' do
      before do
        @home_page = HomePage.new
        @home_page.visit!
    end

    it 'should appear on the home page', smoke: true do
      expect(@home_page.pilot_filter).to be_visible
    end

    it 'starts off closed' do
      expect(@home_page.pilot_filter).not_to be_opened
    end

    it 'opens when selected' do
      @home_page.pilot_filter.open
      wait_for(@home_page.pilot_filter).to be_opened
    end
  end
```

7.3 PENGEMBANGAN BERBASIS PERILAKU

Saat harus menjawab pertanyaan pertama apakah perubahan berhasil atau tidak pengujian otomatis yang diperkenalkan oleh pengembang sering kali tidak cukup. Masalah utamanya adalah pengembanglah yang mendefinisikan perubahan itu dan apakah perubahan itu berhasil atau tidak. Ini tidak masalah jika pengembang dan organisasi sepakat sepenuhnya tentang seperti apa perubahan yang berhasil, dan seperti apa perubahan yang diinginkan, tetapi ini, tidak mengherankan, jarang terjadi. Seperti yang dibahas sebelumnya, banyak perubahan yang diinginkan di tingkat organisasi sebenarnya cukup abstrak, tetapi fungsi produk dan UX tim harus memiliki gagasan yang cukup baik tentang tindakan konkret apa yang diperlukan untuk menerapkan perubahan tersebut, dan tindakan konkret itulah yang menjadi tanggung jawab tim pengembangan.

Dalam dunia proyek waterfall, persyaratan sering kali ditentukan dalam dokumen Word besar yang dikenal sebagai dokumen persyaratan fungsional, dengan pengenal persyaratan. Dokumen-dokumen ini diserahkan kepada tim pengembangan, yang melakukan apa yang menurut mereka diminta untuk mereka lakukan, dan kemudian ke tim pengujian terpisah, yang membuat kasus dan skenario pengujian yang memenuhi setiap persyaratan. Masalah utama dengan pendekatan ini adalah kecepatannya, pengecekan bahwa pemahaman pengembang dan penguji tentang persyaratan selaras dilakukan di akhir proyek. Sebagian besar kerangka kerja tangkas mencoba memperkenalkan gagasan tentang pemahaman

bersama di awal pembuatan fitur, dan pengembangan berbasis perilaku (BDD) adalah teknik khusus yang dapat membantu hal ini. Seperti TDD, BDD sering disalahpahami, sebagian besar karena semakin banyaknya alat BDD.

Menggunakan alat BDD seperti Cucumber, tanpa sepenuhnya memahami proses BDD lainnya, dapat menjadi cara yang lebih rumit dalam melakukan pengujian unit tanpa memberi Anda manfaat tambahan apa pun. Banyak alat BDD didasarkan pada gagasan spesifikasi yang dapat dieksekusi spesifikasi yang ditulis oleh non-coder bertanggung jawab untuk mengeksekusi pengujian Anda. Spesifikasi yang dapat dieksekusi ini kemudian diurai dan dicocokkan dengan aturan tertentu, yang mengarah pada eksekusi kode yang sebenarnya. Ada banyak masalah dengan pendekatan ini. Bahasa alami adalah proksi yang buruk untuk kode, jadi Anda akan berakhir dengan banyak prosa ambigu yang sulit diurai, atau, untuk mempermudah penulisan parser, Anda membatasi bahasa spesifikasi, yang dapat membuat frustrasi perwakilan bisnis.

Spesifikasi yang dapat dieksekusi dirancang untuk membantu perwakilan bisnis dan tim pengembangan berkomunikasi, dan untuk memastikan bahwa pengujian benar-benar mencakup semua persyaratan organisasi. Namun, ada cara lain untuk mencapai tujuan yang sama ini. Proyek Waterfall sering kali menggunakan pengenal persyaratan dan konsep keterlacakan untuk memastikan cakupan yaitu, Anda harus dapat melihat persyaratan mana yang menyebabkan beberapa pengujian dilakukan, dan bahwa ada pengujian yang cukup untuk mencakup suatu persyaratan. Ini kemudian dapat diaudit untuk memastikan cakupan. Dengan menyederhanakan proses ini hingga ke hal-hal penting, kita dapat menggunakannya kembali dalam proyek tangkas dan mendapatkan yang terbaik dari kedua hal tersebut.

Dengan memberikan setiap kriteria penerimaan sebuah ID, lalu memberikan komentar kode pada setiap pengujian otomatis yang menerapkan pemeriksaan untuk kriteria yang ditentukan dalam ID tersebut, kita dapat mempertahankan keterlacakan. ID ini dapat dibuat dengan mudah. Saya sering menggunakan daftar bernomor pada kartu cerita (baik di bagian belakang kartu fisik atau pada sistem seperti Jira), dan kemudian ID dapat menjadi sekadar nomor pada poin-poin ditambah nomor tiket misalnya, #24/6 untuk poin-poin ke-6 pada kartu ke-24. Alih-alih melakukan audit penuh, tinjauan kode sederhana kemudian dapat digunakan untuk memeriksa ulang cakupan. Memisahkan spesifikasi dari kode pengujian sambil mempertahankan keterlacakan akan mempermudah penulisan pengujian pada tingkat piramida pengujian yang sesuai, yang dapat membantu menghindari banyak masalah yang dialami oleh tim yang mencoba memperkenalkan spesifikasi yang dapat dieksekusi.

Three Amigos

Jadi, bagaimana kita dapat membuat kriteria penerimaan tersebut sejak awal untuk memperoleh pemahaman bersama? Melakukan hal ini dengan benar sebenarnya adalah inti dari BDD, dan kebanyakan orang yang hanya berfokus pada alat mungkin melihat upaya mereka untuk menggunakan BDD sebagai kegagalan karena mereka gagal memahami poin ini. Cara terbaik untuk mencapainya adalah melalui penulisan bersama antara semua pihak yang terlibat. Teknik yang dikenal sebagai "Tiga Sahabat" merupakan cara yang sangat efektif untuk melakukan hal ini. Ketiga sahabat tersebut adalah perwakilan produk, pengembang, dan

penguji. Idenya adalah bahwa perwakilan produk adalah orang yang mengetahui dan memahami masalah, pengembang dapat mengajukan pertanyaan untuk memperoleh pemahaman bersama, dan penguji dapat memandu ketiga sahabat tersebut untuk mengeksplorasi seluruh ruang masalah dan interaksi dengan bagian lain dari sistem untuk memastikan semua bagian tercakup.

Pada kenyataannya, perannya agak lebih kabur, dan bisa jadi lebih dari tiga orang yang terlibat; spesialis bisnis lain, spesialis UX, dan manajer proyek biasanya disertakan. Spesifikasi juga boleh saja berkembang seiring pertanyaan diajukan, dan sepenuhnya dapat diterima bagi pengembang untuk mengusulkan solusi alternatif yang mungkin lebih mudah diterapkan selama esensi persyaratan masih terpenuhi. Bagian terpenting dari pertemuan tiga sahabat adalah, paling tidak, orang-orang yang akan membangun solusi dan pelaku bisnis yang tahu apa yang dibutuhkan, melakukan percakapan bersama, dan diberi wewenang untuk membuat dan menindaklanjuti keputusan dalam pertemuan ini.

Sesi sahabat yang hanya melibatkan penguji dan pengembang akan berakhir dengan sekadar menebak-nebak persyaratan. Demikian pula, sesi di mana sahabat tidak diberi wewenang untuk membuat keputusan akan berakhir dengan permainan tenis komite, di mana proposal akan dibawa ke dewan lain untuk ditinjau dan disetujui, mungkin dengan perubahan tambahan. Seringkali hasil dari sesi amigos adalah spesifikasi yang dapat dieksekusi, tetapi serangkaian kriteria penerimaan bisa sama efektifnya, terutama jika lebih mudah menggunakan ekspresi bahasa alami untuk menyampaikan apa saja harapannya. Namun, kriteria penerimaan ini bukanlah satu-satunya titik acuan.

Pengembang dan penguji yang mengerjakan sebuah cerita mungkin tidak sama seperti dalam sesi amigos, tetapi komunikasi dapat membantu, terutama karena menghindari ambiguitas dalam bahasa alami itu sulit! Akan sangat membantu untuk membuat glosarium di halaman wiki dari waktu ke waktu, sehingga para pemangku kepentingan memahami apa arti istilah tertentu. Ini dapat sangat berguna jika ada nama yang cukup umum yang digunakan (misalnya, sebuah aplikasi mungkin memiliki halaman "tentang" untuk aplikasi tersebut secara umum, dan halaman "info" untuk setiap item tertentu).

Percakapan sama pentingnya dengan hasilnya. Oleh karena itu, seringkali tidak membantu untuk mengembangkan kriteria penerimaan untuk sebuah cerita jauh sebelum pembuatan dimulai, sebagian karena kebutuhan bisnis mungkin telah berubah, dan sebagian karena orang-orang melupakan diskusi tersebut. Menetapkan diskusi tersebut dalam bentuk kode adalah tujuan utamanya. Salah satu teknik populer untuk menentukan kriteria penerimaan adalah dengan menggunakan bentuk spesifikasi Given-When-Then. Bentuk ini dimulai dengan menyiapkan prasyarat atau asumsi untuk bagian tertentu dari cerita, lalu menentukan tindakan apa yang diambil pengguna, lalu prasyarat pasca, atau apa yang seharusnya terjadi saat pengguna melakukan tindakan tersebut.

- Diberikan keranjang belanja saya kosong
- Saat saya menambahkan produk yang tersedia ke keranjang belanja saya
- Maka keranjang belanja seharusnya menunjukkan bahwa keranjang tersebut tidak kosong

Jika Anda menggunakan pendekatan ini, Anda dapat dengan mudah terjebak dalam spesifikasi yang berlebihan (ini terutama berlaku saat Given-When-Then digunakan sebagai dasar untuk spesifikasi yang dapat dieksekusi). Skenario di atas dapat dianggap sebagai spesifikasi tingkat tinggi yang masih menangkap esensi dari apa yang dibutuhkan. Saat dikombinasikan dengan serangkaian wireframe dari seorang desainer, sering kali akan menunjukkan dengan tepat apa yang dibutuhkan. Contoh berikut menunjukkan skenario yang sama tetapi terlalu banyak spesifikasinya.

- Jika keranjang belanja saya kosong
- Dan ada produk yang tersedia di basis data
- Ketika saya mengunjungi halaman produk yang tersedia
- Maka saya akan melihat tombol “Tambahkan ke Keranjang”
- Ketika saya mengklik tombol “Tambahkan ke Keranjang”
- Maka ikon keranjang belanja akan berubah untuk menunjukkan keranjang penuh
- Dan angka ‘1’ akan muncul pada ikon keranjang belanja

Masalah dengan spesifikasi yang berlebihan seperti ini adalah akan menyulitkan untuk melihat esensi skenario, yang dapat melepaskan pemangku kepentingan bisnis. Gaya alternatif untuk Diberikan-Ketika-Lalu adalah dengan menggunakan contoh konkret. Saya pribadi lebih suka yang lebih abstrak, tetapi itu hanya masalah preferensi. Skenario di atas dapat diungkapkan seperti ini:

- Jika keranjang belanja saya kosong
- Dan wortel tersedia
- Ketika saya menambahkan wortel ke keranjang belanja saya
- Maka keranjang belanja akan menunjukkan bahwa keranjang belanja tersebut tidak kosong

Bagi tim yang mempertimbangkan untuk mengadopsi BDD, penting untuk memulai dengan esensinya dan kemudian memilih alat berdasarkan kebutuhan mereka. Anda dapat melakukan BDD tanpa menggunakan Cucumber, dan Anda dapat melakukan BDD tanpa menggunakan format Given-When-Then untuk kriteria penerimaan Anda. Lakukan eksperimen dan temukan apa yang paling cocok untuk tim Anda!

7.4 PENGUJIAN MANUAL

Terlepas dari seberapa banyak Anda mengotomatiskan, akan selalu ada kebutuhan untuk beberapa bentuk QA manual. Otomatisasi menghilangkan rutinitas pengujian sederhana (apakah mengklik tombol ini berfungsi dengan benar?), yang memungkinkan penguji untuk fokus pada perilaku yang lebih kompleks dan muncul. Ini dapat berupa pemeriksaan kewarasan cepat pada fitur baru, atau upaya untuk mereproduksi beberapa perilaku tak terduga yang dilaporkan pengguna, atau bantuan untuk mengembangkan pengujian otomatis (jika pengujian otomatis gagal, apakah pengujian manual berhasil, atau saat mengembangkan pengujian otomatis baru, mengetahui langkah-langkah otomatis dapat diperoleh dari pengujian manual pertama).

Kerumitan terbesar dalam suatu sistem berasal dari interaksi antara komponen. Ketika

seluruh sistem diuji bersama-sama, komponen-komponen tersebut dapat berinteraksi dengan cara yang tidak terduga, yang menimbulkan perilaku yang tidak terduga. Perilaku ini sering kali perlu dikendalikan, dan mengidentifikasinya adalah keunggulan pengujian manual. Dua bagian utama pengujian manual adalah pengujian fungsional dan pengujian regresi. Pengujian fungsional memeriksa bahwa setiap perubahan yang telah dibuat berperilaku seperti yang diharapkan, dan masuk akal dalam konteks keseluruhan; pengujian regresi memeriksa apakah fungsionalitas yang sudah ada sebelumnya tidak rusak oleh fitur baru. Pengujian regresi adalah area di mana otomatisasi dapat memberikan dampak paling besar, karena ini adalah jenis pengujian yang paling berulang. Otomatisasi sendiri seringkali tidak cukup, dan di sinilah kolaborasi erat antara pengembang dan penguji dapat membantu. Pengembang akan menyadari jalur kode mana yang mungkin telah mereka sentuh dalam mengimplementasikan fitur (yang terkadang tidak jelas), yang memungkinkan pengujian untuk fokus pada area yang paling mungkin telah diubah.

Penguji biasanya menulis rencana pengujian yang menjabarkan apa yang akan mereka uji dan pendekatan mereka untuk melakukannya, dan secara historis dokumen ini bisa sangat besar. Kasus pengujian tertentu mungkin menguraikan langkah-langkah yang harus diikuti dan respons yang diharapkan, tetapi pada tingkat itu menjadi agak mirip dengan penulisan kode, jadi tingkat detail ini sekarang sering dinyatakan dalam kode. Terutama saat menggunakan pengembangan yang digerakkan oleh perilaku, rencana pengujian dapat merujuk kriteria penerimaan tersebut dan fokus pada pendekatan tingkat tinggi, jika rencana pengujian memang ada. Dulu, tim penguji khusus biasanya terpisah dari tim pengembangan, yang menerima hasil pengembangan di akhir sprint, lalu menghabiskan sprint pengujian untuk menguji rilis sebelumnya hal ini menyebabkan siklus umpan balik yang sangat lambat. Pendekatan yang lebih modern adalah dengan menempatkan penguji bersama tim pengembangan dan bekerja bersama pengembang, menguji fitur secara bertahap saat setiap komponen dikembangkan. Bug muncul dalam sprint yang sama dan dapat diperbaiki saat pekerjaan masih segar dalam ingatan pengembang.

Model yang lebih disfungsional dapat melibatkan pengalihdayaan fungsi seperti pengujian regresi ke fasilitas luar negeri tempat penguji cukup mengikuti skrip pengujian yang dinyatakan dalam rencana pengujian. Pengujian semacam ini cocok untuk diotomatisasi. Otomatisasi pengujian mungkin tampak menghilangkan kebutuhan akan penguji, tetapi tanggung jawab dan pola pikir penguji masih memainkan peran penting dalam tim modern. Untuk tujuan tersebut, cukup umum bagi penguji untuk menyebut diri mereka sebagai QA, atau analis kualitas, yang mengambil peran yang lebih mirip dengan analis bisnis daripada penguji tradisional. Perbedaan peran ini paling baik dicontohkan dalam situasi tiga sahabat, di mana pandangan kritis dan perspektif penguji dapat membantu menyempurnakan kriteria penerimaan tambahan. Area lain tempat QA dapat membantu adalah pengujian "eksploratif".

Dalam pengujian eksploratif, QA mengambil pola pikir pengguna, dan melakukan pengujian menyeluruh dari berbagai bagian fungsionalitas, memastikan bahwa semuanya terjadi seperti yang diharapkan (bukan hanya seperti yang ditentukan). QA harus memahami produk dan pelanggannya agar dapat melakukan ini secara efektif, dan meskipun menguji

"jalur yang menyenangkan" (kasus penggunaan paling umum yang diharapkan diikuti pengguna) berguna, QA juga harus menyimpang dari jalur ini jika sesuai dan melakukan hal-hal yang mungkin tampak aneh, tetapi mungkin merupakan sesuatu yang akan dilakukan oleh pengguna yang bingung. Pengujian eksploratif semacam ini, terutama jika dikombinasikan dengan pengujian pengguna dalam proses UX, adalah tempat pengujian memberikan nilai tambah yang signifikan. Bukan hanya bug yang dapat ditemukan, tetapi juga spesifikasi yang kurang memadai.

7.5 PENGUJIAN VISUAL

Area lain yang menjadi keunggulan pengujian manual adalah "melihat" halaman untuk memastikannya efektif secara visual. Telah banyak upaya untuk mengotomatiskan proses ini, dengan berbagai tingkat keberhasilan, dan tidak ada solusi umum yang memenuhi pertanyaan, "apakah halaman web ini terlihat seperti yang seharusnya?". Selenium berguna untuk menguji apakah konten muncul di halaman dan terlihat atau tidak, dan dimungkinkan untuk menulis pengujian yang membuat pernyataan mengenai posisi (dalam hal koordinat x dan y) elemen tertentu di halaman, tetapi pengujian ini akan sangat rapuh. Perbedaan kecil antara versi browser, atau font yang tersedia, dapat menimbulkan kesalahan kecil, dan perubahan seperti memperkenalkan komponen baru di halaman dapat menyebabkan sejumlah besar pengujian diperbarui.

Pengujian yang tidak dapat diandalkan akan sering diabaikan, dan dikombinasikan dengan jumlah pekerjaan yang diperlukan untuk mempertahankan pengujian tersebut, biasanya berarti tidak ada gunanya menulisnya. Oleh karena itu, mata manusia adalah alat terbaik untuk memverifikasi tampilan dan nuansa halaman web, tetapi seiring berkembangnya aplikasi web, menjalankan uji regresi bisa jadi melelahkan jika semuanya dilakukan secara manual. Untungnya, ada alat yang dapat membantu di sini, meskipun alat tersebut sering kali memiliki biaya perawatan yang tinggi dan tidak sering digunakan. Ide umumnya adalah mengambil seluruh halaman atau komponen individual pada suatu halaman dan mengambil tangkapan layar yang "diketahui baik" dari halaman atau komponen tersebut.

Saat perubahan dilakukan, pengujian kemudian dijalankan kembali dan tangkapan layar baru diambil. Jika ada tangkapan layar yang berubah, maka pengguna alat pengujian akan diberi tahu, dan menyetujui perubahan tersebut atau menandai regresi. Penerapan alat ini harus selalu dilakukan dengan hati-hati, karena alat ini sering kali akan menandai hasil positif yang salah.

7.6 PENGUJIAN LINTAS FUNGSIONAL

Sejauh ini, kita telah membahas cara memeriksa bagaimana aplikasi Anda berperilaku dan apakah aplikasi tersebut memenuhi persyaratan yang ditetapkan padanya. Pengujian semacam ini sering disebut "pengujian fungsional", tetapi ada atribut lain dari aplikasi Anda yang ingin Anda uji. Persyaratan ini tidak terkait dengan fitur (atau fungsi) tertentu, tetapi menangani atribut aplikasi Anda secara keseluruhan. Jenis persyaratan dalam kategori ini sering disebut "non-fungsional" (karena tidak terkait dengan satu fungsi aplikasi Anda), atau,

mungkin tidak terlalu membingungkan, "lintas-fungsional" (karena mencakup setiap bagian aplikasi Anda). Menguji persyaratan lintas-fungsional ini sama pentingnya dengan menguji fungsi eksplisit aplikasi Anda. Jenis hal yang dapat Anda klasifikasikan sebagai lintas-fungsional meliputi keamanan, kegunaan, aksesibilitas, kinerja, kemampuan menangani beban, dan kompatibilitas perangkat. Sebagian besar, jenis persyaratan lintas-fungsional ini umum untuk banyak aplikasi web, meskipun beberapa detailnya mungkin berbeda.

Karena itu, ada banyak perangkat umum yang tersedia untuk membantu Anda menguji aplikasi, dan banyak di antaranya hanya memerlukan konfigurasi yang tepat. Namun, perangkat ini tidak selalu diperlukan. Misalnya, jika Anda menetapkan persyaratan bahwa panggilan API akan kembali dalam waktu kurang dari 200 ms, maka Anda dapat menulis permintaan HTTP yang dibungkus dengan pengatur waktu dan membuat pernyataan pada pengatur waktu tersebut, seperti pengujian lainnya. Namun, jika persyaratan Anda adalah halaman dimuat dan dirender ke status yang dapat digunakan dalam waktu kurang dari 2 detik, maka menulis pengujian untuk melakukan hal ini menjadi lebih rumit karena kompleksitas rendering.

Alat pengujian kinerja dapat dikonfigurasi untuk membuat jenis pernyataan ini. Untuk persyaratan lintas fungsi lainnya, seperti keamanan dan aksesibilitas, alat dapat membantu, tetapi seperti pengujian fungsional, sering kali memerlukan pendekatan khusus, dan tidak ada peluru ajaib otomatisasi. Pendekatan pengujian umum untuk faktor-faktor semacam ini sering kali sangat mirip dengan pengujian normal, tetapi dilakukan di seluruh produk (atau berfokus pada tempat perubahan terjadi) dan mungkin memerlukan keterampilan khusus. Detail selengkapnya tentang cara memeriksa aksesibilitas dengan cepat dapat ditemukan di bab Aksesibilitas, tetapi pemeriksaan keamanan sederhana dapat dilakukan dengan cara yang sama seperti memeriksa fitur lain dengan mencoba kasus pengujian negatif dan memeriksa apakah fitur tersebut berfungsi seperti yang diharapkan.

Bab Keamanan akan membahas beberapa aspek keamanan yang perlu diperhatikan, yang dapat membantu Anda mengembangkan kasus pengujian, tetapi pengujian keamanan sering kali mengharuskan Anda mencoba cara yang sangat tidak jelas untuk merusak aplikasi, yang dapat diuntungkan dari keterampilan khusus untuk aplikasi dengan persyaratan keamanan yang sangat ketat. Seperti jenis pengujian lainnya, pengujian non-fungsional seperti keamanan dan aksesibilitas tidak membuktikan tidak adanya bug, tetapi pengujian kinerja dan beban beroperasi dengan cara yang sama sekali berbeda. Setiap kali Anda menjalankan pengujian ini, Anda selalu mendapatkan hasil yang dapat dievaluasi terhadap tolok ukur, daripada sekadar skenario "lulus" atau "gagal". Jenis pengujian ini juga sangat spesifik lingkungan, yang dapat membuatnya sulit dijalankan secara teratur sebagai bagian dari siklus pengembangan normal Anda.

Pengujian beban (kadang disebut pengujian volume) adalah jenis pengujian yang mana aplikasi diberi banyak lalu lintas simulasi untuk memeriksa apakah aplikasi berperilaku dengan benar saat dibebani. Namun, pengujian beban yang dijalankan pada lingkungan pengembangan lokal mungkin tidak memberikan hasil yang berarti, karena perangkat keras tempat aplikasi berjalan mungkin sangat berbeda. Beberapa organisasi mengonfigurasi

lingkungan pengujian mereka agar memiliki perangkat keras yang sama dengan lingkungan langsung mereka agar lingkungan ini dapat digunakan untuk pengujian beban, tetapi ini bisa sangat mahal. Komputasi awan dapat membantu dalam hal ini, memungkinkan Anda untuk menjalankan lingkungan yang mencerminkan produksi untuk pengujian beban. Beberapa organisasi merasa cukup percaya diri untuk menjalankan pengujian beban mereka pada lingkungan langsung mereka tanpa memengaruhi pelanggan mereka secara negatif.

Pengujian beban sering kali dilakukan dalam skala besar, dan oleh karena itu menggunakan metrik yang dapat diukur untuk menentukan keberhasilan atau kegagalannya. Untuk menetapkan metrik ini, Anda harus terlebih dahulu menentukan kinerja dasar untuk aplikasi Anda, yang merupakan cara kerjanya saat tidak dibebani apa pun sering kali dengan satu pengguna. Anda kemudian harus menentukan tingkat penggunaan aplikasi yang umum sering kali berdasarkan metrik dunia nyata jika memodifikasi sistem yang ada, atau perkiraan terbaik berdasarkan apa yang diketahui organisasi dan penyimpangan yang dapat diterima dari tingkat ini. Hal ini dapat dinyatakan dalam bentuk persentase peningkatan, atau target tertentu jika metrik yang pasti diketahui. Apakah suatu sistem lulus atau gagal, uji beban kemudian dievaluasi berdasarkan metrik ini.

Uji beban juga dapat dimodifikasi untuk menguji "hingga kerusakan", di mana jumlah permintaan atau volume lalu lintas meningkat hingga sistem benar-benar gagal, untuk memberikan indikasi yang baik tentang seberapa banyak ruang gerak yang dimiliki sistem Anda. Dengan pengujian semacam ini, sering kali berguna untuk melacak hasil pengujian dari waktu ke waktu guna mengidentifikasi tren, serta memeriksa apakah ada yang lolos/gagal terhadap tolok ukur. Karena lingkungan tidak dapat diprediksi saat menjalankan pengujian beban dan kinerja, sering kali perlu mengulangi pengujian untuk mendapatkan hasil yang berarti. Banyak alat yang dapat melakukannya untuk Anda misalnya, alat pengujian kinerja akan sering kali merender halaman yang sama 100 kali dan mengambil nilai rata-rata yang akan memberi Anda keyakinan lebih daripada menjalankannya hanya sekali.

Ada baiknya juga untuk memeriksa persyaratan ini dengan menganalisis perilaku pengguna sebenarnya, terutama untuk pengujian beban dan kinerja. Dengan memantau kinerja pengguna sebenarnya, Anda bisa mendapatkan gambaran yang jauh lebih baik tentang kinerja situs Anda yang sebenarnya daripada dengan mensimulasikan pengujian beban (meskipun pengujian beban tetap penting untuk mendapatkan keyakinan awal). Pemantauan hal-hal seperti waktu respons server dan CPU dibahas lebih rinci dalam bab Dalam Produksi. Untuk menguji kinerja kode front-end Anda, serangkaian alat pengujian yang dikenal sebagai RUM (real user monitoring) memungkinkan Anda untuk menangkap analitik pada statistik dari perangkat nyata, yang mungkin jauh lebih rumit tetapi memberi Anda gambaran yang lebih baik tentang kinerja situs Anda pada berbagai perangkat di dunia nyata.

7.7 KESIMPULAN

Pengujian adalah kunci untuk setiap proyek perangkat lunak, dan harus digunakan untuk tidak hanya memastikan bahwa suatu sistem bekerja dengan benar, tetapi juga bahwa sistem yang benar telah dibangun. Beberapa tingkat pengujian dapat dilakukan secara

otomatis dengan menulis kode pengujian yang menjalankan masing-masing komponen sistem secara terpisah (pengujian unit), atau disatukan (pengujian integrasi) dalam berbagai status, dan menegaskan bahwa respons atau tindakannya benar.

Pengembangan berbasis pengujian adalah mekanisme untuk menggunakan pengujian guna membantu menyusun pengembangan Anda. Pengujian ditulis terlebih dahulu, kemudian kode yang diperlukan untuk melewati tahap kedua, akhirnya setelah fungsionalitas terbukti benar, kode tersebut dirapikan (refactoring). Hal ini memungkinkan masalah dipecah menjadi bagian-bagian kecil yang dapat diuji secara individual, dan dapat menjadi rangkaian yang efektif untuk memeriksa bahwa tidak ada yang rusak atau mengalami kemunduran secara tidak sengaja. Pengujian manual terjadi saat pengguna menjalankan aplikasi dan memeriksa apakah perilakunya memenuhi spesifikasi dan masuk akal.

Hal ini dapat menerapkan beberapa tingkat otomatisasi, di mana kode pengujian dapat menjalankan aplikasi sebagaimana yang dilakukan pengguna ini dikenal sebagai pengujian menyeluruh. Pengujian manual juga efektif dalam menemukan kesalahan visual, yang dapat terlewatkan oleh pengujian menyeluruh. Pengujian ini juga dapat relatif rapuh terhadap perubahan. Pengujian manual paling baik diterapkan saat elemen yang akan diperiksa tidak jelas atau abstrak, dan kreativitas manusia lebih diutamakan. Pengujian eksploratif juga merupakan sesuatu yang dapat dilakukan manusia, di mana penguji melakukan perjalanan yang mengalir bebas melalui aplikasi untuk menguji skenario dengan cepat, daripada yang telah ditetapkan sebelumnya sebagai persyaratan.

Pengembangan yang digerakkan oleh perilaku membawa pengembangan yang digerakkan oleh pengujian selangkah lebih maju dan menambahkan spesifikasi pada tingkat abstraksi yang lebih tinggi. TDD mungkin berfokus pada pengujian unit, tetapi BDD biasanya melihat pada pengujian integrasi atau menyeluruh. BDD sebagian besar digunakan untuk memastikan bahwa ada pemahaman bersama antara pengembang, penguji, dan perwakilan produk ("tiga sahabat"), yang mendokumentasikan berbagai skenario penggunaan produk, yang sering dinyatakan dalam bentuk Diberikan-Ketika-Lalu. Aspek terakhir pengujian yang perlu dipertimbangkan membahas komponen yang mencakup semua fungsi aplikasi Anda. Ini dapat mencakup pengujian kinerja, celah keamanan, atau kelemahan aksesibilitas. Seperti pengujian fungsional, otomatisasi dapat membantu, tetapi menerapkan keterampilan manusia untuk menguji ini akan membantu menemukan area yang dapat terlewatkan oleh otomatisasi.

BAB 8
JAVASCRIPT

8.1 PERKEMBANGAN JAVASCRIPT: DARI WEB KE SERVER

Dirancang dalam 10 hari, JavaScript memiliki reputasi yang cukup beragam di kalangan pengembang perangkat lunak. Namun, jika Anda melakukan apa pun di web, Anda perlu memiliki beberapa keterampilan JavaScript. JavaScript awalnya dirancang sebagai bahasa untuk memanipulasi halaman web menggunakan API yang dikenal sebagai Document Object Model (DOM). Awalnya dibuat di Netscape, JavaScript dinamai JavaScript sebagai upaya untuk memanfaatkan gelombang kehebohan di balik bahasa Java yang semakin populer, sebuah keputusan yang telah menyebabkan kebingungan bagi pengembang baru selama bertahun-tahun, karena bahasa tersebut memiliki sedikit kesamaan dengan Java.

Hari-hari awal web tidak membantu, dengan Microsoft mengembangkan variannya sendiri, yang dikenal sebagai JScript yang menambahkan fitur-fitur baru, seperti XMLHttpRequest (sekarang menjadi bagian dari bahasa utama, dan umumnya disingkat menjadi XHR), yang memungkinkan pengembang untuk membuat permintaan ke server eksternal untuk menyegarkan informasi di halaman web. Asosiasi Produsen Komputer Eropa (ECMA) akhirnya memutuskan untuk mencoba menggabungkan implementasi yang bersaing ini menjadi standar baru, yang disebut ECMAScript.

Oleh karena itu, apa yang disebut JavaScript saat ini sebenarnya adalah berbagai implementasi ECMAScript, itulah sebabnya Anda mungkin mendengar istilah seperti "ES6" atau "ES2018" digunakan untuk merujuk ke JavaScript (sebelumnya versinya diurutkan berdasarkan nomor, tetapi sekarang berdasarkan tahun ES2018 adalah JavaScript edisi kesembilan). Ini merujuk ke berbagai spesifikasi yang diterbitkan oleh ECMA, yang dipatuhi oleh berbagai browser. Dalam bab ini, saya akan merujuk ke versi ES2018 dari bahasa tersebut. Bagi para pengembang yang pernah bekerja dengan JavaScript sebelumnya, ES6 memperkenalkan beberapa perubahan bahasa yang signifikan, seperti pengenalan fungsi arrow, dengan iterasi tahunan setelah itu memperkenalkan perubahan yang lebih kecil, tetapi tetap signifikan.

JavaScript mungkin berasal dari masa lalu yang terburu-buru, tetapi karena bahasa tersebut terus disempurnakan, beberapa duri terbesar di sisi pengembang (terutama mereka yang berasal dari bahasa lain) menghilang. Namun, beberapa masih ada, dan saya akan menyebutkan masalah tersebut di bawah ini. Selama ini, JavaScript pada peramban hanya digunakan untuk menambahkan penyempurnaan ekstra pada halaman, seperti validasi formulir sebaris atau animasi sederhana. Hal ini sebagian disebabkan karena mesin JavaScript pada peramban tidak cukup cepat untuk mendukung operasi yang rumit; meskipun ada beberapa aplikasi awal yang mendukungnya, mesin tersebut sangat disesuaikan untuk performa dan karenanya rumit untuk dibuat.

Google mengubah semua ini dengan memperkenalkan mesin V8, yang secara signifikan mempercepat JavaScript dan memungkinkan kode yang jauh lebih rumit untuk diperkenalkan

ke halaman. Kemudian, proyek Node diluncurkan, yang memungkinkan JavaScript berjalan di server, bersama dengan pustaka standar baru untuk mendukung persyaratan sisi server. Selain memungkinkan bahasa yang sama di sisi klien dan server, hal ini juga memungkinkan berbagi kode di antara keduanya, memperkenalkan gaya aplikasi baru yang dikenal sebagai JavaScript isomorfik atau universal di mana kode tidak bergantung pada tempat menjalankannya.

Asinkronisitas Buku ini bukan tempat untuk membahas berbagai macam peristiwa dan API yang tersedia di browser, karena peristiwa dan API tersebut berkembang dengan cepat dan sudah ada banyak sumber daya yang fantastis di luar sana. Namun, ada beberapa hal penting yang perlu dipahami. Hal utama yang perlu dipahami tentang JavaScript adalah bahwa JavaScript bersifat single-threaded dan sangat bergantung pada konsep asinkronisitas. Kelemahannya adalah, saat JavaScript dijalankan, runtime "diblokir." Di browser, ini berarti bahwa seluruh browser tidak akan bereaksi terhadap interaksi apa pun dari pengguna saat fungsi JavaScript sedang berjalan. Di server, ini berarti tidak ada koneksi lain yang akan diterima atau ditangani.

Banyak aplikasi JavaScript yang terikat IO, bukan terikat CPU artinya, sebagian besar waktu ini dihabiskan untuk menunggu peristiwa eksternal terjadi, daripada melakukan penghitungan angka yang berat. Dalam lingkungan seperti ini, pendekatan single threading ini sangat masuk akal, karena Node tidak perlu secara aktif menunggu peristiwa eksternal ini terjadi. Ia dapat menjalankan suatu fungsi, kemudian saat fungsi tersebut berakhir, ia dapat menggunakan callback dan event listener untuk berbagai aktivitas asinkron. Saat peristiwa yang diatur oleh pengendali tersebut terjadi, fungsi tersebut dapat dijalankan. Di antara kedua contoh tersebut, callback atau event listener lainnya dapat dijalankan sebagai respons terhadap peristiwa yang terjadi saat fungsi asli berjalan, atau yang terjadi saat tidak ada metode yang aktif.

Hal ini membuat hidup lebih mudah, karena konsep keamanan utas tidak perlu dipertimbangkan, yang menghilangkan seluruh kelas kesalahan dan kompleksitas. Jika Anda menangani panggilan balik, Anda tidak perlu khawatir bahwa tiba-tiba fungsi lain dapat dimulai di tengah konteks Anda dan meninggalkan beberapa status bersama dalam mode yang tidak konsisten atau tidak terduga. Untuk contoh yang lebih konkret tentang apa artinya ini, ambil, misalnya, beberapa kode teoritis untuk membuat permintaan web:

```
    function fetchAndLog(url) {
      const request = new XMLHttpRequest();
      request.open('GET', url, false);
      request.send();
      console.log(request.responseText);
}
```

Masalahnya di sini adalah bahwa mungkin diperlukan waktu beberapa detik agar permintaan selesai, dan selama waktu tersebut pengguna tidak dapat berinteraksi dengan halaman—bagi mereka, halaman tersebut tampak seperti macet. Bahkan ketika permintaan lebih cepat, halaman tersebut tampaknya menjadi lambat. Pendekatan asinkron mungkin terlihat seperti ini:

```
function fetchAndLog(url) {
  const request = new XMLHttpRequest();
  request.open('GET', url, false);
  request.onreadystatechange = () => {
    if (request.readyState === 4) {
      console.log(request.responseText);
    }
  }
  request.send();
}
```

Sementara pengguna menunggu permintaan HTTP terjadi, mereka dapat terus berinteraksi dengan halaman sebagaimana mestinya. Hal ini dapat menyebabkan komplikasi pada antarmuka pengguna. Jika pengguna telah melakukan tindakan yang memerlukan sesuatu yang tidak sinkron, mungkin tidak selalu jelas bahwa tindakan mereka berhasil, jadi mereka mengekliknya lagi. Memperbarui UI untuk menyertakan status "tertunda" atau "sedang berlangsung" yang terputus-putus sebagai respons terhadap suatu tindakan diperlukan untuk menghindari hal ini.

8.2 CALLBACK, PROMISES, ASYNC & AWAIT

Untuk fungsi yang rumit, penumpukan callback sedemikian rupa dapat mengakibatkan "callback hell," di mana terdapat banyak sekali callback yang bertumpuk. Ambil contoh berikut untuk mengunduh file video, mentranskodekannya menggunakan ffmpeg, lalu menghapus file asli.

```
function downloadAndTranscode(url, outputFilename, callback) {
  downloadFile(url, (err, downloadedFilename) => {
    if (err) {callback(err); }
    ffmpegTranscode(downloadedFilename, outputFilename, (err) => {
      if (err) {callback(err); }
      deleteFile(downloadedFilename, (err) => {
        if (err) {callback(err); }
        callback(null);
      });
    });
  });
}
```

Setiap kali operasi asinkron baru terjadi, tingkat penumpukan lebih lanjut ditambahkan, yang dapat membuat kode sulit diikuti. Promise adalah teknik yang dapat menghindari hal ini dengan membangun rantai promise individual yang masing-masing merespons promise sebelumnya. Dalam contoh di atas, kita dapat menulis ulang ini menggunakan promise agar terlihat seperti ini:

```
function downloadAndTranscode(url, outputFilename) {
     return downloadFile(url)
          .then(downloadedFilename =>
           ffmpegTranscode(downloadedFilename, outputFilename))
             .then(() => deleteFile(downloadedFilename))
     )
}
```

Dalam contoh di atas, dengan asumsi bahwa fungsi itu sendiri juga mengembalikan promise, rantai disiapkan. Promise dapat dianggap sebagai placeholder untuk fungsi asynchronous yang akan diselesaikan ke beberapa nilai atau ditolak jika terjadi kesalahan. Objek promise itu sendiri sebenarnya tidak berubah, tetapi pengendali dapat ditambahkan ke dalamnya dengan memanggil `.then()` dengan callback. Nilai pengembalian `.then()` kemudian menjadi promise baru, yang kemudian dapat dirantai lebih lanjut. Promise adalah subjek yang sangat kaya, dan Mozilla Developer Network menyediakan pengantar yang lebih mendalam tentang promise dan cara menggunakannya.

Promise.all() memiliki keuntungan khusus dibandingkan gaya callback penanganan asynchronous sebelumnya, yang memungkinkan beberapa tindakan asynchronous terjadi secara paralel dan kemudian berhasil atau gagal sebagai satu, dengan hasil yang tersedia untuk `then()` berikutnya dalam rantai. async dan await juga merupakan fitur bahasa yang diperkenalkan ke JavaScript yang bekerja di balik layar untuk menangani konstruksi rantai promise dan dapat menyederhanakannya lebih lanjut. Dengan menggunakan async dan await, rantai di atas dapat disederhanakan lebih lanjut menjadi:

```
async function downloadAndTranscode(url, outputFilename) {
  let downloadedFilename = await downloadFile(url);
  await ffmpegTranscode(downloadedFilename, outputFilename);
  await deleteFile(downloadedFilename);
}
```

Masalah utama dengan tindakan asinkron adalah sering kali sulit untuk melakukan refaktor API yang sebelumnya bekerja secara sinkron menjadi API asinkron. Hal ini terkadang diperlukan ketika cara kerja API di balik layar telah berubah menjadi asinkron. Namun, melakukan refaktor API asinkron menjadi API sinkron jauh lebih mudah, karena panggilan balik dapat dipanggil secara sinkron. Dengan mengingat hal ini, sering kali lebih mudah saat mendesain antarmuka dalam JavaScript untuk membuatnya berperilaku asinkron, jika ada kemungkinan apa pun bahwa antarmuka tersebut perlu menjadi asinkron di masa mendatang.

Pendekatan ini dapat menimbulkan bug kecil terkait kondisi pemesanan dan perlombaan. Bahkan jika Anda dapat mengevaluasi fungsi asinkron segera (misalnya, memanggil panggilan balik kesalahan segera pada metode yang membuat permintaan HTTP jika parameternya tidak valid, sedangkan terkadang panggilan balik dipanggil secara asinkron jika harus menunggu respons), masuk akal untuk memaksanya agar selalu berperilaku

asinkron, menggunakan mekanisme seperti `setImmediate()` atau `process.nextTick()` Node.

8.3 JAVASCRIPT DI PERAMBAN

JavaScript dirancang untuk berjalan di peramban web, dan untuk waktu yang lama, hanya di situlah ia berjalan. Namun, JavaScript tidak pernah menjadi bahasa yang disukai banyak orang, dan untuk waktu yang lama, orang-orang telah menggunakan bahasa alternatif untuk Web. Hal ini sebagian disebabkan oleh kekurangan yang dirasakan dalam JavaScript, dan keinginan untuk menstandardisasi pada satu tumpukan teknologi. Hal ini menyebabkan munculnya plugin peramban seperti Flash (yang seharusnya mengatasi yang pertama) dan Java (untuk mengatasi yang kedua), tetapi plugin-plugin ini tidak lagi disukai karena ketegangan antara sumber tertutup, plugin berpemilik, dan sifat terbuka Web meningkat.

JavaScript juga berevolusi untuk mengatasi banyak masalah yang membuatnya tidak menarik untuk menjadi bahasa yang hebat seperti sekarang ini. Kekuatan pendorong lain untuk bahasa alternatif adalah bahwa banyak bagian fungsionalitas harus diimplementasikan dua kali satu kali di server (untuk memvalidasi permintaan formulir, misalnya), dan sekali lagi di klien (untuk memberikan pengalaman pengguna yang lebih baik dan respons yang lebih cepat). Beberapa kerangka kerja merespons dengan secara otomatis menghasilkan JavaScript untuk halaman berdasarkan logika sisi server, tetapi ini menimbulkan masalah. Kode yang dihasilkan secara otomatis tidak fleksibel dan sulit di-debug, dan sering kali berkinerja buruk.

Dengan munculnya Node, tujuan utama penulisan kode DRY (“jangan ulangi sendiri”) mulai terlihat, karena modul dapat dibagikan antara klien dan server. Ini memiliki tingkat keberhasilan yang bervariasi, karena konteks eksekusi di server dan eksekusi di browser sangat berbeda dan memerlukan rekayasa yang cermat agar tepat. Pendekatan alternatif adalah bahasa yang dikompilasi ke JavaScript. Layanan GMail Google memiliki front end yang ditulis dalam Java, dengan kompiler yang menghasilkan JavaScript. Bahasa lain, seperti TypeScript dan CoffeeScript, telah mengambil pendekatan yang sama, dan menempati ceruk yang signifikan, tetapi sebagian besar pengembangan web masih dilakukan dalam JavaScript itu sendiri.

Hal ini sebagian karena pendekatan alternatif ini dapat mempersulit debugging, tetapi juga karena mencampur pustaka antar bahasa yang berbeda dan pekerjaan build tambahan yang dibutuhkan sering kali dapat menambah lebih banyak kerumitan daripada upaya yang dihemat. Sebuah fitur yang dikenal sebagai asm.js diciptakan untuk memudahkan kemampuan mengkompilasi ke JavaScript, yang memungkinkan subset JavaScript sederhana yang dapat ditargetkan oleh kompiler ini yang, secara teori, menawarkan tingkat kinerja yang tinggi. Ide asli ini telah berkembang menjadi standar yang disebut WebAssembly, semacam bahasa assembly tingkat rendah untuk Web, yang dapat berkinerja tinggi tetapi tidak memiliki akses ke DOM, sehingga tidak dapat sepenuhnya menggantikan JavaScript.

Satu pengecualian penting di mana kompilasi telah dimulai adalah dalam kasus JavaScript yang dikompilasi ke dalam bentuk JavaScript lain, suatu proses yang dikenal sebagai transpilasi. Untuk waktu yang lama, JavaScript tidak mendukung modul atau paket.

Asynchronous Module Definitions (AMDs) adalah upaya pertama untuk menyelesaikan ini, dan berjalan secara native di browser hanya dengan penambahan pustaka, tetapi Node menggunakan metode alternatif untuk memuat modul yang dikenal sebagai CommonJS. CommonJS tidak kompatibel secara native dengan browser, jadi diperlukan alat untuk mengemas modul CommonJS ke dalam skrip yang kompatibel dengan browser (Browserify adalah alat paling populer yang digunakan untuk melakukan ini).

Ini berarti tidak ada lagi pemetaan langsung antara kode yang ditulis dalam IDE dan kode yang berjalan di browser. Bahkan sebelum ini, alat minifikasi digunakan untuk meningkatkan kinerja, menerapkan pengoptimalan pada kode dan menyediakan satu file yang dapat diunduh. Seperti halnya bahasa lain yang dikompilasi ke JavaScript, ini membuat penelusuran kesalahan menjadi lebih sulit. Untungnya, teknik yang dikenal sebagai peta sumber diperkenalkan yang memungkinkan debugger dalam browser untuk membantu memetakan JavaScript yang diperkecil atau diubah kembali ke aslinya. Dengan ini, langkah selanjutnya adalah transpilasi JavaScript. Meskipun banyak orang menggunakan browser terbaru dan, berkat pembaruan otomatis, tetap mengikuti perkembangan terkini, masih ada banyak orang yang menggunakan browser lama, dan tidak dapat atau tidak mau memperbarui.

Bagi sebagian orang, ini bisa jadi akibat penggunaan perangkat lama dan tidak didukung, tetapi bagi organisasi, kebijakan sering kali mendikte pendekatan konservatif untuk meluncurkan perangkat lunak baru hingga perangkat lunak tersebut diuji secara menyeluruh, yang berarti browser bisa jadi tertinggal. Seiring dengan meningkatnya laju evolusi web, pengembang ingin menggunakan fitur-fitur baru ini, tetapi dibatasi oleh browser lama. Pada awalnya, teknik yang dikenal sebagai polyfilling digunakan. Polyfill adalah skrip yang menyediakan versi JavaScript murni dari API baru yang tersedia dalam JavaScript, dan efektif dalam memungkinkan orang menggunakan API JavaScript baru dalam versi JavaScript lama. Namun, skrip ini tidak dapat menutupi semua kekurangan misalnya, menambahkan sesuatu seperti Geolocation API ke browser yang tidak mendukungnya adalah hal yang mustahil.

Keterbatasan lain dari polyfill adalah bahwa mereka tidak dapat benar-benar mengubah bahasa yang mendasarinya. Seiring dengan berkembangnya versi JavaScript yang lebih baru, sintaksisnya pun berubah, memperkenalkan cara-cara baru untuk mengekspresikan kelas dan mendefinisikan fungsi anonim. Pengenalan transpiler memungkinkan pengembang untuk menggunakan fitur-fitur baru ini dalam kode yang mereka tulis, tetapi untuk diterjemahkan ke dalam varian yang kurang mudah dibaca, tetapi kompatibel dengan versi lama untuk memungkinkan eksekusi di browser lama. Ini dapat dianggap sebagai versi yang lebih canggih dari autoprefixer yang digunakan dalam CSS. Namun, dengan transpiler ini, semakin banyak teknologi yang muncul yang lebih dekat dengan pendekatan kompiler tradisional.

React Facebook memperluas JavaScript ke "JSX," yang memungkinkan Anda untuk menentukan struktur HTML dari komponen React menggunakan sintaksis seperti HTML. Penggunaan teknologi semacam itu secara hati-hati dapat bermanfaat dalam kasus React, karena Anda hanya dapat menggunakan JSX untuk menulis komponen React, maka Anda tidak memperkenalkan ketergantungan baru selain menggunakan React. Namun, penggunaan

teknologi serupa secara luas untuk tujuan lain dapat menyebabkan situasi yang rentan, karena Anda kemudian sangat bergantung pada pendekatan nonstandar untuk sebagian besar aplikasi Anda, dan bukan hanya pada area yang cakupannya telah ditetapkan secara cermat. Sebelum diperkenalkannya polyfill dan JavaScript, pendekatan yang berbeda diambil untuk kompatibilitas mundur.

Hal ini sebagian karena kurangnya standardisasi antara dialek JavaScript, yang baru muncul beberapa lama kemudian. Beberapa perbedaan ini mendasar, seperti Mozilla dan Internet Explorer yang memiliki model yang sangat berbeda untuk menangani peristiwa JavaScript. Untuk mengatasinya, jauh lebih umum menggunakan pustaka untuk mengabstraksikan perbedaan, sehingga pustaka tersebut menjadi antarmuka Anda ke peramban. Sejauh ini, pustaka yang paling umum digunakan untuk melakukan ini adalah JQuery, yang menjadi ada di mana-mana di seluruh web, hadir di sebagian besar situs web pada puncaknya.

Seiring berkembangnya Web, praktik ini menjadi bermasalah. Di peramban modern, lapisan abstraksi yang ditawarkan oleh JQuery memperlambat banyak hal, dan banyak fitur yang ditawarkan JQuery kini ditawarkan langsung oleh JavaScript itu sendiri. Selain itu, JQuery bekerja dalam namespace global, yang dapat menyebabkan ketegangan saat Anda mencoba mengembangkan komponen yang terhubung secara longgar. JQuery masih merupakan alat yang berguna dan tidak boleh diabaikan, tetapi dalam kasus sederhana, tidak jarang melihat modul yang berinteraksi dengan DOM secara langsung, alih-alih pengembang yang mengandalkan JQuery. Gunakan JQuery saat dibutuhkan, tetapi ketahuilah bahwa tidak perlu lagi menjadikannya sebagai default.

Karena orang-orang membangun aplikasi JavaScript yang semakin kaya, keputusan desain awal untuk hanya memiliki satu thread, dan menggunakan model asinkron untuk menjaga UI tetap responsif, telah mulai menimbulkan masalah. Pengembang front-end sering menyebut masalah ini sebagai "jank." Situs menjadi janky jika responsnya tidak lancar (misalnya, animasi lambat, atau pengguliran tidak lancar, yang sering kali disebabkan oleh event handler "scroll"). Untungnya, browser baru mendukung "web workers," yang merupakan cara untuk menghadirkan struktur tipe thread ke JavaScript. Web workers dapat dianggap lebih sebagai proses latar belakang daripada thread. Mereka berjalan di lingkungan yang lebih terbatas, tanpa akses ke set lengkap API browser. Yang terpenting, web worker tidak memiliki akses ke DOM untuk mengubah halaman secara langsung. Tidak ada pula status bersama, meskipun pesan dapat diteruskan bolak-balik antara konteks JavaScript utama dan pekerja.

8.4 PENGEMBANGAN OFFLINE-FIRST

Web pada dasarnya adalah media yang terhubung. Jika Anda tidak memiliki koneksi jaringan, Anda tidak dapat memuat halaman web. Aplikasi asli selalu memiliki keunggulan ini dibandingkan aplikasi web, karena kode dan konten disimpan secara lokal di perangkat. Karena aplikasi yang semakin kompleks kini dikirimkan melalui Web, dan dengan munculnya telepon pintar yang mungkin memiliki konektivitas yang tidak merata, solusi untuk kesenjangan ini

diperlukan. HTML5 memperkenalkan konsep cache aplikasi, di mana file manifes menentukan sumber daya mana yang harus di-cache oleh browser, yang mana yang "hanya daring", dan apakah konten placeholder luring apa pun harus digunakan saat jaringan tidak tersedia. Pendekatan ini terbukti sulit untuk digunakan dan tidak lagi digunakan dalam standar. Alternatif yang jauh lebih fleksibel kini telah muncul, yang dikenal sebagai pekerja layanan.

Pekerja layanan adalah aplikasi JavaScript yang berjalan di latar belakang browser, tanpa akses ke DOM. Ia berperilaku seperti server proksi, di mana semua permintaan untuk halaman atau domain tempat ia didaftarkan melewatinya, yang berarti bahwa pekerja layanan dapat memutuskan cara menangani cache untuk permintaan. Pekerja layanan pada dasarnya tidak sinkron. Mereka menerima peristiwa dari browser, lalu memanggil respondWith() pada peristiwa tersebut dengan janji yang diselesaikan dengan respons terhadap permintaan yang ditandai oleh peristiwa tersebut, atau ditolak jika terjadi kesalahan.

Misalnya, peristiwa pemasangan terjadi saat pekerja layanan pertama kali dimuat. Dalam kasus ini, pekerja layanan biasanya mengunduh semua aset statis dan menambahkannya ke cache browser. Sering kali peristiwa terpenting, setelah service worker dipasang, adalah peristiwa fetch. Peristiwa ini dipanggil saat browser ingin membuat permintaan jaringan. Terserah Anda untuk menanganinya dengan tepat. Anda dapat memilih untuk kembali dari cache secara instan jika ada, yang menghasilkan peningkatan kinerja tetapi berpotensi menyajikan konten basi kepada pengguna, atau sebaliknya membuat permintaan jaringan untuk mendapatkan konten baru, hanya menggunakan cache jika browser sedang offline.

Caching adalah area yang rumit, dengan banyak pendekatan berbeda tergantung pada tujuan pasti Anda. Caching secara umum dibahas di tempat lain dalam buku ini, tetapi khusus untuk browser, ada Cache API. Cache API di browser tidak menangani kedaluwarsa item dalam cache, yang berarti Anda sering kali perlu menulis beberapa kode sendiri (atau menggunakan kerangka kerja) untuk menanganinya, selain menerapkan pola yang sesuai seperti "stale-while-revalidate" jika diinginkan (lihat bab Sistem untuk informasi lebih lanjut tentang teknik caching). Anda juga harus berhati-hati tentang apa yang Anda cache. Situs web berbagi foto dapat dengan mudah memenuhi disk pengguna jika semuanya di-cache tanpa batas waktu. Untuk menghindari hal ini, browser membatasi jumlah yang dapat di-cache untuk situs tertentu.

Jika Anda melakukan cache terlalu banyak, browser akan memangkas cache elemen penting dan situs Anda mungkin tidak berfungsi dengan baik saat offline. Offline-first lebih dari sekadar menggunakan pekerja layanan untuk menangani caching. Sama seperti aplikasi web mobile-first yang dirancang untuk bekerja dengan ukuran layar dan gaya interaksi yang paling terbatas terlebih dahulu, aplikasi web yang offline-first dirancang untuk bekerja tanpa koneksi jaringan sebagai mode utamanya (biasanya dengan menyinkronkan cache lokal dengan server dan kemudian hanya mengakses cache lokal tersebut), daripada menambahkan mode offline-only di kemudian hari. Hal ini dapat menambah kerumitan pada beberapa jenis aplikasi, tetapi pada akhirnya akan menghasilkan kinerja yang lebih baik dan pengalaman yang lebih tangguh bagi pengguna.

Dalam kasus situs web konten, kerumitan menjadi offline-first mungkin tidak menjadi masalah; menyinkronkan seluruh arsip situs ke perangkat akan berlebihan. Namun, selalu ada pengecualian. Panduan perjalanan yang digunakan di luar rumah melalui ponsel tanpa koneksi internet akan lebih cocok. Namun, situs web berita tidak cocok. Untuk situs web yang hanya dapat dibaca, offline-first bisa jadi cukup sederhana, yang memerlukan pengunduhan data tertentu secara berkala. Data ini kemudian dapat disimpan menggunakan IndexedDB API dari pekerja layanan dan diakses di klien. Background Sync API dirancang untuk kasus penggunaan ini, tetapi pada saat penulisan, ini adalah API yang sangat baru tanpa dukungan yang luas.

Untuk situs web tempat pengguna diharapkan memberikan data, kompleksitasnya dapat meningkat secara signifikan. Misalnya, di rumah sakit, mungkin berguna bagi dokter untuk menggunakan tablet untuk mengeluarkan resep. Namun, jangkauan wi-fi di rumah sakit mungkin tidak konsisten, jadi meskipun basis data obat dapat disinkronkan secara lokal, resep yang sebenarnya dapat disimpan hingga perangkat terhubung lagi, lalu dikeluarkan, mirip dengan cara kerja aplikasi email. Aplikasi semacam ini dapat ditangani dengan cukup mudah oleh Background Sync API dengan berperilaku berlawanan dengan sinkronisasi konten sisi server. Saat sinkronisasi terjadi, pekerja layanan akan memeriksa pesan dalam tabel pending atau outbox di IndexedDB lalu mengaktifkannya, menandainya sebagai berhasil saat selesai.

Pengalaman pengguna penting di sini, karena dokter perlu tahu apakah mereka berhasil mengeluarkan resep, atau apakah resepnya gagal. Kasus penggunaan lainnya menjadi lebih rumit. Ambil contoh, penyuntingan dokumen kolaboratif, di mana banyak orang menyunting dokumen dan menginginkan akses luring (mungkin untuk menyunting presentasi penting di kereta atau pesawat). Setiap pengembang perangkat lunak yang pernah bekerja dalam satu tim pasti pernah mengalami konflik penggabungan, yaitu ketika dua orang mengerjakan kode yang sama dan perubahan yang mereka buat saling bertentangan. Cara yang tepat untuk mengatasi hal ini bergantung pada kasus penggunaan Anda.

Terkadang, "kemenangan terkini" adalah cara sederhana untuk mengatasi hal ini, terutama jika dikombinasikan dengan pembuatan versi untuk memungkinkan kesalahan diperbaiki. Teknik yang berasal dari tahun 1980-an tetapi baru-baru ini dipopulerkan oleh Google Drive, yang dikenal sebagai transformasi operasional, adalah cara yang jauh lebih rumit tetapi efektif untuk menangani sinkronisasi ini, karena teknik ini dapat menangani operasi asinkron apa pun, bukan hanya operasi luring. Terlepas dari kerumitannya, jika Anda membangun aplikasi yang diharapkan tangguh pada perangkat seluler yang konektivitasnya mungkin tidak terjamin, offline-first akan memungkinkan Anda mengatasi kerumitan itu secara langsung, daripada menundanya ke lain waktu.

Model Objek Dokumen

Jika Anda bekerja di JavaScript sisi browser, pada suatu saat Anda harus berinteraksi dengan Model Objek Dokumen. DOM pada dasarnya adalah pohon objek JavaScript yang diekspos pada objek global yang disebut window yang mewakili struktur HTML halaman (dimulai dari window.document untuk `tag <html>`) dan API HTML lainnya. Anda memanipulasi halaman dengan mengubah pohon ini misalnya, memasukkan simpul baru sebagai anak dari simpul lain untuk membuat konten baru, atau dengan mengambil simpul di

pohon dan mengubah propertinya (misalnya, `innerText` pada `tag <p>`).

Setiap elemen HTML memiliki kelas terkait dalam JavaScript yang menawarkan API, serta beberapa API umum yang berlaku untuk semua elemen (seperti kueri untuk menemukan elemen anak, baik pada document.body untuk kueri seluruh halaman, atau di dalam elemen untuk kueri yang lebih luas). Konsep penting lain yang terkait dengan DOM adalah peristiwa. Hingga baru-baru ini, terdapat perbedaan mendasar antara cara peristiwa ditangani di browser, tetapi untungnya saat ini browser modern menerapkan standar tersebut. Peristiwa DOM terjadi sebagai respons terhadap interaksi pengguna, seperti mengklik bagian halaman atau menggulirnya. Anda dapat mengikat fungsi ke peristiwa yang terjadi pada elemen DOM tertentu dengan memanggil `addEventListener()` pada elemen DOM yang ingin Anda dengarkan. Pendekatan lama adalah menetapkan fungsi ke atribut elemen DOM yang ingin Anda tanggapi misalnya, `target.onclick = myFunction`—tetapi ini memiliki kelemahan karena hanya memiliki satu pengendali per peristiwa.

Peristiwa DOM berbeda dari jenis peristiwa lain yang memerlukan penerusan panggilan balik tertentu ke API, karena tidak ada elemen DOM untuk didengarkan. Misalnya, memanggil `setInterval()` atau `requestAnimationFrame()` keduanya menerima panggilan balik dengan hasil operasi. Perbedaan lain dalam peristiwa DOM adalah konsep penangkapan dan penggelembungan peristiwa. Ketika pengguna mengklik tombol, peristiwa tersebut pertama-tama "ditangkap" melalui semua simpul induk hingga mencapai target, lalu "menggelembung", sehingga pendengar peristiwa pada tombol, lalu pada simpul induk tombol, lalu pada induknya, dan seterusnya, dipanggil, kembali ke akar pohon DOM. Anda dapat menggunakan event.target untuk melihat elemen DOM mana yang benar-benar memicu peristiwa tersebut. Misalnya, jika pengendali klik berada pada `<p>` yang memiliki `<span>` di dalamnya, mungkin pengguna benar-benar mengklik `<span>`.

Sebagian besar pengendali peristiwa berjalan dalam fase penggelembungan, dan jarang melihat pengendali fase penangkapan dalam kode aktual. Anda dapat menggunakan addEventListener untuk menentukan apakah pengendali berjalan dalam fase penangkapan atau penggelembungan. Penangan peristiwa juga dapat mencegah penggelembungan atau penangkapan lebih lanjut dengan memanggil `stopPropagation()` pada objek peristiwa yang digunakan penangan tersebut, tetapi hal ini dapat menimbulkan konsekuensi yang tidak diharapkan. `preventDefault()` adalah alternatif yang lebih berguna yang memiliki tujuan yang sama, dengan menghentikan peramban agar tidak melakukan tindakan default-nya sambil tetap membiarkan peristiwa tersebut menyebar ke pendengar lain.

Jumlah elemen HTML, peristiwa, dan antarmukanya berubah dengan cepat, meskipun yang paling umum dari semua ini pada umumnya stabil dan membentuk inti dari standar HTML. Bahkan untuk membahasnya secara terperinci akan memerlukan buku utuh, dan saya merekomendasikan Mozilla Developer Network dan banyak sumber daya lain yang menawarkan referensi yang luas dan terkini untuk semua ini.

8.5 JAVASCRIPT SISI SERVER

Seperti JavaScript di browser, JavaScript sisi server bersifat asinkron, tetapi efek

sampingnya muncul sedikit berbeda. Jika JavaScript di browser sedang sibuk, pengguna tidak dapat berinteraksi dengan halaman, tetapi jika JavaScript di server sedang sibuk, maka server tidak akan memproses koneksi lain sama sekali pengguna harus menunggu hingga satu operasi tersebut selesai. Meskipun asinkronisitas ini pada awalnya tampak sulit dipahami (terutama bagi pengembang back-end yang terbiasa menangani thread dan teknik lain untuk memparalelkan pekerjaan), dalam praktiknya ini merupakan pendekatan yang jauh lebih sederhana dan lebih aman daripada alternatif dalam bahasa lain seperti threading, ketika pekerjaan yang perlu dilakukan oleh aplikasi terikat IO (yaitu, menunggu respons dari sistem lain seperti disk, basis data, atau server API) daripada terikat CPU. Banyak aplikasi web mengikuti pola ini.

Kelemahan utama dari sifat single-threaded JavaScript adalah bahwa pada server multi-CPU, ia terbatas untuk hanya menggunakan satu CPU, dan jika Anda ingin memanfaatkan semua CPU pada sistem multi-core, Anda perlu memulai beberapa proses aplikasi (untungnya, modul cluster Node memungkinkan Anda untuk berbagi port antara semua proses ini). Ini berarti bahwa apa pun yang disimpan dalam memori belum tentu tersedia sebagai respons terhadap permintaan mendatang dari pengguna, karena proses yang berbeda pada kotak yang sama dapat melayani permintaan itu. Untuk menyimpan informasi apa pun yang mungkin perlu bertahan melampaui beberapa permintaan, server caching eksternal seperti Memcached atau Redis digunakan.

Menggunakan pendekatan ini adalah praktik yang baik, karena jika aplikasi Anda menjadi populer, Anda mungkin ingin menskalakannya ke beberapa server di belakang penyeimbang beban untuk menangani semua permintaan. Dengan server caching eksternal, mudah untuk menskalakan ke beberapa kotak seperti halnya menskalakan ke berbagai proses. Pendekatan ini disebut penskalaan horizontal, dan dibahas lebih lanjut dalam bab Sistem. Perbedaan terbesar antara JavaScript di browser dan JavaScript di server adalah DOM. Tidak ada DOM di server (dan tidak masuk akal jika ada DOM!). Sebaliknya, Node memiliki pustaka standar yang sepenuhnya terpisah yang menyediakan fungsi yang masuk akal di server, tetapi tidak di browser, seperti membuka soket TCP mentah dan mengakses file.

Bahasa dan sintaksisnya sama saja, tetapi bagi pengembang front-end yang baru mengenal pengodean sisi server, terkadang mengejutkan untuk menyadari bahwa hal-hal seperti objek jendela global sebenarnya bukan bagian dari bahasa JavaScript, tetapi
hanya DOM. Tentu saja, Anda masih dapat merender HTML, tetapi ini dilakukan dengan menggunakan teknik templating (seperti yang dibahas dalam bab Front End) daripada dengan membangun DOM, atau dengan menggunakan pustaka DOM virtual yang berperilaku seperti DOM tetapi merender string HTML. Pustaka dan kerangka kerja yang digunakan oleh JavaScript mengalami perubahan yang sangat cepat, yang hampir pasti berarti apa pun yang Anda baca di sini akan kedaluwarsa saat buku ini dicetak.

Namun, JavaScript sisi server tampaknya telah mengembangkan budaya pustaka yang kecil dan saling terkait, berbeda dengan bahasa sisi server lain yang memiliki kerangka kerja yang jauh lebih besar dan mencakup semuanya (seperti Spring milik Java atau Symfony milik PHP). Tidak jarang memiliki banyak dependensi pada aplikasi Anda. Hal ini dapat membuat

memulai dengan cepat sedikit lebih sulit, karena Anda harus menyatukan sejumlah hal sebelum memulai, tetapi juga memberi Anda banyak fleksibilitas untuk membangun apa yang Anda butuhkan.

8.6 MODUL JAVASCRIPT

Selama ini, JavaScript tidak menggunakan konsep modul hanya file yang berbeda yang semuanya bertindak dengan cara yang sama di lingkungan yang sama. JavaScript tidak memiliki pustaka standar seperti bahasa lain semua fungsi tersedia setiap saat dan DOM adalah objek global tunggal yang disebut window yang terus ditambahkan API baru selama rilis browser. Jika Anda ingin menyertakan fungsi lain yang tidak dibangun dalam bahasa tersebut, Anda biasanya akan menambahkan tag <script> lain ke HTML Anda sebelum kode Anda dimuat, dan kemudian pustaka tersebut paling banter akan menambahkan sesuatu yang lain ke objek jendela (seperti JQuery), dan paling buruk akan membocorkan sejumlah fungsi internal ke mana-mana, dan Anda hanya bisa berharap fungsi-fungsi tersebut tidak berbenturan dengan hal lain.

Solusi pertama untuk itu adalah memanfaatkan aturan cakupan JavaScript untuk hanya mengekspos apa yang ingin Anda bocorkan. Salah satu cara untuk melakukannya adalah mekanisme yang dikenal sebagai ekspresi fungsi yang langsung dipanggil, atau IIFE. Ketika sesuatu didefinisikan di dalam suatu fungsi, itu hanya akan terlihat di dalam fungsi itu, jadi untuk menghindari meletakkan semuanya ke jendela, modul-modul dibungkus dalam suatu fungsi yang kemudian langsung dipanggil:

```
    (function() {
        function somethingPrivate() { ... }
        function externallyUsableFunction() { ... }
        window.MyLibrary = externallyUsableFunction;
})();
```

Ini mendefinisikan sebuah fungsi dan langsung memanggilnya, yang berarti bahwa ketika sebuah file yang memuatnya dijatuhkan ke halaman, ia akan langsung dijalankan dan `window.MyLibrary` tersedia, tetapi tidak ada fungsi privat yang tersedia. Namun, ada kekurangan di sini. Apa yang terjadi jika dua pustaka secara tidak sengaja memilih nama yang sama, atau Anda menginginkan dua versi berbeda dari pustaka yang sama? Anda juga harus berharap bahwa jika Anda membuat pustaka yang dapat didistribusikan yang memiliki dependensi lain, setiap halaman HTML yang menyertakan kode Anda dengan benar menambahkan tag <script> lainnya sebelum Anda dipanggil.

Ini bergantung pada pengguna yang mengikuti beberapa dokumentasi, bukan pustaka yang mendeklarasikan dependensinya sendiri dalam kode. Setelah ini, mekanisme baru yang dikenal sebagai definisi modul asinkron (AMD) digunakan. Ini adalah sintaksis yang disepakati untuk mendefinisikan modul dan kemudian mengimpornya ke modul lain.

```
  define('MyLibrary',['dependency-1','dependency-2'],function(dep1, dep2)
```

```
  {
      function MyLibrary() { ... }
      ...
      return MyLibrary;
  })
  require(['MyLibrary'], function(MyLibrary) {
      MyLibrary();
});
```

Contoh pertama di atas mendefinisikan pustaka yang disebut MyLibrary, yang mengekspor satu fungsi (disebut MyLibrary) dan memiliki akses ke dua dependensi. Ini dapat dianggap sebagai peningkatan dari IIFE; fungsi masih digunakan untuk memastikan bahwa cakupan dapat dikontrol, tetapi alih-alih melampirkan hal-hal ke objek jendela secara langsung, hal-hal tersebut dikembalikan dari fungsi dan disimpan hingga pustaka itu sendiri bergantung padanya. Definisi tersebut juga memungkinkan Anda untuk menambahkan nama-nama dependensi yang Anda perlukan, yang diberikan sebagai argumen ke fungsi yang Anda definisikan, sehingga hanya dapat diakses oleh modul yang secara eksplisit memintanya.

Apa yang sebenarnya dikembalikan oleh fungsi tersebut dapat berupa apa saja. Itu bisa berupa fungsi, atau bisa juga berupa objek dengan banyak hal di dalamnya, atau mungkin bahkan string sederhana (jika digunakan untuk mendefinisikan konsep seperti pengaturan konfigurasi atau terjemahan). Sebuah pustaka kemudian diperlukan untuk menyediakan cara mengelola dependensi ini, yang paling populer disebut RequireJS. Saat Anda menentukan dependensi, RequireJS akan memuatnya dari jaringan sesuai kebutuhan, yang dapat mengurangi waktu pra-muat, dan itulah sebabnya definisi modul ini bersifat asinkron. Pengoptimalan umum adalah menggabungkan semua dependensi bersama-sama, sehingga hanya satu file yang perlu diunduh saat runtime.

NodeJS memperkenalkan metode lain untuk memperkenalkan modul, yang dikenal sebagai modul CommonJS. Alih-alih memerlukan IIFE, NodeJS secara default menjalankan setiap file di lingkungannya sendiri untuk menghindari pencemaran namespace global. Oleh karena itu, file menjadi modul secara default, dan untuk membuat sesuatu tersedia bagi modul lain, Anda menambahkan `module.exports = MyLibrary` di bagian akhir. Modul lain kemudian penting untuk menggunakan sintaksis yang mirip dengan const `MyLibrary = require('./my-library.js')`, sebagai jalur ke file yang mendefinisikannya (atau jika jalur non-relatif, seperti jalur relatif ke folder khusus bernama node_modules tempat semua dependensi Anda berada, sering kali dikelola oleh alat manajemen dependensi).

Fungsi require () ini, tidak seperti yang ada di AMD, bersifat sinkron, oleh karena itu mengembalikan segera, alih-alih memerlukan panggilan balik. Ini baik-baik saja di server tempat semua file bersifat lokal (meskipun dapat menyebabkan pemblokiran), tetapi memberi Anda fleksibilitas yang berkurang di desktop, karena semua dependensi yang mungkin sekarang harus dikirimkan sebagai satu bundel terlebih dahulu, alih-alih memuat sebagian nanti (sekarang ada beberapa alat yang mencoba mengatasi batasan ini). Dalam upaya untuk menjembatani kesenjangan antara modul AMD dan CommonJS, pendekatan lain yang dikenal

sebagai UMD (Universal Module Definition) juga digunakan, meskipun paling sering sebagai output dari alat build untuk pustaka, yang kompatibel dengan keduanya.

Jenis modul terakhir yang akan Anda temui adalah modul ES6, yang diperkenalkan dalam spesifikasi JavaScript ES6. Modul ES6 mirip dengan modul CommonJS, kecuali sintaksis untuk mengimpor dan mengekspor modul sekarang adalah kata kunci bahasa, alih-alih variabel dan fungsi khusus.

Dalam modul ES6, Anda dapat menetapkan ekspor default seperti ini:

```
export default function() {...}
```

Dan kemudian Anda dapat mengimpor ini ke modul lain:

```
Import MyLibrary from “./my-library.js”
```

Hal ini membuat fungsi tersebut tersedia di modul kedua sebagai `MyLibrary`. Modul ES6 juga menambahkan kemampuan untuk mengekspor beberapa hal dari satu modul, bukan satu hal saja, seperti pada `CommonJS`. Dalam kasus ini, kita dapat memperluas modul pertama sebagai berikut:

```
export default function() { ... }
export function utility() { ... }
export let configurationKey = ' ... ';
```

dan sekarang di modul kedua:

```
import MyLibrary, {configurationKey} from "./my-library.js"
```

yang membuat `MyLibrary` dan string `configurationKey` tersedia, tetapi bukan fungsi utilitas. Kita juga dapat menghindari impor default sepenuhnya:

```
import { utility } from “./my-library.js”
```

Jika nama ekspor bentrok, maka Anda dapat memberinya nama berbeda saat mengimpornya:

```
import MyLibrary as AnotherLibrary from “.my-library.js”
```

Kelemahan utama modul ES6 adalah, karena merupakan fitur bahasa tersebut, nama modul harus berupa string literal. Contoh berikut tidak diperbolehkan:

```
let libraryName ="./my-library.js";
import MyLibrary as AnotherLibrary from libraryName
```

Hal ini dikarenakan semua impor dalam modul ES6 dilakukan sebelum kode dijalankan, tidak seperti pendekatan pemuatan modul lainnya, di mana pemuatan modul terjadi pada saat runtime. Hal ini dikarenakan pemuatan modul mungkin memerlukan pengunduhan kode di browser, sehingga semua modul yang dirujuk diunduh sekaligus, dan tidak harus dimuat terlebih dahulu.

Menyusun JavaScript Anda

Sebelum adopsi modul JavaScript secara umum, biasanya terdapat sejumlah kecil file yang berisi segmen fungsionalitas yang besar, atau terkadang JavaScript sebaris dalam HTML. Ini berfungsi saat JavaScript sebagian besar digunakan untuk peningkatan fungsionalitas kecil, tetapi aplikasi JavaScript modern cenderung lebih besar, dan penggunaan satu file untuk seluruh aplikasi Anda sulit dikelola. Saat ini, menggunakan modul untuk menyusun aplikasi Anda adalah hal yang normal. Mengetahui cara menyusun modul Anda bisa jadi cukup sulit. Ada dua metode utama untuk mengatasinya.

Salah satunya adalah memiliki struktur folder untuk setiap jenis komponen misalnya, mengelompokkan modul UI apa pun di satu tempat, dan modul untuk mengelola status dan logika bisnis di tempat lain, mungkin dengan pemisahan lebih lanjut berdasarkan model, tampilan, dan pengontrol untuk aplikasi MVC, atau tindakan, pereduksi, dan status untuk aplikasi Redux. Metode ini telah dikritik karena menjadi pemisahan sewenang-wenang yang mengharuskan melihat di banyak tempat berbeda saat mengubah satu fitur. Pilihan lainnya adalah memiliki direktori per fitur atau komponen fungsional dengan semua bagian berbeda dari fitur tersebut (model, tampilan, dan pengontrol, atau yang serupa) di folder tersebut. Satu masalah dengan metode ini adalah bahwa sering kali model dan konsep umum lainnya akan dibagikan di banyak fitur.

Seperti banyak hal lainnya, keseimbangan adalah kuncinya. Secara umum dianggap sebagai praktik yang baik untuk memastikan komponen yang terkait dengan tampilan Anda bebas dari logika non-tampilan apa pun. Mengeluarkan status dari komponen tampilan juga memudahkan untuk menggunakannya kembali, berpotensi di seluruh modul lain. Dalam React with Redux, ada baiknya juga untuk menjaga pemisahan antara dua jenis komponen tampilan: komponen itu sendiri, dan komponen yang terikat pada status.

Dalam kasus ini, semua tampilan Anda dapat dikelompokkan bersama, mungkin dengan subpengelompokan untuk komponen tampilan kompleks yang mungkin memiliki subkomponen, dan area logika bisnis dikelompokkan berdasarkan domain fungsional. Ini berarti tampilan mungkin mengambil dari banyak area status yang berbeda untuk merender komponen UI, tetapi untuk status, semua logika untuk memanipulasi bagian status tersebut tetap disatukan.

Jenis JavaScript

Bagi banyak pengembang yang berpengalaman dengan bahasa seperti Java atau C#, pendekatan JavaScript terhadap tipe data dapat mengejutkan. JavaScript dikenal sebagai bahasa dengan tipe lemah, yang berarti bahasa ini memiliki tipe, tetapi tipe tersebut tersirat, dan jika ada yang salah tipe, runtime JavaScript akan mencoba secara implisit mengubah

sesuatu ke tipe yang diharapkan. Perilaku ini dapat mengejutkan, dan dapat membingungkan pengembang JavaScript baru, dan menjadi bahan lelucon tentang JavaScript. Misalnya: 1 + "2" akan menghasilkan "12" karena + dibebani untuk berarti penggabungan string serta penjumlahan, dan bilangan bulat 1 akan dipaksa menjadi string untuk membuat tipenya cocok.

+[] akan berakhir menjadi 0, karena operator + mencoba membuat sesuatu menjadi positif, tetapi daftar tidak memiliki nilai yang masuk akal dan berakhir menjadi 0. Masalah terbesar yang disebabkan oleh hal ini muncul saat memeriksa kesetaraan. Saat menggunakan == (dan !=), JavaScript akan mencoba memaksa kedua sisi operator kesetaraan untuk cocok. Misalnya, "1" == 1 akan dievaluasi sebagai benar, yang biasanya tidak terduga dan tidak diinginkan. Saat menggunakan operator identitas, pemaksaan tipe ini tidak terjadi, jadi tipe dan nilai kedua sisi harus cocok. Hal ini dapat menghasilkan tampilan yang unik pada kode JavaScript, di mana operator identitas === (dan padanannya !==) sering muncul.

Untungnya, dengan penggunaan pemeriksaan kesetaraan dan identitas yang benar, Anda dapat menghindari sebagian besar jebakan pengetikan yang longgar ini. Meskipun ada kekhawatiran dari para pengembang yang terbiasa bekerja dalam bahasa dengan tipe yang kuat seperti Java, jarang sekali menemukan bug yang diakibatkan oleh pemaksaan tipe otomatis ini. Meskipun demikian, Anda juga harus menghindari penulisan kode yang menggunakan pengetikan longgar ini, karena dapat menimbulkan bug yang tidak kentara. Adalah masuk akal untuk memanggil .toString() saat menggabungkan string ketika ada keraguan atas tipe variabel, atau menjalankan input melalui parseInt untuk memastikan bahwa itu memang integer.

Pengecualian untuk ini sering kali ada pada pernyataan if. Karena string null, undefined, atau kosong menjadi false saat diubah ke Boolean (nilai-nilai ini disebut sebagai "falsey"), akan menjadi jalan pintas cepat untuk menulis pemeriksaan if seperti if (!input) { return; } di awal fungsi. Sebaliknya, sejumlah nilai dievaluasi sebagai true saat dianggap sebagai Boolean, dan ini disebut "truthy." Anda mungkin melihat sintaksis seperti !!input digunakan, yang meniadakan nilai dua kali dan, sebagai hasilnya, mengubah variabel true atau false menjadi true atau false literal. JavaScript, sejak ECMAScript 6, memiliki tujuh tipe data. Enam di antaranya dianggap primitif, dan mereka adalah Boolean, null, undefined, number, string, dan symbol.

Boolean adalah true atau false, dan mereka berperilaku seperti yang Anda harapkan. Hanya ada satu nilai yang memiliki tipe null, dan itu adalah null. Undefined serupa, hanya memiliki satu nilai: undefined. Perbedaan antara null dan undefined dapat membingungkan, tetapi perbedaan utamanya adalah bahwa undefined digunakan untuk mewakili variabel yang tidak pernah memiliki nilai yang ditetapkan padanya, sedangkan null digunakan untuk menunjukkan bahwa sesuatu telah sengaja tidak diberi nilai. Null juga berperilaku membingungkan ketika digunakan dengan operator typeof.

Meskipun null merupakan sebuah tipe, typeof null === 'object' adalah true. Ini adalah hasil dari bug dalam implementasi awal JavaScript yang dibiarkan terlalu lama, dan sekarang tidak dapat diperbaiki tanpa merusak kode lama. Selain itu, tidak seperti bahasa lain, JavaScript hanya memiliki satu tipe untuk nilai numerik: angka. Ini dapat menyederhanakan

kode, meskipun jika Anda terbiasa dengan bahasa yang memisahkan kedua tipe tersebut, Anda harus ingat bahwa cara elemen seperti pembagian integer berperilaku akan berbeda dari cara yang biasa Anda lakukan ini akan menjadi float. String juga harus menjadi konsep yang familier. Dalam JavaScript, string tidak dapat diubah, artinya operasi apa pun yang Anda lakukan pada string tidak mengubah string asli. Misalnya:

```
const url = 'https://www.example.com';
console.log(url.slice(0,8));
```

Ini akan mencetak delapan karakter pertama dari variabel url, tetapi tidak benar-benar mengubahnya. Sebaliknya, string baru dikembalikan. Dalam hal implementasi, string JavaScript terdiri dari karakter UTF-16 (dikenal sebagai titik kode), yang dapat bertindak aneh saat bekerja dengan karakter Unicode. Dalam Unicode, beberapa karakter memiliki ID yang terlalu panjang untuk dimasukkan ke dalam ruang UTF-16, jadi dua karakter UTF-16 digunakan, yang dikenal sebagai pasangan pengganti, untuk mewakili satu karakter. Ini dapat terwujud sebagai string yang tampak lebih panjang dari yang sebenarnya.

"`.length` akan mengembalikan 2, bukan 1 seperti yang diharapkan, karena emoji ada di area Unicode yang dikenal sebagai "alam astral", jadi memerlukan dua karakter UTF-16. Operasi lain juga memerlukan kehati-hatian. Misalnya, jika Anda ingin membalikkan string, pendekatan yang naif adalah mengulanginya secara terbalik, tetapi ini akan menghasilkan string yang tidak valid, karena titik kode UTF-16 individual akan dibalik, dalam hal ini merujuk pada karakter Unicode yang tidak valid. Ada pustaka yang dapat membantu Anda bekerja dengan jenis string ini, tetapi detail lebih lanjut berada di luar cakupan pengantar ini.

Primitif terakhir adalah tambahan terbaru untuk JavaScript. Simbol ditambahkan dalam ES6 sebagai cara untuk mengidentifikasi kunci pada objek secara unik. Perbedaan utama antara string dan simbol adalah bahwa dua simbol tidak identik meskipun memiliki nilai yang sama. Walaupun "example" === "example" bernilai benar, Symbol("example") !== Symbol("example") mengharuskan perbandingan tersebut tidak sama agar bernilai benar, kecuali simbol pada kedua sisi merupakan contoh objek yang sama. Objek adalah pekerja keras JavaScript, dan dapat melakukan banyak hal yang berbeda.

Objek pada dasarnya memetakan kunci ke nilai, berperilaku seperti peta atau kamus dalam bahasa lain. Fungsi juga merupakan objek, yang berarti bahwa objek dapat dieksekusi. Objek juga digunakan untuk mengimplementasikan tipe tingkat tinggi lainnya, seperti daftar tertaut (dikenal sebagai array dalam JavaScript) atau set. Array JavaScript adalah objek dengan kunci numerik dan metode pembantu lainnya. Objek yang berbeda selalu memiliki nilai yang berbeda, meskipun isinya sama.

Misalnya `{ foo: 'bar' } !== { foo: 'bar' }` akan dievaluasi menjadi benar, karena dalam kasus ini, dua objek yang dibuat adalah contoh objek yang berbeda. Karena array juga merupakan objek, ini berarti bahwa [1, 2] !== [1, 2] juga benar. JavaScript tidak menyediakan cara bawaan untuk memeriksa kesetaraan dalam kasus ini, tetapi ada pendekatan sederhana untuk mengimplementasikan jenis pemeriksaan kesetaraan ini. Untuk

array (atau objek yang urutan kuncinya dijamin), menggunakan `JSON.stringify()` untuk menerjemahkan sesuatu menjadi string JSON lalu melakukan pemeriksaan kesamaan string adalah cara cepat untuk memeriksanya. Untuk pemeriksaan yang lebih rumit, seperti objek yang sangat bertingkat, tersedia banyak pustaka utilitas.

8.7 NOTASI OBJEK JAVASCRIPT (JSON)

Saat data ditransfer antar sistem, data tersebut sering kali perlu diubah menjadi string, atau berkas biner, agar dapat ditransfer melalui jaringan. Proses ini dikenal sebagai serialisasi, dan sering kali dapat digunakan untuk membuat serial objek yang berubah-ubah. Misalnya, dalam Python, hal ini dikenal sebagai pickling. JSON adalah padanan JavaScript, meskipun hanya mendukung serialisasi tipe data fundamental, bukan kelas atau yang serupa, tidak seperti bahasa lain. JSON diusulkan pada pertengahan tahun 2000-an oleh Douglas Crockford sebagai alternatif yang lebih ringan untuk format serialisasi berbasis XML yang awalnya digunakan JavaScript. JSON adalah sekumpulan sintaks terbatas yang digunakan JavaScript untuk mendeklarasikan literal Boolean, objek, string, angka, dan array.

Misalnya, ia hanya mendukung tanda kutip ganda (") untuk string, dan tidak mendukung tanda koma di akhir pada objek atau item array. Hal ini dimaksudkan untuk membuat bahasa lebih sederhana dan mudah digunakan untuk mengurai, Anda dapat menggunakan fungsi eval JavaScript untuk menjalankannya sebagai JavaScript, dan hasilnya akan menjadi objek yang diurai. Namun, karena eval memungkinkan menjalankan JavaScript yang sembarangan, hal ini membuka celah keamanan jika ada kemungkinan JavaScript tidak terbentuk dengan benar atau tidak tepercaya, sehingga versi bahasa yang lebih baru memperkenalkan `JSON.parse()` dan `JSON.stringify()` sebagai metode untuk mengonversi objek JavaScript ke dan dari string JSON.

Sebagai hasil dari kesederhanaannya yang relatif dan dukungan untuk tipe-tipe yang juga mendasar dalam bahasa lain, JSON telah menjadi format serialisasi umum untuk bahasa dan sistem lain juga, dan ada banyak pustaka untuk bahasa lain untuk membuat serial dan mengurai JSON bahkan yang server maupun kliennya bukan JavaScript. Sebagai hasil dari kurangnya dukungan JSON untuk tipe yang lebih kompleks, baik server maupun klien harus sepakat tentang cara mengonversi sesuatu yang lebih kompleks menjadi JSON dan kemudian kembali ke tipe yang lebih kompleks, sehingga JSON sendiri tidak dapat sepenuhnya mendeskripsikan data sendiri. Ia harus dipasangkan dengan beberapa informasi tipe tambahan. Ini adalah sumber kritik untuk JSON, terutama dari komunitas XML, di mana skema XML dapat digunakan untuk lebih ketat mendefinisikan bentuk dan makna data yang dikodekan.

Ada alat yang tersedia untuk JSON yang membantu mendefinisikan makna dan memeriksa validitas JSON yang diserialkan Skema JSON adalah salah satu opsi yang lebih umum digunakan. Keuntungan terbesar dari pengetikan lemah JavaScript adalah pengetikan bebek dapat digunakan saat meneruskan objek (misalnya, jika suatu objek berperilaku dengan cara yang diharapkan, meskipun tidak didefinisikan dalam hal tipe yang diharapkan, seperti jika objek tersebut memiliki kunci yang sama atau serupa). Namun, untuk sebagian besar,

pengetikan lemah sering kali dianggap sebagai hal negatif yang harus dihindari. Untuk waktu yang lama, ada bahasa lain yang mengkompilasi ke JavaScript, dan seperti yang disebutkan sebelumnya dalam bab ini, ada transpiler JavaScript yang mengambil JavaScript baru dan membuatnya kompatibel dengan browser lama.

Untuk tujuan ini, Microsoft telah mengembangkan bahasa yang disebut TypeScript yang terlihat dan terasa seperti JavaScript, tetapi menyertakan anotasi tipe yang kemudian diperiksa dalam proses transpilasi. Pendekatan alternatif telah diambil oleh Facebook dengan pemeriksa tipe Flow mereka. Alih-alih menggunakan bahasa baru yang berasal dari JavaScript, Flow memungkinkan Anda untuk menambahkan anotasi ke kode JavaScript yang ada, yang kemudian dihapus selama proses transpilasi. Pemeriksa tipe terpisah berjalan di atas kode untuk memeriksa apakah semua tipe seperti yang diharapkan.

```
function fetchTransactionValue(id: string): Promise<number> {
  ...
}
```

Kode sebelumnya adalah contoh anotasi tipe dalam Flow, yang menunjukkan bahwa fungsi tersebut mengambil satu parameter bertipe string, lalu mengembalikan promise, yang diselesaikan menjadi angka (sering kali, tipe yang dikembalikan disimpulkan, jadi tidak perlu ditentukan secara eksplisit). TypeScript dan Flow adalah metode populer untuk membantu tim pengembangan menulis JavaScript dan menghindari bug, dan pada saat penulisan ini, tampaknya adopsi alat-alat ini hanya akan meningkat.

Pemrograman Berorientasi Objek

Pemrograman berorientasi objek telah menjadi metodologi utama dalam rekayasa perangkat lunak selama beberapa dekade terakhir, menggantikan pemrograman struktural. Meskipun JavaScript mendukung orientasi objek, ia tidak melakukannya dengan cara yang biasa dilakukan kebanyakan orang. Sebagian besar programmer dilatih pada orientasi objek dalam pengertian klasik (di sini, klasik berarti "berkaitan dengan kelas," bukan "tradisional"), di mana objek adalah perwujudan kelas, dan kelas dapat memiliki hierarki. Hingga ES6, JavaScript berbeda karena kelas JavaScript bersifat prototipe. Alih-alih objek menjadi contoh kelas, objek merupakan contoh prototipe.

Dengan demikian, objek tidak memiliki banyak fitur yang mungkin dianggap biasa oleh pengembang Java atau PHP, seperti antarmuka, pewarisan, atau bahkan visibilitas metode dan bidang yang ditentukan. ES6 telah mengubah semua itu dengan memperkenalkan kelas ke dalam JavaScript, yang jauh lebih familiar bagi mereka yang berasal dari bahasa lain. Di balik layar, objek masih berbasis prototipe, tetapi kini Anda dapat memperlakukannya sebagaimana Anda memperlakukan kelas. Pustaka juga tersedia untuk varian JavaScript yang lebih lama agar prototipe terasa lebih seperti kelas, tetapi pustaka tersebut tidak pernah menjadi bagian inti bahasa tersebut. Teknik lain memberikan dukungan untuk bidang privat, seperti menjaga bidang privat dalam cakupan yang berbeda dan memaksakan akses melalui pengambil/penyetel, tetapi hal ini sering kali memiliki beban kinerja atau memori.

JavaScript bukan satu-satunya bahasa yang tidak memiliki metode/bidang privat. Python juga tidak, dengan mengambil pendekatan "kesepakatan sopan". Metode dan bidang "privat" diawali dengan garis bawah dan tidak didokumentasikan, dan meskipun hal ini tidak mencegahnya diakses secara langsung, metode dan bidang tersebut menunjukkan kepada pengembang bahwa mereka melakukan sesuatu yang salah dan tidak boleh diandalkan. Untuk banyak aplikasi JavaScript, pendekatan ini juga cukup, bahkan jika Anda mendistribusikan pustaka untuk konsumsi yang lebih luas. Objek prototipe dalam JavaScript dimulai dengan mendefinisikan konstruktor sebagai fungsi normal, lalu menambahkan fungsi ke objek fungsi tersebut yang disebut prototipe. Bentuknya seperti ini:

```
}
function ShoppingBasket() {
    this._items = [];
}

ShoppingBasket.prototype.addItem = function(item) {
    this._items.push(item);
}
```

Menjalankan new `ShoppingBasket()` akan membuat objek baru dari prototipe, menetapkan this sebagai objek tersebut, lalu menjalankan konstruktor. Kelas ES6 menyederhanakan sintaksis ini agar terlihat lebih familiar bagi pengembang yang menggunakan bahasa lain, artinya Anda dapat menentukan hal di atas dalam pengertian yang lebih familiar:

```
class ShoppingBasket {
    constructor() {
        this. items = [];
    }

    addItem(item) {
        this.items.push(item);
    }
}
```

Sekarang, ingat bahwa dalam contoh pertama kita, ShoppingBasket hanyalah sebuah fungsi. Itu berarti bahwa jika kita memanggil `ShoppingBasket()`, lupa menambahkan new di awal, maka kode akan dijalankan, tetapi akan melakukan hal yang salah. Objek baru tidak akan dibuat, jadi ini akan menjadi objek jendela, dan tidak ada yang akan dikembalikan darinya. JavaScript memiliki mode yang disebut mode ketat untuk membantu mencegah hal ini terjadi.

Mode ketat dapat diaktifkan dengan menambahkan "use strict"; di bagian atas file atau fungsi, dan dirancang untuk membuat kesalahan umum benar-benar memunculkan kesalahan,

daripada diabaikan begitu saja. Dalam kasus ini, mode ini menghentikan variabel yang tidak dikenal agar tidak ditetapkan pada this kecuali jika itu adalah objek yang sebenarnya. Untuk menghindari hal ini, ada pola yang lebih memilih untuk tidak menggunakan new sama sekali, tetapi menggunakan fungsi untuk semuanya. Misalnya, kode sebelumnya dapat dinyatakan sebagai:

```
function ShoppingBasket() {
    const items = [];
    return {
        addItems(item) {
            items.push(item);
        },
    };
}
```

Kode ini adalah fungsi yang mengembalikan objek dengan fungsi. Kode ini juga menjadikan item bersifat privat dengan tidak benar-benar meletakkannya pada objek yang dikembalikan. Hal ini memiliki sisi negatif yaitu meningkatkan memori, karena setiap metode pada objek merupakan contoh baru, bukan prototipe bersama. Namun, untuk JavaScript modern, kelas ES6 adalah cara yang paling disukai untuk mengekspresikan kelas, sedangkan pendekatan lainnya hanya umum dalam kode lama. Ada juga berbagai pustaka yang dapat membantu membuat berbagai jenis kelas JS, tetapi sebagian besar sudah ada sebelum kelas ES6. Salah satu manfaat terbesar kelas ES6 adalah kelas tersebut membuat pewarisan kelas berperilaku seperti yang diharapkan banyak orang. Ambil contoh:

```
class Animal {
    ...
}
class Dog extends Animal {
    constructor() {
        super();
        this.kennel = null;
    }
}
```

Dalam hal ini, kelas Dog memperluas Animal, jadi ketika new `Dog()` dipanggil, ia memiliki metode Dog dan metode Animal (dengan asumsi bahwa Dog belum menimpanya). Kata kunci super juga tersedia, dan dapat memanggil metode pada induk (atau lebih jauh, jika rantai pewarisan memiliki beberapa lapisan dalam). Ketika menggunakan kelas ES6, ini menyembunyikan banyak kerumitan yang diperlukan untuk menentukan prototipe secara langsung (inilah sebabnya ada beberapa helper dalam JavaScript untuk melakukan pewarisan sebelum ES6). Cara terbaik untuk memahami cara kerja pewarisan dalam JavaScript adalah dengan mengetahui tentang rantai prototipe.

Sebuah instance dari kelas JavaScript adalah objek yang memiliki prototipe. Namun, prototipe ini sendiri hanyalah sebuah objek, jadi ia sendiri dapat memiliki prototipe. Ini disebut rantai prototipe, dan ketika Anda mengakses objek JavaScript, rantai prototipe diikuti hingga hal pertama dengan nama yang Anda minta ditemukan. Pada akhirnya, objek terakhir yang Anda definisikan dalam rantai akan memiliki prototipe dari Object bawaan JavaScript, yang sendiri memiliki prototipe null, yang menunjukkan akhir dari rantai. Ini menunjukkan bahwa tidak pernah ada yang disebut kelas dalam JavaScript hanya objek dan prototipe tetapi detail implementasi ini sebagian besar tidak relevan untuk sebagian besar penggunaan JavaScript sehari-hari.

Peringatan terakhir mengenai objek dalam JavaScript, yang melibatkan kata kunci ini. Bagi pengembang yang berasal dari hampir semua bahasa berorientasi objek lainnya, this selalu merujuk pada contoh objek tempat metode tersebut berada. Dalam JavaScript, this merujuk pada konteks tempat fungsi tersebut beroperasi. Misalnya, saat fungsi ditambahkan sebagai pengendali peristiwa klik, saat dijalankan, this akan menjadi elemen yang memicu peristiwa tersebut, karena itulah konteks tempat panggilan balik dijalankan. JavaScript tidak membedakan antara metode dan fungsi, jadi meneruskan metode ke panggilan balik klik akan memisahkannya dari objek tempat metode tersebut berada.

Cara umum untuk mengatasi hal ini adalah dengan membuat variabel bernama that atau _this atau yang serupa, dan membuat fungsi yang memiliki cakupan that (ini disebut penutupan lebih lanjut tentang itu di bagian Pemrograman Fungsional), yang memberinya akses ke objek tersebut. Dalam ES6, ada dua mekanisme alternatif yang digunakan: menggunakan `.bind(this)`, yang membuat fungsi baru dengan variabel this yang merujuk this dengan benar (yang sama dengan “this” yang dimasukkan sebagai argumen ke bind), atau menggunakan singkatan panah untuk mendefinisikan fungsi (() => { ... }), yang melakukannya secara otomatis.

Pemrograman Fungsional

Pemrograman fungsional bukanlah hal baru ini mendahului munculnya pemrograman berorientasi objek tetapi telah mengalami kebangkitan, dan beberapa keputusan desain awal JavaScript telah membuatnya cocok untuk menerapkan teknik pemrograman fungsional ini, meskipun tidak memaksakan pemrograman fungsional murni. Salah satu keuntungan JavaScript dalam konteks ini adalah memungkinkan Anda untuk beralih di antara gaya berbasis kelas, prosedural, dan fungsional dengan mudah, dan bahkan menggunakannya dalam kombinasi satu sama lain. Beberapa orang menganggap pemrograman fungsional murni tidak cocok untuk jenis tugas tertentu, dan tentu saja bagi banyak pengembang hal ini memerlukan perubahan dalam pemikiran tentang cara menyusun program.

Hal ini telah menahan munculnya pemrograman fungsional untuk waktu yang lama. Intinya, pemrograman fungsional telah bangkit dari dunia matematika. Terkadang hal ini bisa jadi menakutkan bagi pendatang baru, karena banyak istilah (functor, monad, dsb.) yang sangat asing, dan konsepnya sering kali tidak mudah dijelaskan tanpa mengandalkan pengetahuan dasar tentang teori kategori. Namun, Anda dapat menggunakan teknik pemrograman fungsional tanpa mengembangkan pengetahuan yang mendalam. Inti dari

pemrograman fungsional adalah gagasan tentang fungsi. Fungsi adalah sesuatu yang, untuk input tertentu, selalu menghasilkan output yang sama. Bandingkan ini dengan pendekatan berorientasi objek, seperti objek tanggal ini:

```
class Date {
    ...
    addDays (daysToAdd) {
        this.timestamp += daysToAdd * 24 * 60 * 60;
    }
}
```

jika kita menjalankan addDays beberapa kali, hasil yang kita dapatkan berbeda setiap kali. Kita dapat menulis fungsi serupa yang melakukan hal yang sama setiap kali, tetapi fungsi tersebut tidak menyimpan status apa pun. Fungsi tersebut mengembalikan hasil baru, dan tidak mengubah hasil aslinya.

```
function addDays(date, daysToAdd) {
    return {
        timestamp: date.timestamp + (daysToAdd * 24* 60 * 60),
        timezone: date.timezone,
    };
}
```

Ini sangat ampuh, karena menghilangkan serangkaian bug, dan juga memungkinkan kita untuk merangkai metode dengan cara yang baru dan menarik. Tentu saja, metode juga dapat dikodekan dengan cara ini, sehingga mengembalikan contoh baru alih-alih memodifikasi contoh yang ini. Jika Anda telah bekerja secara ekstensif dengan JavaScript, ada kemungkinan besar Anda telah menggunakan teknik fungsional, tanpa menyadarinya. Sebelum .bind menjadi fitur bahasa tersebut, Anda mungkin telah melihat teknik umum ini:

```
Widget.prototype.setupEventListeners = function() {
    var that = this;
    foo.addEventListener('click', function(ev) {
        that.handleClick(ev);
    });
}
```

Ini menggunakan dua teknik yang merupakan inti dari pemrograman fungsional: penutupan dan fungsi sebagai objek kelas satu. Penutupan hanyalah sebuah fungsi yang memiliki akses ke cakupan tempat ia didefinisikan. Dalam fungsi `setupEventListeners`, kami mendeklarasikan sebuah variabel yang, dan penangan panggilan balik klik. Meskipun penangan panggilan balik adalah fungsi yang berbeda, dan berpotensi berjalan lama setelah `setupEventListeners()` selesai dijalankan, penangan tersebut masih dapat

menggunakannya, karena penangan tersebut memiliki visibilitas ke cakupan (dan juga cakupan bersarang tempat penangan tersebut dideklarasikan).

Apa Itu Ruang Lingkup?

Ruang lingkup menentukan fungsi mana yang dapat "melihat" variabel atau fungsi tertentu. Ambil contoh:

```
function getCircleArea(radius) {
    const PI = 3.1415926535;
    return PI * radius * radius;
}
```

(pada kenyataannya Anda akan menggunakan Math.PI)

Variabel PI berada dalam cakupan fungsi ini, tetapi tidak ada fungsi atau area kode lain yang dapat mengaksesnya. Karena JavaScript memungkinkan kita untuk menumpuk fungsi dan blok, cakupan juga dapat ditumpuk di dalam satu sama lain untuk menciptakan tingkat cakupan yang berbeda. Banyak kerangka kerja pemuatan modul juga akan menciptakan cakupan pada tingkat file, jadi Anda tidak dapat menggunakan variabel secara sembarangan di file lain kecuali jika dideklarasikan sebagai ekspor, dan Anda mengimpornya.

Bahasa lain juga menggunakan gagasan cakupan "global", di mana sesuatu tersedia secara otomatis tanpa harus diimpor. JavaScript tidak menggunakan konsep yang sama, tetapi memiliki objek global (jendela di lingkungan berbasis browser) yang dapat Anda tambahkan. Sebelum ES6, satu-satunya cara untuk mendeklarasikan variabel adalah dengan kata kunci var atau function (untuk fungsi), tetapi ini tunduk pada sesuatu yang dikenal sebagai variable hoisting. Dalam variable hoisting, semua variabel didefinisikan sebelum kode apa pun dijalankan. Misalnya, dalam mode ketat, Anda akan mengharapkan ini gagal, karena a dideklarasikan saat disetel:

```
a = 6;
var a = 5;
console.log(a);
```

Karena pengangkatan variabel `var a`, saat `a = 6` dijalankan, `var` telah dideklarasikan. `a = 5` mengganti nilai, karena nilai awal tidak diangkat, hanya definisinya. Ini dapat berguna untuk fungsi rekursif; misalnya:

```
function recurseDeep(items) {
  return [].concat(recurseWide(items[o]), recurseWide(items.splice(1));
}
function recurseWide(items) {
  if (items[o].length > 1) {
    return recurseDeep(items);
  } else {
    return [].concat([items[o][o]], recurseWide(items.splice(1));
```

```
    }
  }
```

Anda ingin `recurseWide` berada dalam cakupan untuk recurseDeep, meskipun telah dideklarasikan sebelumnya. ES6 telah memperkenalkan dua kata kunci baru yang dapat membantu melindungi Anda dari kesalahan yang dapat secara tidak sengaja diperkenalkan oleh var. Kata kunci tersebut adalah let dan const, dan keduanya dikenal sebagai block-scoped. Alih-alih dideklarasikan secara otomatis di bagian atas fungsi yang berisi definisi, keduanya dideklarasikan pada titik saat benar-benar digunakan, dan hanya berada dalam cakupan di dalam blok (rangkaian kurung kurawal) tempat definisi tersebut muncul.

let digunakan untuk variabel yang dapat ditetapkan ulang, sedangkan const untuk variabel yang tidak ditetapkan ulang. const tidak selalu konstan; misalnya, daftar atau objek yang didefinisikan sebagai const dapat memiliki hal-hal baru yang ditambahkan atau dihapus darinya, karena selalu menunjuk ke daftar yang sama. Praktik terbaik adalah selalu menggunakan const, kecuali jika Anda perlu menetapkan ulang, dalam hal ini Anda harus menggunakan let. Contoh penggunaan let dan const yang agak dibuat-buat adalah:

```
function calculate(a, b, callback) {
  let wrappedCallback;
  if (callback === null) {
    wrappedCallback = () => {
      console.log('done');
    };

  } else {
    wrappedCallback = (result) => {
      callback(result);
      console.log('done');
    };

  }

  if (a > b) {
    const result = a + b;
    wrappedCallback(result);
  }

  wrappedCallback(null);
}
```

Dalam contoh ini (yang memanggil kembali hasil penambahan a dan b jika a lebih besar dari b, lalu null untuk menunjukkan bahwa perhitungan telah selesai), `wrappedCallback` harus dibiarkan karena nilainya ditetapkan setelah kejadian, sedangkan result adalah const karena tidak pernah ditetapkan ulang. Di baris terakhir fungsi, kita tidak akan dapat mengakses result (akan memberikan kesalahan sebagai variabel yang tidak dideklarasikan) karena cakupan

result terbatas pada badan fungsi if tempat ia dideklarasikan.

Fungsi sebagai objek kelas satu berarti bahwa fungsi adalah objek yang dapat diteruskan seperti objek lainnya, seperti string atau angka. Salah satu konsep penting yang dimungkinkan oleh hal ini adalah bahwa fungsi dapat diteruskan ke fungsi lain sebagai panggilan balik. Bandingkan hal ini dengan bahasa seperti Java dan PHP, di mana teknik yang dikenal sebagai refleksi diperlukan untuk mengakses fungsi-fungsi ini, yang memberikan sejumlah besar overhead untuk meneruskan hal-hal seperti panggilan balik.

Hal ini memungkinkan JavaScript memiliki banyak fitur asinkron, dibandingkan dengan bahasa seperti PHP, yang harus menunggu fungsi untuk dieksekusi (dikenal sebagai pemblokiran) sebelum melanjutkan ke baris kode berikutnya, atau Java, yang menggunakan utas untuk menghindari pemblokiran, meskipun utas individual masih dapat memblokir.

Ada beberapa teknik yang digunakan dalam pemrograman fungsional yang tidak umum dalam pendekatan yang berorientasi objek. Salah satu teknik tersebut adalah fungsi parsial. Fungsi parsial adalah versi fungsi lain yang beberapa argumennya sudah "diisi sebelumnya". Ambil contoh, cuplikan berikut:

```
function foo(a, b, c, d) { ... }

function makePartialFoo(a, b) {
  return function(c, d) { foo(a, b, c, d); };
}
```

`makePartialFoo` ini diberi argumen a dan b dan mengembalikan fungsi yang hanya memerlukan c dan d yang diteruskan untuk dijalankan. Setiap kali fungsi baru itu dipanggil, a dan b sudah disediakan. Ini dapat berguna jika suatu fungsi mengambil beberapa dependensi (melalui injeksi dependensi) atau status lain ke dalam beberapa argumen pertamanya dan kemudian menggunakannya sebagai argumen umum untuk sejumlah pemanggilan. `bind()` memungkinkan kita untuk membuat fungsi parsial jauh lebih mudah. Kita dapat membuat versi parsial dari fungsi foo di atas dengan memanggil: `foo.bind(this, a, b)` alih-alih memerlukan fungsi untuk membuat parsial foo (yang dapat menjadi sulit jika Anda menginginkan beberapa versi foo).

Dalam bahasa fungsional murni, semua fungsi dapat dibuat parsial hanya dengan memanggilnya dengan sebagian argumennya, suatu proses yang dikenal sebagai currying. Dalam pemrograman fungsional, fungsi tidak boleh memiliki beberapa status. Sebaliknya, status direpresentasikan dalam struktur data yang diteruskan ke atau dikembalikan dari fungsi. Demikian pula, fungsi murni tidak boleh memanipulasi apa yang telah diberikan. Misalnya, jika Anda memiliki fungsi yang menambahkan uang ke akun, dan fungsi tersebut disebut sebagai `addMoneyToAccount(akun, uang)`, maka fungsi tersebut harus mengembalikan objek akun baru dengan uang yang ditambahkan ke dalamnya, dan contoh objek akun sebelumnya tetap berada dalam keadaan sebelumnya.

Ini disebut kekekalan, karena setelah objek atau benda dibuat, objek atau benda tersebut tidak akan pernah berubah sebagai efek samping atau hal lainnya. Sebaliknya, objek

baru dibuat sebagai hasil operasi. Ini berguna karena memungkinkan Anda untuk merangkai operasi dengan memanggil fungsi lain pada hasilnya. Dalam JavaScript, ini bisa jadi cukup sulit—bayangkan `baz(bar(foo(thing)))`, tetapi berpotensi lebih dalam lagi. Kode tersebut bisa jadi sulit diikuti, karena urutan operasinya terbalik (misalnya, foo dieksekusi terlebih dahulu, lalu bar, dan terakhir baz). Akibatnya, bahasa pemrograman fungsional menawarkan berbagai cara untuk merangkai fungsi seperti ini (dalam Haskell, sintaksnya adalah foo . bar . baz thing), dan banyak kerangka kerja pemrograman fungsional dalam JavaScript menawarkan cara untuk meniru penggabungan ini.

Mungkin terlihat asing, tetapi ingatlah bahwa itu tetap panggilan fungsi. Sebagai bahasa yang tidak sepenuhnya berorientasi objek atau fungsional, tetapi memungkinkan Anda untuk mencampur kedua gaya, sulit untuk menentukan apa itu JavaScript idiomatis. Kapan kelas harus digunakan, dan kapan fungsi harus digunakan? Sering kali, semuanya bergantung pada apa yang Anda sukai, apa yang sudah ada dalam basis kode, atau gaya yang digunakan oleh pustaka atau kerangka kerja eksternal yang Anda buat di atasnya. Merupakan hal yang umum untuk mencampur dan mencocokkan gaya dalam basis kode, tergantung pada tugas yang sedang dikerjakan.

Aturan praktis yang masuk akal adalah menggunakan gaya fungsional ketika Anda ingin membuat fungsi yang lebih generik, yang bekerja pada struktur data yang lebih generik seperti daftar atau kamus, dan ada banyak cara yang mungkin Anda inginkan untuk memanipulasi data tersebut. Kelas berguna jika Anda bekerja dengan data dan ingin metode pengoperasiannya terikat erat. Tanggal dan waktu adalah contoh bagus dari hal-hal yang dapat menjadi kelas, karena logika untuk memanipulasinya terkait erat dengan sifat data. Dalam kasus ini, Anda mungkin ingin mempertimbangkan untuk membuat kelas-kelas semacam ini tidak dapat diubah, sehingga metode-metode tersebut benar-benar mengembalikan contoh-contoh kelas yang baru daripada mengubah yang sudah ada, untuk menghindari beberapa masalah yang muncul dengan kelas-kelas dan status bersama.

Di waktu lain, kelas-kelas mungkin berguna ketika Anda memerlukan pembungkus di sekitar sumber daya tertentu, dan beberapa operasi dapat beroperasi pada sumber daya yang sama. Contohnya mungkin klien HTTP yang mengambil beberapa opsi konfigurasi, atau koneksi basis data. Dimungkinkan untuk menggunakan ini secara fungsional, tetapi ini sering kali menyulitkan, karena memerlukan penerusan konfigurasi atau penanganan di banyak tempat (fungsi parsial dapat membantu, tetapi juga dapat memerlukan sejumlah besar fungsi untuk diteruskan, sedangkan kelas dapat menyederhanakan prosesnya). Dalam kasus ini, status sering kali tidak diubah setelah kelas dibangun, dan kelas tersebut bertindak hanya sebagai kumpulan fungsi melalui beberapa konfigurasi atau koneksi bersama. Kelas-kelas dalam kasus ini juga dapat menangani logika dengan benar, seperti penggabungan koneksi, penyambungan ulang, atau pemutus sirkuit, sambil menyajikan antarmuka yang sederhana kepada pengguna.

Berkomunikasi Antar Komponen

JavaScript awal sering kali memiliki persyaratan yang sangat sederhana, dan tidak memerlukan banyak kode untuk ditulis. Satu file JavaScript dengan sedikit penumpukan atau

modularisasi, dan variabel global, mungkin sudah cukup untuk memenuhi kebutuhan tersebut, tetapi seiring dengan semakin besarnya dan kompleksnya aplikasi JavaScript, hal itu tidak lagi menjadi masalah. Sebaliknya, kita menyusun JavaScript sebagai modul, tetapi modul-modul tersebut perlu berinteraksi satu sama lain. Kopling adalah istilah yang digunakan untuk menggambarkan interaksi antara dua modul, dan seberapa erat modul-modul tersebut terikat satu sama lain.

Biasanya diinginkan agar setiap modul terhubung secara longgar yaitu, untuk meminimalkan jumlah interaksi antara keduanya melalui antarmuka yang terdefinisi dengan baik. Sebaliknya, koneksi yang kuat berarti bahwa dua modul saling terhubung erat, berinteraksi satu sama lain sangat sering dalam berbagai cara. Koneksi yang kuat dapat berarti bahwa kedua modul menjadi sulit dipisahkan dan digunakan secara independen. Konsep koneksi yang longgar lazim dalam desain berorientasi objek, tetapi berlaku juga untuk pemrograman fungsional.

Koneksi yang paling longgar adalah koneksi yang tidak ada interaksi sama sekali, tetapi komponen sering kali harus setidaknya mengomunikasikan status atau perubahan satu sama lain. Salah satu cara untuk melakukannya adalah dengan memanggil metode secara langsung pada komponen mana pun yang perlu mengetahui kapan perubahan peristiwa terjadi. Namun, hal ini menimbulkan kelemahan dalam desain komponen Anda suatu komponen harus mengetahui setiap komponen lain yang mungkin dapat berubah berdasarkan status tersebut, yang dapat menimbulkan sejumlah besar koneksi. Cara alternatif untuk memecahkan masalah mengomunikasikan perubahan status adalah dengan memecahkan masalah tersebut dengan cara sebaliknya.

Daripada sesuatu yang dapat menghasilkan perubahan status dengan mengirimkan pesan kepada mereka yang perlu menanggapi perubahan tersebut, Anda dapat menyiapkan komponen yang bergantung padanya untuk mendaftarkan panggilan balik saat suatu peristiwa terjadi. Namun, ini memperkenalkan bentuk penggabungan yang berbeda, karena sekarang komponen-komponen ini harus tahu persis komponen lain mana yang dapat menghasilkan perubahan status tersebut. Cara umum untuk mengatasinya adalah dengan menggunakan bus peristiwa. Dalam skenario ini, hal-hal yang menyebabkan perubahan status mengirimkan pesan ke bus peristiwa saat suatu peristiwa terjadi, lalu yang lain berlangganan dengan panggilan balik untuk mengetahui kapan itu terjadi.

Dalam JavaScript, peristiwa adalah cara utama DOM menunjukkan perubahan pada kode yang diminati, tetapi peristiwa khusus juga dapat diaktifkan pada objek jendela global, yang memungkinkan bus peristiwa langsung dibuat, tempat peristiwa diaktifkan dengan cara yang sama seperti DOM dan pelanggan dapat berlangganan sesuai kebutuhan. Ada banyak pola arsitektur berbeda yang dapat digunakan dengan bus peristiwa. Satu pola dikenal sebagai arsitektur heksagonal. Dalam arsitektur heksagonal, aplikasi Anda dibagi menjadi beberapa lapisan (mirip dengan model-view-controller). Misalnya, pada intinya biasanya terdapat model domain, kemudian dikelilingi oleh lapisan yang mengimplementasikan logika bisnis, dan akhirnya lapisan yang mengimplementasikan interaksi dengan dunia luar bukan hanya UI, tetapi juga database dan penyimpanan persistensi lainnya.

Setiap lapisan berkomunikasi dengan lapisan di atasnya melalui port dan adapter. Port pada dasarnya adalah antarmuka ke dalam lapisan, dan adapter adalah apa yang diimplementasikan lapisan untuk berkomunikasi dengan port tertentu untuk beralih dari satu lapisan ke lapisan lainnya. Arsitektur heksagonal ini sering kali berfungsi dengan baik dengan pola desain lain yang dikenal sebagai CQRS: pemisahan tanggung jawab kueri perintah. Dalam CQRS, metode dan struktur data yang Anda gunakan untuk berinteraksi dengan data dan yang Anda gunakan untuk membaca data berbeda, dengan yang satu sesuai dengan adapter dan yang lainnya dengan port.

Arsitektur khusus ini bisa menjadi sangat rumit saat aplikasi Anda lebih dekat dengan kasus penggunaan CRUD (create-read-update-delete). Dalam CRUD, Anda biasanya hanya mengedit struktur data yang mendasarinya, tetapi CQRS lebih baik untuk memicu peristiwa tertentu di mana tindakan UI tidak secara langsung ditautkan ke perubahan struktur data yang mendasarinya. Untuk aplikasi CRUD, komponen UI tertentu mungkin hanya tertarik secara langsung pada satu bagian status (atau model data), sehingga dapat menyiapkan pengendali peristiwa pada model data tersebut secara langsung, tanpa harus menggunakan bus peristiwa. Komponen yang dimaksud dapat memperbarui model data, lalu menyiapkan peristiwa sesuai kebutuhan.

Ini dikenal sebagai pengikatan dua arah, karena komunikasi terjadi dalam dua cara. Untuk aplikasi yang lebih rumit dan berbasis tugas, memisahkan masalah melalui bus peristiwa sering kali lebih mudah dikelola, dan sebagai gantinya pengikatan satu arah ke model data kemudian digunakan. Dalam pengikatan satu arah, komponen UI memicu peristiwa ke bus peristiwa, lalu menerima model data baru/yang diperbarui berdasarkan perubahan tersebut. Pemisahan ini berarti Anda tidak perlu mengetahui detail seluruh struktur model data yang mendasarinya atau bagaimana model tersebut perlu dimanipulasi hal itu dapat diserahkan ke lapisan logika bisnis. Anda hanya perlu menghubungkannya ke bit yang memberi Anda informasi yang perlu Anda baca, bukan apa yang perlu Anda ubah.

Dengan pengikatan satu arah, aplikasi Anda lebih dekat dengan pola aplikasi CQRS, meskipun masih menjalankan aplikasi seperti CRUD. Sebagian besar sistem bus peristiwa dalam JavaScript tidak secara langsung memicu peristiwa melalui DOM, tetapi memiliki sistem peristiwa mereka sendiri di luar DOM, yang memungkinkan lapisan perantara seperti logika bisnis untuk disisipkan dengan lebih baik. Satu pustaka JavaScript umum untuk mengelola peristiwa dengan aliran satu arah adalah Redux. Dalam Redux, peristiwa dipicu dengan mengirimkan tindakan. Redux menyediakan fungsi yang disebut pengiriman, yang terikat pada contoh tertentu dari bus peristiwa, dan tindakan didefinisikan oleh pengembang sebagai hal tertentu yang dapat dipicu.

Tindakan ini sering kali ditautkan ke tindakan UI, tetapi dapat ditautkan ke peristiwa asinkron lainnya. Tindakan memiliki tipe, tetapi juga dapat membawa data lain. Misalnya, Anda mungkin memiliki tindakan yang disebut SEND_MESSAGE yang memiliki isi teks sebagai argumen. Komponen UI mengirimkan tindakan ini saat pengguna mengklik "enter" di jendela obrolan. Untuk tindakan asinkron, Anda mungkin memiliki pengatur waktu yang memicu penyegaran latar belakang status pengguna, yang kemudian dapat memicu peristiwa yang

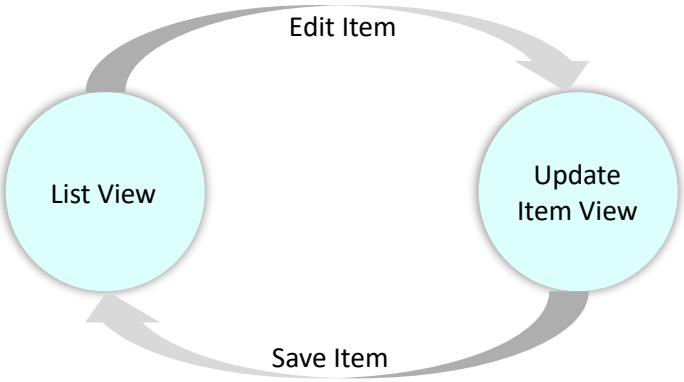
disebut REQUEST_USER_STATUS untuk memperbarui indikator daftar teman. Kemudian, tindakan tersebut menyebabkan logika bisnis memicu permintaan untuk mendapatkan daftar baru, dan kemudian, setelah selesai, memanggil tindakan lain, UPDATE_USER_STATUS, dengan respons permintaan tersebut.

Mengirimkan tindakan menyebabkan terciptanya suatu peristiwa, yang kemudian diteruskan ke logika bisnis. Dalam Redux, bit logika bisnis ini disebut pereduksi, tetapi pada dasarnya mengambil status terkini dan peristiwa, lalu mengembalikan status sistem baru yang telah dimanipulasi sesuai dengan kebutuhan tindakan tersebut. Komponen kemudian diberi tahu tentang perubahan status, dan Redux menyediakan fungsi pemetaan yang memungkinkan komponen hanya diikat ke bagian status yang benar-benar dibutuhkannya untuk memberi kita kopling longgar yang kita inginkan.

Inilah yang dimaksud dengan aliran searah; perubahan status dapat dimulai dengan tindakan yang dikirim dalam suatu komponen, tetapi perubahan status aktual hanya terjadi dalam satu arah, bukan komponen yang mengubah status atau model data secara langsung, seperti yang terjadi dalam sistem pengikatan dua arah. Redux bekerja sangat baik dengan React. React adalah pustaka JavaScript yang dikembangkan oleh Facebook yang merender ulang komponen UI saat model data yang diikatnya berubah, yang dapat sangat menyederhanakan logika. Rincian lengkap React berada di luar cakupan buku ini, tetapi pada dasarnya, React memungkinkan Anda untuk mendefinisikan komponen UI, sering kali menggunakan bahasa yang dikenal sebagai JSX, yang memungkinkan Anda untuk mengekspresikan komponen UI ini dengan cara seperti HTML.

React kemudian mengambil pohon dari komponen-komponen ini dan merendernya ke HTML. Setiap komponen dapat mempertahankan statusnya sendiri, dan diberikan properti (props) yang dapat digunakan untuk menentukan bagaimana komponen tertentu berperilaku. Sebuah komponen tidak dapat secara langsung mengubah props dari induknya hanya anak langsungnya. Jika ada bagian dari status sistem yang berubah, yang mengakibatkan perubahan pada props, React mengevaluasi ulang komponen-komponen tempat props telah berubah untuk melihat apakah mereka telah mengakibatkan perubahan apa pun pada HTML yang dihasilkan. Jika demikian, React hanya membuat perubahan yang dibutuhkannya.

Ini bisa lebih berkinerja daripada kemungkinan merender ulang seluruh DOM, atau membuat serangkaian pembaruan tambahan saat status berubah. Dengan React dan Redux, penggabungan untuk suatu komponen menjadi tindakan yang perlu dikirim ke penyimpanan, dan bagian-bagian spesifik dari status yang perlu dibaca, yang dilakukan dengan memetakan tindakan dan status ke properti komponen menggunakan connect. Logika bisnis sepenuhnya dipisahkan dari logika, yang memungkinkan React untuk mencari tahu bagaimana status dipetakan ke atribut yang dibutuhkan UI, sementara Redux menangani peristiwa.



Gambar 8.1 Mesin Status Untuk Mengedit Dan Menyimpan Item

Menggunakan sistem bus peristiwa semacam ini juga memungkinkan kita untuk mengimplementasikan mesin status untuk menangkap status tertentu dari suatu sistem, alih-alih hanya memetakan langsung ke nilai model data tertentu. Mesin status digunakan untuk memetakan aliran melalui suatu system sering kali, komponen tingkat yang lebih tinggi mengatur komponen lain berdasarkan status, dan kemudian tindakan yang dapat dipicu oleh komponen lain dapat menyebabkan transisi ke status lain.

Gambar 8.1 menunjukkan satu cara untuk menggambarkan mesin status, di mana lingkaran mewakili status dan anak panah diberi anotasi dengan tindakan yang dapat menyebabkan transisi antara status tersebut. Pada mesin status di atas, kita dapat melihat proses untuk mengedit item dalam aplikasi web. Kita mulai dalam status "Tampilan Daftar", dan memilih "Edit Item" akan menyebabkan status tersebut beralih ke "Tampilan Perbarui item", dengan item yang sedang diedit ditetapkan ke status tersebut.

Dalam "Tampilan Perbarui item", menekan item simpan menyebabkan status tersebut beralih kembali ke tampilan daftar (biasanya dengan efek samping menyimpan perubahan kembali ke basis data), tetapi Anda mungkin juga memiliki transisi "batal" yang juga beralih kembali ke "Tampilan Daftar", tetapi tanpa efek samping. Sebagian besar sistem akan memiliki mesin status di suatu tempat, dan meskipun ini sering kali dapat ditentukan melalui nilai bagian tertentu dari model data, akan berguna untuk secara langsung memodelkan status saat ini sebagai bagian dari penyimpanan data Anda, daripada mencoba memperolehnya dari nilai model dalam penyimpanan data, dan menjadi eksplisit dalam logika bisnis Anda tentang transisi, daripada membiarkannya tersirat.

Namun, dalam state machine, beberapa transisi tidak valid, jadi harus diperhatikan untuk hanya memicu tindakan yang valid dari state tertentu oleh suatu komponen. Hal ini sering dilakukan dengan membuat komponen yang dapat menyebabkan transisi tidak aktif selama state tertentu.

Menghubungkan Komponen Bersama-sama

Meskipun dimungkinkan untuk mengurangi hubungan antarkomponen, sering kali masih ada kasus di mana Anda memerlukan akses ke komponen atau pustaka lain dalam kode Anda, dan ada dua cara untuk melakukannya. Yang pertama adalah dengan mengimpor komponen atau pustaka yang Anda andalkan secara langsung, dan yang kedua adalah teknik

yang dikenal sebagai injeksi ketergantungan, di mana komponen diberi objek tertentu yang menjadi andalannya, alih-alih mengimpornya sendiri.

Mempertimbangkan pendekatan bus peristiwa dengan Redux yang dibahas sebelumnya, kasus ini juga mencakup fungsi seperti dispatcher, yang dapat digunakan untuk mengirimkan tindakan, dan nilai terikat dari penyimpanan, selain utilitas lain yang mungkin diperlukan. Untuk pustaka utilitas, yang tidak dapat dikonfigurasi, pendekatan pertama sering kali masuk akal. Misalnya kita perlu membuat pengidentifikasi unik untuk sesuatu dalam kode kita. Kita mungkin ingin melakukan hal berikut:

```
const uuid = require('uuid');

function buildCommentModel(commentText, threadId) {
    if (commentText.length === 0) {
        throw new Error('Comment is empty');
    }

return {
    id: uuid.v4(),
    body: commentText,
    thread: threadId,
    };
}
```

Sering kali, pustaka dan fungsi yang Anda impor dengan cara ini akan berfungsi—pustaka dan fungsi tersebut tidak akan memiliki status apa pun, hanya hal-hal yang dapat Anda konfigurasikan setelah mengimpor. Demikian pula, saat mengimpor kelas dari sebuah file, status akan berada di instans, bukan kelas. Pendekatan penarikan dependensi ini dapat dianggap sebagai modul yang meminta apa yang dibutuhkannya, alih-alih diberikan. Namun, ada banyak kasus saat mengimpor sesuatu dengan cara ini tidak membantu misalnya, jika modul memerlukan akses ke sumber daya bersama, atau pustaka eksternal memerlukan beberapa konfigurasi, tetapi Anda ingin mengonfigurasinya sekali, alih-alih setiap kali digunakan.

Solusi yang menggoda mungkin adalah membuat modul yang mengekspor instans kelas Anda, atau serangkaian opsi yang telah dikonfigurasi untuk fungsi Anda. Misalnya, sering kali di NodeJS kita perlu berkomunikasi dengan layanan back-end lain, dan kita mengimpor klien HTTP. Namun, khususnya di lingkungan perusahaan, bukan hal yang aneh bagi mesin untuk berada di balik proxy, atau memerlukan konfigurasi khusus mungkin kita menginginkan pencatatan terperinci di lingkungan pengembangan, tetapi tidak dalam produksi.

Jika setiap kali kita membuat permintaan HTTP, kita harus mengonfigurasi klien HTTP tersebut di setiap bagian kode yang mengimpor kelas, itu akan menghasilkan banyak kode berulang dan memerlukan banyak perubahan setiap kali kita perlu mengubah perilakunya. Pendekatan serupa dapat berupa koneksi ke basis data—akan jauh lebih mudah bagi modul

untuk sekadar mendapatkan koneksi basis data aktif daripada membuat yang baru dari kelas atau fungsi mentah. Sangat menggoda untuk hanya mengimpor instans klien HTTP yang dikonfigurasi.

Namun, ini melanggar aturan penggabungan. Modul itu sekarang secara khusus ditautkan ke instans tertentu dalam modul itu. Ini mungkin tampak baik-baik saja, tetapi jika menyangkut pengujian unit kode Anda, Anda mungkin tidak ingin menggunakan klien HTTP yang sebenarnya, karena ini dapat mengakibatkan pengujian yang sangat lambat, dan membuat pengaturan data pengujian menjadi jauh lebih sulit. Kami ingin mengganti klien HTTP tersebut dengan tiruan, tetapi ketika sebuah modul meminta ketergantungan, tidak ada cara mudah untuk benar-benar memberinya ketergantungan yang berbeda dari yang dimintanya.

Ada alat, seperti proxyquire, yang mengganti `require()CommonJS` dengan yang dapat mengembalikan opsi alternatif untuk tujuan pengujian unit, tetapi ini dapat menyebabkan masalah yang tidak jelas dan kode spaghetti, terutama jika, di lain waktu, kami ingin mengubah klien HTTP mana yang kami gunakan dalam produksi (mungkin lingkungan pengembangan dan produksi menggunakan proxy yang berbeda, atau otoritas sertifikat TLS yang berbeda). Lebih jauh lagi, pendekatan ini sebenarnya mengurangi fleksibilitas komponen (efek samping dari peningkatan kopling). Jika kami membutuhkan klien HTTP yang dikonfigurasi dengan cara yang berbeda, maka kami harus membuat modul baru yang mengonfigurasinya secara berbeda, yang menyebabkan duplikasi.

Solusi yang lebih baik adalah "injeksi dependensi", di mana modul diberikan apa yang dibutuhkan untuk melakukan pekerjaan, daripada mengimpornya secara langsung. Untuk pustaka dan komponen yang memerlukan konfigurasi agar bermanfaat, ini adalah pendekatan yang jauh lebih baik yang dapat menghasilkan kode yang lebih bersih. Bagi pengembang yang memiliki latar belakang Java, injeksi dependensi langsung membangkitkan ide tentang kerangka kerja besar untuk mengelola ini, tetapi ini tidak perlu terjadi, dan banyak aplikasi JavaScript tidak memerlukan kerangka kerja untuk mengelola ini untuk Anda.

Sering kali, memiliki fungsi "utama" yang menyiapkan semua dependensi Anda dan kemudian menghubungkannya sudah cukup. Untuk kelas, biasanya dependensi ini ditetapkan melalui konstruktor, lalu merujuknya sebagai bidang bila diperlukan:

```
class ProductCatalogueApi {
  constructor(httpClient, catalogueApiUrl) {
    this ._httpClient = httpClient;
    this._catalogueApiUrl = catalogueApiUrl
  {
  fetchCatalogueItem(id) {
    return this ._ httpClient
      .get(`${this ._ catalogueApiUrl}/product/${id} )
      .json();
  }
}
```

Dalam contoh di atas, kelas `ProductCatalogueApi` kemudian dapat diwujudkan dalam fungsi utama dan diteruskan ke pengontrol mana pun yang memerlukan akses ke katalog produk, sekali lagi menggunakan injeksi dependensi.

Untuk modul yang bertujuan agar lebih fungsional, penutupan dapat digunakan sebagai gantinya, di mana modul mengekspor fungsi yang mengambil dependensi dan kemudian mengembalikan fungsi yang melakukan pekerjaan modul tersebut, yang sekarang memiliki akses ke dependensi tersebut dalam cakupan penutupan.

```
function fetchCatalogueItemFactory(httpClient, catalogueApiUrl) {
  function fetchCatalogueItem(id) {
    return this ._httpClient
      .get(`${this ._catalogueApiUrl}/product/${id} )
      .json();
  }
  return fetchCatalogueItem;
}
```

Dengan pendekatan ini, saat menguji kelas atau fungsi Anda, Anda dapat mengaturnya dengan cara yang sama seperti kode aplikasi utama Anda, tetapi meneruskan rintisan yang kompatibel dengan data dummy sebagai dependensi.

Pengujian

Ada banyak opsi untuk menjalankan pengujian dalam JavaScript, dengan yang paling populer mendukung mekanisme arrange-act-assert yang dibahas dalam bab Pengujian. Gaya tertentu cenderung bervariasi, dengan pustaka seperti Jasmine yang memungkinkan Anda menulis pengujian, dan alat seperti Karma yang memungkinkan Anda menjalankannya dari baris perintah dan melaporkan keberhasilannya. Alat pengujian unit sebelumnya, seperti QUnit, dijalankan dengan membuka browser dan memvisualisasikan hasilnya. JSTestDriver adalah alat pengujian awal populer lainnya untuk JavaScript.

Ditulis dalam Java, alat ini memungkinkan Anda menulis pengujian yang sangat mirip dengan gaya JUnit, dan juga mengelola pelaporan dan eksekusi di browser. Alat-alat ini jarang digunakan untuk proyek baru, meskipun Anda mungkin menemukannya. Munculnya browser tanpa kepala seperti PhantomJS, dan kemudian runtime sisi server seperti NodeJS, telah menyederhanakan pengujian JavaScript. Saya akan merekomendasikan Jest sebagai alat terbaik untuk menjalankan pengujian; ini adalah pustaka pengujian dan spesifikasi lengkap yang kompatibel dengan Jasmine.

Di Jasmine dan Jest, Anda mendefinisikan suatu pengujian dengan memanggil fungsi itu, lalu memberikan nama pengujian yang dapat dibaca manusia dan fungsi yang akan dijalankan saat pengujian dijalankan.

```
const ShoppingBasket = require(' .. /lib/shopping-basket');
const Product = require(' .. /lib/product');
```

```
describe('shopping basket', () => {
  it('can be added to', () => {
    const shoppingBasket = new ShoppingBasket();
    shoppingBasket.addItem(new Product('t-shirt'));
    expect(shoppingBasket.items).toBe(1);
  });
  });
```

Metode `describe()` memungkinkan pengelompokan, dan blok pengujian di dalam `describe()` juga dapat menyertakan `describe()` lain yang bersarang di dalamnya. Nama dan deskripsi metode dimaksudkan untuk mendorong Anda menulis nama yang terinspirasi dari bahasa alami, karena saat pengujian berjalan, deskripsi semua describe secara bersamaan, dan deskripsi pengujian itu sendiri, digunakan sebagai nama pengujian keseluruhan dalam output. Dalam contoh di atas, ini akan menjadi "keranjang belanja dapat ditambahkan ke".

Saat menulis banyak pengujian untuk sebuah modul, biasanya akan menjalankan pengaturan yang sama untuk setiap pengujian, mungkin harus membuat instance dari sebuah kelas, atau menghapusnya di bagian akhir. Sebagian besar alat memungkinkan Anda untuk menentukan sedikit kode umum yang akan dijalankan sebelum dan setelah setiap pengujian (terlepas dari apakah pengujian tersebut lulus atau tidak) untuk menghindari pengulangan ini. Dalam Jest/Jasmine, hal ini dilakukan dengan memanggil `beforeEach()` dan `afterEach()` dengan sebuah callback

```
describe('shopping basket', () => {
  let shoppingBasket;

  beforeEach(() => {
    shoppingBasket = new ShoppingBasket();
  });

  it('can be added to', () => {
    shoppingBasket.addItem(new Product('t-shirt'));
    expect(shoppingBasket.items).toBe(1);
  });

  afterEach(() => {
    shoppingBasket.empty();
  });
});
```

Secara default, setiap pengujian akan dijalankan secara sinkron. Jika Anda menguji kode asinkron (misalnya, jika metode yang Anda uji mengembalikan sebuah promise), maka Anda perlu memiliki beberapa cara untuk memberi tahu pengujian bahwa pengujian harus

menunggu hingga semua kode asinkron selesai. Di Jest, Anda dapat mengembalikan sebuah promise untuk melakukan ini, dan pengujian akan berhasil jika promise tersebut berhasil, atau gagal jika promise tersebut ditolak. Mari kita asumsikan bahwa menambahkan item ke keranjang belanja sekarang menjadi operasi asinkron yang mengembalikan sebuah promise:

```
describe('shopping basket', () => {
  let shoppingBasket;

  beforeEach(() => {
    shoppingBasket = new ShoppingBasket();
});

  it('can be added to', () => {
    return shoppingBasket.addItem(new Product('t-shirt'))
      .then(()=> expect(shoppingBasket.items).toBe(1));
  });

  afterEach(() => {
    shoppingBasket.empty();
  });
});
```

Cara lain untuk melakukan ini adalah dengan memasukkan satu argument biasanya disebut done ke fungsi pengujian Anda. Anda kemudian memanggil `done()` di akhir pengujian Anda, dan `done.fail()` jika Anda perlu menggagalkan pengujian tersebut.

```
describe('shopping basket', () => {
  let shoppingBasket;
  beforeEach(() => {
    shoppingBasket = new ShoppingBasket();
  });

  it('can be added to', (done) => {
    shoppingBasket.addItem(new Product('t-shirt'))
      .then(() => {
        expect(shoppingBasket.items).toBe(1);
        done();
      });
  });
  afterEach(() => {
    shoppingBasket.empty();
  });
});
```

Jika ada bug dalam kode asinkron Anda sehingga panggilan balik atau janji tidak pernah dipanggil, Jest/Jasmine akan kehabisan waktu menunggu pengujian selesai (walaupun pesan kesalahan mungkin tidak berguna! Menguji logika sederhana dengan pengujian JavaScript dapat dilakukan dengan mudah, karena output bergantung sepenuhnya pada input, terkadang dengan beberapa efek samping pada anggota kelas lain atau dependensi lain yang dapat dimasukkan dan diganti dengan stub atau tiruan (versi palsu yang menyederhanakan perilaku sistem eksternal.

Anda dapat membaca lebih lanjut tentang ini nanti di bab ini). Komponen JavaScript yang beroperasi pada DOM dapat sulit diuji. JavaScript mungkin menganggap bahwa DOM berada dalam bentuk tertentu saat dibuat (misalnya, disediakan oleh HTML), dan DOM adalah bentuk status global. Satu pengujian mungkin membiarkannya dimanipulasi dengan cara yang dapat menyebabkan pengujian berikutnya gagal, meskipun pengujian itu seharusnya baik-baik saja. Menggunakan pustaka DOM virtual seperti React dapat mempermudah hal ini, karena DOM virtual dapat diganti dengan versi tiruan, dan pernyataan dapat dibuat pada versi itu. Ada pustaka (seperti Enzyme for React) yang dapat membantu dalam hal ini. Untuk JavaScript "vanilla" tradisional, teknik yang populer adalah dengan meneruskan elemen DOM tempat komponen beroperasi sebagai ketergantungan, daripada membiarkannya menemukannya di DOM global.

Misalnya, saat menggunakan fungsi untuk menambahkan penangan klik ke sesuatu, alih-alih fungsi menggunakan document. querySelector, argumen yang diberikan adalah objek HTMLElement untuk menambahkan penangan klik. Ini adalah penerapan injeksi ketergantungan. Dengan pendekatan ini, lapisan baru diperlukan di atas masing-masing komponen yang melakukan semua pencarian dari dokumen asli dan meneruskannya, tetapi ini memungkinkan penggunaan kembali komponen individual yang lebih besar jika diperlukan. Ini terkadang disebut wiring, karena menghubungkan masing-masing komponen bersama-sama.

Saat kita ingin menguji suatu komponen, kode pengujian kita dapat mengambil peran lapisan wiring ini. Kita kemudian dapat membuat potongan kode palsu dari DOM dalam pengujian, mungkin dengan memuat dalam file "fixture" (file HTML yang berisi potongan kode kecil dari situs yang diperlukan untuk menguji komponen ini), lalu meneruskan elemen yang dibuat ini, tanpa benar-benar melampirkannya ke dokumen global. Ini memungkinkan kita untuk mengisolasi setiap pengujian dari satu sama lain, meskipun masih ada risiko kebocoran ke status global, terutama jika hal-hal seperti pengatur waktu disiapkan.

Anda akan sering menulis cara untuk menghapus manipulasi status global apa pun misalnya, memiliki metode "`destroy()`" pada kelas yang menggunakan `clearTimeout()`, atau menggunakan kait siklus hidup seperti `componentWillUnmount` di React, bahkan jika kode destroy tidak perlu dijalankan dalam aplikasi yang sedang berjalan. Kode pengujian perlu memanggil ini, dan ini juga dapat menjadi kebiasaan yang baik untuk menulis kode nyata, karena dapat membantu Anda menghindari masalah kinerja. Bahkan jika pada awalnya di halaman nyata hanya satu contoh komponen yang akan dibuat, ini mungkin tidak selalu benar, dan kebiasaan ini akan membantu Anda di sana.

Jenis pengujian yang ditulis dengan cara ini biasanya disebut pengujian unit. Anda dapat membaca lebih lanjut tentang pengujian ini nanti di bab Pengujian, tetapi secara umum pengujian ini memungkinkan Anda menguji satu bit kode secara terpisah, dalam banyak konfigurasi berbeda (termasuk yang seharusnya bermasalah). Ada banyak manfaat pengujian unit. Selain rangkaian regresi dan memastikan bahwa kode Anda berfungsi dengan benar, pengujian unit membantu desain antarmuka modul Anda. Menerapkan pengembangan berbasis pengujian dapat membantu memfokuskan peran dan tanggung jawab modul, tetapi modul Anda sekarang secara otomatis memiliki dua hal yang menggunakannya: kode pengujian dan kode produksi. Jika kode Anda sulit diuji, desain antarmuka mungkin tidak optimal, atau mungkin terlalu banyak dikaitkan dengan modul lain.

Hal ini mungkin tidak langsung menyulitkan Anda, tetapi dapat terjadi di kemudian hari, karena kode yang memiliki karakteristik ini sering kali sulit diubah, dan mampu bereaksi cepat terhadap perubahan kebutuhan bisnis adalah salah satu manfaat nyata pengembangan perangkat lunak tangkas. Bab tentang pengujian memiliki pengantar lengkap untuk menerapkan pengembangan berbasis pengujian. Setelah Anda menulis pengujian, Anda akan memerlukan cara untuk menjalankannya. Meskipun pengujian sisi server dapat dijalankan sebagai kode di NodeJS, pengujian untuk kode sisi klien harus dijalankan di lingkungan yang mirip dengan browser agar kode yang diuji dapat berfungsi dengan baik. Ketika JQuery menonjol, ia hadir dengan kerangka pengujian unit yang dikenal sebagai QUnit.

Dengan QUnit, pengujian biasanya dijalankan secara manual di jendela browser untuk setiap browser yang perlu Anda targetkan, karena browser tentu saja merupakan lingkungan yang paling mirip browser di luar sana. Hal ini terintegrasi dengan buruk ke dalam alur kerja integrasi berkelanjutan, dan juga menambah biaya pemeliharaan. Tidak lama kemudian muncul alat otomatisasi seperti Karma, yang menjalankan pengujian secara terprogram dengan mem-boot dan mengendalikan browser dari jarak jauh. Namun, hal ini menyebabkan masalah untuk pengujian otomatis yang berjalan di server CI, yang berjalan dalam mode headless atau pada sistem Linux, bukan OS yang dibutuhkan browser.

Membuka browser untuk menjalankan pengujian menjadi lambat, terutama jika dibandingkan dengan alur kerja kode sisi server dengan kerangka kerja seperti JUnit, yang dapat menjalankan pengujian dalam milidetik dari dalam editor. PhantomJS adalah peramban "tanpa kepala", yang berbasis WebKit, yang bekerja secara lintas platform tanpa UI dan menjadi cara populer untuk menjalankan pengujian unit, karena mempercepat waktu muat peramban secara signifikan, dan karenanya pengujian itu sendiri. Perkembangan yang lebih baru telah memungkinkan kode sisi server dan sisi klien untuk digabungkan. Alih-alih mencoba memaksa peramban untuk berjalan secara terprogram, pustaka seperti JSDOM malah membuat runtime NodeJS berperilaku seperti peramban.

Meskipun mungkin tampak berbahaya untuk tidak menguji kode di lingkungan sebenarnya tempat kode akan dijalankan, risikonya sebenarnya minimal, dan manfaat dari pengujian cepat lebih besar daripada risikonya. Di peramban modern, sebagian besar bug merupakan hasil dari kesalahan logika dalam kode Anda, bukan ketidakcocokan peramban. Ada beberapa opsi bagi mereka yang ingin menjalankan kode mereka di browser yang

sebenarnya. Pengujian menyeluruh, berbeda dengan pengujian unit, umumnya dijalankan di browser yang sebenarnya. Pengujian menyeluruh cenderung lebih sedikit daripada pengujian unit, tetapi setiap pengujian akan menyentuh lebih banyak bagian sistem dan dapat memberi Anda tingkat kepercayaan yang lebih tinggi pada sistem Anda daripada pengujian unit tunggal. Pengujian menyeluruh juga lebih lambat dijalankan, dan dikombinasikan dengan beban saat memulai browser, hal ini dapat membuat pengujian ini sangat lambat dijalankan.

Menjalankan pengujian ini juga cenderung terjadi dengan cara yang berbeda dari pengujian unit Anda alih-alih kode pengujian Anda berjalan dalam konteks browser yang sama dengan kode yang diujinya, kode tersebut malah menjalankan browser yang berbeda dan mengendalikan browser tersebut dari jarak jauh (atau mungkin beberapa browser secara paralel). Ini berarti Anda benar-benar dapat menulis pengujian integrasi Anda dalam bahasa lain (Java dan Ruby adalah dua bahasa yang umum), meskipun JavaScript sendiri masih umum.

Sebuah pustaka digunakan untuk mendukung hal ini, dan sejauh ini pustaka yang paling matang dikenal sebagai Selenium. Selenium adalah pustaka Java, tetapi mendefinisikan API yang dikenal sebagai WebDriver, yang berjalan di browser. Bahasa lain memiliki pustaka seperti Selenium, yang berinteraksi dengan API WebDriver untuk mengendalikan browser dari jarak jauh untuk menjalankan pengujian. Dengan WebDriver, Anda dapat membuat pengujian dalam kerangka pengujian normal, tetapi pengujian ini kemudian menggunakan pustaka tersebut untuk mengendalikan browser dari jarak jauh dan membuat ekspektasi/pernyataan melalui pustaka tersebut seperti biasa.

```
describe('shopping basket', () => {
  it('increases the number of items shown on the page', () => {
    browser.url('http://localhost:3000/catalogue/t-shirt');
    expect(browser.getText('#cart_size')).toEqual("Empty");
    browser.click('#add_to_cart');
    expect(browser.getText('#cart_size')).toEqual("1 item");
  });
});
```

Perbedaan utama antara Selenium dan alat tiruan seperti JSDOM adalah Anda perlu mendapatkan contoh browser yang sebenarnya untuk dijalankan. Saat menjalankan pengujian ini di desktop, bukan hal yang aneh untuk melihat salinan browser memulai dan secara fisik melihat interaksi yang dikendalikan dari jarak jauh terjadi. Di lingkungan CI, lingkungan GUI harus disediakan untuk memungkinkan hal ini terjadi. Ini tidak selalu mudah, dan memelihara rangkaian OS dan browser yang berbeda untuk menyediakan cakupan penuh dapat menjadi beban yang cukup besar bagi suatu organisasi.

Beberapa penyedia SaaS menawarkan browser "berbasis cloud" bayar sesuai penggunaan pada infrastruktur yang mereka kendalikan, yang dapat membantu dalam beberapa keadaan (tetapi tidak yang lain misalnya, jika Anda ingin memeriksa intranet atau mengunci lingkungan pengembangan Anda ke IP tertentu). Masalah umum dengan rangkaian pengujian menyeluruh adalah keandalan, khususnya di seluruh browser.

Di browser, hampir semua tindakan harus diperlakukan sebagai asinkron, yang dapat membuat penulisan pengujian menjadi sulit. Cara umum untuk mengatasi hal ini, khususnya dalam bahasa yang tidak memiliki dukungan sebanyak JS untuk fungsi asinkron, adalah dengan menambahkan penundaan pada pengujian. Misalnya, membuka kotak dialog memiliki animasi 200 ms; kode seperti ini umum ditemukan:

```
describe('control panel', () => {
  it('opens when clicked', (done) => {
    browser.url('http://localhost:3000/')
    browser.click('#open_control_panel')
    setTimeout(() => {
      expect(browser.isVisible('#control_panel')).toBeTruthy();
      done();
    }, 250);
  });
});
```

Waktu tunggu di atas cobeparnya adalah 250 me bukan 200 untuk mopcakomodaci h
Waktu tunggu di atas sebenarnya adalah 250 ms, bukan 200, untuk mengakomodasi beberapa overhead atau varians dalam mesin rendering browser. Pendekatan ini benar-benar tidak cocok untuk tindakan yang penundaannya tidak ditentukan secara pasti, seperti untuk panggilan AJAX.

Sering kali, penundaan adalah angka berdasarkan skenario terburuk, yang mungkin masih belum 100% dapat diandalkan, dan dapat menyebabkan kelambatan jika panggilan selesai lebih cepat. Pendekatan alternatif dikenal sebagai "menunggu", di mana Anda secara berkala memeriksa apakah suatu tindakan telah selesai, dan kemudian melanjutkannya segera setelah selesai. Jika tidak, setelah batas waktu tercapai, tindakan tersebut gagal.

```
describe('control panel', () => {
  it('opens when clicked', (done) => {
    browser.url('http://localhost:3000/')
    browser.click('#open_control_panel')
    browser.waitForVisible('#control_panel');
    expect(browser.isVisible('#control_panel')).toBeTruthy();
  });
});
```

Hal ini memungkinkan Anda menulis pengujian yang lebih cepat dan lebih tangguh, dan juga membantu memisahkan kode Anda dari implementasi. Misalnya, jika kecepatan animasi Anda menjadi sedikit lebih lambat, Anda tidak perlu memperbarui kode pengujian Anda, yang harus Anda lakukan jika Anda menggunakan sleep dengan interval hard-coded. Pendekatan lain untuk memisahkan kode pengujian Anda dari detail implementasi spesifik lebih jauh dikenal sebagai model objek halaman.

Selenium menggunakan kelas CSS (atau ID) untuk menemukan item pada halaman, tetapi terkadang kelas tersebut perlu diubah (jika perubahan gaya besar sedang berlangsung), tetapi perilaku dan struktur komponen yang sebenarnya tetap sama. Alih-alih menulis beberapa pengujian yang merujuk ke komponen yang sama dengan nama kelas, Anda dapat merangkum semua logika di dalam kelas helper, lalu membuat pernyataan tentang kelas tersebut. Jika ada informasi tentang objek pada halaman yang berubah, maka definisi di kelas tersebut akan diperbarui dan semua kelas menggunakannya. Ini adalah penerapan pola DRY (don't repeat yourself) yang baik. Saat menjalankan pengujian integrasi, Anda juga harus mempertimbangkan cara mengontrol status aplikasi di balik layar.

Dalam pengujian unit, teknik seperti mocking dapat digunakan, tetapi dengan pengujian semacam ini, Anda sering kali menjalankan aplikasi sebagai server mandiri, yang tidak dapat Anda akses saat runtime (meskipun memulai instans server dalam kode pengujian juga merupakan kemungkinan, dan memungkinkan Anda untuk langsung memanipulasi status kode sisi server). Salah satu cara untuk mengatasinya adalah dengan mengarahkan aplikasi Anda ke database atau API palsu yang dapat Anda kendalikan, alih-alih yang asli, yang memudahkan pengujian integrasi aplikasi Anda dalam beberapa cara. Pendekatan menarik lainnya adalah dengan benar-benar menjalankan pengujian ini terhadap situs web asli Anda.

Beberapa orang lebih menyukai ini karena menawarkan tingkat kepercayaan yang tinggi pada versi situs web yang benar-benar akan dilihat audiens Anda, tetapi tidak selalu tepat. Misalnya, membuat pesanan asli di situs web e-commerce bisa jadi sulit untuk diuji. Terkadang, situs web pra-pementasan digunakan sebagai gantinya. Biasanya, situs web asli harus dilengkapi dengan beberapa data uji yang dapat menyebabkan kebingungan jika pengguna asli menemukannya secara tidak sengaja (lihat pengujian "Webdriver Torso" YouTube), atau bergantung pada keberadaan bit konten tertentu yang, jika diubah, dapat merusak pengujian Anda.

Alat Bangun

Seperti yang disebutkan sebelumnya, jarang sekali menulis JavaScript yang kemudian dieksekusi oleh browser tanpa transformasi sementara. Bahkan ketika Anda melakukannya, tetaplah baik untuk memiliki beberapa perkakas otomatis yang dapat membantu Anda menjalankan pengujian atau menerapkan pemeriksaan kualitas lainnya ke basis kode Anda. Pada saat penulisan ini, tidak ada alat build standar untuk JavaScript, dengan beberapa kerangka kerja populer untuk melakukannya. Namun, sebagian besar toolchain build memiliki kesamaan, dan cenderung membantu dalam tiga cara: mengelola dependensi, memeriksa kualitas kode, dan menghasilkan produk akhir (sebuah "bundel"), yang dieksekusi oleh browser.

Merupakan hal yang umum untuk menyatukan berbagai alat untuk memenuhi semua fungsi ini dalam sebuah toolchain, meskipun beberapa melakukan lebih dari yang lain. Ketika berbagai alat perlu digabungkan, diperlukan lapisan koordinasi. Ini bisa sesederhana skrip shell yang menjalankan berbagai perintah secara berurutan, atau sesuatu yang lebih canggih (seperti Gulp atau Grunt) yang mendukung paralelisasi dan rantai yang lebih kompleks, serta mengawasi perubahan dan menjalankan kembali build selama pengembangan. Tujuan umum

dari alat build adalah untuk menggabungkan beberapa file JavaScript menjadi satu, menjalankan transpilasi yang diperlukan, dan menyelesaikan dependensi dari kode yang diimpor baik modul lain dalam sistem yang sama atau kode pihak ketiga yang diinstal dari repositori.

Satu file tunggal kemudian dapat dikirimkan ke browser dalam satu permintaan, yang dapat menawarkan beberapa peningkatan kinerja (meskipun HTTP/2 menawarkan cara alternatif untuk meniadakan dampak kinerja dari membuat beberapa permintaan dari browser). File ini juga sering diperkecil, terutama dalam produksi, yang berarti bahwa semua hal yang manusiawi, seperti spasi, dihilangkan dan variabel dibuat lebih kecil untuk lebih mengurangi ukuran file. Kelemahannya adalah dapat mempersulit debugging, karena browser hanya melihat kode yang diperkecil. Peta sumber diciptakan untuk mengatasi hal ini, dan peta sumber menyediakan cara bagi alat pengembang browser untuk menghubungkan kode yang diperkecil dan diubah kembali ke aslinya.

Merupakan hal yang umum bagi alat build untuk memiliki mode "pengembangan" dan "produksi" yang terpisah untuk proses ini, karena peta sumber dapat secara signifikan meningkatkan ukuran file, sehingga peta sumber hanya disertakan pada build pengembangan lokal. Meskipun sering kali bukan bagian dari alat build itu sendiri, test runner, dependency manager, dan code quality checker sering kali digabungkan menjadi satu toolchain yang sama. Proyek JavaScript awal akan menyimpan semua dependensi dalam repositori kode bersama kode untuk aplikasi ini (proses yang dikenal sebagai "vendoring," karena lokasi umum untuk file-file ini berada di subdirektori "vendor"), atau terkadang menggunakan fitur seperti SVN eksternal atau submodul Git.

Juga umum (dan masih umum hingga saat ini) untuk menyertakan versi pustaka umum yang dihosting dari jarak jauh seperti JQuery sebagai tag skrip di halaman Anda dan menganggap dependensi tersebut ada di sana. Pada tahun 2012, Twitter memperkenalkan alat yang dikenal sebagai "Bower," yang menambahkan perkakas pada populasi direktori vendor, dengan menentukan dependensi dan versi mana yang Anda perlukan dalam file konfigurasi dan mengunduhnya secara otomatis. Seiring dengan semakin populernya NodeJS, alat yang dikenal sebagai Node Package Manager (NPM) menjadi populer di sampingnya, yang mengelola dependensi JavaScript untuk kode sisi server. Tidak lama kemudian, ini menjadi cara paling umum untuk mendistribusikan kode JavaScript yang ditujukan untuk dieksekusi di browser juga.

NPM (dan alat terkait, seperti Yarn) sekarang dianggap sebagai cara standar untuk menyertakan dependensi ke dalam JavaScript Anda. Alat pembangunan umum akan memahami cara mengatasi dependensi yang telah diinstal menggunakan NPM, baik menggunakan impor ES6 atau sintaksis require CommonJS. Pemeriksa gaya dan kualitas kode telah lama dikenal sebagai teman yang ampuh dalam penulisan kode yang dapat dipelihara. Pemeriksa gaya pertama, yang disebut "lint," diciptakan untuk bahasa C, dan "linter" kini menjadi istilah yang digunakan untuk merujuk ke alat yang sama dalam bahasa lain. Ada beberapa linter untuk JavaScript, dengan ESLint menjadi salah satu yang paling modern dan umum.

Seiring berjalannya waktu, linter tidak hanya bertanggung jawab untuk memeriksa konsistensi gaya, tetapi juga untuk potensi kesalahan dalam logika (seperti menggunakan variabel yang mungkin tidak memiliki nilai yang ditetapkan padanya). Alat pemeriksa tipe seperti Flow, seperti yang dibahas sebelumnya, juga sering diintegrasikan ke dalam rangkaian alat pembuatan di sini. Gaya kode juga menjadi sumber dari banyak perdebatan sengit dalam tim pengembangan, yang dapat tercermin dalam beragam pilihan konfigurasi yang tersedia untuk alat linting. Untuk banyak bahasa lain, badan yang bertanggung jawab untuk menerbitkan bahasa tersebut juga akan menerbitkan panduan gaya yang direkomendasikan, seperti PSR-2 PHP, atau PEP-8 Python. Untuk JavaScript, tidak ada satu standar, tetapi beberapa gaya populer. Saya sarankan untuk memilih salah satu gaya tersebut dan tetap menggunakannya, sebagai pendekatan yang lebih sederhana daripada menentukan gaya internal Anda sendiri.

Melakukan hal itu akan menghemat banyak diskusi yang panas, dan sering kali tidak perlu, di antara anggota tim Anda mengenai preferensi pribadi! Beberapa di antaranya akan tampak serupa dengan peran alat build yang digunakan untuk bagian lain dari front end, dan dalam kasus ini, alat tersebut sering digabungkan menjadi satu alat atau task runner yang menangani keduanya.

8.8 KESIMPULAN

Meskipun awalnya merupakan bahasa untuk memanipulasi halaman web, JavaScript kini juga sering digunakan di sisi server. Untuk saat ini, JavaScript tetap menjadi bahasa dominan di Web untuk kode sisi klien, jadi harus menjadi alat yang lebih tajam untuk setiap pengembang web full stack. JavaScript adalah bahasa yang berkembang pesat. Setelah periode stagnasi yang panjang, dengan beberapa browser menambahkan ekstensi yang tidak kompatibel yang menjadi bagian ad-hoc dari bahasa tersebut, tahun 2015 menyaksikan diperkenalkannya edisi keenam dari spesifikasi bahasa tersebut, yang disebut ES6, dan juga peralihan ke model tahunan (jadi ES6 adalah ES2015), dengan iterasi yang lebih kecil terus-menerus ditambahkan.

Selain digunakan untuk penyempurnaan sederhana pada suatu halaman, JavaScript sekarang dapat digunakan untuk aplikasi web yang sangat kaya dan kompleks. Sebagai sebuah bahasa, JavaScript memiliki beberapa fitur yang unik. Salah satunya adalah sifatnya yang asinkron. Runtime JavaScript menjalankan kode dalam satu utas, dengan operasi I/O yang terjadi di latar belakang, dan fungsi yang terdaftar sebagai pengendali peristiwa atau panggilan balik yang terjadi saat peristiwa UI atau proses I/O latar belakang terjadi. JavaScript memiliki beberapa fitur untuk membantu menyederhanakan penggunaan panggilan balik, dan khususnya panggilan balik berantai: promises dan, yang terbaru, kata kunci async dan await.

Di browser, JavaScript berinteraksi dengan UI menggunakan API Document Object Model (DOM). Karena kode JavaScript dijalankan oleh berbagai macam browser yang berpotensi beragam dan berumur panjang, hal ini dapat membatasi Anda pada penyebut umum terendah dari bahasa tersebut, tetapi ada alat yang mengatasi hal ini dengan menyediakan fitur bahasa baru di browser lama (polyfill), atau dengan mengubah sintaksis

yang lebih baru menjadi bentuk yang kompatibel dengan versi sebelumnya (transpilasi). Beberapa pustaka, seperti JQuery, juga secara historis telah digunakan untuk menyediakan API umum melalui beberapa implementasi yang tidak kompatibel dari fungsionalitas yang sama.

Di server, runtime NodeJS digunakan untuk menjalankan kode, tetapi dengan pustaka standar yang berbeda, dan khususnya tanpa DOM. NodeJS juga memiliki teknik yang memungkinkan modul JavaScript disertakan untuk mengelola dependensi, menggunakan teknik yang dikenal sebagai CommonJS. Di browser, definisi modul asinkron (AMD) umum digunakan untuk mendefinisikan dan memuat modul dan namespace lain, tetapi kode CommonJS dapat diubah ke dalam bentuk yang kompatibel dengan browser modern. Tipe modul JavaScript asli kini tersedia, yang dapat digunakan tanpa dukungan alat tambahan apa pun.

JavaScript mendukung sekumpulan kecil tipe dasar, dengan fungsionalitas yang lebih kompleks yang dikembangkan dengan menggabungkannya. "Gotcha" umum dengan JavaScript adalah operator kesetaraan (==), yang akan memaksa kedua belah pihak ke dalam tipe yang sama untuk membandingkannya, menghasilkan hasil yang mengejutkan. Anda hampir selalu ingin menggunakan operator identitas (===), yang berperilaku dengan cara yang tidak terlalu mengejutkan. Pendekatan orientasi objek JavaScript juga berbeda dari bahasa lain, menggunakan pewarisan berbasis prototipe, bukan berbasis kelas, meskipun ES6 memungkinkan Anda bekerja dalam sintaksis yang lebih familiar. JavaScript juga mendukung beberapa primitif dasar untuk pemrograman fungsional—yang terpenting, fungsi anonim yang dapat diteruskan sebagai objek kelas satu.

Aturan cakupan JavaScript juga sering mengejutkan, di mana variabel yang didefinisikan menggunakan var dapat mengalami "pengangkatan", yang terjadi saat definisi dievaluasi sebelum kode. Alternatif modern let dan const lebih disukai untuk kode yang lebih baru. Kesalahan terbesar JavaScript lainnya adalah kata kunci this. Dalam bahasa berorientasi objek lainnya, this merujuk ke objek tempat metode berada, tetapi dalam JavaScript merujuk ke konteks tempat metode dipanggil (berpotensi sebagai metode, tetapi mungkin juga sebagai panggilan balik peristiwa). Penutupan atau pengikatan dapat digunakan untuk memastikan bahwa Anda memiliki akses ke kata kunci this atau yang setara saat menambahkan pengendali peristiwa atau panggilan balik.

Pengendali peristiwa dan panggilan balik adalah cara umum untuk berkomunikasi antarkomponen, dan ada banyak pustaka dan kerangka kerja untuk membantu menyusun kerangka kerja pemrosesan peristiwa, serta pengikatan antara model dan kode UI. Sebagai sebuah bahasa, JavaScript memiliki dukungan alat yang baik untuk menulis dan menjalankan pengujian otomatis. Ini termasuk pengujian unit, yang menguji bagian tertentu dari sebuah aplikasi, dan pengujian integrasi, yang berjalan di lingkungan browser penuh. Kerangka kerja pengujian unit JavaScript banyak mengambil dari prinsip-prinsip umum pengujian unit, meskipun sering kali terstruktur menggunakan pernyataan describe dan it, bukan sebagai metode pada kelas.

Cara menyusun pengujian ini mencerminkan sifat JavaScript yang bukan bahasa OO murni, prosedural, atau fungsional. Meskipun memungkinkan untuk menulis JavaScript secara

langsung untuk dieksekusi oleh browser, lebih umum untuk menggunakan alat untuk menerjemahkan JavaScript ke dalam bentuk yang kompatibel dengan versi lama dan dioptimalkan untuk digunakan di browser web. Alat-alat pembuatan ini juga bekerja dengan alat manajemen dependensi seperti NPM, di mana pustaka JavaScript pihak ketiga dapat digunakan dan digabungkan ke dalam aplikasi akhir Anda. JavaScript sering kali merupakan bahasa yang tidak sempurna. Namun, JavaScript adalah bahasa web, dan sebagai pengembang web, memiliki pemahaman yang baik tentang cara mengembangkan perangkat lunak dalam JavaScript akan membantu Anda memenuhi persyaratan teknis sebagai pengembang tumpukan penuh.

BAB 9
AKSESIBILITAS

9.1 PRINSIP DAN PENTINGNYA AKSESIBILITAS DALAM PENGEMBANGAN WEB

Aksesibilitas tampak misterius bagi banyak orang, tetapi konsep dasar di baliknya sama dengan kegunaan seseorang yang mengunjungi halaman web harus dapat membaca kontennya dengan mudah, dan mereka yang membuka aplikasi web harus dapat menggunakannya. Situs web yang dapat diakses adalah situs yang juga dapat digunakan oleh semua orang yang mungkin mencoba mengaksesnya. Faktanya, banyak fitur aksesibilitas digunakan secara opsional oleh orang-orang yang berbadan sehat navigasi keyboard dapat lebih cepat daripada mouse untuk mengisi formulir, dan subtitle dalam video dapat berguna bahkan jika Anda tidak mengalami gangguan pendengaran.

Aksesibilitas mencakup berbagai topik, dan seperti sebagian besar pengembangan perangkat lunak, aksesibilitas jarang ada dalam ruang hampa. Aksesibilitas sering digunakan untuk berbicara tentang teknologi bantuan (AT), seperti pembaca layar; atau ukuran font yang lebih besar; atau pertimbangan khusus untuk orang-orang yang memiliki disabilitas; tetapi lebih dari itu. Bandingkan membangun situs web dengan membangun gedung, misalnya semua gedung baru harus dapat diakses. Aspek yang paling terlihat dari hal ini mungkin adalah jalur landai untuk pengguna kursi roda, tetapi lebih dari itu. Papan tanda mungkin memiliki alternatif braille, dan mungkin memiliki teks besar dengan kontras tinggi bagi mereka yang kesulitan melihat.

Lift akan mengumumkan lantai mereka dengan jelas, dan tidak akan ada lampu sorot atau lampu berkedip yang mungkin "terlihat bagus" tetapi berisiko memicu epilepsi. Banyak dari detail kecil ini bukan sekadar detail yang ditambahkan di bagian akhir oleh spesialis aksesibilitas. Dapatkah Anda membayangkan situasi di mana sebuah gedung dirancang dan dibangun, lalu sebelum dibuka seorang spesialis datang dan meminta pemasangan jalur landai dan mengganti semua papan tanda di gedung tersebut? Ketika situs web dibangun dengan cara itu, mudah untuk melihat bagaimana aksesibilitas dapat diabaikan sebagai biaya tambahan dan diabaikan dalam proyek yang dibatasi anggaran.

Hal ini terutama berlaku jika ada masalah mendasar yang tidak teridentifikasi dengan aksesibilitas situs Anda; membiarkannya sampai akhir dapat mengakibatkan hanya menambal masalah yang dangkal, atau sejumlah besar pengerjaan ulang. Mendesain dan membangun situs web agar dapat diakses sejak awal membutuhkan sedikit usaha ekstra dibandingkan dengan menambahkannya nanti. Aksesibilitas dapat dianggap sebagai persyaratan lintas fungsi, yang berarti aksesibilitas merupakan sesuatu yang harus dibangun dalam setiap fitur atau cerita yang Anda buat, bukan fitur mandiri yang ditambahkan di akhir proyek. Memang, sebagai seorang insinyur perangkat lunak, Anda memiliki kewajiban etis dan hukum untuk tidak mengecualikan golongan orang dari aplikasi Anda berdasarkan faktor yang tidak dapat mereka kendalikan.

Untungnya, aksesibilitas bukanlah debu ajaib yang diterapkan, tetapi sebenarnya

merupakan serangkaian prinsip yang cukup kecil yang didukung oleh API yang baik yang tersedia untuk semua pengembang web. Secara default, semua halaman dapat diakses kami membangun kompleksitas dengan menambahkan gaya dan skrip kaya yang dapat menyebabkan ketidakaksesan. Ada pedoman untuk memeriksa aksesibilitas halaman, tetapi seperti banyak alat otomatis, ini bersifat umum dan mungkin tidak sesuai dengan konteks aplikasi spesifik Anda. Mematuhi pedoman itu baik, tetapi tidak cukup untuk memastikan situs Anda benar-benar dapat diakses.

Aksesibilitas Sejak Awal

Aksesibilitas dimulai dari desain situs web. Jika teks terlalu kecil, atau jika kontras warnanya buruk (misalnya, abu-abu muda di atas putih), maka situs web akan sulit digunakan oleh orang yang sehat bugar, apalagi mereka yang mungkin memiliki kesulitan penglihatan. Membangun elemen desain semacam ini agar mudah diakses akan bermanfaat untuk membantu semua orang. Bahkan mereka yang memiliki penglihatan sempurna yang mungkin baik-baik saja dengan sesuatu yang sulit bagi orang yang penglihatannya terbatas dapat memperoleh manfaat jika desainnya dikerjakan ulang agar dapat dibaca oleh semua audiens.

Jenis aksesibilitas ini lebih dari sekadar membantu mereka yang memiliki disabilitas tertentu. Wajar jika penglihatan memburuk seiring bertambahnya usia, dan bahkan orang dengan penglihatan normal yang lebih tua mungkin kesulitan dengan sesuatu yang dianggap dapat diterima oleh orang yang lebih muda. Masalah ini sering kali membingungkan karena tim pengembangan sering kali relatif muda, karena industri kita secara umum masih muda, sehingga masalah ini sering kali terlewatkan.

Lebih jauh lagi, batasan aksesibilitas, kegunaan, dan pengalaman pengguna dapat mulai kabur. Misalnya, orang yang mengalami disleksia atau kesulitan belajar mungkin merasa teks yang padat sulit dipahami. Penggunaan judul yang tepat, yang seharusnya berfungsi untuk menunjukkan hierarki informasi, dapat membuat halaman dapat digunakan oleh audiens ini, serta memberi manfaat bagi pengguna lain dengan memudahkan pemindaian halaman untuk menemukan konten tertentu.

Pengembangan yang mudah diakses semacam ini dilakukan pada tahap desain, sering kali sebelum pengembang terlibat. Namun, hasil akhir dari proses pengembangan selalu menjadi tanggung jawab semua orang, dan menemukan masalah aksesibilitas sejak dini dapat mempermudah perbaikannya. Saat bekerja dengan desainer untuk membuat kerangka kerja atau spesifikasi untuk diterapkan, pandangan kritis terhadap masalah aksesibilitas dapat sangat membantu.

9.2 BEKERJA DENGAN TEKNOLOGI PENDUKUNG

Saat penerapan fitur atau cerita baru dimulai, peran pengembang sering kali berkisar pada memastikan halaman web tetap kompatibel dengan teknologi pendukung, yang dapat menjelaskan mengapa AT dan aksesibilitas sering kali digabungkan. Namun, memfasilitasi kompatibilitas ini bukan sepenuhnya tugas pengembang. Terutama saat menerapkan aplikasi web, khususnya pola UX yang kompleks mungkin memerlukan penerapan alternatif dan metode interaksi tertentu, yang dapat memerlukan kolaborasi lebih lanjut dengan desainer

interaksi untuk menentukan cara kerja interaksi tersebut di lingkungan yang terbatas ini.

Namun seperti mengatasi masalah aksesibilitas lainnya, hal ini juga dapat menguntungkan semua pengguna. Ketika hukum Inggris diubah untuk mewajibkan trotoar yang diturunkan di tempat penyeberangan pejalan kaki, hal ini sebagian besar dilakukan untuk menguntungkan pengguna kursi roda. Namun, hal ini juga diapresiasi oleh pengguna yang tidak diinginkan lainnya, seperti orang tua dengan kereta bayi, atau pemain skateboard, di mana perubahan ketinggian antara jalan dan trotoar sebelumnya merupakan tantangan untuk diatasi. Menambahkan pintasan keyboard ke aplikasi web yang sebagian besar bergantung pada tombol klik dan gerakan mouse dapat menguntungkan pengguna yang ahli, di mana pintasan dapat dipelajari untuk mempercepat tugas-tugas umum.

Anda tidak akan terkejut mengetahui bahwa peningkatan progresif adalah cara yang efektif untuk membuat halaman web berfungsi dengan baik dengan teknologi bantu. Dengan memulai dengan dasar halaman sederhana dan kemudian melapisinya dengan peningkatan tambahan, Anda selalu mempertahankan dasar yang dapat diakses itu, dan yang perlu Anda lakukan selanjutnya adalah memastikan bahwa Anda tidak merusak aksesibilitas dasar tersebut dengan peningkatan ini. Manfaat lain dari HTML yang dapat diakses secara default adalah jika Anda menggunakan HTML dengan cara yang dirancang untuk digunakan, maka peningkatan yang Anda bawa juga akan dapat diakses dengan sedikit kerja ekstra.

Hal utama yang perlu diperhatikan jika Anda membangun mekanisme interaksi yang sangat disesuaikan (dan ini sering terjadi saat Anda membangun aplikasi web) adalah Anda perlu menggunakan beberapa cara untuk menunjukkan kepada teknologi bantu apa makna mendasar dari elemen-elemen tersebut. Teknologi yang disebut ARIA dapat digunakan untuk melakukan ini, dan ARIA digunakan dengan menambahkan atribut ke HTML untuk menunjukkan apa peran elemen-elemen khusus ini.

Alasan alat seperti ARIA berguna adalah karena banyak teknologi bantu, terutama yang mendukung pengguna yang tidak dapat melihat, bekerja dengan mengevaluasi struktur halaman. Secara default, alat ini akan memahami arti elemen default HTML. Dari perspektif visual murni, mungkin menggoda untuk menjadikan semuanya sebagai dan menerapkan gaya dengan tepat. Ini dapat mengatasi masalah dengan gaya default browser yang berbenturan dengan tombol atau tag heading, misalnya, tetapi sebenarnya merupakan antipola yang dikenal sebagai div soup (atau tag soup), karena semua informasi sekarang sama sekali tidak terstruktur dan sulit dipahami oleh alat.

HTML semantik adalah nama yang diberikan untuk teknik penggunaan HTML sebagaimana seharusnya heading harus ditandai sebagai h1, h2, h3, dst., dan paragraf sebagai tag p. Ini seharusnya masuk akal, tetapi bisa jadi sulit dilakukan dengan benar. Misalnya, jika Anda memiliki sesuatu yang secara visual tampak seperti tombol, dan membawa Anda ke halaman lain, mungkin menggoda untuk menggunakan tag <button> untuk mendapatkan gaya visual tombol. Ini dapat menyebabkan masalah, karena orang yang menavigasi tautan dengan keyboard (serta mesin pencari yang mungkin mengikutinya) sekarang tidak akan dapat melihat `<button>` sebagai tautan. Ini berfungsi dua arah.

Jika Anda memiliki sesuatu yang tampak seperti teks sebaris yang memicu tindakan

saat ditekan (tetapi tidak benar-benar menavigasi ke halaman penuh), itu seharusnya berupa `<button>` yang ditata agar tampak seperti tautan. Hal ini terutama berlaku saat peningkatan progresif diterapkan, sehingga `tag <a>` akan memuat ulang halaman penuh dengan benar, meskipun mungkin ditingkatkan secara progresif untuk menggunakan AJAX guna menghadirkan pembaruan parsial. Ada jenis alat lain yang menganalisis struktur alih-alih tampilan visual halaman: bot pencarian. Situs web yang mudah diakses sering kali juga "dioptimalkan untuk mesin pencari" (SEO). Meskipun ada teknik SEO yang melibatkan lebih dari sekadar penggunaan HTML semantik, Google khususnya akan menghargai halaman HTML yang terstruktur dengan baik.

Namun, peringatan harus diterapkan di sini. Tidak seperti peramban web, alat AT sering kali bersifat hak milik dan berbayar, yang berarti hanya ada sedikit insentif untuk melakukan pemutakhiran, yang berarti menghabiskan lebih banyak uang. Karena alasan ini, pemahaman tentang tag dan semantik baru jauh tertinggal dari teknologi yang akan mendukungnya. Misalnya, semantik yang ditentukan dari `<section>` sejauh ini adalah untuk mengatur ulang semua level tajuk, tetapi dukungan ini di pembaca layar sangat tidak merata. Meskipun memiliki `<h1>` di bagian atas setiap `<section>` mungkin tampak sepenuhnya sesuai dengan spesifikasi, itu tidak kompatibel dengan HTML4, dan beberapa perangkat lunak pembaca layar akan menafsirkannya secara tidak benar.

Penggunaan semacam ini sekarang secara resmi tidak dianjurkan, tetapi berhati-hatilah saat berhadapan dengan elemen struktural semacam ini, terutama yang baru muncul. Ada juga kasus di mana elemen HTML biasa (bahkan saat diberi gaya) tidak sesuai. Mungkin tidak ada yang sepenuhnya memenuhi apa yang ingin Anda lakukan, jadi Anda membangun komponen dari awal menggunakan tombol dan div. Spesifikasi Aplikasi Internet Kaya yang Dapat Diakses dari Web Accessibility Initiative (biasanya disebut WAI-ARIA atau hanya ARIA) dapat menjadi alat yang berguna dalam hal ini. ARIA memungkinkan Anda menambahkan petunjuk yang menunjukkan semantik dan struktur elemen-elemen ini. Ambil contoh, tampilan tab pada halaman web.

Umumnya, ini diterapkan dengan daftar yang diikuti oleh serangkaian div yang dibuat terlihat secara selektif berdasarkan item daftar mana yang dipilih. Dengan sendirinya, struktur ini tidak memberikan informasi yang cukup dalam HTML untuk melakukan hal yang benar, dan CSS sering digunakan untuk memberi gaya agar terlihat seperti struktur tab yang sudah dikenal yang mungkin diharapkan pengguna. Mirip dengan cara kita menggunakan CSS untuk menata daftar agar tampak seperti tab sehingga pengguna tahu bahwa itu adalah tab, ARIA memungkinkan kita untuk memberi daftar sebuah "peran" sehingga pengguna non-visual juga tahu tab mereka. Dalam kasus tab, daftar akan diberi peran tablist dengan menambahkan atribut HTML5 dari peran, misalnya, `<ul class="tabs-list" role="tablist">`. Anda juga akan memberi tautan ke setiap tab peran tab dan div yang berisi konten aktual peran tabpanel.

Bagian terakhir yang hilang adalah untuk mendeskripsikan hubungan antara div dan item yang sesuai dalam daftar. aria-describedby adalah atribut lain yang dapat membantu di sini, dan digunakan dengan menambahkan ke konten div untuk merujuk ke item daftar yang

mendeskripsikan konten ini. Kelemahannya di sini adalah ia merujuk ke elemen lain dengan ID, dan penggunaan atribut id dalam HTML sering kali tidak disukai (karena dapat membatasi penggunaan ulang dan menyebabkan masalah dengan spesifisitas dalam CSS), tetapi pemilihan ID yang cermat untuk tujuan aksesibilitas dapat menghindari masalah penggunaan ulang dan menjaga agar tetap dapat dikelola.

Perlu dicatat bahwa penggunaan ARIA secara berlebihan terkadang dapat menimbulkan lebih banyak masalah daripada yang dapat dipecahkannya. Anda dapat membuat elemen dapat diakses dan masuk akal tanpa harus memberikan semua atribut ARIA. ARIA paling membantu jika digunakan untuk mengisi celah dan menambahkan petunjuk tambahan saat struktur HTML standar gagal. Ada banyak jenis atribut ARIA yang dapat digunakan untuk menambahkan makna dan struktur semantic lebih banyak dari yang dapat dijelaskan di sini. Untungnya, ada sekelompok pakar aksesibilitas khusus dalam komunitas pengembangan web yang telah menerbitkan banyak panduan dan dokumentasi tentang hal ini. Hal terpenting yang harus dilakukan adalah menyertakan pengujian dengan AT saat Anda mengembangkan situs web, lalu mencoba menemukan pola yang tepat untuk digunakan guna memperbaiki masalah aksesibilitas yang Anda temukan.

Berurusan dengan UI Interaktif

Meskipun ARIA dan HTML semantik akan sangat berguna untuk situs web bergaya konten, aplikasi web dicirikan dengan memiliki jenis interaktivitas yang jauh lebih kaya. Untuk teknologi bantuan, ini dapat menimbulkan masalah, karena ini bukan lagi sekadar menganalisis struktur halaman dan kemudian menavigasinya, karena strukturnya dinamis dan berubah. Situasinya tidak seburuk kedengarannya alat AT memang menginterogasi DOM secara dinamis, sehingga setiap perubahan tercermin di dalamnya.

Masalah utamanya adalah seputar pemberitahuan kepada pengguna saat suatu tindakan atau peristiwa penting lainnya telah terjadi (ini sering kali dilakukan secara visual, seperti dengan kesalahan validasi di bawah bidang formulir), memastikan bahwa Anda dapat memicu semua interaksi yang Anda perlukan menggunakan alat seperti papan ketik, dan memastikan bahwa jika konten DOM berubah, pengguna tidak kehilangan tempatnya. Untuk menangani perubahan dinamis pada konten, kita dapat menambahkan atribut tambahan ke konten kita untuk menunjukkan elemen mana yang dinamis dan memberikan konteks kepada pembaca layar untuk membantu pengguna menavigasinya.

Ada dua atribut di sini yang berguna: atribut role dan aria-live. role lebih disukai, terkadang bersama dengan aria-live karena alasan kompatibilitas, karena memberikan gambaran yang lebih baik tentang konteks: aria-live hanya mengatakan "elemen ini akan diperbarui," sedangkan role mengatakan jenis informasi apa yang sedang disajikan. Role berikut berguna untuk menunjukkan area konten yang dapat berubah;

- `role="alert"`
- `role="status"`
- `role="log"`
- `role="timer"`
- `role="progressbar"`

Alert mungkin merupakan peran yang paling umum, dan dibaca segera saat halaman dimuat, elemen yang memuatnya ditambahkan ke DOM, atau atribut ditambahkan ke elemen yang sudah ada (misalnya, menyorot instruksi yang mungkin terlewat saat melengkapi formulir). Penggunaannya yang paling umum adalah untuk pesan kesalahan misalnya, selama validasi formulir, atau setelah keluar karena tidak aktif. Banyak pembaca layar memungkinkan pengguna untuk memeriksa status halaman saat ini, jadi role="status" mungkin sesuai untuk indikator pemuatan, atau mungkin fitur seperti indikator "Menyimpan... / Tersimpan".
Google Docs.

Untuk bilah pemuatan, peran bilah kemajuan mungkin lebih sesuai. Peran ini sendiri tidak cukup, dan harus dikombinasikan dengan atribut yang lebih deskriptif aria-valuemin, aria-valuemax, dan aria-valuenow. aria-valuetext juga tersedia untuk memberikan informasi yang lebih terperinci tentang apa yang terjadi di setiap tahap dalam proses multi-langkah. Peran log digunakan untuk streaming konten, seperti ruang obrolan (atau bahkan log di alat pengembang), tempat konten yang ditambahkan dibacakan pada waktu yang tepat. Peran terakhir yang perlu dibahas adalah pengatur waktu.

Sebagian besar pembaca layar tidak akan membaca konten yang diberi anotasi dengan peran ini saat berubah kecuali diminta secara eksplisit, karena diasumsikan akan berubah cukup sering dan, seperti namanya, ini paling sesuai untuk hal-hal seperti menampilkan waktu saat ini, pengatur waktu, atau hitungan mundur. aria-live memiliki tiga nilai yang mungkin: off, polite, dan assertive. off menunjukkan bahwa wilayah tersebut bukan wilayah aktif, sedangkan polite berarti pembaca layar akan mengumumkan pembaruan saat jeda berikutnya. assertive akan menyela apa yang sedang dibacakan kepada pengguna, jadi harus digunakan dengan hemat.

aria-live dapat dikombinasikan dengan atribut aria-atomic dan aria-relevant untuk mengontrol apakah seluruh elemen diucapkan kembali saat berubah, atau bagian mana yang paling relevan. Semua ini harus dikombinasikan dengan aria-describedby dan peran lain yang dijelaskan sebelumnya untuk memberikan tingkat cakupan yang memadai untuk elemen dinamis ini jika pembaca layar gagal memahaminya sendiri. Beberapa elemen HTML memiliki peran ARIA implisit yang ditetapkan padanya misalnya, elemen <progress> akan berperilaku seolah-olah memiliki role="progressbar" yang ditetapkan tanpa tindakan lebih lanjut, serta atribut aria lainnya yang dapat diturunkan dari atribut HTML biasa.

Mekanisme interaksi umum untuk berpindah melalui halaman web adalah dengan menggunakan tombol tab untuk mengakses elemen seperti kolom formulir dan tombol. Ada kemungkinan besar Anda menggunakan ini saat mengisi kolom formulir sendiri. Ketika elemen telah diberi tab, dikatakan bahwa elemen tersebut memiliki fokus. Hal ini dapat dianggap analog dengan mengarahkan kursor ke elemen dengan mouse, dan memungkinkan alat AT untuk menunjukkan di mana pengguna berada di halaman. Jika Anda menggunakan komponen HTML standar, maka sebagian besar komponen interaktif dapat diberi tab dan berinteraksi secara langsung tanpa pekerjaan lebih lanjut.

Namun, jika Anda menggunakan elemen seperti div, maka Anda dapat menggunakan atribut tabindex untuk menunjukkan bahwa elemen tersebut dapat dipilih dengan tombol tab.

Atribut tabindex diberi nilai numerik untuk menunjukkan di mana elemen tersebut akan muncul saat dokumen sedang diberi tab. Bahasa Indonesia: Anda harus selalu menyetel ini ke 0 untuk membuat elemen dapat diberi tab, yang artinya urutannya harus seperti yang akan muncul secara alami dalam dokumen (sebaliknya, Anda dapat menyembunyikan elemen yang biasanya dapat diakses dengan menyetel tabindex ke -1).

Angka lain dimungkinkan, tetapi ini dapat menimbulkan bug dan dependensi halus dengan bidang lain (misalnya, jika Anda menambahkan elemen di bagian atas halaman, Anda harus meningkatkan tabindex dari semua yang muncul setelahnya, karena harus berurutan). Ini dianggap sebagai anti-pola jika tabindex eksplisit (yaitu, bukan 0) diperlukan, karena ini sering kali berarti bahwa struktur halaman Anda tidak cocok dengan aliran visual, dan alat AT dapat menjadi sangat membingungkan jika demikian halnya. Harus ada hubungan yang kuat antara struktur HTML Anda dan aliran visual halaman dan informasinya.

Anda juga harus memastikan bahwa Anda mendengarkan peristiwa yang sesuai dalam JavaScript. Yang terpenting adalah peristiwa fokus dan kabur, yang harus Anda gunakan jika sewaktu-waktu Anda menanggapi peristiwa tetikus seperti onmouseover. Dengan mendengarkan peristiwa ini, Anda memastikan bahwa, bagi pengguna yang menavigasi dengan papan ketik, komponen merespons dengan cara yang sama seperti mereka yang menavigasi dengan tetikus. Jenis fokus papan ketik ini juga memiliki pseudo-selector yang setara dalam CSS, seperti `:hover` dan `:focus`. Tentu saja, dengan munculnya perangkat layar sentuh, mengandalkan gerakan tetikus dan peristiwa serupa semakin menurun popularitasnya. Sebaliknya, mendengarkan peristiwa sentuh atau klik eksplisit sekarang menjadi mode interaksi umum, tetapi untungnya memilih item dengan papan ketik masih memicu peristiwa klik, yang dapat menyederhanakan logika.

Aspek terakhir yang perlu dipertimbangkan, terutama untuk pembaca layar, adalah memastikan bahwa tombol berfungsi dengan jelas. Tautan seperti "klik di sini" memberikan sedikit indikasi kepada pengguna tentang apa yang sebenarnya dilakukan tautan tersebut tanpa konteks visual yang melingkupinya. "Kirim" juga bukan nama yang bagus, jadi sering kali berguna untuk membuat tombol kirim formulir secara eksplisit menamai tindakan yang akan diselesaikannya (seperti "Tambah Produk"). Hal ini terutama berlaku ketika ada beberapa tombol dengan nama yang sama di halaman, mungkin untuk formulir yang berbeda, karena pembaca layar mungkin tidak memiliki gambaran lokasi di halaman, sehingga tidak jelas formulir mana yang dirujuk oleh tombol tersebut.

Ikon populer untuk digunakan di tombol, tetapi bahkan bagi pengguna yang menjelajah tanpa AT, ikon ini dapat membingungkan tanpa konteks. Akibatnya, sangat umum untuk memiliki ikon dan deskripsi tekstual dari tindakan secara berdampingan untuk semua pengguna, yang kemudian menjadikan ikon sebagai elemen dekoratif yang tidak terbaca untuk pembaca layar. Cara umum untuk menjadikannya elemen yang murni dekoratif adalah dengan menggunakan pseudo-elemen CSS dengan elemen gambar latar belakang untuk menampilkan logo, atau tag img dengan tag alt kosong untuk menunjukkan bahwa ikon tersebut murni dekoratif.

Jika Anda memutuskan ikon saja sudah cukup, maka Anda harus menyediakan

alternatif tekstual. Berhati-hatilah saat Anda melakukan ini! Banyak penelitian telah dilakukan terhadap ikon dan menemukan bahwa, kecuali beberapa ikon yang sangat umum digunakan dalam konteks yang familiar, ikon dapat membingungkan pengguna. Misalnya, simbol silang. Jika ini muncul di sudut kanan atas popup, maka ini umumnya dianggap berarti menutup popup. Namun, menggunakan ikon silang yang sama sebagai tombol dalam item daftar mungkin berarti "hapus item ini dari daftar," atau mungkin "nonaktifkan," atau "buang perubahan," sehingga bahkan sebagian kecil ikon yang dikenal secara universal hanya dapat diidentifikasi dalam konteks yang sesuai.

Memiliki Teks Hanya Untuk Pembaca Layar

Mungkin ada kasus penggunaan lain saat Anda ingin menyediakan alternatif tekstual hanya untuk pembaca layar. Misalnya, terkadang Anda ingin konten tag h1 menjadi logo atau nama situs Anda, tetapi menggunakan tag img dapat berdampak negatif pada SEO, karena sering kali tag alt akan diabaikan, sehingga tampak seperti Anda memiliki h1 yang kosong. Teknik umum untuk menghindari hal ini adalah dengan menentukan h1 Anda seperti biasa, tetapi dengan span bersarang di dalamnya:

```
<h1 class="site-title">
  <span class="site-title_inner">My Site</span>
</h1>
```

CSS kemudian digunakan untuk menyembunyikan span bagian dalam dan background-image yang digunakan pada h1 untuk menyisipkan gambar logo. Namun, cara Anda menyembunyikan teks dapat menyebabkan teks tersebut juga disembunyikan dari pembaca layar, jika tidak hati-hati. Pendekatan yang naif mungkin adalah dengan memberikan span bagian dalam display: none;, tetapi ini juga menyembunyikan elemen tersebut dari alat AT dan mesin pencari yang mendukung CSS (termasuk Google).

Pendekatan tradisional adalah dengan memberikan elemen h1 atribut CSS text-indent: -9999px; yang akan mendorong teks keluar dari viewport, tetapi tetap menampilkannya di layar, sehingga pembaca layar (dan bot) akan tetap menemukannya. Perhatikan bahwa jika span bagian dalam mengganggu Anda, Anda dapat memperoleh hasil yang sama menggunakan elemen semu CSS. Metode text-indent bukan tanpa kekurangan. Yang pertama adalah dampak pada kinerja, karena menyebabkan browser merender teks di luar halaman, sehingga meningkatkan area yang harus dirender browser.

Beberapa orang juga bercanda dengan menyebut "10.000px-pocalypse," di mana setelah ukuran layar rata-rata lebih dari 10.000px, teks tersembunyi akan mulai muncul kembali. Lebih realistisnya, jika situs Anda memiliki komponen yang bergulir secara horizontal (misalnya, tayangan slide/carousel), maka teknik ini dapat gagal. Alih-alih metode text-indent, alternatif yang lebih modern telah muncul, yang menggunakan persegi panjang kliping dan mengubah ukuran wadah untuk menyembunyikan teks dari tampilan sambil tetap mempertahankannya di lokasi saat ini di DOM, sehingga terhindar dari penurunan kinerja. Ini dapat diimplementasikan menggunakan cuplikan CSS berikut:

```
.sr-only {
   clip: rect(0, 0, 0,0);
   clip-path: inset(50%);
   height: 1px;
   overflow: hidden;
   padding: 0;
   position: absolute;
   white-space: nowrap;
   width: 1px;
}
```

Kecuali jika Anda perlu mendukung browser yang sangat lama, ini adalah metode yang lebih baik untuk digunakan, dan banyak kerangka kerja CSS akan menyediakannya sebagai kelas utilitas (sering disebut `.sr-only` untuk "screen-reader only). Jika kotak 1px dan metode indentasi teks terasa seperti peretasan bagi Anda, itu karena memang demikian, jadi penggunaan metode ini secara berlebihan dapat menjadi indikasi masalah aksesibilitas di situs Anda. Berhati-hatilah saat menggunakannya.

Pembaca layar dan alat AT lainnya menggunakan konsep fokus untuk melacak posisi terkini pengguna yang sedang menavigasi halaman. Fokus dapat dianggap sama dengan menggulir halaman web yang panjang, di mana posisi terkini dalam gulir menunjukkan di mana pengguna saat ini berada. Merupakan hal yang umum bagi elemen interaktif untuk membuat perubahan pada DOM sebagai respons terhadap peristiwa pengguna, seperti menambahkan item baru ke daftar, atau membuka/menciutkan baki. Teknik yang digunakan untuk memberi tahu pengguna tentang pesan dan konten akan berfungsi dalam kasus tersebut, tetapi kurang berhasil jika perubahan tersebut sebenarnya bukan pengumuman atau pesan.

Bagi pengguna alat ini, melakukan tindakan seperti membuka daftar atau laci tampak sama dengan mengeklik tombol yang efeknya terjadi di luar layer pengguna tidak langsung mengetahui apa yang baru saja terjadi. Untungnya, kita dapat memanipulasi fokus menggunakan JavaScript untuk memindahkan pengguna ke area halaman yang dimodifikasi, sehingga alur mereka tidak terputus. Metode `focus()` pada HTMLElement memungkinkan kita untuk mengubah fokus. Kita juga perlu mempertimbangkan kasus-kasus yang terjadi sebaliknya suatu item menghilang dari halaman. Sering kali, tidak banyak yang dapat dilakukan di sini selain pengumuman keberhasilan untuk memastikan bahwa pengguna mengetahui suatu tindakan telah berhasil, tetapi ada beberapa kasus yang tidak sesederhana itu. Ketika elemen yang saat ini menjadi fokus dihapus atau disembunyikan dari DOM, maka browser dikatakan telah kehilangan fokus, dan menyetel ulang ke awal halaman. Ini bisa menjadi pengalaman yang sangat mengagetkan bagi pembaca layar.

Ambil contoh, laci yang memiliki tombol "tutup" di dalamnya. Ketika tutup dipilih, maka laci disembunyikan dari DOM, termasuk tombol tutup yang ada di dalamnya. Ini menyebabkan fokus hilang. Teknik yang sama dengan sengaja mengatur fokus dapat digunakan di sini untuk menghindari gangguan tersebut. Misalnya, Anda mungkin ingin

mengatur fokus kembali ke tombol buka, atau sebaliknya ke elemen berikutnya dalam daftar, tergantung pada apa yang ingin Anda capai. Hal terakhir yang perlu dipertimbangkan adalah sesuatu yang dikenal sebagai jebakan keyboard. Jebakan keyboard terjadi ketika pengguna tidak dapat meninggalkan suatu elemen menggunakan keyboard dan menjadi "macet". Ini bisa menjadi berkah sekaligus kutukan.

Salah satu cara terjadinya hal ini adalah jika Anda memetakan ulang tombol tab ke tujuan lain, atau jika aplikasi Anda memiliki bug yang terus-menerus menyetel ulang fokus ke elemen yang sama. Hal ini sangat membuat frustrasi bagi pengguna, karena membuat mereka tidak dapat menjelajahi halaman sepenuhnya. Sebaliknya, jebakan keyboard dapat berguna jika Anda memiliki dialog modal, karena Anda dapat menjebak pengguna keyboard ke dalam dialog tersebut, mirip dengan cara Anda memblokir interaksi UI dengan halaman. Sebagian besar kontrol akan berperilaku dengan cara yang dapat diakses secara default, tetapi jika Anda ingin menerapkan perilaku kustom Anda sendiri, uraian di atas memberikan gambaran umum tentang cara mulai mempertimbangkan aksesibilitas.

Namun, hal ini jauh dari kata komprehensif; ada banyak panduan daring yang menyediakan pola yang harus Anda ikuti, dengan spesifikasi ARIA dianggap sebagai versi yang paling definitif dan paling dihormati.

9.3 PENGUJIAN AKSESIBILITAS

Tentu saja, kepatuhan terhadap spesifikasi ARIA tidak cukup untuk membuat halaman web dapat diakses. Anda juga harus menguji cara kerja dan performa situs web Anda dengan alat AT. Pengujian aksesibilitas serupa dengan pengujian situs web pada umumnya. Alih-alih hanya menggunakan browser untuk menavigasi situs, pembaca layar atau alat serupa juga harus digunakan, sehingga interaksi dijalankan dengan cara yang sama seperti yang dilakukan pengguna sebenarnya. Dan, karena pengembang akan selalu melakukan pemeriksaan kewarasan fitur pada mesin mereka sebelum menyerahkannya ke QA, Anda juga bertanggung jawab untuk melakukan pemeriksaan kewarasan bagi pembaca layar.

Seperti browser web normal, alat AT memiliki berbagai tingkat dukungan untuk standar, serta keanehan dan bug per alat. Ini sering kali memerlukan solusi sementara dan perbaikan untuk memberikan pengalaman yang dapat diterima pengguna. Sebagai pengembang web, Anda harus mempelajari cara menggunakan pembaca layar untuk menyelesaikan tindakan dasar guna menavigasi halaman web atau aplikasi Anda. Untungnya, ada banyak artikel pengantar tentang cara melakukannya. Jika Anda menggunakan macOS, maka ini adalah fitur bawaan sistem operasi yang dikenal sebagai VoiceOver. Menekan ⌘ + F5 akan mengaktifkan dan menonaktifkan fitur ini, dan Anda kemudian dapat menggunakan papan ketik untuk menjelajahi peramban web Anda.

Di Windows, aplikasi NVDA dan JAWS populer. Setelah Anda terbiasa dengan pembaca layar, Anda kemudian dapat melakukan beberapa pengujian kewarasan ad hoc pada aksesibilitas/kegunaan sebelum QA dapat melakukan pengujian yang lebih lengkap. Hal paling sederhana yang dapat dilakukan adalah menjalankan dan memastikan halaman Anda masuk akal saat dibacakan dengan lantang; jika Anda menggunakan papan ketik untuk menavigasi ke

suatu tindakan, jelas apa yang akan dilakukan tindakan itu; dan melakukan suatu tindakan memastikan pembaca layar mencocokkan tindakan apa pun yang akan dilihat oleh pengguna peramban biasa.

Setelah Anda yakin bahwa konten Anda dapat dibaca dan tindakan ditandai dengan benar agar berfungsi dengan benar, langkah terakhir adalah memeriksa "jebakan" saat menavigasi dengan papan ketik. Terutama jika Anda memanipulasi fokus, atau menampilkan/menyembunyikan sesuatu pada fokus keyboard, Anda dapat terjebak dalam lingkaran fokus keyboard yang membuat Anda tidak dapat keluar atau melangkah lebih jauh untuk mengakses bagian lain dokumen. Di luar pemeriksaan kewarasan ini, ada serangkaian panduan khusus untuk menguji aksesibilitas. Panduan ini sendiri tidak cukup untuk memastikan situs web dapat diakses.

Panduan Aksesibilitas Konten Web (WCAG) adalah yang paling umum, dan ada alat otomatis yang dapat memeriksa kepatuhan terhadap panduan ini. Namun, seperti pengujian kegunaan secara umum, alat ini belum berkembang ke titik di mana mereka dapat menggantikan manusia, karena mereka tidak memahami konteks aplikasi Anda. Pemeriksa ini hampir pasti akan melewatkan beberapa masalah dan salah mendiagnosis yang lain, jadi tidak dapat digunakan sebagai alternatif pemeriksaan manual. Pemeriksa standar dapat memberi tahu Anda bahwa situs web Anda memiliki HTML yang terbentuk dengan baik, tetapi tidak akan memberi tahu Anda apakah maksud desain Anda akan ditampilkan dengan benar di semua browser pengguna.

Alat aksesibilitas memiliki masalah yang sama. Yang dapat mereka lakukan adalah memeriksa apakah kaitan terprogram sudah ada, tetapi Anda harus menggunakannya bersamaan dengan pengujian menggunakan alat aksesibilitas seperti pembaca layar untuk memastikan kepatuhan penuh. Pa11y, yang dirilis oleh Nature, adalah salah satu alat yang dapat Anda jalankan untuk mendapatkan laporan tentang seberapa patuh Anda terhadap berbagai spesifikasi. Salah satu efek samping yang baik dari membangun situs web yang mudah diakses, dengan kaitan terprogram yang baik untuk pembaca layar, adalah bahwa hal itu benar-benar membuat pengujian otomatis umum situs web atau aplikasi web Anda jauh lebih mudah. Alat otomatisasi seperti Selenium dapat mengendalikan browser dengan cara yang sangat mirip dengan alat aksesibilitas browser (mereka sering kali melampaui ini juga, seperti dengan kaitan JavaScript), tetapi jika Anda memiliki komponen yang sulit dihubungkan untuk pengujian dengan Selenium, ada kemungkinan besar komponen tersebut juga tidak kompatibel dengan AT.

Menghindari Kesalahan Umum

Terkadang sangat mudah untuk secara tidak sengaja merusak aksesibilitas. Bagian terakhir dalam bab ini membahas beberapa kesalahan umum tersebut dan cara menghindarinya.

Gaya Arahkan dan Fokus

Saat menggunakan pseudo-selector CSS:hover untuk memberi gaya pada elemen saat mouse mengarahkan kursor di atasnya, Anda hampir selalu harus menggunakan pseudo-selector :focus juga, sehingga pengguna keyboard mendapatkan perilaku yang sama untuk

menunjukkan saat sesuatu disorot.

outline: 0

Sering kali, bug akan dilaporkan seperti "cincin jelek muncul di luar elemen saat diklik." Ini adalah cincin outline yang menunjukkan elemen mana yang menjadi fokus keyboard, dan outline: 0 akan menyembunyikannya, terkadang menghasilkan estetika yang lebih menyenangkan. Namun, ini dapat membuat situs sama sekali tidak dapat digunakan oleh pengguna keyboard, karena mereka tidak dapat melihat di mana situs tersebut menjadi fokus. "Cincin jelek" muncul di banyak situs web yang berbeda, sehingga pengguna sering kali terbiasa melihatnya.

Menyingkirkannya atas nama estetika sering kali tidak perlu! Jika Anda benar-benar harus menyingkirkan outline, pastikan bahwa beberapa cara lain untuk menunjukkan fokus disertakan, tetapi perlu diingat bahwa pengguna terbiasa dengan cincin outline, jadi cara lain untuk menunjukkan fokus mungkin tidak seefektif cincin outline default.

Urutan Judul

Sangat mudah terjebak dalam upaya membuat struktur HTML sesuai dengan struktur visual, tetapi judul merupakan alat bantu navigasi yang penting. Secara khusus, level judul harus digunakan secara berurutan, daripada melewati atau menggunakan kembali level untuk menyesuaikan dengan gaya visual. Langsung menggunakan h4 saat elemen judul sebelumnya adalah h2 dapat membingungkan navigasi pembaca layar. Pola anti-pola umum lainnya adalah saat kartu dimulai dengan gambar:

```
<div class="card">
  <img src="/contents.jpg" alt="Photograph of John">
  <h3>John</h3>
  <p>John is the marketing director of FooBar Inc.< /p>
</div>
```

Dari struktur judul saja, tidak jelas bahwa gambar tersebut sebenarnya termasuk dalam judul "John." Sebaliknya, `<h3>` harus menjadi hal pertama dalam div dan gambar di bawahnya. Padding-top dalam CSS kemudian dapat digunakan untuk memberi ruang bagi gambar, dengan gambar diposisikan secara absolut di bagian atas kartu untuk membuat gaya visual yang benar. Ini mungkin tampak berlawanan dengan intuisi, tetapi bagi pengguna AT yang menavigasi menggunakan level judul, ini adalah satu-satunya cara untuk menunjukkan struktur dokumen dengan benar.

Beberapa h1

Spesifikasi HTML5 sebelumnya menetapkan bahwa penggunaan beberapa elemen semantik baru, seperti <section>, menyetel ulang makna hierarki judul, tetapi pembaca layar lambat dalam mengadopsi semantik ini. Demi konsistensi, sebaiknya abaikan ini dan asumsikan hierarki global untuk struktur judul. Secara khusus, seharusnya hanya ada satu h1 per halaman, karena ini sering digunakan untuk melompat ke badan utama halaman.

Lewati Tautan

Satu masalah umum dengan desain halaman web adalah bahwa elemen pertama pada halaman dapat berupa bilah navigasi, logo, dll., yang merupakan banyak hal yang harus dinavigasi oleh pengguna alat AT untuk mendapatkan konten yang sebenarnya. Meskipun melompat ke h1 dapat menjadi cara untuk mengatasi hal ini, cara lain adalah dengan menggunakan pola yang dikenal sebagai "tautan lewati". Tautan lewati adalah tautan jangkar yang muncul di dekat bagian atas halaman dan memungkinkan pengguna untuk melompat ke konten utama halaman. Cukup sering, Anda ingin membuat jangkar ini tidak terlihat oleh pengguna non-keyboard karena alasan estetika, dan pola umum untuk mengatasi hal ini adalah hanya menggunakan pseudo-selector: focus di CSS untuk menyembunyikannya dari layar hingga mendapat fokus keyboard.

Tombol vs. Jangkar

Saat membuat elemen dinamis, mungkin tergoda untuk menggunakan `tag <a>` dan menambahkan pengendali peristiwa klik ke dalamnya dalam JavaScript. Hal ini dapat membingungkan bagi pengguna alat AT, karena semantik `tag <a>` yang biasa membawa Anda ke tempat lain (bagian lain dari dokumen yang sama, atau dokumen lain sama sekali), berbeda dengan `<button>`, yang digunakan untuk memicu tindakan. Saat menerapkan tindakan, penggunaan elemen HTML yang tepat dapat membantu pembaca layar menafsirkan halaman serta menyarankan semantik yang benar, meskipun menata tombol mungkin tampak lebih sulit daripada menata tag a. Penggunaan `<a href="#">` adalah anti-pola yang harus dihindari, karena href menunjukkan bahwa tautan tidak mengarah ke mana pun, yang menunjukkan bahwa tautan tersebut sebenarnya adalah tindakan, dan seharusnya berupa tombol.

Penggunaan Atribut alt yang Benar

Penggunaan atribut alt pada gambar merupakan persyaratan aksesibilitas langka yang dikodekan secara kaku ke dalam spesifikasi HTML, dan yang akan diperiksa oleh validator standar. Meskipun demikian (atau karena hal ini), penggunaan atribut alt yang tidak tepat adalah hal yang umum. Atribut alt hanya boleh digunakan saat gambar menambahkan konten (bukan hanya estetika) ke halaman, dan seperti namanya, gambar tersebut harus berupa alternatif tekstual, bukan sekadar deskripsi gambar. Jika gambar hanya ada untuk menambahkan beberapa informasi gaya ke halaman, maka sering kali lebih baik untuk benar-benar menyertakannya sebagai gambar latar belakang dalam CSS, daripada sebagai `tag <img>`. Meskipun atribut alt wajib, tidak apa-apa untuk membiarkannya kosong untuk menunjukkan bahwa gambar tersebut murni untuk alasan gaya. Hal ini sangat keterlaluan saat atribut alt hanya mengulang judul, karena semua ini hanya menyebabkan teks duplikat terbaca di pembaca layar.

Font Ikon

Pengoptimalan umum untuk ikon adalah mengirimkan satu set ikon ke browser sebagai font ikon, menggunakan Unicode atau karakter lain untuk menunjukkan karakter khusus, yang kemudian diketik ke dalam dokumen. Namun, pembaca layar tidak tahu cara "membaca" karakter ini, dan dapat menyebabkan mereka membaca omong kosong saat menafsirkan

dokumen. Alih-alih menggunakan font ikon, sprite CSS atau teknik lain merupakan cara yang lebih baik untuk menyertakan ikon, baik dengan menggunakan teks tersembunyi bagi pembaca layar untuk memberikan makna tekstual pada ikon atau, lebih baik lagi, menggunakan label teks di samping gambar.

Kontras Warna

Kesalahan umum terakhir yang harus diperhatikan adalah kontras warna. Meskipun hal ini sering dianggap sebagai bagian dari desain visual, cara warna diterapkan dapat membuat perbedaan besar pada aksesibilitas halaman. Mudah untuk berasumsi bahwa ketika teks berada di atas gambar latar belakang yang sesuai, kontras warnanya bagus, tetapi jika gambar gagal, atau lambat dimuat, atribut warna latar belakang juga harus ditetapkan, di samping gambar latar belakang, yang seharusnya memberikan kontras yang cukup pada teks di atasnya.

Sering kali, ketika teks putih digunakan pada gambar latar belakang gelap, dan warna latar belakang default halaman adalah putih, maka pada awalnya teks putih akan ditampilkan di atas latar belakang putih hingga gambar dimuat, yang menghasilkan pengalaman pemuatan yang kurang optimal. Demikian pula, penggunaan warna saja sebagai indikator sesuatu tidak dapat diandalkan, karena buta warna cukup umum terjadi. Ketika warna digunakan, maka cara lain untuk menunjukkan arti warna juga diperlukan (misalnya, teks atau ikon).

9.4 KESIMPULAN

Sebagai seorang profesional, Anda memiliki kewajiban untuk membangun produk Anda dengan cara yang tidak mendiskriminasi mereka yang menggunakan teknologi bantuan. Untungnya, Web dan browser telah dibangun dengan cara yang mengurangi tingkat gesekan yang diperlukan untuk mendukung alat aksesibilitas di luar apa yang dapat ditawarkan oleh aplikasi asli. Membengkokkan standar web, atau menerapkan komponen yang sepenuhnya khusus dari elemen HTML generik, dapat menyebabkan rusaknya kompatibilitas dengan teknologi bantuan, yang berpotensi mengasingkan sebagian besar basis pengguna Anda.

Ada banyak jenis aksesibilitas yang perlu dipertimbangkan, beberapa di antaranya menggunakan alat seperti pembaca layar untuk memberikan pengalaman, dan yang lainnya hanya diterapkan di browser dan desain. Penting untuk memperlakukan alat AT ini sama seperti browser web lainnya, dan memperbaiki bug yang ditemukan di dalamnya, serta menguji situs Anda menggunakan mekanisme interaksi di luar mouse atau layar sentuh standar. Seperti banyak persyaratan lintas fungsi, aksesibilitas harus dibangun sejak awal proyek, dan ada beberapa teknik untuk mengatasinya, baik untuk menyusun dokumen menggunakan HTML semantik maupun menyiapkan pengendali peristiwa JavaScript untuk jenis peristiwa tertentu dan mempertimbangkan alur lain untuk berinteraksi dengan elemen interaktif.

BAB 10
APPLICATION PROGRAMMING INTERFACE (API)

10.1 PERBANDINGAN API SOAP DAN REST DALAM PENGEMBANGAN WEB

Setiap aplikasi memiliki API dalam bentuk tertentu, baik API eksternal yang dirancang untuk digunakan oleh aplikasi selain milik Anda, atau API internal yang memungkinkan modul dalam aplikasi Anda saling berkomunikasi. Mendesain API jenis pertama jauh lebih sulit daripada membangun API jenis kedua, karena akan sangat sulit untuk membuat perubahan yang tidak merusak aplikasi lain tersebut. Pada API jenis kedua, kode yang mendefinisikan API dan kode yang menggunakannya sering kali berpindah bersama, sehingga pemfaktoran ulang menjadi jauh lebih sederhana.

Pada masa awal komputasi, API eksternal adalah pustaka bersama yang dapat Anda gunakan untuk menautkan aplikasi Anda, mengakses fungsi dan kelas dalam pustaka tersebut menggunakan panggilan fungsi normal. Dalam pengembangan web, lingkungan tempat Anda menjalankan akan menyediakan serangkaian API untuk Anda gunakan, serta pustaka atau kerangka kerja apa pun yang mungkin Anda instal, seperti dari NPM. Namun, jenis API yang akan kita bahas di sini adalah yang tersedia melalui jaringan, dan sering kali berada di mesin yang berbeda dari mesin tempat kode Anda berjalan. Pada masa-masa awal Web, mekanisme yang dikenal sebagai RPC (remote procedure calls) sangat populer.

Mekanisme ini merupakan URL tunggal yang dapat Anda tekan dengan parameter, tempat fungsi akan dijalankan dan respons diberikan kepada Anda. Namun, hal ini tidak sesuai dengan struktur Web karena memiliki dokumen dan sumber daya di URL yang berbeda, dan RPC di satu URL menjadi sulit ditangani karena melibatkan cache dan penskalaan browser. Simple Object Access Protocol (SOAP) adalah contoh paling umum dari hal ini di dunia web. SOAP tumbuh dari mekanisme ad-hoc lain untuk berkomunikasi melalui Web, menggunakan pola yang dikenal sebagai AJAX dan XML-RPC, tetapi dirancang untuk bekerja melalui protokol selain HTTP (seperti antrean pesan atau email).

SOAP diimplementasikan dengan memiliki skema yang menentukan jenis pesan yang tersedia, dan struktur permintaan dan respons sebagai XML, lalu satu URL tempat HTTP POST dikirim. Isi POST menentukan permintaan (mungkin untuk mendapatkan data, atau melakukan tindakan lain) lalu server merespons dalam format yang diharapkan. Kekuatan SOAP terletak pada skemanya. Skema ini dapat menyederhanakan interaksi dengan API, serta mendokumentasikan sendiri tentang jenis panggilan prosedur yang tersedia, tetapi ini juga merupakan titik lemah utamanya. Alat dan pustaka khusus diperlukan untuk mendukung skema ini dan menggunakan titik akhir secara efektif, yang mempersulit debugging dan menambahkan overhead pengembangan lainnya.

Sebagian besar kerumitan yang SOAP coba selesaikan adalah kerumitan yang diperkenalkan oleh protokol itu sendiri. Meskipun sudah digunakan sebelumnya, Roy Fielding mengkodifikasikan struktur alternatif untuk membangun API, yang disebutnya Representational State Transfer (REST). 1 API jenis ini menangani sumber daya pada layanan,

menggunakan kembali mekanisme HTTP yang ada yang menangani sumber daya ini mirip dengan cara peramban web memperlakukan dokumen. Ini menyederhanakan implementasi dan cara berinteraksi dengan API dengan membuat alat web yang ada, seperti lapisan keamanan dan penyimpanan sementara, bekerja secara alami.

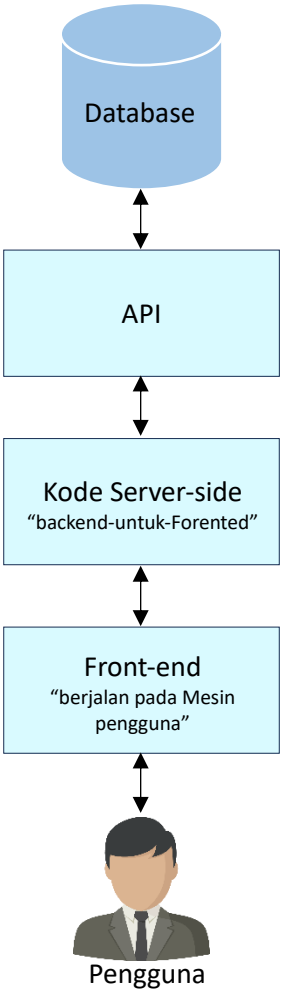
Pendekatan ini sudah digunakan secara ad-hoc sebelum Fielding menerbitkan disertasinya, tetapi uraiannya telah menetapkan praktik standar terbaik untuk diikuti. Pada intinya, REST mungkin tampak tentang penggunaan metode HTTP yang tepat (kadang-kadang disebut kata kerja) dan struktur URL yang tepat, tetapi bagian terpentingnya adalah batasan desain yang mendasari API ini: pemisahan perhatian antara klien dan server, statelessness, cachability, sistem berlapis, dan antarmuka yang seragam. REST dapat digunakan untuk menggambarkan sistem apa pun yang memiliki batasan ini, tetapi "antarmuka seragam" biasanya diartikan sebagai HTTP, sehingga menjadi serangkaian antarmuka web.

Perbedaan terbesar antara SOAP dan REST adalah dalam hal kemudahan penggunaannya. SOAP biasanya memerlukan pustaka khusus untuk membantu menangani pembuatan permintaan dalam bentuk yang tepat dan menangani respons, serta hal-hal seperti autentikasi, tetapi REST dapat menggunakan kembali mekanisme standar untuk membuat permintaan HTTP. Di peramban web, teknik ini dikenal sebagai AJAX, yang menggunakan API yang disebut XMLHttpRequest (XHR) untuk membuat permintaan standar ke server (di ES6, API baru yang disebut Fetch menggantikan XHR, dan menyederhanakan cara permintaan dibuat).

Tanggung Jawab API

Kesalahan umum bagi orang yang membuat API adalah mencoba membuat satu API untuk memenuhi setiap situasi saat API tersebut dipanggil. Misalnya, API produk mungkin dirancang agar dapat diakses dari aplikasi JavaScript yang berjalan di browser dan dari proses impor batch otomatis. Hal ini dapat menyebabkan masalah saat API akhirnya harus mengimplementasikan fungsionalitas yang tidak perlu, atau banyak fitur khusus browser. Komunikasi server ke server biasanya menggunakan kunci API, tetapi pengguna aplikasi Anda harus memiliki beberapa login umum. Jika API Anda menangani autentikasi server-ke-server dan autentikasi pengguna yang umum ini, hal itu dapat menyebabkan kerumitan, dan kerumitan tersebut menyebabkan bug.

Sebaliknya, yang dapat kita lakukan adalah memperkenalkan pelapisan pada API. Salah satu rangkaian lapisan tersebut adalah API backend yang kaya, dan API yang memungkinkan UI front-end berkomunikasi kembali ke server. Lapisan ini adalah konsep yang disebut "backend untuk frontend." Dalam hal ini, API produk kami dikunci, sehingga hanya tersedia untuk komunikasi antar-server (ini dapat dilakukan dengan meminta kunci API atau rahasia lain yang tidak terekspos ke pengguna, atau dengan menerapkan firewall di tingkat jaringan). Kami kemudian memiliki lapisan API "backend-for-frontend", yang sering kali merupakan aplikasi yang sama yang menyajikan HTML asli untuk halaman tersebut. API ini hanya memaparkan serangkaian titik akhir AJAX yang dapat digunakan front end untuk meminta back end. Gambar 10.1 menunjukkan hal ini.



Gambar 10.1 Lapisan Pola "Backends-For-Frontends"

Lapisan ini sering bertindak sebagai lapisan penerjemahan, karena dapat menyederhanakan API yang perlu berinteraksi dengan front end Anda dan memetakannya ke konsep API back-end Anda (yang bisa lebih abstrak atau kompleks daripada yang perlu diperhatikan oleh front end). Ini juga menjadi tempat yang sangat baik untuk menempatkan pemeriksaan validasi dan autentikasi yang kuat, yang berarti API back-end Anda tidak perlu memiliki aturan validasi atau struktur otorisasi yang rumit, yang menyederhanakannya dan mengurangi kemungkinan bug.

API back-end kemudian dapat dikunci dengan ketat sehingga hanya komponen lain yang langsung berada di bawah kendali Anda yang dapat mengaksesnya secara langsung, dan dapat mempercayai bahwa tepi sistem menyediakan keamanan tambahan untuk informasi dan interaksi yang terjadi dengannya. Pola backend-untuk-frontend sangat kuat ketika beberapa front end atau sistem ingin berkomunikasi dengan API back-end yang sama.

10.2 MENDESAIN REST API

Mendesain API mirip dengan mendesain antarmuka pengguna grafis, yaitu Anda perlu memikirkan kebutuhan pengguna akhir (mungkin Anda sendiri, pengembang lain di tim Anda, atau tim eksternal yang menggunakan API Anda), tindakan yang mungkin mereka ambil, dan seperti apa struktur dan konsistensi data Anda. Namun, API yang dirancang dengan sempurna

tidak selalu dapat dicapai, karena dapat dibatasi oleh kinerja basis data yang mendasarinya. Bagian pertama API yang perlu dipertimbangkan saat mendesainnya adalah struktur URL. Secara teknis, REST mencakup konsep Hypermedia As the Engine of Application State (HATEOAS), yang menetapkan bahwa API harus memiliki titik masuk tetap yang sederhana, dan Anda harus menavigasi tautan yang dijelaskan dari titik masuk tersebut untuk menemukan sumber daya dan tindakan yang ingin Anda lakukan. Misalnya, respons dari API yang dipaginasi mungkin terlihat seperti ini:

```
{
  "items": [ ... ],
  "links": {
    "self": "https://api.example.com/blogposts/2",
    "first": " https://api.example.com/blogposts/1",
    "prev": " https://api.example.com/blogposts/1",
    "next": " https://api.example.com/blogposts/3"
  }
}
```

Hal ini sering kali tidak diterapkan untuk keseluruhan API demi kinerja. Jika klien harus menavigasi API untuk menemukan tindakan yang ingin dilakukan, hal ini dapat menyebabkan banyak permintaan dan memperlambat segalanya. Banyak API yang masih menerapkan fitur ini, tetapi fitur ini lebih membantu pengembang layanan, daripada digunakan oleh server yang memanfaatkan API tersebut. Sebaliknya, pengetahuan tentang bagaimana URL disusun digunakan untuk berinteraksi langsung dengan sumber daya yang diinginkan. Namun, pengetahuan tentang bagaimana URL disusun dapat menjadi sangat mudah untuk diandalkan, padahal dalam banyak situasi, mengandalkan HATEOAS sebenarnya dapat menyederhanakan kode Anda.

Misalnya, dalam katalog produk, Anda mungkin ingin menampilkan beberapa produk terkait di samping item yang telah Anda pilih. Hal ini dapat mengakibatkan beberapa panggilan API, dan satu pendekatan mungkin adalah menyertakan ID item katalog lain untuk dicari, dan kemudian front end perlu mengetahui cara mencari ID tersebut untuk mengambil metadata. Cara yang lebih sesuai dengan HATEOAS adalah dengan menyertakan URI lengkap, untuk menghindari keharusan mengetahui cara mencari ID. Anda harus berhati-hati saat bekerja dengan API eksternal untuk tidak mengikuti URL secara membabi buta, karena jika API tersebut tidak tepercaya, hal itu dapat membawa Anda ke tempat yang mungkin tidak ingin Anda muat.

Saat membangun API, Anda mungkin tergoda untuk menerapkan semua praktik terbaik HTTP, dan khususnya negosiasi konten, untuk mendukung berbagai jenis klien. Namun, kecuali jika Anda memiliki kebutuhan khusus untuk ini, hal itu dapat memperumit aplikasi Anda tanpa memberikan banyak manfaat. Misalnya, memberikan satu produk di API produk Anda URL seperti “https://api.example.com/products/533-4499” dan kemudian URL yang ditujukan kepada pengguna menjadi “https://www.example.com/products/533-4499” dianggap bukan praktik terbaik, karena Anda memiliki dua URL berbeda yang merujuk ke hal yang sama.

Namun, ini adalah kejadian umum dan dapat sangat membantu dalam hal penerapan

keamanan dan caching (Anda dapat mengunci seluruh nama domain api.example.com sambil tetap menjaga www.example.com tetap terlihat). Lebih jauh lagi, banyak API memudahkan pemberian representasi berbeda dari sumber daya yang sama. Dalam contoh di atas, api.example.com/products/533-4499 dapat memberikan versi JSON dari data produk Anda ketika diminta dengan header “Accept: application/json,” dan versi XML ketika diminta dengan header “Accept: application/xml.” Ini mungkin tampak menggoda, terutama jika Anda mendapatkannya secara “gratis” dengan kerangka kerja Anda, tetapi konsep di balik XML dan JSON cukup berbeda sehingga mencoba menerjemahkan secara otomatis di antara keduanya akan menghasilkan representasi yang secara idiomatis tidak satu pun dan akan sulit untuk digunakan.

Selain itu, dalam hal caching, Anda sekarang perlu menyertakan header “Vary: Accept” dalam respons Anda agar cache dapat menangani ini dengan tepat. Hal ini dapat berarti bahwa server cache dapat memiliki salinan cache yang berbeda dari URL yang sama tergantung pada header Terima yang dikirim oleh klien, yang meminimalkan manfaat yang diberikan caching kepada Anda. Prinsip KISS (“keep it simple, stupid”) bekerja dengan sangat baik di sini, meskipun mungkin tergoda untuk mengikuti praktik terbaik teoritis HTTP dan REST. Cukup dengan selalu mengembalikan JSON (atau XML) dan menyajikan API Anda dari serangkaian URL yang berbeda ke front-end Anda dapat sangat menyederhanakan implementasi Anda.

Praktik terbaik HTTP juga menyarankan penggunaan tipe MIME “vendor” untuk respons JSON Anda agar sesuai dengan skema, tetapi hal itu berpotensi mempersulit klien untuk menggunakannya (mereka perlu mengetahui tipe MIME tertentu sekarang dapat diurai sebagai JSON, yang dapat disimpulkan banyak orang secara otomatis jika header “Content-Type” adalah application/json).Hal ini mungkin tampak melanggar batasan REST bahwa pesan bersifat mendeskripsikan diri sendiri, tetapi banyak orang menganggapnya sebagai versi yang lebih lemah daripada yang awalnya dimaksudkan untuk kemudahan implementasi (misalnya, Anda masih dapat mengurai sintaksis pesan sebagai JSON, tetapi pesan tersebut tidak serta-merta mendefinisikan sendiri semantik tentang bagaimana respons tersebut harus ditangani).

Hal terpenting yang perlu dipertimbangkan tentang API RESTful adalah bahwa API tersebut tidak boleh menyimpan status apa pun tentang klien yang menggunakannya. Permintaan tunggal dari klien harus dianggap terpisah dari permintaan lainnya. API Anda tidak boleh mengharuskan pengguna membuat beberapa permintaan untuk mengubah sumber daya sebelum perubahan tersebut benar-benar terjadi. Hal ini memberi Anda kebebasan untuk menskalakan implementasi API Anda dengan mengizinkannya berjalan di banyak server berbeda, dengan hanya berbagi basis data yang mendasarinya. Keterbatasan implementasi ini memungkinkan kita memanfaatkan properti yang disebut idempotensi untuk titik akhir kita.

Idempotensi berarti bahwa membuat permintaan yang sama harus selalu menghasilkan hasil yang sama. HTTP mendefinisikan beberapa "metode" yang dapat dipanggil pada URL tertentu. Metode-metode tersebut adalah GET, HEAD, POST, PUT, DELETE, PATCH, OPTIONS, dan CONNECT. Dari ini, CONNECT hanya digunakan oleh server proxy HTTP untuk memulai koneksi TLS ke server jarak jauh, jadi umumnya dapat diabaikan, dan HEAD dan OPTIONS sering diimplementasikan oleh server web atau kerangka kerja untuk Anda, kecuali

jika Anda melakukan implementasi tingkat sangat rendah.

HEAD didefinisikan sebagai melakukan permintaan GET tanpa benar-benar mengembalikan badan, hanya tajuk respons. OPTIONS serupa, tetapi untuk permintaan yang mengubah status: POST, PUT, PATCH, dan DELETE). OPTIONS tidak benar-benar mengubah status, tetapi akan memberi tahu Anda metode apa yang dapat digunakan pada suatu sumber daya. Ini juga digunakan oleh browser dalam teknik yang dikenal sebagai CORS preflight, di mana tajuk yang dikembalikan diperiksa untuk kebijakan lintas asal guna menentukan apakah permintaan yang sebenarnya harus diizinkan atau tidak. Permintaan ke server web terdiri dari beberapa header, dan secara opsional sebuah body.

Header menentukan setidaknya URL dan metode, tetapi juga beberapa pasangan nilai kunci yang dapat menambahkan informasi tambahan ke permintaan. Beberapa kolom umum termasuk User-Agent, yang menunjukkan jenis program yang digunakan untuk membuat permintaan, dan Host, yang menunjukkan nama host situs web yang diminta (untuk memungkinkan satu server melayani beberapa nama host yang berbeda). Yang lain digunakan untuk menunjukkan hal-hal tentang respons, seperti Accept-Language, yang menentukan bahasa (manusia) mana yang harus dikembalikan ke halaman (misalnya, untuk mendukung internasionalisasi), atau Referer, yang menunjukkan halaman mana yang ditautkan ke halaman ini.

Tidak, saya tidak membuat kesalahan ejaan; spesifikasi HTTP asli salah mengeja "referer" sebagai "referer" (tampaknya, tidak ada kata yang ada dalam kamus pemeriksa ejaan UNIX yang digunakan editor). Kesalahan ketik "referer" sekarang menjadi ejaan yang diterima saat membahas HTTP, karena standar tersebut tidak pernah memperbaikinya dan telah diterapkan dengan cara itu oleh server dan browser. Ada banyak hal yang perlu dibicarakan tentang metode dan penggunaan metode HTTP yang benar oleh API. GET pada dasarnya adalah permintaan untuk membaca beberapa konten.

Permintaan GET dapat di-cache, jadi tidak boleh digunakan untuk mengubah konten atau melakukan tindakan (karena Anda tidak yakin apakah permintaan tersebut benar-benar mencapai server atau hanya dilayani oleh cache). Akibatnya, kita dapat menganggap permintaan GET sebagai permintaan yang idempoten: membuat permintaan yang sama harus selalu mengembalikan informasi yang sama (kecuali, misalnya, status sumber daya yang mendasarinya telah berubah melalui beberapa cara lain). PUT digunakan untuk mengganti konten URL dengan beberapa konten baru. PUT selalu mengganti deskripsi sumber daya tersebut secara keseluruhan, atau dapat membuat sumber daya baru di mana sesuatu tidak ada.

Misalnya, jika Anda memiliki katalog produk dan ingin memperbarui kolom di dalamnya, Anda dapat melakukan GET pada URL tersebut, mengubah kolom yang ingin diubah, lalu PUT seluruh deskripsi kembali ke URL tersebut. PUT terkadang juga digunakan untuk membuat seluruh sumber daya dari awal, dengan melakukan PUT ke URL yang sebelumnya tidak ada untuk membuatnya. Namun, cara yang lebih umum untuk membuat sesuatu adalah dengan membuat permintaan POST ke titik akhir yang didedikasikan untuk membuat sumber daya baru dan yang akan membuat URL dan ID baru untuk Anda. Salah satu masalah dengan

PUT adalah jika Anda ingin memperbarui sumber daya, pertama-tama Anda harus GET, mengubah kolom, lalu PUT kembali semuanya. Pasti akan ada celah, betapapun kecilnya, antara GET dan PUT.

Jika pengguna atau aplikasi lain membuat perubahan yang tidak terkait pada sumber daya yang sama pada saat itu, maka Anda akan mengganti perubahan Anda pada PUT Anda. Salah satu metode untuk menghindari hal ini disebut PATCH (dibahas nanti), tetapi mekanisme juga ada di samping PUT untuk menangani kasus ini. Pada GET, biasanya ditambahkan header ke respons yang dikenal sebagai "ETag," atau tag entitas. Tag ini digunakan untuk mengidentifikasi versi sumber daya tersebut secara unik (mungkin berupa nomor versi aktual, atau hash respons). Saat klien menerima respons, klien harus mencatat ETag tersebut, lalu saat permintaan PUT dibuat, header If-Match dikirim dengan nilai ETag. Header If-Match menunjukkan bahwa tindakan hanya boleh dilakukan jika nilai header tersebut cocok dengan ETag sumber daya, jadi jika sumber daya telah berubah sementara itu, kesalahan akan diterima dan Anda dapat mencoba lagi. Seperti halnya header permintaan, ada banyak jenis header respons lainnya.

Misalnya, alih-alih ETag dan If-Match, Anda dapat menggunakan Last-Modified dan If-Not-Modified-Since. Yang lain digunakan untuk alasan lain guna mengekspresikan metadata, tetapi terlalu banyak untuk dibahas di sini. DELETE berfungsi seperti yang Anda harapkan. Jika Anda membuat permintaan HTTP dengan metode DELETE pada URL, maka sumber daya tersebut akan dihapus dari basis data. Tidak seperti GET, PUT, dan DELETE, PATCH dan POST tidak idempoten, dan mengulang permintaan dapat menimbulkan efek samping yang tidak diharapkan. POST digunakan untuk memicu tindakan pada server.

Seperti yang disebutkan dalam bagian tentang PUT, terkadang tindakan ini adalah untuk membuat sesuatu yang baru dalam basis data Anda, tetapi POST dimulai sebagai istilah yang lebih umum untuk menggambarkan pemrosesan pengiriman formulir HTML. POST tidak idempoten, karena mengirimkan permintaan yang sama beberapa kali akan menyebabkan tindakan tersebut dipicu sebanyak jumlah yang sama. Misalnya, bayangkan Anda memiliki titik akhir untuk menambahkan sesuatu ke katalog. Jika Anda POST dua kali, maka dua item akan dibuat, tetapi jika PUT digunakan sebagai gantinya, yang kedua hanya akan mengganti item yang baru dibuat dengan deskripsi yang identik.

POST berguna untuk memicu item seperti tindakan latar belakang (misalnya, transkode video) atau tindakan lain yang memiliki efek samping, seperti mengirim email. PATCH sering digunakan dalam kolaborasi dengan PUT. Dengan PUT, Anda harus mengirim seluruh deskripsi sumber daya karena PUT akan menggantikannya secara keseluruhan, tetapi PATCH memungkinkan Anda memperbarui satu bidang atau sebagian dari sumber daya. Seperti PUT, ada kemungkinan kondisi persaingan dengan PATCH jika dua perbedaan yang saling bertentangan diterapkan dalam waktu yang hampir bersamaan, tetapi dampaknya dapat diminimalkan, atau dihindari sepenuhnya.

Isi PATCH yang tepat untuk menjelaskan perubahan juga bervariasi di antara penerapan PATCH. Setelah Anda memilih kata kerja HTTP yang tepat untuk setiap jenis permintaan yang Anda buat ke URL yang telah Anda tetapkan di API, Anda dapat mulai berpikir tentang

bagaimana Anda harus menyusun respons Anda kepada orang yang membuat permintaan tersebut. Respons HTTP, seperti permintaan, dibagi menjadi dua bagian: header dan body. Header respons serupa dengan permintaan karena header tersebut menjelaskan metadata tambahan tentang respons dan server yang mengirimkannya yang mungkin tidak ada di body (seperti ETag, seperti yang dibahas sebelumnya, atau kebijakan lintas asal sumber daya tersebut untuk respons OPTIONS).

Perbedaan struktur respons adalah bahwa alih-alih metode HTTP, terdapat kode status yang menunjukkan jenis responsnya. Kode status dibagi menjadi lima kategori utama, dan dalam setiap kategori terdapat banyak variasi. Hanya yang paling umum yang akan dibahas di sini. Kode status adalah angka tiga digit, dan digit pertama menunjukkan kategori. Kode yang dimulai dengan 1 digolongkan sebagai informasional, 2 sebagai sukses, 3 sebagai pengalihan, 4 sebagai kesalahan klien, dan 5 sebagai kesalahan server. Anda akan jarang menemukan kode 1xx selama pengembangan, kecuali "101 Switching Protocols," yang mungkin Anda lihat di konsol dev jika Anda menggunakan WebSockets (WebSockets secara teknis merupakan protokol yang berbeda dari HTTP, tetapi dimulai sebagai HTTP lalu beralih ke protokol WebSockets untuk memungkinkan komunikasi dua arah).

"200 OK" secara luas dianggap sebagai respons normal untuk Web, dan sebagian besar kerangka kerja akan mengirimkan respons 200 secara default, kecuali Anda secara eksplisit menggantinya. 200 menunjukkan bahwa permintaan apa pun yang dibuat adalah permintaan yang valid, dan responsnya adalah apa yang Anda harapkan. Kode 2xx lain yang menarik adalah "201 Created," yang dapat Anda kirim bersama header Location setelah permintaan untuk membuat sumber daya baru guna menunjukkan tempat sumber daya tersebut dibuat.

Misalnya, jika API Anda untuk mendaftarkan artikel baru di CMS adalah POST ke /articles/, maka Anda mungkin ingin merespons dengan "201 Created" dan header Location dengan URL artikel yang baru dibuat (misalnya, menunjukkan ID yang telah dibuat secara otomatis untuknya). Kode status 3xx digunakan untuk pengalihan, tetapi kesalahan yang dibuat oleh browser awal menyebabkan 301 dan 302 diimplementasikan secara tidak benar, sehingga tampaknya ada banyak kode yang serupa.

Respons 3xx biasanya digabungkan dengan header Location, yang berisi URL baru yang harus dituju klien. Setelah browser mengimplementasikan 301 dan 302 dengan cara yang berbeda dari spesifikasi, server kemudian mencocokkan apa yang diinginkan browser daripada apa yang ditunjukkan oleh standar. Untuk memungkinkan semantik asli, dan untuk menghindari kebingungan, kode baru dibuat untuk memaksa browser berperilaku dengan benar, dan kode lama didefinisikan ulang agar sesuai dengan semantik browser, meskipun penggunaan kode baru direkomendasikan.

301 dan 308 keduanya menunjukkan perubahan permanen pada URL. Untuk permintaan GET, keduanya berperilaku sama, tetapi untuk jenis permintaan lainnya, sebagian besar browser akan GET URL target untuk permintaan 301, meskipun itu adalah POST atau yang serupa, tetapi untuk permintaan 308 metode asli harus tetap sama. 302 awalnya digunakan untuk menunjukkan perubahan sementara, tetapi penggunaannya bahkan kurang jelas, dengan beberapa browser mengubah POST menjadi GET (seperti 301), dan yang lainnya

tetap sama.

303 dibuat untuk secara eksplisit mengatakan bahwa POST harus berubah menjadi GET, dan 307 untuk mengatakan bahwa metode permintaan harus tetap sama. Perbedaan utama antara pengalihan permanen dan sementara adalah bahwa pengalihan permanen di cache, sedangkan pengalihan sementara tidak. Pengalihan ini juga berdampak pada SEO. Jika Anda merestrukturisasi situs dan menghapus atau menggabungkan beberapa halaman Bersama sama, atau hanya mengubah URL, maka Anda harus menggunakan pengalihan 301 permanen. Pengalihan sementara berguna jika pengalihan tersebut harus diubah.

Misalnya, beberapa situs web konferensi dapat mengalihkan beranda mereka ke tahun untuk acara terkini (misalnya, https:// www.example.com/conference/ ke https://www.example.com/conference/2019/), tetapi kemudian memperbarui pengalihan tersebut pada tahun berikutnya. Dalam kasus ini, pengalihan 302 (atau 307) dapat digunakan, dan Anda dapat memberikan URL yang sama kepada orang-orang dari tahun ke tahun, tetapi tidak merusak bookmark yang dimiliki orang-orang ke tahun-tahun sebelumnya saat Anda meluncurkan acara baru.

303 sebagian besar berguna untuk menangani pengiriman formulir. Jika pengguna mengirimkan formulir ke situs, Anda dapat memprosesnya dan kemudian mengalihkan mereka ke halaman hasil menggunakan 303. Kode 4xx digunakan untuk pelaporan saat pengguna API atau browser telah membuat semacam kesalahan. Kode 4xx yang paling terkenal adalah "404 Not Found," yang menunjukkan bahwa URL telah salah ketik atau sumber daya yang tidak ada diminta.

Secara teknis, jika halaman tersebut pernah ada di sana, tetapi telah dihapus, kode lain, "410 Gone," harus digunakan, tetapi banyak orang tidak menerapkan fungsi ini. Kode 4xx populer lainnya adalah "401 Unauthorized" dan "403 Forbidden," dengan kode pertama menunjukkan bahwa pengguna perlu mengirim beberapa kredensial yang valid dengan permintaan tersebut, dan kode kedua menunjukkan bahwa kredensial Anda valid, tetapi Anda tidak memiliki izin untuk mengakses URL tersebut.

Misalnya, jika Anda memiliki API tempat beberapa pengguna bersifat administratif tetapi yang lainnya tidak memiliki hak istimewa, maka untuk titik akhir administratif, Anda akan mengirim 401 jika tidak ada (atau tidak valid) kredensial yang dikirim, dan 403 jika kredensial tersebut valid, tetapi kredensial tersebut untuk pengguna non administratif. Kode 4xx terakhir yang akan saya bahas adalah "400 Bad Request." Kode ini menunjukkan bahwa ada yang salah dengan permintaan tersebut, dan dapat sangat berguna untuk API.

Misalnya, jika badan permintaan POST atau PUT gagal diurai sebagai JSON, atau gagal dalam pemeriksaan validasi lain (mungkin ada bidang yang hilang), maka 400 akan menjadi kode status yang tepat untuk dikembalikan. Titik akhir 4xx harus mengembalikan pesan kesalahan kepada pengguna. Untuk server non API, ini sering kali berupa halaman HTML, tetapi untuk API, akan berguna untuk mengembalikan pesan JSON (atau yang serupa) yang dapat diurai oleh pengguna API.

Misalnya, dalam respons 400, mengirimkan kembali daftar terstruktur dari setiap kesalahan validasi yang terjadi dapat berguna bagi klien API untuk membangun pengalaman

pengguna pesan kesalahan yang baik bagi klien. RFC 7807 mencoba untuk menentukan cara standar untuk membangun pesan kesalahan ini, tetapi banyak API juga menerapkan struktur JSON mereka sendiri untuk pesan kesalahan.

HTTP 418

Jika melihat daftar kode status HTTP, Anda akan melihat satu kode yang menonjol: kode 418. Internet Engineering Task Force, yang mendefinisikan banyak standar Internet, memiliki tradisi panjang standar "lelucon" pada April Mop, dan pada tahun 1998, standar lelucon yang disebut "Hypertext Coffee Pot Control Protocol" diperkenalkan, termasuk kode kesalahan 418 "Saya teko," yang menunjukkan bahwa Anda telah meminta teko untuk menyeduh kopi. Sebagai telur Paskah, banyak pengembang server web telah mengadopsi kode kesalahan 418 dari standar kopi lelucon ke HTTP, meskipun Anda dapat mengabaikannya dengan aman saat mengimplementasikan aplikasi Anda! Rangkaian kode galat HTTP terakhir adalah 5xx.

"500 Internal Server Error" adalah salah satu yang pernah Anda temui sebelumnya, dan merupakan status galat "catch all", di mana beberapa jenis galat telah terjadi pada server (yang bukan kesalahan klien). Seperti kode 4xx, kode 5xx juga harus mengirimkan isi permintaan, tetapi tidak seperti kode 4xx, sebaiknya informasi disembunyikan di sini. Beberapa kerangka kerja memiliki mode debug di mana permintaan 500 akan mengembalikan jejak tumpukan, tetapi ini harus dinonaktifkan dalam produksi, dan informasi galat hanya boleh masuk ke log (ini juga dapat menjadi ide yang bagus untuk dilakukan dalam pengembangan).

Alasan utamanya adalah bahwa jejak tumpukan atau pesan galat bertele-tele lainnya dapat secara tidak sengaja membocorkan data sensitif (seperti melaporkan kata sandi basis data jika koneksi basis data gagal) yang tidak ingin Anda ketahui oleh dunia luar. Setelah metadata di sekitar permintaan Anda dirancang, Anda juga perlu merancang seperti apa isi permintaan dan respons Anda. JSON sangat populer, tetapi bukan satu-satunya format yang perlu dipertimbangkan. XML pernah disukai, tetapi sekarang sudah tidak umum lagi, karena dapat menjadi cara yang sangat rumit untuk mengekspresikan informasi.

XML lebih unggul daripada JSON jika Anda ingin pengguna mengekspresikan metadata tambahan sebaris dengan teks (misalnya, mungkin mengekspresikan informasi pemformatan dalam pengiriman email), tetapi untuk sebagian besar permintaan kunci/nilai sederhana, JSON efektif. Jika klien API adalah peramban web, ada baiknya juga mempertimbangkan format dengan tipe MIME “application/x-www-form-urlencoded,” yang merupakan format asli formulir HTML. Keunggulan historis XML dibandingkan JSON adalah XML memiliki definisi skema yang menyediakan cara yang baik untuk memeriksa kelengkapan format XML, dan skema XML didukung secara luas.

Ada alat yang menyediakan kemampuan serupa untuk JSON, meskipun tidak ada dukungan universal. Skema JSON adalah format populer untuk mendokumentasikan struktur data, dan umumnya dikombinasikan dengan alat seperti RAML dan Swagger untuk membuat dokumentasi dan validasi API. Ada banyak format serialisasi lain yang perlu dipertimbangkan, dan beberapa kerangka kerja akan bersifat agnostik terhadap format dan mengonversinya untuk Anda, yang memungkinkan Anda menerima banyak format dalam satu permintaan. Namun, hal ini menambah kompleksitas dan diketahui secara tidak sengaja menyebabkan

masalah keamanan.

Misalnya, YAML adalah bahasa yang sangat canggih yang memungkinkan Anda untuk menentukan fungsi dalam kode, yang telah menyebabkan peretasan situs-situs populer seperti GitHub dan Equifax, di mana bug atau deserialisasi yang tidak tepat telah menyebabkan permintaan mengeksekusi kode pada sistem Anda. Berpegang pada satu format serialisasi adalah hal yang wajar untuk dilakukan dan membatasi kompleksitas aplikasi Anda. Bahkan ketika Anda melakukannya, kehati-hatian harus dilakukan. Misalnya, eksploitasi XML terkenal yang dikenal sebagai "billion laughs" dapat membuat dokumen yang relatif kecil didekompresi untuk menggunakan beberapa gigabyte dalam memori saat diurai.

Banyak parser sering kali memiliki mode "aman" yang memperlakukan input sebagai tidak tepercaya sebelum mengurainya untuk melindungi dari serangan semacam ini, tetapi saat melakukan reserialisasi konten dari sumber yang tidak tepercaya, Anda harus memastikan bahwa Anda sepenuhnya memahami fitur pustaka yang Anda gunakan untuk melakukan deserialisasi dan membangun penanganan kesalahan yang kuat. Struktur aktual permintaan dan respons Anda tentu saja akan bergantung pada kasus penggunaan Anda.

Namun, menjaga paritas antara respons dan permintaan dapat membuat hidup lebih sederhana. Misalnya, jika Anda GET URL yang mendukung pembaruan, Anda seharusnya dapat GET kembali isi permintaan tanpa benar-benar membuat perubahan apa pun. Namun, terkadang struktur data akan merujuk ke sumber daya lain yang tertaut, seperti artikel yang mungkin memiliki tautan ke artikel lain yang direkomendasikan.

Saat Anda GET sumber daya, untuk alasan kinerja klien tidak akan selalu ingin mencari semua sumber daya yang ditautkan melalui beberapa permintaan untuk merender halaman, jadi hal yang umum dilakukan adalah mengizinkan tanda? expand=true, atau yang serupa, untuk ditetapkan pada permintaan GET yang menambahkan data tambahan di sekitar sumber daya yang ditautkan untuk meminimalkan jumlah pekerjaan yang perlu dilakukan klien. Sumber daya yang diperluas ini mungkin tidak dapat dikembalikan sebagaimana adanya.

Hal lain yang perlu dipertimbangkan saat mendesain skema untuk API Anda adalah bahwa beberapa tingkat perubahan tidak dapat dihindari. Misalnya, Anda mungkin ingin menambahkan kolom baru di masa mendatang. Anda harus dapat menambahkan kolom baru tanpa harus memperbarui klien mana pun (klien harus mengabaikan hal tambahan apa pun yang tidak mereka kenali), tetapi setiap restrukturisasi besar akan memerlukan beberapa kerusakan, atau Anda harus menangani struktur data baru dan lama secara paralel selama beberapa waktu sementara semua pengguna API melakukan pembaruan.

Salah satu kesalahan umum adalah menggunakan daftar JSON sebagai sumber daya tingkat atas. Misalnya, jika Anda memiliki titik akhir yang mengembalikan hasil penelusuran, Anda mungkin hanya memiliki daftar objek. Namun, di masa mendatang, Anda mungkin memutuskan untuk memperkenalkan pagination, yang melibatkan penambahan kolom, seperti nomor halaman saat ini. Jika API malah mengembalikan objek alih-alih daftar, Anda dapat menambahkan kolom tersebut, tetapi dengan mengembalikan daftar di tingkat atas, itu sekarang menjadi perubahan yang merusak. Anda harus selalu menggunakan objek sebagai sumber daya tingkat atas.

Salah satu teknik umum adalah menyertakan nomor versi di URL titik akhir API misalnya, “https://api.example.com/v1/products/.” Aliran pemikiran lain mengatakan bahwa jika tipe MIME konten kustom digunakan, nomor versi dapat dikodekan di dalamnya. Beberapa orang lebih suka menghilangkan pembuatan versi sepenuhnya, dengan alasan bahwa jika titik akhir memerlukan penulisan ulang yang lengkap, itu harus berupa URL yang benar-benar baru, atau layanan yang benar-benar baru, daripada versi baru dari API yang sudah ada.

Versi yang lebih lama kemudian dapat berjalan secara paralel dengan yang lebih baru hingga semua pengguna telah bermigrasi ke sistem baru, yang pada saat itu mereka dapat dinonaktifkan. Apa pun yang Anda putuskan untuk menangani pembuatan versi dan perubahan, Anda harus menyertakannya dari awal untuk mengantisipasi perubahan, meminimalkan risiko merusak pengguna API Anda.

10.3 MENGAMANKAN API

API sering kali merupakan bagian yang cukup kuat dari infrastruktur Anda, dan Anda sering kali ingin membatasi beberapa, atau semua, API Anda dari Internet publik. Ada sejumlah cara untuk mencapainya, dan ikhtisar keamanan yang lebih umum dibahas dalam bab Keamanan. Penting untuk memahami perbedaan antara autentikasi dan otorisasi (terkadang disingkat sebagai "*authn*" dan "*authz*"). Autentikasi berkaitan dengan pembuktian bahwa seseorang adalah orang yang mereka katakan, dan otorisasi adalah pembuktian bahwa seseorang memiliki akses ke sumber daya dan tindakan yang mereka coba akses.

Cara sederhana untuk mengamankan API Anda mungkin dengan membuat firewall sehingga hanya aplikasi yang memerlukan akses yang dapat mengaksesnya. Ini berisiko jika digunakan sendiri, karena jika seseorang membahayakan server yang memiliki akses ke API tersebut, mereka tiba-tiba memiliki kunci kerajaan. Ini juga berarti bahwa semua aplikasi memiliki akses penuh ke semua API, bukan hanya sebagian.

Melangkah lebih jauh dari ini mungkin melibatkan pengenalan tingkat autentikasi, tetapi tetap memberikan akses ke semua panggilan API kepada siapa saja yang dapat mengautentikasi ("berada di balik firewall" mungkin dianggap sebagai bentuk autentikasi yang paling sederhana). HTTP sendiri memiliki mekanisme bawaan untuk autentikasi dalam bentuk header permintaan, dan jika Anda menggunakan server seperti Apache atau nginx, Anda dapat mengonfigurasi autentikasi berbasis nama pengguna/sandi sederhana dengan cukup mudah.

Namun, dalam pengaturan yang lebih besar, memelihara berkas nama pengguna/sandi sederhana dapat menjadi tantangan, dan beberapa perusahaan besar menggunakan fitur HTTPS yang dikenal sebagai sertifikat klien TLS untuk mengautentikasi pengguna API. Seperti sertifikat HTTPS normal pada server, yang memverifikasi bahwa server adalah yang diklaimnya, sertifikat klien dikirim ke server dari klien untuk membuktikan bahwa siapa pun yang membuat permintaan juga mengetahui siapa mereka.

Sertifikat klien sangat kuat karena lebih sulit diserang melalui brute-force dan dapat diatur agar kedaluwarsa atau dicabut secara manual, yang dapat memusatkan penanganan kredensial yang dicuri. Mereka juga dapat diintegrasikan dengan login OS di banyak desktop untuk pengguna, yang dapat meminimalkan jumlah nama pengguna dan kata sandi yang harus

diingat pengguna. Namun, menyiapkan perkakas yang diperlukan untuk membuat dan mengelola sertifikat klien (dikenal sebagai infrastruktur kunci publik X.509) itu rumit, jadi hanya umum di perusahaan besar. Pendekatan umum lainnya adalah menggunakan kunci API sebagai header atau dalam string kueri, yang kemudian diperiksa terhadap basis data.

Anda juga sering ingin memperkenalkan otorisasi ke API Anda. Misalnya, jika Anda mendistribusikan aplikasi seluler yang berbicara langsung ke API, maka seseorang dapat mendekompilasi kode Anda untuk mengekstrak kredensial API. Jika hanya memiliki kredensial ini memberikan akses lengkap ke sistem, ini jelas tidak diinginkan. Salah satu cara menerapkan otorisasi mungkin dengan memasukkan ke dalam kode aplikasi Anda bahwa siapa pun yang telah diautentikasi pengguna memiliki akses untuk melakukan apa pun yang telah mereka minta untuk dilakukan.

Ini sering disebut sebagai daftar putih, di mana pengguna tidak memiliki akses untuk melakukan apa pun kecuali mereka ada dalam daftar putih yang menyertakan nama pengguna mereka, dan bagian aplikasi yang dapat mereka akses. Seringkali sifat dari hal-hal ini bergantung pada logika bisnis Anda, jadi mungkin sulit untuk menjadi generik. Terkadang daftar putih dapat digabungkan ke dalam aplikasi atau sebagai konfigurasi, tetapi ini bisa jadi tidak fleksibel (misalnya, Anda mungkin perlu menerapkan perubahan konfigurasi untuk menghapus seseorang dari daftar putih), dan juga sulit untuk melihat apa yang dapat diakses pengguna atas seluruh aset.

Pendekatan umum lainnya adalah memiliki layanan otorisasi pusat yang dapat dihubungi sistem untuk memeriksa apakah pengguna diberi otorisasi atau tidak. Seringkali, logika autentikasi/otorisasi ini dapat berakhir jauh di dalam kode Anda, yang menyebabkan berbagai masalah dalam logika pengontrol apa pun. Beberapa kerangka kerja menggunakan metode seperti dekorator untuk membantu hal ini, tetapi pendekatan umum lainnya adalah menggunakan layanan yang menyediakan gateway API. Gateway API ini berada di depan layanan Anda dan menyediakan autentikasi dan otorisasi. Autentikasi biasanya disediakan menggunakan kredensial HTTP atau dengan kunci API, tetapi pemeriksaan validitasnya terpusat di layanan gateway.

Otorisasi kemudian biasanya diperiksa terhadap daftar putih yang menentukan URL mana yang diizinkan untuk diakses oleh suatu layanan, yang dapat menjadi pemeriksaan tingkat atas yang efektif dan juga dapat menyederhanakan pengembangan. Namun, gateway API menimbulkan risikonya sendiri misalnya, jika dilewati, atau jika otorisasi yang dibutuhkan lebih rinci. Ada juga protokol otorisasi yang tidak menangani autentikasi. Yang paling terkenal adalah OAuth, tetapi standar lain seperti JSON Web Tokens juga ada di ruang ini. Dengan OAuth (dan OAuth 2), aplikasi yang perlu mengakses API (misalnya, beberapa kode front-end atau aplikasi seluler) membuat permintaan untuk izin tertentu, lalu memberikan permintaan tersebut kepada pengguna dan mengirim mereka ke API untuk masuk menggunakan kredensial mereka dan membuat kunci API yang memiliki kemampuan khusus yang dibutuhkan aplikasi.

Setelah aplikasi memiliki kunci API tersebut, aplikasi dapat menggunakannya untuk mengakses titik akhir yang dibutuhkannya, tetapi tidak mengetahui apa pun tentang siapa

pengguna tersebut. Sering kali, salah satu titik akhir yang mungkin disediakan aplikasi adalah fungsi "siapa saya?" yang memungkinkan aplikasi yang diotorisasi untuk mengakses informasi autentikasi, tetapi ini belum tentu terjadi. Ini tidak sering digunakan untuk komunikasi server ke server yang tidak melibatkan pengguna.

JSON Web Tokens (JWT) adalah bentuk yang lebih sederhana dari ini. JWT hanyalah pernyataan tentang sesuatu (mungkin siapa pengguna tersebut, atau apa yang dapat mereka akses) yang kemudian ditandatangani secara kriptografis oleh penerbit. Aplikasi kemudian dapat memverifikasi tanda tangan untuk memeriksa bahwa pernyataan tersebut tidak palsu. Lebih jauh dari sekadar mengautentikasi dan mengotorisasi pengguna, ada juga teknik yang dapat digunakan untuk lebih melindungi API Anda dari klien nakal.

Teknik umum adalah pembatasan laju, yang berarti tidak ada satu pun pengguna API Anda yang dapat menghasilkan muatan yang tidak sesuai yang dapat menyebabkan masalah kinerja bagi klien lain. Ini khususnya berguna untuk API yang tindakannya dapat berdampak tinggi pada biaya atau kinerja. Pembatasan laju umumnya diterapkan dengan mengizinkan klien mengajukan permintaan maksimum dalam jangka waktu tertentu, dan jika melampaui batas tersebut, akan mengembalikan kode kesalahan alih-alih melakukan tindakan. Pembatasan laju dapat diterapkan di dalam aplikasi, tetapi merupakan fitur umum gateway API.

10.4 API BERBASIS PERISTIWA

Meskipun jenis API yang paling populer di Web adalah API RESTful tradisional, ada paradigma API yang berbeda yang dikenal sebagai API berbasis peristiwa. API RESTful berguna saat Anda ingin menemukan atau mengubah status sistem eksternal, tetapi terkadang aplikasi Anda perlu mengetahui kapan perubahan telah terjadi dan berperilaku sebagaimana mestinya. Untuk aplikasi back-end, biasanya aplikasi diberi tahu saat terjadi perubahan. Misalnya, sistem pencetakan label pengiriman mungkin diinstruksikan, melalui permintaan dari sistem koordinasi, untuk mencetak label saat pesanan diambil untuk pemenuhan.

Selain itu, proses yang sangat rumit, di mana mungkin banyak sistem perlu diberi tahu saat terjadi perubahan, dapat memberikan banyak tanggung jawab pada satu komponen, mungkin dengan logika yang didistribusikan jika ada banyak tempat berbeda di mana perubahan dapat terjadi. Untuk aplikasi front-end yang perlu menyadari adanya perubahan, jarang sekali sistem back-end memiliki cara untuk mengomunikasikannya, karena sistem tersebut tidak mengekspos API REST itu sendiri.

Misalnya, pikirkan program obrolan web, di mana pengguna ingin melihat apakah ada pengguna lain di ruang obrolan yang mengatakan sesuatu yang baru. Solusi "cepat" untuk ini adalah membuat beberapa permintaan ke REST API dan kemudian memeriksa apakah responsnya berubah. Namun, ini akan membutuhkan banyak pekerjaan jika perubahannya jarang terjadi, atau ada banyak elemen yang mungkin berubah. Ini juga menambah penundaan misalnya, jika Anda memeriksa perubahan sekali per detik, mungkin perlu waktu hingga satu detik bagi Anda untuk melihat pesan baru.

Sebaliknya, klien dapat membuka koneksi ke server, dan ketika status berubah di server,

server mengirimkan pesan ke semua klien melalui koneksi yang dibiarkan terbuka. Namun, model Web asli tidak mendukung mode interaksi ini, jadi solusi sementara digunakan pada awalnya untuk mengaktifkannya. Pada tahun 2000, sebuah teknik yang dikenal sebagai Comet diperkenalkan, di mana server tampaknya mengirimkan konten halaman dengan sangat lambat, dan halaman yang lambat ini ditambahkan sebagai IFrame. Server kemudian menulis tag <script> dengan beberapa JavaScript yang akan memanggil fungsi yang sesuai setiap kali suatu peristiwa terjadi.

Setelah diperkenalkannya AJAX, Comet diperluas untuk mendukung "long polling." Dalam aplikasi yang menggunakan long polling, pada awalnya diberikan semua data status sistem saat ini, dan kemudian lamanya waktu data tersebut valid. Permintaan AJAX mencakup semua perubahan sejak saat itu, tetapi jika tidak ada perubahan, alih-alih mengembalikan daftar kosong, server tidak mengembalikan respons, membiarkan koneksi terbuka, hingga ada perubahan yang perlu diberitahukan kepada klien, lalu mengirimkan respons. Respons ini mencakup waktu baru, dan klien mengulangi proses ini.

Diperkenalkan pada tahun 2011, WebSockets adalah fitur HTML5 yang menghilangkan kebutuhan akan teknik seperti Comet. WebSockets memungkinkan browser untuk menyiapkan koneksi ke server yang memungkinkan percakapan dua arah dengan mengirimkan pesan ke dan dari server, daripada harus membuat koneksi baru untuk permintaan, yang dipaksakan oleh polling yang panjang. Untuk sistem back-end, teknologi antrean pesan dapat digunakan untuk mencapai tujuan yang sama.

Metode yang baru saja saya jelaskan didasarkan pada gagasan untuk mengirimkan status awal dan kemudian berlangganan perubahan. Ini bergantung pada sistem hulu yang mempertahankan beberapa status awal. Pendekatan lain adalah dengan menerima semua peristiwa yang pernah terjadi dalam sistem dan kemudian memprosesnya agar aplikasi Anda membentuk pandangannya sendiri tentang status sistem, daripada mengandalkan sistem eksternal untuk beberapa status awal. Setelah sistem Anda diterapkan, sistem tersebut kemudian dapat membaca hanya perubahan baru dan memprosesnya, tetapi ini memberi Anda keuntungan.

Misalnya, jika Anda perlu memigrasikan basis data ke skema baru atau membangunnya kembali setelah data rusak, Anda cukup memutar ulang semua peristiwa untuk membangun status baru (meskipun sebaiknya Anda mencatat peristiwa mana yang telah Anda lihat dan proses untuk menghindari tindakan pemicu seperti menerima pembayaran dua kali). Pada titik tertentu, jumlah peristiwa yang pernah diterima dalam suatu sistem mungkin terlalu besar, sehingga beberapa peristiwa awal yang telah sepenuhnya digantikan dibuang (misalnya, dalam kasus peristiwa "pembaruan alamat", hanya peristiwa terbaru dari peristiwa ini untuk pelanggan individual yang benar-benar penting).

Pendekatan "hanya peristiwa" ini sudah maju, tetapi berguna dalam skala besar dalam sistem yang terdistribusi secara masif, di mana menghilangkan satu titik kegagalan (API yang menyimpan status) diinginkan (meskipun dengan mengorbankan pengenalan sistem pialang peristiwa baru Anda sebagai satu titik kegagalan baru).

Menemukan API

Setelah Anda menerapkan API, aplikasi yang ingin menggunakannya akan ingin mengetahui URL tempat API dapat diakses. Untuk aplikasi yang sangat sederhana, ini dapat dikodekan secara permanen ke dalam kode yang membuat permintaan, tetapi praktik ini tidak berkelanjutan, terutama jika Anda bekerja di beberapa lingkungan. Kode di lingkungan produksi Anda mungkin perlu berkomunikasi dengan versi produksi API, tetapi dalam lingkungan pengembangan, sering kali akan ada API yang berbeda.

Ada banyak solusi untuk masalah ini, tetapi yang paling sederhana adalah dengan menjadikan URL sebagai opsi konfigurasi (disediakan dalam file konfigurasi atau sebagai variabel lingkungan, misalnya). Saat API Anda disebarkan, masuk akal untuk memberinya nama yang masuk akal di DNS (misalnya, productapi.dev.example.com vs. productapi.prod.example.com), dan jika ada beberapa server yang berjalan pada API di setiap lingkungan, URL tersebut dapat mengarah ke penyeimbang beban yang memiliki beberapa server di belakangnya. Ini berfungsi saat Anda perlu memigrasikan API ke server lain juga.

Pendekatan ini berfungsi dengan baik dalam banyak kasus penggunaan, meskipun ada beberapa skenario yang mungkin terlalu sederhana. Contoh utamanya adalah saat Anda mencari contoh sumber daya tertentu, jadi mengabstraksikannya di balik penyeimbang beban atau mengodekan nama DNS secara keras tidak selalu berhasil. Dalam beberapa kasus, menyelesaikan masalah ini di balik API dapat membantu, sehingga penemuan layanan menjadi masalah internal komponen tersebut, bukan masalah yang terekspos ke tepi sistem.

Situasi lain saat menentukan nama layanan dependen sebagai nilai konfigurasi mungkin tidak diinginkan adalah jika Anda sering menyediakan lingkungan baru, yang mengharuskan URL baru untuk ditentukan. Skrip bootstrap yang secara otomatis menghasilkan file konfigurasi ini menurut templat dapat mengatasi masalah ini. Ada banyak alat dan aplikasi yang dapat membantu mengatasi masalah penemuan layanan ini. Pendekatan standar adalah dengan mendaftarkan komponen itu sendiri dengan alat penemuan layanan, lalu menggunakannya untuk menjalankan kueri guna menemukan nama host layanan individual.

Dalam kasus ini, Anda hanya perlu mengetahui cara menemukan alat penemuan layanan itu sendiri untuk menemukan alat lainnya. Alat penemuan layanan menjadi ketergantungan penting dalam aplikasi Anda, jadi sebaiknya Anda hanya menggunakannya jika Anda benar-benar yakin membutuhkannya.

Menggunakan API

Setelah Anda membuat API, dan klien telah menemukan lokasi API tersebut, klien juga perlu menggunakan API tersebut. Ini mungkin tampak semudah membuat permintaan HTTP yang sesuai (biasanya dengan semacam kredensial) dan mengurai respons, dan dalam banyak kasus, ini memang yang diperlukan. Namun, untuk beberapa API atau jenis interaksi, Anda mungkin perlu mengambil pendekatan yang lebih cermat terhadap cara Anda menggunakan API tersebut.

Berinteraksi dengan API berarti Anda berkomunikasi dengan bagian lain dari sistem Anda, dan sistem eksternal itu dapat tiba-tiba mulai berperilaku tidak terduga atau gagal

dengan cara yang tidak terduga. Oleh karena itu, penanganan kesalahan menjadi bagian penting dari cara Anda berinteraksi dengan API. Dengan penanganan kesalahan, Anda harus memperhitungkan kesalahan yang diharapkan (seperti kesalahan 404 Tidak Ditemukan saat memeriksa apakah sumber daya tertentu ada), dan kesalahan yang tidak diharapkan (misalnya, kesalahan 500 atau respons yang diindikasikan valid, tetapi mungkin tidak dapat diurai).

Memahami mode kegagalan API yang Anda gunakan penting karena, sering kali, kegagalan perlu disebarkan melalui sistem Anda, mungkin pemberitahuan ke tim operasi atau tercermin dalam antarmuka pengguna Anda. Merupakan hal yang umum untuk membungkus komunikasi Anda ke API eksternal dengan modul di dalam kode klien Anda, sehingga semua penanganan API terjadi di satu tempat.

Modul ini harus dapat menunjukkan ketika kesalahan terjadi pada pengguna modul ini (mungkin dengan menolak janji atau melempar pengecualian), tetapi harus melakukannya dengan meneruskan kesalahan yang masuk akal kepada pemanggil, daripada hanya langsung meneruskan kesalahan dari API jarak jauh. Jika kesalahan diteruskan secara langsung, mungkin tidak jelas bagi pemanggil apa yang sebenarnya terjadi, atau dapat menyebabkan logika diduplikasi di seluruh sistem Anda apakah itu kesalahan yang diharapkan atau tidak terduga.

Penanganan kesalahan menjadi lebih rumit jika panggilan API merupakan salah satu dari serangkaian panggilan. Jika salah satu gagal, maka sistem Anda mungkin berada dalam kondisi yang tidak sesuai (misalnya, jika bagian depan kasir memanggil API pembayaran yang berhasil, tetapi kemudian mengajukan permintaan ke sistem pengiriman, yang gagal). Penanganan kesalahan semacam ini rumit; dalam beberapa keadaan, mungkin tepat untuk membatalkan tindakan yang berhasil (meskipun jika pembatalan gagal, Anda mungkin mengalami kendala) dan menunjukkan kegagalan tersebut kepada pengguna, tetapi dalam keadaan lain Anda mungkin perlu memberi tahu seseorang bahwa intervensi manual diperlukan.

Atau, mengantrekan permintaan untuk dicoba ulang mungkin efektif, tetapi jika permintaan semacam ini umum terjadi di sistem Anda, maka mungkin hal itu menunjukkan bahwa desain API Anda perlu dipikirkan ulang, atau bahwa beralih ke model berbasis peristiwa akan lebih tepat. Bagian Mendesain untuk Kegagalan dalam bab Sistem membahas cara menangani kegagalan antar modul secara lebih terperinci, termasuk pola pemutus arus, yang dapat digunakan untuk membantu melindungi pengalaman pengguna Anda selama pemadaman.

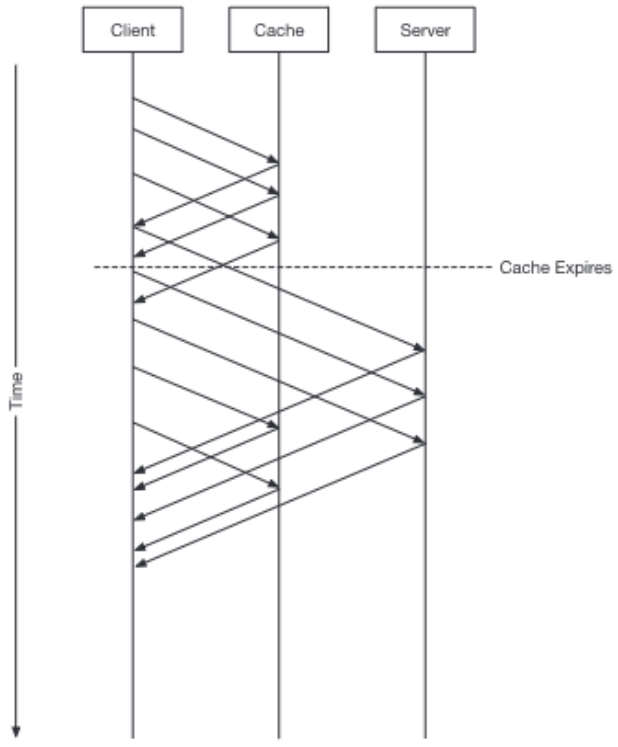
Memiliki modul dalam aplikasi Anda yang menangani API yang perlu Anda gunakan adalah pola umum, karena hal itu menyisakan satu titik tanggung jawab untuk hal-hal lain yang mungkin ingin Anda lakukan saat berinteraksi dengan sistem eksternal. Masalah ini sering kali mencakup pencatatan interaksi dengan sistem eksternal (terutama kegagalan), tetapi juga masalah seperti pemantauan. Mengukur aspek-aspek seperti waktu respons terhadap sistem eksternal, serta menghitung jumlah permintaan yang berhasil dan gagal, memungkinkan Anda untuk menetapkan ambang batas pemantauan yang dapat membantu men-debug masalah apa pun dalam sistem yang diterapkan.

Jika API yang Anda gunakan menerapkan batas kecepatan, Anda mungkin juga ingin

mengelolanya di klien yang menggunakan API tersebut. Salah satu pendekatannya adalah dengan mengintegrasikan dengan pemutus arus, sehingga jika Anda mencapai batas kecepatan, Anda akan mengaktifkan pemutus arus hingga akhir periode batas kecepatan tersebut. Pendekatan lainnya adalah dengan mengantrekan permintaan untuk dicoba ulang, dan mencobanya ulang pada kecepatan di bawah batas.

Caching dapat menjadi pendekatan yang efektif untuk menangani batasan kecepatan, dan juga memiliki keuntungan bagi kinerja aplikasi Anda. Namun, caching bukan tanpa masalah, dan harus dipertimbangkan dalam konteks seluruh aplikasi Anda. Tidak banyak gunanya menambahkan cache ke aplikasi Anda jika sudah ada cache pada API, karena ini meningkatkan risiko informasi basi dan tidak ada peningkatan kinerja. Saat mengimplementasikan cache di aplikasi Anda, hal paling sederhana yang dapat dilakukan adalah merekam respons di cache Anda (mungkin menggunakan alat seperti memcached, sehingga dibagikan ke seluruh tingkatan aplikasi Anda), dan jika respons ada di cache, gunakan itu alih-alih membuat permintaan ke API yang sebenarnya.

Jika tidak ada, buat permintaan dan catat respons di cache. Sering kali, hanya respons positif yang di-cache, untuk menghindari kesalahan yang bertahan lebih lama dari yang seharusnya, tetapi terkadang mungkin diinginkan untuk menyimpan cache respons kesalahan apa pun, terutama jika itu adalah kesalahan yang diharapkan (seperti 404). Beberapa serangan DoS diketahui menyerang halaman 404 berulang kali, karena halaman ini masuk ke bagian belakang untuk memeriksa apakah halaman tersebut ada setiap saat.



Gambar 10.2 Garis Waktu Permintaan Dan Respons Antara Klien Dan Server Melalui Lapisan Caching, Yang Menunjukkan Masalah Kawanan Guntur

Dalam kasus ini, caching disebut caching negatif. Caching negatif dapat menjadi masalah, karena bug dapat menyebabkan status kesalahan di cache secara tidak benar. Pendekatan sederhana terhadap caching ini juga dapat menyebabkan masalah. Pertimbangkan fakta bahwa, saat cache kedaluwarsa, jika ada beberapa permintaan yang masuk sekaligus, akan ada periode waktu antara cache kedaluwarsa dan respons yang berhasil ditambahkan ke cache. Selama waktu ini, API dapat menerima banyak permintaan yang identik sekaligus, yang berpotensi menyebabkan kesalahan. Ini dikenal sebagai "masalah kawanan guntur," dan ditunjukkan pada Gambar 10.2

Di sini, sejumlah permintaan dibuat, tetapi antara cache kedaluwarsa dan respons berikutnya yang dikirimkan, sejumlah permintaan tambahan dibuat, yang dapat membanjiri layanan bagian belakang. Setelah respons itu diterima, permintaan lebih lanjut sekali lagi dilayani dari cache. Salah satu solusi umum untuk hal ini adalah tidak menghapus item dari cache saat item tersebut kedaluwarsa, tetapi menandainya sebagai basi. Klien pertama yang membaca dari cache setelah item menjadi basi kemudian memperoleh kunci dan membuat permintaan untuk menyegarkan cache.

Sementara itu, klien tersebut, dan klien lain yang melihat item basi dalam cache setelah kunci diperoleh, menyajikan item basi kepada klien. Ini dikenal sebagai "stale-while-revalidate."Mempertahankan item basi dalam cache juga memungkinkan Anda untuk menggunakan teknik terkait, yang dikenal sebagai "stale-on-error." Jika API Anda mulai mengalami masalah atau macet, maka front end masih dapat menampilkan informasi dengan menggunakan informasi basi dalam cache. Namun, penggunaan item basi tidak selalu tepat.

Jika pengguna mengandalkan ketepatan waktu informasi, menampilkan kesalahan yang menunjukkan bahwa informasi terkini tidak dapat ditampilkan adalah solusi yang lebih baik. Ringkasan API ada untuk berbagai bagian sistem Anda, dan bagian di luarnya, untuk berkomunikasi. API dapat dianggap sebagai pengalaman pengguna untuk aplikasi lain, karena harus dirancang agar dapat digunakan dan memudahkan integrasi. API juga sulit diubah, karena memerlukan penyesuaian pada banyak sistem.

Terkadang perubahan ini mudah dikoordinasikan, terutama saat Anda bekerja dalam sistem yang semua bagiannya Anda kendalikan, tetapi untuk yang lain, saat pihak ketiga mengintegrasikannya, perubahan tersebut harus dikelola dengan saksama. Pembuatan versi sering kali digunakan untuk melakukan ini, tetapi klien API juga harus tangguh terhadap perubahan misalnya, jika bidang baru dalam respons muncul. Untuk sistem berbasis web, bentuk API yang paling populer adalah REST API (Representational State Transfer).

SOAP merupakan alternatif umum, terutama dalam sistem perusahaan, tetapi REST lebih populer, karena tumbuh dari primitif fundamental web. REST menggunakan kata kerja HTTP seperti GET, PUT, atau DELETE untuk memanipulasi sumber daya, yang diidentifikasi sebagai URL, dan operasi ini harus sesuai dengan beberapa properti. Kode respons HTTP standar digunakan untuk menunjukkan keberhasilan atau kegagalan suatu operasi. SOAP berkomunikasi dengan satu URL menggunakan permintaan dan respons yang sesuai dengan skema yang dipublikasikan, sedangkan permintaan dan respons REST bertujuan untuk mendeskripsikan dirinya sendiri menurut tipe MIME.

API lainnya berbasis peristiwa, di mana koneksi antara sistem tidak hanya berupa permintaan-respons, tetapi di mana satu sistem mendengarkan sistem lain dan mendapatkan pemberitahuan saat peristiwa terjadi. API merupakan bagian penting dari sistem Anda saat mempertimbangkan keamanan. Ini mencakup autentikasi dan otorisasi, untuk memastikan bahwa bukan hanya orang yang mereka klaim, tetapi juga mereka memiliki izin untuk menjalankan operasi yang dimaksud, terlepas dari apakah operasi tersebut telah diekspos kepada mereka di UI atau tidak.

Penggunaan API oleh klien juga merupakan sesuatu yang perlu dipertimbangkan, karena banyak server bergantung pada klien yang berperilaku wajar saat membuat permintaan agar tidak menimbulkan masalah kinerja, dan yang lainnya menegakkan kewajaran ini menggunakan teknik seperti pembatasan kecepatan. Menemukan API yang tepat juga merupakan keputusan penting bagi klien, dan ada banyak mekanisme untuk melakukan ini, mulai dari nilai konfigurasi dengan URL tertentu, hingga server penemuan layanan yang digunakan klien untuk menemukan API yang dibutuhkan.

BAB 11
MENYIMPAN DATA

11.1 PERBANDINGAN BASIS DATA RELASIONAL DAN NOSQL

Untuk situs web yang paling sederhana sekalipun, Anda harus menyimpan data di suatu tempat. Untungnya, penyimpanan data merupakan masalah yang telah diatasi sejak komputer pertama kali dirancang, dan telah ada penelitian selama beberapa dekade tentang cara terbaik untuk menyimpan dan mengambil data dari semua jenis, dalam skala mulai dari katalog produk kecil hingga sistem industri yang melacak dan menyimpan jutaan variabel dan peristiwa per detik. Data Anda sering kali merupakan aset paling berharga milik organisasi Anda, bukan hanya karena membantu Anda mencapai tujuan, tetapi juga karena potensi kerugian bagi Anda jika data tersebut tidak dilindungi dengan benar atau bocor secara tidak sengaja.

Meskipun pengembang yang bekerja di industri yang sangat diatur seperti keuangan atau perawatan kesehatan mungkin terbiasa dengan kepatuhan hukum atas produk mereka, ada banyak pengembang yang bekerja di industri yang tidak terlalu diatur, dan implikasi hukum pada rekayasa perangkat lunak tidak secara rutin dipertimbangkan. Peraturan perlindungan data dan privasi merupakan pengecualian dari aturan ini, karena berlaku untuk hampir semua industri. Memiliki arsitektur yang dipahami dengan baik tentang data yang Anda simpan dan di mana akan memudahkan untuk membangun perlindungan yang memastikan Anda mematuhi hukum.

Persyaratan ini tidak hanya legal, tetapi juga etis. Undang-undang privasi dan perlindungan data yang tepat bervariasi di antara negara-negara, tetapi dalam kebanyakan kasus, ada organisasi yang menawarkan saran tentang cara memastikan Anda tetap mematuhi persyaratan ini, seperti Kantor Komisioner Informasi Inggris. Bab ini hanya membahas sedikit tentang bagaimana Anda dapat menyusun data dan menggunakan basis data, tetapi seperti banyak keputusan lainnya, tidak ada satu jawaban yang benar.

Saat memilih strategi, faktor terpenting adalah data apa yang Anda simpan (artinya strukturnya, dan seberapa penting dan sensitifnya data tersebut), dan seberapa banyak data tersebut ukuran rata-rata setiap item dan berapa proporsi pembacaan terhadap penulisan. Banyak basis data, seperti katalog produk, akan jarang diperbarui tetapi dibaca lebih sering, sementara yang lain, seperti kereta belanja, mungkin memiliki keseimbangan yang lebih merata. Yang lain lagi (seperti analitik log) mungkin memiliki banyak penulisan tetapi pembacaan yang jarang untuk analisis. Fokus bab ini adalah pada jenis data pertama, yang sering kali menjadi bagian data paling penting dalam suatu organisasi.

Jenis-jenis Basis Data

Selama ini, basis data relasional seperti MySQL mendominasi dunia basis data, tetapi kini semakin banyak aplikasi yang menggunakan basis data “NoSQL”, seperti MongoDB atau Redis.Basis data relasional disebut demikian karena Anda dapat mengekspresikan hubungan antara data dalam skema. Basis data relasional terstruktur dalam bentuk tabel, di mana satu

rekaman dalam basis data tersebut disebut baris, dan setiap bidang dalam rekaman tersebut disebut kolom, dan semua baris memiliki kolom yang sama dengan tipe yang sama.

Hubungan berperan karena salah satu kolom ini dapat berupa “kunci asing”, yang merupakan pengenal yang berhubungan dengan rekaman di tabel lain. Biasanya, setiap rekaman memiliki “kunci utama” yang mengidentifikasinya secara unik di tabel tersebut. Misalnya, tabel “postingan” dapat memiliki kunci asing yang disebut “penulis” ke rekaman di tabel “pengguna” yang menentukan penulis postingan blog. Jika penulis kemudian mengubah nama, Anda cukup perlu memperbarui penulis tersebut di tabel pengguna dan semua postingan blog yang mereka tulis akan mewarisi perubahan tersebut.

Basis data relasional didukung oleh banyak sekali teori, dan basis data yang paling populer telah ada selama beberapa dekade, memberi mereka waktu untuk berkembang menjadi perangkat lunak yang berkinerja tinggi dan andal. Namun, basis data tersebut bukannya tanpa batasan. Jaminan yang diberikan banyak basis data relasional terhadap integritas data dapat mengakibatkan hambatan kinerja, dan dipaksa untuk menentukan skema di awal dapat mempersulit pengelolaan perubahan.

Alternatif untuk basis data relasional telah menjadi lebih populer dan telah diklasifikasikan sebagai basis data "NoSQL" (SQL adalah bahasa yang digunakan untuk bekerja dengan sebagian besar basis data relasional). Meskipun model konseptual semua basis data relasional serupa, ada banyak jenis basis data yang diklasifikasikan sebagai NoSQL. Jenis yang paling umum adalah penyimpanan nilai-kunci dan penyimpanan dokumen, tetapi Anda mungkin juga menemukan basis data kolom dan grafik.

Penyimpanan nilai-kunci dapat dianggap serupa dengan hash atau kamus dalam bahasa pemrograman dinamis, karena satu-satunya cara untuk menangani konten adalah dengan menggunakan kunci, dan data aktual yang disimpan pada kunci tersebut tidak jelas. Banyak penyimpanan nilai-kunci yang menawarkan beberapa abstraksi atas hal ini; misalnya, Redis memungkinkan kunci untuk merujuk ke satu set atau daftar, tetapi aturan menemukan nilai hanya menggunakan satu kunci masih berlaku. Nilai sebenarnya yang disimpan pada setiap kunci tidak harus sama, dan dapat sangat berbeda dalam struktur antar kunci. Penyimpanan dokumen seperti MongoDB melangkah lebih jauh.

Dalam penyimpanan dokumen, meskipun dokumen sering kali memiliki ID unik seperti kunci, dokumen tersebut sering kali hanya berupa kolom di dalam dokumen yang lebih besar. "Dokumen" ini cenderung menyimpan data yang terstruktur sebagai JSON atau XML, dan perbedaan utama antara jenis penyimpanan ini dan penyimpanan nilai-kunci adalah Anda dapat mencari data berdasarkan konten dokumen, bukan hanya kuncinya. Ada juga gabungan antara penyimpanan nilai-kunci dan penyimpanan dokumen. DynamoDB, misalnya, berperilaku seperti penyimpanan nilai-kunci, tetapi Anda juga dapat menambahkan skema untuk menjelaskan jenis data yang disimpan pada kunci yang memungkinkan Anda untuk melakukan kueri pada nilai tersebut juga.

Skema ini digunakan untuk menentukan indeks. Untuk kinerja, database relasional dan NoSQL memungkinkan Anda untuk menentukan indeks pada tabel atau koleksi, yang dapat digunakan untuk pencarian cepat dengan mencari data dalam indeks, daripada mencari semua

data mentah sebagai respons terhadap kueri. Basis data grafik seperti Neo4J merepresentasikan data sebagai grafik (yaitu, kumpulan item dengan koneksi yang diketik di antara mereka) dan memungkinkan Anda menulis kueri dengan melintasi grafik, sementara basis data kolom seperti Cassandra bekerja dengan menyimpan data menurut kolom, bukan menurut baris, seperti dalam basis data relasional tradisional.

Ini biasanya dilakukan untuk alasan kinerja saat menangani set data yang sangat besar. Salah satu karakteristik terpenting lainnya dari basis data adalah cara penskalaannya. Beberapa basis data hanya dapat diskalakan dengan tumbuh pada satu mesin besar, dan yang lainnya dapat didistribusikan ke beberapa mesin, yang juga dapat menawarkan ketahanan. Saat Anda ingin mendistribusikan data, teorema yang dikenal sebagai Teorema CAP Brewer berlaku. Teorema ini menyatakan bahwa layanan hanya dapat memiliki dua dari kualitas berikut: konsisten, tersedia, dan toleran terhadap partisi.

Tidak dapat memiliki ketiganya. Konsistensi di sini berarti bahwa sistem akan selalu mengembalikan hasil penulisan terbaru ke setiap pembacaan (atau kesalahan, jika sistem tahu itu tidak dapat diisi); ketersediaan berarti setiap permintaan menerima respons tanpa kesalahan (meskipun tidak ada jaminan bahwa respons tersebut konsisten); dan toleransi partisi berarti bahwa jika komunikasi terputus antara berbagai bagian sistem, sistem tersebut tetap beroperasi.

Teorema CAP berinteraksi dengan dua filosofi yang berlaku yang biasanya selaras dengan konsep basis data relasional dan NoSQL. Basis data relasional mencontohkan filosofi bahwa transaksi basis data harus bersifat ACID, yang berarti bahwa konsistensi dipilih daripada ketersediaan, dan basis data NoSQL sering kali menentukan bahwa transaksi pada akhirnya harus konsisten, di mana ketersediaan dipilih daripada konsistensi. ACID adalah akronim untuk:

- Atomik: Penulisan ke basis data yang terdiri dari beberapa operasi yang selesai secara keseluruhan di mana semua operasi telah memberikan efeknya, atau gagal secara keseluruhan sehingga tidak ada yang diterapkan.
- Konsisten: Setiap transaksi meninggalkan sistem dalam keadaan di mana semua aturan atau batasan validasi yang ditetapkan dalam basis data terpenuhi.
- Terisolasi: Jika beberapa penulisan diterapkan ke basis data pada saat yang sama, atau pembacaan terjadi saat transaksi sedang diproses karena eksekusi paralel, basis data akan dibiarkan dalam keadaan yang sama seolah-olah setiap transaksi telah dilakukan satu demi satu (yaitu, beberapa interaksi pada saat yang sama tidak saling mengganggu).
- Tahan lama: Setelah penulisan selesai (transaksi dilakukan), transaksi akan tetap dalam keadaan tersebut jika server dimatikan atau dimulai ulang.

Di sisi lain, sistem yang pada akhirnya konsisten (kadang-kadang disebut BASE sebagai lawan dari ACID: pada dasarnya tersedia, status lunak, pada akhirnya konsisten) dapat mencapai ketersediaan tinggi, tetapi dapat berarti bahwa penulisan dalam sistem terdistribusi mungkin tidak dapat segera dibaca kembali, atau pembacaan mungkin menghasilkan hasil yang berbeda jika diberikan di seluruh node yang berbeda. Sistem yang pada akhirnya konsisten dapat

berkinerja tinggi dan sangat tersedia, tetapi dapat membuatnya lebih sulit untuk menalar tentang status dalam keadaan tertentu.

Risiko lain dengan konsistensi akhirnya adalah bahwa jika transaksi diterapkan dalam urutan yang berbeda pada bagian lain dari sistem terdistribusi, node dapat berakhir dalam status yang berbeda, yang dapat sangat berbahaya. Bentuk konsistensi akhirnya yang dimodifikasi, yang dikenal sebagai konsistensi akhirnya yang kuat, menambahkan persyaratan tambahan bahwa semua node harus berada dalam status yang persis sama, bahkan jika transaksi diterapkan dalam urutan yang berbeda. Sebagian besar basis data yang pada akhirnya konsisten bersifat kuat.

11.2 PERBANDINGAN DATABASE: NOSQL VS RELASIONAL

Ke SQL, atau NoSQL?

Banyak database NoSQL dipromosikan berdasarkan keunggulan performa yang ditawarkannya atas database relasional, dan manfaat menghindari dilakukannya desain skema atau migrasi (perubahan pada skema yang mengharuskan data tersimpan diperbarui agar sesuai dengan skema baru), tetapi database ini juga memiliki kekurangan. Sebagian besar kekuatan dari basis data relasional berasal dari fakta bahwa tabel dapat "digabungkan" berdasarkan hubungan ini untuk mendapatkan hasil dalam satu kueri dari beberapa tabel yang berbeda.

Sebagian besar basis data NoSQL tidak mengizinkan penggabungan semacam ini, sehingga dapat menjadi kurang berperforma untuk jenis kueri tertentu. Hal ini sering kali diatasi dengan mengulang informasi di dalam dokumen (misalnya, nama penulis, daripada menautkan ke tabel penulis), sebuah proses yang dikenal sebagai denormalisasi. Basis data yang dinormalisasi tidak berisi informasi yang berulang atau berlebihan. Meskipun tabel yang didenormalisasi tidak selalu bermasalah, tabel tersebut mempermudah terjadinya inkonsistensi, dan dapat mempersulit pembaruan data yang muncul di banyak tempat dalam basis data.

Beberapa basis data relasional kini mengadopsi teknik dari dunia NoSQL. Misalnya, PostgreSQL mendukung tipe kolom JSON, daripada harus menyimpan semua data dalam tabel datar. Apakah akan menggunakan relasional atau NoSQL bergantung pada tipe data yang Anda miliki. Meskipun basis data NoSQL menghindari kendala dalam menentukan skema secara formal, Anda sering kali ingin memiliki beberapa jenis skema untuk menjaga struktur data yang wajar, baik untuk dibaca dari aplikasi Anda atau untuk membuat indeks.

Basis data NoSQL juga dapat lebih cepat untuk ditulis, jadi jika Anda memiliki lebih banyak penulisan daripada pembacaan, ini bisa menjadi pilihan yang lebih baik, tetapi untuk sebagian besar jenis data lainnya, basis data relasional akan sangat membantu Anda. Ada banyak pengecualian untuk aturan praktis ini, tetapi yang perlu disebutkan adalah data yang sering disebut "data sesi". Ini dapat mencakup konten keranjang belanja, atau ID pengguna yang saat ini masuk. Sesi sering kali kedaluwarsa, membatasi migrasi apa pun ke data yang berumur panjang, dan hanya disimpan sebagai satu gumpalan data yang diidentifikasi oleh ID.

Penyimpanan nilai-kunci seperti Redis sangat cocok untuk skenario ini. Beberapa

kerangka kerja sering mendukung penyimpanan jenis data ini dalam memori aplikasi ini sangat umum di Java tetapi ini harus dihindari. Memulai ulang atau menerapkan layanan yang menggunakan penyimpanan sesi dalam memori sering kali akan membuat pengguna keluar dan kehilangan data, dan hal itu juga dapat menyebabkan masalah dengan skala horizontal, yang merupakan atribut penting untuk dirancang saat menargetkan infrastruktur cloud.

Tempat Menyimpan Data Anda

Meskipun ada banyak database mandiri, ada juga database yang bekerja dengan menanamkan dirinya sendiri dalam aplikasi Anda, seperti SQLite, atau Neo4J (meskipun ini juga memiliki mode mandiri), tetapi seperti penyimpanan data sesi dalam memori, hal ini dapat berdampak negatif pada skalabilitas, memaksa Anda untuk hanya menjalankan satu instans server aplikasi Anda. Memisahkan database Anda ke dalam layanan terpisah (seperti yang ditawarkan server seperti MySQL dan MongoDB) memungkinkan Anda untuk menskalakan tingkatan database dan aplikasi Anda secara terpisah.

Beberapa bahasa dan kerangka kerja memudahkan untuk secara tidak sengaja memperkenalkan penyimpanan data ke dalam tingkatan aplikasi Anda. Misalnya, di Java, bidang privat pada kelas pengontrol dapat tetap ada di antara permintaan dan menjadi tempat penyimpanan status secara tidak sengaja. Bahasa seperti PHP atau Perl (dalam penerapan umum) sebaliknya mempersulit hal ini, karena setiap eksekusi skrip terikat pada permintaan, dan komponen tidak dapat hidup di antara permintaan kecuali disimpan di tempat lain.

Bahasa yang mendukung paradigma pemrograman fungsional atau prosedural (selama Anda tidak menggunakan variabel global untuk status) dapat berjalan lama (jadi eksekusi aplikasi yang sama melayani beberapa permintaan), tetapi juga memaksa Anda untuk tidak menyimpan status di antara permintaan dalam aplikasi Anda. Terkadang boleh saja menyimpan status, mungkin sebagai cache cepat dalam memori, tetapi Anda tidak boleh mendesain sehingga satu-satunya tempat penyimpanan data adalah di tingkatan aplikasi Anda.

Saat mendesain dan mengimplementasikan aplikasi, sangat penting untuk mengidentifikasi jenis data apa yang perlu Anda simpan, dan di mana Anda akan menyimpannya. Memiliki pemahaman yang kuat tentang konsep-konsep ini memungkinkan Anda untuk bernalar melalui pencadangan dan keamanan, yang dibahas nanti dalam bab ini. Satu pendekatan naif untuk memisahkan database dari server aplikasi mungkin dapat dilakukan sebagai berikut: jika ada beberapa layanan aplikasi yang perlu mengakses data tersebut, maka layanan tersebut dapat terhubung ke database dan mengaksesnya secara langsung.

Pendekatan ini dapat menimbulkan masalah,Jika ada beberapa aplikasi yang mengakses database yang sama, maka hal ini dapat mempermudah masuknya data yang buruk ke dalam database, atau mempersulit koordinasi perubahan apa pun pada skema database. Meskipun banyak database SQL mendukung pengetikan yang kuat, prosedur tersimpan, dan aturan validasi, hal ini tidak berlaku untuk database NoSQL. Hal ini sering kali berarti bahwa ketika beberapa komponen perlu mengakses database tersebut, maka aturan validasi yang sama mungkin perlu diimplementasikan ulang beberapa kali, yang meningkatkan risiko bug

dan duplikasi konsep di beberapa tempat.

Sebaliknya, lebih baik untuk merangkum penyimpanan data aktual di balik layanan sederhana yang menyediakan akses ke lapisan tersebut. Layanan ini kemudian dapat difokuskan pada penerapan aturan validasi dan logika bisnis lain yang sesuai yang berkaitan dengan data, serta menjadi tempat di mana setiap pengoptimalan kinerja dapat difokuskan. Layanan ini kemudian dapat mengekspos API dengan cara yang konsisten dengan layanan lain yang mungkin Anda miliki (biasanya RESTful) dan tetap konsisten bahkan jika Anda mengubah sepenuhnya struktur data atau mesin penyimpanan yang mendasarinya. Bab Mendesain Sistem membahas hal ini lebih lanjut.

Dalam beberapa keadaan, penyimpanan data tunggal yang diakses melalui API mungkin tidak sesuai. Beberapa data sangat penting dan mendasar sehingga hampir setiap layanan perlu menyentuhnya, yang dapat menimbulkan satu titik kegagalan dalam arsitektur Anda. Skenario lain mungkin terjadi ketika Anda memiliki layanan lain yang harus menyadari adanya perubahan pada data tersebut misalnya, layanan yang bertanggung jawab untuk mengirimkan paket keanggotaan kepada pelanggan baru, atau layanan yang mungkin membatalkan kartu kredit jika akun pelanggan ditandai sebagai ditutup.

Dalam kasus sederhana, menambahkan antrean pesan ke API Anda untuk menerbitkan peristiwa dapat berguna. Cara alternatif untuk mengelola data dapat digunakan di sini, yang disebut "sumber peristiwa". Dengan model data sumber peristiwa, tidak ada satu titik kebenaran pun dalam sistem Anda.Sebaliknya, sumber peristiwa terdiri dari umpan peristiwa tempat setiap aplikasi yang peduli dengan data dapat membangun tampilan mereka sendiri terhadapnya. Pembuatan, pembaruan, dan penghapusan data masih terjadi melalui layanan yang bertanggung jawab untuk menerbitkan peristiwa dan validasi serta logika bisnis lainnya seputar pembuatan, tetapi logika bisnis apa pun seputar pembacaan data kemudian didelegasikan ke layanan individual.

Saat aplikasi menerima peristiwa, aplikasi tersebut kemudian memperbarui basis datanya sendiri, sering kali membuang informasi apa pun yang tidak diperlukan. Dalam model ini, masih diperlukan penyimpanan terpusat. Misalnya, jika aplikasi menyadari perlu mengumpulkan lebih banyak data daripada sebelumnya, bug menyebabkan pesan hilang, atau layanan baru muncul. Kemudian, pesan mungkin perlu diputar ulang, dan pesan itu sendiri harus direkam agar hal itu terjadi. Kerangka kerja sumber peristiwa seperti Apache Kafka mendukung fungsionalitas ini.

Cukup sering, pesan juga dapat disederhanakan sehingga, alih-alih memutar setiap pesan, hanya pesan yang diperlukan untuk mewakili status sistem saat ini yang diputar. Jadi mungkin produk yang ditambahkan lalu dihapus tidak pernah dikirim, atau hanya perlu dibuat satu pesan pembuatan, yang berisi nilai terbaru dari semua bidang, alih-alih satu pesan pembuatan dan sejumlah pesan pembaruan. Cara menyimpan data sebagai serangkaian peristiwa yang didistribusikan di seluruh aplikasi Anda dapat menimbulkan banyak kerumitan, dan sebaiknya hanya dipertimbangkan jika Anda memiliki masalah penskalaan khusus yang dapat dipecahkan oleh arsitektur semacam itu.

Sering kali, setiap bagian aplikasi Anda memiliki basis data internalnya sendiri yang

dibangun berdasarkan peristiwa tersebut. Tempat lain yang mungkin Anda inginkan untuk membangun basis data lokal yang berasal dari beberapa data eksternal adalah jika Anda perlu menyimpan beberapa data, baik sebagai hasil dari kueri yang mahal untuk alasan kinerja, atau untuk ketahanan. Pen cache an dibahas secara lebih rinci dalam bab Sistem, tetapi meskipun sering dianggap terpisah, pen-cache-an masih merupakan jenis basis data. Pen cache an sering diimplementasikan menggunakan penyimpanan kunci-nilai, dengan kunci yang berasal dari kueri atau sumber daya yang sedang di cache, dan nilainya menjadi badan mentah item dari sistem atau kueri eksternal yang telah di cache.

Beberapa penyimpanan nilai kunci, seperti Redis, mendukung penentuan waktu kedaluwarsa pada item tertentu agar item tersebut dapat digunakan dengan lebih mudah dalam lapisan caching. Yang lain, seperti Memcached, sangat berfokus pada kasus penggunaan caching dan mengoptimalkan kecepatan daripada keandalan, karena secara sengaja akan mengeluarkan item dari cache sebelum kedaluwarsa jika memori hampir habis fitur yang biasanya tidak Anda inginkan dari penyimpanan data!

Mengakses Data Dari Aplikasi Anda

Setelah Anda menyiapkan database dan menghubungkan server aplikasi ke database tersebut, Anda memerlukan cara untuk mengakses data tersebut. Untuk database relasional, bahasa yang disebut Bahasa Kueri Terstruktur (atau SQL, yang diucapkan sama seringnya sebagai “sequel” dan hanya sebagai inisial individual) telah menjadi standar de facto untuk menulis kueri dan memanipulasi database. Sayangnya, karena database individual menjadi lebih kompleks, SQL telah mengembangkan banyak “dialek” yang berbeda untuk memungkinkan akses ke beberapa fitur yang lebih canggih yang diadopsi oleh beberapa database yang tidak didukung oleh bahasa SQL awal.

Anda akan menemukan bahwa meskipun kueri SELECT, INSERT, dan UPDATE yang sederhana dapat ditransfer antar-server SQL, Anda akan sering menemukan kasus-kasus di mana Anda perlu menulis SQL yang khusus untuk database tertentu yang Anda miliki. Jangan takut dengan hal ini. Mungkin tampak menggoda untuk membatasi diri Anda pada SQL generik untuk memaksimalkan kemampuan Anda dalam menukar database yang berbeda, tetapi keadaan sebenarnya di mana Anda mungkin benar-benar ingin mengubah perangkat lunak database cukup jarang sehingga lebih masuk akal untuk menulis SQL untuk server tertentu yang Anda gunakan.

Lebih jauh lagi, membatasi diri Anda pada SQL generik akan sering kali menghalangi Anda untuk menggunakan fitur-fitur canggih dari database yang dapat membuat hidup Anda jauh lebih mudah. Database NoSQL, seperti namanya, tidak menggunakan SQL untuk berinteraksi dengan database. Sebaliknya, setiap database NoSQL akan sering memiliki cara penulisan kueri yang berbeda. Satu kelemahan dari interaksi SQL dan database adalah bahwa data yang disimpan dalam database tidak selalu memiliki tautan implisit ke tipe data atau struktur dalam kode aplikasi Anda.

Pustaka yang dikenal sebagai "pemetaan objek-relasional" atau ORM dapat menangani hal ini, karena pustaka tersebut akan menerjemahkan antara kueri dan hasil basis data, serta jenis dan objek yang Anda gunakan dalam kode aplikasi. Terlepas dari namanya, ada pustaka

jenis ORM untuk basis data non-relasional juga. Pustaka ORM yang berbeda menyediakan lapisan abstraksi yang berbeda antara kode Anda dan pustaka yang mendasarinya. Pustaka yang sederhana tetap mengharuskan Anda untuk menulis SQL dan cukup memberikan hasil sebagai objek dengan tipe yang Anda harapkan, sedangkan pustaka yang lain mungkin menyediakan metode seperti create(), update(), dan delete() pada objek Anda juga, yang kemudian juga menangani siklus hidup penuh dari rekaman tertentu dalam basis data Anda.

Mengakses objek yang ditautkan (dalam kasus basis data relasional) juga dapat dilakukan. ORM yang paling lengkap sering kali menyediakan serangkaian metode dan kelas untuk menangani pembuatan kueri guna mencari basis data, serta mengelola perubahan apa pun dalam struktur data aplikasi Anda. Ini dapat membantu dalam kasus yang sederhana, tetapi sering kali tidak dapat menangani skenario yang lebih kompleks, sehingga sering kali menyertakan fallback yang memungkinkan Anda untuk menulis SQL secara langsung.

ORM yang paling lengkap dapat cukup membatasi, terutama jika Anda sedang membangun aplikasi yang kompleks, jadi beberapa orang lebih suka menghindarinya. Salah satu alasan umum untuk melakukannya adalah bahwa cara ORM membangun kueri dapat menghasilkan kueri basis data yang jauh dari optimal dan memiliki masalah kinerja. Untuk aplikasi tipe Buat Baca Perbarui Hapus yang lebih sederhana, aplikasi ini dapat menghemat banyak waktu, jadi seperti banyak alat lainnya, kesesuaiannya bergantung pada konteks masalah khusus Anda.

Meskipun basis data berfungsi dengan baik dalam mengabstraksikan banyak kerumitan dalam mengelola kumpulan data besar dari Anda, basis data tidak dapat menyembunyikan semua kerumitan tersebut. Untuk basis data besar, melakukan kueri sederhana untuk mengambil data dapat menjadi sangat lambat jika basis data harus memeriksa setiap bidang untuk melihat apakah data tersebut valid. Sebagian besar basis data mendukung konsep indeks yang mengoptimalkan hal ini. Jika ada indeks, basis data dapat mencari indeks tersebut, bukan seluruh tabel, untuk menemukan data yang tepat.

Sebagian besar basis data memiliki indeks yang disebut kunci utama, yang merupakan cara untuk mencari satu rekaman dengan ID utamanya dengan cepat. Namun, katakanlah Anda ingin memfilter seluruh basis data Anda dengan bidang yang disebut "kategori" maka, jika Anda tidak memiliki indeks pada bidang kategori, saat Anda menerapkan filter tersebut, basis data harus melihat setiap rekaman untuk memeriksa nilai bidang kategori. Namun, jika Anda telah mengindeks bidang kategori, maka pada awalnya indeks (yang jauh lebih kecil) akan dilihat untuk mendapatkan daftar rekaman yang akan diambil, dan kemudian hanya rekaman tersebut yang diambil, yang dapat menghemat banyak waktu untuk basis data yang lebih besar.m

Untuk kueri yang lebih rumit, mendapatkan indeks yang tepat bisa jadi sulit dilakukan. Untungnya, ada beberapa alat yang bagus yang tersedia untuk membantu Anda menjalankan kueri. Basis data SQL mendukung perintah seperti ANALYZE, dan basis data lain memiliki perintah serupa yang memberi tahu Anda jika kueri tertentu menggunakan indeks atau kembali ke kueri yang tidak dioptimalkan, yang dapat memperlambat. Mungkin tampak bijaksana untuk sekadar menambahkan indeks ke semuanya, tetapi ini kontraproduktif.

Indeks memperlambat penyisipan dan pembaruan, karena semua indeks juga perlu diperbarui untuk setiap operasi, jadi penting untuk hanya memiliki indeks pada tabel yang benar-benar Anda gunakan. Aturan praktis yang baik adalah melihat bagian WHERE dari pernyataan SQL (atau yang setara untuk basis data lain) dan memastikan bahwa untuk setiap jenis kueri yang Anda miliki, ada indeks pada setiap kombinasi kolom yang Anda gunakan.

Penting untuk menyadari bahwa ada juga berbagai jenis indeks. Merinci semua ini berada di luar cakupan buku ini, tetapi jika Anda membuat kueri di luar persamaan atau perbandingan sederhana, maka Anda mungkin ingin mencari tahu jenis indeks apa yang ada dalam basis data yang Anda gunakan. Misalnya, indeks geospasial umum digunakan, yang memungkinkan Anda melakukan penelusuran cepat berdasarkan koordinat geografis (misalnya, tunjukkan semua toko dalam jarak 30 km dari lokasi pengguna saat ini), serta indeks penelusuran teks lengkap (untuk kueri seperti, tunjukkan semua kiriman yang berisi kata "Barack Obama").

11.3 MENGELOLA DATA ANDA

Basis data Anda memiliki satu perbedaan penting dari aplikasi Anda yang lain basis data akan sering kali hidup bersama dan berkembang bersama kode Anda untuk waktu yang lama. Dengan kode biasa, yang perlu Anda lakukan jika Anda mengubahnya adalah menyebarkannya, dan kode baru Anda akan digunakan. Namun, jika Anda ingin mengubah cara data Anda terstruktur, hal itu memerlukan manajemen yang jauh lebih cermat. Sudah lama menjadi praktik umum untuk memiliki satu basis data besar, yang dikelola oleh tim pusat yang bertanggung jawab untuk memelihara dan mengoptimalkannya.

Aplikasi Anda kemudian terhubung ke basis data tersebut, dan Anda mendiskusikan persyaratan Anda dengan tim yang menyiapkannya dan membantu merancang skema Anda. Nama yang sering digunakan untuk peran ini adalah "administrator basis data," atau "DBA," dan tim tersebut sering disebut "tim DBA" atau yang serupa. Namun, dengan layanan mikro, kini sering kali terdapat banyak basis data yang lebih kecil di seluruh sistem, dan meskipun tim yang bertanggung jawab atas basis data yang sangat penting mungkin memiliki seseorang dengan keterampilan DBA di tim mereka, hal ini tidak selalu terjadi.

Meskipun peran khas DBA mencakup banyak hal (termasuk saran dalam menyiapkan skema dan indeks yang sesuai, seperti yang dibahas di bagian sebelumnya, dan pencadangan dan enkripsi, seperti yang dibahas di bagian berikutnya), satu peran yang dapat dilakukan DBA adalah mengelola kinerja basis data. Seperti yang dibahas dalam bab Dalam Produksi, metrik aplikasi dapat digunakan untuk menyiapkan peringatan bagi aplikasi Anda, tetapi untuk basis data, ada banyak metrik yang berkaitan dengan kecepatan kueri juga, dan Anda harus memantaunya untuk mengidentifikasi masalah kinerja apa pun dengan basis data Anda.

Akan tiba saatnya ketika struktur data Anda perlu diubah, baik untuk memperbaiki masalah dengan struktur yang ada, menambahkan kolom baru yang perlu diambil untuk mendukung fitur baru, atau mengatasi masalah kinerja. Perubahan struktural semacam ini sering kali sulit, karena Anda perlu mencari tahu cara menerjemahkan data yang ada ke dalam struktur baru, terutama jika Anda menambahkan kolom wajib baru. Ini dapat melibatkan

perubahan pada kode aplikasi Anda juga, jadi jika kolom wajib baru ditambahkan, pengguna mungkin diminta untuk memperbarui profil mereka sebelum melanjutkan, tetapi membiarkan beberapa profil tidak lengkap jika mereka jarang masuk.

Dalam basis data relasional, proses pengelolaan perubahan struktural disebut migrasi, dan sebagian besar bahasa memiliki kerangka kerja untuk membantu mengelola migrasi basis data. Kerangka kerja yang paling efisien dari ini, yang mencakup alat mandiri seperti Flyway serta beberapa yang terintegrasi ke dalam kerangka kerja seperti Django, akan menyimpan migrasi ini sebagai kode di samping aplikasi utama Anda, dan dapat mengelola menjalankan migrasi ini sebagai bagian dari penerapan aplikasi Anda. Berbeda dengan basis data relasional, salah satu keuntungan dari basis data NoSQL adalah Anda tidak perlu memigrasikan semua data sekaligus, dan berbagai bit data dapat hidup berdampingan dalam satu basis data dengan struktur yang berbeda, mungkin tergantung pada kapan rekaman tersebut awalnya dibuat.

Dalam basis data NoSQL, Anda dapat menulis kode aplikasi yang memutakhirkan model ke versi terbaru skema setiap kali diakses, daripada memigrasikan semua rekaman sekaligus (yang, untuk basis data besar, dapat melibatkan waktu henti jika ada waktu pemrosesan yang lama). Menambahkan versi ke dokumen dalam basis data NoSQL dapat membantu di sini. Kebutuhan migrasi dapat bergantung pada jenis data yang Anda simpan. Misalnya, dalam penyimpanan sesi, data tidak selalu bertahan lama, dan mungkin tidak masalah untuk sekadar mengatur ulang semua sesi dan beralih ke versi terbaru skema sesi.

Atau, untuk kasus sederhana seperti menambahkan bidang baru, Anda dapat menjalankan dua skema secara paralel, dan membuat kode aplikasi Anda toleran terhadap fakta bahwa beberapa bidang mungkin hilang dan menganggapnya dalam status default. Satu kelemahan migrasi adalah migrasi dapat mempersulit pengembalian rilis kode aplikasi, dan sebagai hasilnya, migrasi terbalik (atau "turun") juga umum dilakukan, yang akan mengembalikan hasil migrasi untuk mengembalikan semuanya ke struktur sebelumnya.

Menerapkan migrasi tanpa menyebabkan waktu henti juga merupakan tantangan. Pendekatan dasar mungkin mengharuskan Anda membuat versi baru kode yang menggunakan skema baru, kemudian, untuk menerapkan, menempatkan aplikasi ke mode pemeliharaan dan mematikan kode yang ada. Kemudian, Anda akan menerapkan migrasi basis data dan memunculkan versi baru kode aplikasi. Untuk sistem kritis, waktu henti semacam ini mungkin tidak dapat diterima, jadi sering kali umum untuk memiliki rilis kode sementara yang mendukung skema baru dan lama, lalu melakukan rilis kedua yang menghapus dukungan untuk skema lama setelah migrasi selesai.

Dalam kasus perubahan yang sederhana, seperti menambahkan kolom baru, ini dapat dilewati sepenuhnya, selama aplikasi Anda toleran terhadap munculnya kolom baru di basis data Anda, dan mengabaikan kolom yang tidak terduga. Terlepas dari apakah Anda menggunakan database relasional atau NoSQL, penting untuk mempertimbangkan migrasi data dan skema bersamaan dengan penerapan kode. Dalam bab sebelumnya, kita melihat bagaimana teknik seperti integrasi berkelanjutan, pengiriman berkelanjutan, dan penerapan berkelanjutan dapat digunakan untuk membantu meminimalkan waktu yang dibutuhkan untuk memberikan perubahan pada sistem yang aktif.

Jika sistem Anda harus mengelola data, Anda harus mempertimbangkan untuk menggabungkan migrasi Anda ke dalam proses penerapan ini, baik dengan cara yang sepenuhnya otomatis, atau mungkin dengan tingkat intervensi manual, tergantung pada keadaan Anda.

11.4 MELINDUNGI DATA ANDA

Tidak mengherankan jika Anda harus mencadangkan basis data Anda. Namun, menjalankan jenis pencadangan yang tepat bisa jadi sulit. Jenis pencadangan yang paling sederhana melindungi dari kegagalan teknis, seperti kerusakan atau kerusakan hard drive, tetapi tidak selalu melindungi dari jenis kesalahan lain, seperti kesalahan operator atau bug yang menyebabkan kerusakan data. Pencadangan sederhana dapat melibatkan eksekusi dump basis data Anda pada pengatur waktu (mungkin menggunakan alat seperti mysqldump dengan penjadwal seperti cron) dan menyalin file yang dihasilkan ke server lain.

Namun, hal ini menyisakan periode waktu setelah pencadangan terakhir di mana beberapa transaksi akan hilang jika terjadi kerusakan. Apakah hal ini dapat diterima atau tidak sepenuhnya bergantung pada jenis data yang Anda simpan. Strategi pencadangan alternatif melibatkan pengaturan replikasi. Saat menggunakan replikasi, server basis data kedua (dulu dikenal sebagai slave, tetapi baru-baru ini sebagai replika) berjalan, memiliki salinan basis datanya sendiri dan terhubung ke server pertama (dulu disebut master, tetapi baru-baru ini disebut sebagai server utama), lalu membuat perubahan apa pun pada salinannya sendiri yang dibuat oleh server utama. Ini berarti replika terus mengikuti perkembangan server utama dan biasanya hanya tertinggal satu atau dua detik dari status server utama.

Sering kali replika ini kemudian dapat dialihkan menjadi server utama jika terjadi kegagalan pada server utama, sehingga meminimalkan waktu henti sistem Anda. Sistem pencadangan seperti ini memiliki kekurangan, yaitu kesalahan apa pun akan cepat menyebar ke replika, jadi bug, kesalahan, atau bahkan peretasan yang dapat merusak basis data tidak akan mudah dipulihkan setelah perubahan direplikasi. Rezim pencadangan yang paling efektif menggunakan kombinasi dari dua strategi di atas, menggunakan replikasi untuk menyediakan pemulihan jika terjadi kegagalan teknis, lalu dump basis data (yang disimpan selama jangka waktu tertentu) untuk menutupi jenis kerusakan data lainnya.

Seperti pepatah lama, pencadangan yang belum teruji sering kali sama baiknya dengan tidak mencadangkan sama sekali. Pengujian rutin rezim pencadangan Anda, dan memastikan Anda dapat menggunakan pencadangan untuk mengembalikan sistem ke kondisi kerja seperti yang diharapkan, sama pentingnya dengan menyiapkan sistem pencadangan Anda sejak awal. Latihan kebakaran (dibahas lebih lanjut dalam bab Dalam Produksi) merupakan cara yang efektif untuk menguji ini. Sering kali, pengujian rutin sistem pencadangan Anda melibatkan penggunaan pencadangan untuk mengisi lingkungan pengembangan jika tidak ada data sensitif di dalamnya.

Lebih jauh lagi, menyiapkan pemantauan cadangan Anda (misalnya, memeriksa apakah ukuran file berada dalam kisaran yang diharapkan, atau tanggal pada file terbaru tidak terlalu jauh) dapat memberi Anda tingkat perlindungan tambahan. Melindungi data Anda tidak

berarti hanya memastikannya aman dari kehilangan, tetapi juga aman dari kebocoran dan penyebaran yang terlalu jauh. Enkripsi adalah praktik yang dipahami dengan baik, dan banyak basis data mendukungnya secara bawaan. Saat menggabungkan enkripsi dengan cadangan, Anda juga harus memastikan cadangan Anda dienkripsi dan dilindungi dengan tepat, karena mencuri cadangan sama saja dengan mencuri salinan basis data aktif, dan telah menjadi penyebab banyak insiden keamanan.

Untuk kasus sensitif, ini dapat berarti mekanisme pengujian cadangan Anda dengan memulihkan data produksi ke lingkungan pengembangan Anda tidak tepat, karena data sensitif Anda seharusnya sering dibatasi hanya untuk mereka yang memerlukan akses ke sana, yang mungkin bukan seluruh tim pengembang. Ada dua jenis enkripsi utama yang dapat digunakan saat mempertimbangkan data: enkripsi saat transit, dan enkripsi saat tidak aktif. Enkripsi saat transit mengacu pada cara data Anda bergerak antara server aplikasi dan server basis data, atau server lain yang mungkin terlibat dalam komunikasi, dan Transport Layer Security telah menjadi pendekatan yang dominan di sini, dengan sebagian besar server basis data mendukung koneksi TLS.

Enkripsi saat tidak aktif mengacu pada cara basis data Anda disimpan—jika seseorang entah bagaimana memperoleh akses ke hard disk tempat data Anda disimpan (mungkin dengan mencurinya, atau meretas server tempat perangkat lunak tersebut berjalan), dapatkah mereka begitu saja membaca data tersebut tanpa melewati keamanan apa pun dalam perangkat lunak basis data Anda? Beberapa perangkat lunak basis data juga mendukung gagasan enkripsi saat tidak digunakan, sehingga file itu sendiri dienkripsi, dan ini dapat efektif untuk semua data kecuali yang paling sensitif.

Namun, ini berarti bahwa siapa pun yang dapat terhubung ke instans server basis data yang sedang berjalan (mungkin dengan menggunakan rekayasa sosial untuk mencuri nama pengguna dan kata sandi admin) dapat menjalankan kueri yang sesuai dan melihat data yang sebenarnya. Anda dapat melangkah lebih jauh dan benar-benar mengenkripsi data dalam aplikasi Anda, sehingga basis data tidak akan pernah melihat data yang tidak dienkripsi. Ini berarti bahwa akses ke basis data saja tidak cukup pengguna juga memerlukan akses ke kunci enkripsi yang digunakan aplikasi.

Ini mungkin masih dapat dicapai oleh peretas yang bertekad meskipun pada akhirnya perlu ada cara untuk mendekripsi data agar aplikasi dapat menggunakannya tetapi dengan meningkatkan lapisan perlindungan, Anda dapat mencegah pelanggaran keamanan yang paling umum. Satu kelemahan enkripsi basis data adalah bahwa beberapa bentuk pengindeksan dan kueri pada bidang terenkripsi tidak lagi memungkinkan.

11.5 KESIMPULAN

Sebagian besar situs web perlu menyimpan data di suatu tempat dalam sistem yang lebih luas, dan basis data digunakan untuk mencapainya. Basis data sulit dikelola, karena berisi status, dan mungkin perlu berevolusi dalam struktur bersama aplikasi Anda, tanpa menggunakan data di sepanjang prosesnya. Data dalam basis data Anda juga harus dikelola dengan hati-hati, karena dalam banyak kasus ada konsekuensi hukum, keuangan, atau etika

bagi Anda dan organisasi Anda jika data tersebut bocor secara tidak tepat.

Basis data sering dikategorikan sebagai relasional seperti MySQL, MariaDB, dan PostgreSQL-atau NoSQL-seperti Redis, MongoDB, atau CouchDB. Tidak seperti relasional, NoSQL bukanlah satu jenis basis data, dan mencakup sejumlah pendekatan berbeda untuk menyimpan data, termasuk nilai kunci, berbasis dokumen, grafik, deret waktu, atau kolom. Salah satu perbedaan utama adalah cara Anda berpindah di antara versi struktur basis data Anda. Dengan relasional, migrasi digunakan untuk mengubah data dan struktur, dan dengan NoSQL, ini sering dikelola dalam kode, dengan kemungkinan beberapa versi struktur yang berbeda hadir sekaligus, tergantung pada kunci mana yang sedang dilihat. Arsitektur sistem harus mengidentifikasi tempat penyimpanan data di aplikasi Anda, dan harus berhati-hati agar tidak secara tidak sengaja memasukkan penyimpanan lain sebagai bagian dari aplikasi Anda, dengan memastikan semua status berada dalam basis data yang sesuai.

Terkadang basis data lokal dapat digunakan, mungkin sebagai tingkatan penyimpanan sementara atau dalam model sumber peristiwa, untuk membangun tampilan lokal data saat model tersebut digunakan. Aplikasi juga perlu mengakses data. Dalam basis data relasional, bahasa yang dikenal sebagai SQL digunakan, dan beberapa kerangka kerja yang dikenal sebagai pemeta objek-relasional (ORM) dapat membantu membangun kueri dan memetakannya ke objek dalam bahasa pilihan Anda. Basis data NoSQL tidak menggunakan SQL, dan mekanisme lain dapat digunakan untuk mengkuerinya tergantung pada basis data yang tepat. Indeks berdasarkan bidang tertentu dapat digunakan untuk membantu mempercepat kueri.

Data dalam basis data Anda harus dilindungi, dari kehilangan atau kerusakan yang tidak disengaja, serta dari serangan yang disengaja. Cadangan dapat digunakan untuk melindungi dari yang pertama, dan teknik termasuk enkripsi dapat membantu yang terakhir. Ingatlah bahwa cadangan Anda juga harus dilindungi menggunakan teknik seperti enkripsi jika tidak, Anda berisiko meniadakan tingkat perlindungan awal itu!

BAB 12
KEAMANAN

12.1 MEMBANGUN SISTEM AMAN YANG DAPAT DIGUNAKAN

Membangun sistem yang aman merupakan kewajiban hukum dan etika bagi pengembang perangkat lunak, tetapi dunia keamanan komputer dan informasi pada awalnya tampak seperti rintangan yang menakutkan bagi seorang pemula, dengan komunitas yang sering kali tampak seperti kelompok dan berfokus pada hal-hal yang bersifat hipotetis. Kenyataannya adalah bahwa sistem yang paling aman adalah sistem yang terputus dari jaringan dan dimatikan, tetapi sistem ini juga tidak dapat digunakan. Membangun sistem yang aman tetapi dapat digunakan merupakan tantangan yang hanya dapat diatasi melalui kompromi.

Seperti aksesibilitas, keamanan pada awalnya mungkin tampak seperti memerlukan keterampilan khusus untuk ditangani sepenuhnya. Namun, untuk sebagian besar aplikasi di luar sana, pengetahuan khusus dari seorang pakar keamanan tidak diperlukan, karena Anda dapat menggunakan pustaka dan pola yang ada untuk membangun aplikasi web yang aman. Di perusahaan yang lebih besar, tim keamanan khusus adalah hal yang umum, tetapi dapat dilihat sebagai hambatan untuk pengiriman.

Jika Anda dapat berbicara dengan percaya diri tentang aplikasi dan membangun teknik yang sederhana dan efektif sejak awal, hambatan ini dapat dihilangkan dan dibangun ke dalam proses pengiriman Anda, menghasilkan aplikasi yang dapat digunakan yang secara efektif aman. Sebagian besar dari kita tidak membangun sistem komputer bergaya Mission Impossible, dan meskipun ada beberapa bagian infrastruktur penting yang memerlukan pengetahuan lebih besar daripada yang dimiliki seorang generalis, sebagian besar pengembang web tidak akan pernah menemukannya. Bahkan bagi para pengembang yang membangun sistem SCADA untuk reaktor nuklir, konsep-konsep ini merupakan awal yang baik. Dengan mempelajari beberapa prinsip utama dan mengidentifikasi keterbatasan dalam pengetahuan Anda, Anda dapat membangun sistem yang memenuhi persyaratan keamanan Anda.

12.2 KEPERCAYAAN DAN RAHASIA

Selama membangun aplikasi Anda, penting untuk dapat mengenali saat Anda bekerja dengan sesuatu yang mungkin merupakan komponen yang sensitif terhadap keamanan. Sebagai seorang generalis, jika Anda mendapati diri Anda bekerja secara langsung dengan pemrosesan kartu kredit, kata sandi, atau sistem enkripsi/dekripsi, Anda berada di zona bahaya. Membangun sistem semacam ini dari awal tidak boleh dilakukan dengan mudah, dan sering kali merupakan hal yang salah untuk dilakukan.

Ada banyak pustaka dan kerangka kerja untuk tugas-tugas umum ini, jadi ketika Anda berhadapan dengan fungsionalitas seperti ini, menggunakan pustaka yang bereputasi baik (reputasi dapat ditentukan oleh sejumlah faktor, seperti aktivitas repositori GitHub, atau

jumlah pertanyaan StackOverflow tentang kerangka kerja, tetapi ini jauh dari jaminan) akan membuat pekerjaan Anda jauh lebih mudah.

Keamanan melalui ketidakjelasan berarti Anda mengandalkan fakta bahwa penyerang potensial tidak mengetahui sedikit informasi yang diperlukan untuk mengakses bagian sistem Anda yang aman. Misalnya, memiliki URL tersembunyi untuk bagian admin dianggap sebagai keamanan melalui ketidakjelasan. Mengandalkan keamanan melalui ketidakjelasan sangat berbahaya, tetapi ada perbedaan antara ketidakjelasan dan rahasia. Sesuatu yang rahasia seharusnya sangat sulit ditebak, sedangkan sesuatu yang tidak jelas dapat ditebak atau diketahui.

Sebagian besar sistem mengandalkan beberapa pengetahuan yang disembunyikan dari penyerang potensial kunci API, kata sandi, atau kunci SSH tetapi pengetahuan semacam ini dapat dianggap sebagai "rahasia" daripada sekadar ketidakjelasan. Rahasia harus dipisahkan dari kode sumber Anda. Di banyak organisasi, akses ke kode sumber dapat tersebar luas—mungkin setiap orang memiliki akses ke kontrol sumber dan dapat memeriksa salinan kode sumber Anda yang hanya dapat dibaca, atau Anda mengundang kontraktor untuk membantu beberapa fungsi yang rumit. Menyimpan rahasia di samping kode dapat memungkinkannya menyebar lebih luas dari yang dimaksudkan.

Ini juga berarti bahwa bukan kode itu sendiri yang perlu dirahasiakan; jika informasi yang diperlukan untuk menghindari keamanan dapat ditemukan dalam kode Anda, maka ini adalah keamanan melalui ketidakjelasan. Rahasia harus ditangani seperti limbah nuklir dengan hati-hati, dan dengan cara yang terkendali. Orang-orang yang memiliki akses ke rahasia harus dikunci berdasarkan kebutuhan, dan Anda juga harus mengubah rahasia ini secara berkala, terutama jika seseorang meninggalkan organisasi (ini terkadang disebut sebagai rotasi kunci).

Ada banyak cara untuk mengelola rahasia. Beberapa alat penyebaran memungkinkan Anda untuk mengirimkan nilai konfigurasi rahasia (jadi setelah konfigurasi ditetapkan, itu tidak dapat dibaca oleh pengguna), yang terkadang sesuai, tetapi metode lain termasuk memiliki file konfigurasi terpisah yang berisi rahasia, yang disimpan di luar repositori utama (seperti hanya di server langsung atau dalam drive bersama yang dienkripsi), dan kemudian dapat disebarkan dari mesin tepercaya.

Pilihan lain adalah memiliki layanan yang bertanggung jawab untuk mengelola rahasia yang dapat dihubungkan dengan aplikasi lain, dan setelah diautentikasi, dapat memuat rahasia tersebut dari layanan itu. Beberapa rahasia dapat dianggap kurang rahasia dibandingkan yang lain. Misalnya, kata sandi default untuk lingkungan pengembang yang hanya dapat diakses melalui localhost mungkin aman untuk disimpan dalam basis kode demi kecepatan menyiapkan lingkungan baru, selama basis data hanya berisi data uji atau data tiruan.

Menanggapi Insiden

Pelanggaran keamanan dapat menjadi salah satu peristiwa dengan dampak tertinggi yang terjadi pada suatu organisasi, tetapi mustahil untuk mengetahui dampak pelanggaran tersebut tanpa penyelidikan menyeluruh. Di Inggris, dan di banyak negara lain, organisasi memiliki tanggung jawab hukum untuk memberi tahu setiap pengguna yang terdampak tentang pelanggaran data, terlepas dari kebijakan internal dan kewajiban etika untuk

mengidentifikasi pelanggaran dan dampaknya secara menyeluruh.

Seperti banyak hal dalam hidup, pendekatan terbaik di sini adalah berharap yang terbaik, tetapi bersiap untuk yang terburuk. Penting untuk memastikan Anda memiliki alat yang memungkinkan Anda merekonstruksi sebaik mungkin tindakan yang diambil penyerang, untuk memahami dampaknya. Untuk sebagian besar sistem, ini berarti pencatatan audit yang komprehensif. Awalnya mungkin tergoda untuk menganggap jenis pencatatan ini mirip dengan pencatatan aplikasi biasa, tetapi ada baiknya untuk memisahkan log audit ke dalam berkasnya sendiri, karena aktivitas ini dapat menyumbat log aplikasi saat men-debug kesalahan umum.

Ada beberapa alat yang dapat membantu pencatatan audit yang menyediakan fitur anti-rusak (sehingga log tidak dapat diedit setelah ditulis), yang memberikan tingkat kepercayaan yang tinggi pada log Anda. Biasanya merupakan ide yang baik untuk merekam log audit dari infrastruktur yang Anda gunakan, agar lebih sulit dihapus atau dirusak. Konten log audit yang tepat akan bervariasi berdasarkan aplikasi Anda, tetapi Anda biasanya ingin merekam tindakan seperti mengakses atau mengedit data sensitif. Ini dapat mencakup tanggal/waktu, pengguna, alamat IP mereka, tindakan yang dilakukan, dan informasi terperinci tentang apa yang diubah.

Pencatatan saat akses ditolak juga dapat berguna, karena ini menunjukkan setiap upaya yang gagal untuk menyusup ke sistem dan memberi Anda gambaran tentang sejauh mana penyerang berhasil. Log akses penuh dari server web Anda dapat digunakan untuk mengidentifikasi aktivitas mencurigakan lainnya yang dilakukan penyerang. Beberapa pelanggaran terjadi akibat kata sandi yang lemah atau dicuri dari pengguna yang memiliki hak istimewa.

Sering kali hal ini terjadi menggunakan rekayasa sosial, yang sulit untuk dilawan, tetapi ada tindakan pencegahan teknologi yang dapat dilakukan, seperti menerapkan kata sandi yang panjang dan menerapkan autentikasi dua faktor. Banyak pelanggaran terjadi melalui eksploitasi bug dalam aplikasi, dan melihat log aplikasi bersama dengan log audit dapat memberikan pencerahan. Setiap pengecualian atau kesalahan yang ganjil atau tidak terduga dari database (misalnya, kueri SQL yang salah bentuk) dapat membantu menemukan kerentanan. Setelah penyelidikan selesai, hal terpenting yang harus dilakukan adalah memperbaiki kelemahan tersebut.

Ini dapat melibatkan penangguhan akun pengguna yang disusupi, atau mendorong perbaikan kode untuk memperbaiki bug. Dengan dampak serangan yang dipahami, organisasi harus dapat merespons dengan tepat. Meskipun mungkin tergoda untuk meremehkan serangan, kejujuran adalah salah satu mekanisme terbaik untuk menjaga kepercayaan dalam suatu organisasi, daripada mengambil risiko merilis detail secara lambat, yang menunjukkan peningkatan dampak dari apa yang awalnya diumumkan. Demikian pula, ketidakjelasan dalam pengumuman apa pun dapat meningkatkan ketidakpastian. Kejujuran dan hal-hal spesifik adalah kebijakan terbaik di sini.

Aturan Emas

Saat membangun aplikasi web, ada aturan emas yang harus diikuti: validasi pada input, bersihkan pada output. Saat menerima input dari pengguna, atau sistem eksternal apa pun,

sangat penting untuk memastikan bahwa input tersebut kira-kira seperti yang Anda harapkan ini adalah validasi. Secara umum, Anda tidak boleh mencoba membersihkan data yang telah dikirimkan pengguna kecuali dalam keadaan yang sangat terbatas (seperti menghapus spasi awal/akhir). Ini juga menghasilkan UX yang baik.

Sebagian besar kerangka kerja menawarkan pustaka yang membantu Anda melakukan ini, dan jika Anda mendapatkan sesuatu yang gagal divalidasi, Anda harus segera membatalkannya dan menunjukkannya kembali kepada pengguna sebagai kesalahan yang harus diperbaiki daripada mencoba membersihkannya dengan cerdik.

12.3 EKSPRESI REGULER

Ekspresi reguler (regex) adalah mekanisme populer untuk menulis aturan validasi, tetapi Anda harus berhati-hati! Misalnya, jika Anda menentukan bahwa ID harus berupa alfanumerik huruf kecil, maka Anda mungkin tergoda untuk menulis regex dalam bentuk /[a-z0-9_]+/. Ini memiliki efek samping yang tidak diinginkan yaitu mencocokkan di mana pun urutan itu muncul dalam string, bukan seluruh string. Jika seseorang kemudian mengirimkan string dalam bentuk valid_slug But This Bit Is Invalid", regex akan cocok dengannya, karena akan cocok dengan valid_slug, tetapi itu tidak berarti seluruh string cocok dengan regex.

Anchor dapat digunakan untuk memastikan bahwa regex cocok dengan seluruh string. ^ dan $ adalah karakter khusus yang masing-masing menunjukkan awal dan akhir string, jadi Anda dapat menulis regex dalam bentuk /^[a-z0-9_]+$ untuk mencocokkan contoh sebelumnya. Namun, perhatikan bahwa ^ dan $ berarti "awal" dan "akhir" baris, jadi mungkin berarti ini akan cocok dengan string seperti valid_slug\nBut This Bit Is Invalid, karena satu baris cocok. Sebagian besar mesin regex akan memperlakukan seluruh input sebagai satu baris, bahkan jika berisi baris baru, kecuali dimasukkan ke dalam mode multi-baris, jadi akan berfungsi seperti yang Anda harapkan.

Sayangnya, ini semua spesifik implementasi, jadi ada baiknya membaca dokumentasi untuk bahasa Anda sebelum membuat asumsi. Beberapa bahasa dan pustaka (terutama Ruby) beroperasi dalam mode multi-baris secara default, dan jangkar \A dan \z harus digunakan sebagai ganti ^ dan $ untuk menunjukkan awal-string dan akhir-string, meskipun tidak didukung oleh semua bahasa. Anda juga harus melakukan validasi di satu tempat.

Ini bisa di pengontrol, atau di kelas model tertentu yang disiapkan untuk menangani data pengguna. Jika Anda membatasi area kode yang bertanggung jawab untuk menangani masukan pengguna ke satu tempat itu, kecil kemungkinan Anda akan membuat kesalahan (lupa memvalidasi satu hal dari masukan akan terlihat jelas dalam kode), dan berarti area kode lainnya terisolasi dari masukan pengguna langsung dan dapat mengasumsikan bahwa data yang ditanganinya bersih atau, jika pemrograman defensif penuh digunakan, mengurangi dampak validasi yang terlewat di titik lain dalam tumpukan.

Penting untuk menerapkan validasi ini dalam situasi di mana sistem eksternal mana pun dapat mengirimkan data, bukan hanya pengguna. Sekadar memvalidasi dalam JavaScript pada klien dan mengirimkannya ke layanan kedua tidaklah cukup layanan itu juga harus memvalidasi apa yang diterimanya, karena pengguna jahat dapat melewati langkah JavaScript

dan membuat permintaan ke server secara langsung.

Ini adalah aturan yang sering diabaikan, karena hanya jalur interaksi utama yang dipertimbangkan, mengabaikan titik lain tempat data berpotensi disuntikkan. Sanitasi, di sisi lain, mengacu pada upaya memastikan bahwa data apa pun yang ditampilkan kepada pengguna atau dikirimkan ke sistem lain tidak merusak struktur pesan.

Phreaking

Kebutuhan akan sanitasi merupakan hasil dari pencampuran data pengguna dengan data "kontrol" dalam sintaksis seperti JSON atau HTML. Jika data kontrol dan konten dapat dipisahkan sepenuhnya (misalnya, dalam file yang berbeda), maka sanitasi tidak diperlukan, dan ini akan menghilangkan seluruh kelas kesalahan (meskipun akan membuat data lebih sulit ditangani dalam konteks lain). Pencampuran data kontrol dan pengguna ke dalam satu saluran adalah yang menyebabkan "peretasan" paling awal, di mana peretas telepon (phreaker) menemukan frekuensi tertentu yang, dikirim melalui saluran telepon melalui koneksi jarak jauh, akan memungkinkan mereka untuk meniru pertukaran telepon, sehingga pertukaran jarak jauh akan mempercayai perintah yang mereka berikan seolah-olah mereka adalah pertukaran lain.

Ini diperbaiki dengan memfilter frekuensi di pertukaran telepon awal tersebut, dan pertukaran telepon modern tidak mencampur data suara pengguna dengan data kontrol yang digunakan di antara pertukaran telepon. Saat menampilkan konten kepada pengguna yang berasal dari sumber eksternal (misalnya, pesan dari pengguna lain), pesan tersebut mungkin menyertakan karakter khusus HTML, yang dapat mengakibatkan halaman rusak dengan markup yang buruk.

Dalam kasus yang paling sederhana, hal ini dapat merusak desain; dalam kasus terburuk, hal ini dapat memungkinkan JavaScript mencuri kuki pengguna Anda. Untungnya, sebagian besar pustaka templating akan mengonversi HTML apa pun dalam variabel menjadi padanannya yang aman secara default (misalnya, akan dikonversi menjadi <strong>) yang akan menjalankan perintah yang benar proses ini disebut escape. Mungkin Anda tergoda untuk sekadar menghapus karakter "khusus" apa pun dari output, tetapi karakter khusus ini dapat dimasukkan secara sah oleh pengguna, dan terkadang mudah terlewatkan.

Konversi semacam ini harus selalu dilakukan oleh pustaka atau kerangka kerja yang bereputasi baik; mencoba membuat sendiri merupakan strategi berisiko tinggi. Sebaiknya sebutkan kasus-kasus di mana beberapa HTML valid—misalnya, Anda mungkin ingin mengizinkan pengguna Anda untuk menggunakan markup terbatas di forum atau sistem komentar. Dalam kasus ini, pustaka pembersih HTML khusus dapat digunakan, yang akan mengizinkan beberapa tag tetapi menghapus atau menolak yang lain.

Sering kali membantu untuk menjalankan ini ketika pengguna mengirimkan data juga, sebagai langkah validasi, untuk memberi mereka pesan kesalahan yang sesuai dan menolak tag yang tidak valid. Seperti halnya pembersih lainnya, membuat sendiri tidak disarankan. Sebaliknya, Anda harus mempercayai pustaka yang andal. Selain itu, ada dua jenis pendekatan yang dapat diambil oleh pembersih: daftar yang diizinkan (kadang-kadang disebut daftar

putih), dan pemblokiran (kadang-kadang disebut daftar hitam). Daftar yang diizinkan hanya mengizinkan tag HTML tertentu untuk digunakan, sedangkan pemblokiran memberlakukan daftar tag yang dilarang.

Lebih baik menggunakan mekanisme daftar yang diizinkan, karena evolusi pada spesifikasi HTML dapat memperkenalkan tag baru yang tidak diinginkan, dan akan memerlukan pembaruan daftar blokir untuk melarangnya. Tag juga mudah terlewatkan saat mengambil pendekatan daftar blokir. Seperti halnya validasi, sanitasi tidak hanya berlaku saat berhadapan dengan menampilkan informasi kepada pengguna.

Semua data yang keluar dari sistem harus disanitasi. Pola anti-umum terjadi saat permintaan ke sistem eksternal (seperti basis data, atau permintaan JSON ke server lain) dibuat dengan menggabungkan sendiri string-string. Permintaan semacam ini harus selalu disiapkan oleh pustaka yang sesuai, seperti serializer JSON, atau pustaka SQL, daripada langsung memasukkan string. Pustaka-pustaka yang mendasari ini akan menangani sanitasi untuk Anda dan melewati kelas kesalahan umum ini.

Pengecualian utama untuk hal ini adalah saat memanggil program baris perintah, yang sering kali perintahnya dilewatkan melalui shell. Jika memungkinkan, Anda harus menjalankan program secara langsung, daripada melalui pustaka shell (jika Anda menggunakan string tunggal, daripada daftar argumen, untuk memanggil program, kemungkinan besar program tersebut melewati pustaka shell). Namun, bahkan dalam kasus ini, data yang tidak valid dapat lolos. Oleh karena itu, validasi yang cermat diperlukan, selain sanitasi.

Ada beberapa kasus saat sanitasi tidak diperlukan. Jika data Anda telah divalidasi dan dapat dipercaya (misalnya, jika dijamin berupa bilangan bulat, atau berasal dari serangkaian string tepercaya), maka sanitasi mungkin tidak diperlukan, tetapi lebih mudah untuk menyertakannya dan meminimalkan risiko kesalahan daripada mengecualikannya dan secara tidak sengaja menimbulkan kerentanan.

Seperti halnya validasi, sanitasi harus selalu dilakukan di tepi sistem (misalnya, di adaptor, yang membuat permintaan ke sistem eksternal, atau di templat tampilan). Mematuhi aturan ini menyederhanakan desain, karena hanya ada satu tempat untuk memeriksa apakah sanitasi telah terjadi, tetapi juga menghindari risiko sanitasi ganda (terutama dalam HTML), yang dapat menunjukkan data yang rusak kepada pengguna.

Ancaman

Menerapkan praktik pemrograman yang aman sering kali akan membawa Anda jauh, tetapi Anda akan selalu harus membuat beberapa kompromi keamanan saat membangun sistem Anda. Ini bisa jadi karena alasan implementasi praktis, atau karena alasan pengalaman pengguna, tetapi tidak mungkin membangun sistem yang benar-benar aman. Dalam banyak kasus, ini sebenarnya tidak masalah! Kuncinya adalah memahami dengan benar bagian-bagian penting dari aplikasi Anda agar dapat melindunginya dengan tepat.

Keamanan informasi mirip dengan kesehatan dan keselamatan kerja. Jika dilakukan dengan benar, keamanan informasi akan menjaga orang-orang tetap aman dengan sedikit keributan, tetapi jika dilakukan dengan salah, keamanan informasi akan menghalangi semua orang, atau lebih buruk lagi, memungkinkan terjadinya kecelakaan serius. Dalam kesehatan

dan keselamatan, penilaian risiko digunakan sebagai cara bagi organisasi untuk mengembangkan perlindungan yang tepat. Dalam keamanan informasi, pemodelan ancaman digunakan.

Dalam penilaian risiko, Anda diharapkan untuk mengidentifikasi apa pun yang dapat menyebabkan kerugian (risiko; menghitung peluang terjadinya risiko tersebut dan dampaknya jika terjadi; dan kemudian mengidentifikasi cara yang tepat untuk mengurangi risiko tersebut berdasarkan pengetahuan tersebut (kontrol). Pemodelan ancaman serupa, yang dimulai dengan mengidentifikasi lubang keamanan teoritis dan ancaman dalam aplikasi Anda, kemudian mencari tahu peluang terjadinya hal itu dan dampaknya jika terjadi, dan kemudian menyusun strategi mitigasi yang tepat cara untuk mencegah hal itu terjadi.

Terkadang, mitigasi yang tepat mungkin tidak melakukan apa pun. Jika ada risiko teoritis yang dampaknya sangat kecil, dan peluang terjadinya sangat kecil, atau memerlukan banyak hal yang tidak mungkin terjadi, maka mungkin masuk akal untuk mengidentifikasi fakta bahwa hal itu mungkin terjadi, dan kemudian tidak benar-benar melakukan apa pun tentang hal itu, karena biaya untuk memperbaiki lubang tersebut mungkin sangat tinggi.

Misalnya, jika Anda menggunakan autentikasi dua faktor untuk masuk ke sistem admin yang berada di balik firewall, maka ada risiko di mana administrator diperas dan dipaksa untuk mengautentikasi, tetapi risikonya ini rendah dan biaya mitigasi tinggi (mungkin memerlukan dua orang untuk mengambil tindakan apa pun), jadi ini adalah risiko yang dapat diterima. Ada beberapa kerangka kerja yang dapat digunakan untuk pemodelan ancaman, tetapi semuanya mengikuti pendekatan yang sama. Yang pertama adalah memecah sistem Anda menjadi beberapa bagian penyusun, kemudian mengidentifikasi semua ancaman terhadap bagian tertentu dari suatu sistem.

Setiap ancaman kemudian dievaluasi untuk menentukan seberapa penting ancaman tersebut untuk ditangani, dan mitigasi diidentifikasi. Model ancaman Microsoft populer, dan dimulai dengan mengidentifikasi tujuan keamanan yang tepat dari aplikasi Anda. Misalnya, apakah itu melindungi identitas pengguna? Atau apakah serangan akan memiliki konsekuensi finansial langsung? Ketersediaan juga dianggap sebagai fitur keamanan, karena jika kelemahan keamanan dapat membuat situs web Anda offline, bahkan jika tidak ada informasi yang hilang, ini dapat berdampak finansial jika Anda memberikan perjanjian lapisan layanan kepada pelanggan, atau jika sistem tersebut memberi daya pada sistem lain yang diperlukan bagi Anda untuk menjalankan bisnis Anda.

Pemodelan ancaman juga memerlukan pemahaman yang baik tentang arsitektur sistem Anda. Mendesain arsitektur sistem dibahas lebih lanjut dalam bab Sistem, tetapi arsitektur tersebut harus menunjukkan batasan sistem Anda, dan dapat diberi anotasi jika ada kepercayaan antara komponen. Memahami bagaimana data mengalir melalui sistem Anda (menggunakan diagram aliran data) juga berguna dalam memahami bagian mana dari sistem Anda yang mungkin rentan. Untuk bekerja dengan komponen lain dari sistem terdistribusi, kontrak antarmuka juga dapat memberikan pengetahuan ini.

Dengan pemahaman ini, Anda sekarang dapat mencoba mengidentifikasi ancaman yang dapat memengaruhi setiap bagian sistem. Saat membuat model ancaman terhadap

sistem yang sedang berjalan, Anda harus mendekati ini dengan asumsi Anda tidak memiliki mitigasi yang ada, untuk memeriksa apakah asumsi yang telah Anda buat sebelumnya valid. Meskipun mengambil pendekatan tidak terstruktur untuk ini dapat mengakibatkan beberapa ancaman teridentifikasi, ada pendekatan yang lebih efektif yang dapat Anda gunakan dan gabungkan di sini.

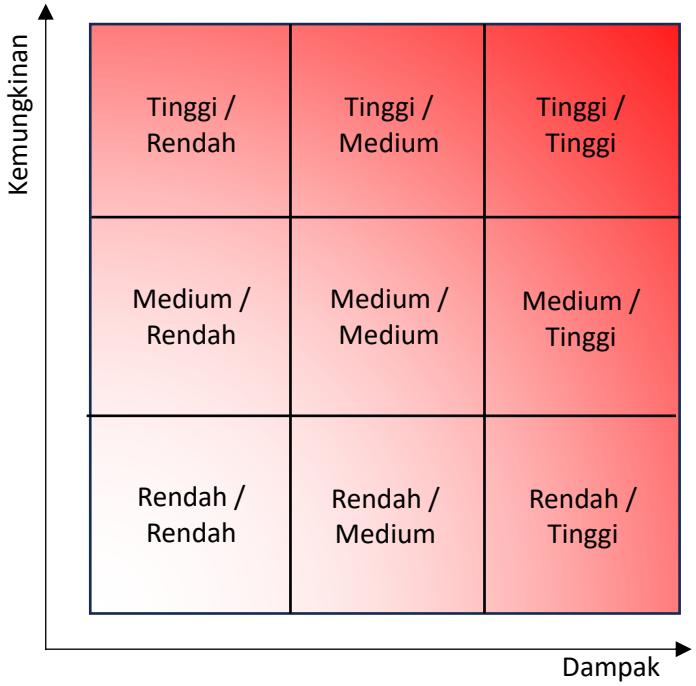
Yang pertama adalah pikirkan hal ini dengan cara yang sama seperti Anda memikirkan pengguna Anda, dan pertimbangkan berbagai jenis penyerang yang mungkin Anda hadapi: script kiddies (yang mencoba sejumlah eksploitasi umum secara otomatis), mantan karyawan yang tidak puas (yang memiliki pengetahuan internal mengenai desain sistem Anda), pengguna biasa (yang mungkin menemukan sesuatu secara tidak sengaja dan memutuskan untuk menjelajah), atau penyerang yang bertekad (termasuk penjahat terorganisasi atau aktor negara yang mungkin memiliki sumber daya yang signifikan). Anda juga dapat mempertimbangkan berbagai jenis serangan. Microsoft memperkenalkan mnemonic STRIDE untuk mengingat beberapa jenis umum:

- Spoofing identity: di mana pengguna dapat menyamar sebagai pengguna lain atau masuk sebagai pengguna lain untuk meniru karakteristik mereka.
- Manipulasi data: di mana pengguna dapat memanipulasi data yang seharusnya tidak dapat mereka akses (sering kali disebabkan oleh kegagalan validasi).
- Repudiation: di mana suatu tindakan tidak dapat dilacak kembali ke pengguna—misalnya, dalam aplikasi e-commerce, pengguna dapat mengklaim bahwa mereka memesan dan membayar lebih dari yang sebenarnya mereka bayar.
- Information reveal: ketika informasi yang seharusnya bersifat pribadi diungkapkan kepada seseorang yang seharusnya tidak memiliki akses ke data tersebut.
- Denial of service: ketika aplikasi dibanjiri permintaan, atau berjalan hingga kapasitas penuh, sehingga menghentikan pengguna lain untuk mengaksesnya. Sering kali, membatasi seberapa banyak sistem yang dapat diakses oleh pengguna tertentu, atau membatasi pencarian yang berjalan lama, dan sejenisnya, dapat mengurangi hal ini.
- Peningkatan hak istimewa: jika pengguna yang berbeda memiliki lapisan hak istimewa yang berbeda (misalnya, peran administrator dan non-administrator), maka aplikasi harus memastikan bahwa pengguna yang masuk tidak dapat memperoleh akses ke fungsi apa pun lebih dari yang seharusnya mereka miliki, dengan melakukan pemeriksaan otorisasi, misalnya.

Setelah aspek-aspek ini dipertimbangkan dan daftar ancaman yang komprehensif diidentifikasi, maka Anda perlu memeringkatnya untuk menentukan apakah ancaman tersebut perlu ditangani, atau apakah pengendalian yang ada sudah sesuai. Cara sederhana untuk melakukannya adalah dengan mengkategorikan setiap ancaman menurut peluang terjadinya rendah, sedang, atau tinggi dan tingkat dampak yang akan ditimbulkannya sekali lagi, rendah, sedang, atau tinggi. Ini adalah pendekatan yang diuraikan oleh pedoman NIST 800-30, tentang manajemen risiko TI.

Anda kemudian dapat menetapkan setiap ancaman sebagai risiko keseluruhan dan menentukan prioritas penanganan masing-masing. Gambar 12.1 menunjukkan cara umum

untuk melakukan ini menggunakan matriks, dan kemudian menggunakan skor keseluruhan untuk menentukan risiko keseluruhan, yang ditunjukkan oleh seberapa merah bagian matriks tersebut.



Gambar 12.1 Matriks Probabilitas Vs Dampak

Mnemonik umum lainnya yang digunakan di sini adalah DREAD, di mana lima atribut berbeda dipertimbangkan dan kemudian skor diberikan. Sering kali, skor nol hingga sepuluh digunakan, meskipun skala apa pun, selama yang sama digunakan untuk semua atribut, akan berfungsi. Kelima atribut tersebut adalah:

- Kerusakan: Apa dampak negatif dari suatu serangan? Nol sering kali berarti "tidak ada," dan sepuluh mungkin berarti bisnis bangkrut.
- Reproduktifitas: Seberapa konsisten serangan dapat dilakukan, atau apakah diperlukan sejumlah kendala di luar kendali penyerang agar tepat sebelum berhasil? Nol menunjukkan bahwa tidak seorang pun, bahkan pengembang, dapat mereproduksi serangan ini terhadap sistem yang sedang berjalan (jadi ini teoretis), dan sepuluh berarti serangan akan berhasil setiap saat.
- Eksploitasi: Seberapa sulitkah untuk benar-benar melakukan serangan? Nol berarti tidak seorang pun memiliki pengetahuan atau keterampilan untuk dapat melakukan serangan, dan sepuluh berarti bahwa orang awam akan dapat menemukan serangan selama penggunaan sistem normal.
- Pengguna yang terpengaruh: Berapa banyak pengguna yang akan terpengaruh oleh serangan ini? Nol berarti tidak ada, dan sepuluh berarti semuanya.
- Keterbukaan: Seberapa mudah bagi penyerang untuk mengetahui ancaman ini? Nol menunjukkan bahwa ancaman ini tidak mungkin diketahui dalam sistem yang sedang

berjalan, dan bahwa ancaman tersebut hanya akan terungkap hanya dengan membaca sumbernya; dan sepuluh berarti ancaman tersebut jelas bagi penyerang biasa atau dengan menggunakan alat otomatis.

Aspek "keterbukaan" sering kali menjadi penyebab kontroversi, karena dianggap bahwa mengandalkan sesuatu yang disembunyikan tidaklah tepat, dan sebagai hasilnya beberapa praktisi pemodelan ancaman akan menetapkan keterbukaan ke sepuluh (atau tingkat tertinggi), atau mengabaikannya sama sekali. Skor keseluruhan untuk ancaman adalah jumlah dari atribut-atribut ini (atau terkadang rata-rata), yang memungkinkan masalah dengan prioritas tertinggi diidentifikasi terlebih dahulu.

Setelah Anda mengidentifikasi setiap ancaman, Anda kemudian dapat merancang mitigasi untuk ancaman tersebut. Beberapa mitigasi ini merupakan bagian dari kebersihan kode yang baik, seperti validasi, dan sering kali tidak memerlukan pekerjaan tambahan, tetapi yang lain mungkin memerlukan upaya lebih lanjut. Dalam banyak sistem yang sangat penting untuk keselamatan, ada pepatah: bukan kesalahan pertama yang membunuh Anda, tetapi kesalahan kedua.

Tentu saja, idealnya suatu sistem tidak boleh membunuh Anda, tetapi membuat dua kesalahan berturut-turut lebih jarang daripada membuat satu kesalahan, jadi memastikan bahwa kesalahan yang menghilangkan satu lapisan keselamatan itu sendiri tidak mematikan sudah cukup baik. Hal yang sama berlaku dalam keamanan mengandalkan satu lapisan saja bisa berbahaya, dan Anda harus selalu bersikap seolah-olah satu lapisan saja dapat dieksploitasi. Ini dikenal sebagai pertahanan berlapis, di mana beberapa lapisan keamanan digunakan. Misalnya, alih-alih mengandalkan server API Anda untuk hanya diakses dari jaringan pribadi, mungkin diberlakukan oleh aturan firewall, Anda juga harus mempertimbangkan untuk menggunakan kunci API atau mekanisme lain.

Ini memberi Anda lapisan perlindungan tambahan jika firewall dikonfigurasi secara tidak benar, atau kerentanan di server lain memberi penyerang akses ke jaringan pribadi Anda. Pemodelan ancaman adalah cara yang bagus untuk mengidentifikasi di mana lapisan tambahan akan paling berguna. Saat Anda memodelkan ancaman, akan berguna juga untuk mengidentifikasi skenario di mana ada kompromi sebagian dari sistem yang Anda andalkan, dan kemudian mengidentifikasi di mana lapisan tambahan dapat paling berguna.

12.4 DAFTAR PERIKSA KEAMANAN

Daftar periksa adalah alat yang ampuh untuk membantu memverifikasi konsistensi dan mengingatkan Anda, dan pengembang baru di tim, tentang semua risiko yang terlibat dalam perangkat lunak, terutama jika tidak jelas. Dalam Kanban, daftar periksa membentuk kriteria masuk/keluar untuk kolom, dan dalam Scrum, definisi selesai. Untuk peninjauan kode, daftar periksa (sering kali otomatis) berguna untuk konsistensi diskusi. Daftar periksa keamanan sama saja, karena memaksa Anda untuk berpikir, meskipun hanya sebentar, dan mengonfirmasi bahwa Anda telah mengatasi masalah keamanan yang relevan.

Profesional keamanan telah membuat daftar "Top N" untuk jenis kerentanan

keamanan yang umum, dan cukup dengan membiasakan diri dengan daftar ini dan mengulanginya akan mengatasi jenis kesalahan yang paling umum. Dua daftar yang paling populer adalah OWASP (Open Web Application Security Project) Top 10, yang sebagian besar difokuskan pada pengembangan web, dan CWE (Common Weakness Enumeration) Top 25, yang lebih umum untuk semua jenis perangkat lunak. Seperti pemodelan ancaman, setiap risiko yang diidentifikasi harus memiliki kontrol yang tepat untuk memastikan bahwa risiko tersebut dikurangi atau tidak relevan.

Misalnya, CWE Top 25 mencakup CWE-120: Penyalinan Buffer Tanpa Memeriksa Ukuran Input ("Classic Buffer Overflow"). Dalam banyak bahasa web modern, hal ini tidak relevan, karena Anda tidak perlu mengelola buffer secara manual, jadi mitigasi/kontrolnya adalah bahwa hal itu ditangani untuk Anda oleh pustaka yang lebih rendah dan tepercaya. Pada bagian di bawah ini, 10 risiko teratas OWASP dibahas, dengan ikhtisar singkat tentang apa arti setiap kelemahan bagi Anda, dan cara melindungi diri dari kelemahan tersebut.

Injeksi

Serangan injeksi adalah satu jenis kelemahan keamanan terpenting yang dapat terjadi dalam aplikasi. Serangan ini berada di peringkat teratas OWASP Top 10 dan dua teratas CWE Top 25 karena suatu alasan. Serangan injeksi terjadi saat data yang tidak tepercaya tercampur dengan data tepercaya misalnya, memasukkan istilah pencarian dalam kueri SQL yang dapat ditafsirkan sebagai bagian dari struktur pernyataan SQL.

Dengan mengikuti "aturan emas" yang dibahas sebelumnya dalam bab ini, dan menggunakan pustaka dan teknik yang tepat untuk menyusun permintaan ini, Anda dapat menghindari jenis kesalahan ini. Serangan injeksi dapat menyebabkan berbagai jenis kerusakan misalnya, serangan ini dapat digunakan untuk menjalankan kueri SQL yang akan mengembalikan hasil untuk pengguna lain, atau bahkan menyebabkan hilangnya data sebagai akibat dari kueri UPDATE atau DELETE yang salah bentuk.

Autentikasi Dan Manajemen Sesi Yang Rusak

Fitur login dan manajemen sesi yang diterapkan dengan buruk dapat menyebabkan banyak kerusakan. Fitur login yang diterapkan dengan buruk mungkin hanya memeriksa nama pengguna dan kata sandi saat login lalu menyetel cookie "Logged_In_User: chris." Aplikasi lainnya secara membabi buta mempercayai bahwa kuki ini ditetapkan oleh aplikasi, tetapi penyerang dapat menetapkannya secara manual tanpa mengetahui kata sandinya, dan memperoleh akses ke akun orang lain. Demikian pula, kuki yang menunjukkan tingkat akses harus dihindari.

Sebaliknya, Anda harus menggunakan pustaka atau kerangka kerja autentikasi yang dapat mengurangi risiko menulis milik Anda sendiri. Ini biasanya mengikuti pendekatan yang direkomendasikan dengan memiliki hash atau token sesi, yang merujuk ke basis data (yang hanya dapat dikontrol oleh aplikasi) yang berisi informasi penting tersebut. Ini berarti bahwa kuki yang dicuri dapat digunakan untuk menyamar sebagai pengguna, tetapi mencuri kuki umumnya dianggap cukup sulit untuk berhasil mengurangi risiko ini.

Namun, beberapa kerentanan umum lainnya, jika berhasil diserang, dapat digunakan untuk mencuri kuki, jadi itu tidak sepenuhnya aman. Pendekatan alternatif adalah membuat

kuki yang memiliki tanda tangan kriptografi yang ditandatangani oleh kunci pribadi yang hanya diketahui oleh server, sehingga server dapat memverifikasi bahwa serverlah yang menetapkan kuki tersebut. Token ini harus kedaluwarsa atau diperbarui secara berkala untuk meminimalkan dampak token yang dicuri. Terkadang token dapat dibuat menjadi cookie sesi, yang berarti cookie tersebut hanya berlaku selama browser terbuka jika tidak, pengguna harus keluar secara eksplisit.

Cara umum untuk menghindari keharusan pengguna masuk terlalu sering adalah dengan mewajibkan cookie sesi untuk fungsi sensitif apa pun (seperti melakukan pembayaran) tetapi memiliki cookie yang lebih tahan lama untuk penelusuran umum, untuk memungkinkan hal-hal seperti personalisasi terjadi. Bagaimanapun, sebagian besar bahasa memiliki banyak pustaka yang mengimplementasikan fungsi ini, dan untuk sebagian besar situs, tidak banyak alasan untuk mengimplementasikannya sendiri daripada menggunakan solusi yang sudah ada.

Mengimplementasikan sendiri harus dianggap sebagai aktivitas tingkat lanjut. Cara lain yang dapat menyebabkan kerentanan ini secara tidak sengaja adalah dengan menyertakan token sesi atau pengenal unik dalam URL sebagai parameter kueri. Ini dapat berarti bahwa menyalin atau membagikan URL dapat secara tidak sengaja memungkinkan pengguna untuk mengizinkan teman mereka masuk sebagai mereka, atau bagi proksi perusahaan untuk secara tidak sengaja mencatat URL tersebut. Token sesi umumnya dianggap sebagai informasi sensitif, jadi hanya boleh disajikan di situs web terenkripsi.

Mekanisme pengaturan ulang kata sandi juga harus dirancang dengan hati-hati, untuk menghentikan orang menggunakannya sebagai cara untuk melewati proses masuk, dengan proses masuk itu sendiri memiliki pertahanan terhadap serangan brute-force (misalnya, membatasi kecepatan pengguna dapat membuat permintaan, atau mengunci akun setelah sejumlah permintaan yang salah).

Cross-Site Scripting (XSS)

Cross-site scripting adalah bentuk injeksi ke dalam struktur halaman yang dapat menyebabkan skrip acak berjalan di browser pengguna Anda. Seperti halnya penyuntikan, aturan emas berlaku di sini untuk mengatasi masalah ini. Peramban web memiliki model keamanan yang dikenal sebagai kebijakan asal yang sama. Inti dari kebijakan ini adalah bahwa kode JavaScript hanya memiliki akses ke halaman web yang berada di domain yang sama (misalnya, www.example.com) dengan halaman yang memuat JavaScript.

Misalnya, Anda tidak dapat membaca atau menyetel kuki untuk domain yang berbeda, yang dapat menghentikan pencurian kredensial, Anda juga tidak dapat memuat iframe untuk situs lain lalu menjalankan kode di halaman tersebut. Dengan XSS, ini berarti penyerang dapat menjalankan kode di domain Anda, dan karenanya mendapatkan akses ke kredensial yang tidak seharusnya mereka dapatkan, dan melakukan aktivitas yang biasanya diblokir oleh kebijakan asal yang sama.

Ketiga serangan di atas dapat digabungkan untuk menyebabkan kerusakan yang lebih parah. Misalnya, menyuntikkan konten yang salah bentuk ke dalam basis data dapat mengakibatkan serangan XSS pada pengguna yang mengunjungi situs web. XSS dapat digunakan untuk mencuri token autentikasi, atau menyebabkan terjadinya tindakan palsu, jika

dibiarkan terjadi di layar administrasi tempat pengguna tingkat tinggi masuk.

Referensi Objek Langsung Yang Tidak Aman

Referensi objek langsung yang tidak aman adalah jenis kesalahan yang terjadi saat sebuah "objek" misalnya, halaman HTML atau gambar tidak memeriksa dengan benar apakah pengguna tersebut diizinkan untuk mengaksesnya atau tidak. Hal ini sering terjadi pada hal-hal seperti konten yang diunggah pengguna, atau PDF yang dibuat. Misalnya, jika Anda sedang membangun sistem faktur yang membuat PDF, maka mengunggah PDF ini ke tempat yang memungkinkan pengguna untuk mengunduhnya dan memberikan tautan ke PDF tersebut di halaman tersebut mungkin tampak masuk akal.

Ada kesalahan mendasar yang dapat dilakukan di sini: jika URL Anda, misalnya, https://invoice-downloads.example.com/invoice0042.pdf, maka pengguna mungkin dapat menebak bahwa faktur lain mungkin ada di https://invoice-downloads.example.com/invoice0041.pdf dan mendapatkan akses ke informasi pengguna lain hanya dengan mengubah URL. Pentingnya melindungi jenis objek ini bergantung pada jenis aplikasi yang Anda miliki.

Untuk aplikasi berisiko rendah, maka cukup dengan memiliki URL acak saja sudah cukup, atau beberapa penyimpanan objek memungkinkan pembuatan URL yang ditandatangani dengan hash terbatas waktu yang menawarkan tingkat perlindungan yang lebih tinggi. Sering kali hal yang paling sederhana untuk dilakukan adalah mengatur aset Anda agar dilayani oleh aplikasi Anda dan menggunakan jenis autentikasi dan otorisasi yang sama dengan yang Anda gunakan untuk kode aplikasi reguler Anda. Halaman yang menyusun situs web Anda juga harus dianggap sebagai objek.

Misalnya, pada tahun 2011, bank CitiGroup mengizinkan orang untuk melihat akun orang lain dengan mengubah nomor akun di URL setelah mereka masuk.1 Jika sesuatu memiliki URL, itu berarti dapat diakses, jadi saat menerapkan keamanan Anda, Anda harus mempertimbangkan autentikasi dan otorisasi apa yang diperlukan untuk URL tersebut (dan parameter kueri apa pun yang mungkin diperlukan).

Kesalahan Konfigurasi Keamanan

Kesalahan konfigurasi keamanan dapat terjadi saat Anda menggunakan alat atau pustaka yang dikonfigurasi secara tidak benar dan dapat membuat Anda rentan terhadap kerentanan. Misalnya, kata sandi yang tersedia mungkin memiliki kata sandi default yang harus diubah, lalu lupa melakukannya akan menimbulkan kerentanan. Masalah seperti konfigurasi firewall yang salah akan diklasifikasikan sebagai kerentanan semacam ini juga.

Masalah umum lainnya terkait dengan mode debug dalam kerangka kerja. Sering kali, dengan mengaktifkannya, saat terjadi kesalahan, Anda akan mendapatkan jejak tumpukan atau beberapa informasi jenis debug lainnya. Sering kali, detail debug ini dapat memberikan informasi yang berguna bagi penyerang, seperti kode sumber atau nama pengguna dan kata sandi untuk basis data.

Pengungkapan Data Sensitif

Kerentanan pengungkapan data sensitif terkait dengan cara Anda mengelola data apa pun yang dapat dianggap sensitif. Penting untuk menentukan data mana yang dianggap

sensitif, lalu menentukan cara untuk melindungi data tersebut. Data dapat dianggap "sedang dalam perjalanan" (berpindah antarsistem), atau "tidak aktif" (disimpan di disk). Misalnya, jika Anda menggunakan cookie sesi untuk mengelola login, maka cookie tersebut harus dianggap sensitif, karena jika seseorang mendapatkannya, mereka dapat menyamar sebagai pengguna tersebut.

Saat tidak aktif, token sesi disimpan sebagai cookie di komputer pengguna, dan di basis data Anda. Kontrol akses yang tepat pada basis data mungkin cukup untuk melindungi token (jika basis data bocor, mengenkripsi token sesi tidak akan membantu, karena informasi pengguna telah terekspos), dan mungkin cukup untuk menganggap bahwa browser dan penyimpanan kuki pengguna sudah cukup terlindungi. Token sesi juga dikirim saat transit sebagai bagian dari kuki pada permintaan HTTP, yang lebih rentan.

Wi-Fi kedai kopi sering kali merupakan wi-fi "terbuka", yang berarti bahwa siapa pun di jaringan yang sama dapat melihat apa yang dikirim pengguna lain. Jika kuki Anda dikirim melalui HTTP biasa, maka itu akan membuatnya terekspos saat transit. Menggunakan enkripsi HTTPS dan menandai kuki Anda sebagai aman akan menghindari kasus penggunaan ini dan akan melindungi kuki Anda saat transit. Dalam contoh lain, basis data pengguna sering kali menjadi tempat penyimpanan bagi banyak data sensitif.

Kata sandi adalah salah satu contohnya, dan penyimpanan kata sandi yang aman dibahas nanti di bab ini. Undang-undang sering kali mendefinisikan informasi "yang dapat diidentifikasi secara pribadi" sebagai informasi sensitif, dan kehati-hatian harus dilakukan terutama terhadap data ini. Ini termasuk hal-hal seperti nomor Asuransi Nasional/Jaminan Sosial dan tanggal lahir. Cara paling sederhana untuk menangani informasi ini adalah dengan tidak pernah menyimpannya, tetapi jika Anda menyimpannya, Anda harus memastikannya ditangani dengan tepat.

Sekali lagi, data mungkin sedang dalam perjalanan ke banyak arah antara server web dan basis data Anda, atau ke server cadangan dan ini harus dienkripsi. Sebagian besar server basis data juga mendukung enkripsi saat tidak digunakan, dan penting untuk memastikan bahwa ini berlaku untuk semua cadangan juga. Sangat penting untuk memastikan bahwa Anda tidak menyimpan kunci dekripsi di samping basis data, terutama untuk cadangan!

Kontrol Akses Tingkat Fungsi yang Hilang

Kontrol akses tingkat fungsi mengacu pada pemeriksaan izin pada tingkat yang sangat rinci dalam suatu aplikasi. Misalnya, tidak cukup jika seseorang harus login untuk mengakses semua fitur, dan jika pengguna tidak memiliki akses ke sesuatu, menyembunyikannya di UI dianggap sebagai praktik yang baik. Namun, membatasi seseorang untuk melakukan tindakan di front end Anda tidaklah cukup; Anda juga perlu melakukan pemeriksaan di sisi server. Kontrol akses tingkat fungsi yang hilang terjadi ketika seseorang yang login melakukan tindakan yang seharusnya tidak dapat mereka lakukan karena server tidak memeriksa dengan tepat apakah mereka memiliki hak untuk melakukannya.

Artinya, pengguna diautentikasi (kami tahu siapa mereka), tetapi tidak diberi otorisasi (kami tidak tahu apa yang dapat mereka lakukan). Misalnya, fitur untuk membuat produk baru dalam katalog mungkin hanya muncul untuk pengguna yang masuk sebagai admin, tetapi jika

titik akhir AJAX tidak melakukan pemeriksaan itu di sisi server, maka setiap pengguna dapat membuat permintaan AJAX ke titik akhir itu secara langsung, melewati pemeriksaan front-end apa pun.

Ada banyak pustaka untuk membantu mengelola autentikasi dan otorisasi, dan jenis kesalahan ini sering muncul ketika Anda lupa melakukan pemeriksaan. Pengembangan berbasis pengujian seputar autentikasi, dan peninjauan kode, dapat membantu melindungi Anda dari jenis kesalahan ini. Jika pustaka dan kerangka kerja Anda mendukungnya, merupakan praktik yang masuk akal untuk menetapkan titik akhir Anda secara default untuk menolak penggunaan hingga pemeriksaan tertentu dilakukan.

Pemalsuan Permintaan Lintas Situs (CSRF)

Pemalsuan permintaan lintas situs terjadi sebagai akibat dari kebijakan asal yang sama yang digunakan peramban web untuk keamanan. Dalam serangan CSRF, pengguna mengunjungi situs web jahat yang membuat permintaan ke situs web Anda, misalnya, dengan menipu mereka agar mengeklik formulir, atau memuatnya melalui tag <img>. Meskipun situs web jahat tidak dapat mengakses kuki pengguna Anda, saat memicu permintaan, peramban mengirimkan kuki bersamanya, jadi jika pengguna Anda masuk ke situs web Anda, maka tampaknya mereka telah membuat permintaan yang valid, dan tindakan tersebut dipicu.

Solusi paling sederhana untuk ini adalah memeriksa apakah header Origin atau Referer ada dan disetel ke domain atau halaman yang diharapkan menjadi asal jenis permintaan tersebut. Solusi yang lebih komprehensif juga tersedia. Dalam desain RESTful API Anda yang baik, setiap tindakan yang berpotensi merusak/jahat harus disembunyikan di balik permintaan POST, bukan permintaan GET, jadi hanya permintaan POST (pengiriman formulir) yang perlu diperiksa.

Kolom tersembunyi dapat disertakan dalam formulir, yang dikirim ke pengguna termasuk token unik. Server kemudian dapat memeriksa apakah token yang diterimanya cocok dengan yang dikirim ke pengguna (jadi diketahui bahwa formulir yang diharapkan menghasilkan permintaan), dan token ini dapat disimpan dalam sesi atau dalam kuki yang juga dikirim ke pengguna.

Menggunakan Komponen Rentan Yang Diketahui

Jenis kerentanan ini terjadi saat dependensi yang Anda gunakan tidak selalu diperbarui. Mengelola dependensi, memastikannya selalu diperbarui, dan menyadari masalah keamanan apa pun tetap menjadi tantangan karena sifat pengembangan yang terfragmentasi: tidak ada standar universal untuk melaporkannya, juga tidak ada basis data pusat atau serangkaian perkakas untuk menyadari kerentanan dalam dependensi Anda. Memang ada perkakas yang difokuskan pada domain tertentu, dan menyertakannya dalam rantai pembuatan dan memiliki pendekatan sangatlah penting. Bagian Serangan Tidak Langsung di bawah ini membahas lebih rinci tentang cara melindungi dari serangan semacam ini.

Pengalihan dan Penerusan yang Tidak Divalidasi

Terkadang situs web Anda mungkin perlu mengalihkan pengguna ke halaman lain di situs Anda misalnya, halaman sukses setelah pengiriman formulir. Beberapa pengalihan ini mungkin menggunakan input pengguna untuk membuat bagian dari URL, dan ini rentan

terhadap jenis serangan injeksi. Pengalihan yang tidak divalidasi berbahaya karena pengguna dapat mengeklik tautan yang tampak valid (karena merupakan URL di situs Anda), tetapi pada akhirnya dapat berakhir di tempat lain karena injeksi. Sekali lagi, menerapkan aturan emas akan menyelesaikan masalah semacam ini.

Masalah serupa terjadi ketika bagian aplikasi Anda mengambil seluruh URL sebagai parameter untuk dialihkan kembali. Halaman login adalah tempat umum tempat kerentanan ini dapat muncul. Misalnya, jika saya mengunjungi https://www.example.com/my-account sebagai pengguna yang tidak login, saya mungkin dialihkan ke https://www.example.com/login?returnto=https://www.example.com/my-account.

Pengalihan ini baik-baik saja, karena tidak ada nilai yang secara langsung ditetapkan oleh pengguna, tetapi setelah login, halaman login mungkin hanya dialihkan ke parameter returnto tanpa memvalidasinya. Jika pengguna jahat menipu pengguna lain agar mengikuti tautan ke https://www.example.com/login?returnto=https://www. mybadsite.com/, mungkin dalam serangan phishing, pengguna mungkin terbuai oleh rasa percaya diri yang salah karena tautan tersebut adalah situs Anda tetapi mereka akhirnya diarahkan ke tempat lain. Ini adalah contoh lain di mana aturan emas dapat diterapkan parameter returnto harus diperiksa untuk memastikan URL yang menjadi tujuan pengalihan ada di domain yang Anda kendalikan.

Kata Sandi

Jika menyangkut identifikasi pengguna, ada dua pendekatan utama yang dapat Anda ambil: meminta mereka memberikan beberapa informasi yang hanya mereka ketahui, atau meminta mereka untuk membuktikan bahwa mereka memiliki sesuatu yang hanya dapat mereka miliki. Kata sandi adalah salah satu cara untuk melakukannya, karena kata sandi merupakan informasi yang, secara teori, hanya diketahui oleh pengguna. Namun, kata sandi telah menjadi salah satu pola keamanan terburuk yang digunakan secara luas.

Salah satu masalah dengan kata sandi adalah bahwa satu-satunya cara untuk memeriksa apakah seseorang mengetahui rahasia tertentu atau tidak adalah jika Anda sendiri juga mengetahui rahasia tersebut. Untuk melakukan ini, kita perlu menyimpan kata sandi tersebut sehingga kita dapat memeriksa apakah kata sandi yang diberikan pengguna kepada kita sudah benar. Ini berisiko, karena artinya siapa pun yang memiliki akses ke basis data (baik secara sah, atau melalui peretasan) kini memiliki kata sandi untuk setiap pengguna dan dapat meniru mereka.

Untuk mengatasinya, kita dapat membuat hash dari kata sandi yang kita simpan. Hashing adalah proses yang mirip dengan menghitung checksum, di mana Anda mengambil data Anda dan menjalankannya melalui beberapa fungsi matematika untuk menghasilkan hash (biasanya angka yang sangat besar yang dinyatakan dalam heksadesimal). Karena hash melibatkan pembuangan informasi, mustahil (untuk fungsi hash yang tidak rusak) untuk kembali dari hash ke kata sandi asli. Aplikasi Anda sekarang perlu mengambil hash dari kata sandi yang diberikan pengguna dan membandingkannya dengan hash Anda, tanpa harus menyimpan kata sandi.

Namun, hashing saja tidak cukup baik untuk melindungi dari kebocoran basis data. Meskipun mustahil untuk membalikkan hash, penyerang dapat memanfaatkan fakta bahwa

sebagian besar kata sandi adalah kata-kata kamus, dan sebagai gantinya cukup menghitung hash dari setiap kata dalam kamus dan kemudian memeriksa apakah ada hash yang cocok. Lebih jauh, karena kata yang sama akan selalu memiliki nilai hash yang sama, jika beberapa orang berbagi kata sandi yang sama, mereka akan memiliki hash yang sama, yang dapat membocorkan informasi.

Kita dapat mengatasinya dengan melakukan salting pada hash. Salting bekerja dengan mengambil hasil hashing kata sandi, menambahkan salt di bagian akhir, dan kemudian melakukan hashing lagi. Salt adalah string yang dibuat secara acak yang disimpan di samping kata sandi dan bersifat unik untuk setiap kata sandi dalam sistem, sehingga serangan dictionary seperti ini menjadi lebih sulit, karena penyerang harus mencoba membuat hash yang berbeda untuk setiap salt, daripada mencoba semua hash dalam database sekaligus. Peppering adalah konsep yang serupa, kecuali nilai tambahannya adalah rahasia dan sama untuk setiap pengguna, tetapi tidak disimpan dalam database (biasanya dikodekan secara permanen ke dalam aplikasi, atau diterapkan sebagai opsi konfigurasi run-time).

Menggunakan salt dan pepper secara bersamaan adalah hal yang umum. Hal terakhir yang perlu dipertimbangkan adalah fungsi hash mana yang akan dipilih. Fungsi hash telah digunakan sejak lama untuk memeriksa apakah transfer file sudah benar (jika Anda mengirim file dan hash, maka jika hash file tidak sesuai dengan yang diharapkan, maka ada kerusakan yang terjadi). Untuk kasus penggunaan ini, kecepatan penting, karena Anda dapat melakukan hashing data dalam jumlah besar. Namun, untuk kata sandi, yang terjadi adalah sebaliknya. Karena kata sandi pendek, meskipun kata sandi diberi salt, kata sandi tersebut masih dapat dipecahkan dengan mencoba banyak kata hingga ditemukan hash yang cocok.

Jika algoritme hashing Anda lambat, hal ini dapat memperlambat proses secara signifikan, jadi memilih hash yang lambat lebih aman. Tentu saja, ada pengorbanan kinerja, jadi fungsi hash Anda tidak akan memakan waktu beberapa detik untuk diselesaikan, tetapi mungkin 100 ms. Jenis fungsi hashing yang memenuhi kriteria ini disebut algoritme peregangan kunci, dan mencakup algoritme seperti bcrypt dan scrypt. Ada cara lain untuk melindungi basis data Anda dari serangan kamus, tetapi cara tersebut cacat.

Salah satu cara yang umum adalah dengan meminta pengguna untuk memperumit kata sandi mereka dengan menyertakan angka, huruf kapital, atau karakter khusus, tetapi hal ini akan meningkatkan beban pengguna secara signifikan karena harus mengingat variasinya. Sebaliknya, terkadang situs web menetapkan batasan panjang maksimum pada kata sandi untuk memastikan pengguna memilih sesuatu yang mudah diingat.

Anda sebaiknya menyertakan panjang maksimum, karena algoritme hashing kata sandi membutuhkan waktu lebih lama untuk kata sandi yang panjang, sehingga penyerang dapat mencoba menggunakan kata sandi yang berukuran beberapa megabita yang dapat memakan waktu beberapa menit untuk di-hash dan menyebabkan situs Anda mogok tetapi buat panjang maksimum menjadi sangat panjang. Anda juga harus memastikan bahwa pengguna dapat menggunakan karakter apa pun dalam kata sandi bahkan emoji harus valid dalam kata sandi!

Pembatasan paling efektif untuk diterapkan pada kata sandi adalah panjang minimum. Kata sandi yang panjang jauh lebih aman daripada yang pendek, tanpa harus membebani

pengguna Anda secara tidak semestinya. Nama alternatif untuk kata sandi adalah frasa sandi menggunakan kata ini mendorong pengguna untuk menggunakan beberapa kata atau bahkan seluruh kalimat sebagai kata sandi. Bahkan jika Anda mengamankan basis data Anda dengan tepat terhadap kebocoran, ada risiko yang jauh lebih besar dengan kata sandi. Sangat umum bagi pengguna untuk menggunakan kata sandi yang sama di beberapa situs untuk mengurangi beban mengingatnya.

Ini berarti bahwa jika situs web lain, yang tidak terkait dengan situs Anda, diretas, dan pengguna Anda menggunakan nama pengguna dan kata sandi yang sama, maka seorang peretas kini dapat masuk ke situs web Anda, meskipun kode Anda tetap aman. Ini adalah masalah mendasar dengan kata sandi, tetapi tidak ada alternatif yang sempurna; hanya sejumlah alternatif dan solusi yang masing-masing memiliki kelebihan dan kekurangannya sendiri. Ada banyak cara untuk menghindari masalah ini. Salah satunya adalah dengan menghindari penggunaan kata sandi sepenuhnya di situs Anda, dan sebagai gantinya meminta pengguna untuk masuk menggunakan akun yang mereka miliki di situs lain.

Ini bisa sangat efektif untuk peralatan internal organisasi, di mana Anda dapat menautkan ke sistem log masuk pusat untuk organisasi tersebut, tetapi pada dasarnya menyelesaikan masalah. Ini juga sering menawarkan pengalaman pengguna yang lebih baik, karena pengguna hanya perlu masuk ke satu situs dan banyak orang lain dapat memanfaatkannya, daripada harus memasukkan nama pengguna dan kata sandi Anda beberapa kali. Untuk situs web yang terbuka untuk umum, situs web media sosial dapat menyediakan fungsi ini, tetapi tidak semua pengguna Anda akan memiliki akun di situs tersebut, jadi nama pengguna/kata sandi tradisional (serta mendukung beberapa situs) juga diperlukan.

Beberapa situs menggunakan alternatif baru untuk ini, dengan mengirimi Anda tautan yang berisi token yang memungkinkan Anda untuk masuk. Namun, ini memiliki masalah, karena akun email menjadi satu titik kegagalan, dan jika Anda kehilangan akses ke sana atau akun email diretas, maka pengguna Anda kehilangan kemampuan untuk masuk ke situs web tersebut. Prosesnya juga tidak selalu lancar, karena pengguna harus membuka email mereka untuk masuk. Alternatif lain untuk kata sandi adalah menggunakan sesuatu yang dimiliki pengguna untuk memvalidasi identitas mereka. Ini bisa berupa token virtual atau fisik.

Sertifikat klien TLS juga berguna di sini, sebagai opsi virtual, dan umum di perusahaan besar. Pengguna mungkin lebih familier dengan perangkat yang menampilkan kode yang berubah seiring waktu.Perangkat ini disinkronkan dengan server saat pertama kali dikeluarkan dan kemudian secara acak menghasilkan angka baru menggunakan "seed" awal (komputer tidak dapat menghasilkan angka yang benar-benar acak, tetapi sebaliknya menghasilkan angka "pseudo-acak" dengan menerapkan fungsi matematika ke seed yang diketahui, atau status awal. Angka pseudo-acak ini seharusnya tidak mungkin diprediksi tanpa mengetahui status awal). Angka tersebut kemudian dimasukkan sebagai faktor kedua.

Token ini umum di perbankan daring dan akses jarak jauh atau koneksi VPN untuk perusahaan besar. Namun, memiliki token fisik untuk setiap situs juga sangat sulit. Sebaliknya, algoritma umum yang dikenal sebagai TOTP (kata sandi sekali pakai berbasis waktu) telah

digunakan secara luas, dan memungkinkan beberapa situs untuk menggunakan benih yang berbeda tetapi memiliki proses yang sama untuk menghasilkan kata sandi yang valid untuk "sekarang." Beberapa situs kemudian ditambahkan ke aplikasi, yang menghasilkan nomor valid saat ini untuk setiap situs.

Ada juga token fisik, seperti Yubikey, yang bekerja dengan cara yang sama tetapi tidak mengharuskan pengguna untuk memasukkan nomor itu sendiri, karena berkomunikasi langsung dengan situs. Salah satu kendala terbesar dalam sistem ini adalah bahwa waktu di semua perangkat harus benar (atau dalam beberapa detik). Jika tidak, angka yang dihasilkan dan yang diharapkan tidak akan cocok. Namun, masalah dengan menggunakan sesuatu yang dimiliki pengguna adalah jika sesuatu tersebut hilang, akun pengguna akan terkunci, dan jika sesuatu tersebut dicuri, pencuri dapat menggunakannya untuk masuk.

Daripada hanya mengandalkan metode ini, pendekatan ini semakin umum dikombinasikan dengan kata sandi. Pendekatan ini dikenal sebagai "autentikasi dua faktor", di mana pengguna harus memberikan kata sandi mereka serta membuktikan bahwa mereka memiliki token fisik. Kedua faktor ini jika digabungkan akan meniadakan hal-hal negatif dari faktor lainnya, tetapi memiliki sisi negatif yaitu lupa kata sandi, atau kehilangan faktor kedua, dapat mengakibatkan akun Anda terkunci. Proses masuk juga sedikit lebih lama dan lebih merepotkan, sehingga mengakibatkan pengorbanan kegunaan.

Autentikasi dua faktor lebih umum digunakan untuk melindungi sistem yang sangat sensitif, seperti rekening bank. Jika penggunaan kata sandi tidak dapat dihindari, Anda juga harus menghadapi kenyataan bahwa pengguna pasti akan melupakannya. Oleh karena itu, diperlukan mekanisme untuk mengatur ulang kata sandi pengguna. Untuk tim kecil, ini bisa sesederhana meminta pengembang untuk mengganti kata sandi dan pengguna menghubungi Anda secara langsung, tetapi ini tidak dapat diterapkan secara luas, terutama saat anggota masyarakat menggunakan aplikasi Anda, jadi solusi yang umum adalah menerapkan cara untuk memungkinkan pengguna mengatur ulang kata sandi mereka sendiri.

Namun, kehati-hatian harus diperhatikan saat merancang solusi untuk ini. Ada beberapa peretasan terkenal yang dilakukan individu akibat mekanisme pengaturan ulang kata sandi yang rentan. Kekuatan sistem hanya bergantung pada mata rantai terlemahnya, jadi meskipun skema kata sandi normal Anda sangat kuat, mekanisme pengaturan ulang kata sandi yang lemah akan membuat pengguna rentan diretas oleh pendekatan tersebut. Contoh umum mekanisme pengaturan ulang kata sandi adalah meminta pengguna untuk mendaftarkan "pertanyaan rahasia" dari daftar umum seperti nama gadis ibu mereka, atau tanggal ulang tahun selama pendaftaran, dengan gagasan bahwa pertanyaan-pertanyaan ini relatif mudah diingat.

Saat pengguna lupa kata sandi, mereka kemudian perlu menjawab pertanyaan-pertanyaan ini untuk membuktikan siapa mereka, tetapi masalahnya adalah jawaban atas pertanyaan-pertanyaan ini sering kali dapat ditebak atau diketahui berdasarkan catatan publik seseorang. Ini pada dasarnya menjadi versi kata sandi yang sangat lemah. Pendekatan lain adalah dengan menggunakan cara alternatif untuk menghubungi pengguna, seperti mengirimi mereka email, menelepon mereka, atau mengirim surat kepada mereka untuk memverifikasi

identitas mereka sebelum mengizinkan mereka menyetel ulang kata sandi mereka.

Cara ini bisa lebih lambat, dan memiliki implikasi biaya, terutama untuk mengirim surat, tetapi lebih aman daripada hanya menggunakan pertanyaan. Namun, cara ini tidak sempurna. Seperti alternatif "login melalui tautan ke email Anda" untuk kata sandi di atas, jika seseorang mengendalikan alamat email Anda, maka mereka juga dapat mengendalikan akun apa pun yang terhubung dengannya. Dalam kasus yang ditargetkan, mungkin juga untuk "mencuri" nomor telepon seluler. Tentu saja, ini juga mengasumsikan bahwa pengguna telah memperbarui detail kontak mereka. Pengaturan ulang kata sandi menjadi lebih rumit saat menggunakan solusi seperti autentikasi faktor kedua TOTP.

Jika pengaturan ulang kata sandi memungkinkan Anda melewati faktor kedua, maka hal itu pada akhirnya akan melemahkan seluruh sistem. Solusi umum adalah memberi pengguna serangkaian "kode cadangan" yang dapat digunakan sebagai alternatif kode TOTP jika perangkat utama hilang. Namun, ini mengharuskan pengguna untuk menyimpannya dengan aman. Saat membangun sistem login kata sandi, hal terakhir yang perlu dipertimbangkan adalah "brute-forcing".

Brute-forcing adalah mekanisme yang digunakan penyerang untuk mencoba meretas situs. Dalam serangan brute force, penyerang akan terus-menerus mencoba banyak kata sandi yang berbeda untuk pengguna, hingga akhirnya salah satunya berhasil. Anda harus membangun perlindungan terhadap brute forcing. Salah satu cara untuk melakukannya adalah dengan mencegah pengguna mencoba login setelah beberapa kali mencoba, baik untuk jangka waktu tertentu, atau hingga diatur ulang secara manual oleh administrator, tetapi ini juga dapat merepotkan bagi pengguna.

Mekanisme lain mungkin dengan memblokir sementara alamat IP setelah beberapa kali login yang gagal. (misalnya, biarkan IP mencoba masuk lima kali setiap 60 detik). Jenis lain dari upaya brute-forcing melibatkan upaya mencari tahu apakah pengguna memiliki akun di suatu layanan, atau alamat email mana yang mereka gunakan. Saat membangun mekanisme pengaturan ulang kata sandi, akan sangat membantu bagi pengguna untuk mengetahui apakah alamat email atau nama pengguna yang mereka masukkan tidak valid atau tidak. Namun, hal ini dapat memberikan informasi berguna bagi penyerang yang dapat membantu menargetkan serangan, jadi kegunaan dan keamanan harus diseimbangkan dengan hati-hati, tergantung pada seberapa berharganya akun pengguna individu.

Meskipun kata sandi jauh dari sempurna, pada akhirnya kata sandi tidak dapat dihindari. Akan ada saatnya Anda harus mengelola kata sandi selama proses pengembangan untuk bagian-bagian sistem pengembangan Anda. Untuk ini, menggunakan basis data kata sandi adalah praktik terbaik modern. Seperti namanya, basis data kata sandi mengelola kata sandi, memungkinkan Anda menyimpan kata sandi yang unik untuk setiap situs, dan kata sandi tersebut dapat panjang dan rumit, karena tidak perlu dihafal. Basis data dienkripsi, dan ada dua jenis aplikasi utama untuk ini: yang mengelola file, tempat Anda menyimpan file secara lokal, dan yang mengelola sinkronisasi antarperangkat.

Yang pertama pada akhirnya lebih aman, tetapi dengan mengorbankan kenyamanan. Saat bekerja di tim pengembangan, Anda sering kali perlu menyimpan beberapa kata sandi

bersama, mungkin untuk akun root. Penggunaan kata sandi bersama sehari-hari harus dihindari, karena sulit untuk melacak siapa yang melakukan tindakan apa, dan jika seseorang meninggalkan tim, Anda perlu mengubah kata sandi untuk memastikannya aman. Jika memungkinkan, Anda harus memiliki akun individual per pengguna, tetapi sering kali diperlukan semacam kata sandi "root". Basis data kata sandi untuk tim Anda menjadi cara yang berguna untuk berbagi rahasia ini.

Serangan Tidak Langsung

Meskipun serangan langsung pada infrastruktur Anda mungkin tampak seperti yang paling jelas untuk dilindungi, serangan itu bukanlah satu-satunya. Banyak serangan pada perangkat lunak yang Anda tulis akan dilakukan oleh alat dan pemindai otomatis yang mencoba teknik umum untuk mendeteksi injeksi SQL, kerentanan XSS, dll., atau oleh pemindai yang mendeteksi kerentanan yang diketahui dalam perangkat lunak dan pustaka populer. Mengikuti praktik terbaik di atas akan melindungi Anda dari jenis kesalahan keamanan umum, tetapi sangat jarang (dan juga merupakan ide yang buruk) untuk menulis perangkat lunak Anda dari awal tanpa menggunakan pustaka atau kerangka kerja apa pun.

Setiap situs web di luar sana bergantung pada beberapa kode yang ditulis oleh orang lain, baik itu pustaka, kerangka kerja, atau bahasa pemrograman yang bermanfaat, atau alat seperti Apache HTTPD Server, Varnish, atau bahkan utilitas tingkat OS seperti driver jaringan atau kernel OS. Untuk setiap dependensi yang Anda perkenalkan, Anda perlu mempertimbangkan dampak keamanannya. Ini menjadi lebih buruk lagi ketika Anda melakukan pengembangan JavaScript modern, karena sudah menjadi hal yang umum bagi banyak dependensi transitif dan bersarang untuk diperkenalkan, dan menjadi tidak realistis untuk mengaudit semuanya secara efektif.

Saat memperkenalkan dependensi atau pustaka baru, Anda harus meluangkan waktu untuk memikirkan bagaimana Anda tahu bahwa Anda tidak secara tidak sengaja memperkenalkan kerentanan. Perasaan naluriah dapat berguna di sini apakah proyek tersebut tampak terawat dan memiliki komunitas aktif di sekitarnya (atau, jika Anda membelinya, apakah vendornya tampak menyadari keamanannya)? Ungkapan "banyak mata membuat semua bug menjadi dangkal" populer, tetapi ketika kerentanan dalam perangkat lunak sumber terbuka yang besar terjadi, sering kali kerentanan tersebut lebih menonjol, sehingga membawa risiko yang lebih tinggi.

Di sisi lain, proyek yang lebih kecil mungkin tidak memiliki banyak pengawasan yang diterapkan padanya dari perspektif keamanan, sehingga berpotensi memiliki kerentanan yang tidak terdeteksi. Potensi trade-off sering kali bergantung pada konteksnya juga. Pustaka front-end untuk animasi cenderung tidak memiliki kerentanan dibandingkan dengan pustaka yang bertanggung jawab untuk memvalidasi input formulir. Anda juga harus memikirkan tentang bagaimana Anda akan tetap waspada terhadap kerentanan keamanan.

Kerentanan yang terkenal sering kali muncul di situs berita dan komunitas teknologi spesialis, jadi Anda mungkin mendengarnya melalui osmosis, tetapi yang lebih kecil tidak. Sering memeriksa dependensi yang kedaluwarsa (misalnya, menggunakan npm yang sudah ketinggalan zaman) dan memperbaruinya ke rilis patch terbaru dapat menjadi efektif, dan ada

layanan spesialis yang akan memeriksa dependensi Anda untuk melihat apakah Anda mengandalkan versi rentan yang diketahui. Banyak vendor OS memiliki umpan atau alat keamanan yang dapat Anda gunakan untuk melihat apakah ada paket dengan masalah yang diketahui yang diinstal.

Mengotomatiskan ini sekali sebagai bagian dari jalur penyebaran Anda dapat membuat hidup Anda jauh lebih mudah di masa mendatang dan, jika Anda bekerja di dalam perusahaan besar, sering kali dapat dengan cepat membantu Anda berteman dengan tim infosec pusat.Meskipun saran di atas akan melindungi Anda dari sebagian besar jenis serangan, ada juga risiko yang lebih jarang terjadi, tetapi lebih menakutkan, yaitu serangan yang ditargetkan. Meskipun serangan dapat ditargetkan langsung ke aplikasi dan infrastruktur Anda dalam produksi, aplikasi yang ditulis dengan baik dan dikonfigurasi dengan baik akan memiliki peluang bagus terhadap serangan semacam itu.

Meskipun kerentanan berdampak rendah mungkin masih ditemukan, ada cara untuk mendapatkan akses tingkat tertinggi ke aplikasi dan data Anda, yaitu kemampuan untuk mengubah kode yang berjalan di server Anda secara langsung. Alat seperti repositori Git, server CI, atau alat penyebaran adalah target yang sangat menarik, karena jika seseorang dapat mengendalikannya, mereka memiliki kendali penuh atas infrastruktur Anda. Ada sejumlah serangan berprofil tinggi yang disebabkan oleh server Jenkins yang ketinggalan zaman atau tidak diamankan dengan benar, serta sejumlah serangan berprofil rendah tetapi sangat merusak yang dilakukan oleh mantan karyawan yang tidak puas.

Mirip dengan "latihan kebakaran" yang digunakan untuk menemukan mode kegagalan aplikasi Anda untuk tujuan operasi, Anda harus menjalankan latihan untuk mengetahui semua cara seseorang yang jahat dapat mendorong kode ke dalam produksi. Kerentanan umum mencakup server Jenkins tanpa autentikasi yang tepat pada URL "tersembunyi", akun SSH bersama untuk penerapan yang kata sandinya tidak diubah saat orang meninggalkan tim, atau membiarkan orang berada di repositori GitHub setelah mereka meninggalkan organisasi.

Mengamankan cara kode dimasukkan ke dalam produksi sama pentingnya dengan mengamankan kode itu sendiri, dan Anda harus berhati-hati saat mengelola akses ke repositori kode dan kontrol alat penerapan Anda. Mengelola cadangan juga menjadi masalah keamanan; jika Anda secara tidak sengaja mencadangkan kredensial basis data, atau membiarkan cadangan Anda ditimpa saat dibuat, Anda dapat memberi penyerang kemampuan untuk membuat kerusakan yang tidak dapat dipulihkan menggunakan vektor yang bahkan tidak dapat Anda pikirkan.

"Karyawan yang tidak puas" dapat menjadi salah satu ancaman terburuk bagi organisasi. Hal ini relatif umum, dan karyawan diberi akses dalam jumlah besar tanpa banyak kepercayaan. Terkadang, karyawan ini juga memiliki kemampuan untuk menutupi jejak mereka. Log audit penting agar Anda dapat melacak tindakan yang dilakukan oleh pengguna, dan harus dilindungi agar tidak dapat dihapus. Risiko lainnya adalah bagi pengguna yang menggunakan Git. Git memungkinkan Anda mengubah riwayat atau memalsukan nama-nama komiter, karena nama-nama tersebut hanyalah metadata. Jika Anda menggunakan Git di situs seperti GitHub, maka Anda dapat mengaktifkan penandatanganan komitmen GPG untuk

memverifikasi bahwa penulisnya benar.

Bahkan jika Anda mengamankan alur kerja build dan meminimalkan kerusakan yang dapat dilakukan oleh satu karyawan yang tidak puas, lalu menerapkan praktik terbaik dalam membangun perangkat lunak Anda untuk meminimalkan risiko pemindaian drive-by atau bahkan serangan tertarget yang menemukan kerentanan, dependensi Anda tetap dapat menimbulkan kerentanan. Untungnya, selain jarang terjadi, hal ini mudah diatasi, jadi sangat masuk akal untuk melakukannya.

Jika Anda memiliki dependensi dengan rantai build yang kurang terlindungi, maka mungkin saja penyerang yang sangat bertekad dapat mendorong build berbahaya dari dependensi tersebut, dan bagi server build Anda untuk mengunduhnya dan menggabungkannya ke dalam kode Anda. Untuk banyak repositori paket komersial atau yang dijalankan oleh organisasi (seperti repositori Ubuntu atau Red Hat), hanya pengguna tepercaya yang dapat mendorong build, dan mereka memiliki rantai build yang sangat aman.

Namun, untuk repositori yang dijalankan oleh komunitas, seperti NPM atau PyPI, asal usul suatu paket kurang jelas. Jika kredensial pengembang dikompromikan, maka penyerang dapat segera mendorong versi baru. Sering kali, pembuatan versi semantik (“semver”) digunakan untuk memungkinkan perkakas build Anda secara otomatis mengambil rilis patch dari suatu dependensi tanpa masukan dari Anda, tetapi hal ini membuat Anda rentan terhadap masalah semacam ini (perlu dicatat juga bahwa hal ini dapat membuat Anda rentan terhadap bug yang tidak sengaja diperkenalkan semver adalah ide yang bagus, tetapi masih bisa salah!).

Sebagian besar pengelola paket mendukung penguncian versi (misalnya, Composer PHP dengan composer.lock, NPM JavaScript dengan shrinkwrapping, Pipenv Python dengan Pipfile.lock, dan RubyGem Ruby dengan Gemfile.lock), dan sebagian besar repositori paket komunitas tidak memungkinkan Anda untuk menerbitkan ulang sesuatu dengan versi yang sama seperti yang ada sebelumnya. Bagi mereka yang melakukannya, Anda mungkin ingin meletakkan cache lokal paket antara server build dan repositori hulu. Ini biasanya merupakan ide yang baik untuk dilakukan, karena Anda dapat menjamin versi yang "baik" yang diketahui tidak akan tiba-tiba menjadi buruk, dan ini akan membuat build Anda tangguh terhadap waktu henti server paket jarak jauh.

12.5 KESIMPULAN

Web menjadi tempat yang semakin tidak bersahabat, dengan semakin banyaknya serangan beberapa terarah dan yang lainnya lebih acak menjadi tantangan nyata bagi banyak organisasi. Membangun situs web dan aplikasi yang dapat diakses publik menghadirkan titik lemah potensial bagi penyerang untuk membobolnya. Anda memiliki kewajiban sebagai pengembang tumpukan penuh untuk membangun aplikasi yang aman dengan tepat, dan membangun keamanan pada tingkat dasar adalah cara terbaik untuk mencapainya.

Keamanan tidak hanya mencakup bug yang secara tidak sengaja dapat memberi penyerang akses ke informasi atau sistem yang seharusnya tidak mereka miliki, tetapi juga campur tangan dalam fase desain untuk menentukan di mana kepercayaan berada dalam suatu sistem, dan bagaimana menjalankan manajemen rahasia yang cermat seperti kata sandi

atau data pengguna. Kepercayaan ini juga mencakup pihak ketiga misalnya, jika Anda menggunakan dependensi dari repositori publik, atau menyertakan JavaScript di situs Anda langsung dari domain lain, apakah Anda yakin domain tersebut tidak dapat dikompromikan?

Anda juga harus merencanakan kasus terburuk, jadi jika insiden keamanan benar-benar terjadi, Anda memiliki proses yang tepat untuk menangani dan meminimalkan dampaknya. Menganalisis sistem dan desain Anda untuk masalah keamanan juga merupakan cara yang efektif untuk membangun sistem yang aman, dan ada sistem yang membantu melakukan ini. Saat berhadapan dengan risiko teoretis, analisis kemungkinan/dampak dapat menentukan risiko mana yang layak diperbaiki.

Sistem hanya seaman titik terlemahnya. Keamanan melalui ketidakjelasan sering kali difitnah, tetapi ini berarti bahwa keamanan tidak hanya dapat dilakukan melalui ketidakjelasan, dan pada kenyataannya ketidakjelasan dapat menjadi bagian dari pendekatan berlapis yang efektif. Ini berarti Anda tidak boleh hanya mengandalkan satu lapisan keamanan, kecuali, misalnya, basis data dapat dilindungi oleh firewall dan kata sandi. Ini termasuk infrastruktur dan lingkungan pengembangan Anda, karena ini sering kali dapat menjadi jalan masuk bagi penyerang jika tidak diamankan dengan tepat.

Terkait pengodean yang aman, aturan emasnya adalah memvalidasi masukan untuk menghindari data yang rusak masuk ke sistem, dan membersihkan keluaran untuk menghindari menampilkan data yang merusak yang secara tidak sengaja dapat merusak UI Anda. Memeriksa fitur baru terhadap daftar periksa keamanan juga merupakan cara yang baik untuk memastikan Anda telah memikirkan kemungkinan masalah.

Membangun sistem yang menggunakan kata sandi juga memerlukan pertimbangan khusus, karena kata sandi merupakan titik lemah dalam banyak sistem keamanan, terutama mekanisme pengaturan ulang kata sandi. Ada alternatif, tetapi pendekatan yang umum adalah menggabungkan kata sandi dengan faktor kedua (menggabungkan sesuatu yang dimiliki seseorang dengan sesuatu yang diketahui seseorang) untuk memastikan bahwa seseorang benar-benar seperti yang mereka klaim.

Tidak ada peluru ajaib untuk keamanan; itu adalah tambal sulam dari hal-hal kecil yang membentuk keseluruhan, jadi berhati-hatilah terhadap alat atau organisasi yang menjanjikan untuk menyelesaikan masalah keamanan Anda. Keamanan harus menjadi bagian mendasar dari produk Anda. Pengecualian utama untuk hal ini, di mana Anda mungkin ingin mendatangkan bantuan khusus dari luar, adalah pengujian penetrasi. Menemukan kerentanan menggunakan analisis eksternal merupakan keterampilan yang penting dan dipelajari, dan penguji penetrasi yang baik harus bekerja sama dengan Anda untuk menemukannya.

BAB 13
PENERAPAN

Setelah Anda membangun situs web baru, Anda pasti ingin meletakkannya di suatu tempat yang dapat dilihat dan digunakan orang. Namun, menunggu hingga pembangunan selesai untuk melakukan ini sering kali terlambat cara penerapan situs web dapat menambah kendala tambahan pada cara Anda membangunnya. Misalnya, jika Anda membangun aplikasi untuk skala horizontal, maka Anda tidak dapat menyimpan sesi dalam memori. Jika Anda menangani file yang diunggah dari pengguna, maka ukuran hard drive Anda dapat membuat perbedaan besar.

Inti dari gerakan DevOps adalah gagasan bahwa pengembang tidak lagi "melempar sesuatu ke dinding" kepada staf operasi. Di sebagian besar organisasi, orang-orangan sawah ini tidak pernah sepenuhnya akurat selalu ada semacam hubungan antara dev dan ops, tetapi DevOps adalah tentang menyatukan kedua disiplin ilmu tersebut untuk membantu tim ops menyediakan lingkungan yang lebih baik untuk menjalankan perangkat lunak, dan bagi pengembang untuk membantu memahami kendala lingkungan tempat mereka menerapkan.

Konsep penting lainnya yang perlu diingat adalah pengiriman berkelanjutan. Pengiriman berkelanjutan menuntut tim untuk merilis build ke produksi secara berkala dan lebih awal guna meminimalkan pemborosan dan "waktu simpan", dan memperluas gagasan integrasi berkelanjutan, yang telah dibahas dalam bab sebelumnya. Memang, tim yang matang akan merilis komitmen awal "hello world" mereka yang paling awal ke produksi untuk membuktikan bahwa alur build dan penyebaran mereka berfungsi sebelum beralih ke pengembangan produk. Tim-tim ini sering kali dapat membuat lingkungan baru dari awal dalam hitungan menit untuk meminimalkan biaya overhead dan membuktikan sistem otomasi mereka.

13.1 APLIKASI DUA BELAS FAKTOR

Heroku, penyedia "platform-as-a-service" (PaaS) awal, mengkodifikasikan serangkaian prinsip untuk menyederhanakan proses menjalankan aplikasi pada platform umum. "12 faktor" ini adalah batasan yang diterapkan pada aplikasi untuk memberikan kesan yang konsisten pada penyebaran dan pengoperasian aplikasi, dan untuk menghindari masalah umum yang dapat membuat menjalankan aplikasi dalam produksi menjadi sulit. Ke-12 faktor tersebut telah dikritik karena sangat berpusat pada Heroku, tetapi terlepas dari platform penyebaran Anda, faktor-faktor tersebut relatif umum, dan Anda harus mempertimbangkan setiap faktor untuk melihat apakah faktor tersebut sesuai untuk aplikasi Anda.

Satu Basis Kode Dilacak Dalam Kontrol Versi, Dengan Banyak Penerapan

Saat mematuhi faktor ini, setiap layanan yang Anda terapkan akan memiliki repositori kontrol versinya sendiri, yang merupakan tempat penerapan kode. Terlepas dari lingkungan tempat Anda menerapkan, kode harus berasal dari tempat yang sama (dan juga cabang yang

sama). Jika aplikasi Anda memerlukan beberapa basis kode untuk disalin ke server, maka Anda harus mempertimbangkan untuk memperlakukan setiap basis kode sebagai layanan terpisah atau menanganinya bersama-sama, dan jika aplikasi Anda memiliki cabang yang berbeda di lingkungan yang berbeda, itu akan mirip dengan memiliki repo yang berbeda.

Di sisi lain, jika Anda menerapkan repo yang sama beberapa kali untuk beberapa layanan, maka Anda harus mempertimbangkan untuk membagi fungsionalitas umum ke dalam pustaka bersama yang Anda masukkan ke dalam sistem dependensi, yang menetapkan tepat satu repositori per layanan. Ada teknik populer yang melanggar aturan ini, yang disebut "mono-repo." Dalam mono-repo, semua layanan dan pustaka dalam suatu organisasi berasal dari repositori yang sama.

ini sering digunakan untuk menghindari memiliki banyak versi pustaka bersama sekaligus, karena mengubah beberapa kode bersama akan menyebabkan semua bagian lain dari aplikasi mengambil perubahan tersebut, yang memungkinkan penyesuaian besar di beberapa basis kode sekaligus untuk dikonfigurasi. Ada keuntungan lain dari monorepo, termasuk menyederhanakan manajemen konfigurasi repositori, dan ini menyoroti bahwa 12 faktor tersebut tidak selalu menjadi kebenaran mutlak, tetapi lebih merupakan praktik yang perlu dipertimbangkan dengan saksama.

Nyatakan Dan Pisahkan Ketergantungan Secara Eksplisit

Sebuah layanan harus eksplisit tentang ketergantungan yang dimilikinya dan mengisolasinya dari semua ketergantungan sistem. Bahasa seperti JavaScript yang menggunakan alat seperti NPM bagus dalam hal ini, karena mereka akan menginstal semua ketergantungan secara lokal ke dalam folder bernama node_modules secara default, yang terpisah dari semua ketergantungan sistem. Agar ketergantungan dapat sampai di sana, ketergantungan tersebut harus ditentukan dalam file package.json.

Menjadi eksplisit mengurangi jumlah langkah manual yang diperlukan untuk menyebarkan aplikasi Anda, meminimalkan risiko salah langkah, karena sebagian besar alat otomatis akan memastikan bahwa ketergantungan Anda diinstal bersama aplikasi. Isolasi berarti Anda dapat memiliki kontrol yang lebih besar atas versi eksplisit daripada yang dapat diberikan oleh pustaka tingkat sistem, sehingga menghindari masalah apa pun jika pustaka tingkat sistem juga digunakan oleh alat lain dan versi yang berbeda diperlukan.

Isolasi memberikan keuntungan lain dalam hal penyebaran, yaitu aplikasi dan dependensinya dapat dipindahkan sebagai satu unit (misalnya, sebagai arsip .tar.gz tunggal), sehingga lebih mudah untuk disebarkan. Banyak bahasa memiliki sistem dependensi sendiri yang dapat menyediakan keduanya, seperti Ruby's Bundler, virtualenv Python, dan bahasa lain menyediakannya secara umum secara default, seperti "fat JAR" di Java, yang mencakup semua dependensi. Terkadang ada baiknya untuk melanggar sebagian aturan ini. Misalnya, beberapa program, seperti ImageMagick atau curl, dapat sangat rumit untuk dikemas sebagai dependensi, dan mungkin masuk akal untuk menggunakan versi yang diinstal sistem.

Jika Anda ingin bergantung pada dependensi tingkat sistem seperti ini, Anda harus tetap eksplisit dalam menentukannya. Pola penerapan yang umum adalah mengemas aplikasi Anda bersama dependensi terisolasinya menggunakan alat pengemasan sistem, yang

memungkinkan Anda menentukan dependensi tingkat sistem dengan cara ini (ini akan dibahas lebih lanjut nanti).

Simpan Konfigurasi di Lingkungan

Dengan mengikuti faktor ini, aplikasi Anda harus membaca semua nilai konfigurasi yang diperlukan dari variabel lingkungan, dan bukan dari file di disk. Konfigurasi untuk aplikasi Anda adalah apa pun yang mungkin dapat diubah di antara penerapan, atau yang mungkin ingin Anda ubah dalam waktu singkat tanpa harus mengubah kode. Konfigurasi tentang cara Anda menggunakan kerangka kerja atau pustaka apa pun di aplikasi Anda (misalnya, menyiapkan rute di Express) tidak sama dengan konfigurasi aplikasi Anda, dan tidak apa-apa untuk membuatnya menjadi hardcode.

Jenis hal yang mungkin ingin Anda miliki dalam konfigurasi Anda meliputi nama pengguna, kata sandi, dan nama host server basis data, "rahasia" lainnya, atau variabel per lingkungan, seperti alamat situs (lingkungan pengembang mungkin akan berjalan di URL yang berbeda dengan situs aktif Anda). Aturan ini mungkin salah satu yang paling sering dilanggar. Meskipun variabel lingkungan adalah cara yang sangat umum untuk mengatur konfigurasi, sering kali alat ops disiapkan untuk menangani file konfigurasi, sehingga aplikasi Anda dapat memuat konfigurasi dari file di lokasi yang dikenal.

Namun, hal ini relatif mudah diubah. SystemD menjadi alat umum untuk menjalankan layanan di tingkat sistem, dan merupakan metode default untuk menjalankan program layanan pada banyak distribusi Linux. Relatif mudah untuk mengatur SystemD agar memuat variabel lingkungan untuk layanan dari berkas konfigurasi, dan alat lain seperti dotenv dapat digunakan untuk mensimulasikan hal ini dalam bahasa lain secara langsung. Cara umum lain aturan ini dilanggar adalah dengan menyimpan konfigurasi sebagai file pengaturan dalam repo Anda, lalu memiliki satu variabel lingkungan atau sakelar yang menentukan lingkungan tempat Anda menjalankan, dan oleh karena itu file konfigurasi mana yang harus dimuat.

Meskipun nilai konfigurasi yang menunjukkan apakah Anda menjalankan dalam mode pengembangan atau tidak boleh, jenis file tingkat tinggi ini tidak boleh, karena dapat membuat sangat sulit untuk mengubah konfigurasi dengan cepat tanpa mengedit pengaturan pada instans yang digunakan atau harus menyebarkan ulang seluruh basis kode. Tidak memeriksa konfigurasi Anda ke kontrol versi juga dapat meningkatkan keamanan, dengan tidak membocorkan kata sandi basis data atau kunci API kepada siapa pun yang memiliki akses ke basis kode Anda.

Kelemahannya adalah Anda sekarang memerlukan sistem manajemen konfigurasi eksternal untuk mengelola konfigurasi ini, dan idealnya agar versi tersebut dikendalikan agar dapat mengembalikan perubahan konfigurasi apa pun yang dapat merusak sesuatu, serta memungkinkan audit atas perubahan apa pun. Beberapa tim melakukan ini dengan memiliki repositori "konfigurasi" yang lebih terkunci daripada repositori aplikasi utama, tetapi koordinasi beberapa repositori ini melanggar faktor pertama.

Perlakukan Layanan Pencadangan sebagai Sumber Daya Terlampir

Layanan pencadangan adalah apa pun yang dikomunikasikan aplikasi Anda melalui jaringan, seperti basis data atau API lain. Memperlakukannya sebagai sumber daya terlampir

berarti Anda harus berasumsi bahwa layanan tersebut adalah layanan lain yang berbeda dari aplikasi Anda dan meminimalkan asumsi apa pun tentang tempat layanan tersebut berjalan. Mungkin itu adalah basis data yang berjalan pada VM yang sama di lingkungan pengembangan, tetapi dalam produksi berjalan di lingkungan yang dihosting pihak ketiga. Mungkin selama pengujian diganti dengan varian yang rusak untuk mensimulasikan kegagalan.

Yang penting adalah aplikasi Anda tidak perlu peduli, dan harus melakukan semua komunikasi melalui jaringan menggunakan nilai yang ditentukan dalam konfigurasi Anda. Demikian pula, jika Anda perlu berbicara dengan dua API yang berbeda, setiap API harus dapat dikonfigurasi secara independen. Bahkan jika keduanya diinstal dan berjalan pada nama host yang sama, keduanya dapat dipisahkan dalam konfigurasi mendatang.

Bangun, Rilis, Jalankan

Proses penerapan aplikasi Anda harus memiliki tiga fase berbeda:

1. Bangun. Ubah kode sumber Anda menjadi sesuatu yang benar-benar dapat berjalan. Ini dapat melibatkan pemasangan dependensi apa pun, atau mengubah SASS front-end apa pun menjadi CSS final.
2. Rilis. Fase ini memberi kita nomor versi dan ID yang berbeda untuk pembuatan. Konfigurasi biasanya ditambahkan di sini juga, jadi mengubah konfigurasi melibatkan merilis ulang pembuatan (tetapi tidak membangunnya kembali).
3. Jalankan. Letakkan rilis (pembuatan + konfigurasi) di suatu lingkungan (mungkin di server fisik, di mesin virtual cloud, atau dalam kontainer) sehingga benar-benar dimulai dan melakukan apa pun yang perlu dilakukannya.

Menyelesaikan ini sebagai tiga tahap terpisah akan menyederhanakan cara Anda menalar perangkat lunak yang sedang berjalan. Idealnya, setiap build sesuai dengan tag dalam kontrol versi (atau setidaknya ID komit), sehingga Anda tahu persis versi sumber mana yang berjalan pada satu waktu, sedangkan jika garisnya kabur dan Anda telah mengubah kode secara langsung dalam produksi (dalam fase run), maka akan lebih sulit untuk melihat dengan tepat apa yang terjadi. Ini juga membuat rilis dan rollback lebih mudah, karena untuk membuat perubahan Anda hanya perlu menempatkan rilis baru ke dalam suatu lingkungan, dan untuk memutar kembali rilis, Anda mengambil bundel rilis sebelumnya.

Mengaburkan tahapan menyebabkan masalah lain. Misalnya, jika Anda menginstal dependensi dalam tahap run Anda, dan kemudian server pihak ketiga yang berisi dependensi Anda mati atau dependensi telah berubah, dan Anda perlu memulai ulang layanan Anda di server baru karena yang lama telah mogok, Anda tidak akan dapat melakukannya. Demikian pula, beberapa kerangka kerja memungkinkan Anda untuk membangun file sumber Anda dengan cepat, seperti menerjemahkan SCSS ke CSS. Ini dapat menciptakan overhead tambahan, memperlambat aplikasi Anda, dan lebih sesuai untuk lingkungan pengembangan. Membangun semua aset Anda terlebih dahulu memungkinkan waktu memulai atau memulai ulang yang cepat.

Satu-satunya situasi di mana Anda ingin mengaburkan batasan ini adalah saat menjalankan lingkungan pengembangan secara lokal, saat Anda ingin dapat membuat

perubahan dengan sangat cepat. Membangun ulang aplikasi Anda, mungkin hanya sebagian saja, saat aplikasi tersebut berjalan di lingkungan pengembangan memungkinkan Anda untuk merespons perubahan dengan cepat. Tidak ada alasan untuk membuat rilis baru dalam situasi ini, karena hal itu tidak menambah nilai apa pun dan Anda akan berakhir dengan banyak rilis dengan sangat cepat. Level terendah tempat rilis harus dibuat adalah dalam kontrol versi.

Jalankan Aplikasi sebagai Satu (atau Lebih) Proses Tanpa Status

Faktor ini memiliki beberapa implikasi. Pertama, aplikasi Anda harus dimulai dengan memanggil file yang dapat dieksekusi yang terus berjalan (mungkin membuat proses pekerja terkait jika sesuai). Selain itu, aplikasi tidak boleh menyimpan status apa pun secara lokal, tetapi selalu di penyimpanan cadangan yang terpasang. Ini mungkin berarti bahwa data sesi apa pun disimpan dalam basis data atau lapisan cache seperti Memcached, atau bahwa file apa pun masuk ke penyimpanan bersama seperti S3, bukan secara lokal di disk. Memiliki file secara lokal di memori atau di disk saat sedang diproses tidak masalah, tetapi permintaan berikutnya atau pekerja lain tidak boleh menganggapnya ada di memori.

Hal ini tidak berlaku jika menyangkut aplikasi PHP, dan aplikasi lain yang berasal dari gaya kerja lama yang dikenal sebagai CGI. Dalam gaya kerja ini, aplikasi Anda terdiri dari serangkaian skrip di disk yang dieksekusi sebagai respons terhadap permintaan dari pengguna, lalu dihentikan. Mekanisme lain seperti mod_php atau FastCGI mempercepat hal ini dengan menerapkan efisiensi, tetapi pada dasarnya prinsip dasarnya sama. Tidak ada yang salah secara mendasar dengan melanggar faktor ini dalam kasus ini selama Anda memperhitungkan gaya penerapan yang berbeda dari aplikasi ini, tetapi Anda harus memastikan bahwa skrip Anda tidak memiliki status.

Ini sebenarnya lebih mudah dalam beberapa hal, karena Anda tidak dapat menyimpan hal-hal dalam memori dengan pendekatan ini, tetapi Anda harus memastikan bahwa teknik apa pun yang Anda gunakan (seperti sesi atau menyimpan file) menggunakan penyimpanan cadangan yang terpasang, daripada, misalnya, menulis ke disk. Tentu saja, jika aplikasi Anda sendiri adalah basis data, aturan ini tidak dapat diikuti.

Aturan ini adalah yang paling penting dalam hal memungkinkan Anda untuk melakukan penskalaan. Jika Anda tidak memiliki apa pun yang disimpan dalam memori atau secara lokal, akan mudah untuk memindahkan aplikasi Anda ke server lain, atau menjalankannya di banyak server berbeda di balik penyeimbang beban untuk menyediakan kapasitas yang lebih besar. Ini juga berarti Anda dapat memulai ulang aplikasi Anda, baik selama penerapan atau untuk mengubah nilai konfigurasi, dengan cara yang meminimalkan dampak pada pengguna Anda.

Ekspor Layanan dengan Pengikatan Port

Ini berarti bahwa komunikasi ke aplikasi Anda harus melalui jaringan, dan bukan secara lokal dengan file atau mekanisme lain (seperti soket UNIX). Hal ini tidak hanya berlaku untuk layanan Anda, tetapi juga untuk layanan apa pun yang bergantung padanya. Dengan melakukan hal ini, Anda dapat membagi aplikasi Anda ke beberapa mesin. Bahkan jika penerapan awal Anda melakukan sesuatu yang sederhana, seperti menghubungkan ke database pada mesin yang sama, jika Anda melakukannya melalui port, Anda dapat memindahkan database tersebut ke mesin lain sesuai kebutuhan aplikasi Anda.

Ini adalah tempat lain di mana PHP dan aplikasi CGI lainnya gagal. Alih-alih mengekspos antarmuka mereka melalui jaringan, mereka mengekspos diri mereka sendiri sebagai file dan membiarkan alat lain (seperti Apache) mengeksposnya melalui jaringan. Alasan untuk menggunakan port adalah untuk menyederhanakan penerapan Anda ke aplikasi Anda, daripada aplikasi Anda ditambah server tambahan yang diperlukan untuk menjalankannya. Ini juga berlaku untuk aplikasi Java "WAR" yang berjalan di server aplikasi seperti Tomcat, berbeda dengan JAR mandiri yang memulai server jaringan mereka sendiri.

Skala Keluar melalui Model Proses

Ini mirip dengan pendekatan model stateless. Dengan menjalankan aplikasi Anda sebagai proses yang dapat dieksekusi secara individual, Anda dapat melakukan skala dengan memulai banyak proses (mungkin pada inti CPU lain di mesin yang sama, atau pada mesin lain). Dengan memecah aplikasi Anda yang lebih luas menjadi jenis proses per jenis layanan (mungkin pekerja antrean yang terpisah dari front end web Anda), setiap jenis beban kerja dalam aplikasi Anda juga dapat diskalakan secara independen.

Maksimalkan Ketahanan dengan Startup Cepat dan Shutdown yang Anggun

Secara tradisional, jenis startup dan shutdown aplikasi tidak dianggap sebagai tempat penting untuk dioptimalkan, karena peristiwa ini jarang terjadi. Aplikasi dengan dua belas faktor mungkin diharapkan melakukan ini lebih sering, baik sebagai akibat dari peningkatan atau penurunan skala, atau untuk menanggapi masalah seperti kegagalan VM lain dan perlu menjalankan yang baru untuk menggantikannya. Ini juga memungkinkan Anda untuk melakukan deploy lebih sering, selain menguntungkan alur kerja pengembangan Anda.

Dengan menghabiskan waktu untuk mengoptimalkan waktu startup Anda (sehingga tidak memerlukan waktu lebih dari beberapa detik), Anda dapat membuat proses individual "sekali pakai", yang berarti proses tersebut memerlukan lebih sedikit perawatan. Jika mulai berperilaku tidak semestinya, proses tersebut dapat dihentikan dan diganti dengan instance baru. Penutupan yang anggun berarti bahwa ketika suatu proses diminta untuk berhenti, maka proses tersebut harus mengambil waktu sejenak untuk menyelesaikan permintaan yang sedang berlangsung (tetapi tidak memulai yang baru) dan menutup semua koneksi basis data sehingga tidak meninggalkan jejak dirinya sendiri,

tetapi juga bahwa ketika terjadi perubahan penerapan atau konfigurasi, hal itu meminimalkan kemungkinan pengguna mengetahui apakah mereka terhubung pada saat itu. Jika aplikasi Anda membutuhkan waktu lama untuk memproses permintaan (mungkin menggunakan soket web atau merupakan pekerja antrean), maka aplikasi tersebut harus memberi tahu klien untuk menyambungkan kembali atau menempatkan permintaan kembali pada antrean agar proses lain dapat mengambilnya.

Jaga Pengembangan, Pementasan, dan Produksi Sedekat Mungkin

Meminimalkan celah antara semua lingkungan memungkinkan Anda untuk mencegah masalah khusus lingkungan dan meningkatkan kepercayaan diri Anda dalam penerapan (dan karenanya melakukannya lebih sering dengan lebih sedikit overhead). Ini mungkin berarti sejumlah besar perubahan yang relatif kecil misalnya, menggunakan database yang sama di semua lingkungan; beberapa kerangka kerja menggunakan SQLite dalam pengembangan,

tetapi yang dihosting untuk lingkungan lain.

Ini dapat mempercepat penyiapan awal, tetapi dapat menyebabkan bug kecil karena implementasi database yang berbeda yang membuang lebih banyak waktu dalam jangka panjang. Sebaliknya, berinvestasi dalam otomatisasi untuk menyiapkan lingkungan pengembangan yang mewakili produksi (menggunakan VM atau kontainer) dapat meminimalkan masalah ini. Ini dibahas lebih rinci di bagian berikut. Definisi asli dari aplikasi 12 faktor mencakup tiga area yang penting untuk menjaga lingkungan sedekat mungkin: waktu, personel, dan peralatan. Meminimalkan celah waktu berarti mengurangi waktu yang dibutuhkan antara membuat perubahan dan melihatnya dalam produksi.

Misalnya, jika Anda memiliki bug yang mengganggu yang muncul dalam produksi,jika waktu sejak penerapan terakhir Anda singkat, maka status lingkungan pengembangan seharusnya tidak terlalu banyak berubah, dan Anda seharusnya dapat mereproduksi dan memperbaiki bug tersebut secara efektif. Konsep ini sendiri sekarang dikenal sebagai "pengiriman berkelanjutan". Meminimalkan perbedaan personel antara orang yang mengembangkan aplikasi Anda dan orang yang menjalankannya berarti bahwa pengetahuan apa pun yang dimiliki orang-orang ini tentang kedua aktivitas tersebut dibagikan sebanyak mungkin.

Hal ini telah berkembang menjadi gerakan DevOps. Kesenjangan terakhir yang harus diminimalkan adalah perkakas. Menggunakan perkakas yang berbeda, atau bahkan versi atau konfigurasi yang berbeda dari perkakas yang sama, di lingkungan yang berbeda menurunkan keyakinan bahwa sesuatu yang telah berhasil di lingkungan sebelumnya akan berhasil di lingkungan berikutnya, karena belum memiliki kesempatan untuk membuktikan dirinya. Tentu saja, beberapa perbedaan diperbolehkan misalnya, lingkungan pementasan mungkin berjalan pada instans yang lebih lambat untuk tujuan penghematan biaya tetapi perbedaan itu harus ditandai dan dikontrol.

Perlakukan Log sebagai Aliran Peristiwa

Log harus dianggap sebagai serangkaian peristiwa berorientasi waktu yang terjadi di aplikasi Anda. Log biasanya ditulis ke berkas log di disk, tetapi terkadang alat eksternal mungkin ingin menggabungkan log dari beberapa server, atau mengumpulkannya secara terpusat. Aplikasi yang mencoba mengambil terlalu banyak kendali atas log dapat mempersulit hal ini, jadi sebaiknya Anda memperlakukan semua peristiwa log sebagai satu hal, dan menuliskannya di satu tempat. Sering kali tempat ini hanyalah stdout, yang dapat membantu dalam pengembangan seperti yang ditunjukkan di konsol, tetapi juga memberi kekuatan kepada alat seperti SystemD untuk mencatatnya secara terpusat, atau dengan cara menarik lainnya.

Membuat log dalam format terstruktur, seperti JSON, dapat berguna, tetapi log harus dapat dibaca oleh manusia (misalnya, di mesin pengembang) serta diurai oleh alat eksternal ini, jadi berhati-hatilah. Log harus diberi cap waktu dan memberikan informasi yang cukup dengan sendirinya agar dapat digunakan dalam mendiagnosis masalah atau memberikan informasi tentang kesalahan.

Jalankan Tugas Admin dan Manajemen sebagai Proses Sekali Saja

Penting untuk mendapatkan kode yang tepat untuk setiap tugas admin atau manajemen, meskipun hanya berjalan satu kali. Dengan demikian, kode untuk menjalankan tugas-tugas ini harus disimpan di repositori Anda dan disimpan bersama kode untuk aplikasi Anda yang lain. Ini memberi Anda keyakinan bahwa ketika Anda perlu menjalankan tugas itu, Anda menjalankan versi skrip tugas yang tepat di lingkungan yang tepat, karena Anda memulai proses itu dalam fase "jalankan" aplikasi di atas. Pendekatan lain (misalnya, memiliki skrip yang Anda jalankan langsung di shell, atau menghubungkan ke database dari mesin pengembang daripada lingkungan yang sama dengan kode yang sedang berjalan) terbuka terhadap kesalahan karena alasan yang sama dengan alasan Anda telah mengurangi masalah tersebut pada faktor-faktor lainnya.

13.2 MESIN PENGEMBANG

Untuk tim yang belum sampai di sana, dan ingin mencapainya, menyiapkan lingkungan pengembangan sering kali merupakan langkah pertama menuju dunia DevOps. Dulunya umum bagi pengembang untuk menginstal lingkungan pengembangan pada mesin mereka yang sebenarnya, menggunakan alat seperti MAMP, tetapi ini dapat menyebabkan masalah. Jika seorang pengembang harus menjalankan kode mereka di Windows atau OSX, maka kode tersebut sering kali harus lintas platform, atau berjalan dari jalur instalasi yang berbeda, atau memiliki dependensi yang tidak teridentifikasi yang kebetulan diinstal pada mesin tersebut.

Untungnya, munculnya virtualisasi telah mengubah hal ini. Pengembang kini dapat menginstal mesin virtual dengan OS yang sama dengan kode mereka, yang langsung menghilangkan berbagai masalah. Alat manajemen VM seperti Vagrant telah melangkah lebih jauh, memungkinkan konfigurasi otomatis dan berulang antara lingkungan pengembangan, yang berarti bahwa setiap pengembang kini juga dapat menyiapkan lingkungan yang persis sama, yang dapat meningkatkan efisiensi secara besar-besaran. Saya pernah bekerja di tim yang menganggap mendapatkan lingkungan kerja sebagai "ritus peralihan", dan menjalankan contoh produk dalam waktu kurang dari sehari dianggap sebagai pencapaian besar! Sejujurnya, ini tidak dapat diterima.

Vagrant pada dasarnya adalah konsep yang cukup sederhana ia menciptakan mesin virtual di mesin lokal Anda dari citra dasar (misalnya, instalasi Ubuntu atau CentOS yang baru dan belum dikonfigurasi, meskipun untuk perusahaan yang lebih besar, ini dapat berupa citra standar lainnya), lalu menjalankan skrip (atau serangkaian skrip) untuk menyediakannya. Seharusnya memungkinkan untuk mengonfigurasi "Vagrantfile" yang memungkinkan pengembang baru di tim Anda untuk menjalankan vagrant dan memiliki lingkungan pengembangan yang berjalan sepenuhnya saat perintah selesai.

Di luar mesin virtual terdapat konsep kontainerisasi, dengan Docker sebagai alat manajemen kontainer yang populer. Kontainer mengurangi overhead mesin virtual dengan berbagi beberapa sumber daya sistem seperti kernel, tetapi prinsipnya serupa dengan Vagrant, di mana serangkaian skrip digunakan untuk membangunnya. Kontainer dapat menjadi sangat rumit dengan cepat jika Anda memiliki banyak komponen yang bergerak, seperti server basis data.

Dalam mesin virtual, biasanya hanya menginstal dependensi ini pada mesin virtual yang sama yang menjalankan server pengembangan Anda, tetapi dengan kontainer, satu kontainer per tugas adalah hal yang umum, jadi kontainer basis data juga harus berjalan dan kemudian "diatur" sehingga kontainer aplikasi Anda dapat mengaksesnya. Orkestrasi kontainer dapat membantu dalam mengelola lingkungan produksi yang kompleks, tetapi ini berada di luar cakupan buku ini.

Lingkungan Produksi

Satu pertanyaan yang saya dengar dari tim yang baru mengenal Vagrant adalah, "Bisakah saya menjalankan vagrant ke produksi?" tetapi ini adalah kesalahpahaman tentang peran Vagrant. Vagrant sangat baik dalam mengotomatiskan proses pembuatan VM lokal dan menjalankan skrip untuk menyediakan mesin virtual tersebut, tetapi tidak memiliki kontrol yang terperinci tentang bagaimana mesin virtual tersebut disediakan, disebarkan, atau dikelola. Dalam dunia yang ideal, Anda dapat menggunakan skrip penerapan yang sama yang anda gunakan untuk menyediakan mesin virtual lokal seperti saat diterapkan ke lingkungan produksi, tetapi sering kali terdapat perbedaan kecil antara lingkungan ini yang berarti skrip yang sama tidak selalu cocok.

Misalnya, di mesin virtual lokal, Anda dapat menggunakan folder bersama untuk menyediakan kode antara desktop dan mesin virtual, mengaktifkan fitur seperti hot-reloading, dan menjalankan dalam mode debug. Untuk instance yang diterapkan, Anda mungkin tidak ingin melakukan hal-hal ini, dan lingkungan Anda mungkin akan menerapkan kode Anda baik dari paket yang dibuat atau tag Git. Namun, mungkin akan ada bagian dari proses penerapan yang dapat Anda gunakan kembali, dan memang harus! Jika digunakan dengan tepat, kontainer dapat mempermudah hal ini, karena Anda dapat menjalankan build kontainer yang sama persis secara lokal dan dalam produksi.

Lebih jauh, Anda sering kali menginginkan kontrol yang lebih besar atas infrastruktur Anda daripada yang dapat diberikan Vagrant atau kontainer saja misalnya, saat menyiapkan firewall atau kunci SSH, dan, untuk penerapan yang lebih kompleks, hal-hal seperti grup penskalaan otomatis, penyeimbang beban, dan pencadangan. Namun, aturan yang sama masih berlaku Anda dapat menggunakan API untuk mengotomatiskan pembuatan dan penyediaan lingkungan ini, dan membuatnya dapat diulang.

Sebagian besar tim memiliki beberapa lingkungan lingkungan pengembang lokal (berjalan di mesin mereka); sering kali beberapa lingkungan praproduksi seperti QA, UAT, atau staging; dan lingkungan produksi (yang berhadapan dengan pengguna). Dalam hal bagaimana aplikasi Anda disebarkan, setelah kode Anda meninggalkan mesin pengembang, semuanya harus dianggap sebagai lingkungan produksi. Tidak ada lingkungan yang harus dilihat sebagai kasus khusus, sehingga Anda dapat menghindari masalah apa pun yang hanya muncul dalam produksi; Anda memiliki tingkat keyakinan yang tinggi bahwa itu berfungsi.

Itu juga membuktikan bahwa otomatisasi Anda berfungsi. Tentu saja, akan ada beberapa perbedaan (URL yang berbeda, ukuran mesin yang berbeda, atau aturan firewall yang berbeda), dan mengekspresikan perbedaan ini dapat dicapai dengan nilai konfigurasi. Mirip dengan cara Anda dapat mengonfigurasi aplikasi Anda di berbagai lingkungan agar

berperilaku berbeda, Anda dapat mengonfigurasi infrastruktur Anda untuk melakukan hal yang sama. Sebagian besar alat infrastruktur mendukung konsep "parameter" untuk definisi Anda.

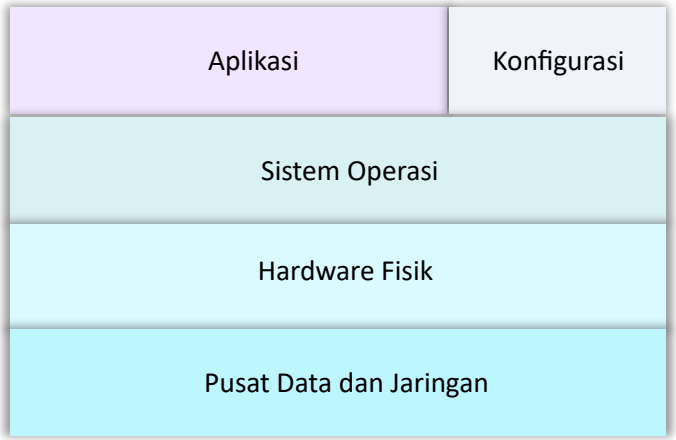
Saat menggunakan parameter ini dalam definisi Anda untuk hal-hal seperti "jenis instans" atau "jumlah mesin", maka Anda hanya perlu menjalankan kode yang sama dengan berkas konfigurasi yang berbeda persis sama dengan aplikasi Anda. Karena templat dan parameter ini didefinisikan dalam kode menggunakan alat otomatisasi, keduanya dapat diversikan dan dikelola persis seperti kode lainnya. Sering kali ada baiknya untuk benar-benar membuat versi kode ini dalam repositori yang sama dan di samping kode aplikasi Anda, yang memudahkan penerapan dan pengembalian perubahan infrastruktur dan kode secara bersamaan, daripada harus mencari tahu ketergantungan apa pun antara infrastruktur dan logika aplikasi Anda dan menerapkannya secara individual.

Memindahkan Kode ke Produksi

Pada tim DevOps, setiap aspek cara aplikasi Anda berjalan menjadi tanggung jawab tim. Terkadang tim infrastruktur dapat membantu, tetapi pada tim DevOps murni, tanggung jawab akhir selalu berada di tangan tim pengembangan. Di beberapa organisasi, terutama yang besar, hal ini tidak selalu terjadi, tetapi sering kali merupakan kompromi yang dapat diterima. Setiap bagian berikut akan membahas lapisan tumpukan aplikasi web yang sedang berjalan, seperti yang ditunjukkan pada Gambar 13.1.

Mari kita asumsikan sejenak bahwa Anda bekerja di organisasi TI tradisional, dan tim ops Anda bertanggung jawab untuk menyediakan server atau VM yang dikonfigurasikan tempat Anda akan menerapkan aplikasi Anda. Cara paling sederhana untuk melakukannya adalah dengan melakukan SSH, menggunakan Git untuk memeriksa perangkat lunak Anda di komputer, mengubah berkas konfigurasi, dll. Anda mungkin perlu menjalankan gulp atau npm install juga, untuk mendapatkan semua dependensi Anda di komputer atau membangun aset Anda. Saat Anda beralih ke DevOps, mudah untuk mengotomatiskan ini dengan skrip shell atau pekerjaan di jalur penerapan Anda untuk melakukan SSH dan menjalankannya, tetapi ini dapat menimbulkan tantangan.

Apa yang terjadi jika versi Node yang diinstal sedikit berbeda, atau beberapa perbedaan konfigurasi kecil lainnya berarti bahwa aset statis yang dibangun sedikit berbeda dari yang ada di lingkungan lain? Apa yang terjadi jika Composer, atau Ruby Gems, atau NPM mati saat Anda mencoba menginstal? Ini dapat menunda perbaikan bug penting yang diterapkan. Salah satu faktor dari aplikasi 12 faktor yang dibahas di atas adalah bahwa fase pembuatan, penyebaran, dan pengoperasian aplikasi Anda harus berbeda. Dengan menerapkan prinsip ini, Anda dapat menyederhanakan proses penyebaran yang rumit dengan membawa pembangunan ke fase lain.



Gambar 13.1 Lapisan Abstraksi Dalam Aplikasi Web Yang Diterapkan

Sebagai pengembang, Anda harus menyebarkan paket, daripada memeriksa aplikasi Anda dari kontrol sumber. Bagi pengembang Java, menyebarkan JAR tampak wajar, tetapi untuk bahasa lain hal ini belum tentu demikian. Akan sangat ampuh untuk menggabungkan dan menyebarkan aplikasi Anda menggunakan jenis paket yang sama dengan OS Anda: Deb pada Ubuntu/Debian atau RPM pada Red Hat/CentOS, dll. Jika Anda menggunakan pendekatan ini, Anda dapat mengelola aplikasi Anda seperti paket lain yang diinstal.

Namun, ini mungkin tampak menakutkan bagi pemula; banyak pembangun paket OS telah berkembang dari waktu ke waktu menjadi alat yang sangat ampuh, tetapi rumit. Alat seperti fpm, yang dapat digunakan untuk membangun paket jenis tertentu dengan cepat, berguna, tetapi ada baiknya juga untuk memahami cara membangun paket menggunakan alat asli. Sama saja dengan mengemas aplikasi Anda ke dalam file. tar.gz yang diunduh dan dibongkar ke dalam kotak tempat Anda ingin menyebarkan aplikasi, tetapi salah satu keuntungan menggunakan RPM atau deb adalah Anda dapat menentukan dependensi dan menjalankan skrip pasca-instal.

Misalnya, Anda dapat menyertakan file unit systemd di RPM Anda lalu menjalankan systemctl enable my-application, yang akan menyebabkan layanan Anda memulai otomatis saat boot dan dikelola oleh OS sebagai bagian dari proses boot normal. Alternatif umum adalah menggunakan alat seperti forever yang mengelola proses yang berjalan lama, tetapi alat ini perlu ditautkan ke proses boot Anda, dan membuatnya tetap sederhana dengan menggunakan alat yang sama yang digunakan OS masuk akal.

Menentukan dependensi juga sangat penting, dan aplikasi 12 faktor harus melakukannya secara eksplisit. RPM dan deb memungkinkan Anda menentukan dependensi apa pun sebagai bagian dari paket, dan saat menginstal paket, penginstal akan secara otomatis memasukkan apa pun yang dibutuhkannya. Penting untuk memisahkan dependensi tingkat OS semacam ini dari dependensi tingkat aplikasi apa pun. Misalnya, dalam aplikasi Java, .jar akan menyertakan semua dependensi, dan untuk bahasa seperti PHP, NodeJS, atau Ruby, Anda dapat menginstal dependensi yang diperlukan dalam folder di samping kode (misalnya, folder vendor untuk PHP dengan Composer, atau node_modules untuk Node.JS), yang dapat Anda gabungkan sebagai bagian dari paket Anda, dan gunakan alat asli untuk bahasa Anda untuk

mengelolanya, daripada mencoba dan menanganinya sebagai dependensi tingkat OS.

Dependensi tingkat OS mencakup hal-hal seperti runtime bahasa itu sendiri, jadi bergantung pada nodejs (terkadang versi tertentu) dalam RPM Anda adalah hal yang baik, selain pustaka lain (seperti ImageMagick atau libxml) yang biasanya diinstal menggunakan pengelola paket tingkat OS. Anda juga dapat, dalam paket Anda, menyertakan file konfigurasi untuk Nginx atau Apache, lalu bergantung padanya juga, jika Anda menggunakannya sebagai proxy terbalik. Jika dijalankan dengan benar, yang perlu Anda lakukan untuk menginstal aplikasi Anda ke dalam kotak adalah menggunakan pengelola paket Anda untuk menginstalnya ke gambar baru.

Ini sangat mudah dan sangat mudah diotomatisasi. Karena Anda hanya menginstal kode, alih-alih melakukan pembangunan apa pun sebagai bagian dari proses instalasi, kemungkinan besar menginstal RPM tersebut ke kotak yang berbeda (di lingkungan yang berbeda) akan berhasil. Paket-paket ini biasanya dibangun oleh server CI setelah berhasil melewati semua pengujian, lalu dipublikasikan ke repositori paket, siap untuk disebarkan. Jika Anda menggunakan kontainer, maka prinsip yang sama berlaku. Perbedaan utamanya terletak pada alat yang akan Anda gunakan, dan sering kali alih-alih membuat paket untuk aplikasi Anda, Anda malah membuat kontainer secara langsung, lalu memperlakukannya sebagai paket Anda.

Mengonfigurasi Kotak Anda

Meskipun Anda dapat menyederhanakan penerapan dengan membangun aplikasi Anda sedemikian rupa sehingga dapat diinstal dengan mudah, itu bukanlah satu-satunya hal yang perlu Anda khawatirkan. Menginstal paket berarti kode kita terinstal, tetapi tidak berarti kode tersebut dikonfigurasi. Sering kali ada komponen lain yang mungkin perlu disiapkan dan dikonfigurasi, seperti server basis data. Perbedaan konfigurasi antar lingkungan memang umum terjadi, tetapi perbedaan ini harus diminimalkan.

Misalnya, mengaktifkan tingkat pencatatan yang lebih tinggi pada lingkungan pengujian mungkin tidak masalah, tetapi menginstal versi basis data yang berbeda pada lingkungan produksi Anda ke lingkungan QA Anda tidak masalah. Dengan menstandardisasi lingkungan dan pendekatan Anda untuk mengonfigurasinya, Anda meminimalkan risiko apa pun yang dapat disebabkan oleh lingkungan produksi Anda yang berbeda dari lingkungan pra-rilis Anda. Untungnya, ada banyak alat di sini yang dapat membantu.

Alat yang paling sederhana mungkin adalah skrip shell, yang Anda jalankan pada kotak yang tidak dikonfigurasi untuk mengatur semuanya sesuai keinginan Anda, tetapi ada banyak alat yang jauh lebih canggih yang juga dapat berguna. Alat seperti Puppet, Chef, dan Ansible bekerja dengan mendefinisikan seperti apa seharusnya status suatu sistem, lalu alat tersebut berjalan pada sistem tersebut untuk memastikan bahwa sistem tersebut berada dalam status tersebut. Alih-alih serangkaian instruksi, Anda mendeskripsikan seperti apa konfigurasi sistem yang Anda inginkan, lalu alat tersebut mencari tahu perbedaan antara status saat ini dan yang baru, lalu membuat perubahan untuk memperbaruinya.

Sering kali hal pertama yang harus dikonfigurasi adalah aplikasi itu sendiri. Prinsip aplikasi 12 faktor menganjurkan Anda untuk mengonfigurasi aplikasi Anda berdasarkan

variabel lingkungan. Yang lain lebih suka menyediakan file config.ini atau JSON yang berisi pengaturan konfigurasi. Sering kali tugas alat otomatisasi Anda adalah membuat file-file ini, karena konten yang tepat mungkin bergantung pada mesin dan lingkungan tempat aplikasi tersebut digunakan. Terlepas dari cara Anda memilih untuk memasukkan konfigurasi ke dalam aplikasi, Anda harus menyuntikkan nilai konfigurasi ke dalam aplikasi, daripada membuatnya menjadi kode keras.

Merupakan hal yang umum untuk mengirimkan file konfigurasi sebagai bagian dari aplikasi Anda, atau untuk memperkenalkan kode yang bekerja berdasarkan lingkungan tempat Anda berada, tetapi lebih baik untuk memisahkan konfigurasi dari aplikasi Anda. Anda mungkin memerlukan beberapa untuk mengonfigurasi beberapa alat umum yang bekerja bersama aplikasi Anda, tetapi yang berhubungan dengan infrastruktur Anda dan bukan secara khusus dengan aplikasi Anda. Ini mungkin termasuk hal-hal seperti manajemen log (mengirim log ke agregator pusat) atau pengumpul metrik untuk pemantauan (lihat bab Dalam Produksi untuk detail lebih lanjut), dan jenis alat konfigurasi ini juga harus menyediakannya.

Konfigurasi OS tingkat rendah juga penting di sini. Ini mencakup hal-hal seperti aturan firewall atau mengonfigurasi pengguna yang diizinkan untuk melakukan SSH ke suatu mesin, dan alat otomatisasi umum seharusnya memudahkan Anda untuk mengaturnya juga.Semua hal ini dapat dan seharusnya diotomatisasi, dan setelah Anda mengotomatiskannya, Anda akan memiliki tingkat kepercayaan yang tinggi terhadap penerapan Anda, termasuk kemampuan untuk menjalankan lingkungan baru untuk pengujian, serta menyederhanakan proses penerapan Anda, karena Anda dapat menjalankan ulang skrip untuk setiap lingkungan untuk membuat penerapan.

Bastion Ssh

Catatan tambahan tentang SSH

secara umum dianggap sebagai praktik yang baik untuk menyembunyikan mesin individual dari Internet terbuka. Hal ini dapat dilakukan untuk web dengan menempatkan server web di balik firewall atau di subnet yang dilindungi firewall yang hanya mengizinkan koneksi dari penyeimbang beban atau gateway pada port 80 dan 443, tetapi itu berarti Anda tidak dapat melakukan SSH langsung ke kotak. Ada aliran pemikiran yang mengatakan bahwa Anda tidak boleh melakukan SSH ke kotak, dan pemantauan Anda harus cukup baik untuk memungkinkan Anda mendiagnosis masalah dari jarak jauh, tetapi bagi banyak orang, kemampuan untuk melakukan SSH tetap menjadi alat diagnostik yang ampuh.

Untuk mengatasi batasan ini, Anda dapat menggunakan bastion SSH (atau jumpbox/jumphost). Bastion SSH adalah mesin yang diamankan secara khusus yang hanya digunakan untuk menerima koneksi SSH yang masuk. Mesin ini juga harus dipercaya dalam firewall untuk SSH, sehingga pengembang dapat melakukan SSH terlebih dahulu ke mesin ini, lalu menggunakannya untuk melakukan SSH ke server individual sesuai kebutuhan.

13.3 INFRASTRUKTUR

Dalam organisasi yang belum sepenuhnya merangkul gerakan DevOps, infrastruktur secara tradisional dikonfigurasi oleh sysadmin, dan kemudian disediakan bagi pengembang

untuk menyebarkan kode mereka. Lingkungan "produksi" sering kali dikunci lebih lanjut di mana sysadmin juga menyebarkan kode. Seperti dijelaskan di atas, kita dapat menyiapkan server individual (atau mungkin armada server) menggunakan alat otomatisasi, tetapi dengan munculnya komputasi awan, kita juga dapat mengotomatiskan infrastruktur yang menjalankan server ini.

Secara historis, ini mungkin melibatkan pemasangan server fisik dan membuat perubahan jaringan fisik, tetapi dengan virtualisasi dan teknologi awan, semua tugas ini sekarang tersembunyi di balik API. Infrastruktur dalam hal ini dapat terdiri dari sumber daya komputasi dalam bentuk mesin virtual dan infrastruktur jaringan (seperti aturan firewall), tetapi juga mencakup sejumlah komponen yang mungkin sebelumnya disebut "peralatan" saat disebarkan di pusat data sendiri.

Peranti biasanya berupa kotak yang Anda pasang dan konfigurasikan, bukan harus memasang perangkat lunak pada mesin yang sudah ada. Sebagian besar penyedia cloud akan menawarkan beberapa fungsi umum, seperti basis data dan penyeimbang beban, sebagai peranti, terkadang menyediakan versi perangkat lunak yang dihosting untuk menghindari pengelolaannya sendiri. Dalam konteks cloud, infrastruktur mengacu pada komponen tambahan yang memungkinkan kode aplikasi Anda melayani pengguna.

Di satu ujung spektrum, ini mungkin mesin virtual (atau bahkan server bare metal) tempat perangkat keras dan host dikelola oleh penyedia cloud, tetapi di ujung spektrum lainnya, yang disebut opsi tanpa server, tempat kode Anda dieksekusi sebagai panggilan fungsi alih-alih proses yang berjalan lama, masih memerlukan konfigurasi host tanpa server Anda. Meskipun ini mungkin cukup untuk menjalankan aplikasi, Anda sering kali ingin menerapkan lebih banyak komponen di luar aplikasi Anda, atau menyiapkan firewall dan aturan keamanan lainnya yang sesuai dengan kemampuan komputasi Anda dan sesuai dengan peranti ini.

Salah satu bagian infrastruktur yang lebih umum adalah penyeimbang beban. Dengan menempatkan penyeimbang beban di depan aplikasi Anda, Anda dapat menjalankan beberapa instans aplikasi untuk menyebarkan beban dan menyediakan ketahanan, atau meminimalkan waktu henti dengan menjalankan versi baru aplikasi Anda secara paralel dengan instans yang sudah ada, lalu hanya perlu mengubah versi yang menjadi tujuan pengiriman lalu lintas penyeimbang beban. Bagian infrastruktur lainnya, seperti nameserver untuk hosting DNS, juga biasanya diperlukan.

Ada banyak komponen infrastruktur lain yang dapat digunakan. Layanan seperti AWS dan GCP juga memungkinkan Anda menyediakan kunci API untuk layanan mereka sebagai infrastruktur, dengan izin yang sesuai yang ditetapkan pada layanan tersebut, serta konfigurasi untuk alat pemantauan mereka sendiri guna mendukung aplikasi Anda. Penyedia layanan ini juga dapat menawarkan API lain untuk fitur seperti akses masuk tunggal atau alat pembelajaran mesin, selain utilitas yang lebih umum seperti penyeimbang beban.

Jenis yang paling umum dari layanan ini mencakup bentuk penyimpanan, seperti penyimpanan objek (untuk menyimpan file) dan basis data, dari basis data relasional hingga NoSQL atau penyimpanan nilai kunci. Alat bermanfaat lainnya yang tersedia sebagai komponen infrastruktur meliputi antrean pesan dan cache, meskipun terkadang Anda

mungkin memilih untuk menerapkannya sendiri di atas lapisan komputasi daripada menggunakan alat khusus vendor untuk menghindari penguncian, atau jika alat tersebut tidak sepenuhnya memenuhi kasus penggunaan Anda.

Bagian infrastruktur terakhir yang perlu dipertimbangkan adalah jaringan pengiriman konten (CDN). CDN pada dasarnya adalah lapisan cache HTTP yang menyimpan data lebih dekat ke pengguna Anda untuk meminimalkan waktu respons. CDN hanyalah mekanisme distribusi. CDN sendiri tidak memungkinkan Anda untuk benar-benar menyajikan konten apa pun. Untuk menggunakan CDN, Anda harus mengonfigurasi "asal" untuknya, yang merupakan tempat konten yang disajikan CDN berasal. Asalnya bisa berupa penyimpanan objek sederhana untuk menyajikan file statis, atau bisa juga aplikasi itu sendiri.

Dalam kedua kasus tersebut, header cache dalam respons HTTP digunakan oleh CDN untuk menyimpannya dalam cache, yang berarti kunjungan mendatang tidak perlu sampai ke asal, tetapi malah disajikan dari "tepi" CDN tempat data tersebut disimpan dalam cache. Mereka yang mempraktikkan DevOps biasanya bertanggung jawab untuk menyediakan infrastruktur mereka sendiri, dan ada dua pendekatan utama untuk melakukannya. Yang pertama disebut "platform-sebagai-layanan" dan yang kedua "infrastruktur-sebagai-layanan."

Meskipun istilah-istilah ini sering dikaitkan dengan perusahaan pihak ketiga seperti Heroku dan Amazon Web Services, ada konsep yang dikenal sebagai "private cloud", yang pada awalnya mungkin tampak seperti kontradiksi, tetapi dapat memungkinkan organisasi untuk terus mengelola server dan infrastruktur fisik mereka sendiri. Private cloud sering kali hanya menyediakan API dan alat pada infrastruktur internal yang menawarkan fungsionalitas serupa dengan yang tersedia dari pemasok pihak ketiga, tetapi kendala utamanya adalah bahwa alat internal ini terkadang tampak belum matang jika dibandingkan, dan kurang memiliki fleksibilitas yang dapat diberikan oleh penyedia eksternal.

PaaS lebih seperti model organisasi tradisional yang mengharuskan sysop menjalankan server dan pengembang melakukan penerapan pada server tersebut, meskipun komponen infrastruktur tambahan seperti CDN mungkin atau mungkin tidak menjadi bagian default dari platform ini. Salah satu pelopor awal di sini adalah Heroku. Infrastruktur-sebagai-layanan memberi Anda kontrol yang sama seperti yang mungkin dimiliki sysadmin atau teknisi jaringan tradisional, kecuali Anda tidak perlu benar-benar pergi ke pusat data untuk menyimpan server atau memelihara infrastruktur fisik. Anda cukup menentukan sumber daya yang Anda inginkan, dan konfigurasi di sekitarnya, dan semuanya akan tersedia untuk Anda.

PaaS sering kali lebih sederhana daripada infrastruktur sebagai layanan, tetapi sebagai gantinya kesederhanaan itu disertai dengan kurangnya fleksibilitas, sehingga ada beberapa kendala, seperti kurangnya persistensi atau terbatas pada satu OS. Dalam PaaS, Anda masih harus melakukan beberapa definisi infrastruktur, tetapi ini umumnya pada tingkat yang sangat abstrak mungkin hanya sejumlah dan jenis server. Infrastruktur sebagai layanan (IaaS) menawarkan tingkat kontrol yang tinggi, dengan risiko mengungkap kompleksitas yang mendasarinya kepada Anda, yang dapat menjadi hal yang menakutkan bagi pendatang baru.

Merupakan hal yang umum bagi organisasi yang menggunakan penyedia IaaS untuk menyediakan perkakas dan serangkaian default di sekitar layanan berdasarkan preferensi

organisasi tersebut agar terasa lebih seperti platform, bukan hanya sumber daya mentah. Pendekatan "default yang beropini" ini memungkinkan tim yang membutuhkannya untuk mencapai kompleksitas dan mengubah berbagai hal, tetapi bagi tim yang mengikuti praktik umum organisasi tersebut, pendekatan ini dapat mempercepat proses dan menurunkan hambatan untuk masuk dengan menyederhanakan pengaturan. Pendekatan terakhir ini bisa sangat ampuh dan memberikan yang terbaik dari kedua hal tersebut.

Dalam kedua kasus tersebut, API dan alat tersedia untuk mengelola cara penyiapan penyebaran tersebut, tetapi alternatif manual juga tersedia. Untuk tim DevOps yang efektif, otomatisasi harus digunakan, jadi penggunaan alat dan API tersebut untuk mengelola penyebaran menjadi penting. Misalnya, Amazon Web Services menyediakan CloudFormation, tempat templat JSON digunakan untuk mencantumkan setiap sumber daya Amazon yang Anda inginkan, dan setiap ketergantungan/referensi di antara keduanya (misalnya, Anda dapat menentukan basis data dan server web, lalu menyiapkan aturan firewall untuk mengizinkan satu sama lain mengakses satu sumber daya hanya dengan merujuk ke sumber daya lainnya, daripada perlu mengetahui alamat IP mereka, dan informasi lainnya, yang mungkin hanya tersedia setelah sumber daya tersebut dibuat, dan dapat berubah di antara lingkungan).

Banyak penyedia lain yang tidak menawarkan bahasa templating semacam ini, melainkan API yang memungkinkan Anda memanipulasi sumber daya secara langsung. Untungnya, alat yang disebut Terraform menyediakan pengalaman seperti CloudFormation, tempat Anda menentukan status yang ingin Anda miliki, tetapi kemudian menggabungkan API penyedia lain untuk menyediakan sumber daya yang dibutuhkan, dan memelihara status infrastruktur itu sendiri. Terkadang, Anda perlu menentukan berbagai aspek infrastruktur Anda, dan sebagian besar alat mendukung beberapa gagasan tentang parameterisasi, tempat Anda dapat menentukan beberapa nilai definisi infrastruktur Anda sebagai jenis konfigurasi.

Misalnya, Anda mungkin memutuskan untuk menggunakan mesin yang lebih murah/kurang bertenaga pada lingkungan pengujian daripada dalam produksi, tetapi ingin semua infrastruktur lainnya didefinisikan dengan cara yang sama. Dalam skenario seperti itu, Anda akan menggunakan templat yang sama untuk infrastruktur pada prod dan test tetapi menggunakan parameter untuk menentukan spesifikasi mesin apa yang harus digunakan. Alasan lain untuk menggunakan parameterisasi adalah sebagai bagian dari proses penerapan Anda.

Misalnya, jika Anda menerapkan mesin virtual, maka output dari proses pembuatan Anda kemungkinan besar adalah mesin virtual dalam registri (di Amazon, ini adalah AMI) dengan ID. Biasanya definisi infrastruktur Anda akan berisi ID mesin virtual untuk di-boot, jadi untuk menerapkan aplikasi Anda, Anda dapat mengubah parameter dengan ID mesin virtual yang di-boot ke yang baru, dan juga jika Anda perlu melakukan rollback. Dalam hal pengembangan aplikasi, memiliki lingkungan lokal, sering kali pada mesin pengembangan Anda, merupakan cara yang efektif untuk bekerja, seperti yang telah kita bahas sebelumnya dalam bab ini.

Namun, sisi sebaliknya adalah Anda mungkin tidak menjalankan infrastruktur yang sama dengan lingkungan produksi, tetapi pada sesuatu yang menirunya. Misalnya, kode lokal

Anda mungkin tidak disajikan melalui CDN, atau aplikasi Anda mungkin tidak berada di belakang penyeimbang beban. Ini adalah pilihan yang wajar untuk dilakukan, tetapi Anda harus menyadarinya. Untuk aplikasi yang lebih rumit misalnya "big data," atau jenis pembelajaran mesin maka akan lebih mudah untuk menjalankan beberapa infrastruktur Anda secara lokal, dengan bagian lain yang berjalan pada lingkungan yang disediakan dengan cara yang sama seperti infrastruktur produksi untuk aplikasi Anda.

Keinginan untuk lingkungan pengembangan lokal dapat menghasilkan pengembangan kode yang relatif portabel antar infrastruktur, yang dapat menjadi penting untuk menghindari ketergantungan vendor. Saat ini, penyedia infrastruktur Anda umumnya adalah pihak ketiga, tetapi ada pilihan yang penting antara terlalu bergantung pada infrastruktur mereka yang dapat membuat Anda berisiko pihak ketiga menghentikan layanan yang Anda gunakan atau mengubah harga dengan cara yang tidak berkelanjutan bagi Anda atau mengabaikan kemudahan untuk menghindari ketergantungan, tetapi kemudian membuat versi sistem yang lebih buruk.

Infrastruktur yang Tidak Dapat Diubah

Infrastruktur yang tidak dapat diubah berarti bahwa setelah Anda menerapkan aplikasi, infrastruktur dan kode serta konfigurasi yang berjalan di dalamnya tidak akan pernah berubah. Jika Anda perlu membuat perubahan, alih-alih mencoba mengubah bagian layanan yang sedang berjalan, Anda dapat menghapus bagian yang diubah dan menggantinya dengan versi baru dengan konfigurasi yang diperbarui. Hal ini membantu Anda menghindari infrastruktur yang "rapuh" dengan mendeskripsikan status infrastruktur secara lengkap dalam kode, bukan sebagai akibat dari perubahan yang dilakukan dari waktu ke waktu.

Hal ini juga memudahkan pengaturan lingkungan lain yang identik, dari definisi infrastruktur yang sama, untuk memungkinkan pengujian. Hal ini terkadang disebut sebagai pola "penerapan phoenix", karena Anda secara teratur membakar lingkungan Anda dan kemudian meluncurkannya lagi dari awal. Dengan melakukan ini secara berkala, Anda memastikan bahwa aplikasi Anda tetap dapat disebarkan dari awal, alih-alih secara tidak sengaja membangun di atas sesuatu yang telah ditentukan sebelumnya tetapi kemudian dihapus dan tetap tersedia karena tidak dibersihkan. Sangat mudah untuk memperkenalkan bug sehingga suatu program tidak dapat lagi diinstal dengan bersih, tetapi hanya berfungsi karena sesuatu yang tersisa dari penyebaran sebelumnya (mungkin direktori log).

Anda dapat melakukan ini secara berkala dalam lingkungan pengujian, tetapi ini juga berguna dalam konteks produksi. Pola yang dikenal sebagai penyebaran "biru-hijau" berarti bahwa Anda memiliki dua lingkungan produksi, meskipun hanya satu pada satu waktu yang benar-benar melayani lalu lintas produksi. Untuk menggunakan infrastruktur yang tidak dapat diubah di sini, Anda merobohkan lingkungan yang tidak digunakan dan kemudian membangunnya dari awal. Ini dapat dikombinasikan dengan strategi peluncuran "canary", di mana Anda kemudian dapat secara bertahap mengarahkan beberapa lalu lintas ke lingkungan baru ini untuk mengonfirmasi bahwa itu berfungsi, dan kemudian mengalihkan semua lalu lintas pada akhirnya.

Jika penerapan gagal, Anda dapat mengalihkan lalu lintas kembali agar dilayani dari

lingkungan sebelumnya. Aplikasi 12 faktor, terutama yang berkaitan dengan faktor "tanpa status", memudahkan pengenalan infrastruktur yang tidak dapat diubah. Anda dapat dengan mudah menerapkan pola ini dengan menyimpan status apa pun secara eksplisit di layanan yang terpasang dan bukan di aplikasi Anda. Bahkan saat layanan Anda adalah basis data itu sendiri, layanan tersebut tetap harus menyimpan datanya di suatu tempat pada akhirnya, sebuah disk. Anda kemudian dapat memperlakukan disk ini sebagai layanan yang terpasang dan menetapkan infrastruktur lainnya di sekitar basis data Anda sebagai tidak dapat diubah.

13.4 PENGIRIMAN BERKELANJUTAN & PENERAPAN BERKELANJUTAN

Pengiriman berkelanjutan dan penerapan berkelanjutan adalah dua teknik berbeda yang sering kali membingungkan satu sama lain. Pengiriman berkelanjutan adalah serangkaian prinsip rekayasa yang bertujuan untuk mengurangi waktu yang dibutuhkan organisasi untuk membuat perubahan pada perangkat lunak. Penerapan berkelanjutan adalah teknik di mana setiap perubahan secara otomatis diterapkan dari pengembangan hingga aktif.

Pengiriman berkelanjutan dibahas dalam bab Perencanaan, tetapi sebagai pengingat, inti dari pengiriman berkelanjutan adalah konsep alur, yang dilalui fitur atau perubahan dari awal hingga pengiriman. Setiap tahap alur kerja memiliki fitur otomatisasi tugas rutin untuk meminimalkan risiko dan biaya item yang melewati alur kerja. Menjelang akhir alur kerja, biasanya Anda ingin menerapkan kode Anda secara langsung, dan di sinilah penerapan berkelanjutan dan pengiriman berkelanjutan saling tumpang tindih.

Terkadang Anda mungkin ingin memiliki gerbang manual di alur kerja Anda misalnya, Anda mungkin memiliki pemeriksaan kewarasan manusia atau pengujian kegunaan sebelum rilis final tetapi banyak tim memilih untuk mengotomatiskan seluruh proses, dan di sinilah penerapan berkelanjutan berperan. Penerapan berkelanjutan mungkin tampak berisiko, dan memerlukan disiplin yang tinggi untuk mewujudkannya sepenuhnya, tetapi memiliki kemampuan teknis untuk melakukannya, bahkan jika Anda tidak sepenuhnya menggunakannya, sangatlah hebat dan dapat membantu menerapkan pengiriman berkelanjutan meskipun ada pemeriksaan manual.

Penerapan berkelanjutan berasal dari otomatisasi penuh penerapan, dan terwujud dalam tim yang memiliki satu perintah untuk menerapkan versi terbaru aplikasi Anda ke suatu lingkungan. Perintah tunggal ini dapat dipicu oleh pengembang atau, lebih sering, oleh beberapa jenis alat yang dihosting, seperti server CI tradisional seperti Jenkins, atau oleh server CD generasi baru seperti GoCD, yang memiliki alur kerja yang dibangun di sekitar metafora alur kerja alih-alih pekerjaan individual, seperti yang cenderung dilakukan server CI.

Ada teknik lain yang dapat Anda gunakan dengan pengiriman berkelanjutan, terutama saat menggunakan penerapan berkelanjutan, untuk membantu mengelola proses ini. Saat membangun suatu fitur, Anda dapat membuat beberapa komitmen ke basis kode yang berkontribusi pada fitur yang berfungsi, tetapi Anda mungkin tidak ingin segera mengungkapkan fitur tersebut. Bagi tim yang tidak menerapkan integrasi berkelanjutan murni, ini sering kali berarti mengerjakan cabang fitur hingga fitur tersebut selesai, lalu menggabungkannya. Integrasi berkelanjutan didasarkan pada gagasan bahwa penggabungan

itu berisiko, dan karenanya menganjurkan komitmen terus-menerus ("mengintegrasikan") ke satu cabang, setidaknya setiap hari, yang kemudian menjadi tempat semua rilis dibuat.

Pengiriman berkelanjutan didasarkan pada gagasan bahwa rilis itu berisiko, jadi merilis sering kali berarti Anda dipaksa untuk mengurangi risiko itu dalam proses Anda. Bagi tim yang mempraktikkan integrasi berkelanjutan tetapi tidak melakukan pengiriman berkelanjutan, sering kali rilis hanya dilakukan setelah fitur dibuat dan diuji, jadi tidak dirilis sebelum waktunya kepada audiens. Ini menimbulkan risiko baru: jika ada perbaikan bug cepat yang perlu dilakukan, atau perubahan dengan prioritas lebih tinggi muncul, maka rilis yang hanya berisi perbaikan itu menjadi sulit dilakukan, karena cabang utama juga akan berisi fitur yang sedang dalam proses. Anda dapat mengatasinya dengan membuat cabang dan meninggalkan integrasi berkelanjutan untuk sementara, tetapi itu menimbulkan kembali risiko penggabungan yang ingin dihindari oleh integrasi berkelanjutan.

Bendera fitur (atau tombol alih) adalah mekanisme tingkat kode untuk mengaktifkan integrasi berkelanjutan dan pengiriman berkelanjutan. Implementasi bendera fitur yang paling sederhana adalah pernyataan if di sekitar beberapa kode yang menonaktifkannya berdasarkan nilai konfigurasi eksternal. Misalnya, jika Anda sedang membangun bagian baru dari situs web, Anda mungkin memiliki file konfigurasi di lingkungan pengembang yang menetapkan nilai konfigurasi, lalu menggunakan nilai konfigurasi tersebut saat menentukan rute untuk aplikasi Anda dan memaparkan tautan ke halaman baru tersebut misalnya, di bilah navigasi.

Cara mengaktifkan/menonaktifkan fitur bergantung pada cara penerapannya, dan ada cara yang lebih rumit untuk mengelola bendera fitur selain hanya file konfigurasi, tetapi inilah inti ide di baliknya. Bendera fitur akan mengubah kode Anda menjadi kode mati saat dinonaktifkan untuk meminimalkan dampak bug yang mungkin Anda perkenalkan dan mengurangi overhead QA dari pemeriksaan regresi saat bendera dimatikan. Bendera fitur juga berguna, jika Anda meningkatkan atau mengganti fitur yang sudah ada. Cara paling sederhana untuk mengimplementasikan fitur bendera adalah menggunakan if/else untuk memilih antara dua jalur kode, yang sudah ada, atau yang baru.

Penggunaan tanda fitur dapat menghasilkan kode yang lebih rumit karena periode tumpang tindih antara versi lama suatu fitur dan versi baru, dan memerlukan disiplin untuk membersihkan setelah menyelesaikan fitur baru. Namun, tanda fitur dapat memungkinkan Anda untuk mengontrol peluncuran fitur baru dengan lebih baik jika diperlukan, mungkin mengaktifkannya hanya untuk sebagian pengguna atau pada sebagian server untuk memverifikasi bahwa fitur tersebut berfungsi di lingkungan produksi sebelum meluncurkannya sepenuhnya.

Mengembalikan fitur dengan tanda fitur, jika masalah teridentifikasi, tidak berarti harus benar-benar mengembalikan kode atau menerapkan versi lama aplikasi Anda, tetapi sebaliknya mengubah nilai konfigurasi, yang tetap memberi Anda kebebasan untuk membuat perubahan lain pada aplikasi Anda yang tidak terkait dengan fitur tersebut. Ini juga dapat dimasukkan ke dalam pengujian kegunaan dalam skala besar, di mana mekanisme yang sama yang memungkinkan Anda menandai suatu fitur dapat digunakan untuk menyajikan varian yang berbeda kepada pengguna Anda guna mengumpulkan data untuk pengujian A/B.

13.5 KESIMPULAN

Gerakan DevOps merupakan bagian penting dari pengembangan perangkat lunak modern, karena memberdayakan tim pengembangan tumpukan penuh untuk mengelola perangkat lunak mereka dalam produksi, meminimalkan overhead komunikasi dan, berpotensi, memberikan perubahan lebih cepat. Untuk mencapai hal ini, perangkat lunak harus dibangun dengan meminimalkan hambatan dalam mendukungnya di lingkungan produksi. Salah satu metode yang digunakan untuk melakukan ini adalah menerapkan "12 faktor" Heroku, yang merupakan masalah tentang cara aplikasi berjalan yang memungkinkannya dikelola melalui beberapa lingkungan dan mudah didukung.

Langkah pertama dalam banyak perjalanan DevOps adalah mengotomatiskan pembuatan dan pengelolaan lingkungan lokal. Mesin virtual umumnya digunakan untuk melakukan ini, karena memungkinkan Anda untuk menjalankan lingkungan yang dekat dengan lingkungan produksi, serta memberikan beberapa derajat isolasi antara lingkungan tersebut dan sistem yang sedang berjalan. Setelah ini selesai, proses pemindahan kode ke lingkungan produksi atau praproduksi akan diotomatisasi. Alih-alih menyebarkan kode dari sumber di server secara manual, satu paket harus dibangun, dan paket tersebut disebarkan ke server.

Untuk semua lingkungan ini, alat dapat digunakan untuk menyebarkan tidak hanya paket tersebut, tetapi juga konfigurasinya secara terpisah ini berarti Anda dapat dengan cepat mengubah konfigurasi layanan yang sedang berjalan tanpa harus membangun ulang dan menyebarkan ulang kode. Seiring dengan semakin banyaknya tim yang beralih ke komputasi awan, mereka memperoleh kemampuan layanan mandiri untuk mengelola infrastruktur yang mendasarinya. Infrastruktur ini dapat didefinisikan menggunakan templat dan dikelola bersama kode Anda dalam kontrol versi, suatu teknik yang dikenal sebagai infrastruktur-sebagai-kode.

Teknik yang lebih canggih, termasuk infrastruktur yang tidak dapat diubah, juga dapat diterapkan pada penerapan Anda, yang dapat meminimalkan risikonya dengan mengganti infrastruktur Anda dengan versi baru saat berubah, daripada mencoba memutakhirkannya di tempat. Dengan otomatisasi ini, penerapan yang lebih teratur dapat dilakukan, masing-masing dengan perubahan yang lebih kecil. Penerapan ini bahkan melibatkan kode yang sedang dalam proses, yang dapat tidak dapat diakses oleh aplikasi yang sedang berjalan atau diaktifkan dan dinonaktifkan menggunakan nilai konfigurasi yang dikenal sebagai tanda fitur.

Kemampuan untuk membuat penerapan yang lebih kecil dan lebih teratur merupakan faktor utama dalam membedakan tim tumpukan penuh dalam organisasi digital dari mereka yang berada di lingkungan yang lebih tradisional. Hal ini memungkinkan laju perubahan yang jauh lebih cepat dan meminimalkan waktu yang diperlukan untuk membuat perubahan.

BAB 14
DALAM PRODUKSI

14.1 DEVOPS: MENGINTEGRASIKAN PENGEMBANG DAN OPERASI

Dalam organisasi digital, para pemangku kepentingan menaruh banyak kepercayaan pada kemampuan tim full stack mereka untuk membuat keputusan yang tepat, dan pada pendekatan yang mereka ambil untuk mengembangkan perangkat lunak. Namun, seperti yang dikatakan Paman Ben dari Spider-Man, di balik kekuatan yang besar, datanglah tanggung jawab yang besar. Ketika sebuah tim memiliki kendali atas cara mereka membangun perangkat lunak, apa yang dibangun, dan kapan perangkat lunak itu diterapkan, tim tersebut juga harus bertanggung jawab atas cara perangkat lunak itu berjalan dalam produksi.

Istilah "DevOps," singkatan dari "developers-in-operations," mewujudkan keahlian dan pola pikir baru ini. Tim full stack akan memiliki QA yang tertanam (hasil dari gerakan yang disebut sebagai "developers-in-test"), tetapi ketika ini terjadi, ini bukan hanya tentang meningkatkan tingkat otomatisasi pengujian dalam suatu produk, tetapi juga tentang memastikan seluruh tim bertanggung jawab atas kualitas output. DevOps terkadang dikontekstualisasikan sebagai "tim DevOps" atau "DevOp," tetapi ini sering merujuk pada tim operasi tradisional yang menggunakan teknik otomasi modern, bukan tim yang menyatukan pengembang dan operasi dalam satu tim bersama.

Paling banter, tim seperti itu hanya meniru proses DevOps dengan menggunakan alat otomasi baru, yang membingungkan sebab dan akibat. Tim yang menanamkan budaya DevOps dapat menggunakan otomasi, tetapi itulah hasil dari budaya yang telah muncul. Bagian terpenting dari budaya DevOps adalah bahwa tim secara keseluruhan bertanggung jawab atas bagaimana produk berperilaku dalam produksi. Salah satu cara umum hal ini terwujud adalah dalam tim "bangun-dan-jalankan", di mana tim tersebut adalah garis depan dukungan untuk setiap masalah yang terjadi dalam produksi.

Beberapa tim, terutama yang memiliki basis pengguna yang besar, mungkin memiliki tim dukungan eksternal yang menangani garis depan, dengan daftar periksa dan buku petunjuk untuk membantu mendiagnosis masalah umum, tetapi dokumen tersebut dikembangkan oleh tim, yang memiliki hubungan dekat dengan dukungan garis depan. TI dalam organisasi tradisional, khususnya yang timnya dibangun berdasarkan fungsi teknis, dapat mengembangkan mentalitas silo. Terlalu sering, tim beroperasi dengan mentalitas "bekerja pada mesin saya", alih-alih menerima tanggung jawab bersama untuk kebaikan organisasi dan penggunanya.

Pada intinya, DevOps bertujuan untuk mendobrak hambatan tersebut, dengan perangkat lunak yang lebih andal dan waktu yang singkat untuk memperbaiki masalah saat terjadi, menjadi tujuannya. Ketika tim tempat saya bekerja mengadopsi DevOps, harus membawa "batphone" (ponsel siaga yang dapat digunakan sebagai titik kontak pertama untuk masalah) menunjukkan betapa nyata tanggung jawab baru ini. Tidak seorang pun menginginkan panggilan telepon pukul 3 pagi karena produk mereka rusak.

Latihan Kebakaran

Orang-orang yang paling tahu bagaimana suatu produk dapat rusak adalah orang-orang yang membuatnya. Latihan kebakaran adalah cara yang efektif untuk mengeksplorasi skenario ini dan mengidentifikasi pekerjaan yang diperlukan untuk memperkuat perangkat lunak Anda. Bagi tim yang baru mengenal DevOps, latihan kebakaran mungkin hanya sekadar latihan teoritis. Latihan ini dimulai dengan mengajukan pertanyaan seperti "apa yang akan terjadi jika basis data kita rusak?" Tim mengidentifikasi setiap masalah potensial, dan menyarankan solusi untuk masalah tersebut. Untuk semua skenario yang teridentifikasi, tim dapat memeriksa hal berikut:

- Bagaimana kita mengetahui situasi ini terjadi? Apakah kita memiliki pemantauan yang cukup di sekitar aplikasi untuk mengidentifikasi situasi ini, dan bagaimana hal itu akan terwujud?
- Apakah kita mengetahui dampaknya? Apakah ada mitigasi sementara yang dapat kita lakukan?
- Apakah kita mengetahui cara memulihkan dari situasi ini?
- Seberapa besar kemungkinan situasi ini terjadi?

Dengan menjawab pertanyaan-pertanyaan ini, tim dapat mengidentifikasi pekerjaan yang mungkin diperlukan; mungkin pencatatan atau alarm tambahan perlu diterapkan, atau daftar periksa ditambahkan ke buku petunjuk aplikasi. Mitigasi terhadap skenario ini dapat dianggap sebagai pekerjaan dan ditambahkan ke daftar pekerjaan seperti pekerjaan lainnya. Kisah pengguna seperti "Sebagai pengembang panggilan, saya ingin sistem beralih ke replika basis data saat yang utama tidak tersedia, sehingga saya dapat meminimalkan waktu henti selama insiden," atau "Sebagai anggota tim dukungan, saya ingin melihat nama host dalam log kesalahan saat koneksi gagal, sehingga saya dapat dengan cepat mengidentifikasi apakah satu server back-end gagal" dapat diterima sepenuhnya, dan mengungkapkannya dengan cara tersebut membantu pemilik produk memprioritaskan pekerjaan.

Tidak seorang pun menginginkan pemadaman! Namun, mengidentifikasi kemungkinan juga dapat membantu dalam hal ini. Jika aplikasi Anda disebarkan ke penyedia cloud di beberapa wilayah, ada kemungkinan terjadi kegagalan di beberapa wilayah (jadi Anda mungkin ingin juga menggunakan penyedia kedua untuk menghosting instans aplikasi Anda guna mengurangi risiko ini), tetapi kemungkinan terjadinya hal tersebut mungkin sangat rendah sehingga tidak sepadan dengan investasi untuk melakukan perbaikan.

Setelah tim merasa yakin bahwa mereka memiliki produk yang tangguh, mereka dapat mulai mengubah latihan tanggap darurat ini menjadi latihan yang lebih praktis. Mengidentifikasi kelemahan dalam rencana pemulihan di lingkungan yang terkendali jauh lebih baik daripada menemukannya dalam insiden yang sebenarnya. Misalnya, dalam lingkungan pengujian, tim dapat memutuskan untuk merusak basis data, atau menonaktifkan server utama, dan memeriksa apakah sistem menangani kegagalan seperti yang diharapkan, atau langkah pemulihan manual apa pun (seperti memulihkan cadangan) berfungsi sebagaimana mestinya. Tentu saja, ini hanya berhasil jika ada keseimbangan antara lingkungan ini dan lingkungan produksi.

Tim yang sangat berani sering kali memilih untuk menjalankan latihan tanggap darurat ini di lingkungan produksi mereka, terutama jika seharusnya tidak ada dampak pada pengguna. Bahkan tim yang memiliki niat terbaik akan memiliki perbedaan antara lingkungan seperti volume data, atau beban pada system yang dapat mengubah dampak insiden. Setiap anggota tim harus mengikuti latihan tanggap darurat, dan latihan ini harus dilakukan secara teratur. Ini dapat mencakup anggota tim non-teknis, seperti pemilik produk, untuk membantu membangun pola pikir seluruh tim tentang pengoperasian layanan. Terkadang jelas bahwa perubahan atau fitur baru pada sistem dapat memengaruhi cara sistem merespons insiden, atau memperkenalkan cara baru sistem dapat gagal, dan hal tersebut dapat diidentifikasi dan ditangani sejak dini tetapi terkadang hal tersebut kurang jelas.

Mengidentifikasi dan melatih cara menangani kegagalan yang paling umum atau mungkin terjadi akan meningkatkan waktu Anda untuk merespons secara signifikan. Ada serangkaian alat yang terus berkembang yang dapat mengotomatiskan proses menjalankan latihan kebakaran dengan sengaja menyuntikkan kegagalan ke dalam sistem secara terus-menerus. Hal ini dapat menghasilkan tingkat ketahanan yang tinggi, yang memaksa Anda untuk membangun pemulihan otomatis untuk jenis kegagalan tertentu (dan mengurangi beban kerja dukungan Anda).

Simian Army dari Netflix adalah contoh paling terkenal dari hal ini, dan mencakup Chaos Monkey, yang secara acak mematikan server tunggal, tetapi ada alat lain (seperti memperkenalkan latensi ke koneksi jaringan) juga. Jika tim Anda merasa nyaman menjalankan alat ini dalam produksi, maka Anda harus bangga karena Anda memiliki tingkat kepercayaan yang tinggi terhadap ketahanan produk Anda.

Buku Panduan

Buku panduan untuk aplikasi menjadi semacam kitab suci saat insiden terjadi. Dalam situasi yang penuh tekanan seperti pemadaman listrik, hal terakhir yang ingin Anda lakukan adalah mencoba mengingat di mana file log disimpan pada disk, URL untuk halaman status, atau detail sistem hulu penting yang menjadi tumpuan aplikasi. Hal lain yang harus disertakan dalam buku petunjuk adalah daftar periksa. Saat alarm berbunyi saat penerbangan, pilot maskapai penerbangan akan memiliki serangkaian daftar periksa untuk membantu mendiagnosis apa yang salah dengan pesawat dan cara memperbaikinya.

Hal terakhir yang ingin mereka lakukan adalah jatuh karena lupa memeriksa apakah penutup sayap berada di posisi yang benar, yang merupakan tugas sederhana yang mudah diabaikan dalam keadaan darurat. Untungnya, saat insiden terjadi pada aplikasi Anda, kecil kemungkinannya itu akan menjadi skenario hidup atau mati, tetapi hampir pasti akan ada tekanan nyata untuk menyelesaikannya, yang dapat menyebabkan kepanikan dalam kasus yang paling parah.

Memiliki daftar periksa yang ditulis oleh Anda dan tim saat Anda dalam keadaan tenang akan memungkinkan Anda untuk meniadakan sebagian kepanikan itu, dan insiden itu ditangani lebih cepat. Apa yang seharusnya dilakukan oleh daftar periksa yang baik? Daftar periksa harus memberi Anda serangkaian tindakan yang jelas untuk diambil sebagai respons terhadap peringatan. Peringatan bisa berupa laporan kesalahan dari pengguna, atau yang

dibuat oleh sistem pemantauan Anda. Ambil contoh, dua daftar periksa berikut:

Zenoss Telah Membuat Alarm Lowdiskspace Untuk Server Mysql

Dampak: Belum ada, mungkin meningkat menjadi penghentian layanan katalog e-commerce

1. Masuk ke server MySQL
2. Hapus semua rekaman dari tabel sesi yang terakhir diperbarui lebih dari tujuh hari yang lalu

Pengguna Melaporkan Bahwa Mereka Belum Menerima E-Mail Laporan Pemasaran Harian

Dampak: Pengguna internal mungkin tidak dapat melacak kinerja pengujian A/B yang sensitif terhadap waktu

1. Minta pengguna untuk memberikan alamat email mereka, dan untuk memeriksa folder spam mereka.
2. Jika pesan tidak ada di folder spam, masuk ke sistem laporan di https://reports.marketing.example.com/admin/ dan pilih “Edit Penerima Laporan,” dan pastikan alamat email mereka ada dalam daftar penerima. Jika tidak ada, tambahkan kembali.
3. Setelah pengguna ada di daftar penerima, pilih pengguna dan tindakan “Kirim Ulang Manual” di layar untuk memicu pengiriman ulang. Pastikan pengguna telah menerima laporan.
4. Jika pengguna yakin bahwa mereka seharusnya menjadi anggota sebelumnya, pilih “Log Audit” di layar admin dan periksa tindakan apa pun yang dapat menghapus mereka, dan tindak lanjuti dengan pengguna yang bertanggung jawab.
5. Jika pengguna ada di daftar penerima, pilih “Status Pembuatan Laporan” dan pastikan bahwa kolom waktu proses laporan menunjukkan waktu sekitar pukul 6 pagi hari itu. Jika laporan belum berjalan, eskalasikan insiden tersebut, karena akan memengaruhi semua pengguna, dan ikuti daftar periksa “Laporan Pemasaran Harian Tidak Dihasilkan”.
6. Jika laporan telah dibuat, masuklah ke gateway SMTP dengan SSH di smtp-gw.platform.example.com dan jalankan: grep <alamat email> /var/log/mail/outgoing.log. Periksa apakah ada kesalahan atau penundaan dalam log. Jika ada penundaan, paksa pesan untuk diproses dengan menjalankan “process-mail-mid <messageid>” dengan ID pesan dari berkas log. Jika ada kesalahan, laporkan ke tim email perusahaan.
7. Jika pesan tidak muncul di log keluar, periksa log aplikasi pelaporan email dengan mengakses portal pencatatan menggunakan kredensial di penyimpanan kata sandi tim. Terapkan filter "kesalahan laporan pemasaran", lalu periksa pesan kesalahan yang sesuai.

Daftar periksa kedua mungkin tampak terlalu bertele-tele, tetapi jika Anda cukup tidak beruntung menjadi pengembang baru di tim yang melakukan rotasi panggilan setelah jam kerja saat insiden terjadi, Anda akan berterima kasih atas detailnya. Bersikap eksplisit juga dapat membantu menghindari kesalahan. Misalnya, pada daftar periksa pertama, mungkin mudah untuk secara tidak sengaja mengakses basis data pengujian untuk membuat

perubahan, dan kemudian bertanya-tanya mengapa masalah tersebut belum teratasi.

Jika ini adalah sistem yang telah stabil selama beberapa tahun dengan sedikit kebutuhan untuk pemeliharaan, maka mudah untuk melupakan server MySQL mana yang menjalankannya, atau apa kredensialnya apakah login pengembang pribadi saya yang akan memberi saya akses, atau yang global? Bagaimana tepatnya saya mengakses basis data apakah ada phpMyAdmin, atau dapatkah saya terhubung menggunakan alat desktop, atau apakah saya perlu masuk SSH dan menjalankan baris perintah? Baris kedua juga berpotensi berbahaya; yang diperlukan hanyalah seseorang untuk menulis pernyataan SQL dengan kurang dari ditukar dengan lebih besar dari, dan semua sesi terkini telah hilang.

Lebih baik memiliki skrip sederhana yang dapat digunakan, atau lebih baik lagi, tugas pemeliharaan terjadwal untuk menghindari situasi tersebut terjadi. Daftar periksa pertama juga tidak lengkap: apa yang terjadi jika bukan karena tabel sementara yang penuh sehingga server kehabisan ruang disk? Tulis daftar periksa dengan asumsi bahwa pembacanya tidak tahu apa-apa tentang produk atau organisasi Anda. Hal ini terutama berlaku jika Anda bukan pendukung lini pertama untuk aplikasi Anda.

Akan sangat membantu jika menautkan ke insiden sebelumnya (yang mungkin terekam dalam sistem tiket), karena komentar pada tiket tersebut dapat membantu mendiagnosis masalah yang rumit. Elemen terakhir yang harus disertakan dalam buku petunjuk adalah jalur komunikasi yang jelas. Untuk insiden dengan tingkat keparahan tinggi, mengomunikasikan dampaknya adalah kuncinya. Misalnya, dalam pemadaman listrik di mana daftar periksa belum menyelesaikan insiden, menghubungi pimpinan teknis untuk tim tersebut dapat memberikan wawasan tambahan tentang cara melanjutkan.

Dalam kasus lain, jika pemadaman listrik telah memengaruhi fungsi bisnis penting, seperti kemampuan untuk mendaftar ke situs web, maka tim pemasaran atau dukungan pelanggan mungkin perlu mengetahuinya, sehingga mereka dapat menangani keluhan apa pun yang masuk melalui email, atau menangguhkan kampanye pemasaran utama yang diharapkan dapat mendorong pendaftaran hingga insiden tersebut diselesaikan. Banyak insiden yang tertunda hanya karena tidak ada yang tahu siapa yang bertanggung jawab atas layanan yang menyebabkan gangguan, atau mereka tidak dapat menghubungi orang yang tepat untuk membantu menyelesaikannya, jadi penting untuk selalu memperbarui daftar ini.

Orang-orang bergabung, berpindah-pindah, dan meninggalkan organisasi secara terus-menerus. Anda mungkin ingin mempertimbangkan untuk sekadar menyertakan peran, dan menautkan ke buku alamat global, atau solusi lain yang paling sesuai untuk organisasi Anda. Jika produk Anda hanya memerlukan dukungan selama jam kerja, maka memiliki alamat email tim atau yang serupa mungkin sudah cukup, tetapi untuk dukungan di luar jam kerja, Anda mungkin perlu menghubungi orang tertentu yang merupakan kontak dukungan yang ditunjuk, atau orang yang dituju oleh sistem pemantauan otomatis, untuk jangka waktu tertentu.

Untuk mencapai hal ini, sistem panggilan dapat digunakan, di mana anggota tim bergiliran memberikan dukungan di luar jam kerja. Untuk sistem ini, jadwal dibuat dan dipublikasikan, dan orang yang bertugas memastikan bahwa mereka tersedia di luar jam yang dipublikasikan tersebut. Sering kali, jadwal dengan informasi kontak ini ditautkan atau

dimasukkan ke dalam buku petunjuk, dan ketika orang lain perlu mengeskalasi kepada Anda, mereka dapat merujuk ke buku tersebut, atau sistem pemantauan dapat secara terprogram menanyakannya untuk mengetahui siapa yang harus dihubungi.

Pendekatan lain termasuk memiliki telepon fisik yang diteruskan seperti tongkat dengan nomor telepon yang ditetapkan, atau cukup memperbarui detail kontak di buku petunjuk dan sistem pemantauan di setiap serah terima. Untuk masalah yang rumit, mungkin perlu ada rencana eskalasi untuk melibatkan anggota tim lain, atau anggota organisasi yang lebih luas, jika situasinya sangat rumit dan kritis dan tidak dapat diselesaikan sendiri. Ini juga harus dilakukan, tetapi keputusan untuk melakukan eskalasi diserahkan kepada orang yang bertugas, bukan kepada pihak ketiga. Beberapa sistem panggilan otomatis juga memungkinkan eskalasi otomatis jika suatu insiden tidak diakui. Ini dapat berarti bahwa tim selalu siap sedia, tetapi eskalasi ini harus jarang terjadi.

14.2 FAKTOR MANUSIA DALAM ROTA ON-CALL

Banyak organisasi memperkenalkan rotasi on-call sebagai bagian dari perpindahan ke budaya DevOps, tetapi ini harus dilakukan dengan hati-hati, karena dapat mengubah sifat pekerjaan dan berdampak besar pada keseimbangan kehidupan kerja bagi sebuah tim. Paling tidak, orang yang on-call harus dapat memengaruhi pola shift yang diberikan kepada mereka. Berada on-call terlalu lama dapat menyebabkan kelelahan, dan menghabiskan sepanjang malam untuk memadamkan insiden langsung dan masih diharapkan untuk muncul untuk hari kerja pukul sembilan sampai lima biasanya tidak memungkinkan.

Banyak yurisdiksi memiliki aturan seputar jam kerja dan persyaratan istirahat minimum yang harus diperhitungkan saat merencanakan rotasi. Saat memperkenalkan sistem on-call, Anda mungkin juga perlu mempertimbangkan bagaimana kompensasi untuk anggota tim dapat berubah, karena mereka diharapkan untuk mengambil pekerjaan tambahan, dan fleksibilitas apa yang mungkin diperlukan untuk mendukung keseimbangan kehidupan kerja mereka.

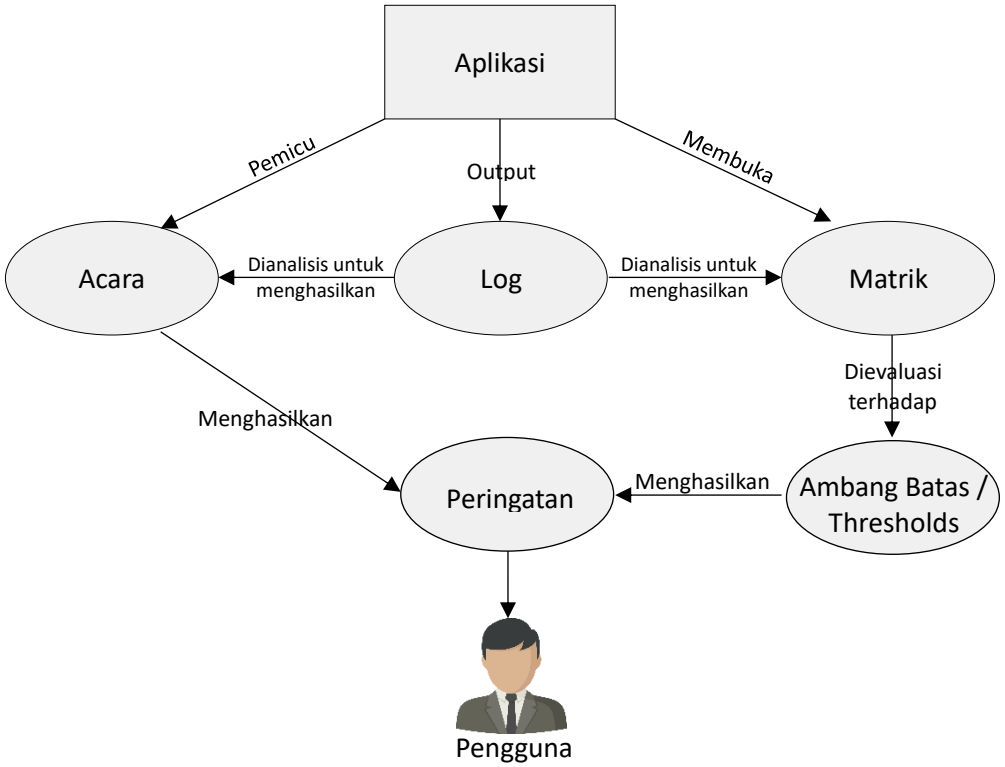
Pemantauan

Setiap produk yang memiliki pengguna akan dipantau secara default, tetapi jika cara terbaik untuk mengetahui apakah sistem Anda sedang tidak berfungsi adalah karena pelanggan Anda men-tweet tentang hal itu, berarti Anda memiliki masalah. Menerapkan pemantauan yang efektif akan memungkinkan Anda mengetahui masalah sebelum pelanggan mengetahuinya, dan menyelesaikannya lebih cepat. Yang lebih baik lagi adalah pemantauan yang dapat memberi tahu Anda tentang masalah sebelum berubah menjadi pemadaman total (misalnya, peningkatan waktu respons atau ruang disk yang rendah).

Pemantauan dapat memberi kita dua jenis data: kualitatif dan kuantitatif. Data kualitatif mudah dibaca dan ditafsirkan oleh manusia, tetapi lebih sulit bagi komputer. Kuantitatif mengacu pada angka: jumlah permintaan, beban CPU, dll. Ini adalah data yang sangat bagus dalam pemrosesan komputer. Dalam hal pemantauan, kita biasanya berbicara tentang log (kualitatif) dan metrik (kuantitatif). Terkadang log dapat sangat terstruktur, seperti log akses HTTP, dan ini sering diubah menjadi metrik, tetapi sering kali tidak terlalu terstruktur,

seperti traceback atau pesan log lainnya. Selama insiden, kedua jenis data tersebut penting.

Metrik dapat memberi tahu Anda bahwa ada yang salah dan memberi Anda petunjuk awal tentang di mana harus mencari, dan log dapat memberi Anda konteks lebih lanjut dan tingkat detail yang tinggi. Pemantauan tidak lengkap tanpa alarm atau peringatan. Menangkap banyak data dapat berguna untuk post-mortem atau analisis lainnya, tetapi untuk tujuan memantau sistem Anda dalam produksi, Anda memerlukan data untuk memberi tahu Anda sesuatu di tempat. Alarm ini biasanya disetel terhadap metrik yang ditangkap sistem pemantauan Anda, sehingga file log tersedia untuk memberi Anda wawasan yang lebih baik tentang sistem Anda.



Gambar 14.1 Menunjukkan Bagaimana Aplikasi Dapat Dipantau

Peringatan sering kali dipicu oleh aturan yang menetapkan ambang batas terhadap metrik, dan sering kali berisi tingkat keparahan. Misalnya, Anda dapat menetapkan aturan seperti "Aktifkan alarm untuk 'LowDiskSpace' dengan peringatan level saat ruang disk kosong kurang dari 5GB," selain tingkat keparahan yang berbeda terhadap ambang batas yang berbeda. Sering kali, peringatan ini dapat dikonfigurasi untuk dihapus secara otomatis saat aturan tidak lagi berlaku, dan banyak sistem peringatan menggunakan konsep "flapping," yang terjadi saat peringatan dimunculkan dan kemudian dihapus beberapa kali secara berurutan.

Sistem sering kali memiliki aturan notifikasi yang berbeda untuk alarm ini misalnya, pesan peringatan mungkin hanya mengirim email untuk ditangani selama jam kantor, tetapi tingkat keparahan yang lebih tinggi dapat menyebabkan pesan teks atau panggilan telepon otomatis dikirim. Gambar 14.1 menggambarkan contoh bagaimana aplikasi dapat dipantau. Saat mengatur peringatan, penting untuk mempertimbangkan kasus apa pun yang mana

ketiadaan data dapat mengindikasikan adanya masalah.

Menetapkan ambang batas untuk mengatakan "lebih dari lima kesalahan per menit" masuk akal, tetapi ambang batas seperti "melayani kurang dari 10 permintaan yang baik per detik" dapat membantu dalam mendiagnosis masalah jaringan atau penyeimbang beban yang menghentikan permintaan agar tidak sampai ke server Anda. Terkadang peringatan dapat dipicu secara langsung sebagai akibat dari suatu peristiwa. Misalnya, jika pemutus arus dipicu (lihat bab Mendesain Sistem untuk informasi lebih lanjut tentang pemutus arus), Anda mungkin ingin menandai peringatan secara langsung, daripada melalui metrik.

Metrik juga tersedia dalam berbagai bentuk dan ukuran. Sesuatu seperti penggunaan CPU atau ruang disk adalah nilai absolut yang dapat diambil sampelnya, dan ambang batas ditetapkan secara langsung. Yang lainnya hanyalah penghitung. Penghitung bertambah saat peristiwa tertentu terjadi (seperti jumlah permintaan atau kesalahan yang dilayani). Penghitung ini memerlukan pemrosesan agar dapat digunakan agar ambang batas dapat ditetapkan terhadapnya, biasanya menjadi kecepatan per detik atau per menit, atau bahkan mengukur laju perubahan.

Kumpulan data bahkan dapat lebih canggih dalam bentuk waktu respons yang dapat diproses, seperti rata-rata, untuk memberikan metrik yang lebih bermakna. Ada banyak alat yang tersedia untuk membantu Anda memantau sistem, mulai dari sistem yang dihosting dan dikelola seperti Datadog atau New Relic hingga sistem yang dihosting sendiri seperti Nagios, Sensu, atau Grafana. Beberapa mencoba melakukan semuanya, tetapi yang lain memisahkan pemantauan dan pemberitahuan.

Beberapa mencoba memberi Anda banyak metrik secara langsung, tetapi akan selalu ada metrik khusus domain yang ingin Anda tampilkan secara langsung, jadi saat membuat kode, penting untuk mempertimbangkan bagaimana Anda ingin menghubungkan pemantauan dan metrik di sekitar fungsi-fungsi utama. Ini mungkin termasuk komunikasi dengan dependensi (jumlah respons yang dibuat ke sistem back-end, kesalahan yang diterima, waktu respons, cache berhasil dan gagal), atau aturan bisnis lainnya, seperti ukuran rata-rata respons.

Mungkin tergoda untuk menetapkan metrik sistem pemantauan, KPI bisnis, dan analitik Anda untuk melacak hal yang sama, tetapi sebenarnya semua masalah ini berbeda. Yang pertama berkaitan dengan pemantauan kesehatan sistem Anda dari perspektif teknis, dan yang lainnya adalah tentang melacak seberapa baik aplikasi Anda memenuhi sasaran bisnis dan perilaku pengguna. Solusi yang sama mungkin tidak memenuhi ketiga kebutuhan yang berbeda. Catatan akhir tentang pencatatan log. Memisahkan log berdasarkan masalah akan membuat hidup Anda jauh lebih mudah saat terjadi insiden.

Mencoba menemukan kesalahan di tengah log akses yang sibuk itu sulit, jadi Anda harus mencoba menyimpan log aktivitas yang terjadi di sistem Anda (seperti log akses) terpisah dari pencatatan audit atau pencatatan kesalahan. Dalam suatu insiden, log kesalahan akan menjadi tempat pertama yang Anda tuju, jadi memastikan hanya informasi yang relevan yang tersisa akan membuat hidup Anda jauh lebih mudah. Pesan kesalahan yang Anda catat harus berisi informasi yang cukup untuk membantu Anda men-debug masalah.

Jika permintaan gagal, pastikan Anda mencatat informasi yang cukup tentang permintaan tersebut (seperti URL yang coba diakses), serta alasan kesalahan tersebut sehingga Anda dapat mencoba mereproduksi kondisi kesalahan, dan untuk memahami bagaimana kondisi kesalahan tersebut dapat menyebar melalui kode Anda. Bahasa Indonesia: Jika kesalahan yang sama dapat terjadi di beberapa titik dalam basis kode Anda, pastikan Anda menyertakan cara melacaknya kembali ke baris kode yang tepat di mana pesan log benar-benar ditulis, untuk membantu Anda memverifikasi bahwa cabang kode yang Anda harapkan untuk dieksekusi melakukannya.

Pencatatan traceback dari pengecualian (berhati-hatilah untuk tidak mengekspos ini ke pengguna akhir) bisa sangat berharga. Hal terakhir yang ingin Anda lakukan adalah harus membuat penerapan kode Anda selama insiden untuk menambahkan pencatatan yang cukup untuk mendiagnosis masalah. Namun, berhati-hatilah dengan apa yang Anda catat. Menambahkan pesan log di sekitar login atau validasi kata sandi dapat menyebabkan pencatatan kata sandi orang secara tidak sengaja dan menciptakan pelanggaran keamanan. Ini terutama berlaku saat mengaktifkan log mentah misalnya, untuk koneksi basis data atau koneksi web, tempat badan permintaan disimpan.

Ada banyak kerangka kerja pencatatan yang tersedia untuk sebagian besar bahasa. Paling tidak, Anda menginginkan kerangka kerja yang memungkinkan Anda menambahkan stempel waktu ke pesan, serta memperjelas di berkas log biasa tempat pesan log multi-baris berakhir dan pesan berikutnya dimulai. Ada juga alat yang bagus untuk membaca log, termasuk beberapa yang dapat menggabungkan pesan yang sama (atau serupa) bersama-sama dan memberikan hitungan, yang dapat membantu Anda memilah-milah pesan yang sibuk.

Namun, memiliki berkas log teks biasa di disk di tengah krisis dapat menyediakan akses cepat ke log yang dapat menyebabkan masalah dengan alat yang lebih rumit ini. Jika aplikasi Anda disebarkan di beberapa server, atau Anda menggunakan pola infrastruktur yang tidak dapat diubah, Anda juga perlu mempertimbangkan cara tertentu untuk menggabungkan log: mengirimkannya dari kotak Anda ke layanan pusat yang memungkinkan Anda melihat status seluruh aplikasi Anda (termasuk mencari tahu apakah suatu masalah terbatas pada satu server). Dalam kasus infrastruktur yang tidak dapat diubah, penggabungan diperlukan untuk memelihara log aplikasi Anda sehingga Anda dapat mendiagnosisnya di kemudian hari. Jika tidak, data tersebut akan hilang saat instance di-deploy atau dimuat ulang.

Menanggapi Insiden

Insiden selalu terjadi sebagai akibat dari perubahan perubahan kode, konfigurasi, atau basis data yang mendasarinya, atau permintaan oleh pengguna yang berniat jahat. Perubahan ini bisa sangat halus. Bisa jadi merupakan kombinasi tertentu dari permintaan lama yang mengungkap bug yang tidak pernah diuji, atau bahkan sekadar aliran waktu yang terus berlanjut. Perubahan tersebut mungkin tidak dilakukan oleh tim pengembangan, tetapi selalu ada akar penyebab insiden, meskipun mungkin tidak mungkin untuk mendiagnosis akar penyebabnya (misalnya, hard disk yang rusak).

Jika terjadi insiden, tugas Anda bukan hanya menyelesaikannya dan memulihkan

kehilangan layanan, tetapi juga memahami perubahan apa yang menyebabkan insiden tersebut agar tidak terjadi lagi (analisis akar penyebab). Insiden idealnya dimulai dengan tim yang diberi tahu saat sistem gagal. Jika Anda diberi tahu oleh pihak ketiga, seperti ketergantungan, atau oleh pengguna Anda, ini menyoroti celah dalam pemantauan Anda yang harus diselesaikan di masa mendatang.

Hal pertama yang harus dilakukan adalah dengan melihat buku petunjuk untuk mengetahui apakah ada langkah-langkah untuk mengatasinya, atau jika ada masalah yang diketahui atau kejadian sebelumnya, sehingga dapat menunjukkan tindakan yang dapat diterapkan. Saat mengikuti langkah-langkah tersebut, Anda harus mencatat apa yang telah Anda lakukan, mungkin dalam sistem obrolan yang tercatat atau pada sistem tiket. Ini akan memungkinkan Anda untuk melihat kembali apa yang telah Anda coba, dan jika Anda perlu meningkatkannya, untuk segera menyerahkan informasi tersebut.

Ini juga memungkinkan beberapa pemangku kepentingan untuk memantau kemajuan, dan di akhir insiden, bagi Anda untuk melihat kembali apa yang telah Anda lakukan dan menggunakannya untuk mendorong perbaikan di masa mendatang. Tentu saja, tindakan aktual yang diambil bergantung pada konteks, tetapi menggunakan alat yang telah disiapkan sebelumnya seperti buku petunjuk dan alat debugging apa pun yang mungkin telah Anda identifikasi sebagai kebutuhan dalam latihan kebakaran Anda diharapkan akan memungkinkan Anda untuk mengatasi insiden tersebut dalam jangka pendek, dan kemudian mengidentifikasi akar penyebab dan kekurangan apa pun dalam sistem Anda yang harus diatasi.

14.3 KESIMPULAN

Di banyak organisasi, tim dengan tumpukan penuh diberi wewenang untuk membangun dan menjalankan sistem mereka sendiri. Menjalankan sistem memerlukan keahlian yang berbeda dengan membangunnya, dan ini sering tercermin dalam struktur organisasi, dengan tim operasi dan tim pengembangan yang terpisah. Ketika tim pengembangan menjalankan sistem mereka sendiri, maka mereka harus menerapkan keahlian operasi tersebut sendiri, yang mengarah ke cara kerja yang dikenal sebagai DevOps (pengembang dalam operasi).

Ketika membangun sistem, Anda harus mempertimbangkan bagaimana Anda akan mendeteksi dan mendiagnosis masalah operasional apa pun. Cara yang paling umum untuk melakukannya adalah dengan mengeluarkan log dan metrik yang berguna yang dapat dikumpulkan dan dicari, dan menggunakan alat pemantauan yang menjalankan pemeriksaan terhadap sistem untuk menemukan kegagalan umum.

Ketika pemeriksaan gagal, maka peringatan dikirim ke tim, atau pengembang panggilan yang ditunjuk, yang menyebabkan mereka mulai menyelidiki kesalahan tersebut. Peringatan juga dapat dipicu ketika metrik melanggar ambang batas tertentu misalnya, jika jumlah kesalahan melonjak. Untuk memastikan bahwa suatu insiden dapat diselesaikan dengan cepat, latihan yang dikenal sebagai latihan kebakaran dapat dijalankan yang mensimulasikan kegagalan dengan cara yang terkendali untuk menanamkan kepercayaan diri dalam menyelesaikan masalah.

Prosedur ini untuk mengatasi masalah umum, atau petunjuk tentang cara men-debug atau apa arti peringatan tertentu, harus dicatat dalam buku petunjuk untuk aplikasi tertentu. Dengan prosedur ini, tim dapat menjalankan layanan mereka dengan percaya diri dan menangani insiden dengan tenang.

BAB 15
PEMBELAJARAN BERKELANJUTAN

Satu area yang membedakan pengembang tumpukan penuh modern yang bekerja di organisasi digital dengan perusahaan tradisional adalah bahwa produk yang mereka kerjakan akan terus berkembang sebagai respons terhadap dunia nyata dengan kecepatan tinggi, bukan sekadar sebagai respons terhadap persyaratan yang dipaksakan kepada mereka oleh organisasi. Tidak lagi cukup untuk membangun sesuatu yang memenuhi beberapa kriteria penerimaan lalu mendorongnya ke dunia; Anda dan tim Anda harus memeriksa apakah setiap perubahan yang Anda buat benar-benar memberikan dampak yang Anda inginkan.

Mendapatkan posisi untuk melakukan ini bisa jadi sulit kecuali tim Anda benar-benar bekerja sebagai bagian inti dari organisasi Anda. Setiap fitur baru yang Anda bangun atau perubahan yang Anda buat juga mengharuskan Anda untuk memahami mengapa perubahan itu dilakukan apakah untuk meningkatkan pendaftaran atau penjualan, atau untuk memenuhi beberapa kebutuhan bisnis lainnya? Setelah Anda memahami motivasi mendasar ini, maka mengirimkan fitur tidak lagi cukup, jika fitur itu tidak dapat memenuhi penyebab mendasar ini.

15.1 MENGUMPULKAN ANALISIS

Banyak organisasi kini mengandalkan data untuk membuat keputusan yang tepat tentang strategi, tetapi sebelum data dapat diinterogasi, data tersebut harus dikumpulkan dan disimpan secara terpusat. Istilah populer "big data" terkadang digunakan untuk membicarakan hal ini, tetapi ini biasanya berarti mengumpulkan sejumlah besar data tidak terstruktur dan menganalisisnya untuk mendapatkan wawasan. Bagi banyak organisasi, memahami setiap bagian data bisa sangat bermanfaat.

Ini bisa berupa angka sederhana, seperti nilai penjualan, dan dapat dicatat oleh proses bisnis standar, lalu dikumpulkan dan ditampilkan kepada para pemangku kepentingan. Jenis analisis lain diperoleh dari data tentang cara pengguna berinteraksi dengan situs web Anda: jalur yang mereka ambil ke halaman web tertentu, berapa lama mereka menghabiskan waktu di sana saat mereka sampai di sana, apakah ada kesalahan yang dibuat saat mengisi formulir, dll. Web sangat bermanfaat dalam memungkinkan Anda mengumpulkan data ini, tetapi ada banyak penyalahgunaannya (misalnya, "God View" Uber) yang membuat banyak orang waspada terhadap skrip analitik dan pelacakan.

Di banyak negara, hukum privasi tidak lagi mengizinkan Anda untuk mengumpulkan data ini kecuali pengguna menyetujuinya, dan saat menggunakan analitik, Anda mungkin tergoda untuk sekadar mengumpulkan semuanya lalu memutuskan cara menggunakannya nanti. Pendekatan ini mungkin tampak lebih sederhana secara teknis dan memberi Anda lebih banyak fleksibilitas, tetapi sering kali melanggar hukum ini, karena data tersebut kemudian menjadi "identitas pribadi" dan mungkin berisi informasi yang sangat sensitif. Ini berarti Anda harus melindungi dan mengelola data ini dengan cara yang jauh lebih hati-hati.

Untuk menghindarinya, Anda harus mempertimbangkan data apa yang ingin Anda kumpulkan terlebih dahulu. Pada akhirnya, cara Anda ingin mengumpulkan sebagian besar data adalah dengan menggabungkannya menjadi penghitung, seperti "berapa banyak orang yang mengeklik tombol ini?" Memiliki satu penghitung mungkin berguna, tetapi sering kali Anda ingin mengajukan pertanyaan seperti "berapa banyak orang yang mengeklik tombol ini kemarin?", "berapa banyak orang yang mengeklik tombol ini di ponsel?", atau "berapa banyak orang yang mengeklik tombol ini lalu akhirnya membeli sesuatu?".

Menerapkan penghitung sederhana saja tidak cukup, jadi Anda sering kali harus memikirkan "segmentasi". Dalam kasus ini, metrik tingkat tinggi adalah keranjang yang berisi sejumlah metrik lain yang memungkinkan Anda mengirisnya menjadi bagian-bagian lain yang bermakna. Misalnya, alih-alih menambah penghitung, Anda menambahkan rekaman ke keranjang yang berisi sejumlah pasangan kunci/nilai. Katakanlah Anda memiliki keranjang yang disebut "product_button_clicked." Kemudian, setiap kali seseorang mengklik tombol itu, Anda dapat menambahkan entri seperti:

```
product=12345;
browser=Safari;
device=Windows_PC;
time=2017-12-19T09:00:17Z
location=ManchesterUK
```

Anda harus berhati-hati untuk tidak mengumpulkan informasi yang pada akhirnya dapat mengarah kembali ke pengguna, karena ini menjadi informasi yang dapat diidentifikasi secara pribadi, tetapi label ini memungkinkan Anda untuk mengelompokkan jumlah total item dalam keranjang dengan cara yang berbeda untuk mengajukan pertanyaan yang menarik. Sebagian besar perangkat analitik akan mengumpulkan beberapa hal ini untuk Anda (terutama waktu, lokasi, dan info peramban), dan yang lainnya mengharuskan Anda untuk menambahkan segmen sendiri.

Namun, ini tidak membantu menjawab pertanyaan terakhir, yang sering kali seperti, "berapa banyak orang yang mengklik tombol itu dan kemudian membeli sesuatu?". Untuk mencapai ini, sebagian besar perangkat lunak analitik juga akan menambahkan "ID sesi" ke keranjang, yang kemudian memungkinkan Anda untuk melihat tindakan mana yang terjadi dalam sesi yang sama. Ini bisa berbahaya, karena mengkorelasikan tindakan di seluruh sesi tunggal dapat memungkinkan Anda untuk mengidentifikasi pengguna individu, sehingga kehilangan anonimisasi yang dicapai dengan menempatkan tindakan ke dalam keranjang.

Banyak orang menerima ini sebagai risiko dan menyimpan semua aktivitas, tetapi yang lain hanya menyimpan data sesi untuk jangka waktu tertentu hingga setelah sedikit aktivitas terakhir dalam sesi itu, dan kemudian melihat data sesi untuk menghasilkan jawaban atas pertanyaan-pertanyaan itu sebelum membuangnya. Pendekatan terakhir adalah dengan menetapkan tanda seperti "clicked_button=true" dalam sesi, lalu menyimpannya sebagai segmen tambahan nanti dalam tindakan yang sesuai, yang menghindari pengambilan data sesi apa pun.

Hal terakhir yang perlu dipertimbangkan adalah data interaksi apa yang akan dikumpulkan. Hal-hal sederhana seperti "membuka halaman" dan "mengklik tombol" (atau melakukan interaksi lain) berguna dan terjadi sebagai respons terhadap tindakan langsung pengguna. Hal-hal lain mungkin terjadi secara lebih implisit (seperti mencatat waktu yang dihabiskan di halaman, apakah pengguna mencapai akhir artikel, atau jika halaman dibiarkan terbuka tanpa interaksi untuk jangka waktu yang lama).

Untuk menentukan apa yang berguna, penting untuk berbicara dengan pemangku kepentingan dan praktisi pengalaman pengguna di tim Anda untuk mengetahui apa yang mereka butuhkan. Mekanisme yang sama yang digunakan untuk mengumpulkan analitik dapat digunakan di luar kasus penggunaan ini misalnya, untuk mengembangkan sistem personalisasi dan rekomendasi tetapi kasus penggunaan tersebut tidak dibahas di sini, dan memiliki serangkaian implikasi etika dan hukumnya sendiri. Anda harus selalu menjelaskan kepada pengguna informasi apa yang Anda kumpulkan dan mengapa—jika tidak, mereka mungkin berasumsi yang terburuk.

15.2 EKSPERIMEN

Selain merefleksikan performa situs Anda menggunakan analitik, Anda juga dapat bereksperimen secara langsung dengan pengguna dengan memberi mereka versi berbeda dari halaman yang sama dan melihat bagaimana mereka merespons. Bereksperimen pada orang dapat menimbulkan risiko moral, tetapi itu terjadi terus-menerus, jadi bagaimana cara melakukannya secara etis? Pertanyaan intinya adalah menanyakan apakah salah satu varian dapat mengakibatkan kerugian bagi pengguna atau tidak.

Misalnya, menguji ukuran dan penempatan tombol mungkin tidak akan menghasilkan kerugian, tetapi menerapkan "pola gelap" untuk menarik pengguna agar menghabiskan lebih banyak uang, atau menguji struktur harga A/B di mana beberapa pengguna mungkin membayar lebih banyak daripada yang lain berdasarkan segmen tempat mereka ditempatkan, dapat menyebabkan kerugian dan harus dihindari.

Skenario lain kurang jelas, dan persetujuan yang diinformasikan dapat berguna, mungkin dengan mengizinkan pengguna untuk ikut serta dalam "uji coba beta" dan menjelaskan bahwa mereka akan berpartisipasi dalam eksperimen, tetapi kemudian mengabaikan sebagian besar pengguna. Eksperimen dapat berguna saat Anda menguji coba fitur situs baru dan Anda tidak yakin bagaimana mencapai tujuan tertentu.

Pengujian A/B adalah eksperimen yang cukup sederhana: Anda mulai dengan memberikan 90% pengguna versi halaman yang ada ("kontrol"), lalu 5% varian "A", yang mungkin memiliki fitur baru, dan 5% varian "B", yang mungkin memiliki desain atau alur kerja yang berbeda untuk fitur tersebut. Anda menjalankan eksperimen selama jangka waktu tertentu dan menambahkan analitik yang sesuai (ingat untuk melakukan segmentasi apakah pengguna berada dalam grup kontrol, A, atau B).

Lalu bandingkan hasilnya untuk melihat varian A atau B mana yang lebih baik (atau jika fitur baru tersebut sebenarnya tidak berguna atau kurang membantu daripada tidak ada sama sekali, dengan membandingkannya dengan kontrol). Dengan pengujian A/B, Anda hanya

dapat menguji satu perubahan dalam satu waktu, yang dapat berlangsung lambat. Pengujian multivariat telah berkembang dari pengujian ini yang memungkinkan Anda menjalankan beberapa pengujian A/B secara paralel, tetapi analisis hasilnya lebih rumit untuk memisahkan efek yang mungkin dimiliki satu pengujian terhadap pengujian lainnya.

15.3 MENGANALISIS HASIL

Tidaklah cukup hanya membingkai pembaruan Anda dalam hal perubahan mendasar yang ingin Anda buat, tetapi Anda juga harus memiliki cara untuk mengukur perubahan tersebut dengan cara yang berarti. Meskipun sering disalahgunakan, indikator kinerja utama (KPI) dapat menjadi cara yang berguna untuk mengukurnya. KPI tidak akan berhasil jika mengukur hal-hal yang mudah diukur tetapi sebenarnya bukan tujuan yang mendasarinya.

Misalnya, KPI mungkin adalah jumlah pengunjung ke suatu halaman, karena ini mudah diukur, tetapi jika banyak dari pengunjung tersebut meninggalkan halaman tanpa menyelesaikan tindakan yang berarti, maka dengan menjadikan tampilan halaman sebagai KPI, ini dapat membuat tim fokus untuk mendatangkan orang ke halaman, tetapi melupakan apa yang mereka lakukan setelah benar-benar berada di sana, yang merupakan hal yang sebenarnya penting. Seperti biasa, KPI yang tepat bergantung pada konteks, dan dapat memerlukan pemikiran yang mendalam untuk memastikan Anda melakukannya dengan benar.

Dalam beberapa kasus, KPI sulit diukur secara kuantitatif secara langsung, tetapi beberapa ukuran kualitatif dapat diterapkan sebagai gantinya. Namun, ukuran kuantitatif sering kali diinginkan, karena mudah dikumpulkan dan tampaknya mudah dipahami. Angka utama dapat mudah dipahami, tetapi angka sederhana dapat memiliki banyak makna yang dapat mengarah pada interpretasi yang naif dan salah. Proses memperoleh angka dapat menjadi rumit, dan penting untuk memahami pertentangan tersebut.

Teorema Bayes

Teorema Bayes adalah teori dasar dalam probabilitas, dan Anda mungkin pernah menemukannya sebelumnya. Jika dinyatakan dalam bentuk persamaan, maka rumusnya adalah sebagai berikut

$$\boldsymbol{P(A\backslash B) = \frac{P(B\backslash A)P(A)}{P(B)}}$$

Ini menyatakan probabilitas kejadian A terjadi jika beberapa kejadian B lainnya benar ("probabilitas bersyarat"), berdasarkan probabilitas kebalikannya dan probabilitas masing-masing kejadian tersebut terjadi secara independen satu sama lain. Ini menyoroti bagaimana beberapa angka utama dapat memberikan hasil yang menyesatkan. Jika tes untuk kanker 99% akurat, dan kanker ini terjadi pada 0,1% populasi, maka melalui teorema Bayes, jika tes ini mendeteksi Anda menderita kanker, itu berarti Anda hanya memiliki peluang 9% untuk benar-benar menderita kanker.

Ini mungkin tampak berlawanan dengan intuisi karena akurasi 99%, tetapi karena kanker tersebut relatif jarang, kelangkaan ini membatalkan akurasi untuk memberi Anda

"probabilitas bersyarat" sebesar 9%. Buku ini bukan tempat untuk membahas teori statistik, dan jika Anda beruntung, Anda akan memiliki akses ke orang-orang yang memiliki pemahaman yang baik tentang analitik dan data (kadang-kadang disebut ilmuwan data), atau alat yang baik, untuk membantu Anda memahaminya. Konsep penting untuk dipahami disebut interval kepercayaan, yang menyatakan persentase keyakinan bahwa suatu nilai berada dalam rentang yang ditentukan.

Yang penting untuk diingat adalah untuk tidak mempercayai angka utama. Misalnya, dalam pengujian A/B, jika hasil A adalah konversi 52% dan hasil B adalah konversi 48%, maka sebenarnya kedua hasil tersebut bisa saja sama, atau hasil B sebenarnya bisa berkinerja lebih baik, tergantung pada margin kesalahan, yang menyebabkan interval kepercayaan tumpang tindih. Semakin banyak data yang Anda miliki, semakin kecil margin kesalahan Anda, tetapi jika, katakanlah, margin kesalahan adalah ±3%, maka hasil untuk A adalah (dengan kepastian 99%) antara 45-51%, dan untuk B, antara 49-55%.

Jadi ada kemungkinan A bisa mencapai 51% dan B bisa mencapai 49%. Saat melihat analitik, meskipun data numerik ini dapat membantu memberi Anda gambaran tentang tren tingkat tinggi, data tersebut tidak akan sering memberi tahu Anda mengapa tren tersebut terjadi. Untuk itu, Anda perlu menggunakan data kualitatif. Data kualitatif ini diperoleh melalui pemahaman terhadap pelanggan Anda, bukan sekadar mengamati metrik agregat. Pengujian desain oleh pengguna adalah salah satu cara untuk mencapainya (Anda dapat menguji desain yang dibuat oleh pengguna), serta memiliki panel dan kelompok fokus yang memungkinkan Anda mengajukan pertanyaan secara langsung.

Alternatif lain adalah dengan mengambil analitik untuk setiap metrik yang memungkinkan dan menyimpannya dengan cara yang memungkinkan Anda melihat semua analitik dari sesi individual dan menyelami setiap sesi secara individual, tetapi jika dilakukan secara massal, hal ini melanggar privasi pengguna individual, dan merupakan tingkat pengamatan yang harus disetujui secara eksplisit oleh pengguna (inilah sebabnya pengujian pengguna merupakan tempat yang baik untuk melakukannya).

15.4 PENGEMBANGAN BERBASIS HIPOTESIS

Pengembangan berbasis hipotesis telah diusulkan sebagai cara alternatif untuk mengekspresikan pekerjaan yang harus dilakukan dalam cerita pengguna, dengan mengungkapkannya dengan cara yang familiar bagi siswa sains sekolah menengah. Tidak seperti cerita pengguna, yang mengekspresikan perubahan dari perspektif pengguna, hipotesis melangkah lebih jauh dan mengekspresikan perubahan dalam bentuk pertanyaan. Barry O'Reilly mengusulkan bentuk berikut:

- Kami yakin <perubahan ini>
- Akan menghasilkan <beberapa hasil>
- Dan kami akan yakin untuk melanjutkan ketika <dampak yang terukur>

Misalnya, untuk halaman arahan pemasaran:

- Kami yakin bahwa menambahkan pendaftaran milis ke halaman arahan produk
- Akan menghasilkan peningkatan jumlah pendaftaran milis

- Dan kami akan yakin untuk melanjutkan ketika jumlah pendaftaran milis dalam seminggu adalah 15% lebih tinggi dari jumlah pendaftaran yang umum setelah tujuh hari

Ini berarti bahwa fitur Anda tidak selesai setelah dikirim, tetapi Anda harus meninjaunya kembali tujuh hari setelahnya untuk memastikan bahwa fitur tersebut telah memberikan dampak yang diinginkan, dan kemudian memutar kembali perubahan tersebut, meninjau ulang desain, atau mengajukan hipotesis perubahan baru untuk menguji agar memiliki keuntungan yang sama.

Satu tantangan dengan desain yang digerakkan oleh hipotesis adalah mengidentifikasi perubahan yang berarti yang ingin Anda terapkan, daripada sekadar menyebutkan sesuatu yang mungkin mudah diukur. Misalnya, menempatkan beberapa konten di belakang tanda masuk dapat meningkatkan metrik tanda masuk Anda, tetapi apakah itu benar-benar ukuran utama yang Anda pedulikan, atau apakah itu sesuatu yang lebih inti dari dasar-dasar organisasi Anda?

15.5 KESIMPULAN

Pekerjaan tim pengembangan tumpukan penuh tidak selesai setelah fitur dikirimkan. Sebaliknya, Anda harus memantau fitur tersebut untuk memastikannya berfungsi seperti yang Anda harapkan, serta terus belajar dari cara situs Anda berinteraksi dengan pengguna untuk mengidentifikasi peningkatan yang dapat diterapkan guna terus meningkatkan produk Anda. Mengumpulkan analitik, dan berhati-hati dalam melakukannya, dapat memberikan wawasan yang sangat berharga tentang bagaimana produk Anda benar-benar digunakan oleh pengguna akhir, dan bagaimana penggunaan tersebut dapat berubah seiring waktu.

Jika Anda kurang yakin tentang suatu perubahan, Anda juga dapat menjalankan eksperimen di situs yang sedang berjalan untuk melihat bagaimana berbagai varian berperilaku dalam praktik. Kehati-hatian harus dilakukan saat menganalisis data ini agar tidak terjebak dalam perangkap statistik yang secara keliru menarik kesimpulan. Pengembangan yang didorong oleh hipotesis adalah cara memformalkan pendekatan ini untuk ditinjau, dengan menyatakan perubahan dalam bentuk hipotesis yang akan diuji, yang menyoroti bahwa pekerjaan belum selesai hingga hipotesis ini diuji.

DAFTAR PUSTAKA

Abouzaid, O. (2020). *Modern web development with React: A comprehensive guide*. Packt Publishing.

Ahmadi, S., & Naghibi, F. (2019). The role of web frameworks in modern web development. *International Journal of Computer Applications, 179*(10), 15-20. https://doi.org/10.5120/ijca2019919255

Allen, P. (2021). *Learning modern web development: From HTML5 to progressive web apps*. O'Reilly Media.

Berman, D. (2020). *Building modern web applications with Node.js*. Apress.

Brown, C., & Green, H. (2021). *Mastering JavaScript for web development*. Wiley.

Buxton, B. (2019). *Designing for interaction: Creating innovative applications*. New Riders.

Card, S. K., Mackinlay, J. D., & Shneiderman, B. (2020). *Readings in information visualization: Using vision to think*. Morgan Kaufmann.

Cascante, G. R. (2022). *Vue.js for web development: From beginner to pro*. Packt Publishing.

Choi, D., & Lim, J. (2021). Progressive web apps: The future of modern web development. *Journal of Web Development, 18*(2), 24-30. https://doi.org/10.1080/xyz_web_2021

Crust, J. (2020). *Building websites with HTML5, CSS3, and JavaScript: Responsive design fundamentals*. O'Reilly Media.

DeMers, J. (2018). *Web development with Laravel: A modern approach*. Packt Publishing.

Duckett, J. (2020). *HTML and CSS: Design and build websites*. Wiley.

Fowler, M. (2021). *Refactoring: Improving the design of existing code*. Addison-Wesley Professional.

Free, S. (2020). *The modern web: A beginner's guide to HTML, CSS, and JavaScript*. No Starch Press.

Friedman, L. (2021). *React 16 in action*. Manning Publications.

Ghosh, S., & Ray, S. (2022). Responsive web design: Best practices for modern websites. *Journal of Web Design, 14*(4), 12-18. https://doi.org/10.1234/jwd.2022.034

Green, T., & Lewis, J. (2019). *Node.js design patterns*. Packt Publishing.

Hassan, A., & Khan, M. (2021). Trends and challenges in modern web development frameworks. *International Journal of Web Development and Design, 7*(1), 43-50. https://doi.org/10.4234/ijwdd.2021.234

Haverbeke, M. (2020). *Eloquent JavaScript: A modern introduction to programming*. No Starch Press.

He, D., & Zhang, Y. (2021). Web performance optimization in modern web development. *Web Technologies Journal, 3*(2), 34-39. https://doi.org/10.1145/wt_optimization.2021

Heller, M. (2021). *Modern web development with Angular*. O'Reilly Media.

Ho, P. (2020). *Web development for beginners: Understanding modern frameworks*. Packt Publishing.

Hunt, A., & Thomas, D. (2022). *The Pragmatic Programmer: Your journey to mastery*. Addison-Wesley Professional.

Jackson, P., & Johnson, W. (2019). *Learning React: A hands-on guide to building modern web applications*. O'Reilly Media.

Jackson, R. (2020). *Progressive web apps: The new frontier of web development*. Smashing Magazine.

Karp, G., & Bender, T. (2021). *Modern web development with Svelte*. Packt Publishing.

Katz, S., & Lee, J. (2021). *Mastering frontend web development*. Wiley.

Krug, S. (2021). *Don't make me think: A common sense approach to web usability*. New Riders.

Larson, D. (2020). *JavaScript: The definitive guide*. O'Reilly Media.

Lee, M. (2021). *Building modern web apps with React and Redux*. Apress.

Lewis, C. (2021). *Practical guide to web development with Vue.js*. Packt Publishing.

Li, X., & Liu, Y. (2020). The evolution of modern web development technologies. *Journal of Software Engineering, 34*(2), 100-105. https://doi.org/10.1287/jse.2020.034

Meier, T. (2022). *Web development with React, Node, and Express*. Apress.

Moncrieff, S. (2020). *Web design with HTML, CSS, and JavaScript*. Pearson.

Myers, D., & O'Hara, S. (2021). *Introduction to modern web development using Vue.js*. Packt Publishing.

Nguyen, A., & Tran, P. (2021). Modern web development practices and technologies. *International Journal of Computer Science and Web Engineering, 14*(1), 44-49. https://doi.org/10.1016/ijcswe.2021.100

Pacheco, D. (2020). *Mastering JavaScript design patterns*. Packt Publishing.

Perez, A. (2021). *Designing modern web applications*. O'Reilly Media.

Pilgrim, M. (2021). *Learning jQuery: A hands-on guide to building modern web applications*. O'Reilly Media.

Preston, B. (2019). *Web development with Angular: Building modern applications*. O'Reilly Media.

Reynolds, R. (2020). *Building real-time web applications with WebSockets and Node.js*. Packt Publishing.

Rockwell, J. (2021). *HTML5 and CSS3: The complete guide*. Wiley.

Roubtsov, I. (2020). *Building progressive web apps: A practical guide*. O'Reilly Media.

Sadler, R., & Richards, J. (2021). *JavaScript and the web: Advanced techniques*. Wiley.

Sampson, M. (2022). *Understanding CSS Grid Layout*. Smashing Magazine.

Sargent, S. (2021). *Responsive web design with HTML5 and CSS3*. Pearson.

Shaw, K., & Wilson, T. (2020). *Mastering React: An advanced guide to building modern applications*. O'Reilly Media.

Smith, R. (2020). *Building websites with CSS and HTML5*. Packt Publishing.

Stevens, B. (2022). *Designing modern websites with modern tools*. Wiley.

Taylor, C., & Hardy, L. (2021). *Building modern front-end applications with Vue.js*. Apress.

Tilley, C., & Green, R. (2020). *The essentials of web development*. Wiley.

Tran, H., & Yang, X. (2021). *The modern web developer's toolkit*. Apress.

Turner, J. (2020). *Modern web performance in practice*. Smashing Magazine.

Wicks, D. (2021). *JavaScript frameworks for modern web development*. Packt Publishing.

Williams, J. (2022). *Designing interactive websites with JavaScript and CSS3*. Pearson.

Wilson, K. (2021). *Web development with JavaScript and Node.js: Master the essentials*. O'Reilly Media.

Wong, J., & Lee, T. (2021). *The future of the web: Trends and technologies in web development*. Springer.

Yang, L. (2021). *Creating scalable web applications with Node.js and MongoDB*. Apress.

Young, S. (2020). *Modern web design and development fundamentals*. Wiley.

Zeng, J. (2021). *Building mobile-first web applications*. O'Reilly Media.

Panduan Pengembang Web Modern

Dr. Budi Raharjo, S.Kom, M.Kom, MM.

BIODATA PENULIS

Dr. Budi Raharjo, S.Kom, M.Kom, MM lahir di Semarang, tanggal 22 Februari 1985. Beliau adalah Alumni dari Universitas Bina Nusantara (BINUS University) Jakarta dan juga alumni Universitas Kristen Satya wacana (UKSW) Salatiga. Dr. Budi Raharjo telah menjadi Dosen pada Universitas STEKOM pada mata kuliah Kepemimpinan (Leadership), mata kuliah Pengantar Akuntansi, Manajemen Proses, Manajemen Akuntansi dan Manajemen Resiko Bisnis. Selain sebagai dosen Universitas STEKOM, Dr. Budi Raharjo, M.Kom, MM juga mempunyai bisnis sendiri dalam bidang perhotelan dan juga sebagai wirausaha dalam bidang pemasok unggas (ayam) beku, ke berbagai kota besar, khususnya Jakarta dan sekitarnya.

Pengalaman beliau berwirausaha menjadi bekal utama dalam penulisan buku ajar yang diterbitkan oleh Yayasan Prima Agus Teknik (YPAT) Semarang. Oleh sebab itu bukunya berisi langkah langkah praktis yang mudah diikuti oleh para mahasiswa, saat mahasiswa mengikuti proses perkuliahan pada Universitas Sains dan Teknologi Komputer (Universitas STEKOM). Memiliki Jabatan Akademik Lektor 300 dan Menjabat sebagai Wakil Rektor 1 bidang (Akademik) di kampus Universitas STEKOM Semarang.

YAYASAN PRIMA AGUS TEKNIK

PENERBIT :
YAYASAN PRIMA AGUS TEKNIK
Jl. Majapahit No. 605 Semarang
Telp. (024) 6723456. Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

ISBN 978-623-8642-67-0 (PDF)
9 786238 642670