YAYASAN PRIMA AGUS TEKNIK

Aplikasi Web dengan REACT

Dr. Joseph Teguh Santoso, S.Kom M.Kom.

Aplikasi Web dengan REACT

Dr. Joseph Teguh Santoso, S.Kom M.Kom.

BIODATA PENULIS

Dr. Joseph Teguh Santoso, M.Kom adalah pemimpin yang visioner dan praktisi industri berpengalaman, yang menjabat sebagai Rektor Universitas Sains dan Teknologi Komputer (Universitas STEKOM), salah satu universitas terkemuka di Jawa Tengah, Indonesia. Dengan pengalaman lebih dari 13 tahun di dunia bisnis dan praktisi industri di China, beliau membawa perspektif global dan inovasi yang signifikan ke dalam dunia akademis.Sebagai seorang entrepreneur, penulis adalah pencipta TopLoker.com, sebuah platform inovatif yang merevolusi cara mencari dan menawarkan pekerjaan. TopLoker.com adalah portal lowongan bursa kerja terbesar di Indonesia, khusus untuk pendidikan SMA/SMK sederajat. TopLoker.com telah mendapatkan penghargaan sebagai juara 1 Startup4industry 2022 oleh Kementerian Perindustrian Republik Indonesia. Kontribusi Dr. Joseph dalam menyediakan akses pekerjaan yang luas bagi lulusan SMA/SMK telah membantu banyak individu menemukan peluang kerja yang sesuai dengan keahlian mereka. Selain itu, Dr. Joseph Teguh Santoso, M.Kom adalah pendiri dari dua organisasi yaitu (1) organisasi guru/pendidik PTIC (Perkumpulan Teacherpreneur Indonesia Cerdas) yang bertujuan untuk meningkatkan kualitas pendidikan dan kesejahteraan guru/pendidik dengan wawasan entrepreneurship, serta (2) organisasi industri PERKIVI (Perkumpulan Komunitas Industri dan Vokasi Indonesia) yang berfokus pada pengembangan link and match antara industri dan dunia pendidikan. Sebagai Rektor, Dr. Joseph Teguh Santoso, M.Kom memiliki kepemimpinan yang berorientasi pada hasil, dan berkomitmen untuk mendorong kemajuan Universitas Sains dan Teknologi Komputer (Universitas STEKOM). Saat ini Universitas STEKOM telah mengalami transformasi positif dalam peningkatan kualitas pendidikan, perluasan fasilitas, serta penguatan kemitraan Perguruan Tinggi Nasional dan Internasional. Beliau memprioritaskan pengembangan sumber daya manusia dan penelitian, serta memastikan bahwa universitas berada di garis depan dalam inovasi dan teknologi untuk mencapai tujuan akhir, yaitu lulusan yang mampu bekerja dan sukses setelah lulus.Dr. Joseph Teguh Santoso, M.Kom sering diundang sebagai pembicara di berbagai konferensi nasional maupun internasional dan telah menerima berbagai penghargaan atas dedikasinya dalam bidang pendidikan, industri, dan kewirausahaan.

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

Aplikasi Web dengan React

Penulis :
Dr. Joseph Teguh Santoso, S.Kom., M.Kom

ISBN :

Editor :
Dr. Ir. Agus Wibowo, M.Kom, M.Si, MM.

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

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. 08122925000
Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

Distributor Tunggal :
Universitas STEKOM
Jl. Majapahit no 605 Semarang
Telp. 08122925000
Fax. 024-6710144
Email : info@stekom.ac.id

KATA PENGANTAR

Puji dan syukur kami panjatkan ke hadirat Tuhan Yang Maha Esa atas rahmat dan karunia-Nya, sehingga buku yang berjudul "***Aplikasi Web dengan React***" dapat diselesaikan dengan baik. Buku ini ditulis dengan tujuan memberikan pemahaman mendalam mengenai konsep-konsep dasar React, termasuk JSX, komponen, state, props, dan lifecycle methods serta membantu pembaca memahami arsitektur aplikasi berbasis React, termasuk penggunaan hook dan pengelolaan state secara global menggunakan Context API maupun pustaka eksternal seperti Redux. Selain itu buku ini juga dirancang untuk memperkenalkan praktik terbaik dalam pengembangan aplikasi web modern, termasuk integrasi dengan API, optimisasi performa, dan pengujian dan menyediakan panduan praktis melalui studi kasus dan proyek aplikasi nyata yang dapat langsung diimplementasikan supaya menjadi manfaat untuk pengembang perangkat lunak, mahasiswa teknologi informasi, dan siapa saja yang ingin mempelajari React.

React adalah pustaka JavaScript yang dirancang untuk membangun antarmuka pengguna yang responsif, modular, dan dapat digunakan kembali. Dalam buku ini, kami memandu pembaca melalui berbagai konsep penting, mulai dari dasar-dasar React seperti JSX, komponen, dan pengelolaan state, hingga teknik tingkat lanjut seperti optimasi performa, pengelolaan data, dan penggunaan React Router untuk membuat aplikasi halaman tunggal (Single-Page Applications). Buku ini juga memberikan wawasan tentang bagaimana React dapat diintegrasikan dengan teknologi modern lainnya, seperti GraphQL dan TypeScript, untuk menghasilkan aplikasi yang lebih fleksibel dan kuat.

Dalam buku ini, Anda akan menemukan 17 bab yang mencakup berbagai aspek penting dari React dan React Native. Berikut adalah beberapa sorotan dari isi buku: Bab pertama memberikan pemahaman dasar tentang apa itu React dan bagaimana cara kerjanya. Membangun Aplikasi Pertama, Di sini, Anda akan belajar bagaimana membangun aplikasi sederhana sebagai langkah awal. Materi selanjutnya akan memberikan penjelasan tentang komponen dalam React, memahami komponen adalah kunci untuk mengembangkan aplikasi yang efisien. Pokok bahasan selanjutnya adalah penataan gaya dengan mempelajari cara menata konten menggunakan CSS dan teknik styling lainnya. Materi selanjutnya adalah membuat Aplikasi daftar tugas dalam sebuah proyek praktis untuk menerapkan semua yang telah dipelajari. Buku ini juga menyediakan wawasan tentang bagaimana mengembangkan aplikasi mobile yang menarik.

Penulis berharap buku ini dapat menjadi sumber daya yang berguna bagi Anda dalam perjalanan belajar mengembangkan aplikasi menggunakan React dan React Native. Dengan pendekatan yang praktis dan mudah diikuti, kami yakin Anda akan dapat menguasai keterampilan baru ini dan menerapkannya dalam proyek nyata.

Terima kasih telah memilih buku ini sebagai panduan Anda. Selamat belajar dan selamat berkarya! Semoga kata pengantar ini dapat memberikan gambaran yang jelas tentang isi buku serta memotivasi pembaca untuk menjelajahi dunia React dan React Native.

Semarang, Januari 2025
Penulis

Dr. Joseph Teguh Santoso, S.Kom M.Kom.

DAFTAR ISI

BAB 1
MEMPERKENALKAN REACT

Jika kita mengabaikan sejenak fakta bahwa aplikasi web saat ini terlihat dan terasa lebih bagus daripada sebelumnya, ada sesuatu yang lebih mendasar yang telah berubah. Cara kita merancang dan membangun aplikasi web kini sangat berbeda. Untuk menyoroti hal ini, mari kita lihat aplikasi yang ditunjukkan pada Gambar 1.1.

Gambar 1.1 Sebuah Aplikasi

Aplikasi ini adalah peramban katalog sederhana untuk sesuatu. Seperti aplikasi sejenis lainnya, Anda memiliki serangkaian halaman biasa yang berkisar di sekitar halaman beranda, halaman hasil pencarian, halaman detail, dan seterusnya. Di bagian berikut, mari kita lihat dua pendekatan yang kita miliki untuk membangun aplikasi ini. Ya, dengan cara yang misterius, ini membawa kita pada gambaran umum React juga!

Desain Multi-Halaman Jadul

Jika Anda harus membuat aplikasi ini beberapa tahun lalu, Anda mungkin telah mengambil pendekatan yang melibatkan beberapa halaman individual. Alurnya akan tampak seperti yang ditunjukkan pada Gambar 1.2.

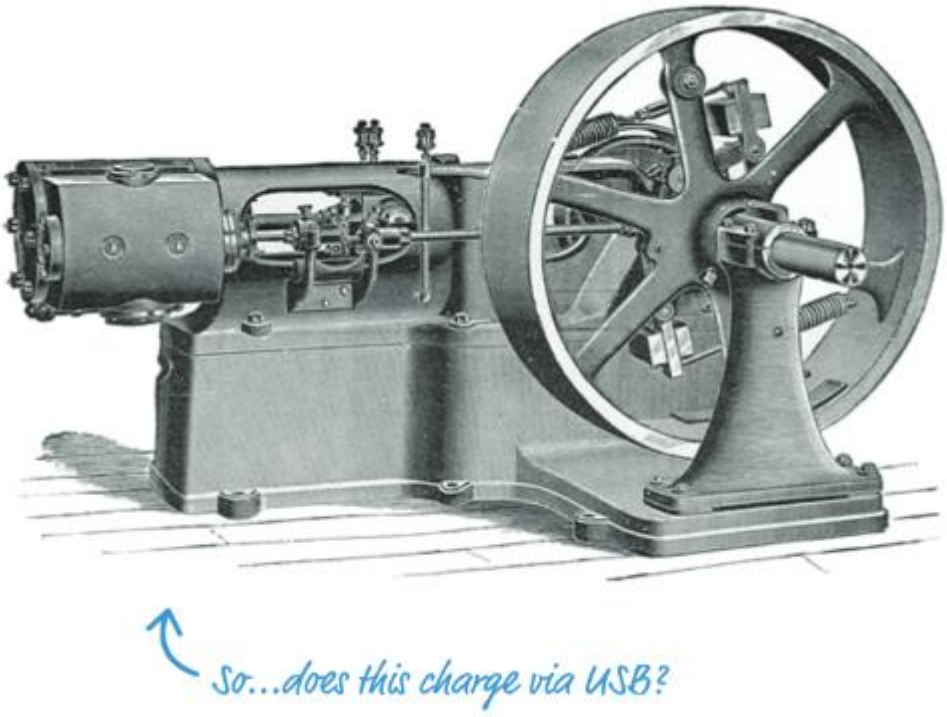
Gambar 1.2 Desain Multihalaman

Untuk hampir setiap tindakan yang mengubah tampilan browser, aplikasi web akan mengarahkan Anda ke halaman yang sama sekali berbeda. Ini adalah masalah besar di luar pengalaman pengguna yang kurang memuaskan yang akan dilihat pengguna saat halaman dirobohkan dan digambar ulang.

Ini berdampak besar pada cara Anda mempertahankan status aplikasi. Di luar penyimpanan beberapa data pengguna melalui cookie dan beberapa mekanisme sisi server, Anda tidak perlu peduli. Hidup itu indah.

Aplikasi Satu Halaman Gaya Baru

Saat ini, menggunakan model aplikasi web yang mengharuskan navigasi antar halaman individual tampak ketinggalan zaman...seperti, benar-benar ketinggalan zaman, seperti yang ditunjukkan pada Gambar 1.3.



Gambar 1.3 Model Halaman Individual Agak Ketinggalan Zaman Seperti Mesin Uap Ini. Sumber: Katekismus Baru Mesin Uap, 1904

Sebaliknya, aplikasi modern cenderung mengikuti apa yang dikenal sebagai model Aplikasi satu halaman (SPA). Ini adalah dunia di mana Anda tidak pernah menavigasi ke halaman yang berbeda atau bahkan memuat ulang halaman. Sebaliknya, tampilan aplikasi Anda yang berbeda dimuat dan diturunkan ke halaman yang sama itu sendiri. Untuk aplikasi kita, ini mungkin terlihat seperti Gambar 1.4.

Saat pengguna berinteraksi dengan aplikasi kami, kami mengganti konten di area merah putus-putus dengan data dan HTML yang sesuai dengan apa yang coba dilakukan pengguna. Hasil akhirnya adalah pengalaman yang jauh lebih lancar. Anda bahkan dapat menggunakan banyak teknik visual agar konten baru Anda bertransisi dengan baik seperti yang mungkin Anda lihat di aplikasi keren di perangkat seluler atau desktop Anda. Hal semacam ini tidak mungkin dilakukan saat menavigasi ke halaman yang berbeda.

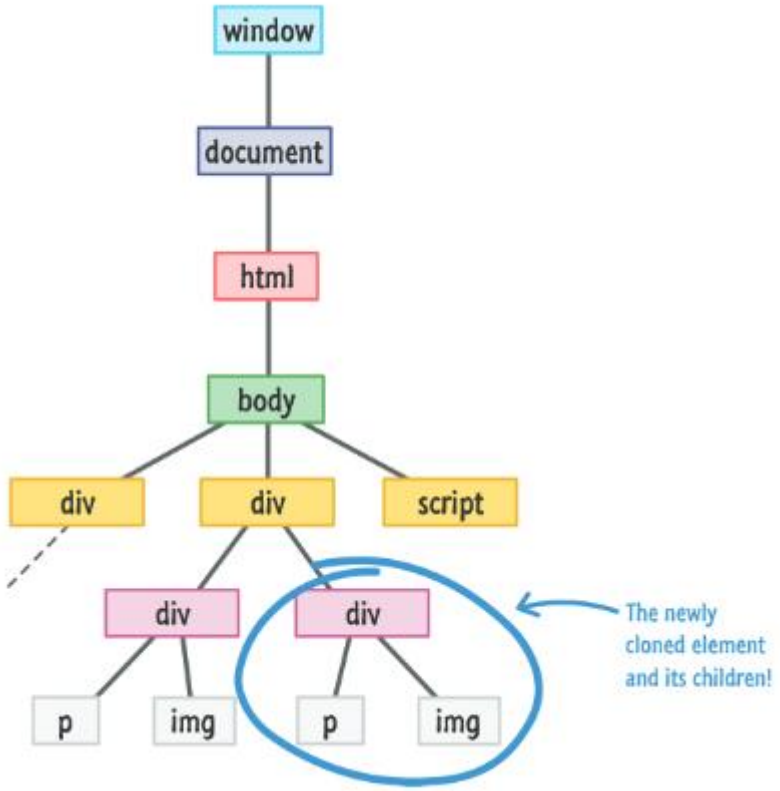
Semua ini mungkin terdengar agak aneh jika Anda belum pernah mendengar tentang aplikasi satu halaman sebelumnya, tetapi ada kemungkinan besar Anda pernah menemukannya di alam liar. Jika Anda pernah menggunakan aplikasi web populer seperti Gmail, Facebook, Instagram, atau Twitter, Anda menggunakan aplikasi satu halaman. Di semua aplikasi tersebut, konten ditampilkan secara dinamis tanpa mengharuskan Anda menyegarkan atau menavigasi ke halaman lain.

Sekarang, saya membuat aplikasi satu halaman ini tampak sangat rumit. Itu tidak sepenuhnya benar. Berkat banyaknya peningkatan hebat pada JavaScript dan berbagai framework dan library pihak ketiga, membangun aplikasi satu halaman tidak pernah semudah ini. Namun, itu tidak berarti tidak ada ruang untuk peningkatan.

Gambar 1.4 Aplikasi Satu Halaman

Saat membuat aplikasi satu halaman, ada tiga masalah utama yang akan Anda hadapi. Dalam aplikasi satu halaman, sebagian besar waktu Anda akan dihabiskan untuk menjaga data Anda tetap sinkron dengan UI Anda. Misalnya, jika pengguna memuat konten baru, apakah kita secara eksplisit menghapus kolom pencarian? Apakah kita tetap menampilkan tab aktif pada elemen navigasi? Elemen mana yang kita simpan di halaman, dan mana yang kita hancurkan? Ini semua adalah masalah yang unik untuk aplikasi satu halaman. Saat menavigasi antar halaman dalam model lama, kita berasumsi semua yang ada di UI kita akan dihancurkan dan dibangun kembali. Ini tidak pernah menjadi masalah.

Masalah selanjutnya yakni saat memanipulasi DOM benar-benar SANGAT lambat. Menanyakan elemen secara manual, menambahkan anak (lihat Gambar 1.5 di bawah), menghapus subpohon, dan melakukan operasi DOM lainnya adalah beberapa hal paling lambat yang dapat Anda lakukan di browser Anda. Sayangnya, dalam aplikasi satu halaman, Anda akan melakukan banyak hal ini. Memanipulasi DOM adalah cara utama Anda dapat menanggapi tindakan pengguna dan menampilkan konten baru.



Gambar 1.5 Menambahkan Anak

Terakhir, bekerja dengan templat HTML bisa jadi menyusahkan. Navigasi dalam aplikasi satu halaman tidak lebih dari sekadar Anda berurusan dengan fragmen HTML untuk mewakili apa pun yang ingin Anda tampilkan. Fragmen HTML ini sering dikenal sebagai templat, dan menggunakan JavaScript untuk memanipulasinya dan mengisinya dengan data menjadi sangat rumit dengan sangat cepat. Lebih buruknya lagi, tergantung pada kerangka kerja yang Anda gunakan, tampilan templat dan interaksinya dengan data bisa sangat bervariasi. Misalnya, berikut ini tampilan penggunaan templat di Mustache:

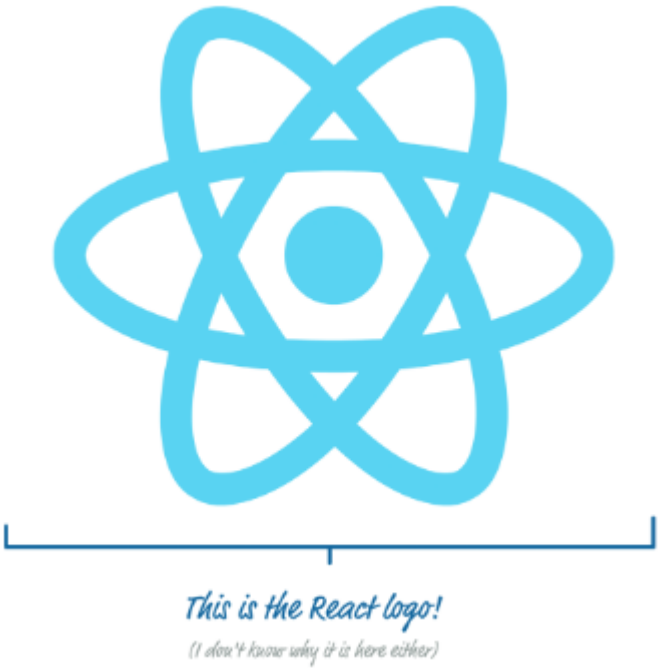
```
var view = {
      title: "Joe",
      calc: function () {
            return 2 + 4;
      }
};
var output = Mustache.render ("{{title} } spends {{calc} }", view);
```

Terkadang, templat Anda mungkin terlihat seperti HTML bersih yang dapat Anda pamerkan dengan bangga di depan kelas. Di lain waktu, templat Anda mungkin tidak dapat dipahami, dengan banyak sekali tag khusus yang dirancang untuk membantu memetakan elemen HTML Anda ke beberapa data. Meskipun ada kekurangan ini, aplikasi satu halaman tidak akan hilang. Aplikasi tersebut adalah bagian dari masa kini, dan akan sepenuhnya membentuk masa depan cara aplikasi web dibuat. Itu tidak berarti bahwa kita harus menoleransi kekurangan ini. Untuk

mengatasinya, kenali React!

1.1 KENALI REACT

Facebook (dan Instagram) memutuskan bahwa sudah cukup. Mengingat banyaknya pengalaman mereka dengan aplikasi satu halaman, mereka merilis pustaka yang disebut React (logo React ditunjukkan pada Gambar 1.6) untuk tidak hanya mengatasi kekurangan ini, tetapi juga mengubah cara kita berpikir tentang pembuatan aplikasi satu halaman.

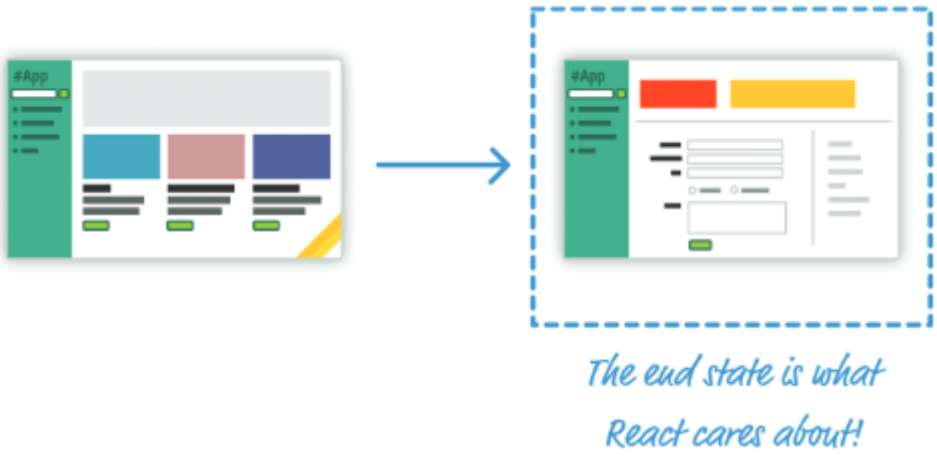


Gambar 1.6 Logo React

Pada bagian berikut, mari kita lihat hal-hal besar yang ditawarkan React.

1.2 MANAJEMEN STATUS UI OTOMATIS

Dengan aplikasi satu halaman, melacak UI dan mempertahankan status sulit dan sangat memakan waktu. Dengan React, Anda hanya perlu mengkhawatirkan satu hal: status akhir UI Anda. Tidak masalah status awal UI Anda. Tidak masalah rangkaian langkah apa yang mungkin telah diambil pengguna untuk mengubah UI. Yang penting adalah di mana UI Anda berakhir (lihat Gambar 1.7).



Gambar 1.7 Keadaan Akhir Atau Akhir UI Anda Adalah Hal Yang Penting Dalam React.

React mengurusi hal-hal lainnya. React mencari tahu apa yang perlu terjadi untuk memastikan UI Anda terwakili dengan benar, jadi semua hal manajemen keadaan itu bukan lagi urusan Anda.

Memulai dengan React Native

Secara umum, memulai dengan React Native cukup mudah karena hanya membutuhkan sedikit persyaratan. React Native tidak mengharuskan penggunaan lingkungan pengembangan terintegrasi (IDE) tertentu. Dalam buku ini, saya akan fokus pada penggunaan antarmuka baris perintah (command line interface). Perlu dicatat bahwa saya adalah pengguna Windows, jadi tangkapan layar yang ditampilkan akan diambil dari sistem operasi Windows. Namun, seharusnya tidak ada perbedaan signifikan jika Anda menggunakan Mac atau Linux. Perbedaan yang ada, seperti penggunaan tanda / alih-alih , dan perbedaan platform umum lainnya, mudah dipahami sendiri.

Prasyarat

Seperti halnya banyak teknologi saat ini, React Native memerlukan instalasi Node.js (yang sering disebut hanya Node) dan Node Package Manager (NPM). Jika Anda sudah familiar dengan kedua alat ini dan telah menyiapkannya, Anda dapat langsung melanjutkan ke bagian berikutnya. Namun, jika Anda belum mengenalnya, bacalah panduan singkat berikut untuk memahami apa itu Node dan NPM serta bagaimana cara menyiapkannya.

Node

Node adalah platform yang dirancang untuk menjalankan kode sisi server dengan kinerja tinggi, mampu menangani banyak permintaan dengan efisien. Node dibangun menggunakan JavaScript, bahasa pemrograman yang paling banyak digunakan di dunia. Meskipun mudah dipelajari, JavaScript memberikan kekuatan luar biasa bagi pengembang, terutama berkat model pemrograman asinkron dan berbasis peristiwa yang dimilikinya. Di Node, hampir semua operasi bersifat non-blokir, yang berarti kode tidak akan menghalangi pemrosesan permintaan lainnya.

Keunggulan Node semakin diperkuat dengan penggunaan mesin JavaScript V8 milik Google, yang juga digunakan oleh peramban Chrome. Mesin ini memberikan kinerja yang

sangat tinggi, memungkinkan Node untuk menangani jumlah permintaan yang besar. Oleh karena itu, tidak mengherankan jika banyak perusahaan besar telah mengadopsi Node dalam pengembangan aplikasi mereka, termasuk DuckDuckGo, eBay, LinkedIn, Microsoft, Walmart, dan Yahoo.

Node adalah lingkungan runtime kelas satu, yang berarti Anda dapat melakukan berbagai hal, seperti berinteraksi dengan sistem berkas lokal, mengakses database relasional, atau melakukan panggilan ke sistem jarak jauh. Sebelumnya, untuk melakukan hal-hal seperti ini, Anda harus menggunakan runtime yang "tepat", seperti Java atau .NET. Namun, dengan Node, JavaScript kini dapat digunakan untuk melakukan tugas-tugas tersebut, menjadikannya pilihan yang sangat fleksibel dan powerful.

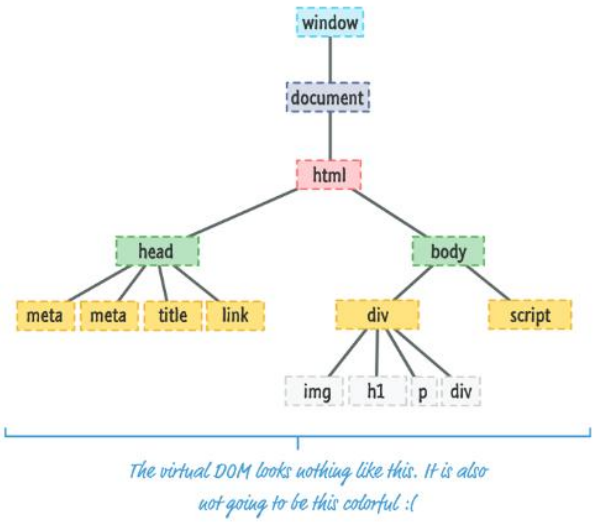
Untuk lebih jelasnya, Node bukanlah server itu sendiri, meskipun paling sering digunakan untuk membuat server. Namun sebagai runtime JavaScript generik, ini adalah runtime yang dijalankan oleh banyak alat non-server, dan jika Anda sekarang menebak bahwa rangkaian alat React Native melakukan hal itu, maka tepuk punggung Anda sendiri.

Itulah inti dari Node. Perlu diketahui bahwa bagian ini tidak dimaksudkan untuk mengulas Node secara menyeluruh. Node memiliki lebih dari sekadar ini, dan jika Anda baru mengenalnya, saya sarankan Anda untuk membaca situs Node (nodejs.org). Namun, untuk keperluan buku ini, pemahaman dasar ini sudah cukup.

Mendapatkan, memasang, dan menjalankan Node adalah latihan yang mudah, apa pun preferensi sistem operasi Anda. Tidak ada penginstalan yang rumit dengan segala macam dependensi, juga tidak ada sekumpulan besar berkas konfigurasi yang harus diutak-atik sebelum Anda dapat menjalankan aplikasi Node

1.3 MANIPULASI DOM

Karena modifikasi DOM sangat lambat, Anda tidak pernah memodifikasi DOM secara langsung menggunakan React. Sebagai gantinya, Anda memodifikasi DOM virtual dalam memori. Gambar 1.8 melambangkan DOM virtual dalam memori tersebut.



Gambar 1.8 Bayangkan DOM Virtual Dalam Memori.

Memanipulasi DOM virtual ini sangat cepat, dan React mengurus pembaruan DOM asli saat waktunya tepat. React melakukannya dengan membandingkan perubahan antara DOM virtual Anda dan DOM asli, mencari tahu perubahan mana yang benar-benar penting, dan membuat perubahan DOM paling sedikit yang diperlukan untuk menjaga semuanya tetap mutakhir dalam proses yang disebut rekonsiliasi.

Virtual DOM

Jika Anda sudah pernah terlibat dalam pengembangan web (dan saya mengasumsikan Anda sudah, untuk tujuan buku ini), Anda pasti akrab dengan istilah Document Object Model (DOM). DOM adalah struktur pohon terbalik yang menggambarkan elemen-elemen dalam sebuah halaman web. Di puncak pohon ini terdapat objek dokumen, yang memiliki anak-anak seperti <head> dan <body>, yang sesuai dengan tag HTML yang biasa kita kenal. Di bawahnya, terdapat berbagai elemen lainnya, misalnya elemen di bawah <body>, <h1> di bawahnya, dan seterusnya. Setiap kali Anda menggunakan JavaScript untuk mengubah sesuatu di halaman, atau ketika pengguna melakukan interaksi yang menyebabkan perubahan, DOM akan diperbarui, dan browser akan merender perubahan tersebut. Tergantung pada jenis perubahan yang terjadi, DOM bisa berubah signifikan, yang mengharuskan browser untuk merender ulang sebagian besar halaman, sebuah proses yang bisa sangat lambat meskipun vendor browser telah berusaha sebaik mungkin untuk mempercepatnya.

Masalah utama terletak pada bagaimana DOM bekerja di browser, karena setiap perubahan pada DOM membutuhkan pembaruan yang rumit dan mahal secara komputasi di layar. Ketika browser mengurai dokumen HTML, ia membangun pohon DOM, yang mencocokkan setiap tag HTML dengan simpul dalam pohon tersebut. Di sampingnya, sebuah pohon render juga dibangun, yang memuat informasi gaya terkait dengan setiap tag. Setiap kali informasi gaya diproses, sebuah proses yang disebut *attachment* terjadi, menggunakan metode attachment() yang sesuai. Masalah muncul karena setiap pemanggilan ke metode ini bersifat sinkron. Artinya, setiap kali simpul baru dimasukkan, simpul dihapus, atau status elemen diubah, metode attachment() akan dipanggil. Setiap perubahan pada satu elemen bisa menyebabkan perubahan pada elemen lain, bahkan banyak elemen lainnya, karena tata letak halaman harus dihitung ulang dan dirender ulang. Semua perubahan ini, yang bisa mencapai ratusan atau ribuan tergantung pada interaksi pengguna, akan memicu panggilan sinkron yang sangat mahal dari segi performa. Ini menciptakan masalah besar dalam hal kinerja.

Dalam kasus Virtual DOM, browser tidak menggunakannya untuk merender atau menghitung apapun secara langsung. Sebaliknya, Virtual DOM berfungsi sebagai lapisan abstraksi di atas DOM asli browser, tetapi secara konsep tetap berupa pohon. Namun, Virtual DOM terdiri dari POJO (Plain Old JavaScript Objects) yang ringan dan sederhana, berbeda dengan DOM asli. Perbedaan pentingnya adalah setiap kali Anda membuat perubahan pada Virtual DOM, sejumlah kode akan dieksekusi sebelum perubahan tersebut ditangani oleh browser. Kode ini menggunakan algoritma diffing untuk mengelompokkan perubahan yang diperlukan sehingga perubahan tersebut dapat diterapkan pada DOM asli dalam satu kali proses. Ini juga berfungsi untuk memastikan bahwa hanya sebagian kecil dari DOM asli yang

diperbarui, yang membuatnya jauh lebih efisien. Dengan demikian, React, dalam hal ini, dapat menghitung perbedaan antara Virtual DOM yang ada dan perubahan yang dilakukan kode Anda, dan hanya membuat perubahan minimal pada DOM asli. Hal ini meningkatkan kinerja karena perubahan dilakukan sekaligus, daripada mengubah DOM browser secara langsung. Pendekatan ini jauh lebih efisien, terutama saat halaman menjadi lebih kompleks, meskipun mungkin tidak terlalu terlihat pada halaman kecil.

Pada React Native, proses ini bahkan lebih menarik. Ketika perubahan pada Virtual DOM perlu ditampilkan di layar, alih-alih mengubah DOM browser dan membuat browser merender layar berdasarkan itu, React Native membuat komponen asli dari platform dan menggambarnya di layar menggunakan metode asli dari platform tersebut. Jadi, di React biasa pada Web, tag akan diubah menjadi elemen di DOM browser yang ditampilkan di layar. Namun, di React Native, elemen <View> (yang akan Anda pelajari lebih lanjut) akan digambar sebagai komponen UIView di iOS dan komponen View di Android. Ini berarti kedua komponen tersebut adalah komponen asli, bukan elemen di dalam browser. Perbedaan ini sangat signifikan! Meskipun ada perbedaan besar dalam cara kerjanya, Anda tetap menulis aplikasi dengan sintaks yang mirip dengan aplikasi web biasa, meskipun ada perbedaan dalam cara penulisannya, yang akan Anda lihat lebih lanjut nanti.

1.4 API UNTUK MEMBUAT UI YANG BENAR-BENAR DAPAT DISUSUN

Daripada memperlakukan elemen visual dalam aplikasi Anda sebagai satu bagian monolitik, React mendorong Anda untuk memecah elemen visual menjadi komponen yang semakin kecil. Seperti halnya semua hal lain dalam pemrograman, sebaiknya semuanya bersifat modular, ringkas, dan mandiri.

React memperluas ide yang sudah mapan itu ke cara kita berpikir tentang antarmuka pengguna juga. Banyak API inti React yang memudahkan pembuatan komponen visual yang lebih kecil yang nantinya dapat digabungkan dengan komponen visual lain untuk membuat komponen visual yang lebih besar dan lebih kompleks seperti boneka Matryoshka Rusia (lihat Gambar 1.9).

Gambar 1.9 Boneka Matryoshka Rusia Karya Gnomz007

Ini adalah salah satu cara utama React menyederhanakan (dan mengubah) cara kita berpikir

tentang membangun visual untuk aplikasi web kita.

Pendekatan untuk Membahas Komponen dan API di React Native

Setiap komponen di React Native memiliki berbagai properti yang dapat digunakan untuk mengonfigurasinya, beberapa di antaranya memiliki banyak opsi. Selain itu, banyak komponen juga menyediakan metode yang dapat dipanggil pada instance untuk melakukan tindakan tertentu, atau bertindak sebagai metode pembantu statis. Di balik layar, beberapa komponen juga melibatkan tipe JavaScript terkait. Namun, saya tidak akan membahas setiap properti, metode, atau tipe terkait secara mendalam, atau menunjukkan setiap kemungkinan variasi penggunaan komponen. Tujuan bagian ini bukan untuk memberikan penjelasan rinci tentang setiap detail, melainkan memberikan gambaran umum tentang apa yang Anda dapatkan *langsung dari kotak* dengan React Native.

Setelah membaca bagian ini, Anda akan memiliki pemahaman dasar tentang berbagai komponen dan API yang tersedia, cukup untuk mulai menulis kode nyata dalam aplikasi React Native. Ada beberapa alasan mengapa saya tidak memberikan penjelasan yang sangat rinci. Pertama, React Native memiliki banyak komponen dan API yang perlu dibahas, dan satu bab saja tidak cukup untuk menjelaskan semuanya secara menyeluruh. Untuk informasi lebih lanjut, Anda bisa merujuk pada dokumentasi resmi React Native, yang terus diperbarui seiring dengan perkembangan framework ini. Mengingat kecepatan perubahan dalam React Native, saya lebih memilih untuk tidak mencoba menyertakan semua aspek dengan lengkap di sini.

Kedua, banyak detail tentang penggunaan komponen dan API tertentu akan dijelaskan lebih lanjut dalam bab-bab selanjutnya ketika kita mulai membangun aplikasi nyata. Ketiga, beberapa komponen dan API mungkin hanya memiliki dokumentasi yang terbatas atau bahkan kurang informasinya. Saya memilih untuk tidak membahasnya di sini, karena saya lebih suka memberikan informasi yang dapat saya verifikasi dan pastikan akurat. Jika Anda memerlukan informasi lebih lanjut, Anda bisa mencarinya sendiri saat membutuhkannya.

Keempat, beberapa komponen lebih baik diperkenalkan melalui skenario penggunaan nyata, yang akan dibahas dalam bab-bab berikutnya. Oleh karena itu, saya sengaja melewatkannya di sini.

Terakhir, saya tidak menyertakan komponen atau API yang digunakan dalam penulisan proyek kode asli dalam buku ini, karena itu di luar cakupan buku. Saya tidak ingin memberikan informasi yang mungkin tidak bisa langsung Anda gunakan tanpa mengikuti langkah-langkah tambahan. Meskipun demikian, komponen-komponen ini relatif jarang digunakan, jadi Anda tidak akan kehilangan banyak jika tidak membahasnya di sini. Namun, saya ingin memberi tahu Anda bahwa komponen-komponen tersebut ada jika Anda membutuhkan informasi lebih lanjut.

Secara keseluruhan, jangan anggap bagian ini sebagai referensi terperinci tentang semua komponen dan API React Native. Sebaliknya, anggap ini sebagai survei atau gambaran umum yang memberikan pemahaman dasar yang diperlukan untuk mulai membangun aplikasi nyata yang akan kita buat di bab-bab berikutnya.

Komponen

Untuk memulai, saya akan mengusulkan bahwa ada dua kategori luas "hal" yang

menjadi perhatian bab ini: komponen dan API. Komponen, tentu saja, adalah elemen visual yang Anda lihat di layar dalam aplikasi React Native, dan bahkan sebelum Anda mempertimbangkan pustaka komponen pihak ketiga (sesuatu yang benar-benar dapat Anda lakukan dengan React Native), ada beberapa komponen yang tersedia untuk Anda, cukup untuk membangun aplikasi dunia nyata, sebenarnya.

Komponen-komponen ini sendiri dapat dipecah secara luas menjadi beberapa pengelompokan, berdasarkan berbagai karakteristik enam kelompok, tepatnya. Catatan React Native selalu mengharuskan Anda mengimpor komponen apa pun yang akan Anda gunakan, dan komponen yang dijelaskan dalam bab ini semuanya berasal dari modul react-native. Saya tidak akan menunjukkan impor tersebut berulang kali, tetapi pahamilah bahwa impor tersebut diperlukan, dan Anda dapat melihatnya di bagian atas aplikasi contoh komponen untuk bab ini, yang tentu saja disertakan dalam bundel sumber yang dapat diunduh untuk buku ini.

Komponen Dasar

Kelompok pertama dari enam kelompok mencakup komponen "dasar". Hampir setiap aplikasi menggunakan satu atau beberapa komponen dasar, dan kategori ini merupakan gabungan untuk komponen yang mendasari semua komponen lainnya. Komponen tersebut mendasari komponen lain baik secara langsung, dalam kasus di mana komponen lain mungkin merupakan subkelas komponen ini, atau dengan cara yang lebih umum, dalam arti bahwa komponen lain menjadi anak dari komponen ini (atau antarmuka pengguna dibangun dari komponen ini "di bawah" jenis komponen lainnya). Faktanya, salah satu komponen ini bahkan bukan komponen, dalam pengertian tradisional yang Anda anggap sebagai komponen, karena komponen ini tidak visual, tetapi kita akan membahasnya terakhir. Mari kita mulai dengan komponen yang mungkin paling sering digunakan dalam kotak peralatan React Native: View.

View

Komponen View mungkin merupakan satu-satunya komponen andalan, dan kemungkinan besar Anda akan lebih sering menggunakannya daripada komponen lainnya. Seperti yang dijelaskan dalam Bab 1, komponen View dipetakan ke komponen OS asli yang mendasar komponen UIView pada iOS dan komponen View pada Android (dan, jika React Native dirender ke HTML di browser, komponen tersebut akan dipetakan ke elemen yang ada di mana-mana).

Komponen View, secara sederhana, adalah elemen kontainer yang mendukung tata letak dengan flexbox, penataan gaya melalui CSS, beberapa penanganan sentuhan umum, dan kontrol aksesibilitas. Dengan demikian, komponen View hadir dalam berbagai bentuk, tetapi penggunaan dasarnya mungkin sebagai berikut:

```
<View style={{width:200,height:100,backgroundColor:"#ff0000" }} >
```

Ini menciptakan kotak merah selebar 200 piksel dan tinggi 100 piksel di layar. Perhatikan bahwa gaya ditampilkan sebaris di sini, tetapi lebih umum untuk mengeksternalisasikannya dalam objek StyleSheet, sesuatu yang Anda lihat di Bab 1 dan yang akan kita bahas lebih lanjut, nanti di bab ini.

Tampilan dapat memiliki nol atau lebih anak, dan anak-anak ini dapat membuat hierarki bertingkat sedalam yang diperlukan untuk mencapai tata letak yang Anda inginkan. Misalnya, jika Anda ingin memiliki dua kotak berwarna dalam satu baris, Anda dapat melakukan hal berikut:

```
<View style={{ flexDirection : "row", height : 100, padding: 20 }}>
    <View style={{ backgroundColor : "#ff0000", flex : 0.5}} />
    <View style={{ backgroundColor : "#00ff00", flex : 0.5}} />
</View>
```

Saya akan membahas lebih lanjut tentang tata letak di bab berikutnya, tetapi ini akan memberikan gambaran awal.

Teks

Komponen Teks dalam banyak hal sama seperti komponen Tampilan, kecuali bahwa komponen ini secara khusus ditujukan untuk menampilkan teks, tetapi seperti komponen Tampilan, komponen ini mendukung penataan gaya, penyusunan, dan beberapa penanganan sentuhan.

Komponen Teks mungkin sesederhana ini:

```
<Text>Hello, I am a Text component</Text>
```

Atau mungkin ada beberapa gaya:

```
<Text style={{color:"red" }}>Hello,I am a Text component</Text>
```

Secara default, komponen Teks mewarisi informasi gaya dari komponen Teks induknya, tetapi hal itu dapat diganti.

```
<Text style={{color:"red"}}>
    <Text>I am red</Text>
    <Text style= {{color:"green"}}> I am green</Text>
</Text>
```

Seperti yang Anda lihat, komponen Teks tidak harus berisi teks. Dan, seperti yang Anda lihat, komponen Teks dapat disarangkan di dalam satu sama lain.

Satu keanehan dengan komponen Teks adalah bahwa ketika tata letak mulai berlaku, konten apa pun di dalam komponen Teks tidak menggunakan tata letak flexbox, seperti halnya View dan komponen React Native lainnya yang mendukung tata letak. Sebaliknya, di dalam komponen Teks, tata letak teks digunakan. Ini berarti bahwa setiap elemen yang disarangkan di dalam komponen Teks tidak lagi berbentuk persegi panjang. Sebaliknya, elemen tersebut akan terbungkus ketika akhir baris ditemukan. Artinya dalam praktiknya adalah bahwa semua komponen Teks bertindak seolah-olah mereka adalah satu string teks yang besar dan panjang di bawah induk yang sama. Misalnya:

```
<Text style={{ fontWeight : "bold" }}>I am bold
       <Text style={{ color : "#ff0000" }}and red</Text>
 </Text>
```

Saat ini ditampilkan, jika wadah cukup lebar untuk menampung komponen Teks induk, kedua komponen Teks anak akan ditampilkan sebagai "Saya adalah model jenderal besar modern," seolah-olah itu adalah satu komponen Teks, satu string.

Komponen Teks mendukung beberapa peristiwa sentuh melalui properti onPress dan onLongPress. Cukup beri mereka nilai yang merupakan fungsi, dan Anda pada dasarnya dapat membuat tombol Anda sendiri dengan kombinasi teks dan gaya (dan semacam "buat hal yang dapat disentuh" ini cukup umum dalam pengembangan React Native).

Komponen Teks juga dapat disarangkan dalam formulir web biasa, yang memungkinkan pemformatan teks yang mudah. Misalnya:

```
<Text style={{ fontWeight : "bold" }}>I am bold
    <Text style={{ color : "#ff0000" }}and red</Text>
</Text>
```

Ini akan menghasilkan string tebal, "Saya tebal dan merah," di mana kata-kata dan merah—tunggu saja—BERWARNA MERAH! Perhatikan bahwa tidak ada spasi di antara keduanya, karena spasi kosong di akhir konten komponen Teks pertama dan sebelum komponen Teks bertingkat diabaikan, untuk tujuan menampilkannya di layar.

Gambar

Komponen Gambar persis seperti namanya: komponen ini memungkinkan Anda untuk menampilkan gambar. Komponen ini mendukung gambar yang diambil dari jaringan, seperti tag HTML `<img>`, sumber daya statis yang dibaca dari sistem berkas atau dikodekan sebagai URL data atau dari lokasi tertentu, seperti rol kamera perangkat. Penggunaannya yang paling sederhana mungkin

```
<Image source={require("./image.png")}/>
```

Dengan asumsi Anda memiliki berkas bernama image.png di akar kode aplikasi Anda, berkas tersebut akan dibaca dan ditampilkan. Atau, Anda mungkin mendapatkannya dari jaringan.

```
<Image source={ uri:"https://www.etherient.com/logo.png"}/>
```

Anda dapat menerapkan gaya pada komponen Image, menggunakan properti style yang telah Anda lihat beberapa kali.

Beberapa properti terkait peristiwa tersedia. Ini termasuk onLoad, yang merupakan fungsi yang Anda sediakan yang akan dijalankan saat gambar berhasil dimuat (terutama

berguna saat mengambil dari jaringan, karena itu bisa memakan waktu, dan Anda mungkin ingin melakukan sesuatu saat dimuat); onLoadStart, yang dijalankan saat pemuatan dimulai; dan onLoadEnd, yang dijalankan baik saat pemuatan gagal atau tidak (dan jika gagal, ada onError yang tersedia untuk Anda kaitkan).

Komponen Image adalah komponen pertama yang Anda temui yang juga menyediakan beberapa metode yang dapat dipanggil. Semua metode ini adalah metode statis pada objek Image itu sendiri. Jadi, misalnya, jika Anda ingin mendapatkan lebar dan tinggi gambar di jaringan, Anda dapat melakukannya seperti ini:

```
Image.getSize("https://www.etherient.com/logo.png,
      (width,height)=>{console.log(width,height);}
);
```

Komponen Image juga menawarkan metode prefetch (), yang memuat gambar ke dalam memori tanpa menampilkannya, dan queryCache(), yang memungkinkan Anda menentukan apakah gambar telah dimuat dan di-cache di memori, di antara metode lainnya.

Satu hal yang perlu diperhatikan di sini adalah penggunaan fungsi console.log () oleh saya. Anda hampir pasti pernah melihat ini dalam pekerjaan pengembangan web Anda, dan karenanya, Anda terbiasa dengan pesan-pesan tersebut yang muncul di beberapa alat pengembang, seperti alat pengembang Chrome, misalnya. Berdasarkan penggunaan Expo, kita juga memiliki objek konsol dengan metode yang mungkin sudah Anda kenal, seperti log (). Namun, ke mana pesan-pesan ini pergi? Ternyata pesan-pesan tersebut dikeluarkan ke konsol tempat Anda memulai aplikasi dengan npm start. Ya, aplikasi yang berjalan pada perangkat nyata dapat mengeluarkan pesan log ke PC yang Anda gunakan untuk melakukan pengembangan. Keren, bukan? Bagus, karena itu berarti sebagian besar trik berorientasi CLI yang mungkin biasa Anda gunakan dapat diterapkan di sini, jika Anda menyalurkan output ke suatu tempat untuk diproses. Namun, selain itu, kemampuan untuk melihat pesan log dengan cepat tanpa harus mencoba dan melihatnya di perangkat itu sendiri sudah sangat bagus.

ScrollView

Secara sederhana, komponen ScrollView pada dasarnya hanyalah komponen View yang memungkinkan pengguliran. Dengan kata lain, komponen ini adalah komponen kontainer, seperti View, tetapi memungkinkan lebih banyak konten untuk ditampilkan daripada yang dapat ditampilkan di layar sekaligus dan kemudian memungkinkan pengguna untuk menggulir konten tersebut.

Anda menggunakan ScrollView persis seperti View.

```
<ScrollView>
... sejumlah komponen, lebih banyak dari yang dapat dimuat di layar ...
</ScrollView>
```

ScrollView harus memiliki tinggi terbatas, yang berarti Anda harus menyetel tingginya secara

langsung, yang tidak disarankan, atau wadah induknya (semuanya, sebenarnya) akan memiliki tinggi terbatas. Cara termudah untuk melakukannya adalah dengan memastikan bahwa flex:1 disetel pada semua tampilan induk ScrollView (meskipun tampaknya jika ScrollView adalah tampilan wadah pertama, maka ini otomatis benar).

Perhatikan bahwa ScrollView merender semua anaknya sekaligus, bahkan yang belum terlihat. Seperti yang dapat Anda tebak, ini dapat menjadi masalah kinerja, tergantung pada kompleksitas hierarki komponen. Anda akan menemui komponen FlatList nanti, yang, untuk sebagian besar maksud dan tujuan, dapat dianggap sebagai ScrollView yang merender anaknya dengan malas, yaitu, hanya saat pengguliran membuatnya terlihat.

Komponen ScrollView memiliki berbagai macam properti, termasuk:

- alwaysBounceVertical: Jika benar, ScrollView memantul ke arah vertikal saat mencapai akhir kontennya, bahkan jika konten tersebut lebih kecil dari ScrollView itu sendiri. • showsHorizontalScrollIndicator: Jika benar, indikator yang menunjukkan posisi gulir horizontal akan ditampilkan (ada juga properti showsVerticalScrollIndicator yang sesuai).
- centerContent: Jika benar, ScrollView secara otomatis memusatkan anak-anaknya, selama konten anak-anak tersebut lebih kecil dari batas ScrollView.
- zoomScale: Skala konten ScrollView saat ini (ini adalah properti khusus iOS)
- ScrollView juga mendukung beberapa kait siklus hidup, termasuk:
- onScroll: Diaktifkan paling banyak satu kali per bingkai setiap kali pengguliran terjadi
- onScrollEndDrag: Diaktifkan saat pengguna berhenti menyeret
- ScrollView, dan ScrollView berakhir atau mulai meluncur Terakhir, ScrollView juga mendukung beberapa metode.
- scrollTo: Menggulir ke lokasi offset x/y tertentu (dapat dilakukan segera dengan atau tanpa animasi)
- scrollToEnd: Menggulir ke bagian bawah ScrollView vertikal atau paling kanan dari ScrollView horizontal

Ada beberapa properti lain yang tersedia, tetapi contoh ini akan memberikan gambaran umum tentang komponen ini.

TouchableHighlight

Komponen TouchableHighlight adalah komponen andalan lain yang, bersama komponen terkaitnya TouchableNativeFeedback, TouchableOpacity, dan TouchableWithoutFeedback, menyediakan cara untuk membuat hampir semua tampilan atau komponen merespons peristiwa sentuhan dengan benar. Saat komponen TouchableHighlight ditekan, opasitas tampilan yang dibungkus berkurang, sehingga warna di bawahnya dapat terlihat, yang menggelapkan atau mewarnai tampilan.

Komponen ini harus selalu memiliki satu dan hanya satu komponen anak (meskipun, jika Anda ingin lebih dari satu komponen dibungkus oleh komponen TouchableHighlight, anak tersebut dapat berupa komponen Tampilan, yang dapat berisi beberapa komponen). Berikut

contoh sederhananya:

```
<TouchableHighlight onPress={()=>{ console.log("Pressed!"); }} >
     <Text>Tap me to hide modal</Text>
</TouchableHighlight>
```

Properti onPress adalah yang utama yang akan Anda gunakan, dan merupakan fungsi yang aktif saat komponen ditekan.

Komponen lainnya bekerja sama tetapi dengan beberapa perbedaan. Komponen TouchableNativeFeedback adalah komponen khusus Android yang menggunakan komponen native state drawable untuk menampilkan umpan balik sentuhan. Ini memberikan tampilan dan nuansa Android yang tepat untuk komponen yang dapat disentuh (biasanya efek riak material). Komponen TouchableOpacity menggunakan Animated API yang akan Anda lihat nanti dan salah satu komponen yang diekspornya, Animated.View, yang dibungkus di sekitar komponen anak, untuk memanipulasi opasitas saat ditekan. TouchableWithoutFeedback, seperti yang saya yakin dapat Anda tebak, membuat elemen yang dapat disentuh yang tidak memberikan umpan balik visual. Dokumen React Native memperingatkan Anda untuk tidak menggunakan ini tanpa alasan yang kuat, dan saya setuju. Menyentuh sesuatu seharusnya memberikan umpan balik visual, jadi Anda harus menghindari penggunaan ini, kecuali Anda memiliki beberapa kasus penggunaan khusus yang hanya dapat diselesaikan dengan komponen ini.

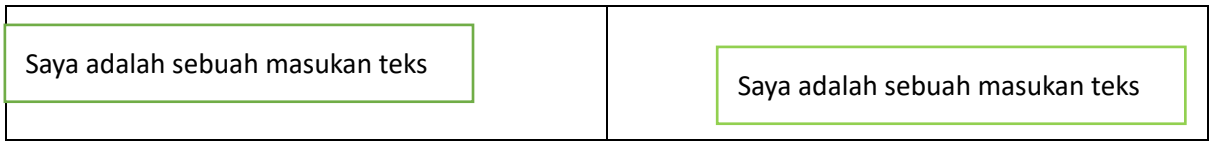
Komponen Input Data, Formulir, dan Kontrol

Jika Anda melihat dokumen React Native, Anda akan melihat apa yang mereka sebut sebagai komponen "antarmuka pengguna". Saya merasa ini agak aneh, karena bukankah semua komponen adalah komponen antarmuka pengguna? Saat saya melihat apa yang ada dalam kelompok kedua ini, menurut saya yang kita miliki di sini adalah komponen untuk input data, untuk mengendalikan berbagai hal, dan yang sering digunakan dalam formulir. Jadi, alih-alih kelompok komponen antarmuka pengguna, saya memutuskan untuk menggunakan komponen input data, formulir, dan kontrol. Saya kira Facebook dapat mengirimi saya pesan singkat, jika mereka sangat tidak setuju.

TextInput

Saya yakin Anda tidak akan terkejut mengetahui bahwa komponen TextInput (Gambar 2-1) memungkinkan pengguna untuk memasukkan informasi melalui papan ketik. Komponen ini memiliki beberapa opsi konfigurasi yang berguna untuk hal-hal seperti koreksi otomatis, kapitalisasi otomatis, dan teks pengganti. Penggunaannya sangat sederhana.

```
<TextInput value={ this.state.textInputValue }
  style={{width:"50%",height:40,borderColor:"green",borderWidth:2}}
    onChangeText={(inText) => this.setState({inText}) }
/>
```



Gambar 2.1 Komponen TextInput (versi iOS di sebelah kiri, versi Android di sebelah kanan)

Nilai komponen saat ini dikaitkan dengan status melalui properti value, tetapi perlu dicatat bahwa mengetik di kolom TextInput tidak secara otomatis memperbarui status. Tidak, Anda harus menyediakan properti pengendali onChangeText yang memanggil setState(), seperti yang dibahas di Bab 1.

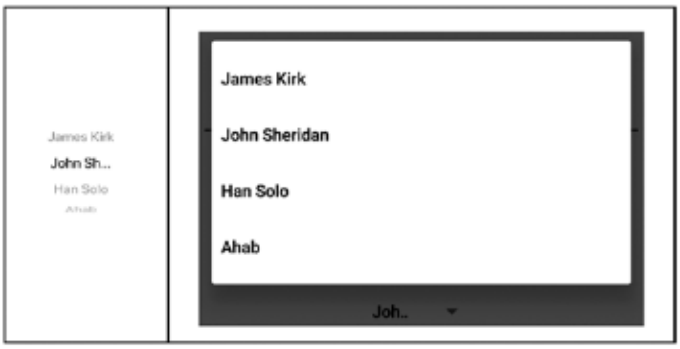
Komponen TextInput memiliki daftar panjang properti yang didukung terlalu banyak untuk dijelaskan secara terperinci di sini tetapi berikut ini adalah contoh beberapa properti yang lebih menarik (menurut saya):

- autoCapitalize: Dapat diatur ke karakter, untuk mengkapitalisasi semua yang dimasukkan; kata, untuk mengkapitalisasi huruf pertama setiap kata; kalimat, untuk mengkapitalisasi hanya kata pertama kalimat; dan none, untuk tidak mengkapitalisasi apa pun secara otomatis.
- autoCorrect: Atur ke true, untuk mengaktifkan koreksi otomatis; atur ke false untuk menonaktifkannya.
- maxLength: Ini menetapkan batas jumlah karakter yang dapat dimasukkan. • multiline: Atur ke true, untuk mengizinkan beberapa baris teks dimasukkan; jika tidak, atur ke false (yang merupakan nilai default).
- onFocus: Fungsi yang akan dijalankan saat komponen memperoleh fokus
- onBlur: Fungsi yang akan dijalankan saat komponen kehilangan fokus
- selectTextOnFocus: Atur ke true, untuk membuat TextInput menyorot teks yang ada saat kolom memperoleh fokus; atur ke false untuk tidak melakukan ini.

Picker

Di dunia React Native, Picker merujuk ke komponen yang memungkinkan pengguna memilih dari serangkaian opsi. Bentuk komponen ini bervariasi di antara platform, tetapi komponen ini memiliki fungsi dasar yang sama di platform mana pun yang mendukungnya. Namun, komponen Picker tidak berfungsi tanpa komponen lain, Picker.Item, seperti yang dapat Anda lihat berikut ini:

```
<Picker selectedValue={ this.state.bestCaptain } style={{ height : 200,
width: 100 }}
  onValueChange={ (inValue, inIndex) => this.setState({ bestCaptain:
  inValue }) }
 >
  <Picker.Item label="James Kirk" value="james_kirk" />
  <Picker.Item label="John Sheridan" value="john_sheridan" />
  <Picker.Item label="Han Solo" value="han_solo" />
  <Picker.Item label="Ahab" value="ahab" />
 </Picker>
```



Gambar 1.10 Komponen Picker (versi iOS di sebelah kiri, versi Android di sebelah kanan)

Dalam kode ini, Anda membuat komponen Picker lalu menumpuk satu atau beberapa komponen Picker. Item di bawahnya. Masing-masing berisi dua properti: label dan value. Properti label adalah yang ditampilkan di layar, dan value adalah nilai dasar untuk opsi tertentu. Seperti halnya TextInput sebelumnya, komponen Picker tidak akan mengubah status, kecuali Anda menyediakan fungsi pengendali untuk melakukannya, melalui properti onValueChange, dalam kasus ini. Anda akan melihat pola ini, kebutuhan untuk menyediakan kode guna mengubah status, yang diulang di seluruh komponen React Native, jadi sebaiknya Anda membiasakan diri dengan ini sekarang.

Daftar properti untuk Picker tidak terlalu luas. Selain selectedValue, yang memberikan nilai awal pada Picker, onValueChange, yang baru saja saya bahas, dan properti style yang selalu ada, ada juga yang diaktifkan, untuk mengaktifkan (true) atau menonaktifkan (false) komponen, mode (hanya Android), yang menentukan apakah Picker ditampilkan sebagai dialog modal (default) atau drop-down yang ditambatkan ke Picker's View, dan itemStyle, yang memungkinkan Anda memberikan gaya untuk komponen Picker.Item dengan cara yang umum.

Slider

Komponen Slider (Gambar 1.11) memungkinkan pengguna untuk memilih nilai dari rentang nilai yang telah ditentukan sebelumnya, dengan menyeret kenop di sepanjang garis slider. Komponen ini memiliki beberapa properti untuk menentukan rentang dan atribut terkait lainnya, seperti yang dapat Anda lihat berikut ini:

```
<Slider style={{ width : "75%" }} step={ 1 } minimumValue={ 0 }
   maximumValue={ 84 } value={ this.state.meaningOfLife }
   onValueChange={inValue=>this.setState({meaningOfLife:inValue})}
 />
```

Gambar 1.11. Komponen Slider (versi iOS di sebelah kiri, versi Android di sebelah kanan)

Di sini, Slider memungkinkan rentang nilai antara 0 dan 84, sebagaimana ditetapkan

oleh minimumValue dan maximumValue, dan setiap gerakan kenop mewakili perubahan nilai sebesar 1, sebagaimana ditetapkan oleh properti step. Seperti pada komponen sebelumnya, Anda harus menyediakan beberapa kode untuk memperbarui status, sebuah fungsi yang ditunjuk oleh properti onValueChange.

Selain properti dasar ini, beberapa properti lain yang tersedia meliputi yang berikut ini:

- disabled: Salah satu yang telah Anda lihat beberapa kali dan sekarang seharusnya Anda sadari tersedia di sebagian besar komponen. Properti ini menentukan apakah Slider dinonaktifkan (ketika disetel ke true) atau tidak (false).
- minimumTrackTintColor/maximumTrackTintColor: Ini memungkinkan Anda menentukan warna untuk membuat garis slider di bawah kenop dan di atas kenop, masing-masing.
- thumbImage: Ini memungkinkan Anda untuk menyediakan gambar khusus untuk kenop Slider.
- onSlidingComplete: Ini adalah fungsi panggilan balik yang dapat Anda tentukan untuk dipanggil saat pengguna melepaskan kenop Slider, terlepas dari apakah nilainya telah berubah atau tidak.

Switch

Komponen Switch (Gambar 1.12) sangat mirip dengan kotak centang HTML, yang mewakili pilihan biner: ya atau tidak, aktif atau nonaktif, 0 atau 1, dst. Seperti yang mungkin Anda bayangkan, menggunakannya cukup mudah.

```
<Switch value={this.state.loveRN}
    onValueChange={(inValue)=>this.setState({loveRN:inValue})}
 />
```

Gambar 1.12 Komponen Switch (versi iOS di sebelah kiri, versi Android di sebelah kanan)

Tidak banyak properti yang tersedia untuk Switch, yang cukup masuk akal, mengingat sifatnya. Selain value dan onValueChange, yang seharusnya sudah cukup Anda kenal sekarang, Anda juga memiliki disabled dan style, seperti kebanyakan komponen lainnya. Ada juga onTintColor, yang merupakan warna latar belakang Switch saat aktif. Anda juga memiliki tintColor, yang sama dengan onTintColor, kecuali untuk status nonaktif. Terakhir, ada thumbTintColor, yang merupakan warna pegangan Switch di latar depan (artinya dapat bervariasi dari satu platform ke platform lainnya, karena tampilan Switch itu sendiri dapat bervariasi).

Tombol

UI tanpa tombol umumnya tidak akan berfungsi banyak. Tombol adalah salah satu

elemen antarmuka paling umum yang menyediakan sarana bagi pengguna untuk menjalankan beberapa tindakan, dan React Native menawarkan komponen Tombol (Gambar 1.13) untuk tujuan tersebut. Komponen tombol, tentu saja, ditampilkan dengan cara khusus platform, dan karena tombol sangat umum dan khusus platform, React Native hanya menawarkan sedikit peluang penyesuaian. Jika Anda menginginkan atau memerlukan tombol khusus yang terlihat sama sekali berbeda dari tombol bawaan platform, Anda biasanya akan menggunakan TouchableHighlight yang dibahas sebelumnya, atau saudaranya, untuk membuat tombol dari awal. Berikut adalah contoh Tombol dasar:

```
<Button title="Go ahead, press me, I dare ya!"
      onPress={()=> console.log("You pressed me!"); }
/>
```

Gambar 1.13 Komponen Tombol (versi iOS di sebelah kiri, versi Android di sebelah kanan)

Tombol tanpa properti onPress tidak akan memiliki banyak kegunaan, dan itulah satu-satunya properti yang akan selalu Anda sediakan. Bersamaan dengan itu ada judul, yang juga diperlukan, dan merupakan teks yang akan ditampilkan pada tombol. Properti lain yang tersedia adalah:

- ✓ **accessibilityLabel**: Teks yang akan ditampilkan untuk fitur aksesibilitas tuna netra (pikirkan pembaca layar, teks yang akan mereka bacakan dengan keras akan ditentukan oleh properti ini)
- ✓ **color**: Warna teks untuk iOS, warna latar belakang tombol untuk Android
- ✓ **disabled**: Tentu saja, Anda dapat menonaktifkan tombol.

Komponen Daftar

Untungnya, kelompok komponen ketiga, komponen daftar, jauh lebih mudah dipahami daripada kelompok sebelumnya. Sesuai namanya, ini adalah komponen yang digunakan untuk menampilkan daftar item yang, biasanya, dapat digulir oleh pengguna. Untuk tujuan tersebut, komponen ini hanya merender elemen yang saat ini terlihat di layar, berbeda dengan komponen ScrollView yang dijelaskan sebelumnya, yang membuat komponen ini jauh lebih efisien dan menjadi pilihan yang baik ketika Anda memiliki daftar item yang panjang untuk ditampilkan.

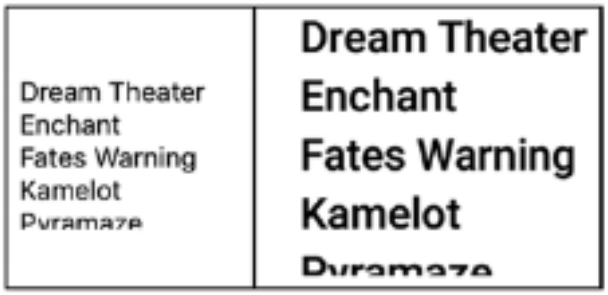
FlatList

Komponen FlatList (Gambar 1.14) adalah komponen andalan lain yang akan sering Anda lihat. Komponen ini dimaksudkan untuk merender daftar datar sederhana (datar berarti satu dimensi data) dan mendukung sejumlah fitur.

- FlatList sepenuhnya lintas platform.
- Komponen ini memiliki mode pengguliran horizontal opsional, selain mode vertikal

default-nya.

- Komponen ini memiliki dukungan elemen header, footer, dan pemisah.
- Komponen ini hanya merender item saat item tersebut terlihat, sehingga kinerjanya sangat baik.
- Komponen ini mendukung interaksi tarik-untuk-menyegarkan yang umum.



Gambar 1.14 Komponen FlatList (versi iOS di sebelah kiri, versi Android di sebelah kanan)

Secara sederhana,

FlatList dapat berupa:

```
<FlatList style={{ height : 80 }}
 data={[
    {key: "1", text: "Dream Theater"},{key:"2", text : "Enchant" },
    {key: "3", text: "Fates Warning"},{key:"4", text : "Kamelot" },
    {key: "5", text: "Pyramaze"},{ key:"6", text : "Rush" },
    {key: "7", text: "Serenity"},{ key: "8", text : "Shadow Gallery" },
    {key: "9", text: "Pink Floyd"},{key:"10", text : "Queensryche" }
  ]}
  renderItem={ ({item}) => <Text>{ item.text }</Text> }
 />
```

Menggunakan FlatList di React Native

FlatList adalah komponen yang sangat berguna untuk menampilkan daftar data yang besar dengan cara yang efisien. Anda dapat memberikan data ke FlatList secara langsung, baik dalam bentuk array atau dengan merujuk ke struktur data yang sudah ada. Properti renderItem digunakan untuk merender setiap item dalam daftar, dan Anda bebas memilih cara untuk menampilkan item tersebut sesuai kebutuhan aplikasi Anda. Dalam contoh ini, item hanya dirender sebagai komponen Text, tetapi Anda bisa menggunakan komponen React Native lainnya atau bahkan hierarki komponen yang lebih kompleks.

FlatList memiliki banyak properti yang bisa digunakan, sebagian besar diwarisi dari komponen VirtualizedList. VirtualizedList adalah komponen yang jarang digunakan secara langsung, namun sangat penting karena mewarisi dari ScrollView, yang telah Anda lihat sebelumnya. Hal ini berarti bahwa semua properti yang tersedia untuk VirtualizedList dan ScrollView juga dapat digunakan di FlatList.

Tips: Properti yang Diturunkan

Perlu dicatat bahwa semua komponen React Native mewarisi properti dari komponen induknya. Banyak komponen dasar yang memiliki properti umum, seperti disabled dan style, yang tersedia di semua komponen turunannya. Karena itu, daftar total properti yang tersedia untuk komponen apa pun merupakan gabungan dari semua properti yang diwarisi dari komponen induknya. Ini berarti Anda mungkin perlu menelusuri dokumentasi untuk menemukan properti tertentu, karena dokumentasi untuk FlatList tidak selalu mencantumkan semua properti yang diwarisinya dari VirtualizedList, ScrollView, atau bahkan View (yang merupakan turunan dari ScrollView).

Meskipun ini bisa sedikit merepotkan, setelah Anda mengetahui struktur ini, Anda akan lebih mudah menemukan informasi yang dibutuhkan dan menggunakannya dengan efektif.

Secara default, FlatList harus menemukan atribut kunci pada setiap item data. Kunci ini dapat berupa nilai unik apa pun yang Anda suka yang masuk akal untuk data Anda. Namun, terkadang Anda mungkin ingin kunci dibuat dengan menggabungkan bagian-bagian data Anda, atau mungkin Anda ingin membuat kunci secara dinamis, berdasarkan beberapa algoritme. Atau, untuk kasus tersebut, Anda dapat menyediakan fungsi, melalui fungsi keyExtractor. Fungsi ini akan dipanggil untuk setiap item data, dan nilai yang dikembalikan dari fungsi tersebut akan menjadi kunci untuk item data tersebut. Ini adalah pola umum yang akan Anda lihat di banyak kode React Native, terutama saat tidak ada kunci eksplisit pada item data Anda, karena FlatList masih harus menemukan kunci, jadi Anda mungkin cukup mengekstrak beberapa elemen dari data Anda dan menjadikannya kunci agar FlatList tetap berfungsi. Sebagai contoh konkret, bayangkan Anda memiliki sekumpulan data seperti ini:

```
[{firstName:"Steve", lastName : "Rogers", { firstName : "Tony",
lastName : "Stark" } ]
```

Untuk menggunakannya dengan FlatList, masing-masing dari dua objek dalam array tersebut harus memiliki atribut kunci. Namun, saat ini keduanya tidak memilikinya, jadi Anda dapat menambahkannya secara eksplisit, atau Anda dapat menyediakan fungsi melalui keyExtractor. Mungkin fungsi tersebut

```
(item, index) => `avenger_${item.firstName}_${lastName}`
```

atau mungkin itu hanya

```
(item, index) => recordNumber++
```

Dengan asumsi recordNumber adalah variabel yang dapat diakses oleh fungsi tersebut yang dimulai dengan nilai 0, setiap item dalam array yang menjadi item dalam FlatList akan memiliki angka sebagai kuncinya, dengan kunci setiap item menjadi satu lebih besar dari item sebelumnya.

Jika Anda menyetel properti horizontal ke true, item-item tersebut akan ditampilkan berdampingan di layar, alih-alih penumpukan vertikal default. Anda juga dapat membalikkan

arah pengguliran dengan menyetel properti terbalik ke true. Anda dapat mengaitkan beberapa peristiwa melalui properti seperti onEndReached (untuk saat pengguna menggulir ke akhir daftar) atau onRefresh (untuk saat panggilan untuk mengambil lebih banyak data selesai). FlatList adalah komponen daftar yang kemungkinan besar akan paling sering Anda gunakan, tetapi itu bukan satu-satunya.

API

Sekarang setelah saya mensurvei komponen, mari kita lihat topik berikutnya dari pembahasan ini: API. API, tentu saja, adalah kumpulan fungsi yang dapat Anda panggil dalam kode aplikasi Anda untuk melakukan berbagai fungsi. API ini memberi Anda akses ke banyak kapabilitas perangkat asli, serta bagian fungsionalitas umum yang dibutuhkan banyak aplikasi. Setiap API memiliki tujuan tertentu, seperti yang akan Anda lihat.

Seperti komponen, beberapa API bersifat lintas platform, dan beberapa khusus untuk iOS atau Android. Nama API, seperti halnya komponen, memberi tahu Anda yang mana yang benar: jika tertulis IOS di akhir nama, itu khusus iOS; jika tertulis Android, itu khusus Android. Jika tidak tertulis keduanya, itu lintas platform. Selain itu, seperti komponen, Anda mengimpor API dengan cara yang sama seperti mengimpor komponen, karena, bagaimanapun juga, itu hanyalah modul JavaScript.

AccessibilityInfo

API AccessibilityInfo menawarkan fungsi yang terkait dengan masalah aksesibilitas, seperti menentukan apakah perangkat saat ini memiliki pembaca layar aktif yang terpasang. Anda dapat menggunakannya untuk menanyakan status pembaca layar saat ini, dengan memanggil metode fetch(). Atau, Anda dapat meminta agar beberapa kode yang Anda berikan dieksekusi saat status berubah, dengan memanggil metode addEventListener() dan menentukan peristiwa yang didukung (change, yang diaktifkan saat status pembaca layar berubah, atau announcefinished, yang untuk perangkat khusus iOS diaktifkan saat pembaca layar selesai membuat pengumuman).

Ada juga metode removeEventListener(), yang biasanya digunakan bersama dengan panggilan addEventListener() yang sesuai. Penggunaan yang umum adalah mendaftarkan beberapa kode dengan addEventListener() dalam metode componentDidMount() suatu komponen, lalu memanggil removeEventListener() dalam componentWillUnmount(). Ini memastikan tidak terjadi kebocoran memori, sesuatu yang cukup mudah dilakukan dengan event listener secara umum.

Sebagai contoh, berikut cara Anda dapat menggunakan API ini:

```
class ScreenReaderStatusExample extends React.Component {
  state = { isEnabled : false, };
  componentDidMount() {
    AccessibilityInfo.addEventListener("change", this.toggleState);
    AccessibilityInfo.fetch().done((isEnabled) => {
      this.setState({ isEnabled : isEnabled, });
   });
```

```
}
componentWillUnmount() {
  AccessibilityInfo.removeEventListener("change", this.toggleState);
}
toggleState = (isEnabled) => {
  this.setState({ isEnabled : isEnabled });
};
Render () {
  return (
   <View>
     <Text>
       The screen reader is {" "}
       {this.state.isEnabled ? "enabled": "disabled"}.
     </Text>
   </View>
  );
 }
}
```

Kiat Ini juga merupakan contoh bagus dari sesuatu yang sering muncul, yaitu bagaimana spasi diabaikan dalam komponen Teks. Lihat ekspresi {" "} setelah baris pertama? Itu perlu karena, tanpanya, kata is dan enabled atau disabled tidak akan memiliki spasi di antara keduanya, karena spasi di akhir dan awal konten di dalam komponen Teks diabaikan. Ini adalah trik praktis yang perlu diingat.

Alert dan AlertIOS

API Alert memiliki satu metode alert() yang meluncurkan dialog peringatan yang sesuai untuk platform dengan judul dan pesan yang ditampilkan. Dialog dapat memiliki satu atau beberapa tombol, yang dapat Anda tentukan sendiri atau izinkan tombol OK default untuk ditampilkan.

Contoh sederhananya adalah kode ini:

```
Alert.alert("Greetings, human!",
  "You know just how to push my buttons!",
  [ {text:"OK"} ], { cancelable : false }
 )
```

Di sini, judul peringatan adalah argumen pertama dan pesan opsional adalah yang kedua.

Meskipun ini adalah default, saya telah menetapkan tombol OK secara eksplisit, sebagai argumen ketiga, yang sepenuhnya boleh dilakukan, meskipun berlebihan. Meskipun API ini lintas platform, ada beberapa detail khusus platform yang perlu diperhatikan. Pertama, di iOS, Anda dapat menetapkan sejumlah tombol, dan setiap tombol dapat menentukan gaya, yang harus berupa default, batal, atau destruktif. Tombol destruktif berwarna merah, dan tombol batal dicetak tebal. Namun, di Android, paling banyak tiga tombol dapat ditetapkan,

dan Android memiliki konsep tombol netral, tombol negatif, dan tombol positif. Jika hanya satu tombol yang ditetapkan, tombol tersebut secara otomatis dianggap positif. Jika dua tombol disediakan, seperti Batal dan OK, satu negatif, dan satu positif. Untuk tiga tombol, setiap jenis akan diwakili oleh satu tombol.

Juga untuk Android, secara default, peringatan dapat ditutup dengan mengetuk di luar peringatan itu sendiri. Anda dapat menyediakan penangan panggilan balik, dengan menyediakan argumen keempat ke alert(), yang nilainya adalah objek yang berisi opsi. Jika Anda menyediakan atribut onDismiss pada objek yang merupakan fungsi tersebut, maka fungsi tersebut akan dijalankan saat peringatan ditutup dengan cara ini. Atau, Anda dapat menonaktifkan perilaku penutupan sepenuhnya, seperti yang ditunjukkan dalam contoh, dengan menyediakan atribut yang dapat dibatalkan yang disetel ke false.

Meskipun menampilkan informasi statis dalam peringatan merupakan kasus penggunaan umum, ada juga kasus penggunaan yang memerlukan input pengguna. Namun, ini bukanlah sesuatu yang didukung pada kedua platform. Hanya iOS yang memiliki kemampuan ini, jadi ada, selain API Peringatan, API AlertIOS. API ini, selain memiliki metode alert() seperti API Peringatan untuk menampilkan informasi statis, juga menyediakan metode prompt().

Metode prompt() diberikan judul dan teks sebagai dua argumen pertama, sama seperti metode alert(), tetapi sekarang argumen ketiga dapat berupa fungsi panggilan balik yang akan diberikan apa yang dimasukkan pengguna ke dalam peringatan. Anda masih dapat menyediakan tombol khusus, jika Anda mau, dan ada beberapa opsi konfigurasi yang tersedia, jika perintah tersebut ditujukan untuk informasi sensitif, seperti kata sandi, sehingga perintah tersebut akan disembunyikan saat dimasukkan, demi tujuan keamanan.

Sebagai contoh, berikut ini perintah peringatan untuk memasukkan kata sandi:

```
AlertIOS.prompt(
  "Password", "Please enter your password",
  (inPassword) => console.log("Your password is " + inPassword),
  "login-password", "???"
 );
```

Animasi

API Animasi adalah API yang cukup luas, dirancang untuk membuat animasi lancar, kuat, dan mudah dibuat serta dirawat. API ini dapat memiliki satu bab khusus yang didedikasikan untuknya. Banyak komponen dalam React Native menggunakan API ini di balik layar, meskipun Anda benar-benar dapat menggunakannya sendiri secara langsung. Ide dasar di balik API ini adalah untuk menentukan titik awal dan akhir untuk sebuah animasi, dengan sejumlah transformasi yang dapat dikonfigurasi di antara keduanya, yang berjalan selama periode waktu tertentu. Dengan kata lain, Anda menentukan status awal sesuatu, seperti warna teks tertentu, dan status akhir, seperti jika Anda ingin memudarkan teks secara bertahap dari merah menjadi hijau. Anda memberi tahu API ini hal-hal tersebut dan mengubah

antara teks merah dan teks hijau selama periode waktu tertentu, dan API ini akan melakukannya untuk Anda, tanpa Anda harus mencari tahu detailnya.

Karena API ini sangat luas, dan saya tahu saya tidak dapat menjelaskannya dengan adil dalam bagian yang singkat seperti ini, saya akan membiarkannya sebagai sesuatu untuk Anda jelajahi lebih lanjut sendiri. Karena itu, saya akan memberikan contoh singkat agar Anda memiliki gambaran dasar tentang seperti apa kode tersebut.

```
Animated.timing(this.state.fadeAnim, {
  toValue: 1, duration:1000, easing : Easing.inOut(Easing.ease), delay :
  1000
}).start();
```

Ya, itu dia! Metode timing() adalah salah satu metode Animated API yang paling umum digunakan, karena metode ini memungkinkan Anda mengubah nilai beberapa variabel selama periode waktu tertentu, secara opsional menggunakan kurva easing yang ditentukan, yang menunjukkan bagaimana nilai berubah seiring waktu, baik melalui perkembangan linier sederhana atau dengan memulai secara perlahan, semakin cepat, lalu melambat di akhir. Argumen pertama adalah nilai yang akan diubah seiring waktu, dan argumen kedua adalah objek konfigurasi.

Sebagian besar atribut dalam objek ini bersifat opsional, tetapi di sini saya telah menetapkan nilai akhir yang mewakili akhir animasi ini sebagai toValue (atribut ini, sebenarnya, diperlukan). Saya juga telah menetapkan durasi animasi (satu detik di sini, atau 1000 milidetik), kurva easing apa yang akan digunakan dari salah satu kurva yang telah ditentukan sebelumnya dalam Easing API (yang hanya berisi kumpulan anggota statis yang merupakan berbagai fungsi easing yang mungkin Anda gunakan). Perhatikan bahwa fungsi easing yang ditunjukkan di sini, sebenarnya, adalah fungsi default, jadi tidak perlu menentukannya. Ada pula penundaan, yang secara opsional memberi tahu Animated API berapa lama harus menunggu sebelum memulai animasi.

Perhatian Meskipun Animated API sangat hebat, penting untuk diketahui bahwa pada saat penulisan ini, Animated API hanya dapat digunakan pada komponen yang memungkinkan animasi, yaitu View, Text, Image, dan ScrollView. Anda juga dapat membuatnya sendiri menggunakan metode Animated.createAnimatedComponent(). Anda pasti ingin berkonsultasi dengan dokumen API ini untuk informasi lebih lanjut, terutama mengingat ada kemungkinan besar apa yang dapat dianimasikan akan berkembang seiring berjalannya waktu.

AppState

AppState API dapat digunakan untuk menentukan apakah aplikasi Anda berada di latar depan atau latar belakang dan secara opsional dapat memberi tahu kode Anda saat status tersebut berubah. Informasi ini paling umum digunakan untuk menentukan bagaimana aplikasi Anda harus bereaksi terhadap pemberitahuan push.

Aplikasi dapat berada dalam salah satu dari tiga status berikut:

- Aktif: Aplikasi berjalan di latar depan.

- Latar Belakang: Aplikasi berjalan di latar belakang (yang berarti pengguna berada di aplikasi lain, di layar beranda, atau, untuk Android, sedang melakukan aktivitas lain, meskipun diluncurkan dari aplikasi yang sama).
- Tidak Aktif: Aplikasi sedang bertransisi antara dua status sebelumnya atau telah lama tidak aktif (seperti yang Anda bayangkan, ini dan, sebenarnya, dua lainnya agak bergantung pada platform, tetapi React Native berupaya untuk menormalkannya, sehingga Anda dapat memperlakukannya sama di seluruh platform, untuk sebagian besar maksud dan tujuan).

Untuk menggunakan API ini, Anda dapat membaca atribut AppState.currentState atau Anda dapat menggunakan logika addEventListener() dan removeEventListener() yang sama seperti yang Anda lihat saat kita melihat AccessibilityInfo, agar beberapa kode yang Anda tentukan dieksekusi setiap kali status aplikasi berubah.

BackHandler

API BackHandler hanya untuk Android dan sangat sederhana. Ia hanya menawarkan metode addEventListener() dan removeEventListener() (ada juga metode exitApp(), dan ada dugaan kuat bahwa ia memaksa keluar dari aplikasi, tetapi itu hanya dugaan, karena tidak ada dokumentasi yang dapat saya temukan). Anda memanggil addEventListener(), meminta panggilan balik untuk acara hardwareBackPress, dan menerapkan logika apa pun yang sesuai saat tombol kembali ditekan, jika ada.

Perlu dicatat bahwa Anda dapat memiliki beberapa langganan untuk acara tertentu, dan pengendali dipanggil dalam urutan terbalik, yaitu, yang terakhir didaftarkan akan dipanggil terlebih dahulu. Mereka akan terus dipanggil dalam urutan terbalik, kecuali dan sampai salah satu mengembalikan true, dalam hal ini, urutannya berakhir.

Clipboard

API Clipboard adalah API langsung yang memberi Anda akses ke clipboard perangkat, menggunakan metode getString() dan setString(). Jadi, untuk menyimpan string ke clipboard, lakukan hal berikut:

```
Clipboard.setString("My string");
```

Untuk mengambil string dari clipboard, gunakan

```
let s = await Clipboard.getString();
```

Karena getString() mengembalikan sebuah promise, menggunakan await adalah cara yang baik untuk mengatasinya.

Ingatlah bahwa agar await berfungsi, fungsi apa pun yang memuat kode ini harus diawali dengan async (atau Anda dapat menangani promise dengan cara lain, jika Anda mau; itu pilihan Anda). Perhatikan bahwa Anda tidak dapat menyimpan apa pun kecuali string ke

clipboard hanya dengan API ini, setidaknya tidak tanpa memperkenalkan beberapa kode asli tambahan ke proyek Anda. Namun, jika Anda dapat membuat serial objek menjadi string, Anda dapat menyimpannya di clipboard, jadi pengodean gambar Base64, misalnya, dan menyimpannya ke clipboard adalah suatu kemungkinan.

Dimensi

API Dimensions adalah API sederhana yang menyediakan sarana untuk mendapatkan dimensi layar perangkat. Panggilan ke metode get()-nya, yang meneruskan dimensi yang Anda inginkan (window menjadi satu-satunya nilai yang didokumentasikan saat ini), mengembalikan objek dengan berbagai atribut, seperti yang dapat Anda lihat di sini:

```
{ "fontScale": 0.8999999761581421, "scale": 3.5,
  "width": 411.42857142857144, "height": 731.4285714285714 }
```

Atribut lebar dan tinggi mungkin adalah atribut yang paling menarik bagi Anda, tetapi siapa saya untuk menghakimi?

Selain itu, API ini menyediakan pasangan addEventListener() dan removeEventListener() dan perubahan jenis peristiwa yang aktif setiap kali ada perubahan dimensi. Mungkin tampak aneh bahwa dimensi layar dapat bervariasi, tetapi ada satu contoh utama di mana keduanya dapat berubah: saat layar berputar. Lebar dan tinggi tentu saja akan bertukar, dan Anda mungkin ingin memiliki kode yang mengubah tata letak saat beralih dari lanskap ke potret dan sebaliknya, dan API ini menyediakan cara untuk melakukannya.

Geolokasi

API Geolokasi agak aneh, karena merupakan komponen atau API pertama yang Anda lihat yang tidak perlu Anda impor secara eksplisit. Itu karena API ini adalah polyfill browser yang memperluas spesifikasi web Geolokasi, dan, dengan demikian, secara otomatis tersedia pada objek global navigator.geolocation. Saya belum pernah menyebutkan cakupan global sebelumnya, dan itu karena, pada umumnya, Anda tidak akan menggunakannya secara langsung dalam kode Anda, tetapi ada situasi di mana Anda akan menggunakannya, dan API ini adalah salah satunya.

Ada cukup banyak hal dalam cakupan global, yang dapat Anda lihat sendiri dengan meletakkan baris console.log(global); dalam beberapa kode dan melihat apa yang dibuang ke konsol. Variabel global adalah variabel khusus yang mewakili cakupan global, yang berarti Anda dapat menggunakan global.navigator.geolocation, dan itu akan berfungsi. Tetapi React Native melakukan beberapa keajaiban di balik layar untuk membuatnya, jadi Anda tidak perlu merujuk atribut global seperti itu. Anda dapat mengakses hal-hal seperti navigator secara langsung.

Selain cakupan global, API ini mudah. Ada metode setRNConfiguration() opsional yang memungkinkan Anda mengatur konfigurasi untuk permintaan lokasi (saat ini hanya ada satu opsi khusus iOS, skipPermissionRequests, yang, jika benar, memaksa pengguna untuk menyetujui permintaan posisi). Lalu ada metode yang mungkin paling menarik bagi Anda: getCurrentPosition(). Metode ini menerima fungsi untuk dipanggil saat penentuan lokasi berhasil, fungsi untuk dipanggil jika terjadi kesalahan, dan opsi. Opsi tersebut meliputi

timeout, untuk menetapkan batas waktu menunggu lokasi, maximumAge, yang menentukan berapa lama lokasi yang di-cache berlaku, dan enableHighAccuracy, yang di Android hanya memungkinkan mendapatkan lokasi dengan akurasi lebih tinggi dari biasanya.

Atau, Anda dapat menggunakan metode watchLocation(), untuk memanggil fungsi yang Anda berikan setiap kali lokasi berubah. Metode ini memiliki penangan sukses, penangan kesalahan, dan daftar parameter opsi yang sama seperti getCurrentPosition(), tetapi memiliki beberapa opsi tambahan yang tersedia. Opsi tersebut adalah distanceFilter dan useSignificantChanges, yang sejujurnya, tidak didokumentasikan di mana pun yang dapat saya temukan, jadi saya tidak dapat mengatakan dengan pasti untuk apa keduanya. Namun, saya rasa nama keduanya memungkinkan tebakan yang masuk akal, yang akan saya serahkan kepada Anda. Ada metode clearWatch() yang sesuai, untuk berhenti memantau perubahan lokasi, dan metode stopObserving(), yang tampaknya melakukan hal yang sama seperti clearWatch(), kecuali bahwa metode ini secara otomatis menghapus semua listener yang ditambahkan dan menghentikan API itu sendiri dari memantau perubahan lokasi, baik kode Anda memantaunya (sebagai tindakan penghematan daya).

Sebagai contoh, API Geolokasi dapat digunakan sebagai berikut:

```
navigator.geolocation.getCurrentPosition(
  (position) => { console.log("getCurrentPosition()", position); },
  (error) => { console.log("getCurrentPosition() error", error); },
  { enableHighAccuracy : true, timeout : 20000, maximumAge : 1000 }
  );
```

Dan output dari ini akan menjadi

```
getCurrentPosition() Object {
  "coords": Object {
    "accuracy": 65,
    "altitude": 41.387497487387 46,
    "altitudeAccuracy": 10,
    "heading": -1,
    "latitude": 32.38729736476293,
    "longitude": -22.192837464023,
    "speed": -1,
 },
 "timestamp": 1527607301269.7148,
}
```

InteractionManager

API InteractionManager adalah API yang memungkinkan tugas yang berjalan lama dijadwalkan, sehingga tugas tersebut hanya akan dijalankan setelah interaksi sentuhan pengguna dan/atau animasi selesai agar semuanya berjalan lancar. API ini hanya berisi

beberapa metode.

- runAfterInteractions: Menjadwalkan fungsi tertentu untuk dijalankan setelah semua interaksi selesai
- createInteractionHandle: Memberitahu manajer bahwa interaksi telah dimulai
- clearInteractionHandle: Memberitahu manajer bahwa interaksi telah selesai
- setDeadline: Menentukan nilai batas waktu untuk menjadwalkan tugas apa pun setelah eventLoopRunningTime mencapai nilai batas waktu; jika tidak, semua tugas akan dijalankan dalam satu batch setImmediate (yang merupakan default)

Jika hal itu tampak sedikit rumit, mungkin contoh akan membantu memperjelasnya.

```
InteractionManager.runAfterInteractions(() => {
   // Implement some long-running task here
 });
 const handle = InteractionManager.createInteractionHandle();
 // Now run any animations you need to
 InteractionManager.clearInteractionHandle(handle);
 // All tasks queued up via runAfterInteractions() are now executed
```

Keyboard

API Keyboard memungkinkan kode Anda untuk mendengarkan peristiwa keyboard asli yang menarik dan bereaksi terhadapnya dengan cara khusus aplikasi dan juga menyediakan beberapa kontrol atas keyboard, seperti menutupnya.

Pertama, ada metode addListener(), yang memungkinkan Anda mendengarkan salah satu peristiwa berikut, yang menurut saya namanya cukup menjelaskan tentang peristiwa tersebut:

Perlu dicatat bahwa saya tidak dapat menentukan apa sebenarnya keyboardWillChangeFrame dan keyboardDidChangeFrame, tetapi keduanya tampaknya khusus untuk iOS. Hei, dokumen React Native cukup bagus, tetapi bab ini membuktikan bahwa pasti ada beberapa celah, dan ini salah satunya.

- ✓ `keyboardWillShow`
- ✓ `keyboardDidShow`
- ✓ `keyboardWillHide`
- ✓ `keyboardDidHide`
- ✓ `keyboardWillChangeFrame`
- ✓ `keyboardDidChangeFrame`

Tentu saja, ada removeListener() yang sesuai, sesuai dengan pola yang telah Anda lihat beberapa kali sebelumnya. Namun, kali ini, ada juga metode removeAllListeners(), jika Anda lebih suka yang singkat. Terakhir, metode dismiss() melakukan hal yang sama: mengabaikan keyboard.

Ini adalah API sederhana, tetapi, kita berbicara tentang keyboard, bukan puncak kecerdikan manusia, jadi saya rasa tidak perlu terlalu mendalam.

LayoutAnimation

API LayoutAnimation memungkinkan Anda memberi tahu React Native untuk secara otomatis menganimasikan tampilan ke posisi barunya saat bergerak saat tata letak berikutnya terjadi. Kasus penggunaan umum adalah memanggil API ini sebelum panggilan ke setState(), sehingga jika perubahan status tersebut mengakibatkan tampilan apa pun bergerak, tampilan tersebut dapat dianimasikan saat melakukannya.

Meskipun ada tiga metode dalam API ini, dua di antaranya, create () dan checkConfig(), adalah metode pembantu opsional (yang kebetulan tidak terdokumentasi dengan baik). Satu metode penting adalah configureNext (). Ini menerima dua argumen: objek konfigurasi (yang dapat dibuat dengan metode create () tersebut, tetapi tidak harus demikian) dan, secara opsional, fungsi yang akan dipanggil saat animasi berakhir (yang hanya didukung di iOS). Objek konfigurasi berisi tiga atribut. Yang pertama adalah duration, yang merupakan lamanya waktu animasi dalam milidetik. Yang kedua adalah create, yang merupakan tipe Anim yang menentukan animasi yang akan digunakan untuk tampilan yang akan ditampilkan. Terakhir, ada update, yang juga merupakan tipe Anim tetapi menjelaskan animasi yang akan digunakan untuk tampilan yang sudah terlihat tetapi telah diperbarui.

NetInfo

API NetInfo menyediakan beberapa metode dan properti yang memungkinkan Anda menentukan status konektivitas perangkat. Cara pertama untuk menggunakan API ini adalah melalui metode getConnectionInfo().

```
NetInfo.getConnectionInfo().then((inConnectionInfo) => {
   console.log("getConnectionInfo()", inConnectionInfo);
});
```

Ini mengembalikan promise yang diselesaikan ke objek dengan dua atribut, `type` dan `effectiveType`. Atribut type bertipe `NetInfo.ConnectionType`, yang merupakan enum dengan nilai none, `wifi`, `cellular`, dan `unknown` dan tiga tipe khusus Android: `bluetooth`, `ethernet`, atau `wimax`. `EffectiveType bertipe NetInfo`. EffectiveConnectionType dan merupakan enum yang selanjutnya mendefinisikan tipe koneksi, saat tipenya adalah cellular, dengan nilai 2g, 3g, 4g, dan unknown.

Anda juga dapat memantau status jaringan dengan menggunakan metode addEventListener() (dan metode removeEventListener() yang sesuai, tentu saja), untuk mendengarkan peristiwa connectionChange, yang dipicu saat status jaringan berubah. Ada juga properti isConnected yang dapat mengambil Boolean secara asinkron untuk menentukan apakah konektivitas Internet saat ini tersedia. Untuk menggunakannya, Anda menulis

```
NetInfo.isConnected.fetch().then(isConnected => {
 console.log(`We are currently ${isConnected ? "Online":"Offline"}`);
});
```

Terakhir, metode ini, hanya untuk Android, menyediakan metode isConnectionExpensive() yang memberi tahu Anda apakah koneksi diukur. Contoh penggunaan terlihat mirip dengan penggunaan isConnected().

```
NetInfo.isConnectionExpensive()
.then(isConnectionExpensive => {
  console.log(`Connection is ${(isConnectionExpensive?"Metered":"Not
Metered")}`);
})
.catch(error => { console.log("isConnectionExpensive() not supported on
iOS");});
```

Tentu saja, untuk menjaga kode Anda lintas platform, penangan kesalahan harus digunakan di sini, untuk menghindari masalah pada perangkat iOS, oleh karena itu kode ini sedikit berbeda dari kode untuk isConnected().

Catatan Selain isConnectionExpensive(), Android menyediakan informasi lebih lanjut tentang konektivitas jaringan daripada iOS, termasuk apakah perangkat terhubung ke Bluetooth, apakah terhubung melalui Ethernet, atau apakah terhubung melalui WiMAX. Anda pasti ingin memeriksa dokumen untuk API ini, untuk memastikan bahwa kode yang Anda tulis bersifat lintas platform atau akan berfungsi di iOS vs. Android.

PixelRatio

API PixelRatio memberi Anda akses ke informasi tentang kepadatan piksel perangkat. Ini berguna, karena aplikasi modern diharapkan menggunakan gambar beresolusi lebih tinggi, jika perangkat memiliki tampilan kepadatan tinggi, jadi kemampuan untuk menentukannya dalam kode Anda sangat penting. Metode yang disediakan oleh API ini adalah

- ✓ get(): Mengembalikan kerapatan piksel perangkat, contoh terbatasnya meliputi 1 (perangkat Android mdpi @ 160 dpi), 1.5 (perangkat Android hdpi @ 240 dpi), 2 (iPhone 4/4S/5/5c/5s/6), 3 (iPhone 6 plus), dan seterusnya
- ✓ getFontScale(): Mengembalikan faktor skala untuk ukuran font, yaitu, rasio yang digunakan untuk menghitung ukuran font absolut (atau rasio piksel perangkat, jika skala font tidak ditetapkan secara eksplisit) untuk Android saja. (Ini mencerminkan preferensi pengguna Setelan ➤ Tampilan ➤ Ukuran font, dan di iOS, ini akan selalu mengembalikan rasio piksel default.)
- ✓ getPixelSizeForLayoutSize(): Mengonversi ukuran tata letak (dp) ke ukuran piksel (px) (dijamin akan mengembalikan angka integer)
- ✓ roundToNearestPixel(): Membulatkan ukuran tata letak (dp) ke ukuran tata letak terdekat yang sesuai dengan angka integer piksel

Seperti yang mungkin dapat Anda tebak dari jenis nilai yang ditampilkan di sini, Anda benar-benar harus merujuk ke dokumentasi untuk menentukan nilai apa yang benar-benar dapat Anda peroleh pada perangkat tertentu dan cara menanganinya. Faktanya, ini adalah satu kali saya akan mengutip langsung dari dokumen React Native, karena saya rasa saya tidak dapat

menjelaskannya dengan lebih baik.

Mengambil gambar dengan ukuran yang benar

Anda akan mendapatkan gambar beresolusi lebih tinggi jika Anda menggunakan perangkat dengan kepadatan piksel tinggi. Aturan praktis yang baik adalah mengalikan ukuran gambar yang Anda tampilkan dengan rasio piksel.

```
var image = getImage({
width: PixelRatio.getPixelSizeForLayoutSize(200),
height: PixelRatio.getPixelSizeForLayoutSize(100),
});
<Image source={image} style={{width: 200, height: 100}} />
```

Pixel grid snapping

Di iOS, Anda dapat menentukan posisi dan dimensi untuk elemen dengan presisi yang sembarangan, misalnya, 29.674825. Namun, pada akhirnya, tampilan fisik hanya memiliki jumlah piksel yang tetap, misalnya, 640×960 untuk iPhone 4 atau 750×1334 untuk iPhone 6. iOS mencoba untuk setepat mungkin dengan nilai pengguna, dengan menyebarkan satu piksel asli menjadi beberapa piksel untuk menipu mata. Kelemahan dari teknik ini adalah membuat elemen yang dihasilkan terlihat buram.

Dalam praktiknya, kami menemukan bahwa pengembang tidak menginginkan fitur ini, dan mereka harus mengatasinya dengan melakukan pembulatan manual, untuk menghindari elemen yang buram. Di React Native, kami membulatkan semua piksel secara otomatis. Kami harus berhati-hati saat melakukan pembulatan ini. Anda tidak ingin bekerja dengan nilai yang dibulatkan dan tidak dibulatkan secara bersamaan, karena Anda akan mengumpulkan kesalahan pembulatan. Bahkan jika ada satu kesalahan pembulatan, itu bisa berakibat fatal, karena batas satu piksel bisa hilang atau menjadi dua kali lebih besar.

Dalam React Native, semua hal dalam JavaScript dan dalam mesin tata letak bekerja dengan angka presisi yang berubah-ubah. Kita hanya bisa membulatkan nilai saat kita mengatur posisi dan dimensi elemen asli pada utas utama. Selain itu, pembulatan dilakukan relatif terhadap akar, bukan induk, sekali lagi untuk menghindari akumulasi kesalahan pembulatan.

Platform

API Platform adalah sesuatu yang akan sering Anda gunakan di tempat-tempat yang mengharuskan Anda membuat cabang kode, berdasarkan platform tempat kode itu dijalankan. API ini mudah digunakan, menyediakan tiga hal utama.

Pertama, adalah anggota statis bernama OS yang memberi tahu Anda nama platform (string dengan nilai ios atau android). Ada juga atribut Version yang menyediakan informasi tentang versi platform (untuk Android, ini akan menjadi level API; untuk iOS, ini adalah string dalam bentuk major.minor, 10.3, misalnya). Anda tentu saja dapat menulis logika percabangan apa pun dengan nilai-nilai ini sesuai keinginan Anda. Misalnya, Anda mungkin harus menyesuaikan tinggi gaya, berdasarkan platform.

```
const styles = StyleSheet.create({
  height : Platform.OS === "ios" ? 250 : 125,
});
```

Atau, mungkin Anda perlu menggunakan API yang hanya tersedia pada versi Android tertentu:

```
if (Platform.Version === 25) {
  // Make version-dependent call here
}
```

Hal lain yang disediakan API ini adalah metode select(). Metode ini sering digunakan dalam definisi gaya, seperti contoh sebelumnya, tetapi sedikit berbeda.

```
const styles = StyleSheet.create({
 container:{
    flex : 1,
    ...Platform.select({
      ios : { backgroundColor : "#ff0000" },
      android: { backgroundColor : "#00ff00" }
    })
  }
});
```

Hasil dari ini akan menjadi StyleSheet (API berikutnya yang akan kita lihat) yang memiliki fleksibilitas 1 dan latar belakang merah di iOS serta hijau di Android. Metode percabangan yang Anda gunakan benar-benar tergantung pada preferensi.

Satu cara terakhir agar API ini dapat digunakan adalah dengan memilih komponen yang bergantung pada platform.

```
const Component = Platform.select({
   ios: () => require("ComponentIOS"),
   android: () => require("ComponentAndroid"),
})();
<Component />
```

Dengan cara ini, Anda dapat mengubah tata letak dan hierarki komponen, berdasarkan platform tempat aplikasi berjalan. Seperti yang dapat Anda duga, Anda akan melihat lebih banyak tentang API ini dalam bab-bab berikutnya, karena ini adalah kunci untuk membuat aplikasi yang berfungsi dengan baik di berbagai platform.

StyleSheet

Anda telah melihat StyleSheet API beraksi melalui metode StyleSheet.create(), jadi Anda sudah tahu bahwa API tersebut mengambil objek JavaScript dan mengembalikan objek

StyleSheet baru darinya. Mengapa kita melakukan ini? Ya, karena beberapa alasan. Pertama, seperti halnya CSS di Web, Anda membuat kode lebih mudah dipahami dengan tidak memasukkan semua gaya dalam metode render (). Metode ini juga menyediakan organisasi dan pemisahan masalah yang lebih diterima secara umum. Selain itu, dengan memberi nama gaya, Anda membantu menambahkan makna pada komponen tingkat rendah dalam metode render ().

Ketiga, saat Anda memasukkan gaya, objek StyleSheet baru secara otomatis dibuat di balik layar, yang berarti kapan pun metode render () dipanggil, yang berarti kapan pun tata letak berubah, yang berarti hal ini dapat sering terjadi! Itu tidak baik untuk kinerja. Terakhir, saat gaya didefinisikan dengan cara ini, gaya tersebut dikirim melalui jembatan render hanya sekali dan di-cache dengan penggunaan berikutnya yang mencarinya berdasarkan ID. (Namun, menurut dokumen pada saat penulisan, hal ini tampaknya belum sepenuhnya diterapkan.)

Selain create(), ada juga metode flatten() yang mengambil serangkaian gaya dan mengembalikan satu objek dengan semua gaya yang digabungkan. Misalnya:

```
const stylesAPITest = StyleSheet.create({
   style1 : { flex : 1, fontSize : 12, color : "red" },
   style2 : { color : "blue" },
});
const stylesAPITestNew =
   StyleSheet.flatten([stylesAPITest.style1, stylesAPITest.style2]);
console.log("stylesAPITestNew", stylesAPITestNew);
```

Ini akan menampilkan

```
stylesAPITestNew Object {
   "flex": 1,
   "fontSize": 12,
   "color": "blue"
}
```

Perhatikan bagaimana atribut warna mengambil nilai item terakhir dalam array.

ToastAndroid

Sesuai namanya, API ini hanya untuk Android dan menyediakan akses ke fasilitas pesan toast yang ditawarkan platform tersebut. Ini adalah pesan pop-up pendek yang biasanya memberi tahu pengguna tentang beberapa tindakan yang telah diselesaikan. Menampilkannya dengan API berikut sangatlah mudah:

```
<Button title="Show Toast Message (Android Only)"
  onPress={ async () => {
   ToastAndroid.show("I am a short message", ToastAndroid.SHORT);
   ToastAndroid.showWithGravity(
```

```
      "I am a message with gravity, centered",
      ToastAndroid.SHORT, ToastAndroid.CENTER
    );
    ToastAndroid.showWithGravityAndOffset(
      "I am a message with gravity, offset from the bottom",
      ToastAndroid.LONG,ToastAndroid.TOP,-75,
      Dimensions.get("window").height / 2
     );
    }}
/>
```

Di sini, tiga pesan toast yang berbeda ditampilkan. API secara otomatis mengantrekannya, sehingga yang kedua tidak muncul hingga yang pertama ditutup (baik secara otomatis setelah beberapa waktu atau melalui pengguna yang mengekliknya), dan yang ketiga juga tidak muncul hingga yang kedua ditutup.

1.5 VISUAL YANG DIDEFINISIKAN SEPENUHNYA DALAM JAVASCRIPT

JSX dalam React Native

JSX, yang merupakan singkatan dari JavaScript XML, adalah gabungan dari tiga elemen utama: JavaScript, HTML (atau lebih tepatnya XML), dan CSS. Secara lebih spesifik, JSX memungkinkan Anda untuk menanamkan kode XML langsung ke dalam JavaScript tanpa menghadapi banyak kesalahan sintaksis.

Kode JSX ini kemudian diproses oleh React Native dan diubah menjadi JavaScript biasa yang dapat dieksekusi saat runtime. Meskipun Anda bisa saja menulis aplikasi React Native sepenuhnya dalam JavaScript murni tanpa menggunakan JSX, ini bukan cara umum yang digunakan dalam pengembangan aplikasi React Native, dan itu juga bukan pendekatan yang akan dibahas dalam buku ini. JSX sebenarnya membuat pengembangan menjadi lebih mudah dan lebih efisien, karena menulis dengan JSX jauh lebih sederhana dan lebih langsung dibandingkan dengan menulisnya dalam JavaScript murni.

Untuk menjelaskannya secara lebih konkret, Anda dapat menuliskan semua kode React Native Anda dalam bentuk ini:

```
React.createElement(
  "div", null,
  React.createElement("img", {src: url, className: "contactPhoto"}),
  React.createElement("span", {className:"contactName"}, firstName +
  lastName)
);
```

Pada titik ini, tidak penting bagi Anda untuk memahami kode tersebut, meskipun saya yakin bahwa jika Anda cukup memperhatikannya, Anda dapat membuat beberapa tebakan yang sangat masuk akal tentang apa yang terjadi dan memperoleh pemahaman kasar

tentangnya. Namun, bandingkan kode tersebut dengan yang berikut, yang sama tetapi dalam bentuk JSX:

```
<div>
   <img src={url} className="contactPhoto" />
   <span className="contactName">{firstName + lastName}</span>
 </div>
```

Meskipun ini terdengar sangat gila dan keterlaluan, dengarkan saya. Selain menggunakan sintaksis yang sangat aneh, templat HTML secara tradisional mengalami masalah besar lainnya. Berbagai hal yang dapat Anda lakukan di dalamnya selain sekadar menampilkan data terbatas. Jika Anda ingin memilih bagian UI mana yang akan ditampilkan berdasarkan kondisi tertentu, misalnya, Anda harus menulis JavaScript di tempat lain di aplikasi Anda atau menggunakan beberapa perintah templating khusus kerangka kerja yang aneh agar berfungsi. Misalnya, berikut ini adalah tampilan pernyataan kondisional di dalam templat EmberJS:

```
{{#if person}}
  Welcome back, <b>{{person.firstName}} {{person.lastName}}</b>!
{{else}}
  Please log in.
{{/if}}
```

Sekali lagi, Anda mungkin belum sepenuhnya memahami React Native, tetapi ini bisa menjadi ujian yang baik: apakah kode ini terasa sedikit lebih jelas? Apakah menurut Anda kode ini tampak lebih bersih dan mudah dipahami? Kebanyakan orang, setelah melewati kebingungannya terhadap JSX karena terlihat seperti mencampur HTML dan JavaScript dengan cara yang aneh, akhirnya merasa bahwa JSX adalah cara yang lebih bersih untuk menulis kode. Itulah salah satu keuntungan utama dari JSX: kode ini cenderung lebih langsung dan lebih mudah dibaca, yang menjadikannya cara standar untuk menulis aplikasi React Native.

Struktur Kode JSX dan React Native

Kode JSX, seperti yang ditemukan dalam aplikasi pertama, biasanya berada dalam file .js. Kode ini dimulai dengan pernyataan impor JavaScript standar yang menyertakan kelas inti React, yang menjadi dasar dalam pengembangan React Native. Setelah itu, pernyataan impor tambahan digunakan untuk memasukkan komponen-komponen React Native yang diperlukan, yang merupakan konsep terpisah yang akan dijelaskan lebih lanjut. Dalam aplikasi sederhana ini, ada tiga komponen utama yang diperlukan: StyleSheet, Text, dan View.

Baris yang dimulai dengan export default class mungkin terlihat aneh, karena jika Anda berpikir dalam konteks JavaScript biasa, Anda mungkin berpikir, "Ini tidak akan berfungsi; ini tidak valid secara sintaksis!" Dan Anda benar, dari perspektif JavaScript biasa, itu memang tampak tidak valid. Namun, dalam dunia JSX, ini sepenuhnya sah meskipun terlihat aneh. Saya akan menjelaskan lebih lanjut tentang apa yang terjadi di bagian berikutnya.

Seperti halnya HTML atau XML, tag dalam JSX dapat berisi konten (misalnya, <Text>Hello</Text>), yang berarti tag tersebut memiliki tag pembuka dan penutup, atau dapat juga berupa tag yang menutup dirinya sendiri (seperti <View />). Tag dalam JSX juga dapat memiliki atribut, mirip dengan tag HTML atau XML. Atribut ini akan dijelaskan lebih rinci nanti, jadi tidak perlu dibahas lebih lanjut untuk saat ini. Terakhir, tag dalam JSX dapat memiliki *anak*, yang bisa berupa teks, ekspresi JavaScript, atau bahkan tag JSX lainnya.

Banyak framework lain menyediakan bahasa templating yang memungkinkan Anda untuk menyematkan kode di dalam template. Hal ini mirip dengan HTML, di mana Anda bisa menyematkan cuplikan JavaScript di dalam markup. Namun, React Native membawa konsep ini lebih jauh. Di React Native, Anda menulis kode yang memiliki markup di dalamnya, memungkinkan Anda untuk menggabungkan tampilan dan logika JavaScript dalam satu tempat yang lebih efisien.

Sekarang, ketika Anda memikirkan visual yang didefinisikan sepenuhnya dalam JavaScript, Anda mungkin memikirkan sesuatu yang mengerikan yang melibatkan tanda kutip, karakter escape, dan banyak sekali panggilan createElement. Jangan khawatir. React memberi Anda opsi untuk menentukan visual Anda menggunakan sintaksis seperti HTML yang dikenal sebagai JSX yang sepenuhnya ada bersama JavaScript Anda. Alih-alih menulis kode untuk mendefinisikan UI Anda, Anda pada dasarnya menentukan markup:

```
ReactDOM.render (
  <div>
    <h1>Batman</h1>
    <h1>Iron Man</h1>
    <h1>Nicolas Cage</h1>
    <h1>Mega Man</h1>
  </div>,
  destination
):
```

Kode yang serupa di JavaScript:

```
ReactDOM.render (React.createBlement (
  "div",
  null,
  React.createBlement (
    "h1",
    null,
    "Batman"
  ),
  React.createBlement (
    "h1",
    null,
    "Iron Man"
  ),
  React.createBlement (
    "h1",
    null,
    "Nicolas Cage"
  ),
  React.createElement (
```

```
        "h1",
        null,
        "Mega Man"
    )
) , destination);
```

Dengan menggunakan JSX, Anda dapat mendefinisikan elemen visual dengan mudah menggunakan sintaksis yang sangat familiar, sambil tetap memanfaatkan semua kekuatan dan fleksibilitas yang ditawarkan oleh JavaScript. Salah satu keuntungan utama dari JSX adalah bahwa di React, elemen visual dan logika JavaScript sering kali berada dalam file yang sama. Ini berarti Anda tidak perlu lagi berpindah-pindah antara berbagai file untuk mendefinisikan tampilan dan perilaku sebuah komponen visual. Ini adalah cara templating yang lebih efisien dan terorganisir dengan baik.

Hanya V dalam Arsitektur MVC

Kita hampir selesai di sini! React bukanlah kerangka kerja lengkap yang memiliki pendapat tentang bagaimana segala sesuatu di aplikasi Anda seharusnya berperilaku. Sebaliknya, React bekerja terutama di lapisan View di mana semua kekhawatiran dan perhatiannya berputar di sekitar elemen visual Anda dan menjaganya agar tetap terkini.

Ini berarti Anda bebas menggunakan apa pun yang Anda inginkan untuk bagian M dan C dari arsitektur MVC Anda. Fleksibilitas ini memungkinkan Anda untuk memilih dan memilah teknologi apa yang Anda kenal, dan ini membuat React berguna tidak hanya untuk aplikasi web baru yang Anda buat tetapi juga untuk aplikasi yang sudah ada yang ingin Anda tingkatkan tanpa menghapus dan memfaktorkan ulang banyak kode.

1.6 KESIMPULAN

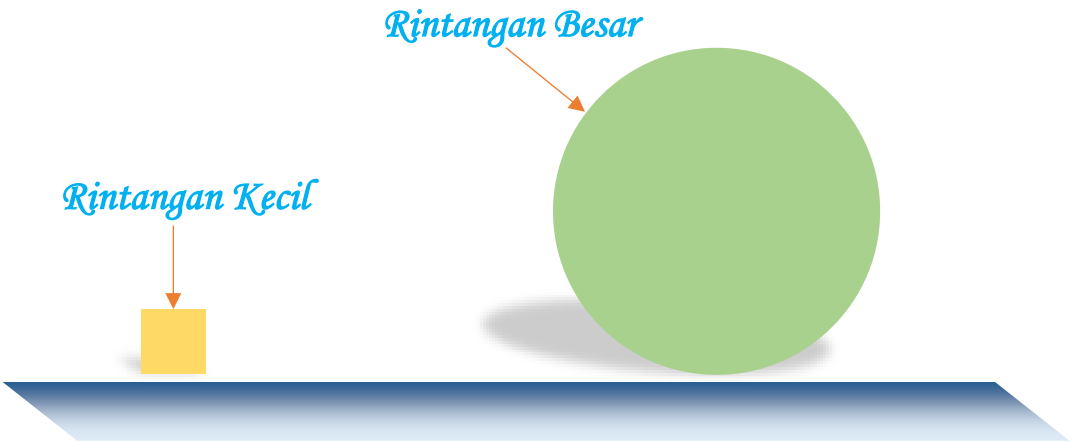
Sebagai kerangka kerja dan pustaka web baru, React cukup sukses besar. React tidak hanya mengatasi masalah paling umum yang dihadapi pengembang saat membangun aplikasi satu halaman, tetapi juga memberikan beberapa trik tambahan yang membuat pembuatan visual untuk aplikasi satu halaman Anda JAUH LEBIH mudah. Sejak dirilis pada tahun 2013, React terus menemukan jalannya ke situs web dan aplikasi populer yang mungkin Anda gunakan. Selain Facebook dan Instagram, beberapa yang terkenal termasuk BBC, Khan Academy, PayPal, Reddit, The New York Times, Yahoo, dan masih banyak lagi.

Tujuan dari bab ini adalah untuk memberi Anda pengantar tentang apa yang dilakukan React dan mengapa melakukannya. Dalam tutorial di bab-bab berikutnya, kami akan membahas lebih dalam semua yang telah Anda lihat di sini dan membahas detail teknis yang akan membantu Anda berhasil menggunakan React dalam proyek Anda sendiri. Tetaplah di sini.

BAB 2
MEMBANGUN APLIKASI REACT PERTAMA

Sekarang, berkat bab sebelumnya, Anda mungkin sudah tahu semua tentang latar belakang React dan bagaimana React membantu antarmuka pengguna Anda yang paling rumit sekalipun agar berfungsi dengan baik. Dengan segala kehebatan yang dihadirkan React, memulainya (mirip seperti kalimat ini) bukanlah hal yang mudah. React memiliki kurva pembelajaran yang curam yang dipenuhi dengan banyak rintangan kecil dan besar.

Dalam bab ini, kita mulai dari awal dan langsung terjun ke lapangan dengan membangun aplikasi React yang sederhana. Kita menghadapi beberapa rintangan ini secara langsung, dan beberapa rintangan ini kita lewati untuk saat ini. Di akhir bab ini, kita tidak hanya akan membangun sesuatu yang dapat Anda pamerkan dengan bangga kepada teman dan keluarga, kita juga akan mempersiapkan diri dengan baik untuk menyelami lebih dalam semua yang ditawarkan React di bab-bab selanjutnya.



Gambar 2.1

2.1 MEMBANGUN APLIKASI SEDERHANA DENGAN REACT

Membangun aplikasi sederhana menggunakan React adalah cara yang menyenangkan dan efektif untuk belajar pengembangan antarmuka pengguna. Langkah pertama, pastikan Anda telah menginstal Node.js, yang diperlukan untuk menjalankan aplikasi React. Anda juga akan memerlukan npm atau yarn, yang biasanya sudah terinstal bersama Node.js.

Untuk memulai, Anda dapat menggunakan Create React App dengan menjalankan perintah berikut di terminal:

lua

```
npx create-react-app my-app
```

Setelah itu, masuk ke folder proyek dengan perintah:

bash

```
cd my-app
```

Kemudian, jalankan aplikasi dengan:
sql

```
npm start
```

Aplikasi Anda akan berjalan di `http://localhost:3000`, dan Anda dapat melihat tampilan awal aplikasi di browser.

Struktur folder proyek yang dibuat akan terdiri dari dua direktori utama: public dan src. File utama aplikasi, yaitu `index.html`, terletak di folder public, sementara komponen utama aplikasi ada di `src/App.js`. Untuk memulai, Anda bisa mengedit App.js dan menambahkan tag `<h1>` untuk menampilkan pesan "Hello, World!".

Selanjutnya, Anda dapat menambahkan interaksi sederhana menggunakan state. Dengan menggunakan hook `useState`, Anda bisa membuat tombol yang akan meningkatkan penghitung setiap kali diklik. Berikut adalah contoh kode untuk memperbarui state `count` ketika tombol ditekan. Selain itu, Anda dapat mempercantik tampilan aplikasi dengan menambahkan gaya di file `App.css`, membuat aplikasi lebih menarik bagi pengguna.

Setelah selesai menambahkan kode, simpan perubahan dan jalankan aplikasi Anda kembali. Sekarang, Anda sudah memiliki aplikasi React sederhana yang dapat menampilkan pesan dan menghitung jumlah klik pada tombol. Dari sini, Anda bisa bereksperimen lebih lanjut dengan menambahkan komponen baru, menggunakan React Router untuk navigasi, atau bahkan mengelola state global dengan Redux atau Context API.

2.2 BERURUSAN DENGAN JSX

Sebelum kita mulai membangun aplikasi, ada hal penting yang harus kita bahas terlebih dahulu. React tidak seperti kebanyakan pustaka JavaScript yang mungkin pernah Anda gunakan. React tidak terlalu menyenangkan ketika Anda hanya merujuk ke kode yang telah Anda tulis menggunakan tag skrip. React sangat istimewa dalam hal itu, dan ini berkaitan dengan cara aplikasi React dibangun. Seperti yang Anda ketahui, aplikasi web Anda (dan semua hal lain yang ditampilkan browser Anda) terdiri dari HTML, CSS, dan JavaScript:

Tidak masalah jika aplikasi web Anda ditulis menggunakan React atau pustaka lain seperti Angular, Knockout, atau jQuery. Hasil akhirnya harus berupa kombinasi HTML, CSS, dan JavaScript. Jika tidak, peramban Anda tidak akan tahu apa yang harus dilakukan. Nah, di sinilah sifat khusus React muncul. Selain HTML, CSS, dan JavaScript biasa, sebagian besar kode React Anda akan ditulis dalam sesuatu yang dikenal sebagai JSX. JSX, seperti yang saya sebutkan di Bab 1, adalah bahasa yang memungkinkan Anda mencampur JavaScript dan tag mirip HTML dengan mudah untuk menentukan elemen antarmuka pengguna (UI) dan fungsinya.

Kedengarannya keren dan sebagainya (dan kita akan melihat JSX beraksi sebentar lagi), tetapi ada sedikit masalah. Peramban Anda tidak tahu apa yang harus dilakukan dengan JSX. Untuk membangun aplikasi web menggunakan React, kita memerlukan cara untuk mengambil JSX kita dan mengubahnya menjadi JavaScript biasa yang dapat dipahami peramban Anda. Jika kita tidak melakukan ini, aplikasi React kita tidak akan berfungsi. Itu tidak keren. Untungnya, ada dua solusi untuk ini:

- Siapkan lingkungan pengembangan di sekitar Node dan beberapa alat bantu pembuatan. Di lingkungan ini, setiap kali Anda melakukan pembuatan, semua JSX Anda secara otomatis diubah menjadi JS dan ditempatkan di disk untuk Anda rujuk seperti berkas JavaScript biasa.
- Biarkan peramban Anda mengandalkan pustaka JavaScript untuk secara otomatis mengubah JSX menjadi sesuatu yang dipahaminya. Anda menentukan JSX secara langsung seperti halnya Anda menentukan bagian JavaScript lama, dan peramban Anda akan mengurus sisanya.

Kedua solusi ini memiliki tempat di dunia kita, tetapi mari kita bahas dampak masing-masing. Solusi pertama, meskipun sedikit rumit dan memakan waktu pada awalnya, adalah cara pengembangan web modern dilakukan saat ini. Selain mengompilasi (mentranspilasi agar lebih akurat) JSX Anda ke JS, pendekatan ini memungkinkan Anda memanfaatkan modul, alat pembuatan yang lebih baik, dan sejumlah fitur lain yang membuat pembuatan aplikasi web yang kompleks menjadi lebih mudah dikelola.

Solusi kedua menyediakan jalur cepat dan langsung di mana Anda awalnya menghabiskan lebih banyak waktu untuk menulis kode dan lebih sedikit waktu untuk mengutak-atik lingkungan pengembangan Anda. Untuk menggunakan solusi ini, yang perlu

Anda lakukan hanyalah merujuk ke berkas skrip. Berkas skrip ini menangani pengubahan JSX menjadi JS saat halaman dimuat, dan aplikasi React Anda akan aktif tanpa Anda harus melakukan sesuatu yang khusus pada lingkungan pengembangan Anda. Untuk tinjauan pendahuluan kita tentang React, kita akan menggunakan solusi kedua.

Anda mungkin bertanya-tanya mengapa kita tidak selalu menggunakan solusi kedua. Alasannya adalah bahwa peramban Anda mengalami penurunan kinerja setiap kali menghabiskan waktu menerjemahkan JSX ke JS. Itu sepenuhnya dapat diterima saat mempelajari cara menggunakan React, tetapi itu sama sekali tidak dapat diterima saat menerapkan aplikasi Anda untuk penggunaan di dunia nyata. Karena hal yang tidak dapat diterima itu, kita akan meninjau kembali semua ini dan melihat solusi pertama dan cara menyiapkan lingkungan pengembangan Anda nanti, setelah Anda merasa nyaman menggunakan React.

2.3 MEMULAI REACT

Pada bagian sebelumnya, kita melihat dua cara yang Anda miliki untuk memastikan aplikasi React Anda berakhir sebagai sesuatu yang dipahami oleh browser Anda. Pada bagian ini, kita akan mempraktikkan semua kata-kata itu. Pertama, kita akan memerlukan halaman HTML kosong yang akan bertindak sebagai titik awal kita. Jika Anda tidak memiliki halaman HTML kosong, silakan gunakan yang berikut ini:

```
<!DOCTYPE html>
<html>

<head>
  <title>React! React! React!</title>
</head>

<body>
  <script>

  </script>
</body>

</html>
```

Halaman ini tidak memiliki hal menarik atau mengasyikkan, tetapi mari kita perbaiki hal itu dengan menambahkan referensi ke pustaka React. Tepat di bawah judul, tambahkan dua baris ini:

```
<script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
<script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></script>
```

Kedua baris ini menyertakan pustaka inti React serta berbagai hal yang dibutuhkan React untuk bekerja dengan DOM. Tanpa keduanya, Anda tidak akan membangun aplikasi React sama sekali. Sekarang, kita belum selesai. Ada satu pustaka lagi yang perlu kita rujuk. Tepat di bawah kedua tag skrip ini, tambahkan baris berikut:

```
  <script src="https://cdnjs.cloudflare.com/ajax/l1bs/babel-core/5.8.23/
  browser.min.js"></script>
```

Yang kami lakukan di sini adalah menambahkan referensi ke kompiler Babel JavaScript (http://babeljs.io/). Babel melakukan banyak hal menarik, tetapi yang kami pedulikan adalah kemampuannya untuk mengubah JSX menjadi JavaScript. Pada titik ini, halaman HTML kita akan terlihat seperti berikut:

```
<!DOCTYPE html>
<html>

<head>
  <title>React! React! React !< /title>
  <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
  <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/
  browser.min.js"></script>
</head>

<body>
  <script>

  </scr1pt>
</body>

</html>
```

Jika Anda melihat pratinjau halaman Anda sekarang, Anda akan melihat bahwa halaman ini masih kosong tanpa ada yang terlihat. Tidak apa-apa. Kami akan memperbaikinya nanti.

Menampilkan Nama Anda

Hal pertama yang akan kita lakukan adalah menggunakan React untuk menampilkan nama kita di layar. Cara melakukannya adalah dengan menggunakan metode yang disebut render. Di dalam tag skrip Anda, tambahkan yang berikut ini:

```
    <h1>Sherlock Holmes</h1>,
    document.body
  );
```

Jangan khawatir jika semua ini tidak masuk akal pada tahap ini. Sasaran kita adalah menampilkan sesuatu di layar terlebih dahulu, dan kita akan memahami apa yang telah kita lakukan segera setelahnya. Sekarang, sebelum kita melihat pratinjau ini di halaman kita untuk melihat apa yang terjadi, kita perlu menetapkan blok skrip ini sebagai sesuatu yang dapat dikerjakan oleh Babel. Cara kita melakukannya adalah dengan menetapkan atribut type pada tag skrip ke nilai text/babel:

```
    <script type="text/babel">
  ReactDOM.render (
```

```
    <h1>Sherlock Holmes</h1>,
    document.body
  );
</script>
```

Setelah Anda membuat perubahan itu, sekarang pratinjau apa yang Anda miliki di peramban Anda. Yang akan Anda lihat adalah kata-kata Sherlock Holmes yang dicetak dengan huruf besar. Selamat! Anda baru saja membuat aplikasi menggunakan React. Sebagai aplikasi, ini tidak terlalu menarik. Kemungkinan nama Anda bahkan bukan Sherlock Holmes. Meskipun aplikasi ini tidak memiliki banyak kelebihan, aplikasi ini memperkenalkan Anda pada salah satu metode yang paling sering digunakan di dunia React metode ReactDOM.render. Metode render membutuhkan dua argumen:

- Elemen mirip HTML (alias JSX) yang ingin Anda hasilkan
- Lokasi di DOM tempat React akan merender JSX

Berikut adalah tampilan metode render kita:

```
ReactDOM.render (
  <h1>Sherlock Holmes</h1>,
  document.body
);
```

Argumen pertama kita adalah teks Sherlock Holmes yang dibungkus dalam beberapa tag h1. Sintaks mirip HTML di dalam JavaScript Anda inilah yang menjadi inti JSX. Meskipun kita akan menghabiskan lebih banyak waktu untuk mempelajari JSX nanti, saya harus menyebutkan ini di awal JSX sama gilanya dengan tampilannya. Setiap kali saya melihat tanda kurung dan garis miring di JavaScript, sebagian dari diri saya mati di dalam karena semua string escape dan tanda kutip yang tidak jelas yang harus saya lakukan. Dengan JSX, Anda tidak melakukan semua itu. Anda cukup meletakkan konten mirip HTML apa adanya seperti yang telah kita lakukan di sini. Secara ajaib (seperti yang super-hebat yang melibatkan naga dan sinar laser), semuanya berfungsi.

Argumen kedua adalah document.body. Tidak ada yang gila atau aneh tentang argumen ini. Argumen ini hanya menentukan di mana markup yang dikonversi dari JSX akan berakhir di DOM kita. Dalam contoh kita, ketika metode render berjalan, tag h1 (dan semua yang ada di dalamnya) ditempatkan di elemen body dokumen kita. Sekarang, tujuan dari latihan ini bukanlah untuk menampilkan nama di layar. Melainkan untuk menampilkan nama Anda. Silakan modifikasi kode Anda untuk melakukannya. Dalam kasus saya, metode render akan terlihat seperti berikut:

```
ReactDOM.render (
  <h1>Batman</h1>,
  document.body
):
```

Nah, akan terlihat seperti itu jika nama saya Batman! Bagaimanapun, jika Anda melihat

pratinjau halaman Anda sekarang, Anda akan melihat nama Anda ditampilkan, bukan Sherlock Holmes.

Semuanya Masih Akrab

Meskipun JavaScript terlihat baru dan berkilau berkat JSX, hasil akhir yang dilihat browser Anda adalah HTML, CSS, dan JavaScript yang bagus dan bersih. Untuk melihatnya sendiri, mari kita buat beberapa perubahan pada cara kerja dan tampilan aplikasi kita.

2.4 MENGUBAH TUJUAN

Hal pertama yang akan kita lakukan adalah mengubah tempat JSX kita mendapatkan output. Menggunakan JavaScript untuk menempatkan sesuatu secara langsung di elemen body Anda bukanlah ide yang bagus. Banyak hal yang bisa salah terutama jika Anda akan mencampur React dengan pustaka dan kerangka kerja JS lainnya. Jalur yang direkomendasikan adalah membuat elemen terpisah yang akan Anda perlakukan sebagai elemen root baru.

Elemen ini akan berfungsi sebagai tujuan yang akan digunakan metode render kita. Untuk mewujudkannya, kembali ke HTML dan tambahkan elemen div dengan nilai id container. Alih-alih memperlihatkan HTML lengkap untuk satu perubahan kecil ini, berikut ini tampilan elemen body kita:

```
<body>
  <div 1d="container"></div>
  <script type="text/babel">
    ReactDOM.render (
      <h1>Batman</h1>,
      document.body
    );
  </script>
</body>
```

Setelah elemen div kontainer kita didefinisikan dengan aman, mari kita ubah metode render untuk menggunakannya sebagai ganti document.body. Berikut ini salah satu cara untuk melakukannya:

```
ReactDOM.render (
  <h1>Batman</h1>,
  document.queryselector ("#container")
);
```

Cara lain untuk melakukan ini adalah dengan melakukan beberapa hal di luar metode render itu sendiri:

```
var destination = document.queryselector ("#container");

ReactDOM.render (
  <h1>Batman</h1>,
  destination
):
```

Perhatikan bahwa variabel tujuan menyimpan referensi ke elemen DOM kontainer kita. Di dalam metode render, kita cukup merujuk ke variabel tujuan yang sama alih-alih menulis sintaks pencarian elemen lengkap sebagai bagian dari argumen itu sendiri. Alasan saya ingin melakukan ini sederhana. Saya ingin menunjukkan kepada Anda bahwa Anda masih menulis JavaScript dan render hanyalah metode lama yang membosankan yang kebetulan membutuhkan dua argumen.

Saatnya untuk perubahan terakhir kita sebelum kita mengakhiri hari. Saat ini, nama kita muncul dalam gaya h1 default apa pun yang disediakan browser kita. Itu sangat buruk, jadi mari kita perbaiki dengan menambahkan beberapa CSS. Di dalam tag head Anda, tambahkan blok style dengan CSS berikut:

```
#container {
padding: 50px;
background-color: #EEE;
}

#container h1 {
font-size: 48px;
font-family: sans-serif;
  color: #0080A8;
  }
}
```

Setelah Anda menambahkan semua ini, pratinjau halaman Anda. Perhatikan bahwa teks kita muncul dengan tujuan yang sedikit lebih jelas daripada sebelumnya saat teks tersebut sepenuhnya bergantung pada gaya default browser (lihat Gambar 2.1).

Gambar 2.1 Hasil penambahan CSS

Alasan mengapa ini berhasil adalah karena badan DOM kita, setelah menjalankan semua kode React, berisi elemen kontainer kita dengan tag h1 di dalamnya. Tidak masalah bahwa tag h1 didefinisikan sepenuhnya di dalam JavaScript dalam sintaks JSX ini atau bahwa CSS Anda

didefinisikan dengan baik di luar metode render. Hasil akhirnya adalah bahwa aplikasi React Anda akan tetap terdiri dari HTML, CSS, dan JavaScript yang 100% organik dan bebas kurungan.

```
<!DOCTYPE html>
<html>

  <head>
  <title>React! React! React !< /title>
  <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
  <script src="https://unpkg.com/react-dom@15.3.2/dist/react-
  dom.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
  core/5.8.23/browser.min.js"></script>

    <style>
      #container {
        padding: 50px;
        background-color: #EEE;
      }
      #container h1 {
        font-size: 144px;
        font-family: sans-serif;
        color: #0080a8;
      }
    </style>
  </head>

      <body>
      <div id="container"></div>
      <script type="text/babel">
        var destination = document.querySelector ("#container") ;

    ReactDOM.render (React.createElement (
      "h1".
      null,
      "Batman"
      ), destination);
      </script>
      </body>

  </html>
```

2.5 KESIMPULAN

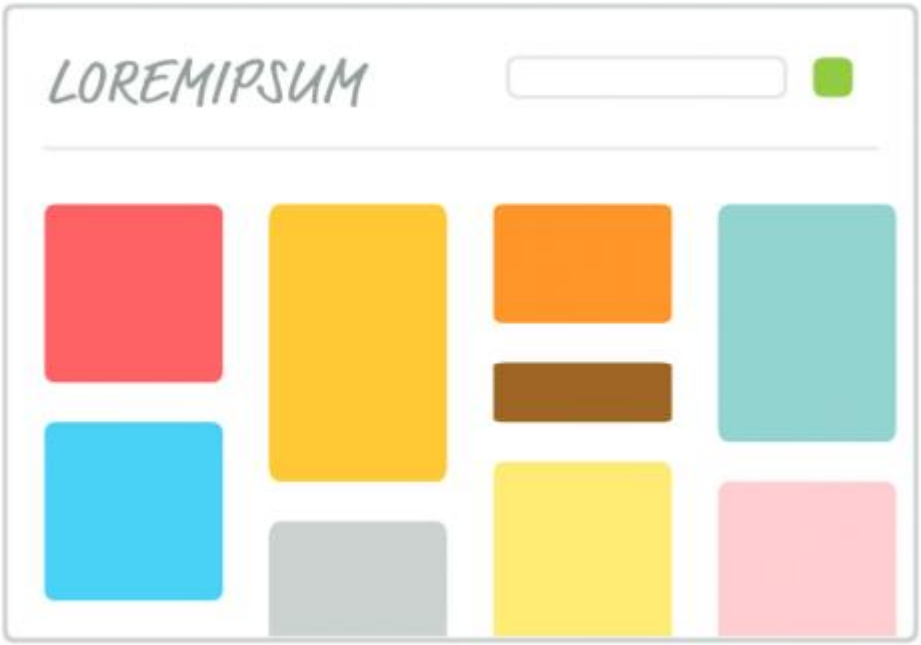
Jika ini pertama kalinya Anda membuat aplikasi React, kami telah membahas banyak hal di sini. Salah satu hal terpenting adalah React berbeda dari pustaka lain karena menggunakan bahasa baru yang disebut JSX untuk menentukan seperti apa tampilan visualnya. Kami mendapatkan sedikit gambaran tentang hal itu di sini saat kami mendefinisikan tag h1 di dalam metode render.

Dampak JSX melampaui cara Anda mendefinisikan elemen UI. Ia juga mengubah cara Anda membuat aplikasi secara keseluruhan. Karena peramban Anda tidak dapat memahami JSX dalam representasi aslinya, Anda perlu menggunakan langkah perantara untuk mengubah JSX tersebut menjadi JavaScript. Salah satu pendekatan adalah membuat aplikasi Anda untuk menghasilkan keluaran JavaScript yang ditranspilasi agar sesuai dengan sumber JSX.

Pendekatan lain (alias yang kami gunakan di sini) adalah menggunakan pustaka Babel untuk menerjemahkan JSX menjadi JavaScript di peramban itu sendiri. Meskipun penurunan kinerja karena melakukan hal itu tidak direkomendasikan untuk aplikasi langsung/produksi, saat membiasakan diri dengan React, Anda tidak dapat mengalahkan kemudahannya. Pada bab-bab selanjutnya, kita akan menghabiskan waktu untuk menyelami lebih dalam JSX dan membahas lebih jauh tentang metode render sembari kita melihat semua hal penting yang membuat React berfungsi.

BAB 3
KOMPONEN DALAM REACT

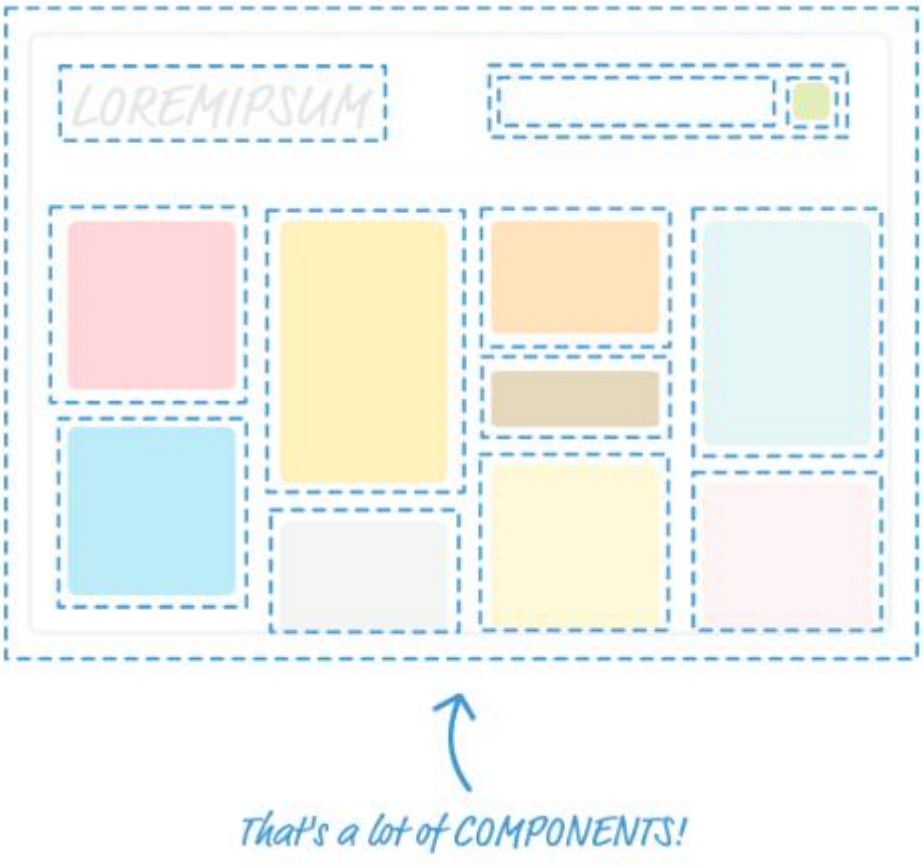
Komponen adalah salah satu hal yang membuat React menjadi React! Komponen adalah salah satu cara utama untuk mendefinisikan visual dan interaksi yang membentuk apa yang dilihat orang saat mereka menggunakan aplikasi Anda. Misalnya, Gambar 3.1 menunjukkan tampilan aplikasi Anda yang sudah jadi.



Gambar 3.1 Aplikasi Hipotetis Anda Yang Sudah Jadi

Ini adalah sosis yang sudah jadi. Selama pengembangan, dilihat dari sudut pandang Anda sebagai pengembang React, hal-hal mungkin terlihat sedikit kurang menarik. Hampir setiap bagian dari visual aplikasi ini akan dibungkus dalam modul mandiri yang dikenal sebagai komponen. Untuk menyoroti apa yang dimaksud dengan "hampir setiap" di sini, lihat diagram pada Gambar 3.2.

Setiap garis putus-putus mewakili komponen individual yang bertanggung jawab atas apa yang Anda lihat serta interaksi apa pun yang mungkin menjadi tanggung jawabnya. Jangan biarkan hal ini membuat Anda takut. Meskipun ini terlihat sangat rumit, seperti yang akan Anda lihat sebentar lagi, ini akan mulai masuk akal setelah Anda memiliki kesempatan untuk bermain dengan komponen dan beberapa hal luar biasa yang mereka lakukan atau setidaknya berusaha keras untuk melakukannya.



Gambar 3.2 Representasi Diagramatik Dari Komponen Aplikasi

3.1 TINJAUAN SINGKAT FUNGSI

Dalam JavaScript, Anda memiliki hal-hal yang dikenal sebagai fungsi. Fungsi memungkinkan Anda membuat kode Anda sedikit lebih bersih dan lebih dapat digunakan kembali. Nah, ada alasan mengapa kami meluangkan waktu untuk melihat fungsi, dan itu bukan untuk mengganggu Anda saya bersumpah! Fungsi, secara konseptual, berbagi banyak area permukaan dengan komponen React, dan cara termudah untuk memahami apa yang dilakukan komponen adalah dengan melihat sekilas fungsi terlebih dahulu. Di dunia yang mengerikan di mana fungsi tidak ada, Anda mungkin memiliki beberapa kode yang terlihat seperti berikut:

```
var speed = 10;
var time = 5;
alert (speed * time) ;

var speed1 = 85;
var timel = 1.5;
alert (speed1 * timel);

var speed2 = 12;
var time2 = 9;
alert (speed2 * time2);

var speed3 = 42;
```

```
var time3 = 21;
alert (speed3 * time3);
```

Dalam dunia yang sangat santai yang melibatkan fungsi, Anda dapat meringkas semua teks duplikat menjadi sesuatu yang sederhana seperti berikut:

```
   function getDistance (speed, time) {
   var result = speed * time;
   alert(result);
}
```

Fungsi getDistance kami menghapus semua kode duplikat yang Anda lihat sebelumnya, dan menggunakan kecepatan dan waktu sebagai argumen untuk memungkinkan Anda menyesuaikan kalkulasi yang dikembalikan. Untuk memanggil fungsi ini, yang harus Anda lakukan adalah:

```
getDistance (10, 5);
getDistance(85, 1.5);
getDistance(12, 9);
getDistance(42, 21);
```

Bukankah ini terlihat lebih bagus? Sekarang ada fungsi bernilai hebat lainnya yang disediakan. Fungsi Anda (seperti peringatan di dalam getDistance) dapat memanggil fungsi lain sebagai bagian dari fungsinya. Berikut ini adalah kita menggunakan fungsi formatDistance untuk mengubah apa yang dikembalikan oleh getDistance:

```
function formatDistance (distance) {
   return distance + "km";
}

function getDistance (speed, time) {
   var result = speed * time;
   alert(formatDistance(result));
}
```

Kemampuan untuk membuat fungsi memanggil fungsi lain memungkinkan kita untuk memisahkan dengan jelas apa yang dilakukan fungsi. Anda tidak perlu memiliki satu fungsi monolitik yang melakukan segalanya. Anda dapat mendistribusikan fungsionalitas ke banyak fungsi yang dikhususkan untuk jenis tugas tertentu. Yang terbaik dari semuanya, setelah Anda membuat perubahan pada cara kerja fungsi, Anda tidak perlu melakukan apa pun untuk melihat hasil dari perubahan tersebut.

Jika tanda tangan fungsi tidak berubah, panggilan apa pun yang ada ke fungsi tersebut akan bekerja secara ajaib dan secara otomatis mengambil setiap perubahan baru yang Anda buat pada fungsi itu sendiri. Misalnya, panggilan getDistance yang ada akan melihat hasil dari

fungsi formatDistance meskipun fungsi formatDistance tidak ada saat panggilan pertama kali didefinisikan. Itu sangat mengagumkan. Singkatnya, fungsi itu mengagumkan. Saya tahu itu. Anda tahu itu. Itulah sebabnya semua kode yang kita tulis memilikinya di mana-mana.

3.2 MENGUBAH CARA MENANGANI UI

Saya rasa tidak ada yang akan tidak setuju dengan hal-hal baik yang dibawa oleh fungsi. Fungsi benar-benar memungkinkan untuk menyusun kode untuk aplikasi Anda dengan cara yang masuk akal. Tingkat kehati-hatian yang sama yang kita gunakan dalam menulis kode tidak selalu memungkinkan saat menulis UI.

Karena berbagai alasan teknis dan non-teknis, kita selalu menoleransi tingkat kecerobohan tertentu dalam cara kita bekerja dengan elemen UI. Saya sadar bahwa itu adalah pernyataan yang cukup kontroversial, jadi izinkan saya menyoroti apa yang saya maksud dengan melihat beberapa contoh. Kita akan kembali dan melihat metode render yang kita gunakan di bab sebelumnya:

```
var destination = document.querySelector ("#container") ;

ReactDOM.render (
  <h1>Batman</h1>,
  destination
);
```

Yang Anda lihat di layar adalah kata Batman yang dicetak dengan huruf besar berkat elemen h1. Mari kita ubah sedikit dan katakan bahwa kita ingin mencetak nama beberapa pahlawan super lainnya. Untuk melakukannya, mari kita ubah metode render kita menjadi seperti berikut:

```
var destination = document.querySelector ("#container") ;

ReactDOM.render (
  <div>
    <h1>Batman</h1>
    <h1>Iron Man</h1>
    <h1>Nicolas Cage</h1>
    <h1>Mega Man</h1>
  </div>,
  Destination
);
```

Perhatikan apa yang Anda lihat di sini. Kami memancarkan div yang berisi empat elemen h1 dengan nama pahlawan super kami.

JSX Gotcha: Mengeluarkan Beberapa Elemen

Ada detail JSX penting yang perlu diperhatikan di sini. Div yang membungkus elemen h1 kita tidak ada di sana karena tampaknya itu ide yang bagus. Div ada di sana karena memang

harus ada di sana. Di React, Anda tidak dapat mengeluarkan beberapa elemen yang berdekatan seperti yang ditunjukkan berikut ini:

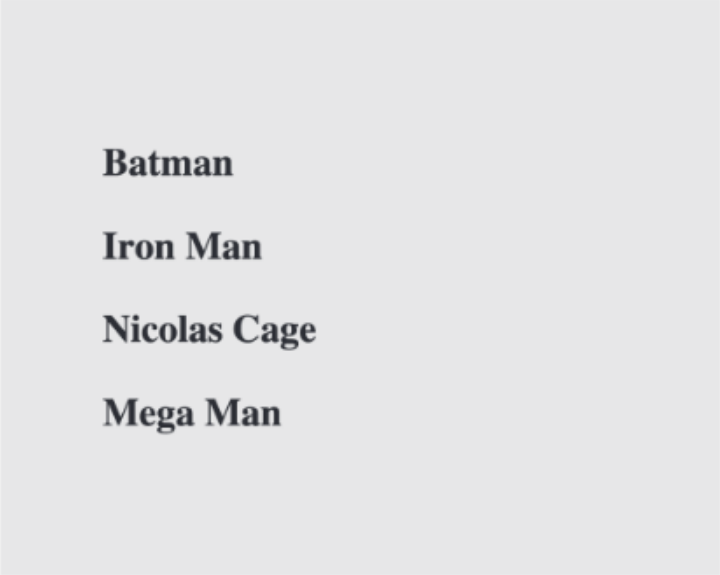
```
var destination = document.querySelector ("#container") ;

ReactDOM.render (
  <h1>Batman</h1>
  <h1>Iron Man</h1>
  <h1>Nicolas Cage</h1>
  <h1>Mega Man</h1>,
  destination
);
```

Meskipun ini adalah HTML yang valid, namun tidak valid di mata aliansi jahat antara JSX dan JavaScript. Itu mungkin terdengar seperti batasan yang mengerikan, tetapi solusinya sangat mudah. Meskipun Anda hanya dapat menampilkan satu elemen, elemen ini dapat memiliki anak sebanyak yang dibutuhkan. Itulah sebabnya kami membungkus elemen h1 kami di dalam div. Kami melakukan ini karena cara JSX diubah menjadi JavaScript.

Rinciannya adalah sesuatu yang akan kita lihat nanti, tetapi saat ini tidak cukup penting untuk mengalihkan perhatian kita dari mempelajari komponen. Oke, jadi yang kita miliki sekarang adalah empat elemen h1 yang masing-masing berisi nama seorang pahlawan super. Bagaimana jika kita ingin mengubah elemen h1 kita menjadi sesuatu seperti h3? Kita dapat memperbarui semua elemen ini secara manual sebagai berikut:

```
var destination = document.querySelector ("#container");

ReactDOM.render (
  <div>
    <h3>Batman</h3>
    <h3>Iron Man</h3>
    <h3>Nicolas Cage</h3>
    <h3>Mega Man</h3>
  </div>,
  destination
);
```

Jika Anda melihat pratinjau apa yang kami miliki, Anda akan melihat sesuatu yang terlihat agak tidak bergaya dan polos (lihat Gambar 3.3).



Gambar 3.3 Nama-Nama Pahlawan Super Biasa

Kita tidak ingin berlebihan dengan gaya di sini. Yang ingin kita lakukan hanyalah membuat miring semua nama ini dengan menggunakan tag <i>, jadi mari kita perbarui secara manual apa yang kita render dengan membuat perubahan ini:

```
var destination = document.querySelector ("#container") ;

ReactDOM.render (
  <div>
    <h3><i>Batman</i></h3>
    <h3><i>Iron Man</i></h3>
    <h3><i>Nicolas Cage</i></h3>
    <h3><i>Mega Man</i></h3>
  </div>,
  destination
) ;
```

Kami menelusuri setiap elemen h3 dan membungkus konten di dalam beberapa tag i. Dapatkah Anda mulai melihat masalahnya di sini? Apa yang kami lakukan dengan UI kami tidak berbeda dengan memiliki kode yang terlihat seperti berikut:

```
var speed = 10;
var time = 5;
alert (speed * time) ;

var speed1 = 85;
var timel = 1.5;
alert (speed1 * timel);

var speed2 = 12;
var time2 = 9;
```

```
alert (speed2 * time2);

var speed3 = 42;
var time3 = 21;
alert (speed3 * time3);
```

Setiap perubahan yang ingin kita buat pada elemen h1 atau h3 kita perlu diduplikasi untuk setiap contohnya. Bagaimana jika kita ingin melakukan sesuatu yang lebih rumit daripada sekadar mengubah tampilan elemen kita? Bagaimana jika kita ingin merepresentasikan sesuatu yang lebih rumit daripada contoh sederhana yang kita gunakan sejauh ini? Apa yang kita lakukan saat ini tidak akan dapat diskalakan karena memperbarui setiap salinan dari apa yang ingin kita ubah secara manual akan memakan waktu. Itu juga membosankan.

Sekarang, inilah pemikiran yang gila: Bagaimana jika semua hal menakjubkan yang kita lihat tentang fungsi entah bagaimana dapat diterapkan pada cara kita mendefinisikan visual aplikasi kita? Bukankah itu akan menyelesaikan semua inefisiensi yang telah kita soroti di bagian ini? Nah, ternyata, jawaban untuk pertanyaan "Bagaimana jika" itu merupakan inti dari apa itu React. Sekarang saatnya bagi Anda untuk mengenal komponen tersebut.

3.3 MENGENAL KOMPONEN REACT

Solusi untuk semua masalah kita (bahkan masalah eksistensial yang kita hadapi!) dapat ditemukan di komponen React. Komponen React adalah potongan JavaScript yang dapat digunakan kembali yang menghasilkan (melalui JSX) elemen HTML. Kedengarannya biasa saja untuk sesuatu yang mampu memecahkan masalah besar dan melakukan hal-hal hebat, tetapi saat kita mulai membangun komponen dan secara bertahap meningkatkan kompleksitasnya, Anda akan melihat bahwa komponen benar-benar hebat dan sama hebatnya seperti yang saya gambarkan kepada Anda. Mari kita mulai dengan membangun beberapa komponen bersama-sama. Untuk mengikutinya, mulailah dengan dokumen React kosong:

```
<!DOCTYPE html>
<html>

<head>
  <title>React Components</title>
  <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
  <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/
  browser.min.js"></script>
  </head>

  <body>
  <div id="container"></div>
  <script type="text/babel">

  </script>
  </body>
</html>
```

Tidak ada yang menarik di halaman ini. Hampir identik dengan apa yang kita bahas di bab sebelumnya, halaman ini cukup sederhana, hanya dengan referensi ke pustaka React dan Babel dan elemen div yang dengan bangga menampilkan nilai id container.

Membuat Komponen Hello, World!

Kita akan mulai dengan sangat sederhana. Yang ingin kita lakukan adalah menggunakan komponen untuk membantu kita mencetak teks terkenal "Hello, world!" ke layar. Seperti yang telah kita ketahui, dengan hanya menggunakan metode render ReactDOM, kodenya akan terlihat seperti berikut:

```
ReactDOM.render (
   <div>
      <p>Hello, world!</p>
   </div>,
   document.querySelector("#container")
);
```

Mari kita buat ulang semua ini dengan menggunakan komponen. Anda memiliki beberapa cara untuk membuat komponen di React, tetapi cara pertama yang akan kita lakukan adalah dengan menggunakan React.createClass. Lanjutkan dan tambahkan kode yang disorot berikut tepat di atas metode render yang sudah ada:

```
var HelloWorld = React.createClass ({
});
ReactDOM.render (
   <div>
      <p>Hello, world!</p>
   </div>,
   document.querySelector ("#container")
);
```

Yang telah kita lakukan adalah membuat komponen baru yang disebut HelloWorld. Komponen HelloWorld ini tidak melakukan apa pun saat ini. Bahkan, pada dasarnya komponen ini adalah objek JavaScript kosong saat ini. Di dalam objek ini, Anda dapat meletakkan berbagai macam properti untuk lebih jauh mendefinisikan apa yang dilakukan HelloWorld.

Beberapa properti yang Anda definisikan bersifat khusus dan digunakan oleh React untuk membantu komponen Anda menjalankan fungsinya. Salah satu properti wajib tersebut adalah render. Lanjutkan dan modifikasi komponen HelloWorld kita dengan menambahkan properti render seperti yang ditunjukkan pada berikut ini:

```
var HelloWorld = React.createClass ({
   render: function() {

   }
});
```

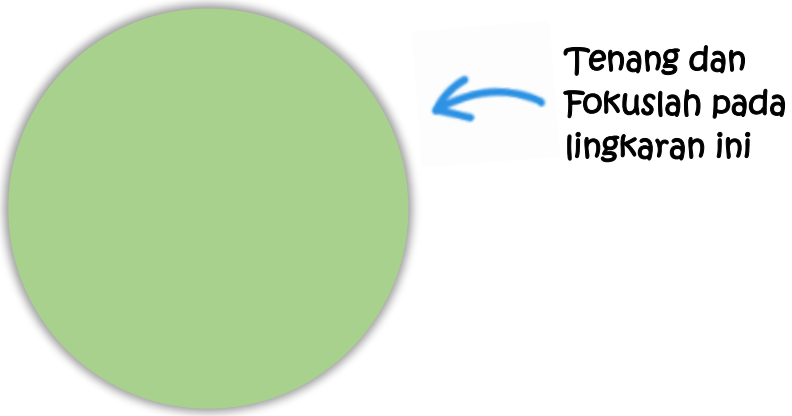
Sama seperti metode render yang kita lihat beberapa saat sebelumnya sebagai bagian dari ReactDOM.render, metode render di dalam komponen juga bertanggung jawab untuk menangani JSX. Mari kita ubah metode render kita untuk mengembalikan Hello, componentized world!, jadi lanjutkan dan tambahkan baris yang disorot berikut:

```
var HelloWorld = React.createClass ({
  render: function() {

  return (
  <p>Hello, componentized world!< /p>
  );
  }
});
```

Yang telah kita lakukan adalah memberi tahu metode render kita untuk mengembalikan JSX yang mewakili teks Halo, dunia yang terkomponenkan! Yang tersisa hanyalah menggunakan komponen ini. Cara Anda menggunakan komponen setelah Anda mendefinisikannya adalah dengan memanggilnya, dan kita akan memanggilnya dari teman lama kita, metode ReactDOM.render:

```
ReactDOM.render (
  <HelloWorld/>,
  document.querySelector ("#container")
);
```

Itu bukan salah ketik! JSX yang kita gunakan untuk memanggil komponen HelloWorld adalah <HelloWorld/> yang sangat mirip HTML. Jika Anda melihat pratinjau halaman di browser, Anda akan melihat teks Halo, dunia yang terkomponen! muncul di layar. Jika Anda menahan napas karena tegang, Anda dapat bersantai. Jika Anda kesulitan bersantai setelah melihat sintaksis yang kita gunakan untuk memanggil HelloWorld, tatap lingkaran berikut selama beberapa saat:



Gambar 3.4 hello world

Oke, kembali ke kenyataan. Apa yang telah kita lakukan sejauh ini mungkin tampak gila, tetapi anggap saja komponen <HelloWorld/> Anda sebagai tag HTML baru dan keren yang fungsinya dapat Anda kendalikan sepenuhnya. Ini berarti Anda dapat melakukan berbagai hal yang berkaitan dengan HTML. Misalnya, lanjutkan dan modifikasi metode ReactDOM.render kita agar terlihat seperti berikut:

```
ReactDOM.render (
   <div>
      <HelloWorld/>
   </div>,
   document.querySelector ("#container")
);
```

Kami membungkus panggilan kami ke komponen HelloWorld di dalam elemen div, dan jika Anda melihat pratinjau ini di browser, semuanya masih berfungsi. Mari melangkah lebih jauh! Daripada hanya melakukan satu panggilan ke HelloWorld, mari lakukan beberapa panggilan. Ubah metode ReactDOM.render kami menjadi seperti berikut:

```
ReactDOM.render (
   <div>
      <HelloWorld/>
      <HelloWorld/>
      <HelloWorld/>
      <HelloWorld/>
      <HelloWorld/>
      <HelloWorld/>
   </div>,
   document.querySelector ("#container")
);
```

Yang akan Anda lihat sekarang adalah sekumpulan teks Hello, yang terkomponenkan di dunia! muncul. Mari kita lakukan satu hal lagi sebelum beralih ke sesuatu yang lebih menarik. Kembali ke deklarasi komponen HelloWorld kita, dan ubah teks yang kita kembalikan ke nilai Hello, world! yang lebih tradisional:

```
var HelloWorld = React.createClass({
   render: function() {
      return (
         <p>Hello, world!</p>
   );
   }
});
```

Cukup buat satu perubahan ini dan pratinjau contoh Anda. Kali ini, semua berbagai panggilan HelloWorld yang kita tentukan sebelumnya kini mengembalikan Hello, world! ke layar. Tidak ada modifikasi manual untuk setiap panggilan HelloWorld. Itu hal yang bagus!

3.4 MENETAPKAN PROPERTI

Saat ini, komponen kita hanya melakukan satu hal. Komponen itu mencetak Hello, world! ke layar kita dan hanya itu! Itu setara dengan memiliki fungsi JavaScript yang terlihat seperti ini:

```
function getDistance () {
   alert("42km");
}
```

Kecuali untuk satu kasus yang sangat khusus, fungsi JavaScript tersebut tampaknya tidak terlalu berguna, bukan? Cara untuk meningkatkan kegunaan fungsi ini adalah dengan memodifikasinya untuk mengambil argumen:

```
function getDistance (speed, time) {
   var result = speed * time;
   alert(result);
}
```

Sekarang, fungsi Anda dapat digunakan secara lebih umum untuk berbagai situasi—bukan hanya situasi yang outputnya akan mencapai 42 km. Hal serupa juga berlaku untuk komponen Anda. Sama seperti fungsi, Anda dapat meneruskan argumen yang mengubah apa yang dilakukan komponen Anda. Ada sedikit pembaruan terminologi yang perlu Anda pahami. Apa yang kita sebut argumen dalam dunia fungsi akan dikenal sebagai properti dalam dunia komponen.

Mari kita lihat properti ini beraksi! Kita akan memodifikasi komponen HelloWorld kita untuk memungkinkan Anda menentukan siapa atau apa yang Anda sapa selain World yang umum. Misalnya, bayangkan dapat menentukan Bono sebagai bagian dari panggilan HelloWorld dan melihat Hello, Bono! muncul di layar. Untuk menambahkan properti ke komponen, ada dua bagian yang perlu Anda ikuti.

Bagian Pertama: Memperbarui Definisi Komponen

Saat ini, komponen HelloWorld kita dikodekan secara keras untuk selalu mengirimkan Hello, world! sebagai bagian dari nilai pengembaliannya. Hal pertama yang akan kita lakukan adalah mengubah perilaku itu dengan membuat return mencetak nilai yang diteruskan oleh properti. Kita perlu memberi nama untuk properti kita, dan untuk contoh ini, kita akan menyebut properti kita greetTarget. Untuk menentukan nilai greetTarget sebagai bagian dari komponen kita, berikut modifikasi yang perlu kita buat:

```
var HelloWorld = React.createClass ({
   render: function() {
      return (
         <p>Hello, {this.props.greetTarget}!</p>
      );
   }
});
```

Cara mengakses properti adalah dengan memanggilnya melalui properti props yang dapat diakses oleh setiap komponen. Perhatikan cara kami menentukan properti ini. Kami menempatkannya di dalam tanda kurung kurawal {dan }. Di JSX, jika Anda ingin sesuatu dievaluasi sebagai ekspresi, Anda perlu membungkusnya di dalam tanda kurung kurawal. Jika Anda tidak melakukannya, Anda akan melihat teks mentah this.props.greetTarget tercetak.

Bagian Kedua: Memodifikasi Panggilan Komponen

Setelah Anda memperbarui definisi komponen, yang tersisa hanyalah meneruskan nilai properti sebagai bagian dari panggilan komponen. Hal itu dilakukan dengan menambahkan atribut dengan nama yang sama dengan properti kita, diikuti dengan nilai yang ingin Anda teruskan. Dalam contoh kita, hal itu akan melibatkan modifikasi panggilan HelloWorld dengan atribut greetTarget dan nilai yang ingin kita berikan. Lanjutkan dan modifikasi panggilan HelloWorld kita sebagai berikut:

```
ReactDOM.render (
  <div>
    <HelloWorld greetTarget="Batman"/>
    <HelloWorld greetTarget="Iron Man"/>
    <HelloWorld greetTarget="Nicolas Cage"/>
    <HelloWorld greetTarget="Mega Man"/>
    <HelloWorld greetTarget="Bono"/>
    <HelloWorld greetTarget="Catwoman"/>
  </div>,
  document.querySelector ("#container")
};
```

Setiap panggilan HelloWorld kita sekarang memiliki atribut greetTarget beserta nama pahlawan super (atau makhluk mitos yang setara!) yang ingin kita sapa. Jika Anda melihat contoh ini di browser, Anda akan melihat ucapan selamat dicetak dengan gembira di layar. Satu hal terakhir yang perlu diingat sebelum kita melanjutkan.

Anda tidak terbatas hanya memiliki satu properti pada suatu komponen. Anda dapat memiliki properti sebanyak yang Anda inginkan, dan properti props Anda akan dengan mudah mengakomodasi permintaan properti apa pun yang Anda miliki tanpa membuat keributan.

Berurusan dengan Anak-Anak

Beberapa bagian yang lalu, saya menyebutkan bahwa komponen kita (di JSX) sangat mirip dengan elemen HTML biasa. Kita melihatnya sendiri ketika kita membungkus komponen di dalam elemen div atau menetapkan atribut dan nilai sebagai bagian dari penetapan properti. Ada satu hal lagi yang dapat Anda lakukan dengan komponen seperti yang dapat Anda lakukan dengan banyak elemen HTML. Komponen Anda dapat memiliki anak. Artinya, Anda dapat melakukan sesuatu seperti ini:

```
<CleverComponent foo="bar">
  <p>Something!</p>
</CleverComponent>
```

Anda memiliki komponen yang disebut CleverComponent dengan sangat cerdik, dan komponen tersebut memiliki elemen p sebagai anak. Dari dalam CleverComponent, Anda memiliki kemampuan untuk mengakses elemen anak p (dan semua anak yang mungkin dimilikinya) melalui properti children yang diakses oleh this.props.children.

Untuk memahami semua ini, mari kita coba contoh lain yang sangat sederhana. Kali ini, kita memiliki komponen yang disebut Buttonify yang membungkus anak-anaknya di dalam sebuah tombol. Komponen tersebut tampak seperti ini:

```
var Buttonify = React.createClass({
  render: function() {
    return (
      <div>
        <button type={this.props.behavior}>{this.props.children}</button>
      </div>
    );
  }
});
```

Cara Anda menggunakan komponen ini adalah dengan memanggilnya melalui metode ReactDOM.render seperti yang ditunjukkan di sini:

```
ReactDOM.render(
  <div>
    <Buttonify behavior="Submit">SEND DATA</Buttonify>
  </div>,
  document.querySelector("#container")
);
```

Ketika kode ini dijalankan, berdasarkan tampilan JSX dalam metode render komponen Buttonify, yang akan Anda lihat adalah kata-kata “SEND DATA” yang dibungkus dalam elemen tombol. Dengan gaya yang tepat, hasilnya bisa terlihat sangat besar seperti pada Gambar 3.4.

Gambar 3.5 Tombol Kirim Data Yang Besar

Bagaimanapun, kembali ke JSX, perhatikan bahwa kita menetapkan properti kustom yang disebut perilaku. Properti ini memungkinkan kita untuk menetapkan atribut tipe elemen tombol, dan Anda dapat melihat kita mengaksesnya melalui this.props.behavior dalam metode render definisi komponen. Ada lebih banyak hal untuk mengakses anak komponen

daripada yang telah kita lihat di sini. Misalnya, jika elemen anak Anda adalah akar dari struktur yang sangat bertingkat, properti this.props.children akan mengembalikan sebuah array.

Jika elemen anak Anda hanya satu elemen (seperti dalam contoh kita), properti this.props.children mengembalikan satu komponen yang TIDAK dibungkus dalam array. Ada beberapa hal lagi yang perlu disebutkan, tetapi alih-alih menyebutkan semua berbagai kasus dan membuat Anda bosan, kita akan secara alami menyentuh kasus-kasus tersebut sebagai bagian dari melihat contoh yang lebih rumit nanti.

3.5 KESIMPULAN

Jika Anda ingin membuat aplikasi menggunakan React, Anda tidak dapat menjelajah terlalu jauh tanpa harus menggunakan komponen. Mencoba membuat aplikasi React tanpa menggunakan komponen itu seperti membuat aplikasi berbasis JavaScript tanpa menggunakan fungsi.

BAB 4
PENATAAN GAYA DI REACT

Selama beberapa generasi, manusia (dan mungkin lumba-lumba yang sangat pintar) telah menata gaya konten HTML mereka menggunakan CSS (alias Cascading Style Sheets). Semuanya berjalan baik. Dengan CSS, Anda memiliki pemisahan yang baik antara konten dan presentasi. Sintaks pemilih memberi Anda banyak fleksibilitas dalam memilih elemen mana yang akan ditata gayanya dan mana yang akan dilewati. Anda bahkan tidak dapat menemukan terlalu banyak masalah yang tidak Anda sukai dari keseluruhan hal cascading yang menjadi inti CSS.

Baiklah, jangan beri tahu React tentang hal itu. Meskipun React tidak secara aktif membenci CSS, ia memiliki pandangan yang berbeda dalam hal menata gaya konten. Seperti yang telah kita lihat sejauh ini, salah satu ide inti React adalah agar bagian visual aplikasi kita menjadi mandiri dan dapat digunakan kembali. Itulah sebabnya elemen HTML dan JavaScript yang memengaruhinya berada dalam wadah yang sama yang kita sebut komponen. Kita sudah melihatnya di bab sebelumnya.

Bagaimana dengan tampilan elemen HTML (alias penataan gayanya)? Di mana seharusnya itu? Anda mungkin dapat menebak ke mana saya akan mengarah dengan ini. Anda tidak dapat memiliki bagian UI yang berdiri sendiri ketika gayanya didefinisikan di tempat lain. Itulah sebabnya React mendorong Anda untuk menentukan tampilan elemen Anda di samping HTML dan JavaScript. Dalam tutorial ini, Anda akan mempelajari semua tentang pendekatan misterius (dan mungkin memalukan!) ini untuk menata konten Anda. Tentu saja, kita juga akan membahas cara menggunakan CSS. Ada ruang untuk kedua pendekatan tersebut meskipun React mungkin tidak menganggapnya demikian

4.1 MENAMPILKAN BEBERAPA VOKAL

Untuk mempelajari cara menata konten React kita, mari kita bekerja sama dalam contoh (yang benar-benar manis dan menarik!) yang hanya menampilkan vokal di halaman. Pertama, Anda memerlukan halaman HTML kosong yang akan menampung konten React kita. Jika Anda tidak memilikinya, silakan gunakan markup berikut:

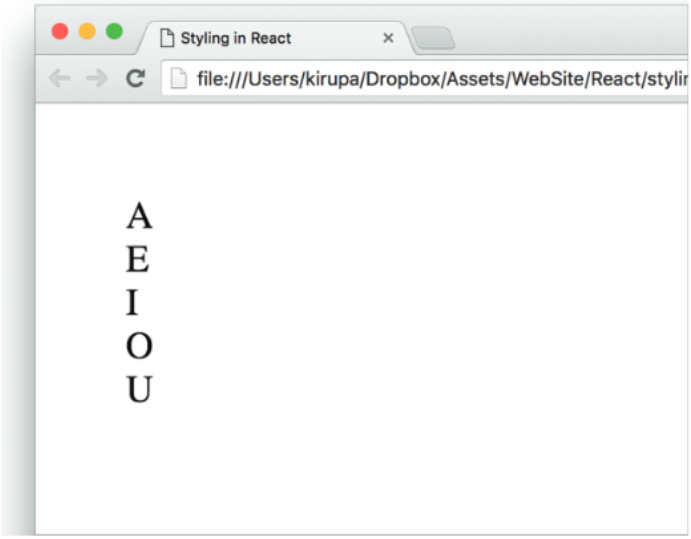
```
<!DOCTYPE html>
<html>
<head>
  <title>Styling in React</title>
  <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
  <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.23/browser.min
.js"></script>
  <style>
    #container {
```

```
      padding: 50px;
      background-color: #FFF;
    }
  </style>
</head>
<body>
  <div id="container"></div>
</body>
</html>
```

Semua markup ini hanya memuat pustaka React dan Babel kita dan menentukan div dengan nilai id container. Untuk menampilkan vokal, kita akan menambahkan beberapa kode khusus React. Tepat di bawah elemen div container, tambahkan yang berikut:

```
<script type="text/babel">

  var Letter = React.createClass({
    render: function() {
      return (
        <div>
          {this.props.children}
        </div>
      );
    }
  });

var destination = document.querySelector("#container");

ReactDOM.render(
  <div>
    <Letter>A</Letter>
    <Letter>E</Letter>
    <Letter>I</Letter>
    <Letter>O</Letter>
    <Letter>U</Letter>
  </div>,
  destination
);
</script>
```

Dari apa yang kita pelajari tentang komponen, tidak ada yang misterius di sini. Kita membuat komponen bernama Letter yang bertanggung jawab untuk membungkus vokal kita di dalam elemen div. Semua ini ditambatkan dalam HTML kita melalui tag skrip yang tipenya menunjuknya sebagai sesuatu yang Babel akan tahu apa yang harus dilakukan dengannya. Jika Anda melihat pratinjau halaman Anda, Anda akan melihat sesuatu yang membosankan yang tampak seperti Gambar 4.1.



Gambar 4.1 Hasil Akhir Yang Membosankan Dari Apa Yang Anda Lihat

Jangan khawatir, kita akan membuatnya terlihat sedikit tidak membosankan dalam beberapa saat. Setelah kita mencoba huruf-huruf ini, Anda akan melihat sesuatu yang lebih mirip dengan Gambar 4.2.

Gambar 4.2 Huruf-Huruf Disusun Secara Horizontal Dan Dengan Latar Belakang Kuning

Huruf vokal kita akan dibungkus dengan latar belakang kuning, disejajarkan secara horizontal, dan menggunakan fon monospace yang mewah. Mari kita lihat cara melakukan semua ini baik dalam CSS maupun pendekatan baru React.

4.2 MENATA KONTEN REACT MENGGUNAKAN CSS

Menggunakan CSS untuk menata konten React kita sebenarnya semudah yang Anda bayangkan. Karena React pada akhirnya mengeluarkan tag HTML biasa, semua trik CSS yang telah Anda pelajari selama bertahun-tahun untuk menata HTML masih berlaku. Hanya ada beberapa hal kecil yang perlu diingat.

Memahami HTML yang Dihasilkan

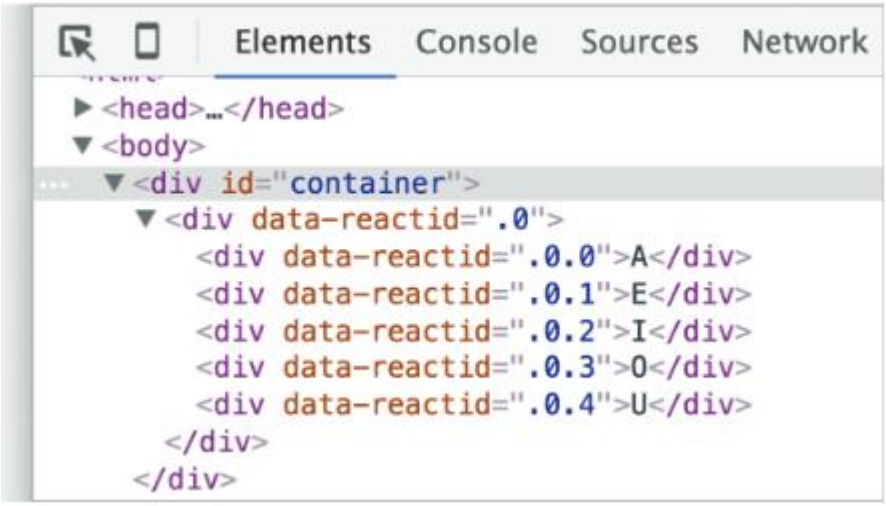
Sebelum Anda dapat menggunakan CSS, Anda harus terlebih dahulu merasakan seperti apa HTML yang dihasilkan React. Anda dapat dengan mudah mengetahuinya dengan melihat JSX yang didefinisikan di dalam metode render. Metode render induk adalah yang berbasis ReactDOM, dan terlihat seperti berikut:

```
<div>
  <Letter>A</Letter>
  <Letter>E</Letter>
  <Letter>I</Letter>
  <Letter>O</Letter>
  <Letter>U</Letter>
</div>
```

Kita telah membungkus berbagai komponen Letter di dalam div. Tidak ada yang terlalu menarik di sini. Metode render di dalam komponen Letter juga tidak jauh berbeda:

```
<div>
  {this.props.children}
</div>
```

Seperti yang dapat Anda lihat, setiap vokal dibungkus dalam kumpulan tag div-nya sendiri. Jika Anda harus memainkan semuanya (misalnya, melihat pratinjau contoh kita di browser), struktur DOM akhir untuk vokal kita tampak seperti Gambar 4.3. Abaikan atribut data-reacroot (yang mungkin tidak Anda lihat tergantung pada versi React Anda!) pada wadah div, tetapi perhatikan hal-hal lain yang Anda lihat. Yang kita miliki hanyalah perluasan HTML dari berbagai fragmen JSX yang kita lihat dalam metode render beberapa saat yang lalu dengan vokal kita yang bersarang di dalam sekumpulan elemen div.



Gambar 4.3 Pratinjau Dari Dalam Browser

Setelah Anda memahami susunan HTML dari hal-hal yang ingin Anda beri gaya, bagian yang sulit sudah selesai. Sekarang tibalah bagian yang menyenangkan dan familier, yaitu mendefinisikan pemilih gaya dan menentukan properti yang ingin Anda tetapkan. Untuk memengaruhi elemen div internal, tambahkan yang berikut di dalam tag gaya kita:

```
div div div {
```

```
  padding: 10px;
  margin: 10px;
  background-color: #ffde00;
  color: #333;
  display: inline-block;
  font-family: monospace;
  font-size: 32px;
  text-align: center;
}
```

Pemilih div div div akan memastikan kita menata hal-hal yang benar. Hasil akhirnya adalah vokal kita ditata agar terlihat persis seperti yang kita lihat sebelumnya. Dengan demikian, pemilih gaya div div div terlihat agak aneh, bukan? Itu terlalu umum. Dalam aplikasi dengan lebih dari tiga elemen div bersarang (yang akan sangat umum), Anda mungkin berakhir menata hal-hal yang salah.

Pada saat-saat seperti inilah Anda ingin mengubah HTML yang dihasilkan React agar konten kita lebih mudah ditata. Cara kita akan mengatasinya adalah dengan memberi elemen div bagian dalam kita nilai kelas huruf. Di sinilah JSX berbeda dari HTML. Buat perubahan yang disorot berikut:

```
var Letter = React.createClass({
  render: function() {
    return (
      <div className="letter">
        {this.props.children}
      </div>
    );
  }
});
```

Perhatikan bahwa kita menetapkan nilai kelas dengan menggunakan atribut className, bukan atribut class. Alasannya berkaitan dengan kata class yang merupakan kata kunci khusus dalam JavaScript. Jika itu tidak masuk akal mengapa itu penting, jangan khawatir tentang hal itu untuk saat ini. Kita akan membahasnya nanti. Bagaimanapun, setelah Anda memberi div Anda nilai atribut className berupa huruf, tinggal satu hal lagi yang harus dilakukan. Ubah pemilih CSS untuk menargetkan elemen div kita dengan lebih jelas:

```
.letter {
  padding: 10px;
  margin: 10px;
  background-color: #ffde00;
  color: #333;
  display: inline-block;
  font-family: monospace;
  font-size: 32px;
  text-align: center;
}
```

Seperti yang Anda lihat, menggunakan CSS adalah cara yang sangat tepat untuk menata konten di aplikasi berbasis React Anda. Di bagian berikutnya, kita akan melihat cara menata konten kita menggunakan pendekatan yang disukai oleh React.

4.3 MENATA KONTEN DENGAN CARA REACT

React lebih menyukai pendekatan sebaris untuk menata konten yang tidak menggunakan CSS. Meskipun awalnya tampak agak aneh, pendekatan ini dirancang untuk membantu membuat visual Anda lebih dapat digunakan kembali. Tujuannya adalah agar komponen Anda menjadi kotak hitam kecil tempat segala sesuatu yang terkait dengan tampilan dan cara kerja UI Anda tersimpan di sana. Mari kita lihat sendiri.

Melanjutkan contoh kita dari sebelumnya, hapus aturan gaya .letter. Setelah Anda melakukan ini, vokal Anda akan kembali ke status tanpa gaya saat Anda melihat pratinjau aplikasi Anda di browser. Untuk kelengkapan, Anda juga harus menghapus deklarasi className dari fungsi render komponen Letter kita. Tidak ada gunanya markup kita berisi hal-hal yang tidak akan kita gunakan. Saat ini, komponen Letter kita kembali ke status aslinya:

```
var Letter = React.createClass({
  render: function() {
    return (
      <div>
        {this.props.children}
      </div>
    );
  }
});
```

Cara Anda menentukan gaya di dalam komponen adalah dengan mendefinisikan objek yang isinya adalah properti CSS dan nilainya. Setelah Anda memiliki objek tersebut, Anda menetapkan objek tersebut ke elemen JSX yang ingin Anda beri gaya dengan menggunakan atribut style. Ini akan lebih masuk akal setelah kita melakukan kedua langkah ini sendiri, jadi mari kita terapkan semua ini untuk memberi gaya pada keluaran komponen Letter kita.

4.4 MEMBUAT OBJEK GAYA

Mari kita mulai dengan mendefinisikan objek yang berisi gaya yang ingin kita terapkan:

```
var Letter = React.createClass({
  render: function() {
    var letterStyle = {
      padding: 10,
      margin: 10,
      backgroundColor: "#ffde00",
      color: "#333",
      display: "inline-block",
      fontFamily: "monospace",
```

```
      fontSize: 32,
      textAlign: "center"
    };

  return (
    <div>
      {this.props.children}
    </div>
  );
  }
});
```

Kita memiliki objek yang disebut letterStyle, dan properti di dalamnya hanyalah nama properti CSS dan nilainya. Jika Anda belum pernah mendefinisikan properti CSS dalam JavaScript sebelumnya (misalnya, dengan menyetel object.style), rumus untuk mengubahnya menjadi sesuatu yang ramah JavaScript cukup sederhana:

- Properti CSS satu kata (seperti padding, margin, color) tetap tidak berubah.
- Properti CSS multikata dengan tanda hubung di dalamnya (seperti background-color, font-family, border-radius) diubah menjadi satu kata camelcase dengan tanda hubung dihilangkan dan kata-kata setelah tanda hubung ditulis dengan huruf kapital. Misalnya, menggunakan properti contoh kita, background-color akan menjadi backgroundColor, font-family akan menjadi fontFamily, dan border-radius akan menjadi borderRadius.

Objek letterStyle dan propertinya pada dasarnya merupakan terjemahan JavaScript langsung dari aturan gaya .letter yang kita lihat beberapa saat yang lalu. Yang tersisa sekarang adalah menetapkan objek ini ke elemen yang ingin kita beri gaya.

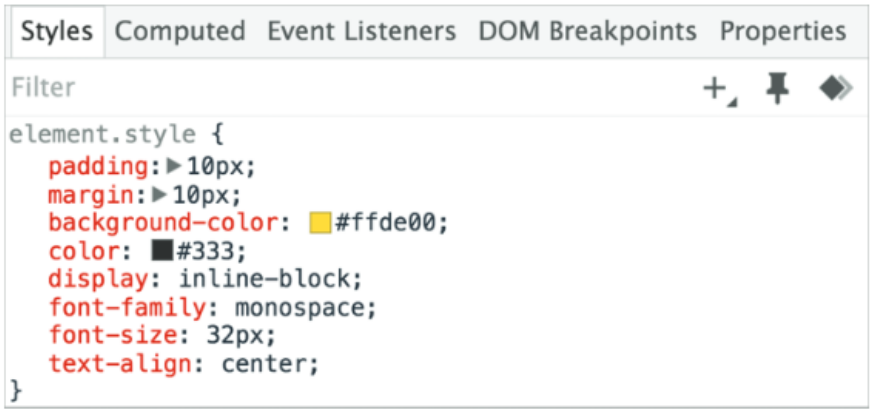
Sebenarnya Menata Konten Kita

Sekarang setelah kita memiliki objek yang berisi gaya yang ingin kita terapkan, sisanya sangat mudah. Temukan elemen yang ingin kita terapkan gayanya dan tetapkan atribut gaya untuk merujuk ke objek tersebut. Dalam kasus kita, itu akan menjadi elemen div yang dikembalikan oleh fungsi render komponen Letter kita. Perhatikan baris yang disorot untuk melihat bagaimana hal ini dilakukan pada contoh kita:

```
var Letter = React.createClass({
  render: function() {
    var letterStyle = {
      padding: 10,
      margin: 10,
      backgroundColor: "#ffde00",
      color: "#333",
      display: "inline-block",
      fontFamily: "monospace",
      fontSize: "32",
      textAlign: "center"
    };

  return (
```

```
      <div style={letterStyle}>
        {this.props.children}
      </div>
    );
    }
});
```

Objek kita disebut letterStyle, jadi itulah yang kita tentukan di dalam kurung kurawal agar React tahu untuk mengevaluasi ekspresi tersebut. Itu saja. Lanjutkan dan jalankan contoh di browser untuk memastikan semuanya berfungsi dengan baik dan semua vokal kita ditata dengan benar. Untuk beberapa validasi tambahan, jika Anda memeriksa gaya yang diterapkan pada salah satu vokal menggunakan alat pengembang browser pilihan Anda, Anda akan melihat bahwa gaya tersebut sebenarnya diterapkan secara inline (lihat Gambar 4.4).



Gambar 4.4 Gaya Diterapkan Sebaris

Meskipun ini bukan hal yang mengejutkan, mungkin sulit bagi kita yang terbiasa dengan gaya yang ada di dalam aturan gaya untuk menerimanya. Seperti kata pepatah, Times They Are A Changin'.

Anda Dapat Menghilangkan Sufiks "px"

Saat mengatur gaya secara terprogram, sangat merepotkan untuk berurusan dengan angka yang memerlukan sufiks nilai piksel. Untuk menghasilkan nilai-nilai ini, Anda perlu melakukan penggabungan string pada angka Anda untuk menambahkan px. Untuk mengonversi dari nilai piksel kembali ke angka, Anda perlu mengurai px. Semua ini tidak terlalu rumit atau memakan waktu, tetapi ini mengganggu. Untuk membantu hal ini, React memungkinkan Anda menghilangkan sufiks px untuk sejumlah properti CSS. Jika Anda ingat, objek letterStyle kita terlihat seperti berikut:

```
var letterStyle = {
  padding: 10,
  margin: 10,
  backgroundColor: "#ffde00",
  color: "#333",
  display: "inline-block",
```

```
    fontFamily: "monospace",
    fontSize: "32",
    textAlign: "center"
};=
```

Perhatikan bahwa untuk beberapa properti dengan nilai numerik seperti padding, margin, dan fontSize, kami sama sekali tidak menentukan sufiks px. Itu karena, saat runtime, React akan menambahkan sufiks px secara otomatis. Satu-satunya properti terkait angka yang tidak akan ditambahkan sufiks piksel secara otomatis oleh React adalah properti berikut: animationIterationCount, boxFlex, boxFlexGroup, boxOrdinal- Group, columnCount, fillOpacity, flex, flexGrow, flexPositive, flexShrink, flex- Negative, flexOrder, fontWeight, lineClamp, lineHeight, opacity, order, orphans, stopOpacity, strokeDashoffset, strokeOpacity, strokeWidth, tabSize, widows, zIndex, dan zoom.

Meskipun saya berharap dapat memberi tahu Anda bahwa saya mengingat informasi ini. Meskipun nilai piksel sangat bagus untuk banyak hal, Anda mungkin ingin menggunakan persentase, ems, vh, dll. untuk mewakili nilai Anda. Untuk nilai non-piksel ini, Anda masih harus memastikan sufiks ditangani secara manual. React tidak akan membantu Anda di sana, jadi jika Anda bukan penggemar nilai piksel, hal-hal yang bagus ini tidak banyak membantu Anda.

4.5 MEMBUAT WARNA LATAR BELAKANG DAPAT DISESUAIKAN

Hal terakhir yang akan kita lakukan sebelum mengakhiri pembahasan adalah memanfaatkan cara kerja React dengan gaya. Dengan mendefinisikan gaya kita di sekitar JSX, kita dapat membuat berbagai nilai gaya dapat disesuaikan dengan mudah oleh induk (alias konsumen komponen). Mari kita lihat cara kerjanya.

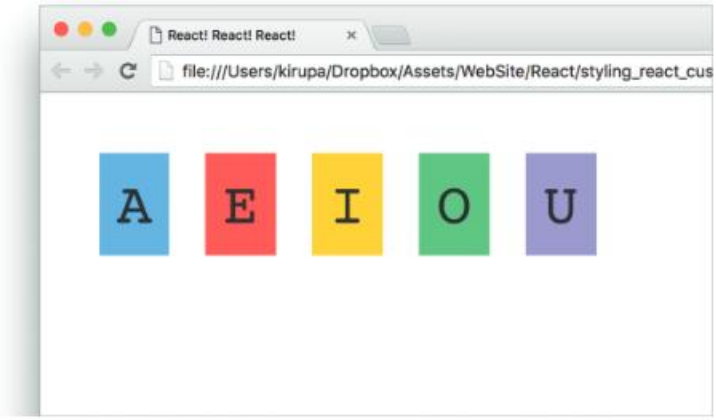
Saat ini, semua vokal kita memiliki latar belakang kuning. Bukankah lebih keren jika kita dapat menentukan warna latar belakang sebagai bagian dari setiap deklarasi Huruf? Untuk melakukannya, dalam metode ReactDOM.render kita, pertama-tama tambahkan atribut bgcolor dan tentukan beberapa warna seperti yang ditunjukkan pada baris yang disorot berikut:

```
ReactDOM.render(
  <div>
    <Letter bgcolor="#58B3FF">A</Letter>
    <Letter bgcolor="#FF605F">E</Letter>
    <Letter bgcolor="#FFD52E">I</Letter>
    <Letter bgcolor="#49DD8E">O</Letter>
    <Letter bgcolor="#AE99FF">U</Letter>
  </div>,
  destination
);
```

Selanjutnya, kita perlu menggunakan properti ini. Pada objek letterStyle kita, tetapkan nilai background- Color ke this.props.bgColor:

```
var letterStyle = {
   padding: 10,
   margin: 10,
   backgroundColor: this.props.bgcolor,
   color: "#333",
   display: "inline-block",
   fontFamily: "monospace",
   fontSize: "32",
   textAlign: "center"
};
```

Ini akan memastikan bahwa nilai backgroundColor disimpulkan dari apa yang kita tetapkan melalui atribut bgColor sebagai bagian dari deklarasi Letter. Jika Anda melihat pratinjau ini di browser Anda, Anda sekarang akan melihat vokal yang sama dengan beberapa warna latar belakang yang sangat menarik seperti yang ditunjukkan pada Gambar 4.5.



Gambar 4.5 Vokal Kita Dengan Warna Latar

Apa yang baru saja kita lakukan adalah sesuatu yang akan sangat sulit ditiru menggunakan CSS biasa. Sekarang, saat kita mulai melihat komponen yang isinya berubah berdasarkan status atau interaksi pengguna, Anda akan melihat lebih banyak contoh seperti itu di mana cara React dalam menata sesuatu memiliki banyak manfaat.

4.6 KESIMPULAN

Saat kita menyelami lebih jauh dan mempelajari lebih lanjut tentang React, Anda akan melihat beberapa kasus lagi di mana React melakukan sesuatu dengan sangat berbeda dari apa yang telah kita ketahui sebagai cara yang benar dalam melakukan sesuatu di web. Dalam tutorial ini, kita melihat React mempromosikan gaya sebaris dalam JavaScript sebagai cara untuk menata konten, bukan menggunakan aturan gaya CSS. Sebelumnya, kita melihat JSX dan bagaimana keseluruhan UI Anda dapat dideklarasikan dalam JavaScript menggunakan sintaksis seperti XML yang agak mirip dengan HTML.

Dalam semua kasus ini, jika Anda melihat lebih dalam di balik permukaan, alasan

mengapa React menyimpang dari kebijaksanaan konvensional sangat masuk akal. Membangun aplikasi dengan persyaratan UI yang sangat kompleks memerlukan cara baru untuk memecahkan tantangan yang terkait dengan UI yang kompleks. Teknik HTML, CSS, dan JavaScript yang mungkin sangat masuk akal saat menangani halaman web dan dokumen mungkin tidak berlaku di dunia aplikasi web tempat komponen digunakan kembali di dalam komponen lain.

Dengan demikian, Anda harus memilih teknik yang paling masuk akal untuk situasi Anda. Meskipun saya bias terhadap cara React dalam memecahkan masalah pengembangan UI kita, saya akan berusaha sebaik mungkin untuk menyoroti metode alternatif atau konvensional juga. Mengkaitkannya kembali dengan apa yang kita lihat di sini, menggunakan aturan gaya CSS dengan konten React Anda sepenuhnya boleh-boleh saja selama Anda membuat keputusan dengan mengetahui hal-hal yang Anda peroleh serta kerugian dengan melakukannya.

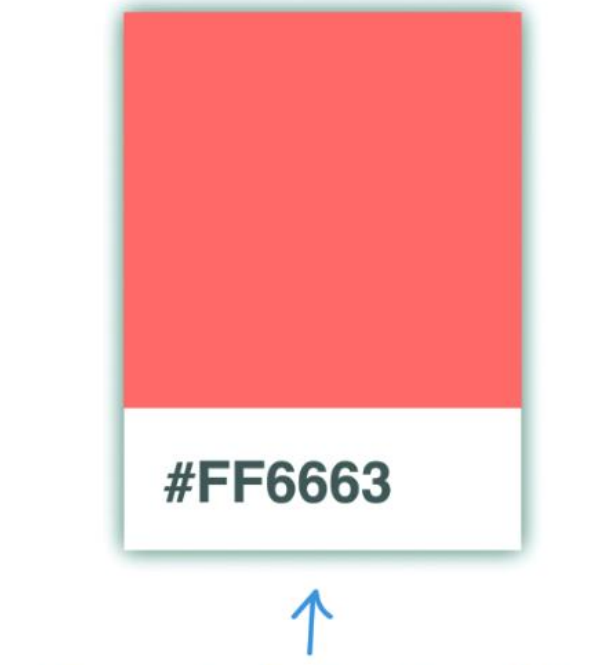
BAB 5
MEMBUAT KOMPONEN KOMPLEKS

5.1 DARI VISUAL KE KOMPONEN

Berbagai contoh yang telah kita bahas sejauh ini cukup mendasar. Mereka bagus dalam menyoroti konsep teknis (lihat Gambar 5.1), namun tidak bagus dalam mempersiapkan Anda menghadapi dunia nyata.

Gambar 5.1 Sangat Bagus Untuk Menyorot Konsep Teknis

Di dunia nyata, apa yang akan diminta untuk Anda terapkan di React tidak akan pernah sesederhana daftar nama atau blok vokal berwarna-warni. Sebaliknya, Anda akan diberikan representasi visual dari beberapa antarmuka pengguna yang kompleks. Visual tersebut dapat berupa berbagai bentuk coretan, diagram, tangkapan layar, video, redline, comp, dll.



Gambar 5.2 Kartu Palet Warna Sederhana

Terserah Anda untuk menghidupkan semua piksel statis tersebut, dan kita akan berlatih langsung untuk melakukannya. Yang akan kita lakukan adalah membuat kartu palet warna sederhana (lihat Gambar 5.2).

Jika Anda tidak yakin apa itu, ini adalah kartu persegi panjang kecil yang membantu Anda mencocokkan warna dengan jenis cat tertentu. Anda akan sering melihatnya di toko perkakas rumah atau tempat penjualan cat. Teman desainer Anda mungkin memiliki lemari besar khusus untuk kartu ini. Bagaimanapun, misi kita adalah membuat ulang salah satu kartu ini menggunakan React.

Ada beberapa cara untuk melakukannya, tetapi saya akan menunjukkan kepada Anda pendekatan yang sangat sistematis yang akan membantu Anda menyederhanakan dan memahami bahkan antarmuka pengguna yang paling rumit sekalipun. Pendekatan ini melibatkan dua langkah:

1. Identifikasi elemen visual utama
2. Cari tahu apa saja komponennya

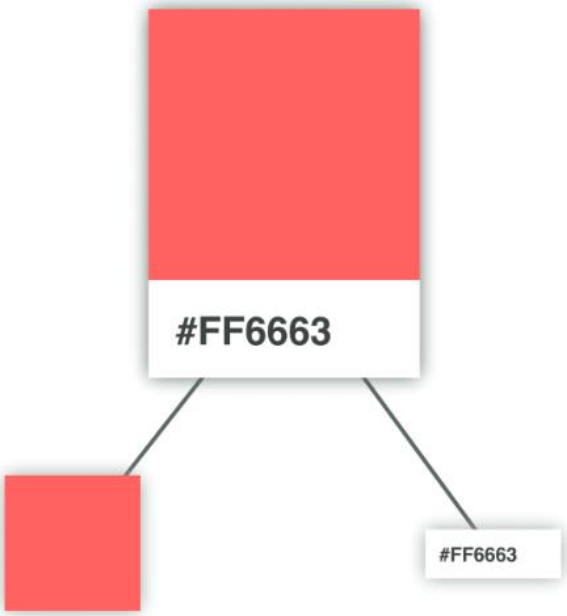
Kedua langkah ini terdengar sangat rumit, tetapi saat kita membahasnya, Anda akan melihat bahwa tidak ada yang perlu dikhawatirkan.

5.2 MENGIDENTIFIKASI ELEMEN VISUAL UTAMA

Langkah pertama adalah mengidentifikasi semua elemen visual yang sedang kita bahas. Tidak ada elemen visual yang terlalu kecil untuk dihilangkan setidaknya tidak pada awalnya. Cara termudah untuk mulai mengidentifikasi bagian-bagian yang relevan adalah dengan memulai dengan elemen visual yang jelas dan kemudian menyelami elemen-elemen yang kurang jelas. Hal pertama yang akan Anda lihat dalam contoh kita adalah kartu itu sendiri (lihat Gambar 5.3).

Gambar 5.3 Kartu

Di dalam kartu, Anda akan melihat bahwa ada dua area yang berbeda. Area atas adalah area persegi panjang yang menampilkan warna tertentu. Area bawah adalah area putih yang menampilkan nilai heksadesimal. Mari kita sebutkan kedua elemen visual ini dan susun menjadi struktur seperti pohon seperti yang ditunjukkan pada Gambar 5.4.

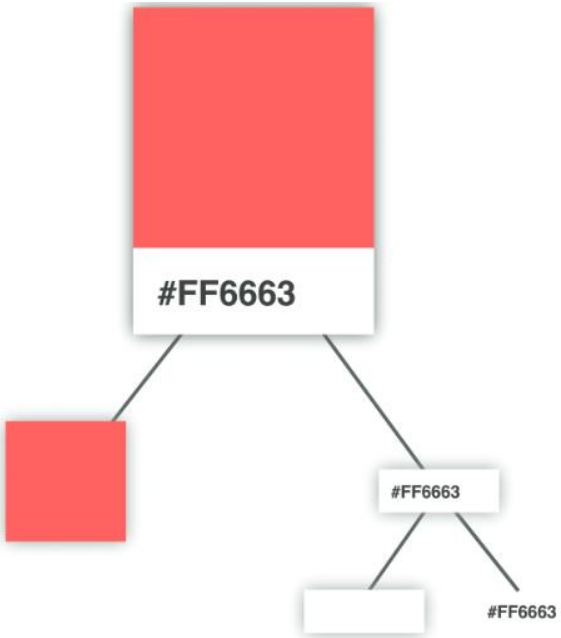


Gambar 5.4 Struktur Seperti Pohon

Menata visual Anda ke dalam struktur seperti pohon ini (alias hierarki visual) adalah cara yang baik untuk mendapatkan gambaran yang lebih baik tentang bagaimana elemen visual Anda dikelompokkan. Tujuan dari latihan ini adalah untuk mengidentifikasi elemen visual yang penting dan membaginya ke dalam susunan induk/anak hingga Anda tidak dapat membaginya lagi.

Coba Abaikan Detail Implementasi

Meskipun sulit, jangan pikirkan detail implementasi dulu. Jangan fokus pada pembagian elemen visual Anda berdasarkan kombinasi HTML dan CSS yang diperlukan. Masih banyak waktu untuk itu nanti! Jika kita lanjutkan, kita dapat melihat bahwa persegi panjang berwarna kita bukanlah sesuatu yang dapat kita bagi lebih jauh. Namun, itu tidak berarti kita sudah selesai. Kita dapat membagi label lebih jauh dari area putih yang mengelilinginya. Saat ini, hierarki visual kita terlihat seperti yang ditunjukkan pada Gambar 5.5 dengan label dan area putih menempati tempat terpisah di pohon kita.



Gambar 5.5 Membagi Hal-Hal Lebih Jauh Ke Dalam Label Dan Area Putih Yang Mengelilinginya

Pada titik ini, kita tidak memiliki hal lain untuk dibagi lebih jauh. Kita telah selesai mengidentifikasi dan membagi elemen visual kita, jadi langkah selanjutnya adalah menggunakan apa yang telah kita temukan di sini untuk membantu kita mengidentifikasi komponen.

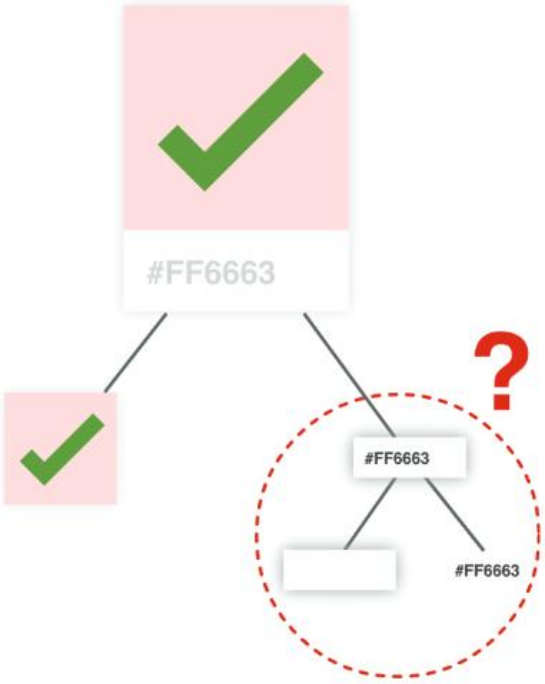
Mengidentifikasi Komponen

Di sinilah hal-hal menjadi sedikit menarik. Kita perlu mencari tahu elemen visual mana yang telah kita identifikasi yang akan diubah menjadi komponen dan mana yang tidak. Tidak setiap elemen visual perlu diubah menjadi komponen, dan kita tentu tidak ingin membuat hanya beberapa komponen yang sangat rumit. Perlu ada keseimbangan (lihat Gambar 5.6).

Gambar 5.6 Komponen Yang Tidak Terlalu Sedikit Dan Tidak Terlalu Banyak

Ada seni untuk menentukan elemen visual mana yang menjadi bagian dari suatu komponen dan mana yang tidak. Aturan umumnya adalah komponen kita harus melakukan satu hal saja. Jika Anda merasa komponen potensial Anda akan melakukan terlalu banyak hal,

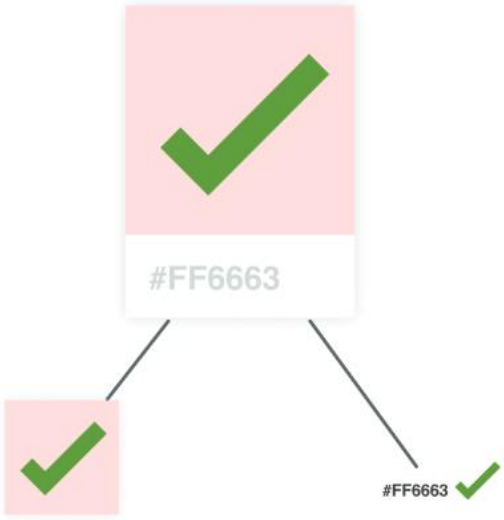
Anda mungkin ingin memecah komponen menjadi beberapa komponen. Di sisi lain, jika komponen potensial Anda melakukan terlalu sedikit hal, Anda mungkin ingin melewatkan pembuatan elemen visual tersebut menjadi suatu komponen sama sekali.



Gambar 5.7 Tanda Tanya Di Sekitar Label Dan Ruang Putih Di Sekitarnya

Mari kita coba mencari tahu elemen mana yang akan menjadi komponen yang baik dalam contoh kita. Dari melihat hierarki visual kita, langsung saja, baik kartu maupun persegi panjang berwarna tampak cocok untuk membuat komponen yang hebat. Kartu bertindak sebagai wadah luar, dan persegi panjang berwarna hanya menampilkan warna. Namun, itu hanya memberi tanda tanya di sekitar label kita dan area putih yang dikelilinginya (lihat Gambar 5.7).

Bagian penting di sini adalah label itu sendiri. Tanpanya, kita tidak dapat melihat nilai heksadesimal. Yang tersisa hanyalah area putih. Tujuannya tidak penting. Itu hanyalah ruang kosong, dan tanggung jawab untuk itu dapat dengan mudah diserahkan kepada label itu sendiri. Bersiaplah untuk apa yang akan saya katakan selanjutnya. Sayangnya, area persegi panjang putih kita tidak akan diubah menjadi komponen. Pada titik ini, kita telah mengidentifikasi tiga komponen kita, dan hierarki komponen tampak seperti pada Gambar 5-8.



Gambar 5.8 Tiga Komponen

Hal penting yang perlu diperhatikan adalah hierarki komponen lebih berperan dalam membantu kita mendefinisikan kode daripada dalam menentukan tampilan produk akhir. Anda akan melihat bahwa tampilannya sedikit berbeda dari hierarki visual yang kita buat sebelumnya. Untuk detail visual, Anda harus selalu merujuk ke materi sumber (alias komposisi visual, redline, tangkapan layar, dan item terkait lainnya). Untuk menentukan komponen mana yang akan dibuat, Anda harus menggunakan hierarki komponen. Baiklah, setelah kita mengidentifikasi komponen dan hubungan di antara semuanya, sekarang saatnya untuk mulai menghidupkan kartu palet warna kita.

5.3 MEMBUAT KOMPONEN

Ini adalah bagian yang mudah atau semacamnya! Sekarang saatnya bagi kita untuk mulai menulis beberapa kode. Hal pertama yang kita perlukan adalah halaman HTML yang sebagian besar kosong yang akan berfungsi sebagai titik awal kita:

```
<!DOCTYPE html>
<html>

<head>
  <title>More Components!</title>
    <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
    <script src="https://unpkg.com/react-dom@15.3.2/dist/react-
    dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
    core/5.8.23/browser.min.js"></script>

  <style>
    #container {
      padding: 50px;
      background-color: #FFF;
  }
  </style>
```

```
</head>

<body>
   <div id="container"></div>
   <script type="text/babel">

      ReactDOM.render(
      <div>

      </div>,
      document.querySelector("#container")
      );
   </script>

</body>
</html>
```

Luangkan waktu sejenak untuk melihat apa yang sedang terjadi di halaman ini. Tidak banyak yang perlu dilakukan hanya hal-hal minimum yang diperlukan agar React dapat merender div kosong ke dalam elemen kontainer kita. Setelah Anda melakukan ini, sekarang saatnya untuk menentukan tiga komponen kita. Nama yang akan kita gunakan untuk komponen kita adalah Card, Label, dan Square. Lanjutkan dan tambahkan baris yang disorot berikut tepat di atas fungsi ReactDOM.render:

```
var Square = React.createClass({
   render: function() {
      return(
         <p>Nothing</p>
   );
   }
});

var Label = React.createClass({
   render: function() {
      return (
         <p>Nothing</p>
   );
   }
   });

var Card = React.createClass({
   render: function() {
      return (

   );
   }
});

ReactDOM.render(
   <div>
```

```
    </div>,
    document.querySelector("#container")
);
```

Dalam tiga komponen kita, kita juga memasukkan fungsi render yang mutlak dibutuhkan setiap komponen agar berfungsi. Selain itu, komponen kita kosong. Di bagian berikut, kita akan memperbaikinya dengan mengisinya.

Komponen Kartu

Kita akan mulai dari bagian atas hierarki komponen kita dan fokus pada komponen Kartu terlebih dahulu. Komponen ini akan bertindak sebagai wadah tempat komponen Kotak dan Label kita akan berada. Untuk menerapkannya, lanjutkan dan buat modifikasi yang disorot berikut ini:

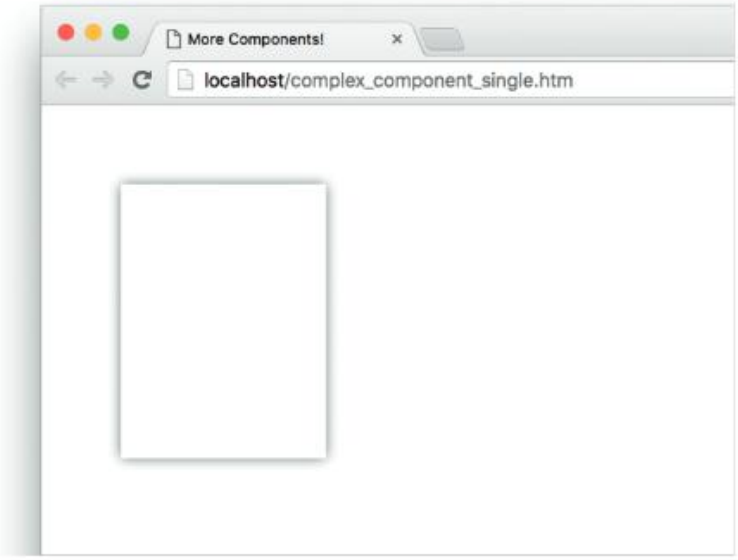
```
var Card = React.createClass({
  render: function() {
    var cardStyle = {
      height: 200,
      width: 150,
      padding: 0,
      backgroundColor: "#FFF",
      WebkitFilter: "drop-shadow(0px 0px 5px #666)",
      filter: "drop-shadow(0px 0px 5px #666)"
    };

  return (
    <div style={cardStyle}>

    </div>
  );
  }
});
```

Meskipun ini tampak seperti banyak perubahan, sebagian besar baris digunakan untuk menata keluaran komponen Card kita melalui objek cardStyle. Di dalam objek, perhatikan bahwa kita menetapkan versi properti filter CSS dengan awalan vendor dengan WebkitFilter. Itu bukan detail yang menarik. Detail yang menarik adalah kapitalisasi. Alih-alih huruf pertama ditulis dengan huruf besar seperti webkitFilter, huruf W sebenarnya ditulis dengan huruf kapital. Itu bukan cara properti CSS normal lainnya direpresentasikan, jadi ingatlah itu jika Anda perlu menetapkan properti dengan awalan vendor.

Perubahan lainnya tidak begitu mengesankan. Kita mengembalikan elemen div, dan atribut style elemen itu ditetapkan ke objek cardStyle kita. Sekarang, untuk melihat komponen Card kita beraksi, kita perlu menampilkannya di DOM kita sebagai bagian dari fungsi ReactDOM.render. Untuk mewujudkannya, lanjutkan dan buat perubahan yang disorot berikut:

```
ReactDOM.render(
  <div>
    <Card/>
  </div>,
  document.querySelector("#container")
);
```

Yang kami lakukan hanyalah memberi tahu fungsi ReactDOM.render untuk merender output dari komponen Card kami dengan memanggilnya. Jika semuanya berjalan dengan benar, Anda akan melihat hal yang sama seperti pada Gambar 5.9 jika Anda menguji aplikasi Anda.



Gambar 5.9 Hasil Pengujian Anda Garis Luar Kartu Palet Warna

Ya, itu hanya garis luar kartu palet warna kita, tetapi itu jelas lebih dari apa yang kita mulai beberapa saat yang lalu!

Komponen Persegi

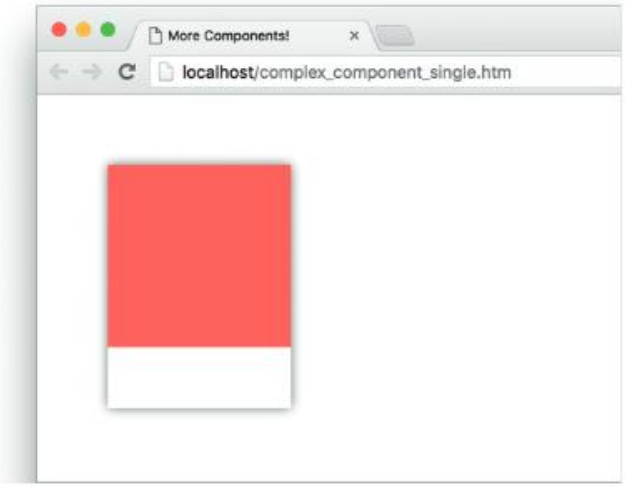
Saatnya untuk turun satu tingkat dalam hierarki komponen dan melihat komponen Persegi. Ini adalah komponen yang cukup mudah, jadi buat perubahan yang disorot berikut:

```
var Square = React.createclass ({
  render: function() {
    var squarestyle = {
      height: 150,
     backgroundColor: "#FF6663"
   };
   return(
    <div style={squarestyle}>

   </div>
  };
 }
});
```

Sama seperti komponen Card, kita mengembalikan elemen div yang atribut style-nya disetel ke objek style yang menentukan tampilan komponen ini. Untuk melihat komponen Square beraksi, kita perlu memasukkannya ke DOM seperti yang kita lakukan dengan komponen Card sebelumnya. Perbedaan kali ini adalah kita tidak akan memanggil komponen Square melalui fungsi ReactDOM.render. Sebaliknya, kita akan memanggil komponen Square dari dalam komponen Card. Untuk melihat maksud saya, kembali ke fungsi render komponen Card, dan buat perubahan berikut:

```
var Card = React.createClass({
  render: function() {
    var cardStyle = {
      height: 200,
      width: 150,
      padding: 0,
      backgroundColor: "#FFF",
      WebkitFilter: "drop-shadow(0px 0px 5px #666)",
      filter: "drop-shadow(0px 0px 5px #666)"
    };

    return (
      <div style={cardStyle}>
        <Square/>
      </div>
    );
  }
});
```

Pada titik ini, jika Anda melihat pratinjau aplikasi kami, Anda akan melihat kotak berwarna-warni muncul (lihat Gambar 5.10).



Gambar 5.10 Bagian Merah Muncul

Hal yang menarik untuk disebutkan adalah kita memanggil komponen Square dari dalam

komponen Card! Ini adalah contoh komposisi komponen di mana satu komponen bergantung pada output dari komponen lain. Hal terakhir yang Anda lihat adalah hasil dari kedua komponen ini yang saling berkolusi. Bukankah kolusi itu indah setidaknya dalam konteks ini?

Komponen Label

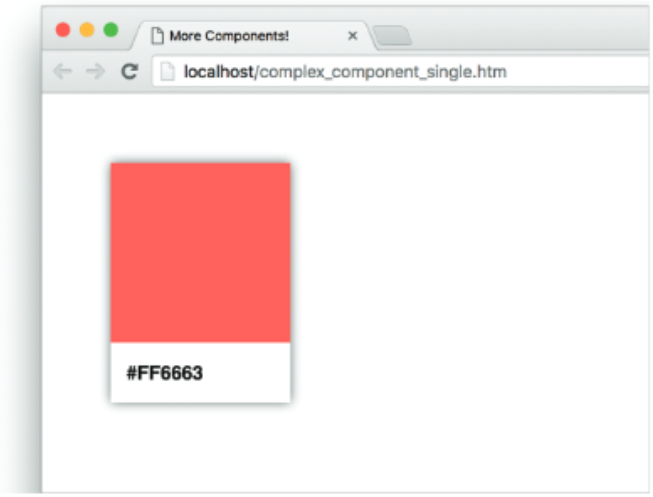
Komponen terakhir yang tersisa adalah Label kita. Lanjutkan dan buat perubahan yang disorot berikut ini:

```
var Label = React.createclass ({
   render: function () {
      var labelstyle = {
   fontFamily: "sans-serif",
   fontweight: "bold",
   padding: 13,
   margin: 0
      };

    return (
        <p style={labelStyle}>#FF6663</p>
     );
   }
});
```

Pola yang kita lakukan seharusnya sudah menjadi rutinitas Anda sekarang. Kita memiliki objek gaya yang kita tetapkan untuk apa yang kita kembalikan. Yang kita kembalikan adalah elemen p yang isinya adalah string #FF6663. Agar apa yang kita kembalikan akhirnya masuk ke DOM kita, kita perlu memanggil komponen Label kita melalui komponen Card kita. Lanjutkan dan buat perubahan yang disorot berikut ini:

```
1 var Card = React.createclass ({
2    render: function() {
3     var cardstyle = {
4        height: 200,
5        width: 150,
6        padding: 0,
7        backgroundColor: "#FFF",
8        WebkitFilter: "drop-shadow(0px 0px 5px #666) ",
9        filter: "drop-shadow(0px 0px 5px #666) "
10    };
11
12    return (
13       <div style={cardstyle}>
14         <Square/>
15         <Label/>
16       </div>
17     );
18    }
19 });
```

Perhatikan bahwa komponen Label kita berada tepat di bawah komponen Square yang kita tambahkan ke fungsi pengembalian komponen Card kita sebelumnya. Jika Anda melihat pratinjau aplikasi Anda di browser sekarang, Anda akan melihat sesuatu yang tampak seperti Gambar 5.11.



Gambar 5.11 Label Muncul

Ya, benar! Kartu palet warna kita sudah jadi dan terlihat, berkat usaha komponen Card, Square, dan Label kita. Namun, itu tidak berarti kita sudah selesai. Ada beberapa hal lagi yang harus dibahas.

Dalam contoh kita saat ini, kita mengodekan nilai warna yang digunakan oleh komponen Square dan Label kita. Itu adalah hal yang aneh untuk dilakukan yang mungkin atau mungkin tidak dilakukan dengan sengaja untuk efek dramatis, tetapi memperbaikinya mudah saja. Kita hanya perlu menentukan nama properti dan mengaksesnya melalui this.props. Kita telah melihat semua ini sebelumnya. Yang berbeda adalah berapa kali kita harus melakukan ini.

Tidak ada cara untuk menentukan properti dengan benar pada komponen induk dan membuat semua turunan secara otomatis mendapatkan akses ke properti itu. Ada banyak cara yang tidak tepat untuk menangani hal ini seperti mendefinisikan objek global, menetapkan nilai pada properti komponen secara langsung, dan sebagainya. Kita tidak akan memikirkan solusi yang tidak tepat seperti itu sekarang. Kita bukan hewan! Bagaimanapun, cara yang tepat untuk meneruskan nilai properti ke komponen anak adalah dengan meminta setiap komponen induk perantara meneruskan properti tersebut juga. Untuk melihat ini dalam tindakan, lihat perubahan yang disorot pada kode kita saat ini di mana kita beralih dari warna yang dikodekan secara kaku dan menentukan warna kartu kita menggunakan properti warna sebagai gantinya:

```
1 var Square = React.createclass ({
2    render: function() {
3     var squarestyle = {
```

```
4         height: 150,
5         backgroundColor: this.props.color
6     };
7     return(
8        <div style={squarestyle}>
9
10       </div>
11      );
12   }
13 });
14
15 var Label = React.createclass ({
16      render: function() {
17    var labelstyle = {
18    fontFamily: "sans-serif",
19    fontweight: "bold",
20    padding: 13,
21    margin: 0
22      };
23
24      return (
25    <p style={labelstyle}>{this.props.color}</p>
26    );
27     }
28 });
29
30 var Card = React.createclass ({
31     render: function() {
32    var cardstyle = {
33    height: 200,
34    width: 150,
35    padding: 0,
36    backgroundColor: "#FFF",
37    WebkitFilter: "drop-shadow(0px 0px 5px #666) ",
38    filter: "drop-shadow(0px 0px 5px #666)"
39     };
40
41     return (
42    <div style={cardstyle}>
43       <Square color={this.props.color}/>
44       <Label color={this.props.color}/>
45    </div>
46        ) ;
47    }
48 });
49
50 ReactDOM.render (
51    <div>
52    <Card color="#FF6663"/>
53    </div>,
54    document.queryselector ("#container")
55 ) ;
```

Setelah Anda membuat perubahan ini, Anda dapat menentukan warna hex apa pun yang Anda inginkan sebagai bagian dari pemanggilan komponen Kartu:

```
1 ReactDOM.render (
2       <div>
3          <Card color="#FFA737"/>
4       </div>,
5       document.queryselector ("#container")
6 );
```

Kartu palet warna yang dihasilkan akan menampilkan warna yang Anda tentukan (lihat Gambar 5.12).

Gambar 5.12 Warna Untuk Nilai Hex #FFA737

Sekarang, mari kita kembali ke perubahan yang kita buat. Meskipun properti warna hanya digunakan oleh komponen Square dan Label, komponen induk Card bertanggung jawab untuk meneruskan properti tersebut kepada komponen-komponen tersebut. Untuk situasi yang lebih bertingkat, Anda akan memiliki lebih banyak komponen perantara yang akan bertanggung jawab untuk mentransfer properti. Keadaannya semakin buruk.

Ketika Anda memiliki beberapa properti yang ingin Anda teruskan ke beberapa level komponen, jumlah pengetikan (atau penyalinan/penempelan) yang Anda lakukan juga akan meningkat banyak. Ada beberapa cara untuk mengurangi hal ini, dan kita akan melihat strategi mitigasi tersebut secara lebih rinci di bab berikutnya.

Mengapa Komposabilitas Komponen Sangat Bagus

Saat kita berkutat di React, kita sering lupa bahwa yang kita buat sebenarnya hanyalah HTML, CSS, dan JavaScript yang biasa saja dan membosankan. HTML yang dihasilkan untuk kartu palet warna kita terlihat seperti berikut:

```
<div id="container">
  <div data-reactid=".0">
```

```
      <div style="height:200px;
                  width:150px;
                  padding:0;
                  background-color:#FFF;
                  -webkit-filter:drop-shadow(0px 0px 5px #666);
                  filter:drop-shadow(0px 0px 5px #666);">
      <div style="height:150px;
                  background-color:#FF6663;"></div>
      <p style="font-family:sans-serif;
                font-weight:bold;
                padding:13px;
                margin:0;">#FF6663</p>
      </div>
    </div>
  </div>
```

Markup ini tidak tahu bagaimana ia sampai di sana. Ia tidak tahu komponen mana yang bertanggung jawab atas apa. Ia tidak peduli tentang komposisi komponen atau cara yang membuat frustrasi saat kita harus mentransfer properti warna dari induk ke anak. Itu memunculkan poin penting yang perlu dijelaskan. Jika kita harus menggeneralisasi hasil akhir dari apa yang dilakukan komponen, yang mereka lakukan hanyalah mengembalikan gumpalan HTML ke apa pun yang memanggilnya. Fungsi render setiap komponen mengembalikan beberapa HTML ke fungsi render komponen lain.

Semua HTML ini terus terakumulasi hingga gumpalan HTML raksasa didorong (dengan sangat efisien) ke DOM kita. Kesederhanaan itulah sebabnya penggunaan kembali dan komposisi komponen bekerja dengan sangat baik. Setiap gumpalan HTML bekerja secara independen dari gumpalan HTML lainnya terutama jika Anda menentukan gaya sebaris seperti yang direkomendasikan React. Ini memungkinkan Anda untuk dengan mudah membuat elemen visual dari elemen visual lain tanpa harus khawatir tentang apa pun.
APA SAJA! Bukankah itu sangat mengagumkan?

5.4 KESIMPULAN

Seperti yang mungkin sudah Anda sadari sekarang, kita perlahan-lahan mengalihkan fokus ke skenario yang lebih canggih yang membuat React berkembang pesat. Sebenarnya, canggih bukanlah kata yang tepat. Kata yang tepat adalah realistis. Dalam bab ini, kita mulai dengan mempelajari cara melihat sepotong UI dan mengidentifikasi komponen dengan cara yang dapat Anda terapkan nanti. Itu adalah situasi yang akan Anda hadapi sepanjang waktu. Meskipun pendekatan yang kami gunakan tampak sangat formal, saat Anda semakin berpengalaman dalam membuat sesuatu di React, Anda dapat mengurangi formalitasnya. Jika Anda dapat dengan cepat mengidentifikasi komponen dan hubungan induk/anaknya tanpa membuat hierarki visual dan komponen, maka itu adalah satu tanda lagi bahwa Anda semakin ahli dalam bekerja dengan React!

Mengidentifikasi komponen hanyalah satu bagian dari persamaan. Bagian lainnya adalah menghidupkan komponen tersebut. Sebagian besar hal teknis yang kita lihat di sini

hanyalah perluasan kecil dari apa yang telah kita lihat sebelumnya. Kita melihat satu tingkat komponen di bab sebelumnya, dan di sini kita melihat cara bekerja dengan beberapa tingkat komponen. Kita telah membahas cara meneruskan properti antara satu induk dan satu anak di bab sebelumnya, dan di sini kita membahas cara meneruskan properti antara beberapa induk dan beberapa anak. Mungkin di bab berikutnya kita akan melakukan sesuatu yang inovatif seperti menggambar beberapa kartu palet warna di layar! Atau, kita mungkin dapat menentukan dua properti alih-alih hanya satu. Siapa tahu?

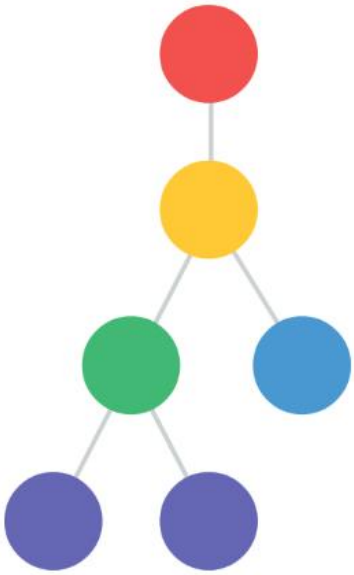
BAB 6
MENTRANSFER PROPERTI (PROP)

6.1 MEMPERMUDAH PROPERTI DI BEBERAPA KOMPONEN

Ada sisi yang membuat frustrasi saat bekerja dengan properti. Kita sudah melihat sisi ini di bab sebelumnya. Melewatkan properti dari satu komponen ke komponen lain itu mudah dan sederhana saat Anda hanya berurusan dengan satu lapisan komponen. Saat Anda ingin mengirim properti ke beberapa lapisan komponen, semuanya mulai menjadi rumit. Hal-hal yang menjadi rumit bukanlah hal yang baik, jadi dalam bab ini, mari kita lihat apa yang dapat kita lakukan untuk mempermudah bekerja dengan properti di beberapa lapisan komponen.

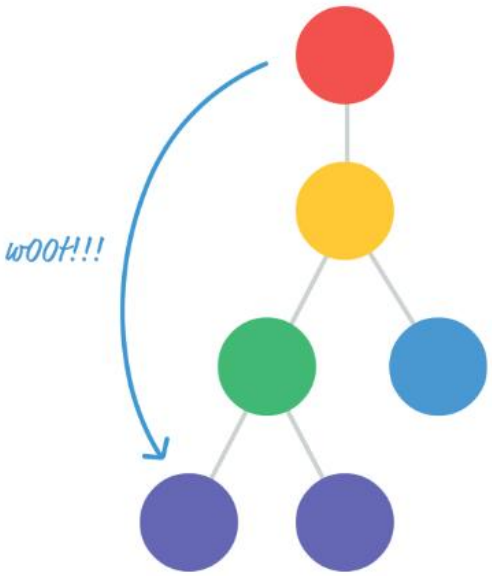
Gambaran Umum Masalah

Misalnya Anda memiliki komponen yang sangat bertingkat, dan hierarkinya (dimodelkan sebagai lingkaran berwarna yang mengagumkan) tampak seperti Gambar 6.1.



Gambar 6.1 Hirarki Komponen

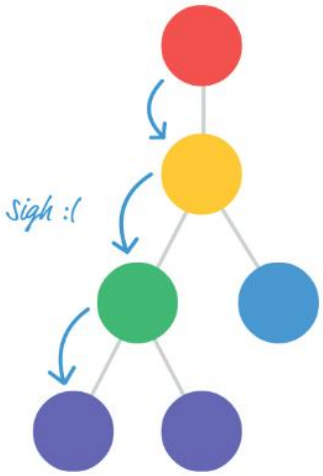
Yang ingin Anda lakukan adalah meneruskan properti dari lingkaran merah ke lingkaran ungu tempat properti tersebut akan digunakan. Yang tidak dapat kita lakukan adalah hal yang sangat jelas dan mudah seperti yang ditunjukkan pada Gambar 6.2.



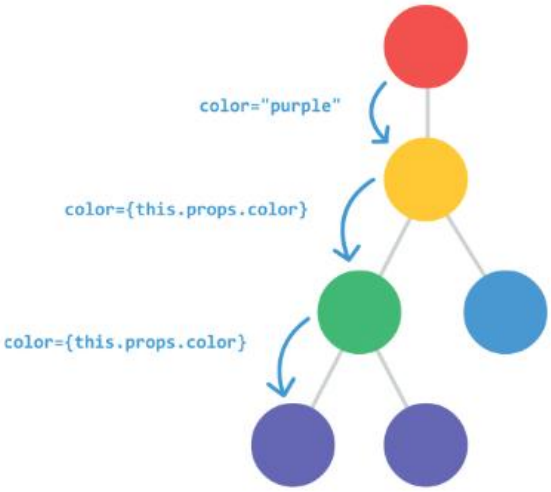
Gambar 6.2 Tidak Dapat Melakukan Ini

Anda tidak dapat meneruskan properti secara langsung ke komponen atau komponen-komponen yang ingin Anda targetkan. Alasannya berkaitan dengan cara kerja React. React memberlakukan rantai perintah di mana properti harus mengalir turun dari komponen induk ke komponen anak langsung. Ini berarti Anda tidak dapat melewati lapisan anak saat mengirim properti. Ini juga berarti anak Anda tidak dapat mengirim properti kembali ke induk. Semua komunikasi bersifat satu arah dari induk ke anak.

Berdasarkan panduan ini, meneruskan properti dari lingkaran merah ke lingkaran ungu tampak seperti Gambar 6.3. Setiap komponen yang berada di jalur yang dituju harus menerima properti dari induknya lalu mengirim ulang properti tersebut ke anaknya. Proses ini berulang hingga properti Anda mencapai tujuan yang dituju. Masalahnya ada pada langkah penerimaan dan pengiriman ulang ini. Jika kita harus mengirim properti yang disebut warna dari komponen yang mewakili lingkaran merah kita ke komponen yang mewakili lingkaran ungu kita, jalurnya ke tujuan akan terlihat seperti Gambar 6.4.

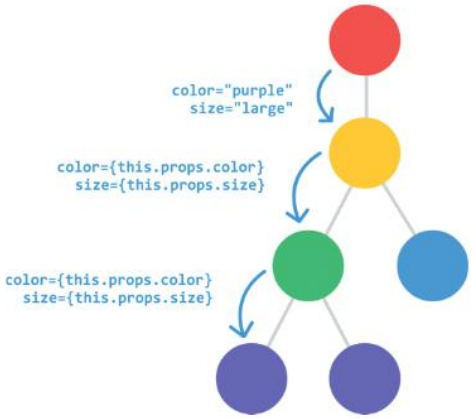


Gambar 6.3 Properti Diteruskan Dari Induk Ke Anak



Gambar 6.4 Mengirimkan Properti Warna

Sekarang, bayangkan kita memiliki dua properti yang perlu kita kirim, seperti pada Gambar 6.5.



Gambar 6.5 Mengirim Dua Properti

Bagaimana jika kita ingin mengirim tiga properti? Atau empat?

Kita dapat dengan cepat melihat bahwa pendekatan ini tidak dapat diskalakan maupun dipelihara. Untuk setiap properti tambahan yang perlu kita komunikasikan, kita harus menambahkan entri untuknya sebagai bagian dari deklarasi setiap komponen. Jika kita memutuskan untuk mengganti nama properti kita di beberapa titik, kita harus memastikan bahwa setiap contoh properti tersebut juga diganti namanya. Jika kita menghapus properti, kita perlu menghapus properti tersebut dari setiap komponen yang bergantung padanya. Secara keseluruhan, ini adalah jenis situasi yang kita coba hindari saat menulis kode. Apa yang dapat kita lakukan untuk mengatasinya?

Tinjauan Mendetail tentang Masalah

Di bagian sebelumnya, kita membahas secara mendalam tentang apa masalahnya. Sebelum kita dapat menyelami untuk mencari solusi, kita perlu melihat lebih jauh dari sekadar

diagram dan melihat contoh yang lebih mendetail dengan kode nyata. Kita perlu melihat sesuatu seperti berikut:

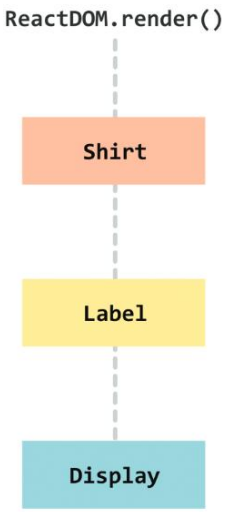
```
var Display = React.createClass({
  render: function() {
    return(
      <div>
        <p>{this.props.color}</p>
        <p>{this.props.num}</p>
        <p>{this.props.size}</p>
      </div>
    );
  }
});

var Label = React.createClass({
  render: function() {
    return (
      <Display color={this.props.color}
        num={this.props.num}
        size={this.props.size}/>
    );
  }
});

var Shirt = React.createClass({
  render: function() {
    return (
      <div>
        <Label color={this.props.color}
          num={this.props.num}
          size={this.props.size}/>
      </div>
    );
  }
});

ReactDOM.render(
  <div>
    <Shirt color="steelblue" num="3.14" size="medium"/>
  </div>,
  document.querySelector("#container")
);
```

Luangkan waktu sejenak untuk memahami apa yang sedang terjadi. Setelah Anda melakukannya, mari kita bahas contoh ini bersama-sama. Yang kita miliki adalah komponen Shirt yang bergantung pada output komponen Label yang bergantung pada output komponen Display. (Coba ucapkan kalimat itu lima kali dengan cepat!) Bagaimanapun, hierarki komponen dapat dilihat pada Gambar 6.6. Ketika Anda menjalankan kode ini, yang dihasilkan tidak

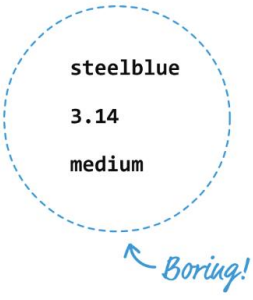
istimewa. Hanya tiga baris teks (lihat Gambar 6.7). Bagian yang menarik adalah bagaimana teks tersebut muncul. Masing-masing dari tiga baris teks yang Anda lihat dipetakan ke properti yang kami tentukan di awal di dalam ReactDOM.render:

```
<Shirt color="steelblue" num="3.14" size="medium"/>
```

Properti color, num, dan size (dan nilainya) melakukan perjalanan hingga ke komponen Display yang akan membuat pengembara dunia yang paling berpengalaman pun iri. Mari kita ikuti properti-properti ini dari awal hingga saat digunakan, dan saya sadar bahwa sebagian besar dari ini akan menjadi tinjauan tentang apa yang telah Anda lihat. Jika Anda merasa bosan, silakan lanjutkan ke bagian berikutnya. Dengan demikian, teruslah maju!



Gambar 6.6 Hirarki Komponen



Gambar 6.7 Tiga Baris Teks

Properti kita dimulai di dalam ReactDOM.render saat komponen Shirt kita dipanggil dengan properti color, num, dan size yang ditentukan:

```
ReactDOM.render(
  <div>
    <Shirt color="steelblue" num="3.14" size="medium"/>
  </div>,
  document.querySelector("#container")
```

```
  );
```

Kita tidak hanya mendefinisikan properti, kita juga menginisialisasinya dengan nilai yang akan dibawanya. Di dalam komponen Shirt, properti ini disimpan di dalam objek props. Untuk mentransfer properti ini, kita perlu mengakses properti ini secara eksplisit dari objek props dan mencantumkannya sebagai bagian dari pemanggilan komponen. Berikut ini adalah contoh tampilannya saat komponen Shirt memanggil komponen Label:

```
var Shirt = React.createClass({
  render: function () {
      return (
         <div>
            <Label color = {this.props.color}
                  num={this.props.num}
                  size={this.props.size}/>

      </div>
      );
    }
});
```

Perhatikan bahwa properti warna, jumlah, dan ukuran dicantumkan lagi. Satu-satunya perbedaan dari apa yang kita lihat dengan panggilan ReactDOM.render adalah bahwa nilai untuk setiap properti diambil dari entri masing-masing dalam objek props, bukan dimasukkan secara manual. Saat komponen Label aktif, objek props-nya diisi dengan benar dengan properti warna, jumlah, dan ukuran yang tersimpan. Anda mungkin dapat melihat pola yang terbentuk di sini. Jika Anda perlu menguap lebar, silakan saja. Komponen Label melanjutkan tradisi dengan mengulangi langkah yang sama dan memanggil komponen Display:

```
var Label = React.createClass({
  render: function () {
      return (
         <Display color={this.props.color}
                  num={this.props.num}
                  size={this.props.size}/>
      );
    }
});
```

Fiuh. Yang ingin kami lakukan hanyalah membuat komponen Display menampilkan beberapa nilai untuk warna, num, dan ukuran. Satu-satunya komplikasi adalah bahwa nilai yang ingin kami tampilkan awalnya ditetapkan sebagai bagian dari ReactDOM.render. Solusi yang menyebalkan adalah yang Anda lihat di sini di mana setiap komponen di sepanjang jalur ke tujuan perlu mengakses dan menetapkan ulang setiap properti sebagai bagian dari penerusannya. Itu sungguh buruk. Kami dapat melakukan yang lebih baik dari ini, dan Anda akan melihatnya dalam beberapa saat!

6.2 MENGENAL OPERATOR SPREAD

Solusi untuk semua masalah kita terletak pada sesuatu yang baru dalam JavaScript yang dikenal sebagai operator spread. Apa yang dilakukan operator spread agak aneh untuk dijelaskan tanpa konteks, jadi saya akan memberi Anda contoh terlebih dahulu dan kemudian membuat Anda bosan dengan definisinya. Lihatlah cuplikan berikut ini:

```
var items = ["1", "2", "3"];

function printStuff(a, b, c) {
  console.log("Printing: " + a + " " + b + " " + c);
}
```

Kita memiliki array yang disebut items yang berisi tiga nilai. Kita juga memiliki fungsi yang disebut printStuff yang mengambil tiga argumen. Yang ingin kita lakukan adalah menentukan tiga nilai dari array items kita sebagai argumen untuk fungsi printStuff. Kedengarannya cukup sederhana, bukan? Berikut ini adalah salah satu cara yang sangat umum untuk melakukannya:

```
printStuff(items[0], items[1], items[2]);
```

Kita mengakses setiap item array secara individual dan meneruskannya ke fungsi printStuff kita. Dengan operator spread, kita sekarang memiliki cara yang lebih mudah. Anda tidak perlu menentukan setiap item dalam array secara individual sama sekali. Anda dapat melakukan sesuatu seperti ini:

```
printStuff(...items);
```

Operator spread adalah karakter ... sebelum array items kita, dan menggunakan ...items identik dengan mencantumkan items[0], items[1], dan items[2] secara individual seperti yang kita lakukan sebelumnya. Fungsi printStuff akan berjalan dan mencetak angka 1, 2, dan 3 ke konsol kita. Cukup keren, bukan? Sekarang setelah Anda melihat operator spread beraksi, saatnya untuk mendefinisikannya. Operator spread memungkinkan Anda untuk mengurai array menjadi elemen-elemen individualnya. Operator spread juga melakukan beberapa hal lainnya, tetapi itu tidak penting untuk saat ini. Kita hanya akan menggunakan sisi operator spread ini untuk menyelesaikan masalah pemindahan properti kita!

6.3 MENTRANSFER PROPERTI DENGAN BENAR

Kita baru saja melihat contoh di mana kita menggunakan operator spread untuk menghindari keharusan menghitung setiap item dalam array kita sebagai bagian dari penerusannya ke suatu fungsi:

```
var items = ["1", "2", "3"];

function printStuff(a, b, c) {
  console.log("Printing: " + a + " " + b + " " + c);
```

```
}

// using the spread operator
printStuff(...items);

// without using the spread operator
printStuff(items[0], items[1], items[2]);
```

Situasi yang kita hadapi saat mentransfer properti antar komponen sangat mirip dengan masalah kita dalam mengakses setiap item array secara individual. Izinkan saya menjelaskannya lebih lanjut. Di dalam suatu komponen, objek properti kita terlihat seperti berikut:

```
var props = {
  color: "steelblue",
  num: "3.14",
  size: "medium"
}
```

Sebagai bagian dari penyampaian nilai properti ini ke komponen anak, kita mengakses setiap item dari objek props kita secara manual:

```
<Display color={this.props.color}
  num={this.props.num}
  size={this.props.size}/>
```

Bukankah akan lebih hebat jika ada cara untuk membongkar objek dan meneruskan pasangan properti/nilai seperti kita dapat membongkar array menggunakan operator spread? Ternyata, ada caranya. Cara ini juga melibatkan operator spread. Saya akan menjelaskannya nanti, tetapi artinya kita dapat memanggil komponen Display dengan menggunakan ...props:

```
<Display {...props}/>
```

Dengan menggunakan ...props, perilaku saat dijalankan sama seperti menentukan properti warna, num, dan ukuran secara manual. Ini berarti contoh sebelumnya dapat disederhanakan sebagai berikut (perhatikan baris yang disorot):

```
var Display = React.createClass({
  render: function() {
    return(
      <div>
        <p>{this.props.color}</p>
        <p>{this.props.num}</p>
        <p>{this.props.size}</p>
      </div>
    );
```

```
  }
});

var Label = React.createClass({
  render: function() {
    return (
      <Display {...this.props}/>
    );
  }
});

var Shirt = React.createClass({
  render: function() {
    return (
      <div>
        <Label {...this.props}/>
      </div>
    );
    }
  });

ReactDOM.render(
  <div>
    <Shirt color="steelblue" num="3.14" size="medium"/>
  </div>,
  document.querySelector("#container")
);
```

Jika Anda menjalankan kode ini, hasil akhirnya tidak akan berubah dari yang kita miliki sebelumnya. Perbedaan terbesarnya adalah kita tidak lagi meneruskan bentuk yang diperluas dari setiap properti sebagai bagian dari pemanggilan setiap komponen. Ini menyelesaikan semua masalah yang awalnya ingin kita selesaikan. Dengan menggunakan operator penyebaran, jika Anda memutuskan untuk menambahkan properti, mengganti nama properti, menghapus properti, atau melakukan berbagai hal lain yang berhubungan dengan properti, Anda tidak perlu membuat satu miliar perubahan yang berbeda. Anda membuat satu perubahan di tempat Anda mendefinisikan properti Anda. Anda membuat perubahan lain di tempat Anda menggunakan properti. Itu saja. Semua komponen perantara yang hanya mentransfer properti akan tetap tidak tersentuh, karena ekspresi {...this.props} tidak berisi detail apa pun yang terjadi di dalamnya.

6.4 KESIMPULAN

Sesuai yang dirancang oleh komite ES6/ES2015, operator penyebaran dirancang untuk bekerja hanya pada array dan makhluk seperti array (alias yang memiliki properti Symbol.iterator). Fakta bahwa ia berfungsi pada objek literal seperti objek props kita disebabkan oleh React yang memperluas standar tersebut. Sampai saat ini, tidak ada browser yang mendukung penggunaan objek spread pada objek literal. Alasan contoh kita berfungsi adalah karena Babel. Selain mengubah semua JSX kita menjadi sesuatu yang dipahami browser

kita, Babel juga mengubah fitur mutakhir dan eksperimental menjadi sesuatu yang ramah lintas-browser. Itulah sebabnya kita dapat menggunakan operator spread pada objek literal, dan itulah sebabnya kita dapat memecahkan masalah pemindahan properti di beberapa lapisan komponen dengan elegan.

BAB 7
MEMAHAMI JSX

Seperti yang mungkin Anda perhatikan sekarang, kita telah menggunakan banyak JSX di bab-bab sebelumnya. Yang benar-benar belum kita lakukan adalah mencermati dengan saksama apa sebenarnya JSX itu. Bagaimana cara kerjanya? Mengapa kita tidak menyebutnya HTML saja? Keunikan apa yang dimilikinya? Di bab ini, kita menjawab semua pertanyaan itu dan banyak lagi! Kami melakukan penelusuran balik (dan penelusuran maju!) yang serius untuk melihat lebih dalam apa yang perlu kami ketahui tentang JSX agar menjadi berbahaya.

7.1 APA YANG TERJADI DENGAN JSX?

Salah satu hal terbesar yang kami abaikan adalah mencoba mencari tahu apa yang terjadi dengan JSX kami setelah kami menulisnya. Bagaimana hasilnya menjadi HTML yang kami lihat di browser kami? Lihat contoh berikut di mana kami mendefinisikan komponen yang disebut Card:

```
var Card = React.createClass({
  render: function() {
    var cardStyle = {
      height: 200,
      width: 150,
      padding: 0,
      backgroundColor: "#FFF",
      WebkitFilter: "drop-shadow(0px 0px 5px #666)",
      filter: "drop-shadow(0px 0px 5px #666)"
    };

    return (
      <div style={cardStyle}>
        <Square color={this.props.color}/>
        <Label color={this.props.color}/>
      </div>
      );
    }
});
```

Kita dapat dengan cepat mengenali JSX di sini. Berikut adalah empat barisnya:

```
<div style={cardStyle}>
  <Square color={this.props.color}/>
  <Label color={this.props.color}/>
</div>
```

Hal yang perlu diingat adalah bahwa peramban kita tidak tahu apa yang harus dilakukan

dengan JSX. Mereka mungkin menganggap Anda gila jika Anda mencoba menjelaskan JSX kepada mereka. Itulah sebabnya kami mengandalkan hal-hal seperti Babel untuk mengubah JSX tersebut menjadi sesuatu yang dipahami peramban: JavaScript. Artinya, JSX yang kami tulis hanya untuk mata manusia (dan kucing yang terlatih). Ketika JSX ini mencapai peramban kami, ia akhirnya diubah menjadi JavaScript murni:

```
return React.createElement(
  "div",
  { style: cardStyle },
  React.createElement(Square, { color: this.props.color }),
  React.createElement(Label, { color: this.props.color })
);
```

Semua elemen HTML yang tersusun rapi, atributnya, dan turunannya semuanya diubah menjadi serangkaian panggilan createElement dengan nilai inisialisasi default. Berikut tampilan keseluruhan komponen Card kita saat diubah menjadi JavaScript:

```
var Card = React.createClass({
  displayName: "Card",

  render: function render() {
    var cardStyle = {
      height: 200,
      width: 150,
      padding: 0,
      backgroundColor: "#FFF",
      WebkitFilter: "drop-shadow(0px 0px 5px #666)",
      filter: "drop-shadow(0px 0px 5px #666)"
    };

  return React.createElement(
    "div",
    { style: cardStyle },
    React.createElement(Square, { color: this.props.color }),
    React.createElement(Label, { color: this.props.color })
    );
  }
});
```

Perhatikan bahwa tidak ada jejak JSX di mana pun! Semua perubahan antara apa yang Anda tulis dan apa yang dilihat peramban kita adalah bagian dari langkah transpilasi yang telah kita bahas di bab pertama. Transpilasi itu adalah sesuatu yang terjadi sepenuhnya di balik layar berkat Babel, yang telah kita gunakan untuk melakukan transformasi JSX→JS ini sepenuhnya di peramban. Kita akhirnya akan melihat penggunaan Babel sebagai bagian dari lingkungan pembangunan yang lebih rumit di mana kita hanya akan menghasilkan berkas JS yang telah diubah, tetapi lebih lanjut tentang itu nanti di masa mendatang. Ya, begitulah. Jawaban atas apa yang sebenarnya terjadi pada semua JSX kita. Itu berubah menjadi JavaScript yang MANIS.

7.2 ATURAN DAN HAL UNIK DALAM JSX

Saat kita bekerja dengan JSX, Anda mungkin memperhatikan bahwa kita menemukan beberapa aturan dan pengecualian yang berubah-ubah terhadap apa yang dapat dan tidak dapat Anda lakukan. Di bagian ini, mari kita gabungkan semua keanehan itu di satu area dan mungkin bahkan menemukan beberapa yang baru.

Anda Hanya Dapat Mengembalikan Satu Node Root

Ini mungkin keanehan pertama yang kami temukan. Di JSX, apa yang Anda kembalikan atau render tidak dapat terdiri dari beberapa elemen root:

```
ReactDOM.render(
   <Letter>B</Letter>
   <Letter>E</Letter>
   <Letter>I</Letter>
   <Letter>O</Letter>
   <Letter>U</Letter>,
   document.querySelector("#container")
);
```

Jika Anda ingin melakukan sesuatu seperti ini, Anda perlu membungkus semua elemen Anda menjadi satu elemen induk terlebih dahulu:

```
ReactDOM.render(
   <div>
      <Letter>A</Letter>
      <Letter>E</Letter>
      <Letter>I</Letter>
      <Letter>O</Letter>
      <Letter>U</Letter>
   </div>,
   document.querySelector("#container")
);
```

Ini tampak seperti persyaratan yang aneh ketika kita melihatnya sebelumnya, tetapi Anda dapat menyalahkan createElement atas alasan kami melakukan ini. Dengan fungsi render dan return, yang pada akhirnya Anda kembalikan adalah satu panggilan createElement (yang pada gilirannya mungkin memiliki banyak panggilan createElement bersarang). Berikut adalah tampilan JSX sebelumnya ketika diubah menjadi JavaScript:

```
ReactDOM.render(React.createElement(
   "div",
   null,
   React.createElement(
      Letter,
      null,
      "A"
   ),
   React.createElement(
```

```
      Letter,
      null,
      "E"
    ),
    React.createElement(
      Letter,
      null,
      "I"
    ),
    React.createElement(
      Letter,
      null,
      "O"
    ),
    React.createElement(
      Letter,
      null,
      "U"
    )
  ), document.querySelector("#container"));
```

Memiliki beberapa elemen root akan merusak cara fungsi mengembalikan nilai dan cara kerja createElement, jadi itulah mengapa Anda hanya dapat menentukan satu elemen return (root)! Sekarang Anda dapat tidur lebih nyenyak dengan mengetahui hal ini.

Anda Tidak Dapat Menentukan CSS Secara Inline

Seperti yang kita lihat di Bab 4, atribut style di JSX Anda berperilaku berbeda dari atribut style di HTML. Di HTML, Anda dapat menentukan properti CSS secara langsung sebagai nilai pada atribut style Anda:

```
<div style="font-family:Arial;font-size:24px">
  <p>Blah!</p>
</div>
```

Di JSX, atribut style tidak dapat memuat CSS di dalamnya. Sebaliknya, atribut tersebut harus merujuk ke objek yang memuat informasi gaya:

```
var Letter = React.createClass({
  render: function() {
    var letterStyle = {
    padding: 10,
    margin: 10,
    backgroundColor: this.props.bgcolor,
    color: "#333",
    display: "inline-block",
    fontFamily: "monospace",
    fontSize: "32",
    textAlign: "center"
  };
```

```
    return (
      <div style={letterStyle}>
        {this.props.children}
      </div>
    );
    }
});
```

Perhatikan bahwa kita memiliki objek bernama letterStyle yang berisi semua properti CSS (dalam bentuk JavaScript camelcase) dan nilainya. Objek itulah yang kemudian kita tentukan pada atribut style.

Kata Kunci Cadangan dan className

JavaScript memiliki banyak kata kunci dan nilai yang tidak dapat Anda gunakan sebagai pengenal. Kata kunci tersebut saat ini (sejak ES2016) adalah:

break	case	class	catch	const	continue
debugger	default	delete	do	else	export
extends	finally	for	funnction	if	impoty
in	instanceof	new	return	super	switch
this	throw	try	typeof	var	void
while	with	yield			

Saat Anda menulis JSX, Anda harus berhati-hati untuk tidak menggunakan kata kunci ini sebagai bagian dari pengenal apa pun yang Anda buat. Itu bisa jadi sulit ketika kata kunci tertentu yang sangat populer seperti class umumnya digunakan dalam HTML meskipun juga ada dalam daftar kata kunci khusus JavaScript.

Perhatikan hal berikut:

```
ReactDOM.render(
  <div class="slideIn">
    <p class="emphasis">Gabagool!</p>
    <Label/>
    </div>,
  document.querySelector("#container")
);
```

Mengabaikan reservasi JavaScript tentang kelas (seperti yang telah kita lakukan di sini) tidak akan berhasil. Yang perlu Anda lakukan adalah menggunakan versi DOM API dari atribut kelas yang disebut className sebagai gantinya:

```
ReactDOM.render(
  <div className="slideIn">
    <p className="emphasis">Gabagool!</p>
    <Label/>
  </div>,
  document.querySelector("#container")
);
```

Anda dapat melihat daftar lengkap tag dan atribut yang didukung di artikel Facebook berikut (https://facebook.github.io/react/docs/tags-and-attributes.html), dan perhatikan bahwa semua atribut menggunakan huruf kecil. Detail itu penting, karena menggunakan versi huruf kecil dari suatu atribut tidak akan berfungsi. Jika Anda menempelkan potongan besar HTML yang ingin Anda tangani oleh JSX, pastikan untuk kembali ke HTML yang Anda tempel dan buat penyesuaian kecil ini untuk mengubahnya menjadi JSX yang valid.

Ini memunculkan poin lain. Karena penyimpangan kecil dari perilaku HTML ini, kita cenderung mengatakan bahwa JSX mendukung sintaksis seperti HTML, bukan hanya mengatakan bahwa JSX mendukung HTML. Ini adalah React-isme yang disengaja. Jawaban paling jelas untuk hubungan antara JSX dan HTML datang dari anggota tim React, Ben Alpert, yang menyatakan hal berikut (http://qr.ae/RUKaON) sebagai bagian dari jawaban Quora:

...menurut kami, keunggulan utama JSX adalah simetri tag penutup yang cocok yang membuat kode lebih mudah dibaca, bukan kemiripan langsung dengan HTML atau XML. Mudah untuk menyalin/menempel HTML secara langsung, tetapi perbedaan kecil lainnya (dalam tag penutup sendiri, misalnya) membuat ini menjadi pertarungan yang tidak akan menguntungkan dan kami memiliki konverter HTML ke JSX untuk membantu Anda. Terakhir, untuk menerjemahkan HTML ke kode React yang idiomatis, biasanya diperlukan cukup banyak pekerjaan untuk memecah markup menjadi komponen yang masuk akal, jadi mengubah class menjadi className hanyalah sebagian kecil dari itu.

Komentar

Sama seperti ide yang bagus untuk mengomentari HTML, CSS, dan JavaScript Anda, ide yang bagus untuk memberikan komentar di dalam JSX Anda juga. Menentukan komentar di JSX sangat mirip dengan cara Anda memberi komentar di JavaScript (https://www.kirupa.com/html5/comments.htm) ...kecuali untuk satu pengecualian. Jika Anda menentukan komentar sebagai anak dari sebuah tag, Anda perlu membungkus komentar Anda dengan tanda kurung kurawal {dan} untuk memastikan komentar tersebut diurai sebagai sebuah ekspresi:

```
ReactDOM.render(
  <div class="slideIn">
    <p class="emphasis">Gabagool!</p>
      {/* I am a child comment */}
    <Label/>
  </div>,
  document.querySelector("#container")
);
```

Komentar kita dalam kasus ini adalah turunan dari elemen div kita. Jika Anda menentukan komentar sepenuhnya di dalam tag, Anda dapat menentukan komentar satu baris atau lebih tanpa harus menggunakan tanda kurung siku { dan }:

```
ReactDOM.render(
  <div class="slideIn">
    <p class="emphasis">Gabagool!</p>
```

```
      <Label
        /* This comment
        goes across
        multiple lines */
        className="colorCard" // end of line
      />
    </div>,
    document.querySelector("#container")
);
```

Dalam cuplikan ini, Anda dapat melihat contoh seperti apa komentar multi-baris dan komentar di akhir baris. Sekarang setelah Anda mengetahui semua ini, Anda memiliki satu alasan lebih sedikit untuk tidak mengomentari JSX Anda :P

7.3 KAPITALISASI, ELEMEN HTML, DAN KOMPONEN

Kapitalisasi itu penting. Untuk merepresentasikan elemen HTML, pastikan tag HTML menggunakan huruf kecil:

```
ReactDOM.render(
  <div>
    <section>
      <p>Something goes here!</p>
    </section>
  </div>,
  document.querySelector("#container")
);
```

Saat ingin merepresentasikan komponen, nama komponen harus diawali dengan huruf kapital, baik di JSX maupun saat Anda mendefinisikannya:

```
ReactDOM.render(
  <div>
    <MyCustomComponent/>
  </div>,
document.querySelector("#container")
);
```

Jika Anda salah dalam penggunaan huruf kapital, React tidak akan menampilkan konten Anda dengan benar. Komponen tersebut tidak akan ditemukan. Mencoba mengidentifikasi masalah penggunaan huruf kapital mungkin adalah hal terakhir yang akan Anda pikirkan ketika sesuatu tidak berjalan dengan baik, jadi ingatlah kiat kecil ini.

JSX Anda Bisa Berada di Mana Saja

Dalam banyak situasi, JSX Anda tidak akan tersusun rapi di dalam fungsi render atau return seperti pada contoh yang telah kita lihat sejauh ini. Lihatlah contoh berikut:

```
var swatchComponent = <Swatch color="#2F004F"></Swatch>;
```

```
ReactDOM.render(
   <div>
      {swatchComponent}
   </div>,
document.querySelector("#container")
);
```

Kita memiliki variabel yang disebut swatchComponent yang diinisialisasi ke baris JSX. Ketika variabel swatchComponent kita ditempatkan di dalam fungsi render, komponen Swatch kita akan diinisialisasi. Semua ini benar-benar valid, dan kita akan melakukan lebih banyak hal seperti itu di masa mendatang ketika kita mempelajari cara membuat dan memanipulasi JSX menggunakan JavaScript.

7.5 KESIMPULAN

Dengan bab ini, kita akhirnya menyatukan berbagai bit informasi JSX yang diperkenalkan pada bab-bab sebelumnya di satu lokasi. Hal terpenting yang perlu diingat adalah bahwa JSX bukanlah HTML. Ia tampak seperti HTML dan berperilaku seperti itu dalam banyak skenario umum, tetapi pada akhirnya ia dirancang untuk diterjemahkan ke dalam JavaScript. Ini berarti Anda dapat melakukan hal-hal yang tidak pernah Anda bayangkan dapat dilakukan hanya dengan menggunakan HTML biasa. Mampu mengevaluasi ekspresi atau memanipulasi seluruh potongan JSX secara terprogram hanyalah permulaan. Dalam bab-bab mendatang, kita akan mengeksplorasi lebih jauh persimpangan JavaScript dan JSX ini.

BAB 8
MEMAHAMI STATE DALAM KOMPONEN

Sampai saat ini, komponen yang telah kita buat tidak memiliki state. Komponen tersebut memiliki properti (alias props) yang diberikan dari induknya, tetapi (biasanya) tidak ada yang berubah setelah komponen tersebut aktif. Properti Anda dianggap tidak dapat diubah setelah ditetapkan. Untuk banyak skenario interaktif, Anda tidak menginginkannya. Anda ingin dapat mengubah aspek komponen Anda sebagai hasil dari beberapa interaksi pengguna (atau beberapa data yang dikembalikan dari server atau miliaran hal lainnya!)

Yang kita butuhkan adalah cara lain untuk menyimpan data pada komponen yang melampaui properti. Kita membutuhkan cara untuk menyimpan data yang dapat diubah. Yang kita butuhkan adalah sesuatu yang dikenal sebagai state! Dalam bab ini, Anda akan mempelajari semua tentangnya dan cara menggunakannya untuk membuat komponen stateful.

8.1 MENGGUNAKAN STATE

Jika Anda tahu cara bekerja dengan properti, Anda benar-benar tahu cara bekerja dengan state kurang lebih begitu. Ada beberapa perbedaan, tetapi terlalu halus untuk membuat Anda bosan sekarang. Sebagai gantinya, mari langsung saja dan lihat state beraksi dengan menggunakannya dalam contoh kecil. Yang akan kita buat adalah contoh penangkal petir sederhana seperti ditunjukkan pada Gambar 8.1.

Gambar 8.1 Aplikasi Yang Akan Anda Buat

Contoh ini tidak terlalu aneh. Petir menyambar permukaan bumi sekitar 100 kali per detik Kita memiliki penghitung yang hanya menambah angka yang Anda lihat dengan jumlah yang sama. Mari kita buat penghitung tersebut.

Titik Awal Kita

Fokus utama contoh ini adalah untuk melihat bagaimana kita dapat bekerja dengan status. Tidak ada gunanya kita menghabiskan banyak waktu untuk membuat contoh dari awal dan menelusuri kembali jalur yang telah kita lalui berkali-kali. Itu bukanlah cara terbaik untuk memanfaatkan waktu seseorang. Daripada memulai dari awal, ubah dokumen HTML yang ada atau buat yang baru dengan konten berikut:

```
<!DOCTYPE html>
<html>

<head>
  <title>More State!</title>
  <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
  <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
  core/5.8.23/browser.min.js"></script>
</head>

  <body>
    <div id="container"></div>
    <script type="text/babel">
      var LightningCounter = React.createClass({
        render: function() {
          return (
            <h1>Hello!</h1>
          );
        }
      });

  var LightningCounterDisplay = React.createClass({
      render: function() {

       var divStyle = {
          width: 250,
          textAlign: "center",
          backgroundColor: "black",
          padding: 40,
          fontFamily: "sans-serif",
          color: "#999",
          borderRadius: 10
      };

  return(
    <div style={divStyle}>
      <LightningCounter/>
    </div>
    );
  }
});
```

```
    ReactDOM.render(
      <LightningCounterDisplay/>,
      document.querySelector("#container")
    );
  </script>
</body>

</html>
```

Pada titik ini, luangkan waktu sejenak untuk melihat apa yang dilakukan kode kita saat ini. Pertama, kita memiliki komponen yang disebut LightningCounterDisplay:

```
var LightningCounterDisplay = React.createClass({
  render: function() {

    var divStyle = {
      width: 250,
      textAlign: "center",
      backgroundColor: "black",
      padding: 40,
      fontFamily: "sans-serif",
      color: "#999",
      borderRadius: 10
    };

    return(
      <div style={divStyle}>
        <LightningCounter/>
      </div>
    );
  }
});
```

Bagian utama dari komponen ini adalah objek divStyle yang berisi informasi gaya yang bertanggung jawab atas latar belakang bulat yang keren. Fungsi render mengembalikan elemen div yang membungkus komponen LightningCounter. Komponen LightningCounter adalah tempat semua tindakan akan berlangsung:

```
var LightningCounter = React.createClass({
  render: function() {
    return (
      <h1>Hello!</h1>
    );
  }
});
```

Komponen ini, seperti yang ada saat ini, tidak memiliki hal menarik apa pun. Komponen ini hanya mengembalikan kata Hello! Tidak apa-apa—kita akan memperbaiki komponen ini nanti.

Hal terakhir yang perlu diperhatikan adalah metode ReactDOM. render kita:

```
ReactDOM.render(
  <LightningCounterDisplay/>,
  document.querySelector("#container")
);
```

Komponen LightningCounterDisplay hanya akan dimasukkan ke dalam elemen div container di DOM kita. Itu saja. Hasil akhirnya adalah Anda akan melihat kombinasi markup dari metode ReactDOM.render dan komponen LightningCounterDisplay dan LightningCounter.

8.2 MENYALAKAN PENGHITUNG

Sekarang setelah kita memiliki gambaran tentang apa yang akan kita mulai, saatnya membuat rencana untuk langkah selanjutnya. Cara kerja penghitung kita cukup sederhana. Kita akan menggunakan fungsi setInterval yang memanggil beberapa kode setiap 1000 milidetik (alias 1 detik). "Beberapa kode" itu akan menambah nilai sebesar 100 setiap kali dipanggil. Tampaknya cukup mudah, bukan? Agar semuanya berfungsi, kita akan mengandalkan tiga API yang diekspos Komponen React kita:

- GetInitialState Metode ini berjalan tepat sebelum komponen Anda dipasang, dan memungkinkan Anda untuk mengubah objek status komponen.
- ComponentDidMount Metode ini dipanggil tepat setelah komponen kita dirender (atau dipasang sebagaimana React menyebutnya).
- SetState Metode ini memungkinkan Anda memperbarui nilai objek status.

Kita akan segera melihat API ini digunakan, tetapi saya ingin memberi Anda pratinjaunya sehingga Anda dapat menemukannya dengan mudah dalam daftar!

Menetapkan Nilai Status Awal

Kita memerlukan variabel untuk bertindak sebagai penghitung, dan mari kita sebut variabel ini strikes. Ada banyak cara untuk membuat variabel ini. Yang paling jelas adalah sebagai berikut:

```
var strikes = 0 // :P
```

Namun, kita tidak ingin melakukan itu. Untuk contoh kita, variabel strikes adalah bagian dari status komponen kita, dan nilainya adalah apa yang kita tampilkan di layar. Yang akan kita lakukan adalah menggunakan metode getInitialState yang kita lihat sekilas beberapa saat yang lalu dan mengurus inisialisasi variabel kita di dalamnya. Anda akan melihat dalam beberapa saat apa hasilnya pada status komponen kita. Di dalam komponen LightningCounter Anda, tambahkan baris yang disorot berikut ini:

```
var LightningCounter = React.createClass({
    getInitialState: function() {
          return {
```

```
            strikes: 0
        };
    },
    render: function () {
        return(
        <hl>{this.state.strikes}</hl>
        );
      }
});
```

Metode getInitialState berjalan secara otomatis jauh sebelum komponen Anda direnderr, dan yang kami lakukan adalah memberi tahu React untuk mengembalikan objek yang berisi properti strikes kami (diinisialisasi ke 0). Anda mungkin bertanya-tanya kepada siapa atau apa kami mengembalikan objek ini? Semua itu adalah keajaiban yang terjadi di balik layar. Objek yang dikembalikan ditetapkan sebagai nilai awal untuk objek status komponen kami. Jika kami memeriksa nilai objek status kami setelah kode ini dijalankan, akan terlihat seperti berikut:

```
var state = {
  strikes: 0
}
```

Sebelum kita mengakhiri bagian ini, mari kita visualisasikan properti strikes kita. Dalam metode render kita, buat perubahan yang disorot berikut ini:

```
var LightningCounter = React.createClass({
    getInitialState: function() {
        return {
          strikes: 0
    };
},
render: function () {
    return (
        <hl>{this.state.strikes}</hl>
        );
     }
});
```

Yang telah kita lakukan adalah mengganti teks Hello! default kita dengan ekspresi yang menampilkan nilai yang disimpan oleh properti this.state.strikes. Jika Anda melihat contoh Anda di browser, Anda akan melihat nilai 0 ditampilkan. Itu adalah permulaan!

8.3 MEMULAI TIMER DAN MENETAPKAN STATUS

Berikutnya adalah menjalankan timer dan menambah properti strikes kita. Seperti yang telah kita sebutkan sebelumnya, kita akan menggunakan fungsi setInterval untuk menambah properti strikes sebesar 100 setiap detik. Kita akan melakukan semua ini segera setelah komponen kita direnderr menggunakan metode componentDidMount bawaan. Kode untuk

memulai timer kita terlihat sebagai berikut:

```
var LightningCounter = React.createClass({
  getInitialState: function() {
    return {
      strikes: 0
    };
  },

  componentDidMount: function() {
    setInterval(this.timerTick, 1000);
  },
  render: function() {
    return (
      <h1>{this.state.strikes}</h1>
    );
  }
});
```

Silakan tambahkan baris-baris yang disorot ini ke contoh kita. Di dalam metode componentDidMount yang dipanggil sekali, komponen kita akan dirender, kita memiliki metode setInterval yang memanggil fungsi timerTick setiap detik (atau 1000 milidetik). Kita belum mendefinisikan fungsi timerTick, jadi mari kita perbaiki dengan menambahkan baris-baris yang disorot berikut ke kode kita:

```
var LightningCounter = React.createClass({
  getInitialState: function() {
    return {
      strikes: 0
    };
  },
  timerTick: function() {
    this.setState({
      strikes: this.state.strikes + 100
    });
  },
  componentDidMount: function() {
    setInterval(this.timerTick, 1000);
  },
  render: function() {
    return (
      <h1>{this.state.strikes}</h1>
    );
  }
});
```

Fungsi timerTick kita cukup sederhana. Fungsi ini hanya memanggil setState. Metode setState hadir dalam berbagai bentuk, tetapi untuk apa yang kita lakukan di sini, fungsi ini hanya mengambil sebuah objek sebagai argumennya. Objek ini berisi semua properti yang ingin Anda

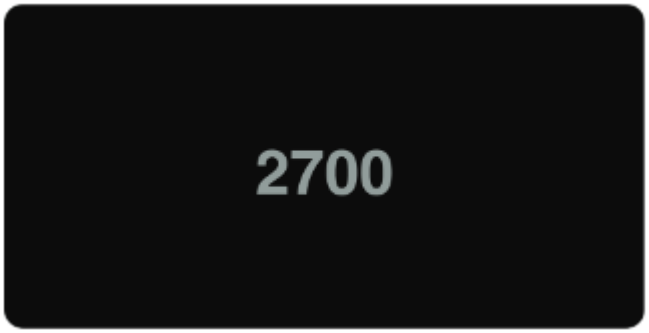
gabungkan ke dalam objek state. Dalam kasus kita, kita menentukan properti strikes dan menetapkan nilainya menjadi 100 lebih banyak dari nilai saat ini.

Bagaimana timerTick mempertahankan konteks?

Dalam JavaScript biasa, fungsi timerTick tidak akan mempertahankan konteks. Anda harus melakukan pekerjaan ekstra untuk mendukungnya. Alasan mengapa fungsi ini berfungsi di dunia React adalah karena sesuatu yang dikenal sebagai autobinding. Nah, senangkah Anda mengetahui hal itu?

Merender Perubahan Status

Jika Anda melihat pratinjau aplikasi Anda sekarang, Anda akan melihat nilai strikes kita mulai bertambah 100 setiap detik (lihat Gambar 8.2).



Gambar 8.2 Nilai Strikes Bertambah 100 Setiap Detik

Mari kita abaikan sejenak apa yang terjadi dengan kode kita. Itu cukup mudah. Hal yang menarik adalah bagaimana semua yang telah kita lakukan berakhir dengan memperbarui apa yang Anda lihat di layar. Pembaruan itu berkaitan dengan perilaku React ini: Setiap kali Anda memanggil setState dan memperbarui sesuatu dalam objek status, metode render komponen Anda akan otomatis dipanggil. Ini memulai serangkaian panggilan render untuk setiap komponen yang outputnya juga terpengaruh. Hasil akhir dari semua ini adalah apa yang Anda lihat di layar Anda dalam representasi terbaru dari status UI aplikasi Anda. Menjaga data dan UI Anda tetap sinkron adalah salah satu masalah tersulit dalam pengembangan UI, jadi senang sekali React mengurusinya untuk kita. Itu membuat semua kesulitan belajar menggunakan React ini benar-benar sepadan hampir! :P

Opsional: Kode Lengkap

Yang kita miliki saat ini hanyalah penghitung yang bertambah 100 setiap detik. Tidak ada yang menunjukkan Penghitung Petir, tetapi mencakup semua hal tentang status yang ingin saya sampaikan kepada Anda saat ini. Jika Anda ingin secara opsional menyempurnakan contoh Anda agar terlihat seperti versi saya yang Anda lihat di awal, berikut adalah kode lengkap untuk apa yang ada di dalam tag skrip kita:

```
var LightningCounter = React.createClass({
  getInitialState: function() {
    return {
      strikes: 0
```

```
    };
  },
  timerTick: function() {
    this.setState({
      strikes: this.state.strikes + 100
    });
  },

  componentDidMount: function() {
      setInterval(this.timerTick, 1000);
  },
  render: function() {
    var counterStyle = {
      color: "#66FFFF",
      fontSize: 50
    };

    var count = this.state.strikes.toLocaleString();

    return (
      <h1 style={counterStyle}>{count}</h1>
    );
  }
});

var LightningCounterDisplay = React.createClass({
  render: function() {
    var commonStyle = {
      margin: 0,
      padding: 0
    }
    var divStyle = {
      width: 250,
      textAlign: "center",
      backgroundColor: "#020202",
      padding: 40,
      fontFamily: "sans-serif",
      color: "#999999",
      borderRadius: 10
    };
    var textStyles = {
      emphasis: {
        fontSize: 38,
        ...commonStyle
      },
      smallEmphasis: {
        ...commonStyle
      },
      small: {
        fontSize: 17,
        opacity: 0.5,
        ...commonStyle
```

```
        }
      }
      return(
        <div style={divStyle}>
          <LightningCounter/>
          <h2 style={textStyles.smallEmphasis}>LIGHTNING STRIKES</h2>
          <h2 style={textStyles.emphasis}>WORLDWIDE</h2>
          <p style={textStyles.small}>(since you loaded this example)</p>
        </div>
      );
    }
  });

  ReactDOM.render(
    <LightningCounterDisplay/>,
    document.querySelector("#container")
  );
```

Jika Anda membuat kode Anda terlihat seperti semua yang Anda lihat di atas dan menjalankan contoh lagi, Anda akan melihat contoh penghitung petir kami dalam semua kemegahannya yang berwarna cyan. Saat Anda melakukannya, luangkan waktu sejenak untuk memeriksa kode tersebut guna memastikan Anda tidak melihat terlalu banyak kejutan.

8.4 KESIMPULAN

Kita baru saja mengupas permukaan tentang apa yang dapat kita lakukan untuk membuat komponen stateful. Meskipun menggunakan timer untuk memperbarui sesuatu dalam objek state kita itu keren, tindakan sebenarnya terjadi saat kita mulai menggabungkan interaksi pengguna dengan state. Sejauh ini, kita telah menghindari sejumlah besar mouse, sentuhan, keyboard, dan hal-hal terkait lainnya yang akan bersentuhan dengan komponen Anda. Dalam bab mendatang, kita akan memperbaikinya.

BAB 9
BERALIH DARI DATA KE UI

Saat Anda membuat aplikasi, berpikir dalam hal properti, status, komponen, tag JSX, metode render, dan React-isme lainnya mungkin menjadi hal terakhir yang terlintas dalam pikiran Anda. Sering kali, Anda berurusan dengan data dalam bentuk objek JSON, array, dan struktur data lainnya yang tidak memiliki pengetahuan (atau minat) dalam React atau apa pun yang bersifat visual. Menjembatani jurang antara data Anda dan apa yang akhirnya Anda lihat bisa jadi membuat frustrasi! Namun, jangan khawatir. Bab ini membantu mengurangi beberapa momen yang membuat frustrasi tersebut dengan menjalankan beberapa skenario umum yang akan Anda hadapi!

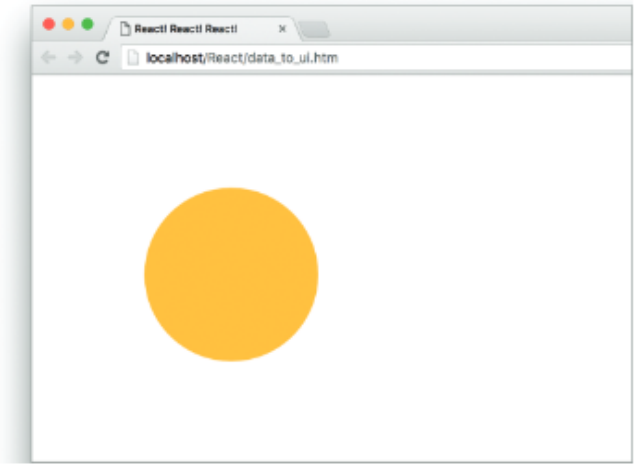
Contoh

Untuk membantu memahami semua yang akan Anda lihat, kita akan memerlukan sebuah contoh. Tidak terlalu rumit, jadi lanjutkan dan buat dokumen HTML baru dan masukkan hal-hal berikut ke dalamnya:

```
<!DOCTYPE html>
<html>

<head>
   <title>React! React! React!</title>
   <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
   <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></script>
   <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min
   .js"></script>

   <style>
   #container {
      padding: 50px;
      background-color: #FFF;
   }
   </style>
</head>
   <body>
      <div id="container"></div>
      <script type="text/babel">
         var Circle = React.createClass({
              render: function() {
              var circleStyle = {
              padding: 10,
              margin: 20,
              display: "inline-block",
              backgroundColor: this.props.bgColor,
              borderRadius: "50%",
              width: 100,
              height: 100,
```

```
            };

              return (
              <div style={circleStyle}>
              </div>
            );
          }
        });

        var destination = document.querySelector("#container");

        ReactDOM.render(
          <div>
            <Circle bgColor="#F9C240"/>
          </div>,
          destination
        );
      </script>
    </body>
    </html>
```

Setelah dokumen Anda siap, lanjutkan dan pratinjau apa yang Anda miliki di browser Anda. Jika semuanya berjalan lancar, Anda akan disambut oleh lingkaran kuning yang ceria (lihat Gambar 9.1).



Gambar 9.1 Jika Semuanya Berjalan Lancar, Anda Akan Mendapatkan Lingkaran Kuning Ini

Jika Anda melihat apa yang saya lihat, bagus! Sekarang, mari luangkan waktu sejenak untuk memahami apa yang dilakukan contoh kita. Sebagian besar dari apa yang Anda lihat berasal dari komponen Lingkaran:

```
var Circle = React.createClass({
  render: function() {
    var circleStyle = {
    padding: 10,
    margin: 20,
```

```
      display: "inline-block",
      backgroundColor: this.props.bgColor,
      borderRadius: "50%",
      width: 100,
      height: 100,
    };

    return (
      <div style={circleStyle}>
      </div>
    );
  }
});
```

Sebagian besar terdiri dari objek circleStyle yang berisi properti gaya sebaris yang mengubah div yang membosankan menjadi lingkaran yang mengagumkan. Semua nilai gaya dikodekan secara kaku kecuali properti backgroundColor. Nilainya diambil dari properti bgColor yang diberikan. Selain komponen, cara kita menampilkan lingkaran adalah melalui metode ReactDOM.render biasa:

```
ReactDOM.render(
  <div>
    <Circle bgColor="#F9C240"/>
  </div>,
  destination
);
```

Kita memiliki satu contoh komponen Circle yang dideklarasikan, dan kita mendeklarasikannya dengan properti bgColor yang ditetapkan ke warna yang kita inginkan untuk lingkaran kita. Sekarang, mendefinisikan komponen Circle sebagaimana adanya di dalam metode render kita agak membatasi - terutama jika Anda akan menangani data yang dapat memengaruhi apa yang dilakukan komponen Circle kita. Dalam beberapa bagian berikutnya, kita akan melihat cara-cara yang kita miliki untuk menyelesaikannya.

JSX Anda Dapat Berada di Mana Saja Bagian II

Dalam bab 7, kita mempelajari bahwa JSX kita sebenarnya dapat berada di luar fungsi render dan dapat digunakan sebagai nilai yang ditetapkan ke variabel atau properti. Misalnya, kita dapat dengan berani melakukan sesuatu seperti ini:

```
var theCircle = <Circle bgColor="#F9C240"/>;

ReactDOM.render(
  <div>
    {theCircle}
  </div>,
  destination
);
```

Variabel theCircle menyimpan JSX untuk membuat komponen Circle kita. Mengevaluasi variabel ini di dalam fungsi ReactDOM.render kita akan menghasilkan lingkaran yang ditampilkan. Hasil akhirnya tidak berbeda dengan yang kita miliki sebelumnya, tetapi dengan membebaskan pembuatan komponen Circle dari belenggu metode render, kita akan memiliki lebih banyak opsi untuk melakukan hal-hal yang gila dan keren. Misalnya, Anda dapat melangkah lebih jauh dan membuat fungsi yang mengembalikan komponen Circle:

```
function showCircle() {
  var colors = ["#393E41", "#E94F37", "#1C89BF", "#A1D363"];
  var ran = Math.floor(Math.random() * colors.length);

  // return a Circle with a randomly chosen color
  return <Circle bgColor={colors[ran]}/>;
};
```

Dalam kasus ini, fungsi showCircle mengembalikan komponen Circle (membosankan!) dengan nilai untuk properti bgColor yang ditetapkan ke nilai warna acak (luar biasa!). Agar contoh kita menggunakan fungsi showCircle, yang harus Anda lakukan adalah mengevaluasinya di dalam ReactDOM.render:

```
1. ReactDom.render (
2. <div>
3.    {showCircle ()}
4. </div>,
5. destination
6. );
```

Selama ekspresi yang Anda evaluasi mengembalikan JSX, Anda dapat meletakkan apa pun yang Anda inginkan di dalam kurung kurawal { dan }. Fleksibilitas itu sangat bagus, karena ada banyak hal yang dapat Anda lakukan saat JavaScript Anda berada di luar fungsi render. BANYAK HAL!

9.1 ARRAY DALAM JSX

Sekarang kita akan membahas beberapa hal yang menyenangkan! Saat Anda menampilkan beberapa komponen, Anda tidak akan selalu dapat menentukannya secara manual:

```
ReactDOM.render(
  <div>
    {showCircle()}
    {showCircle()}
    {showCircle()}
  </div>,
  destination
);
```

Dalam banyak skenario dunia nyata, jumlah komponen yang Anda tampilkan akan terkait dengan jumlah item dalam array atau objek mirip array (alias iterator) yang sedang Anda kerjakan. Hal itu menimbulkan beberapa komplikasi sederhana. Misalnya, katakanlah kita memiliki array yang disebut colors yang terlihat seperti berikut:

```
var colors =    ["#393E41", "#E94F37",  "#1C89BF",  "#A1D363",
                 "#85FFC7",  "#297373",  "#FF8552",  "#A40E4C"];
```

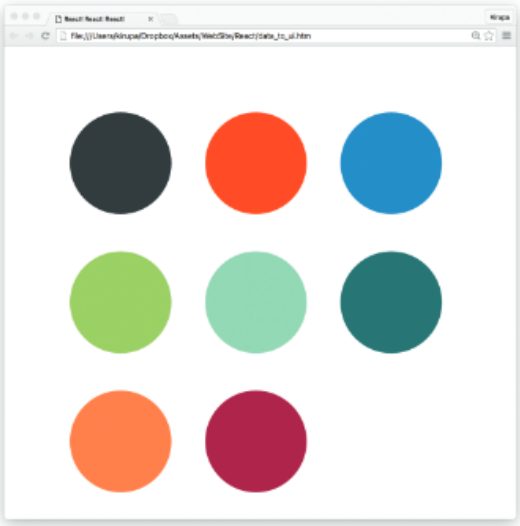
Yang ingin kita lakukan adalah membuat komponen Circle untuk setiap item dalam array ini (dan menetapkan properti bgColor ke nilai setiap item array). Cara kita melakukannya adalah dengan membuat array komponen Circle:

```
var colors =    ["#393E41", "#E94F37",  "#1C89BF",  "#A1D363",
                 "#85FFC7",  "#297373",  "#FF8552",  "#A40E4C"];

var renderData = [];

for (var i = 0; i < colors.length; i++) {
   renderData.push(<Circle bgColor={colors[i]}/>);
}
```

Dalam cuplikan ini, kita mengisi array renderData dengan komponen Circle seperti yang awalnya ingin kita lakukan. Sejauh ini, semuanya berjalan lancar. Untuk menampilkan semua komponen ini, React membuatnya sangat mudah. Lihat baris yang disorot untuk mengetahui semua yang harus Anda lakukan:

```
var colors =    ["#393E41", "#E94F37",  "#1C89BF",  "#A1D363",
                 "#85FFC7",  "#297373",  "#FF8552",  "#A40E4C"];

var renderData = [];

for (var i = 0; i < colors.length; i++) {
    renderData.push(<Circle bgColor={colors[i]}/>);
}

ReactDOM.render(
    <div>
          {renderData}
    </div>,
    destination
);
```

Dalam metode render, yang perlu kita lakukan hanyalah menentukan array renderData sebagai ekspresi yang perlu kita evaluasi. Kita tidak perlu melakukan langkah lain untuk beralih dari array komponen ke tampilan seperti Gambar 9.2 saat Anda melihat pratinjau di browser.



Gambar 9.2 Apa Yang Seharusnya Anda Lihat Di Browser

Meskipun contoh kita tampaknya berhasil, kita belum selesai! Sebenarnya ada satu hal lagi yang perlu kita lakukan, dan ini hal yang tidak kentara. Cara React membuat pembaruan UI sangat cepat adalah dengan memiliki ide bagus tentang apa yang sebenarnya terjadi di DOM Anda. React melakukannya dengan beberapa cara, tetapi satu cara yang benar-benar terlihat adalah dengan menandai setiap elemen secara internal dengan semacam pengenal. "Penandaan" ini terjadi secara otomatis saat Anda secara eksplisit menentukan elemen di JSX Anda.

Saat Anda membuat elemen secara dinamis (seperti yang kita lakukan dengan rangkaian komponen Circle), pengenal ini tidak ditetapkan secara otomatis. Kita perlu melakukan beberapa pekerjaan tambahan. Pekerjaan tambahan itu berbentuk properti key yang nilainya digunakan React untuk mengidentifikasi setiap komponen tertentu secara unik. Untuk contoh kita, kita dapat melakukan sesuatu seperti ini:

```
for (var i = 0; i < colors.length; i++) {
  var color = colors[i];
  renderData.push(<Circle key={i + color} bgColor={color}/>);
}
```

Pada setiap komponen, kami menentukan properti kunci dan menetapkan nilainya ke kombinasi warna dan posisi indeks di dalam array warna. Ini memastikan bahwa setiap komponen yang kami buat secara dinamis akan mendapatkan pengenal unik yang kemudian dapat digunakan React untuk mengoptimalkan pembaruan UI mendatang. Sekarang, kami dapat menggunakan posisi indeks sebagai pengenal, tetapi jika Anda memiliki beberapa blok kode tempat Anda membuat elemen secara dinamis, Anda mungkin mendapatkan beberapa elemen dengan nilai indeks duplikat.

9.2 PERIKSA KONSOL

React sangat bagus dalam memberi tahu Anda saat Anda mungkin melakukan kesalahan. Misalnya, jika Anda membuat elemen atau komponen secara dinamis dan tidak menentukan properti kunci pada elemen atau komponen tersebut, Anda akan disambut dengan peringatan berikut di konsol Anda:

Peringatan: Setiap anak dalam array atau iterator harus memiliki properti "kunci" yang unik. Periksa panggilan render tingkat atas menggunakan . Saat Anda bekerja dengan React, sebaiknya periksa konsol Anda secara berkala untuk mengetahui pesan apa pun yang mungkin ada di dalamnya. Bahkan jika semuanya tampak berjalan dengan baik, Anda tidak akan pernah tahu apa yang mungkin Anda temukan :P

9.3 KESIMPULAN

Semua kiat dan trik yang Anda lihat dalam artikel ini mungkin dibuat karena satu hal: JSX adalah JavaScript. Inilah yang memungkinkan Anda untuk menjalankan JSX di mana pun JavaScript berkembang. Bagi kami, tampaknya kami melakukan sesuatu yang benar-benar aneh ketika kami menentukan sesuatu seperti ini:

```
  for (var i = 0; i < colors.length; i++) {
  var color = colors[i];
  renderData.push(<Circle key={i + color} bgColor={color}/>);
}
```

Meskipun kita memasukkan potongan JSX ke dalam array, seperti sulap, semuanya berfungsi pada akhirnya saat renderData dievaluasi di dalam metode render kita. Saya tidak suka terdengar seperti rekaman rusak, tetapi ini karena apa yang akhirnya dilihat browser kita tampak seperti ini:

```
for (var i = 0; i < colors.length; i++) {
  var color = colors[i];

      renderData.push(React.createElement(Circle,
    {
      key: i + color,
      bgColor: color
    }));
}
```

Saat JSX kita diubah menjadi JS murni, semuanya menjadi masuk akal lagi. Inilah yang memungkinkan kita untuk menempatkan JSX kita dalam berbagai situasi yang tidak nyaman (namun fotogenik!) dengan data kita dan tetap mendapatkan hasil akhir yang kita inginkan! Karena, pada akhirnya, semuanya hanyalah JavaScript.

BAB 10
BEKERJA DENGAN PERISTIWA DI REACT

Sejauh ini, sebagian besar contoh kita hanya berfungsi saat halaman dimuat. Seperti yang mungkin Anda duga, itu tidak normal. Di sebagian besar aplikasi, terutama yang menggunakan UI berat yang akan kita buat, akan ada banyak hal yang dilakukan aplikasi hanya sebagai reaksi terhadap sesuatu. Sesuatu itu dapat dipicu oleh klik tetikus, penekanan tombol, perubahan ukuran jendela, atau sejumlah gerakan dan interaksi lainnya. Perekat yang memungkinkan semua ini adalah sesuatu yang dikenal sebagai peristiwa.

Sekarang, Anda mungkin tahu semua tentang peristiwa dari pengalaman Anda menggunakannya di dunia DOM. Cara React menangani peristiwa sedikit berbeda, dan perbedaan ini dapat mengejutkan Anda dengan berbagai cara jika Anda tidak memperhatikan dengan saksama. Jangan khawatir. Itulah sebabnya Anda memiliki buku ini! Kita mulai dengan beberapa contoh sederhana, lalu secara bertahap melihat hal-hal yang makin aneh, rumit, dan (ya!) membosankan.

10.1 MENDENGARKAN DAN MENANGGAPI PERISTIWA

Cara termudah untuk mempelajari tentang peristiwa di React adalah dengan benar-benar menggunakannya, dan itulah yang akan kita lakukan! Untuk membantu hal ini, kita memiliki contoh sederhana yang terdiri dari penghitung yang bertambah setiap kali Anda mengklik tombol. Awalnya, contoh kita akan terlihat seperti Gambar 10.1.

Gambar 10.1 Contoh Kita

Setiap kali Anda mengklik tombol plus, nilai penghitung akan bertambah 1. Setelah mengklik tombol plus beberapa kali, tampilannya akan seperti Gambar 10.2.

Gambar 10.2 Setelah Mengklik Tombol Plus Beberapa Kali (23?)

Di balik layar, cara kerja contoh ini cukup sederhana. Setiap kali Anda mengklik tombol, sebuah peristiwa akan dipicu. Kita mendengarkan peristiwa ini dan melakukan berbagai hal yang menggunakan React untuk memperbarui penghitung saat peristiwa ini terdengar.

Titik Awal

Untuk menghemat waktu kita semua, kita tidak akan membuat semua hal dalam contoh kita dari awal. Sekarang, Anda mungkin sudah memiliki gambaran yang baik tentang cara bekerja dengan komponen, gaya, status, dan sebagainya. Sebagai gantinya, kita akan memulai dengan contoh yang diimplementasikan sebagian yang berisi semua hal kecuali fungsionalitas terkait peristiwa yang akan kita pelajari di sini. Pertama, buat dokumen HTML baru dan pastikan titik awal Anda terlihat seperti berikut:

```
<!DOCTYPE html>
<html>
<head>
  <title>React! React! React!</title>
  <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
  <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.
  js"></script>

  <style>
    #container {
      padding: 50px;
      background-color: #FFF;
    }
  </style>
</head>

<body>
  <div id="container"></div>
  <script type="text/babel">

  </script>
</body>

</html>
```

Setelah dokumen HTML baru Anda terlihat seperti yang Anda lihat di atas, saatnya menambahkan contoh penghitung yang telah diimplementasikan sebagian. Di dalam tag skrip di bawah div penampung, tambahkan yang berikut:

```
var destination = document.querySelector("#container");

var Counter = React.createClass({
  render: function() {
    var textStyle = {
    fontSize: 72,
    fontFamily: "sans-serif",
    color: "#333",
    fontWeight: "bold"
    };

    return (
      <div style={textStyle}>
        {this.props.display}
      </div>
    );
  }
});

var CounterParent = React.createClass({
  getInitialState: function() {
    return {
      count: 0
    };
  },

  render: function() {
    var backgroundStyle = {
      padding: 50,
      backgroundColor: "#FFC53A",
      width: 250,
      height: 100,
      borderRadius: 10,
      textAlign: "center"
    };
    var buttonStyle = {
      fontSize: "1em",
      width: 30,
      height: 30,
      fontFamily: "sans-serif",
      color: "#333",
      fontWeight: "bold",
      lineHeight: "3px"
    };

    return (
      <div style={backgroundStyle}>
```

```
            <Counter display={this.state.count}/>
            <button style={buttonStyle}>+</button>
          </div>
        );
    }
});

ReactDOM.render(
    <div>
      <CounterParent/>
    </div>,
    destination
);
```

Setelah Anda menambahkan semua ini, pratinjau semuanya di browser Anda untuk memastikan semuanya ditampilkan. Anda akan melihat awal penghitung kita. Luangkan waktu sejenak untuk melihat apa saja yang dilakukan semua kode ini. Seharusnya tidak ada yang tampak aneh. Satu-satunya hal yang aneh adalah mengklik tombol plus tidak akan melakukan apa pun. Kita akan memperbaikinya di bagian berikutnya.

10.2 MEMBUAT TOMBOL YANG DIKLIK MELAKUKAN SESUATU

Setiap kali kita mengklik tombol plus, kita ingin nilai penghitung kita bertambah satu. Yang perlu kita lakukan kira-kira akan terlihat seperti ini:

1. Dengarkan peristiwa klik pada tombol dan tentukan pengendali peristiwa.
2. Terapkan pengendali peristiwa tempat kita meningkatkan nilai properti this.state.count yang diandalkan penghitung kita.

Kita akan langsung ke daftar dimulai dengan mendengarkan peristiwa klik. Di React, Anda mendengarkan peristiwa dengan menentukan semua hal sebaris di JSX itu sendiri. Lebih spesifiknya, Anda menentukan peristiwa yang tengah Anda dengarkan dan pengendali peristiwa yang akan dipanggil, semuanya di dalam markup Anda. Untuk melakukan ini, temukan fungsi return di dalam komponen CounterParent kita, lalu buat perubahan yang disorot berikut ini:

```
.
.
.
return (
    <div style={backgroundStyle}>
      <Counter display={this.state.count}/>
      <button onClick={this.increase} style={buttonStyle}>+</button>
    </div>
);
```

Yang telah kita lakukan adalah memberi tahu React untuk memanggil fungsi increase saat event onClick terdengar. Selanjutnya, mari kita lanjutkan dan terapkan fungsi increase alias event handler kita. Di dalam komponen CounterParent kita, tambahkan baris yang disorot

berikut:

```
var CounterParent = React.createClass({
  getInitialState: function() {
    return {
      count: 0
    };
  },
  increase: function(e) {
    this.setState({
    count: this.state.count + 1
    });
  },

  render: function() {
    var backgroundStyle = {
      padding: 50,
      backgroundColor: "#FFC53A",
      width: 250,
      height: 100,
      borderRadius: 10,
      textAlign: "center"
    };
    var buttonStyle = {
      fontSize: "1em",
      width: 30,
      height: 30,
      fontFamily: "sans-serif",
      color: "#333",
      fontWeight: "bold",
      lineHeight: "3px"
    };
    return (
      <div style={backgroundStyle}>
        <Counter display={this.state.count}/>
        <button onClick={this.increase} style={buttonStyle}>+</button>
      </div>
    );
  }
})
```

Yang kami lakukan dengan baris-baris ini adalah memastikan bahwa setiap panggilan ke fungsi increase akan menambah nilai properti this.state.count kami sebesar 1. Karena kami menangani peristiwa, fungsi increase Anda (sebagai pengendali peristiwa yang ditunjuk) akan mendapatkan akses ke argumen peristiwa. Kami telah menetapkan argumen peristiwa ini untuk diakses oleh e, dan Anda dapat melihatnya dengan melihat tanda tangan fungsi increase kami (alias seperti apa bentuk deklarasinya).

Kami akan membahas berbagai peristiwa dan propertinya sebentar lagi. Sekarang, lanjutkan dan pratinjau apa yang Anda miliki di browser Anda. Setelah semuanya dimuat, klik

tombol plus untuk melihat semua kode yang baru ditambahkan dalam aksi. Nilai counter kami akan meningkat dengan setiap klik! Bukankah itu cukup mengagumkan?

10.3 PROPERTI PERISTIWA

Seperti yang Anda ketahui, peristiwa kami meneruskan apa yang dikenal sebagai argumen peristiwa ke pengendali peristiwa kami. Argumen peristiwa ini berisi sekumpulan properti yang khusus untuk jenis peristiwa yang Anda hadapi. Di dunia DOM biasa, setiap peristiwa memiliki tipenya sendiri. Misalnya, jika Anda berurusan dengan peristiwa tetikus, peristiwa Anda dan objek argumen peristiwanya akan bertipe MouseEvent. Objek MouseEvent ini akan memungkinkan Anda mengakses informasi khusus tetikus, seperti tombol mana yang ditekan atau posisi klik tetikus di layar.

Argumen peristiwa untuk peristiwa yang terkait dengan papan ketik bertipe KeyboardEvent. Objek KeyboardEvent Anda berisi properti yang (di antara banyak hal lainnya) memungkinkan Anda mengetahui tombol mana yang sebenarnya ditekan. Saya dapat terus membahasnya untuk setiap tipe Peristiwa lainnya, tetapi Anda mengerti maksudnya. Setiap tipe Peristiwa berisi kumpulan propertinya sendiri yang dapat Anda akses melalui pengendali peristiwa untuk peristiwa tersebut.

Bertemu dengan Peristiwa Sintetis

Di React, saat Anda menentukan peristiwa di JSX seperti yang kita lakukan dengan onClick, Anda tidak secara langsung berurusan dengan peristiwa DOM biasa. Sebaliknya, Anda berurusan dengan tipe peristiwa khusus React yang dikenal sebagai SyntheticEvent. Penangan peristiwa Anda tidak mendapatkan argumen peristiwa asli bertipe MouseEvent, KeyboardEvent, dll. Mereka selalu mendapatkan argumen peristiwa bertipe SyntheticEvent yang membungkus peristiwa asli browser Anda. Apa dampak dari hal ini pada kode kita? Anehnya tidak banyak. Setiap SyntheticEvent berisi properti berikut:

Property Name	Type
bubbles	boolean
cancelable	boolean
currentTarget	DOMEventTarget
defaultPrevented	boolean
eventPhase	number
is Trusted	boolean
nativeEvent	DOMEvent
preventDefault()	void
isDefaultPrevented()	boolean
isPropagationStopped	void
target	DOMEventTarget
timeStamp	number
type	string

Properti ini seharusnya tampak cukup mudah dipahami dan generik! Hal-hal yang tidak generik bergantung pada jenis peristiwa asli yang dibungkus SyntheticEvent kita. Ini berarti

bahwa SyntheticEvent yang membungkus MouseEvent akan memiliki akses ke properti khusus tetikus seperti berikut:

```
boolean altKey
number button
number buttons
number clientx
number clientY
boolean ctrlkey
boolean getModifierState(key)
boolean metaKey
number pagex
number pageY
DOMEventTarget relatedTarget
number screenx
number screenY
boolean shiftKey
```

Demikian pula, SyntheticEvent yang membungkus KeyboardEvent akan memiliki akses ke properti terkait keyboard tambahan berikut:

```
boolean altKey
number charCode
boolean ctrlKey
boolean getModifierState(key)
string key
number keyCode
string locale
number location
boolean metaKey
boolean repeat
boolean shiftKey
number which
```

Pada akhirnya, semua ini berarti Anda tetap mendapatkan fungsionalitas yang sama di dunia SyntheticEvent seperti yang Anda miliki di dunia DOM asli. Sekarang, inilah sesuatu yang saya pelajari dengan cara yang sulit. Jangan merujuk ke dokumentasi acara DOM tradisional saat menggunakan acara Sintetis dan propertinya. Karena SyntheticEvent membungkus acara DOM asli Anda, acara dan propertinya mungkin tidak memetakan satu-ke-satu. Beberapa acara DOM bahkan tidak ada di React. Untuk menghindari masalah apa pun, jika Anda ingin mengetahui nama SyntheticEvent atau salah satu propertinya, rujuk ke dokumen Sistem Acara React.

10.4 MELAKUKAN SESUATU DENGAN PROPERTI ACARA

Sekarang, Anda mungkin telah melihat lebih banyak tentang DOM dan hal-hal SyntheticEvent daripada yang mungkin Anda inginkan. Untuk menghilangkan rasa dari semua teks itu, mari kita tulis beberapa kode dan manfaatkan semua pengetahuan baru ini dengan baik. Saat ini, contoh penghitung kita bertambah satu setiap kali Anda mengklik tombol plus.

Yang ingin kita lakukan adalah menambah penghitung kita sebanyak sepuluh saat tombol Shift pada keyboard ditekan sambil mengklik tombol plus dengan mouse kita. Cara kita melakukannya adalah dengan menggunakan properti shiftKey yang ada pada SyntheticEvent saat menggunakan mouse:

```
boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEventTarget relatedTarget
number screenX
number screenY
boolean shiftKey
```

Cara kerja properti ini sederhana. Jika tombol Shift ditekan saat peristiwa tetikus ini aktif, maka nilai properti shiftKey adalah true. Jika tidak, nilai properti shiftKey adalah false. Untuk menambah penghitung kita sebesar 10 saat tombol Shift ditekan, kembali ke fungsi increase kita dan buat perubahan yang disorot berikut ini:

```
increase: function(e) {
  var currentCount = this.state.count;

  if (e.shiftKey) {
    currentCount += 10;
  } else {
    currentCount += 1;
  }

  this.setState({
    count: currentCount
  });
},
```

Setelah Anda membuat perubahan, pratinjau contoh kami di browser. Setiap kali Anda mengklik tombol plus, penghitung Anda akan bertambah satu seperti sebelumnya. Jika Anda mengklik tombol plus sambil menekan tombol Shift, perhatikan bahwa penghitung kita bertambah 10. Alasan mengapa semua ini berhasil adalah karena kita mengubah perilaku penambahan tergantung pada apakah tombol Shift ditekan atau tidak. Hal itu terutama ditangani oleh baris berikut:

```
if (e.shiftKey) {
   currentCount += 10;
} else {
   currentCount += 1;
}
```

Jika properti shiftKey pada argumen acara SyntheticEvent kita benar, kita menambah penghitung kita sebesar 10. Jika nilai shiftKey salah, kita cukup menambahnya sebesar 1.

Lebih Banyak Kejenakaan Eventing

Kita belum selesai! Sampai titik ini, kita telah melihat cara bekerja dengan acara di React dengan cara yang sangat sederhana. Di dunia nyata, jarang sekali hal-hal yang langsung seperti yang telah kita lihat. Aplikasi Anda yang sebenarnya akan lebih rumit, dan karena React bersikeras melakukan berbagai hal secara berbeda, kita perlu mempelajari (atau mempelajari kembali) beberapa trik dan teknik baru yang terkait dengan acara agar aplikasi kita berfungsi. Di sinilah bagian ini berperan. Kita akan melihat beberapa situasi umum yang akan Anda hadapi dan cara mengatasinya.

Anda Tidak Dapat Mendengarkan Peristiwa pada Komponen Secara Langsung

Misalnya, komponen Anda tidak lebih dari sekadar tombol atau jenis elemen UI lain yang akan berinteraksi dengan pengguna. Anda tidak dapat melakukan hal seperti yang kita lihat pada baris yang disorot berikut:

```
var CounterParent = React.createclass ({
     getInitialstate: function() {
        return {
           count: 0
        };
},
increase: function() {
     this.setstate ({
        count: this.state.count + 1
     });
},
render: function() {
     return (
       <div>
         <Counter display={this.state.count}/>
         <PlusButton onClick={ .1ncrease}/>
      </div>
     );
}
});
```

Di permukaan, baris JSX ini tampak benar-benar valid. Saat seseorang mengklik komponen PlusButton kita, fungsi peningkatan akan dipanggil. Jika Anda penasaran, berikut tampilan komponen PlusButton kita:

```
var PlusButton = React.createClass({
   render: function() {
```

```
        return (
          <button>
            +
          </button>
        );
    }
});
```

Komponen PlusButton kita tidak melakukan hal yang aneh. Komponen ini hanya mengembalikan satu elemen HTML! Bagaimana pun Anda mengiris dan mengirisnya, semua ini tidak penting. Tidak masalah seberapa sederhana atau jelas HTML yang kita kembalikan melalui komponen. Anda tidak dapat mendengarkan peristiwa secara langsung. Alasannya adalah karena komponen adalah pembungkus untuk elemen DOM. Apa artinya mendengarkan peristiwa pada komponen? Setelah komponen Anda dibuka menjadi elemen DOM, apakah elemen HTML luar bertindak sebagai hal yang Anda dengarkan untuk peristiwa tersebut? Apakah itu elemen lain?

Bagaimana Anda membedakan antara mendengarkan peristiwa dan mendeklarasikan properti dengan nilai? Tidak ada jawaban yang jelas untuk semua pertanyaan tersebut. Terlalu kasar untuk mengatakan bahwa solusinya adalah dengan tidak mendengarkan peristiwa pada komponen. Untungnya, ada solusi di mana kita memperlakukan event handler sebagai prop dan meneruskannya ke komponen. Di dalam komponen, kita kemudian dapat menetapkan peristiwa ke elemen DOM dan menyetel event handler ke nilai prop yang baru saja kita teruskan. Saya sadar bahwa itu mungkin tidak masuk akal, jadi mari kita lihat contohnya. Perhatikan baris yang disorot berikut:

```
var CounterParent = React.createclass ({
      .
      .
      .
   render: function () {
      return (
         <div>
                  <Counter display={this.state.count}/>
                  <PlusButton clickHandler={this.increase}/>
            </div>
         );
    }
});
```

Dalam contoh ini, kita membuat properti yang disebut clickHandler yang nilainya adalah event handler increase. Di dalam komponen PlusButton, kita dapat melakukan hal berikut:

```
var PlusButton = React.createclass ({
      render: function () {
           return (
```

```
                <button onclick={this.props.clickHandler}>
                        +
                </button>

            );
        }
});
```

Pada elemen tombol, kita tentukan peristiwa onClick dan tetapkan nilainya ke properti clickHandler. Saat dijalankan, properti ini dievaluasi sebagai fungsi increase kita, dan mengklik tombol plus memastikan fungsi increase dipanggil. Ini memecahkan masalah kita sekaligus tetap memungkinkan komponen kita berpartisipasi dalam semua kebaikan eventing ini!

10.5 MENDENGARKAN PERISTIWA DOM REGULER

Jika Anda mengira bagian sebelumnya membosankan, tunggu hingga Anda melihat apa yang kami miliki di sini. Tidak semua peristiwa DOM memiliki padanan SyntheticEvent. Tampaknya Anda cukup menambahkan awalan on dan menggunakan huruf kapital pada peristiwa yang Anda dengarkan saat menentukannya secara inline di JSX Anda:

```
var Something = React.createClass({
  handleMyEvent: function(e) {
 // do var Something = React.createClass({
    handleMyEvent: function(e) {
     // do something
  },
    render: function() {
     return (
         <div onMyWeirdEvent={this.handleMyEvent}>Hello!</div>
     );
  }
});
```

Cara kerjanya tidak seperti itu! Untuk kejadian yang tidak dikenali secara resmi oleh React, Anda harus menggunakan pendekatan tradisional yang menggunakan addEventListener dengan beberapa langkah tambahan yang harus diikuti. Lihat bagian kode berikut:

```
var Something = React.createclass ({
      handleMyEvent: function(e) {
                // do something
      },
      componentDidMount: function () {
            window.addEventListener ("someEvent", this.handleMyEvent) ;
      },
      componentWillUnmount: function() {
             window.removeEventListener ("someEvent", this.handleMyEvent) ;
      }
             render: function() {
            return (
```

```
                <div>Hello !< /div>
            );
        }
});
```

Kita memiliki komponen Something yang mendengarkan suatu peristiwa yang disebut someEvent. Kita mulai mendengarkan peristiwa ini di bawah metode componentDidMount yang secara otomatis dipanggil saat komponen kita dirender. Cara kita mendengarkan peristiwa kita adalah dengan menggunakan addEventListener dan menentukan peristiwa dan pengendali peristiwa yang akan dipanggil:

```
var Something = React.createclass ({
     handleMyEvent: function(e) {
           // do something
     },
     componentDidMount: function () {
         window. addEventListener ("someEvent", this.handleMyEvent) ;
     },
     componentWillUnmount: function() {
         window.removeEventListener ("someEvent", this.handleMyEvent) ;
     },
     render: function() {
        return (
           <div>Hello !< /div>
     ) ;
  }
});
```

Itu seharusnya cukup mudah. Satu-satunya hal lain yang perlu Anda ingat adalah menghapus event listener saat komponen akan dihancurkan. Untuk melakukannya, Anda dapat menggunakan kebalikan dari metode componentDidMount, metode componentWillUnmount. Di dalam metode itu, masukkan panggilan removeEventListener Anda untuk memastikan tidak ada jejak event listening yang terjadi setelah komponen kita hilang.

Arti this Di Dalam Event Handler

Saat menangani event di React, nilai this di dalam event handler Anda berbeda dari apa yang biasanya Anda lihat di dunia DOM non-React. Di dunia non-React, nilai this di dalam event handler merujuk ke elemen tempat event Anda mendengarkan:

```
function doSomething(e) {
   console.log(this); //button element
}
var foo = document.querySelector("button");
foo.addEventListener("click", doSomething, false);
```

Di dunia React (ketika komponen Anda dibuat menggunakan React.createClass), nilai this di dalam event handler Anda selalu merujuk ke komponen tempat event handler tersebut

berada:

```
var CounterParent = React.createClass({
  getInitialState: function() {
    return {
      count: 0
    };
  },
  increase: function(e) {
    console.log(this); // CounterParent component
    this.setState({
      count: this.state.count + 1
    });
  },
  render: function() {
    return (
      <div>
        <Counter display={this.state.count}/>
        <button onClick={this.increase}>+</button>
      </div>
    );
  }
});
```

Dalam contoh ini, nilai this di dalam event handler increase merujuk ke komponen CounterParent. Nilai ini tidak merujuk ke elemen yang memicu event. Anda mendapatkan perilaku ini karena React secara otomatis mengikat semua metode di dalam komponen ke this.

Perilaku pengikatan otomatis ini hanya berlaku saat komponen Anda dibuat menggunakan React.createClass. Jika Anda menggunakan kelas ES6 untuk menentukan komponen Anda, nilai this di dalam event handler Anda tidak akan terdefinisi kecuali Anda mengikatnya sendiri secara eksplisit:

```
<button onclick={this.increase.bind(this) }>+</button>
```

Tidak ada keajaiban pengikatan otomatis yang terjadi dengan sintaksis kelas baru, jadi pastikan untuk mengingatnya jika Anda tidak menggunakan React.createClass untuk membuat komponen Anda.

React...Mengapa? Mengapa?!

Sebelum kita mengakhiri hari ini, mari kita gunakan waktu ini untuk membicarakan mengapa React memutuskan untuk menyimpang dari cara kita bekerja dengan peristiwa di masa lalu. Ada dua alasan:

1. Kompatibilitas Peramban
2. Peningkatan Kinerja

Mari kita uraikan sedikit tentang kedua alasan ini.

Kompatibilitas Peramban

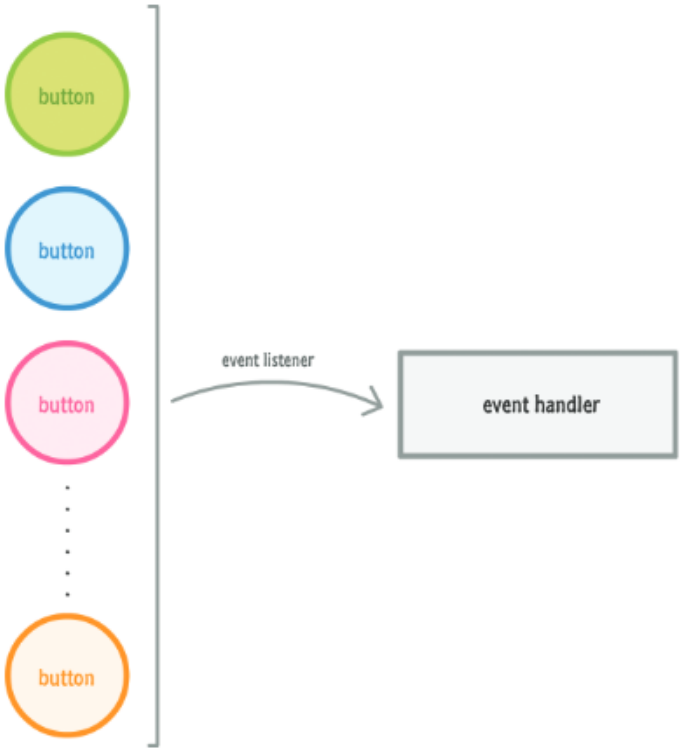
Penanganan peristiwa adalah salah satu hal yang sebagian besar berfungsi secara

konsisten di peramban modern, tetapi begitu Anda kembali ke versi peramban lama, semuanya menjadi sangat buruk dengan sangat cepat. Dengan membungkus semua peristiwa asli sebagai objek bertipe SyntheticEvent, React membebaskan Anda dari berurusan dengan keanehan penanganan peristiwa yang akhirnya harus Anda hadapi.

Peningkatan Kinerja

Dalam UI yang kompleks, semakin banyak penangan peristiwa yang Anda miliki, semakin banyak memori yang digunakan aplikasi Anda. Menangani hal itu secara manual tidaklah sulit, tetapi agak membosankan saat Anda mencoba mengelompokkan peristiwa di bawah induk yang sama. Terkadang, hal itu tidak mungkin dilakukan. Terkadang, kerepotan tidak lebih besar daripada manfaatnya. Apa yang dilakukan React cukup pintar.

React tidak pernah melampirkan pengendali peristiwa ke elemen DOM secara langsung. React menggunakan satu pengendali peristiwa di akar dokumen Anda yang bertanggung jawab untuk mendengarkan semua peristiwa dan memanggil pengendali peristiwa yang sesuai sebagaimana diperlukan (lihat Gambar 10.3).



Gambar 10.3 React Menggunakan Satu Event Handler Di Root Dokumen Anda

Hal ini membebaskan Anda dari keharusan untuk mengoptimalkan kode terkait event handler Anda sendiri. Jika Anda pernah melakukannya secara manual di masa lalu, Anda dapat bersantai karena React akan menangani tugas yang membosankan itu untuk Anda. Jika Anda belum pernah mengoptimalkan kode terkait event handler sendiri, anggaplah diri Anda beruntung.

10.6 KESIMPULAN

Anda akan menghabiskan banyak waktu untuk menangani event, dan bab ini memberikan banyak hal kepada Anda. Kami mulai dengan mempelajari dasar-dasar cara mendengarkan event dan menentukan event handler. Menjelang akhir, kami benar-benar terlibat dan melihat kasus-kasus yang akan Anda hadapi jika Anda tidak cukup berhati-hati. Anda tidak ingin menghadapi situasi yang tidak menyenangkan. Itu tidak pernah menyenangkan.

BAB 11
SIKLUS HIDUP KOMPONEN

Pada awalnya, kami memulai dengan pandangan yang sangat sederhana tentang komponen dan fungsinya. Saat kami mempelajari lebih lanjut tentang React dan melakukan hal-hal yang lebih keren dan lebih rumit, ternyata komponen kami tidak sesederhana itu. Komponen membantu menangani properti, status, peristiwa, dan sering kali bertanggung jawab atas kesejahteraan komponen lain juga. Melacak semua yang dilakukan komponen terkadang bisa jadi sulit.

Untuk membantu hal ini, React menyediakan sesuatu yang dikenal sebagai metode siklus hidup. Metode siklus hidup (tidak mengherankan) adalah metode khusus yang secara otomatis dipanggil saat komponen kami menjalankan bisnisnya. Metode ini memberi tahu kami tentang tonggak penting dalam kehidupan komponen kami, dan kami dapat menggunakan pemberitahuan ini untuk sekadar memperhatikan atau mengubah apa yang akan dilakukan komponen kami. Dalam bab ini, kami melihat metode siklus hidup ini dan mempelajari semua tentang apa yang dapat kami lakukan dengannya.

11.1 MENGENAL METODE SIKLUS HIDUP

Metode siklus hidup tidak terlalu rumit. Kita dapat menganggapnya sebagai pengendali peristiwa yang dimuliakan yang dipanggil pada berbagai titik dalam kehidupan suatu komponen, dan seperti pengendali peristiwa, Anda dapat menulis beberapa kode untuk melakukan berbagai hal pada berbagai titik tersebut. Sebelum kita melangkah lebih jauh, inilah saatnya bagi Anda untuk segera mengenal metode siklus hidup kami. Metode-metode tersebut adalah:

- `componentWillMount`
- `componentDidMount`
- `componentWillUnmount`
- `componentWillUpdate`
- `componentDidUpdate`
- `shouldComponentUpdate`
- `componentWillReceiveProps`

Kami belum selesai. Ada tiga metode lagi yang akan kami masukkan ke dalam campuran meskipun metode-metode tersebut bukan metode siklus hidup yang ketat, yaitu:

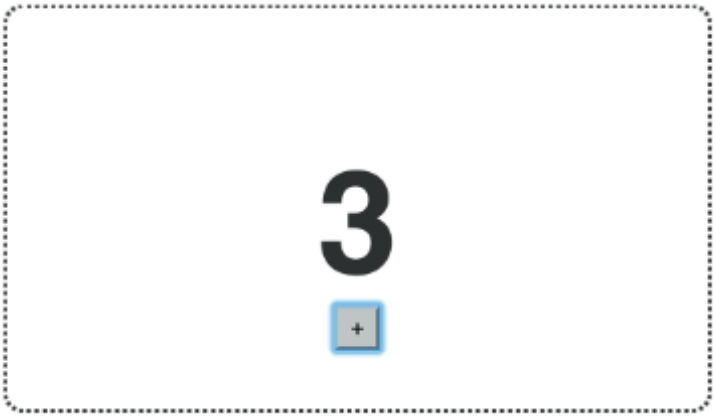
- `getInitialState`
- `getDefaultProps`
- `render`

Beberapa nama ini mungkin terdengar familier bagi Anda, dan beberapa mungkin baru pertama kali Anda lihat. Jangan khawatir. Setelah semua ini, Anda akan mengenal semuanya! Yang akan kita lakukan adalah melihat metode siklus hidup ini dari berbagai sudut dimulai

dengan beberapa kode!

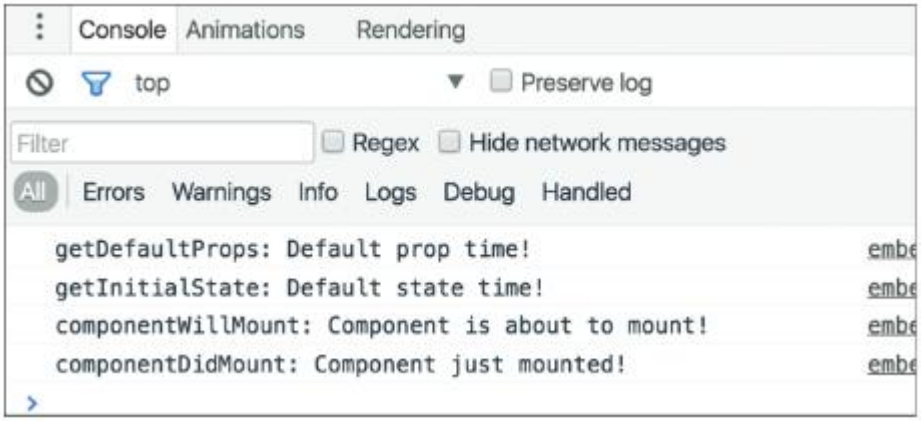
Lihat Metode Siklus Hidup dalam Aksi

Mempelajari metode siklus hidup ini sama menariknya dengan menghafal nama tempat asing (atau sistem bintang yang jauh!) yang tidak akan Anda kunjungi. Untuk membantu membuat semua ini lebih mudah dipahami, saya akan meminta Anda untuk mencoba metode ini melalui contoh sederhana sebelum kita membahasnya secara akademis. Untuk mencoba contoh ini, buka URL berikut: https://www.kirupa.com/react/lifecycle _example.htm Setelah halaman ini dimuat, Anda akan melihat variasi dari contoh tandingan yang kita lihat sebelumnya (lihat Gambar 11.1).



Gambar 11.1 Variasi Pada Contoh Penghitung

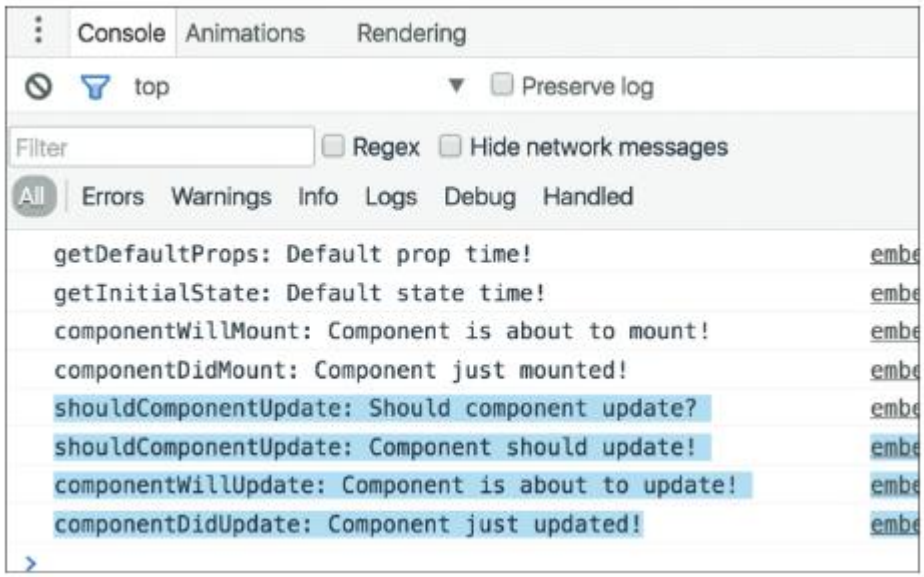
Jangan klik tombol atau apa pun dulu. Jika Anda sudah mengklik tombol, cukup segarkan halaman untuk memulai contoh dari awal. Ada alasan mengapa saya mengatakan itu, dan itu bukan karena OCD saya sedang kambuh :P Kita ingin melihat halaman ini sebagaimana adanya sebelum kita berinteraksi dengannya! Sekarang, buka alat pengembang browser Anda dan lihat tab Konsol. Di Chrome, Anda akan melihat sesuatu yang tampak seperti Gambar 11.2.



Gambar 11.2 Tampilan Konsol Di Chrome

Perhatikan apa yang Anda lihat tercetak. Anda akan melihat beberapa pesan, dan pesan-pesan

ini dimulai dengan nama yang tampak seperti metode siklus hidup. Jika Anda mengklik tombol plus sekali, perhatikan bahwa Konsol Anda akan menampilkan lebih banyak metode siklus hidup yang dipanggil (lihat Gambar 11.3).



Gambar 11.3 Lebih Banyak Metode Siklus Hidup Yang Dipanggil

Cobalah contoh ini sebentar. Contoh ini memungkinkan Anda menempatkan semua metode siklus hidup ini dalam konteks komponen yang telah kita lihat sebelumnya. Saat Anda terus menekan tombol plus, lebih banyak entri metode siklus hidup akan muncul. Akhirnya, setelah penghitung Anda mendekati nilai 5, contoh Anda akan hilang begitu saja dengan entri berikut yang muncul di konsol Anda: componentWillUnmount: Komponen akan segera dihapus dari DOM! Pada titik ini, Anda telah mencapai akhir dari contoh ini. Tentu saja, untuk memulai lagi, Anda cukup menyegarkan halaman! Sekarang setelah Anda melihat contohnya, mari kita lihat sekilas komponen yang bertanggung jawab atas semua ini:

```
var CounterParent = React.createClass({
  getDefaultProps: function(){
    console.log("getDefaultProps: Default prop time!");
    return {};
  },
  getInitialState: function() {
    console.log("getInitialState: Default state time!");
    return {
      count: 0
    };
  },
  increase: function() {
    this.setState({
      count: this.state.count + 1
    });
  },
  componentWillUpdate: function(newProps, newState) {
    console.log("componentWillUpdate: Component is about to update!");
```

```
  },
  componentDidUpdate: function(currentProps, currentState) {
    console.log("componentDidUpdate: Component just updated!");
  },
  componentWillMount: function() {
    console.log("componentWillMount: Component is about to mount!");
  },
  componentDidMount: function() {
    console.log("componentDidMount: Component just mounted!");
  },
  componentWillUnmount: function() {
    console.log("componentWillUnmount: Component is about to be removed
    from the DOM!");
  },
  shouldComponentUpdate: function(newProps, newState) {
    console.log("shouldComponentUpdate: Should component update?");

    if (newState.count < 5) {
      console.log("shouldComponentUpdate: Component should update!");
      return true;
    } else {
      ReactDOM.unmountComponentAtNode(destination);
      console.log("shouldComponentUpdate: Component should not update!");
      return false;
    }
  },
      componentWillReceiveProps: function(newProps){
    console.log("componentWillReceiveProps: Component will get new
    props!");
  },
      render: function() {
    var backgroundStyle = {
      padding: 50,
      border: "#333 2px dotted",
      width: 250,
      height: 100,
      borderRadius: 10,
      textAlign: "center"
    };

    return (
      <div style={backgroundStyle}>
        <Counter display={this.state.count}/>
        <button onClick={this.increase}>
          +
        </button>
      </div>
    );
  }
});
```

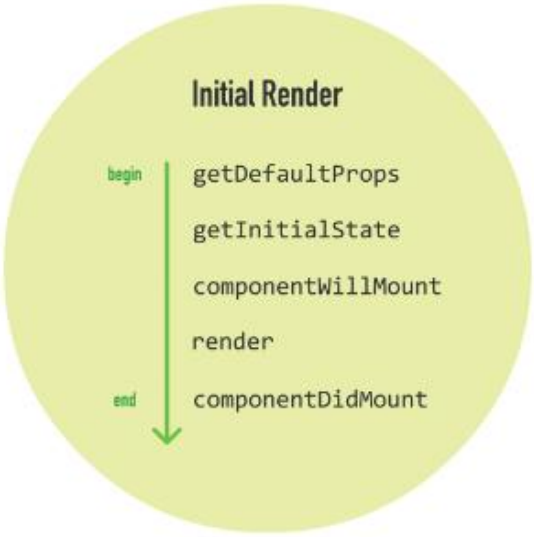
Luangkan waktu sejenak untuk melihat apa saja fungsi semua kode ini. Kode ini tampaknya panjang, tetapi sebagian besar kode ini hanya berisi setiap metode siklus hidup yang tercantum dengan pernyataan console.log yang didefinisikan. Setelah Anda membaca kode ini, cobalah contoh ini sekali lagi. Percayalah.

Semakin banyak waktu yang Anda habiskan untuk mempelajari contoh ini dan mencari tahu apa yang terjadi, semakin menyenangkan Anda nantinya. Bagian berikut yang membahas

setiap metode siklus hidup di seluruh fase rendering, pembaruan, dan pelepasan akan sangat membosankan. Jangan bilang saya tidak memperingatkan Anda.

11.2 FASE RENDERING AWAL

Saat komponen Anda akan memulai masa pakainya dan menuju ke DOM, metode siklus hidup berikut akan dipanggil (lihat Gambar 11.4).



Gambar 11.4 Metode Siklus Hidup Yang Awalnya Dipanggil

Apa yang Anda lihat di konsol saat contoh dimuat adalah versi yang kurang berwarna dari apa yang Anda lihat di sini. Sekarang, kita akan melangkah lebih jauh dan mempelajari lebih lanjut tentang apa yang dilakukan masing-masing metode siklus hidup ini.

getDefaultProps

Metode ini memungkinkan Anda untuk menentukan nilai default dari this.props. Metode ini dipanggil bahkan sebelum komponen Anda dibuat atau properti apa pun dari induknya diteruskan.

getInitialState

Metode ini memungkinkan Anda untuk menentukan nilai default dari this.state sebelum komponen Anda dibuat. Sama seperti getDefaultProps, metode ini juga dipanggil sebelum komponen Anda dibuat.

componentWillMount

Ini adalah metode terakhir yang dipanggil sebelum komponen Anda dirender ke DOM. Ada hal penting yang perlu diperhatikan di sini. Jika Anda memanggil setState di dalam metode ini, komponen Anda tidak akan dirender ulang (alias metode render dipanggil dan memperbarui apa yang ditampilkan di layar).

render

Yang ini seharusnya sudah sangat familiar bagi Anda sekarang. Setiap komponen harus memiliki metode ini yang ditetapkan, dan metode ini bertanggung jawab untuk mengembalikan satu simpul akar (yang mungkin memiliki banyak simpul anak di dalamnya).

Jika Anda tidak ingin merender apa pun (untuk beberapa pengoptimalan mewah yang mungkin Anda lakukan), cukup kembalikan null atau false.

componentDidMount

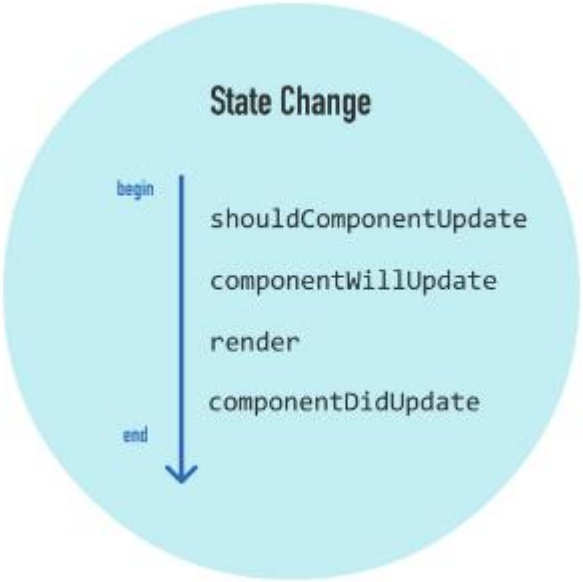
Metode ini dipanggil segera setelah komponen Anda dirender dan ditempatkan di DOM. Pada titik ini, Anda dapat dengan aman melakukan operasi kueri DOM apa pun tanpa perlu khawatir apakah komponen Anda telah berhasil atau belum. Jika Anda memiliki kode apa pun yang bergantung pada kesiapan komponen Anda, Anda dapat menentukan semua kode itu di sini juga. Dengan pengecualian metode render, semua metode siklus hidup ini hanya dapat diaktifkan satu kali. Itu sangat berbeda dari metode yang akan kita lihat selanjutnya.

11.3 FASE PEMBARUAN

Setelah komponen Anda ditambahkan ke DOM, komponen tersebut berpotensi diperbarui dan dirender ulang saat terjadi perubahan properti atau status. Selama waktu ini, kumpulan metode siklus hidup yang berbeda akan dipanggil.

Menangani Perubahan Status

Pertama, mari kita lihat perubahan status! Saat terjadi perubahan status, kami sebutkan sebelumnya bahwa komponen Anda akan memanggil metode rendernya lagi. Setiap komponen yang bergantung pada output komponen ini juga akan mendapatkan metode render yang dipanggil. Ini dilakukan untuk memastikan bahwa komponen kita selalu menampilkan versi terbaru dari dirinya sendiri. Semua itu benar, tetapi itu hanya representasi sebagian dari apa yang terjadi. Saat terjadi perubahan status, semua metode siklus hidup pada Gambar 11.5 dipanggil.



Gambar 11.5 Metode Siklus Hidup Yang Dipanggil Saat Terjadi Perubahan Status.

Apa yang dilakukan metode siklus hidup ini diuraikan dalam bagian berikut.

shouldComponentUpdate

Terkadang, Anda tidak ingin komponen Anda diperbarui saat terjadi perubahan status.

Metode ini memungkinkan Anda untuk mengontrol perilaku pembaruan ini. Jika Anda menggunakan metode ini dan mengembalikan nilai true, komponen akan diperbarui. Jika metode ini mengembalikan nilai false, komponen ini akan melewati pembaruan. Itu mungkin terdengar sedikit membingungkan, jadi berikut cuplikan sederhananya:

```
shouldComponentUpdate: function(newProps, newState) {

   if (newState.id <= 2) {
      console.log("Component should update!");

      return true;
   } else {
      console.log("Component should not update!");

      return false;
   }
}
```

Metode ini dipanggil dengan dua argumen yang kami beri nama newProps dan newState. Yang kami lakukan dalam potongan kode ini adalah memeriksa apakah nilai baru dari properti status id kami kurang dari atau sama dengan 2. Jika nilainya kurang dari atau sama dengan 2, kami mengembalikan true untuk menunjukkan bahwa komponen ini harus diperbarui. Jika nilainya tidak kurang dari atau sama dengan 2, kami mengembalikan false untuk menunjukkan bahwa komponen ini tidak boleh diperbarui.

componentWillUpdate

Metode ini dipanggil tepat sebelum komponen Anda akan diperbarui. Tidak ada yang terlalu menarik di sini. Satu hal yang perlu diperhatikan adalah Anda tidak dapat mengubah status dengan memanggil this.setState dari metode ini.

render

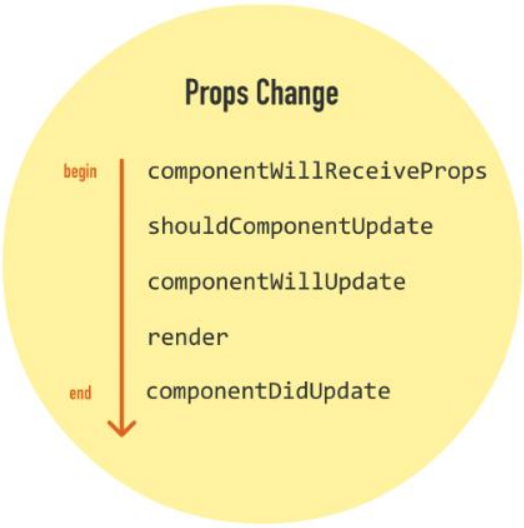
Jika Anda tidak mengganti pembaruan melalui shouldComponentUpdate (dengan mengembalikan false), kode di dalam render akan dipanggil lagi untuk memastikan komponen Anda ditampilkan dengan benar.

componentDidUpdate

Metode ini dipanggil setelah komponen Anda diperbarui dan metode render telah dipanggil. Jika Anda perlu menjalankan kode apa pun setelah pembaruan berlangsung, ini adalah tempat untuk menyimpannya.

Menangani Perubahan Properti

Waktu lain komponen Anda diperbarui adalah ketika nilai propertinya berubah setelah dirender ke DOM. Dalam skenario ini, metode siklus hidup pada Gambar 11.6 dipanggil.

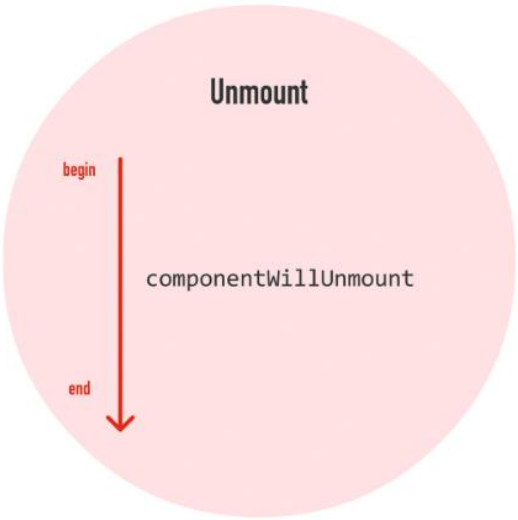


Gambar 11.6 Metode Siklus Hidup Ketika Nilai Properti Komponen Berubah

Satu-satunya metode yang baru di sini adalah componentWillReceiveProps. Metode ini mengembalikan satu argumen, dan argumen ini adalah objek yang berisi nilai properti baru yang akan ditetapkan padanya. Kita melihat metode siklus hidup lainnya sebelumnya saat melihat perubahan status, jadi jangan membahasnya lagi. Perilakunya identik saat menangani perubahan properti.

11.4 FASE PELEPASAN

Fase terakhir yang akan kita lihat adalah saat komponen Anda akan dihancurkan dan dihapus dari DOM (lihat Gambar 11.7).



Gambar 11.7 Hanya Satu Metode Siklus Hidup Yang Aktif Saat Komponen Anda Akan Dihancurkan Dan Dihapus Dari DOM

Hanya ada satu metode siklus hidup yang aktif di sini, yaitu componentWillUnmount. Anda akan melakukan tugas terkait pembersihan di sini seperti menghapus event listener,

menghentikan timer, dll. Setelah metode ini dipanggil, komponen Anda dihapus dari DOM dan Anda dapat mengucapkan Selamat Tinggal padanya.

11.5 KESIMPULAN

Komponen-komponen kita adalah hal-hal kecil yang menarik. Di permukaan, komponen-komponen tersebut tampak tidak memiliki banyak hal yang terjadi. Seperti film dokumenter yang bagus tentang lautan, ketika kita melihat lebih dalam dan lebih dekat, rasanya seperti melihat dunia lain. Ternyata, React terus-menerus mengawasi dan memberi tahu komponen Anda setiap kali sesuatu yang menarik terjadi. Semua ini dilakukan melalui metode siklus hidup (yang sangat membosankan) yang telah kita bahas sepanjang tutorial ini.

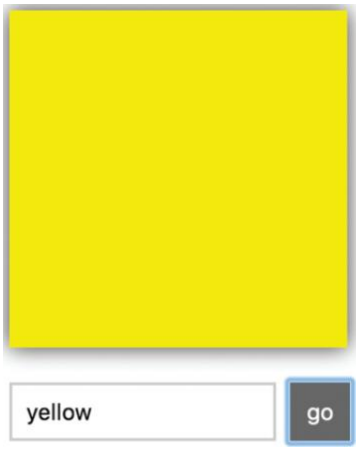
Sekarang, saya ingin meyakinkan Anda bahwa mengetahui apa yang dilakukan setiap metode siklus hidup dan kapan metode itu dipanggil akan berguna suatu hari nanti. Semua yang telah Anda pelajari bukanlah sekadar pengetahuan yang sepele, meskipun teman-teman Anda akan terkesan jika Anda dapat menjelaskan semua metode siklus hidup dari ingatan. Silakan dan cobalah lain kali Anda melihatnya.

BAB 12
MENGAKSES ELEMEN DOM

Akan ada saatnya Anda ingin mengakses properti dan metode pada elemen HTML secara langsung. Dalam dunia React yang berwarna di mana JSX mewakili segala hal yang baik dan murni tentang markup, mengapa Anda ingin berurusan langsung dengan HTML yang mengerikan? Seperti yang akan Anda ketahui (jika Anda belum mengetahuinya), ada banyak kasus di mana berurusan dengan elemen HTML melalui API DOM JavaScript secara langsung lebih mudah daripada mengutak-atik "cara React" dalam melakukan sesuatu. Untuk menyoroti salah satu situasi tersebut, lihat contoh Colorizer pada Gambar 12.1.

Gambar 12.1 Contoh Colorizer

Colorizer mewarnai kotak putih (saat ini) dengan warna apa pun yang Anda berikan. Untuk melihatnya beraksi, masukkan nilai warna di dalam kolom teks dan klik/ketuk tombol mulai. Jika Anda tidak tahu warna apa yang harus dimasukkan, kuning adalah pilihan yang bagus! Setelah Anda memberikan warna dan mengirimkannya, kotak putih akan berubah menjadi nilai warna apa pun yang Anda berikan (lihat Gambar 12.2).



Gambar 12.2 Kotak Putih Berubah Menjadi Kuning

Bahwa kotak berubah warna untuk setiap nilai warna valid yang Anda kirimkan cukup mengagumkan, tetapi bukan itu yang saya ingin Anda fokuskan. Sebaliknya, perhatikan kolom teks dan tombol setelah Anda mengirimkan nilai. Perhatikan bahwa tombol tersebut mendapat fokus, dan nilai warna yang baru saja Anda kirimkan masih ditampilkan di dalam formulir. Jika Anda ingin memasukkan nilai warna lain, Anda perlu mengembalikan fokus secara eksplisit ke kolom teks dan menghapus nilai apa pun yang ada saat ini.

Gambar 12.3 Kita Memperoleh Warna Ungu Dan Bidang Teks Siap Untuk Warna Berikutnya

Nilai ungu yang dimasukkan dihapus, dan fokus dikembalikan ke bidang teks. Ini memungkinkan kita memasukkan nilai warna tambahan dan mengirimkannya dengan mudah tanpa harus terus-menerus berpindah fokus antara bidang teks dan tombol. Bukankah itu jauh lebih baik? Mendapatkan perilaku ini dengan benar menggunakan JSX dan teknik React tradisional itu sulit. Kita bahkan tidak akan repot-repot menjelaskan cara melakukannya.

Mendapatkan perilaku ini dengan benar dengan menangani API DOM JavaScript pada berbagai elemen HTML secara langsung cukup mudah. Coba tebak apa yang akan kita lakukan? Di bagian berikut, kita menggunakan sesuatu yang dikenal sebagai refs yang disediakan React untuk membantu kita mengakses API DOM pada elemen HTML. Bab ini terdengar SANGAT membosankan, tetapi akan menjadi bab yang menyenangkan saya yakin akan hal itu.

12.1 MENGENAL REFS

Seperti yang Anda ketahui dengan baik sekarang, di dalam berbagai metode render kita, kita telah menulis hal-hal seperti HTML yang dikenal sebagai JSX. JSX kita hanyalah deskripsi tentang seperti apa seharusnya tampilan DOM. JSX tidak mewakili HTML yang sebenarnya meskipun tampak sangat mirip. Bagaimanapun, untuk menyediakan jembatan antara JSX dan elemen HTML akhir dalam DOM, React menyediakan sesuatu yang dikenal sebagai refs (singkatan dari references). Cara kerja refs agak aneh. Cara termudah untuk memahaminya adalah dengan melihat contoh. Katakanlah kita memiliki metode render dari contoh Colorizer yang terlihat seperti berikut:

```
  render: function() {
    var squareStyle = {
      backgroundColor: this.state.bgColor
    };

    return (
      <div className="colorArea">
        <div style={squareStyle} className="colorSquare"></div>

        <form onSubmit={this.setNewColor}>
          <input
            onChange={this.colorValue}
            placeholder="Enter a color value">
          </input>
          <button type="submit">go</button>
        </form>
      </div>
    );
  }
```

Di dalam metode render ini, kami mengembalikan sebagian besar JSX yang mewakili (antara lain) elemen input tempat kami memasukkan nilai warna. Yang ingin kami lakukan adalah mengakses representasi DOM elemen input sehingga kami dapat memanggil beberapa API di dalamnya menggunakan JavaScript. Cara kami melakukannya menggunakan refs adalah dengan menyetel atribut ref pada elemen yang ingin kami referensikan HTML-nya:

```
render: function() {
   var squareStyle = {
        backgroundColor: this.state.bgColor
   };

   return (
        <div className="colorArea">
              <div style={squareStyle} className="colorSquare"></div>

            <form onSubmit={this.setNewColor}>
                <input
```

```
            ref={}
            onChange={this.colorValue}
            placeholder="Enter a color value">
          </input>
          <button type="submit">go</button>
        </form>
    </div>
  );
}
```

Karena kita tertarik pada elemen input, atribut ref kita dilampirkan padanya. Saat ini, atribut ref kita kosong. Yang biasanya Anda tetapkan sebagai nilai atribut ref adalah fungsi panggilan balik JavaScript. Fungsi ini dipanggil secara otomatis saat komponen yang menampung metode render ini dipasang. Jika kita menetapkan nilai atribut ref kita ke fungsi JavaScript sederhana yang menyimpan referensi ke elemen DOM yang direferensikan, akan terlihat seperti baris yang disorot berikut:

```
  render: function() {
    var squareStyle = {
      backgroundColor: this.state.bgColor
    };

    var self = this;

    return (
      <div className="colorArea">
        <div style={squareStyle} className="colorSquare"></div>

        <form onSubmit={this.setNewColor}>
          <input
            ref={
                function(el) {
                  self._input = el;
                }
              }
            onChange={this.colorValue}
            placeholder="Enter a color value">
          </input>
          <button type="submit">go</button>
        </form>
      </div>
    );
  }
```

Hasil akhir dari kode ini yang dijalankan setelah komponen kita terpasang adalah sederhana: kita dapat mengakses HTML yang mewakili elemen input kita dari mana saja di dalam komponen kita dengan memanggil this._input. Luangkan waktu sejenak untuk melihat bagaimana baris kode yang disorot membantu melakukannya. Setelah selesai, kita akan

membahas kode ini bersama-sama. Pertama, fungsi panggilan balik kita terlihat seperti berikut:

```
function(el) {
  self._input = el;
}
```

Fungsi anonim ini dipanggil saat komponen kita dipasang, dan referensi ke elemen HTML DOM final dilewatkan sebagai argumen. Kita menangkap argumen ini menggunakan pengidentifikasi el, tetapi Anda dapat menggunakan nama apa pun untuk argumen ini sesuai keinginan. Isi fungsi panggilan balik ini cukup menetapkan properti kustom yang disebut _input ke nilai elemen DOM kita. Untuk memastikan kita membuat properti ini pada komponen kita, kita menggunakan variabel self untuk membuat penutupan di mana this yang dimaksud merujuk ke komponen kita dan bukan ke fungsi panggilan balik itu sendiri.

(Pengikatan otomatis tidak terjadi secara otomatis kali ini! Mengambil langkah mundur dan melihat gambaran yang lebih besar yang menghubungkan semuanya termasuk metode render yang baru saja kita lihat, mari kita lihat komponen Colorizer lengkap dengan semua keanehan terkait ref yang disorot:

```
var Colorizer = React.createClass({
  getInitialState: function() {
    return {
      color: '',
      bgColor: ''
    }
  },
  colorValue: function(e) {
    this.setState({color: e.target.value});
  },
  setNewColor: function(e){
    this.setState({bgColor: this.state.color});

    this._input.value = "";
    this._input.focus();

    e.preventDefault();
  },
  render: function() {
    var squareStyle = {
      backgroundColor: this.state.bgColor
    };
  var self = this;

  return (
    <div className="colorArea">
      <div style={squareStyle} className="colorSquare"></div>
      <form onSubmit={this.setNewColor}>
        <input
```

```
            ref={
               function(el) {
                  self._input = el;
                  }
               }
            onChange={this.colorValue}
            placeholder="Enter a color value">
         </input>
         <button type="submit">go</button>
      </form>
   </div>
   );
  }
});
```

Berfokus hanya pada apa yang terjadi pada elemen input kita, saat formulir dikirimkan dan metode setNewColor dipanggil, kita menghapus konten elemen input kita dengan memanggil this._input.value = "". Kita menetapkan fokus pada elemen input kita dengan memanggil this._input.focus(). Semua pekerjaan kita yang terkait dengan ref hanya untuk mengaktifkan kedua baris ini di mana kita memerlukan cara agar this._input mengarah ke elemen HTML yang mewakili elemen input kita yang kita definisikan di JSX. Setelah kita mengetahuinya, kita tinggal memanggil properti value dan metode focus yang diekspos DOM API pada elemen ini.

12.2 MENYEDERHANAKAN LEBIH LANJUT DENGAN FUNGSI PANAH ES6

Mempelajari React sudah cukup sulit, jadi saya telah mencoba untuk tidak memaksa Anda menggunakan teknik ES6 secara default. Saat bekerja dengan atribut ref, menggunakan fungsi panah untuk menangani fungsi panggilan balik memang sedikit menyederhanakan masalah. Ini adalah salah satu kasus di mana saya sarankan Anda menggunakan teknik ES6. Seperti yang Anda lihat beberapa saat yang lalu, untuk menetapkan properti pada komponen kita ke elemen HTML yang direferensikan, kita melakukan sesuatu seperti ini:

```
<input>
   ref={
      function(el) {
         self._input = el;
      }
   }
   onChange={this.colorValue}
   placeholder="Enter a color value">
</input>
```

Untuk mengatasi keanehan dalam scoping, kami membuat variabel self yang diinisialisasi ke this untuk memastikan kami membuat properti _input pada komponen kami. Itu tampaknya sangat berantakan. Dengan menggunakan fungsi arrow, kami dapat menyederhanakan semua ini menjadi seperti berikut:

```
<input
   ref={
      (el) => this._input = el
   }
   onChange={this.colorValue}
   placeholder="Enter a color value">
</input>
```

Hasil akhirnya identik dengan apa yang telah kita pelajari selama ini, dan karena bagaimana fungsi arrow menangani cakupan, Anda dapat menggunakan ini di dalam badan fungsi dan merujuk komponen tanpa melakukan pekerjaan tambahan. Tidak perlu variabel self eksternal yang setara!

12.3 KESIMPULAN

Dalam tutorial ini, kita melihat betapa "mudahnya" mengakses elemen DOM secara langsung. React dulu menyediakan cara yang jauh lebih mudah untuk merujuk elemen. Anda dapat menyetel atribut refs pada elemen dan menginisialisasinya ke nilai string:

```
<button refs="myButton">Click me!</button>
```

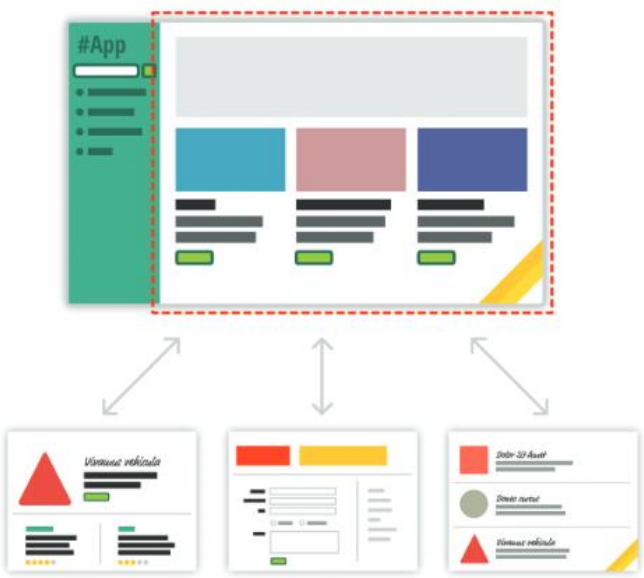
Anda kemudian dapat mengakses elemen ini setelah komponen dipasang dengan melakukan sesuatu seperti this.refs.myButton. Sebelum Anda benar-benar bersemangat menggunakan sesuatu seperti ini pada pendekatan panggilan balik fungsi kita dengan atribut ref, pendekatan berbasis string ini kemungkinan besar akan ditinggalkan. Pendekatan ini berfungsi saat tulisan ini dibuat, tetapi siapa yang tahu kapan pendekatan ini akan berhenti berfungsi. Sekarang, mengingat pendekatan ini akan segera dihapus, Anda mungkin bertanya-tanya mengapa saya memberi tahu Anda tentang hal ini. Sejujurnya, saya benar-benar tidak tahu.

BAB 13
MEMBUAT APLIKASI SATU HALAMAN MENGGUNAKAN REACT ROUTER

13.1 MEMBANGUN SPA DENGAN REACT

Sekarang setelah Anda memahami dasar-dasar cara bekerja dengan React, mari kita tingkatkan beberapa hal. Yang akan kita lakukan adalah menggunakan React untuk membuat aplikasi satu halaman sederhana (juga disebut SPA oleh anak-anak gaul dan orang-orang yang tinggal di Skandinavia). Seperti yang telah kita bahas di Bab 1, aplikasi satu halaman berbeda dari aplikasi multihalaman yang lebih tradisional yang Anda lihat di mana-mana.

Perbedaan terbesarnya adalah menavigasi aplikasi satu halaman tidak melibatkan membuka halaman yang sama sekali baru. Sebaliknya, halaman Anda (umumnya dikenal sebagai tampilan dalam konteks ini) biasanya dimuat sebaris dalam halaman yang sama seperti yang diilustrasikan pada Gambar 13.1.



Gambar 13.1 Aplikasi Satu Halaman Menggunakan Tampilan Pemuatan Sebaris Daripada Memuat Halaman Baru

Saat Anda memuat konten secara inline, banyak hal menjadi sedikit menantang. Bagian tersulitnya bukanlah memuat konten itu sendiri. Itu relatif mudah. Bagian tersulitnya adalah memastikan bahwa aplikasi satu halaman berperilaku dengan cara yang konsisten dengan apa yang biasa dilakukan pengguna Anda. Lebih khusus lagi, saat pengguna menavigasi aplikasi Anda, mereka mengharapkan bahwa:

1. URL yang ditampilkan di bilah alamat selalu mencerminkan hal yang sedang mereka lihat.
2. Mereka dapat menggunakan tombol kembali dan maju pada browser dengan sukses.

3. Mereka dapat menavigasi ke tampilan tertentu (alias deep link) secara langsung menggunakan URL yang sesuai.

Dengan aplikasi multihalaman, ketiga hal ini tersedia secara gratis. Tidak ada hal tambahan yang harus Anda lakukan untuk semua itu. Dengan aplikasi satu halaman, karena Anda tidak menavigasi ke halaman yang sama sekali baru, Anda harus bekerja keras untuk menangani ketiga hal yang diharapkan pengguna Anda agar berfungsi. Anda perlu memastikan bahwa navigasi dalam aplikasi Anda menyesuaikan URL dengan tepat. Anda perlu memastikan riwayat peramban Anda disinkronkan dengan benar dengan setiap navigasi agar pengguna dapat menggunakan tombol kembali dan maju.

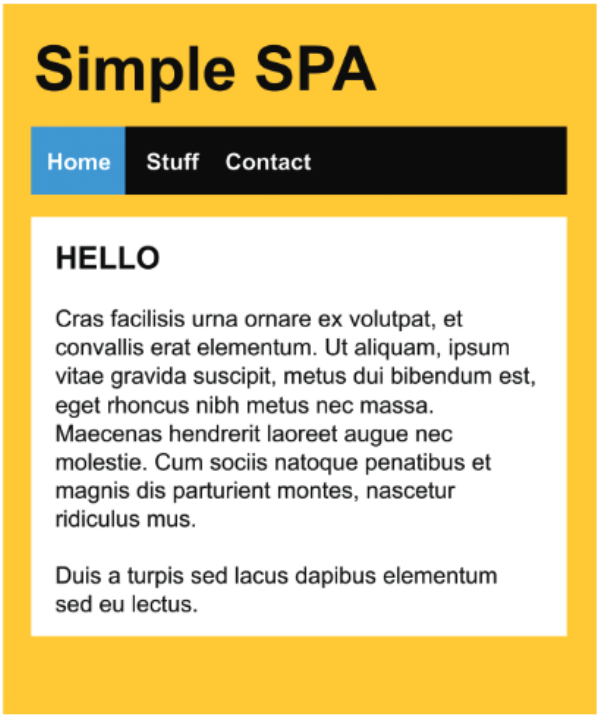
Jika pengguna menandai tampilan tertentu atau menyalin/menempel URL untuk mengaksesnya nanti, Anda perlu memastikan bahwa aplikasi satu halaman Anda mengarahkan pengguna ke tempat yang benar. Untuk mengatasi semua ini, Anda memiliki banyak teknik yang dikenal sebagai perutean. Perutean adalah saat Anda mencoba memetakan URL ke tujuan yang bukan halaman fisik, seperti tampilan individual di aplikasi satu halaman Anda. Kedengarannya rumit, tetapi untungnya ada banyak pustaka JavaScript yang membantu kita mengatasi hal ini. Salah satu pustaka JavaScript tersebut adalah bintang bab ini, React Router (https://github.com/reactjs/react-router).

React Router menyediakan kemampuan perutean ke aplikasi satu halaman yang dibangun di React, dan yang membuatnya menarik adalah ia memperluas apa yang sudah Anda ketahui tentang React dengan cara yang familier untuk memberi Anda semua kehebatan perutean ini. Dalam bab ini, Anda mempelajari semua tentang cara melakukannya—dan semoga lebih banyak lagi!

Contoh

Yang Anda lihat di sini adalah aplikasi React sederhana yang menggunakan React Router untuk menyediakan semua navigasi dan tampilan yang bagus! Meskipun tangkapan layar aplikasi terlihat bagus, ini adalah salah satu kasus di mana Anda ingin mencoba aplikasi untuk melihat lebih banyak fungsinya. Silakan buka halaman ini (https://www.kirupa.com/react/examples/react_router_final.htm) di jendela perambannya sendiri, klik berbagai tab navigasi untuk melihat tampilan yang berbeda, dan gunakan tombol kembali dan maju untuk melihatnya berfungsi.

Di bagian berikut, kita akan membangun aplikasi ini dalam beberapa bagian. Pada akhirnya, Anda tidak hanya akan membuat ulang aplikasi ini, tetapi juga diharapkan telah mempelajari cukup banyak tentang React Router untuk membangun hal-hal yang lebih keren dan lebih hebat.



Gambar 13.2 Aplikasi React Sederhana Yang Menggunakan React Router

13.2 MEMBANGUN APLIKASI

Hal pertama yang perlu kita lakukan adalah menyiapkan markup dan kode boilerplate untuk aplikasi kita. Buat dokumen HTML baru dan tambahkan konten berikut ke dalamnya:

```
<!DOCTYPE html>
<html>

<head>
  <title>React! React! React!</title>
  <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
  <script src="https://unpkg.com/react-dom@15.3.2/dist/react-
  dom.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
  core/5.8.23/browser.min
  .js"></script>

  <style>

  </style>
</head>

<body>

<div id="container">

</div>

<script type="text/babel">
```

```
    var destination = document.querySelector("#container");

ReactDOM.render(
  <div>
    Hello!
  </div>,
  destination
);
</script>
</body>

</html>
```

Titik awal ini hampir sama dengan apa yang telah Anda lihat pada semua contoh kami yang lain. Ini hanyalah aplikasi yang hampir kosong yang kebetulan memuat pustaka React dan React-DOM. Jika Anda melihat pratinjau apa yang Anda miliki di peramban, Anda akan melihat Hello! yang sangat sepi ditampilkan.

Catatan: Tetap Menjaga Kesederhanaan

Untuk saat ini, kami terus mengandalkan peramban kami untuk melakukan semua pekerjaan berat. Kami akan mempertimbangkan untuk mengubahnya dengan proses pembuatan yang "modern" nanti, jadi nikmatilah kesederhanaannya untuk saat ini.

Selanjutnya, karena React Router bukan bagian dari React itu sendiri, kita perlu menambahkan referensi ke dalamnya. Dalam markup kita, cari di mana kita memiliki referensi skrip yang ada dan tambahkan baris yang disorot berikut:

```
<script src=”https://unpkg.com/react@15.3.2/dist/react.js”></script>
<script src=”https://unpkg.com/react-dom@15.3.2/dist/react-dom.js”></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.23/browser.min.js"></script>
<script src="https://npmcdn.com/react-router/umd/ReactRouter.min.js"></script>
```

Dengan menambahkan baris yang disorot, kita memastikan pustaka React Router dimuat bersama pustaka inti React, ReactDOM, dan Babel. Pada titik ini, kita berada dalam kondisi yang baik untuk mulai membangun aplikasi kita dan memanfaatkan fungsionalitas hebat yang dihadirkan React Router.

Menampilkan Bingkai Awal

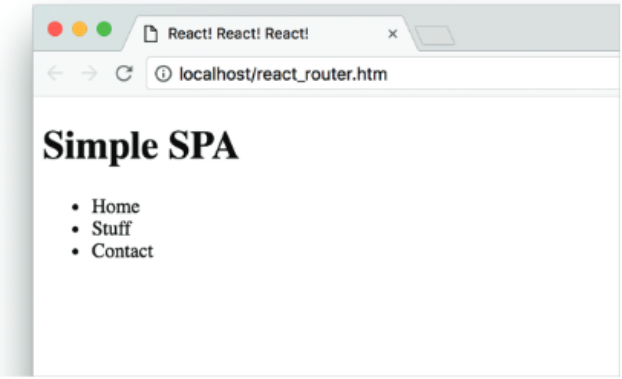
Saat membangun aplikasi satu halaman, akan selalu ada bagian dari halaman Anda yang akan tetap statis. Bagian statis ini, yang juga disebut bingkai aplikasi, bisa jadi hanya satu elemen HTML tak terlihat yang bertindak sebagai wadah untuk semua konten Anda, atau bisa juga menyertakan beberapa hal visual tambahan seperti header, footer, navigasi, dll. Dalam kasus kita, bingkai aplikasi kita akan melibatkan header navigasi dan area kosong tempat konten dimuat. Untuk menampilkannya, kita akan membuat komponen yang akan bertanggung jawab untuk ini. Di dalam tag skrip Anda tepat di atas panggilan ReactDOM.render Anda, lanjutkan dan tambahkan potongan JSX dan JavaScript berikut:

```
var App = React.createClass({
  render: function() {
    return (
      <div>
        <h1>Simple SPA</h1>
        <ul className="header">
          <li>Home</li>
          <li>Stuff</li>
          <li>Contact</li>
        </ul>
        <div className="content">
      </div>
    </div>
    )
  }
});
```

Setelah Anda menempelkannya, lihat apa yang kita miliki di sini. Yang kita miliki adalah komponen bernama App yang mengembalikan beberapa HTML. Untuk melihat seperti apa HTML ini, ubah panggilan ReactDOM.render Anda untuk merujuk ke komponen ini alih-alih menampilkan kata Hello! Lanjutkan dan buat perubahan yang disorot berikut ini:

```
ReactDOM.render(
  <div>
    <App/>
  </div>,
  destination
);
```

Setelah Anda melakukan ini, pratinjau aplikasi Anda di browser. Anda akan melihat versi judul aplikasi tanpa gaya dan beberapa item daftar (lihat Gambar 13.3). Saya tahu ini tidak terlihat mewah dan bergaya, tetapi tidak apa-apa untuk saat ini. Kita akan membahasnya nanti. Jika kita membahas lebih dalam, yang telah kita lakukan adalah membuat komponen bernama App dan menampilkannya melalui panggilan ReactDOM.render. Hal penting yang perlu diingat adalah tidak ada yang khusus untuk React Router di sini. SAMA SEKALI TIDAK ADA! Ini adalah React 101.



Gambar 13.3 Versi Tanpa Gaya

Mari kita perbaiki dengan menambahkan React Router. Ganti konten panggilan ReactDOM.render Anda dengan yang berikut:

```
ReactDOM.render(
  <ReactRouter.Router>
    <ReactRouter.Route path="/" component={App}>

    </ReactRouter.Route>
  </ReactRouter.Router>,
  destination
);
```

Abaikan sejenak betapa anehnya semua hal terlihat, dan cukup pratinjau aplikasi Anda di browser setelah Anda membuat perubahan ini. Jika semuanya berjalan dengan baik, Anda akan melihat komponen Aplikasi ditampilkan seperti yang Anda lihat sebelumnya. Sekarang, mari kita cari tahu mengapa demikian dengan mempelajari lebih lanjut tentang apa yang sebenarnya terjadi di sini. Di sinilah kita sedikit menyimpang dari konsep inti React dan mempelajari hal-hal yang khusus untuk React Router itu sendiri. Pertama, yang kita lakukan adalah menentukan komponen Router kita:

```
ReactDOM.render(
  <ReactRouter.Router>
    <ReactRouter.Route path="/" component={App}>

    </ReactRouter.Route>
  </ReactRouter.Router>,
  destination
);
```

Komponen Router merupakan bagian dari React Router API, dan tugasnya adalah menangani semua logika terkait perutean yang dibutuhkan aplikasi kita. Di dalam komponen ini, kita menentukan apa yang dikenal sebagai konfigurasi perutean. Itu adalah istilah khusus yang digunakan orang untuk menggambarkan pemetaan antara URL dan tampilan. Hal-hal spesifik itu ditangani oleh komponen lain yang disebut Rute:

```
ReactDOM.render(
  <ReactRouter.Router>
    <ReactRouter.Route path="/" component={App}>

    </ReactRouter.Route>
  </ReactRouter.Router>,
  destination
);
```

Komponen Route menggunakan beberapa properti yang membantu menentukan apa yang

akan ditampilkan di URL yang mana. Properti path menentukan URL yang ingin kita cocokkan. Dalam kasus ini, properti tersebut adalah root, alias /. Properti component memungkinkan Anda menentukan nama komponen yang ingin Anda tampilkan.

Untuk contoh ini, properti tersebut adalah komponen App kita. Jika menggabungkan semua ini, apa yang dikatakan Route ini adalah sebagai berikut: Jika URL yang Anda gunakan berisi root, lanjutkan dan tampilkan komponen App. Karena kondisi ini benar saat Anda melihat pratinjau aplikasi, Anda akan melihat hasil dari apa yang terjadi saat komponen App Anda dirender.

Menampilkan Halaman Beranda

Seperti yang dapat Anda lihat, React Router menyediakan semua fungsi routing ini dengan menggunakan konsep-konsep di React yang sudah Anda kenal yaitu komponen, properti, dan JSX. Apa yang kita miliki saat ini untuk menampilkan frame aplikasi kita adalah contoh yang bagus untuk ini. Sekarang, saatnya untuk melangkah lebih jauh.

Yang ingin kita lakukan selanjutnya adalah menentukan konten yang akan kita tampilkan sebagai bagian dari tampilan beranda kita. Untuk melakukan ini, kita akan membuat komponen bernama Home yang akan berisi markup yang ingin kita tampilkan. Tepat di atas tempat Anda mendefinisikan komponen App, tambahkan yang berikut ini:

```
var Home = React.createClass({
  render: function() {
    return (
      <div>
        <h2>HELLO</h2>
        <p>Cras facilisis urna ornare ex volutpat, et
        convallis erat elementum. Ut aliquam, ipsum vitae
        gravida suscipit, metus dui bibendum est, eget rhoncus nibh
        metus nec massa. Maecenas hendrerit laoreet augue
        nec molestie. Cum sociis natoque penatibus et magnis
        dis parturient montes, nascetur ridiculus mus.</p>
        <p>Duis a turpis sed lacus dapibus elementum sed eu lectus.</p>
      </div>
    );
  }
});
```

Seperti yang Anda lihat, komponen Home kita tidak melakukan hal khusus apa pun. Komponen ini hanya mengembalikan gumpalan HTML. Sekarang, yang ingin kita lakukan adalah menampilkan konten komponen Home kita saat halaman dimuat. Komponen ini setara dengan "halaman beranda" aplikasi kita. Cara melakukannya sederhana. Di dalam komponen App kita, kita memiliki div dengan nilai kelas konten. Kita akan memuat komponen Home kita di dalamnya. Solusi yang jelas mungkin terlihat seperti ini:

```
var App = React.createclass ({
  var App = React.createClass({
    render: function() {
```

```
      return (
        <div>
          <h1>Simple SPA</h1>
          <ul className="header">
            <li>Home</li>
            <li>Stuff</li>
            <li>Contact</li>
          </ul>
          <div className="content">
            <Home/>
        </div>
      </div>
      )
    }
});
```

Gambar 13.4 Peningkatan Fungsionalitas

Perhatikan bahwa kita mendefinisikan komponen Home di dalam div konten tersebut. Jika Anda melihat pratinjau aplikasi Anda, semuanya akan tampak berfungsi seperti yang diharapkan (lihat Gambar 13.4). Anda melihat tajuk navigasi Anda, lalu Anda melihat konten komponen Home kita.

Meskipun pendekatan ini berfungsi, sebenarnya itu adalah hal yang salah untuk dilakukan. Itu salah karena mempersulit keinginan kita untuk memuat konten lain saat pengguna menavigasi aplikasi kita. Kita pada dasarnya telah membuat kode keras aplikasi kita untuk hanya menampilkan komponen Home. Itu masalah, tetapi kita akan membahasnya nanti.

13.3 WAKTU PEMBERSIHAN SEMENTARA

Sebelum kita melanjutkan membuat kemajuan pada aplikasi kita, mari kita istirahat sejenak dan membuat beberapa perbaikan gaya pada apa yang telah kita buat sejauh ini.

Menambahkan CSS

Saat ini, aplikasi kita terlihat sangat polos dan seperti sesuatu yang berasal dari tahun 1800-an. Untuk memperbaikinya, kita akan mengandalkan teman lama kita, CSS. Di dalam tag gaya, lanjutkan dan tambahkan aturan gaya berikut:

```
body {
   background-color: #FFCC00;
   padding: 20px;
   margin: 0;
}
h1, h2, p, ul, li {
   font-family: Helvetica, Arial, sans-serif;
}
ul.header li {
   display: inline;
   list-style-type: none;
   margin: 0;
}
ul.header {
   background-color: #111;
   padding: 0;
}
ul.header li a {
   color: #FFF;
   font-weight: bold;
   text-decoration: none;
   padding: 20px;
   display: inline-block;
}
.content {
   background-color: #FFF;
   padding: 20px;
}
.content h2 {
   padding: 0;
   margin: 0;
}
.content li {
   margin-bottom: 10px;
}
```

Ya, kami menggunakan CSS dalam bentuk markupnya. Kami tidak melakukan pendekatan objek gaya sebaris yang kami lihat di Bab 4. Alasannya berkaitan dengan kemudahan. Komponen kami tidak akan digunakan kembali di luar aplikasi tertentu, dan kami benar-benar ingin memanfaatkan pewarisan CSS untuk meminimalkan markup yang diduplikasi.

Jika tidak, jika kami tidak menggunakan CSS biasa, kami akan berakhir dengan sekumpulan objek gaya raksasa yang ditetapkan untuk hampir setiap elemen dalam markup kami. Itu akan membuat orang yang paling sabar sekalipun merasa terganggu saat membaca

kode. Bagaimanapun, setelah Anda menambahkan semua CSS ini, aplikasi kami akan mulai terlihat jauh lebih baik (lihat Gambar 13.5).

Gambar 13.5 Gaya CSS Ditambahkan

Masih ada beberapa pekerjaan lagi yang harus dilakukan (misalnya, tautan navigasi kita menghilang di balik spanduk hitam), tetapi kita akan segera memperbaikinya.

Menghindari Awalan ReactRouter

Kita hanya memiliki satu tugas terkait pembersihan lagi sebelum kita kembali ke pemrograman terjadwal rutin kita. Pernahkah Anda memperhatikan bahwa setiap kali kita memanggil sesuatu yang didefinisikan oleh React Router API, kita mengawali sesuatu itu dengan kata ReactRouter?

```
<ReactRouter.Router>
  <ReactRouter.Route path="/" component={App}>

  </ReactRouter.Route>
</ReactRouter.Router>
```

Agak bertele-tele untuk mengulang setiap panggilan API yang kita buat, dan ini akan menjadi masalah yang lebih besar saat kita menyelami lebih jauh React Router API dan menggunakan lebih banyak hal di dalamnya. Perbaikan untuk ini melibatkan penggunaan trik ES6 baru di mana Anda dapat secara manual menentukan nilai mana yang akan secara otomatis diberi awalan. Di bagian atas tag skrip Anda, tambahkan yang berikut:

```
var { Router,
      Route,
      IndexRoute,
      IndexLink,
      Link } = ReactRouter;
```

Setelah Anda menambahkan kode ini, setiap kali Anda menggunakan salah satu nilai yang ditetapkan di dalam tanda kurung, awalan ReactRouter akan secara otomatis ditambahkan untuk Anda saat aplikasi Anda berjalan. Ini berarti, Anda sekarang dapat kembali ke metode ReactDOM.render dan menghapus awalan ReactRouter dari instans komponen Router dan Route kami:

```
ReactDOM.render(
  <Router>
    <Route path="/" component={App}>

    </Route>
  </Router>,
  destination
);
```

Jika Anda melihat pratinjau aplikasi Anda sekarang, tidak ada yang benar-benar berubah. Hasil akhirnya identik dengan yang Anda miliki sebelumnya. Satu-satunya perbedaan adalah markup kita sedikit lebih padat. Sekarang, sebelum kita melanjutkan, Anda mungkin bertanya-tanya mengapa daftar nilai yang akan secara otomatis diawali dengan ReactRouter berisi banyak hal di luar nilai Router dan Route yang telah kita gunakan dalam kode kita sejauh ini. Anggap nilai-nilai tambahan ini sebagai pratinjau bagian lain dari API React Router yang akan segera kita gunakan.

Menampilkan Halaman Beranda dengan Benar

Kami mengakhiri beberapa bagian yang lalu dengan mengatakan bahwa cara kita saat ini menampilkan halaman beranda tidak benar. Meskipun Anda mendapatkan hasil yang diinginkan saat halaman kita dimuat, pendekatan ini tidak benar-benar memudahkan kita untuk memuat apa pun selain halaman beranda saat pengguna menavigasi.

Panggilan ke komponen Home kita dikodekan secara permanen di dalam App. Solusi yang benar melibatkan membiarkan React Router menangani komponen mana yang akan dipanggil tergantung pada struktur URL Anda saat ini. Ini melibatkan penumpukan komponen Route di dalam komponen Route untuk mendefinisikan pemetaan URL-ke-tampilan dengan lebih baik. Kembali ke metode ReactDOM. render, dan buat perubahan yang disorot berikut:

```
ReactDOM.render(
  <Router>
    <Route path="/" component={App}>
      <IndexRoute component={Home}/>
    </Route>
  </Router>,
  destination
);
```

Di dalam elemen Route root, kita mendefinisikan elemen Route lain bertipe IndexRoute dan

menyetel tampilannya menjadi komponen Home. Ada satu perubahan lagi yang perlu kita buat. Di dalam komponen App, hapus panggilan ke komponen Home dan ganti dengan baris yang disorot berikut:

```
var App = React.createClass({
  render: function() {
    return (
      <div>
        <h1>Simple SPA</h1>
        <ul className="header">
          <li>Home</li>
          <li>Stuff</li>
          <li>Contact</li>
        </ul>
        <div className="content">
          {this.props.children}
        </div>
      </div>
    )
  }
});
```

Jika Anda melihat pratinjau halaman Anda sekarang, Anda akan tetap melihat konten Beranda Anda ditampilkan. Perbedaannya kali ini adalah bahwa kita menampilkan konten Beranda dengan benar dengan cara yang tidak menghalangi konten lain untuk ditampilkan sebagai gantinya. Hal ini disebabkan oleh dua hal:

1. Apa yang ditampilkan di dalam Aplikasi dikontrol oleh hasil turunan this.props. alih-alih komponen yang dikodekan secara kaku.
2. Elemen Rute kita di dalam ReactDOM.render berisi elemen IndexRoute yang tujuan utamanya adalah untuk mendeklarasikan komponen mana yang akan ditampilkan saat aplikasi Anda pertama kali dimuat.

Semua ini mungkin tampak lebih aneh daripada yang Anda harapkan beberapa saat yang lalu, tetapi semuanya akan lebih masuk akal saat kita menggunakan berbagai API ini lebih banyak di bagian berikut.

13.4 MEMBUAT TAUTAN NAVIGASI

Saat ini, kita hanya menyiapkan bingkai dan tampilan beranda. Tidak ada hal lain yang dapat dilakukan pengguna di sini selain hanya melihat apa yang telah kita tetapkan sebagai beranda. Mari kita perbaiki itu dengan membuat beberapa tautan navigasi. Lebih khusus lagi, mari kita buat tautan pada elemen navigasi yang sudah kita miliki:

```
var App = React.createClass({
  render: function() {
    return (
      <div>
```

```
          <h1>Simple SPA</h1>
          <ul className="header">
            <li>Home</li>
            <li>Stuff</li>
            <li>Contact</li>
            </ul>
            <div className="content">
              {this.props.children}
            </div>
          </div>
        )
      }
    });
```

Jika Anda tidak yakin mengapa elemen-elemen ini tidak terlihat saat Anda melihat pratinjau halaman, itu karena elemen-elemen tersebut menyatu dengan latar belakang hitam setelah kita menambahkan CSS. Tidak masalah. Kita akan memperbaikinya dalam beberapa langkah, tetapi pertama-tama mari kita bahas tentang bagaimana kita akan mengubah elemen-elemen ini menjadi tautan.

Cara Anda menentukan tautan navigasi di React Router bukanlah dengan langsung menggunakan tag a yang telah teruji dan memasukkan jalur melalui atribut href. Sebaliknya, Anda menentukan tautan navigasi Anda menggunakan komponen Link React Router yang mirip dengan tag a tetapi menawarkan lebih banyak fungsi. Untuk melihat komponen Link beraksi, lanjutkan dan ubah elemen navigasi yang ada agar terlihat seperti baris yang disorot berikut:

```
var App = React.createClass({
  render: function() {
    return (
      <div>
        <h1>Simple SPA</h1>
        <ul className="header">
          <li><Link to="/">Home</Link></li>
          <li><Link to="/stuff">Stuff</Link></li>
          <li><Link to="/contact">Contact</Link></li>
        </ul>
        <div className="content">
          {this.props.children}
        </div>
      </div>
    )
```

Perhatikan apa yang telah dilakukan di sini. Komponen Link kita menetapkan prop yang disebut to. Prop ini menetapkan nilai URL yang akan kita tampilkan di address bar. Secara tidak langsung, prop ini juga menetapkan lokasi yang akan kita beri tahu React Router bahwa kita sedang menavigasi secara virtual. Link Home kita membawa pengguna ke root (/), link Stuff membawa pengguna ke lokasi bernama stuff, dan link Contact membawa pengguna ke lokasi

bernama contact.

Jika Anda melihat pratinjau halaman dan mengklik link (yang sekarang akan terlihat karena CSS untuk link tersebut telah aktif), Anda tidak akan melihat tampilan baru apa pun. Anda hanya akan melihat konten Home karena hanya itu yang telah kita tetapkan sebelumnya. Dengan demikian, Anda dapat melihat URL diperbarui di address bar. Anda akan melihat halaman Anda saat ini diikuti oleh #/contact, #/stuff, atau #/ tergantung pada link mana yang Anda klik. Oh, Anda juga akan melihat hash acak yang ditambahkan setelah URL. Itu kemajuan!

Menambahkan Tampilan Barang dan Kontak

Aplikasi kita perlahan-lahan mulai terbentuk atau akan benar-benar mendekati bentuk akhirnya saat kita selesai dengan bagian ini! Apa yang akan kita lakukan selanjutnya adalah menentukan komponen untuk tampilan Barang dan Kontak yang telah kita tautkan sebelumnya. Pada kode Anda tepat di bawah tempat Anda meletakkan komponen Beranda, lanjutkan dan tambahkan yang berikut ini:

```
var Contact = React.createClass({
  render: function() {
    return (
      <div>
        <h2>GOT QUESTIONS?</h2>
        <p>The easiest thing to do is post on
        our <a href="http://forum.kirupa.com">forums</a>.
        </p>
      </div>
    );
  }
});

var Stuff = React.createClass({
  render: function() {
    return (
      <div>
        <h2>STUFF</h2>
        <p>Mauris sem velit, vehicula eget sodales vitae,
        rhoncus eget sapien:</p>
        <ol>
          <li>Nulla pulvinar diam</li>
          <li>Facilisis bibendum</li>
          <li>Vestibulum vulputate</li>
          <li>Eget erat</li>
          <li>Id porttitor</li>
        </ol>
      </div>
    );
  }
});
```

Yang baru saja kita tambahkan adalah komponen Stuff dan Contact yang hanya merender HTML. Yang tersisa bagi kita adalah memperbarui konfigurasi perutean kita untuk

menyertakan kedua komponen ini dan menampilkannya di URL yang sesuai. Dalam metode ReactDOM.render kita, lanjutkan dan tambahkan dua baris yang disorot berikut:

```
ReactDOM.render(
  <Router>
    <Route path="/" component={App}>
      <IndexRoute component={Home}/>
        <Route path="stuff" component={Stuff} />
        <Route path="contact" component={Contact} />
      </Route>
    </Router>,
    destination
    );
```

Yang kami lakukan di sini adalah memperbarui logika perutean kami untuk menampilkan komponen Barang jika URL berisi kata barang dan untuk menampilkan komponen Kontak jika URL berisi kata kontak. Jika Anda melihat pratinjau halaman Anda sekarang, klik tautan Barang dan Kontak. Jika semuanya berjalan lancar, Anda akan melihat tampilan ini dimuat di dalam bingkai aplikasi kami saat Anda menavigasi ke sana.

Catatan: Sedikit Tentang Pencocokan Rute

Konfigurasi rute kami tidak lebih dari serangkaian aturan yang menentukan apa yang harus dilakukan saat URL cocok dengan kondisi yang telah kami tetapkan. Istilah keren untuk itu adalah pencocokan rute. Heuristik React Router yang digunakan untuk mencocokkan URL dijelaskan sepenuhnya dalam dokumentasi React Router, tetapi untuk kasus kami, kami memiliki rute bersarang sederhana tempat Anda dapat memiliki beberapa hal yang dapat cocok pada saat yang sama. Rute luar kami cocok jika URL berisi /. Rute dalam kami kemudian cocok jika URL kebetulan berisi barang atau kontak.

Artinya sederhana. Untuk setiap rute yang cocok, komponen yang Anda tentukan untuk ditampilkan akan muncul. Saat Anda menavigasi ke halaman seperti /stuff, komponen Aplikasi akan ditampilkan karena / ada di URL. Komponen Stuff kemudian ditampilkan karena jalur untuk stuff juga ada di URL. Itulah sebabnya saat kita menavigasi ke halaman Stuff atau Kontak, kita melihatnya di samping bingkai kita. Anda juga dapat memiliki rute yang sangat bertingkat. Lihat konfigurasi berikut:

```
ReactDOM.render(
<Router>
  <Route path="/" component={App}>
    <IndexRoute component={Home} />
    <Route path="stuff" component={Stuff}>
      <Route path="blah" component={MyBlah}/>
    </Route>
    <Route path="contact" component={Contact} />
  </Route>
</Router>,
destination);
```

Dalam contoh ini, perhatikan bahwa elemen Route kita yang path-nya adalah stuff sekarang berisi route bersarang untuk path yang berisi blah. Ini berarti jika Anda kebetulan memiliki URL yang merupakan /stuff/ blah, komponen MyBlah akan ditampilkan sebagai tambahan pada komponen Stuff dan komponen App dari rute induk yang cocok.

Dengan menyarangkan route dan mengikuti aturan pencocokan route (https://github.com/reactjs/react-router/blob/master/docs/guides/RouteMatching.md), Anda dapat menampilkan tampilan kustom tergantung pada berbagai pengaturan URL yang dapat Anda tampilkan di app Anda agar pengguna dapat menavigasi ke sana.

13.5 MEMBUAT TAUTAN AKTIF

Hal terakhir yang akan kita bahas adalah sesuatu yang sangat meningkatkan kegunaan aplikasi kita. Bergantung pada halaman mana yang sedang Anda tampilkan, kita akan menyorot tautan tersebut dengan latar belakang biru. Misalnya, Gambar 13.6 adalah tampilan aplikasi kita saat konten Stuff ditampilkan.

Gambar 13.6 Konten Stuff

Cara Anda melakukannya di React Router adalah dengan menyetel properti bernama activeClassName pada instance Link Anda dengan nama kelas CSS yang akan disetel saat tautan tersebut sedang aktif. Untuk melakukannya, kembali ke komponen App Anda dan buat perubahan yang disorot:

```
var App = React.createClass({
  render: function() {
    return (
```

```
        <div>
          <h1>Simple SPA</h1>
          <ul className="header">
            <li><Link to="/" activeClassName="active">Home</Link></li>
            <li><Link to="/stuff" activeClassName="active">Stuff</Link></li>
            <li><Link to="/contact" activeClassName="active">Contact</Link></li>
          </ul>
          <div className="content">
            {this.props.children}
          </div>
        </div>
      )
    }
});
```

Kita tentukan prop activeClassName dan atur ke nilai active. Ini memastikan bahwa setiap kali Kita tentukan prop activeClassName dan atur ke nilai active. Ini memastikan bahwa setiap kali tautan diklik (dan jalurnya menjadi aktif), atribut kelas elemen tautan pada waktu proses akan diatur ke nilai active. Untuk memastikan tautan active kita ditata secara berbeda, lanjutkan dan tambahkan CSS berikut:

```
.active {
  background-color: #0099FF;
}
```

Jika Anda melihat pratinjau aplikasi Anda sekarang, klik salah satu tautan. Perhatikan bahwa tautan aktif (dan tautan Beranda) ditampilkan dengan latar belakang biru. Namun, kita belum selesai. Tautan Beranda kita selalu disorot. Tautan ini seharusnya hanya disorot saat kita memuat halaman Beranda untuk pertama kalinya atau secara eksplisit menavigasi ke tautan Beranda itu sendiri.

Untuk memperbaikinya, kita perlu mengubah cara kita menautkan ke konten Beranda kita. Alih-alih menentukan konten Beranda kita dengan elemen Tautan, kita akan menggantinya dengan elemen IndexLink. Lanjutkan dan buat perubahan ini:

```
var App = React.createClass({
  render: function() {
    return (
      <div>
      <h1>Simple SPA</h1>
        <ul className="header">
        <li><IndexLink to="/" activeClassName="active">Home</IndexLink></li>
        <li><Link to="/stuff" activeClassName="active">Stuff</Link></li>
        <li><Link to="/contact" activeClassName="active">Contact</Link></li>
        </ul>
      <div className="content">
        {this.props.children}
      </div>
      </div>
```

```
        )
    }
});
```

Setelah elemen navigasi Beranda Anda diwakili oleh IndexLink alih-alih Link, pratinjau aplikasi Anda lagi. Kali ini, saat aplikasi dimuat, Anda akan melihat bahwa tautan Beranda Anda memiliki latar belakang biru yang keren secara default. Saat Anda menavigasi ke halaman Barang atau Kontak, tautan Beranda tidak lagi memiliki sorotan yang diterapkan. Dan dengan ini, aplikasi Anda sebagian besar sudah siap digunakan!

13.6 KESIMPULAN

Saat ini, kita telah membahas sebagian besar fungsionalitas keren yang dimiliki React Router untuk membantu Anda membangun aplikasi satu halaman. Ini tidak berarti bahwa tidak ada hal yang lebih menarik untuk Anda manfaatkan. Aplikasi kami cukup sederhana dengan tuntutan yang sangat sederhana pada fungsionalitas perutean apa yang perlu kami terapkan. Ada banyak hal lain yang disediakan React Router, jadi jika Anda membangun aplikasi satu halaman yang lebih kompleks daripada yang telah kita lihat sejauh ini, Anda harus benar-benar meluangkan waktu sore untuk melihat dokumentasi React Router lengkap (https://github.com/reactjs/react-router/) dan contoh-contohnya.

BAB 14
MEMBANGUN APLIKASI DAFTAR TUGAS

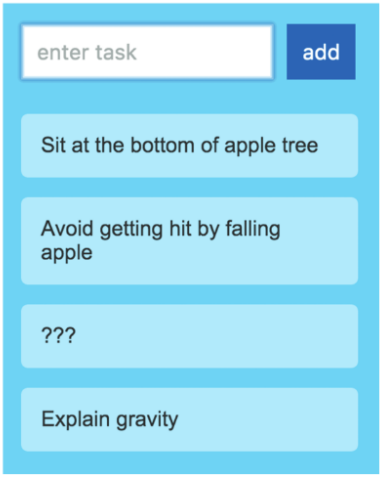
Jika membuat contoh Hello, World! adalah perayaan atas keberhasilan Anda dalam menggunakan React, membuat aplikasi Daftar Tugas yang sempurna adalah perayaan atas keberhasilan Anda dalam menguasai React! Dalam bab ini, kami menggabungkan banyak konsep dan teknik yang telah Anda pelajari untuk membuat sesuatu yang berfungsi sebagai berikut:

https://www.kirupa.com/react/examples/todo.htm

Anda memulai dengan aplikasi kosong yang memungkinkan Anda memasukkan tugas untuk nanti (lihat Gambar 14.1).

Gambar 14.1 Aplikasi Kosong Dengan Entri Tugas

Cara kerja aplikasi Daftar Tugas ini cukup mudah. Ketik tugas atau apa pun yang Anda inginkan ke dalam kolom teks dan tekan Tambah (atau tekan Enter/Return). Setelah Anda mengirimkan tugas, Anda akan melihatnya muncul sebagai entri. Anda dapat terus menambahkan tugas untuk menambahkan entri tambahan dan menampilkan semuanya (lihat Gambar 14.2).



Gambar 14.2 Anda Dapat Menambahkan Tugas Dan Menampilkannya.

Cukup mudah, bukan? Di bagian berikut, kita membangun aplikasi ini dari awal dan mempelajari (dengan sangat teliti) cara kerja aplikasi ini.

14.1 MEMBUAT DOKUMEN HTML

Sekarang, Anda sudah tahu caranya. Kita perlu titik awal, jadi lanjutkan dan buat dokumen HTML baru. Di dalamnya, tambahkan konten berikut ke dalamnya:

```
<!DOCTYPE html>
<html>

<head>
<title>React! React! React!</title>
   <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
   <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></script>
   <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
   core/5.8.23/browser.min.js"></script>

   <style>

   </style>
</head>

<body>
   <div id="container">
   </div>
   <script type="text/babel">
      var destination = document.querySelector("#container");

      ReactDOM.render(
         <div>
            Hello!
         </div>,
         destination
      );
```

```
    </script>
</body>

</html>
```

Jika Anda melihat pratinjau semua ini di browser, Anda akan melihat kata Hello! muncul. Jika Anda melihatnya, berarti Anda dalam kondisi yang baik. Saatnya mulai membangun aplikasi Daftar Tugas kita!

14.2 MEMBUAT UI

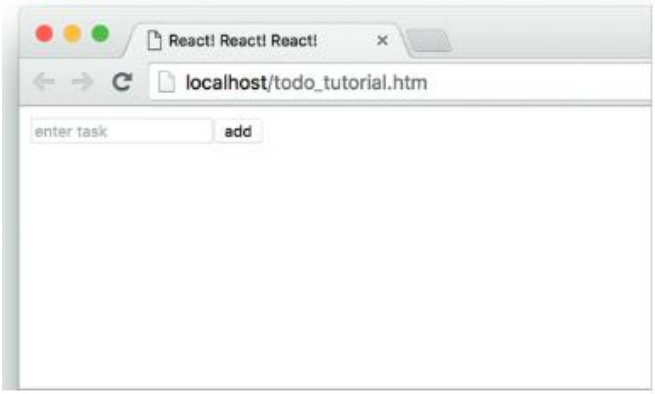
Saat ini, aplikasi kita tidak melakukan banyak hal. Kita akan memperbaikinya dengan terlebih dahulu mengaktifkan dan menjalankan berbagai elemen UI. Itu tidak terlalu rumit untuk aplikasi kita! Hal pertama yang akan kita lakukan adalah membuat kolom input dan tombol muncul. Ini semua dilakukan dengan menggunakan elemen div, form, input, dan button! Semua itu akan ada di dalam komponen yang akan kita sebut TodoList. Lanjutkan dan tambahkan kode berikut di atas tempat Anda meletakkan metode ReactDOM.render:

```
var TodoList = React.createClass({
  render: function() {
    return (
      <div className="todoListMain">
        <div className="header">
          <form>
            <input placeholder="enter task">
            </input>
            <button type="submit">add</button>
          </form>
        </div>
      </div>
    );
  }
});
```

Di dalam metode ReactDOM.render, kita perlu memanggil komponen TodoList yang baru ditambahkan untuk merendernya. Silakan ganti JSX yang sudah ada dengan yang berikut ini:

```
ReactDOM.render(
  <div>
    <TodoList/>
  </div>,
  destination
);
```

Simpan perubahan Anda dan pratinjau apa yang Anda miliki saat ini di peramban Anda. Anda akan melihat sesuatu yang tampak seperti Gambar 14.3.

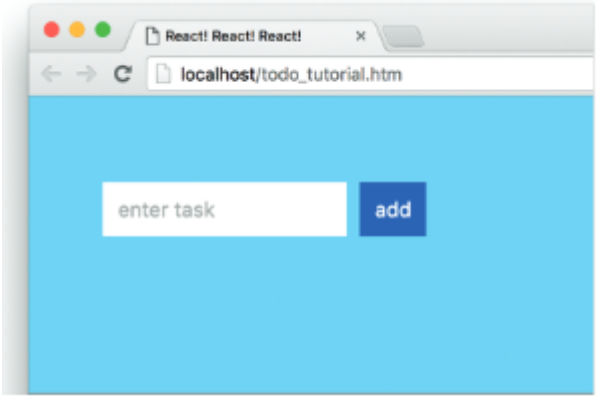


Gambar 14.3 Apa Yang Seharusnya Anda Lihat Di Peramban

Jika Anda terkejut dengan apa yang Anda lihat, luangkan waktu sejenak untuk melihat JSX yang kami definisikan di dalam komponen TodoList. Seharusnya tidak ada yang mengejutkan di sana. Kami baru saja mendefinisikan beberapa elemen HTML yang tampak SANGAT membosankan. Berbicara tentang itu, mari kita buat elemen HTML kita tampak tidak membosankan dengan memperkenalkannya ke beberapa CSS! Di dalam blok gaya Anda, tambahkan yang berikut ini:

```
body {
  padding: 50px;
  background-color: #66CCFF;
  font-family: sans-serif;
}
.todoListMain .header input {
  padding: 10px;
  font-size: 16px;
  border: 2px solid #FFF;
}
.todoListMain .header button {
  padding: 10px;
  font-size: 16px;
  margin: 10px;
  background-color: #0066FF;
  color: #FFF;
  border: 2px solid #0066FF;
}

.todoListMain .header button:hover {
  background-color: #003399;
  border: 2px solid #003399;
  cursor: pointer;
}
```

Setelah Anda menambahkan semua ini, pratinjau aplikasi Anda sekarang. Karena elemen HTML kita memiliki nilai className yang sesuai, CSS kita akan aktif dan contoh kita sekarang akan terlihat seperti Gambar 14.4.

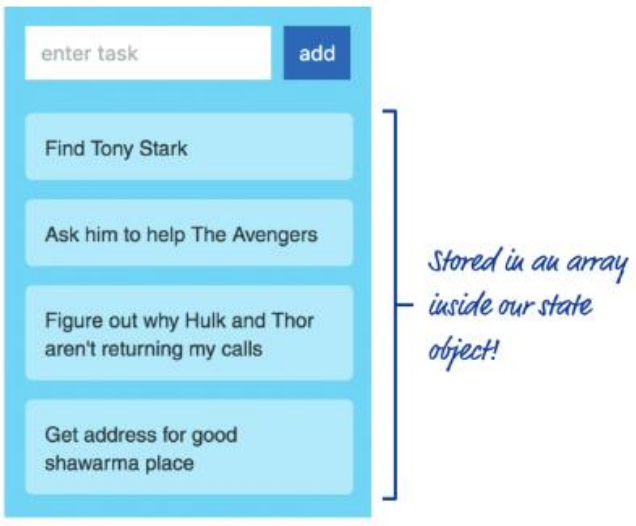


Gambar 14.4 Contoh Yang Ditingkatkan

Pada titik ini, aplikasi kita terlihat cukup bagus. Aplikasi ini tidak melakukan banyak hal, tetapi setidaknya kita membuat kemajuan. Di bagian berikutnya, kita akan mulai membuat aplikasi kita benar-benar melakukan sesuatu.

14.3 MEMBUAT FUNGSIONALITAS

Implementasi aktual dari fungsionalitas aplikasi Daftar Tugas kita tidak segila yang Anda kira. Mari kita lihat secara umum cara kerjanya. Bagian data yang paling penting adalah teks yang Anda masukkan ke dalam bidang teks. Setiap kali Anda memasukkan beberapa teks dan mengirimkan formulir, teks tersebut akan ditampilkan secara visual dalam daftar di bawah setiap bagian teks sebelumnya yang Anda kirimkan. Sejauh ini, ini masuk akal, bukan? Semua ini dilakukan hanya dengan memanfaatkan fungsionalitas status React. Di dalam objek status kita, kita memiliki array yang bertanggung jawab untuk menyimpan semua yang Anda masukkan (lihat Gambar 14.5).



Gambar 14.5 Tugas Kita Disimpan Dalam Suatu Array.

Saya tahu. Tidak terlalu menarik :-(Setiap kali array item ini diperbarui dengan teks baru yang Anda kirimkan, kami memperbarui apa yang Anda lihat dengan teks yang baru dikirimkan. Pekerjaan selanjutnya hanya seputar pengaturan acara dan pengendali acara untuk memastikan kita dapat mengirimkan formulir dan mengetahui dengan tepat teks apa yang akan ditambahkan ke array item kita. Di bagian berikut, kita akan mengubah semua bahasa Inggris yang telah kita lihat di sini menjadi JavaScript dan JSX bercitarasa React!

14.4 MENGINISIALISASI OBJEK STATE

Hal pertama yang akan kita lakukan adalah menginisialisasi objek state kita dengan array yang akan bertanggung jawab untuk menyimpan semua teks yang dikirimkan. Di dalam komponen TodoList kita, tambahkan baris-baris yang disorot berikut:

```
var TodoList = React.createClass({
  getInitialState: function() {
    return {
      items: []
    };
  },
  render: function() {
    return (
      <div className="todoListMain">
        <div className="header">
          <form>
            <input placeholder="enter task">
            </input>
            <button type="submit">add</button>
          </form>
        </div>
      </div>
    );
  }
});
```

Yang kita lakukan di sini adalah menentukan metode getInitialState yang dipanggil sebelum komponen kita dirender. Di dalam metode itu, kita membuat array kosong bernama items yang kemudian dapat kita akses melalui this.state.items dari mana saja di dalam komponen ini.

14.5 MENANGANI PENGIRIMAN FORMULIR

Kita menambahkan item baru ke daftar tugas kita saat Anda mengirimkan formulir baik dengan menekan tombol Tambah atau menekan Enter/Return pada papan ketik Anda. Perilaku ini sebagian besar sudah ada dalam HTML dan peramban kita tahu semua tentang cara menanganinya. Kita tidak perlu menulis kode khusus untuk menangani tombol Enter/Return atau mendengarkan penekanan tombol Tambah. Satu-satunya hal yang perlu kita khawatirkan adalah menangani apa yang terjadi saat formulir benar-benar dikirimkan.

Untuk melakukannya, kita mendengarkan acara onSubmit pada elemen formulir kita.

Acara ini dipicu setiap kali formulir dikirimkan, dan itu termasuk menekan tombol Enter/Return atau mengutak-atik elemen apa pun yang memiliki atribut tipe submit di dalamnya. Saat formulir dikirimkan dan acara itu didengar, kita perlu memanggil pengendali acara. Mari kita beri nama addItem pada event handler tersebut. Gabungkan semua ini, di dalam fungsi render komponen TodoList Anda, buat perubahan yang disorot berikut ini:

```
render: function() {
  return (
    <div className="todoListMain">
      <div className="header">
        <form onSubmit={this.addItem}>
          <input placeholder="enter task">
          </input>
          <button type="submit">add</button>
        </form>
      </div>
    </div>
  );
}
```

Seperti yang kami harapkan, kami cukup menautkan event onSubmit elemen formulir kami ke pengendali event addItem. Pengendali event ini tidak ada, tetapi kami akan memperbaikinya dengan menambahkan baris-baris yang disorot berikut:

```
var TodoList = React.createClass({
  getInitialState: function() {
    return {
      items: []
    };
  },
  addItem: function(e) {

  },
  render: function() {
    return (
      <div className="todoListMain">
        <div className="header">
        <form onSubmit={this.addItem}>
        <input placeholder="enter task">
        </input>
        <button type="submit">add</button>
        </form>
      </div>
    </div>
    );
  }
});
```

Penangan/fungsi acara addItem kita tidak berfungsi banyak saat ini, tetapi yang penting adalah

ia ada! Selanjutnya, kita akan memperbaiki bagian yang tidak berfungsi banyak.

14.6 MENGISI STATE KITA

Saat ini, objek state komponen TodoList kita berisi array items. Yang perlu kita lakukan adalah mengisi array ini dengan teks yang Anda masukkan ke dalam kolom input. Itu berarti kita memerlukan cara untuk mengakses elemen input kita dari dalam React. Cara kita melakukannya adalah dengan menetapkan atribut ref (seperti yang Anda lihat di Bab 12) pada elemen input kita dan menyimpan referensi ke elemen HTML yang dihasilkan. Di dalam metode render komponen TodoList kita, tambahkan baris berikut:

```
render: function() {
  return (
    <div className="todoListMain">
      <div className="header">
        <form onSubmit={this.addItem}>
          <input ref={(a) => this._inputElement = a}
            placeholder="enter task">
          </input>
          <button type="submit">add</button>
        </form>
      </div>
    </div>
  );
}
```

Saat kode yang disorot ini dijalankan, yang terjadi segera setelah komponen ini dipasang, properti _inputElement akan menyimpan referensi ke elemen input yang dihasilkan. Setelah selesai, kita dapat memperlakukan elemen ini seperti elemen DOM yang mungkin kita temukan menggunakan querySelector atau fungsi setara di dunia non-React. Yang akan kita lakukan selanjutnya adalah mengisi array item kita! Lanjutkan dan ubah metode addItem dengan menambahkan baris berikut:

Lanjutkan dan ubah metode addItem dengan menambahkan baris berikut:

```
addItem: function(e) {
  var itemArray = this.state.items;

  itemArray.push(
    {
      text: this._inputElement.value,
      key: Date.now()
    }
  );

  this.setState({
    items: itemArray
  });
```

```
    e.preventDefault();
}
```

Ini terlihat seperti banyak kode yang baru saja Anda tambahkan, tetapi yang kita lakukan di sini hanyalah memasukkan ke dalam JavaScript tujuan yang kita nyatakan sebelumnya untuk mengisi array item kita dengan teks dari kolom input kita. Mari kita bahas kode ini secara lebih terperinci. Hal pertama yang kita lakukan adalah membuat array yang disebut itemArray yang menyimpan referensi ke properti item objek status kita:

```
var itemArray = this.state.items;
```

Setelah kita memiliki array ini, kita tambahkan entri teks yang baru saja kita kirimkan dari elemen input kita:

```
itemArray.push(
  {
    text: this._inputElement.value,
    key: Date.now()
  }
);
```

Perhatikan bahwa kita tidak hanya menambahkan entri teks dari elemen input kita. Sebaliknya, kita menambahkan objek yang terdiri dari properti teks dan kunci. Properti teks menyimpan nilai teks elemen input kita. Properti kunci menyimpan waktu saat ini. Ini terdengar seperti hal yang aneh untuk dilakukan, tetapi seperti yang Anda ingat dari Bab 9, tujuannya adalah agar nilai kunci ini unik untuk setiap entri yang dikirimkan. Ini penting karena (peringatan spoiler!) kita akan menggunakan data dalam array ini untuk akhirnya menghasilkan beberapa elemen UI.

Nilai kunci inilah yang akan digunakan React untuk mengidentifikasi secara unik setiap elemen UI yang dihasilkan, jadi dengan menghasilkan kunci menggunakan Date.now(), kita memastikan tingkat keunikan tertentu. Karena ini adalah detail yang penting (namun mudah diabaikan), kita akan meninjau kembali semua ini dalam beberapa saat. Bagaimanapun, kembali ke jalur, setelah kita selesai dengan itemArray, yang tersisa adalah menyetel properti item objek status kita ke dalamnya:

```
this.setState({
  items: itemArray
});
```

Hampir selesai! Hal terakhir yang kita lakukan dalam metode ini adalah sebagai berikut:

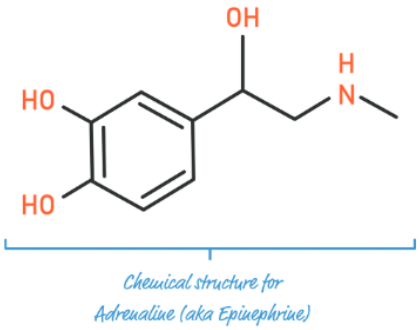
```
    e.preventDefault();
```

Metode preventDefault memastikan kita mengganti peristiwa onSubmit default. Alasan kita melakukan ini agak tidak jelas, tetapi tujuannya adalah untuk memastikan hal berikut: yang ingin kita lakukan saat mengirimkan formulir adalah memanggil metode addItem. Jika kita tidak menghentikan perilaku default, aplikasi kita akan memanggil addItem dengan benar seperti yang diinginkan saat kita mengirimkan formulir.

Ini juga akan memicu perilaku POST default browser kita yang jelas tidak kita inginkan. Dengan menghentikan peristiwa onSubmit agar tidak melakukan perilaku default, kita mendapatkan perilaku yang diinginkan untuk memanggil metode addItem tanpa efek samping yang tidak diinginkan seperti tindakan POST yang tidak perlu yang dapat menyegarkan halaman Anda.

14.7 MENAMPILKAN TUGAS

Kita hampir selesai di sini! Hal terakhir yang akan kita lakukan adalah memvisualisasikan tugas yang saat ini ada di dalam array item objek status kita.



Gambar 14.6 Adrenalin

Ini akan melibatkan pembuatan komponen baru yang disebut TodoItems, menyebarkan beberapa properti, menggunakan fungsi peta, dan melakukan hal-hal hebat lainnya yang memicu adrenalin (Gambar 14.6). Pokoknya, hal pertama yang akan kita lakukan adalah mendefinisikan komponen TodoItems. Dalam kode Anda, tepat di atas tempat Anda mendefinisikan komponen TodoList, lanjutkan dan tambahkan yang berikut ini:

```
var TodoItems = React.createClass({
  render: function() {

  }
});
```

Tidak ada yang terjadi saat ini, tetapi tidak apa-apa. Selanjutnya, yang akan kita lakukan adalah memanggil komponen ini dari dalam metode render komponen TodoList. Tidak hanya itu, kita akan menentukan properti dan meneruskan objek status komponen TodoList yang berisi array item kita. Melakukan semua ini sangat mudah, jadi lanjutkan dan tambahkan baris yang disorot berikut ke metode render komponen TodoList Anda:

```
render: function() {
  return (
    <div className="todoListMain">
      <div className="header">
        <form onSubmit={this.addItem}>
        <input ref={(a) => this._inputElement = a}
          placeholder="enter task">
        </input>
        <button type="submit">add</button>
      </form>
    </div>
    <TodoItems entries={this.state.items}/>
</div>
);
}
```

Yang kami lakukan di sini adalah membuat instance komponen TodoItems dan meneruskan properti status items ke prop yang disebut entri. Pada titik ini, jika Anda menjalankan aplikasi di browser, tidak akan terjadi apa pun yang terlihat. Komponen TodoItems kami siap untuk dirender, dan memiliki akses ke semua tugas yang dikirimkan. Satu-satunya masalah adalah komponen tersebut tidak benar-benar melakukan apa pun dengan semua itu, tetapi kami akan memperbaikinya nanti. Kembali ke komponen TodoItems, hal pertama yang akan kami lakukan adalah membuat variabel baru untuk menyimpan array tugas yang kami masukkan. Untuk melakukannya, tambahkan baris yang disorot berikut:

```
var TodoItems = React.createClass({
  render: function() {
    var todoEntries = this.props.entries;
  }
});
```

Kita baru saja menambahkan variabel yang disebut todoEntries, dan variabel tersebut menyimpan nilai dari properti entri yang kita masukkan berdasarkan nilai this.state.items dari komponen TodoList. Keren! Sekarang, variabel todoEntries kita menyimpan array yang berisi sekumpulan objek yang masing-masing menyimpan tugas dan kunci. Yang tersisa hanyalah membuat elemen HTML yang akan digunakan untuk menampilkan data kita. Pada langkah pertama untuk mencapainya, tambahkan baris kode yang disorot berikut untuk membuat elemen li:

```
var TodoItems = React.createClass({
  render: function() {
    var todoEntries = this.props.entries;
    function createTasks(item) {
      return <li key={item.key}>{item.text}</li>
    }

    var listItems = todoEntries.map(createTasks);
```

```
    }
});
```

Kami menggunakan fungsi peta untuk mengulang setiap item di dalam todoEntries dan memanggil fungsi createTasks untuk membuat elemen daftar untuk setiap entri:

```
function createTasks(item) {
  return <li key={item.key}>{item.text}</li>
}
```

Untuk menegaskan kembali poin yang kami sampaikan sebelumnya, karena elemen-elemen daftar ini dibuat secara dinamis, kami perlu membantu React melacaknya dengan menentukan atribut kunci dan memberikan masing-masing nilai yang unik. Kami telah memecahkan bagian masalah ini saat kami menyimpan tugas-tugas kami pada awalnya, seperti yang Anda ingat:

```
itemArray.push(
  {
    text: this._inputElement.value,
    key: Date.now()
  }
);
```

Karena perencanaan kita sebelumnya, kita akan mengambil jalan pintas sekarang dengan menetapkan atribut kunci kita nilai item.key yang sudah dimiliki setiap item dalam array todoEntries kita. Konten elemen daftar kita yang terlihat hanyalah nilai teks yang disimpan oleh item.text. Tidak ada penjelasan tambahan yang diperlukan tentang cara kita menggunakannya. Cukup menyegarkan, bukan?

Dengan menggabungkan semua ini, kumpulan elemen daftar ini diproses dan disimpan sepenuhnya oleh variabel listItems kita. Yang tersisa pada titik ini adalah beralih dari elemen daftar di dalam array ke elemen daftar yang ditampilkan di layar. Untuk melakukannya, lanjutkan dan tambahkan baris yang disorot berikut:

```
var TodoItems = React.createClass({
  render: function() {
    var todoEntries = this.props.entries;

    function createTasks(item) {
      return <li key={item.key}>{item.text}</li>
    }
    var listItems = todoEntries.map(createTasks);
    return (
      <ul className="theList">
        {listItems}
      </ul>
    );
  }
});
```

Gambar 14.7 Elemen Daftar Untuk Item Daftar

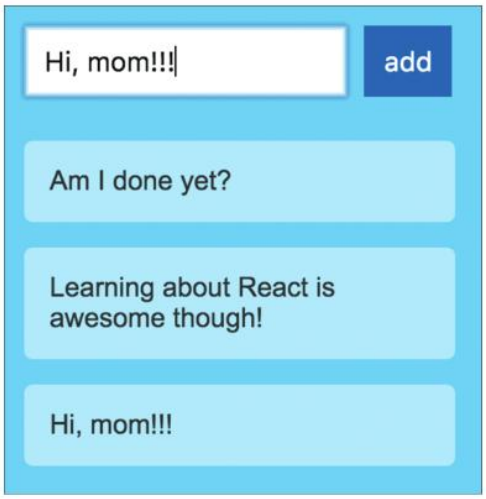
Yang kami lakukan adalah mengembalikan elemen ul yang isinya adalah elemen daftar yang disimpan oleh listItems. Setelah Anda menambahkan ini, simpan dokumen Anda dan pratinjau aplikasi Anda. Anda akan melihat sesuatu yang tampak seperti Gambar 14.7 setelah memasukkan beberapa tugas. Aplikasi kita berfungsi! Setiap tugas yang Anda kirimkan muncul di item daftarnya sendiri. Tarik napas dalam-dalam dan rileks sejenak. Ini adalah kemajuan yang mengagumkan, dan yang tersisa hanyalah beberapa hal kecil di sana-sini yang perlu diselesaikan.

Menambahkan Sentuhan Akhir

Kita hampir selesai di sini! Pertama, apa yang kita miliki saat ini tidak terlihat persis seperti contoh yang kita mulai. Daftar tugas kita terlihat agak polos, tetapi itu dapat diperbaiki dengan sedikit keajaiban CSS. Di dalam blok gaya Anda, tambahkan aturan gaya berikut tepat di bawah tempat aturan gaya Anda yang ada berada:

```
.todoListMain .theList {
  list-style: none;
  padding-left: 0;
  width: 255px;
}
.todoListMain .theList li {
  color: #333;
  background-color: rgba(255,255,255,.5);
  padding: 15px;
  margin-bottom: 15px;
  border-radius: 5px;
}
```

Jika Anda melihat pratinjau aplikasi Anda sekarang, Anda akan melihat bahwa tugas yang dimasukkan terlihat persis seperti yang Anda harapkan:



Selanjutnya, apakah Anda memperhatikan bahwa apa pun yang Anda masukkan ke dalam kolom input tidak hilang setelah Anda mengirimkan formulir? Anda harus menghapus kolom secara manual setiap kali setelah mengirimkan tugas...seperti binatang! Itu menyebalkan, tetapi perbaikannya cukup mudah. Di dalam metode addItem komponen TodoList, tambahkan baris yang disorot berikut:

```
addItem: function(e) {
   var itemArray = this.state.items;

   itemArray.push(
   {
      text: this._inputElement.value,
      key: Date.now()
   }
   );

   this.setState({
      items: itemArray
   });

   this._inputElement.value = "";

   e.preventDefault();
}
```

Yang kami lakukan di sini hanyalah membersihkan properti nilai elemen input saat formulir dikirimkan dan metode addItem dipanggil. Ini memastikan bahwa kami tidak perlu lagi membersihkan kolom input secara manual di antara setiap tugas yang ingin kami kirimkan. Sederhana saja!

14.8 KESIMPULAN

Aplikasi Todo kami cukup sederhana dalam fungsinya, tetapi dengan membangunnya dari awal, kami membahas hampir setiap detail kecil yang menarik yang dihadirkan React. Yang lebih penting, kami membuat contoh yang menunjukkan bagaimana berbagai konsep yang kami pelajari secara individual saling terkait. Itulah detail yang penting. Nah, ini pertanyaan singkat untuk Anda: apakah semua yang telah kita lakukan dalam bab ini masuk akal?

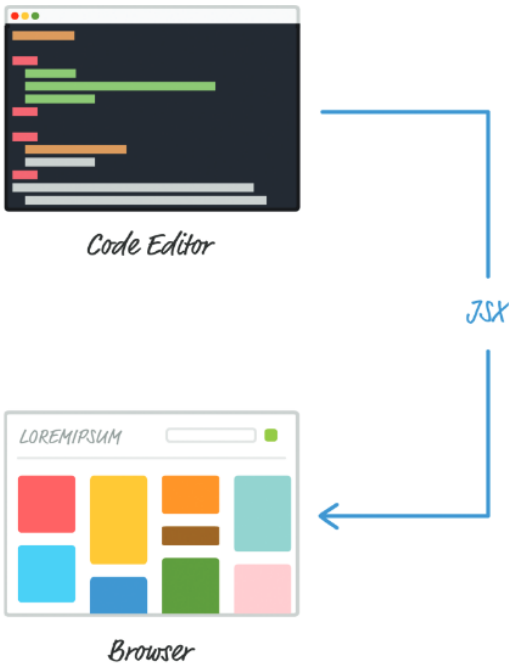
Jika semua yang telah kita lakukan dalam bab ini masuk akal, maka Anda siap memberi tahu teman dan keluarga bahwa Anda hampir menguasai React! Jika ada bagian yang menurut Anda membingungkan, saya sarankan Anda kembali dan membaca ulang bab-bab yang membahas kebingungan Anda.

BAB 15
MENYIAPKAN LINGKUNGAN PENGEMBANGAN REACT

15.1 PENYIAPAN LINGKUNGAN REACT

Topik utama terakhir yang terkait dengan React yang akan kita bahas lebih sedikit membahas React dan lebih banyak membahas penyiapan lingkungan pengembangan Anda untuk membangun aplikasi React. Hingga saat ini, kita telah membangun aplikasi React dengan menyertakan beberapa file skrip:

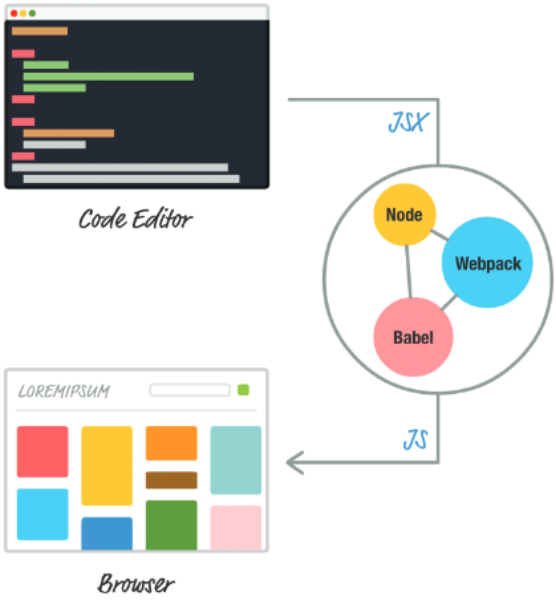
```
<script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
<script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.23/browser.min.js"></script>
```



Gambar 15.1 Pendekatan React Kami

Berkas skrip ini tidak hanya memuat pustaka React, tetapi juga memuat Babel untuk membantu peramban kita melakukan apa yang perlu dilakukan saat menemui hal aneh seperti JSX (lihat Gambar 15.1). Untuk meninjau apa yang kami sebutkan sebelumnya saat berbicara tentang pendekatan ini, sisi negatifnya adalah kinerja. Sebagai bagian dari browser Anda yang melakukan semua hal terkait pemuatan halaman yang biasanya dilakukannya, ia juga bertanggung jawab untuk mengubah JSX Anda menjadi JavaScript yang sebenarnya. Konversi JSX ke JavaScript adalah proses yang memakan waktu yang tidak masalah selama pengembangan. Tidak masalah jika setiap pengguna aplikasi Anda harus membayar penalti

kinerja tersebut. Solusinya adalah menyiapkan lingkungan pengembangan Anda tempat konversi JSX ke JS ditangani sebelum pengguna memuat halaman (lihat Gambar 15-2).



Gambar 15.2 Konversi JSX Ke Javascript Sebagai Bagian Dari Proses Pembuatan Aplikasi Anda

Dengan solusi ini, browser Anda memuat aplikasi Anda dan menangani file JavaScript yang telah dikonversi (dan berpotensi dioptimalkan). Bagus, bukan? Sekarang, satu-satunya alasan mengapa kami menunda membicarakan semua ini hingga sekarang adalah demi kesederhanaan. Mempelajari React sudah cukup sulit. Menambahkan kompleksitas alat pembuatan dan menyiapkan lingkungan Anda sebagai bagian dari pembelajaran React tidaklah keren.

Sekarang setelah Anda memahami sepenuhnya semua yang dilakukan React, saatnya mengubahnya dengan bab ini. Di bagian berikut, kita akan membahas satu cara untuk menyiapkan lingkungan pengembangan Anda menggunakan kombinasi Node, Babel, dan webpack. Jika semua ini terdengar aneh bagi Anda, jangan khawatir. Anda akan terbiasa dengan semua alat ini di akhir pembahasan.

Catatan: Hal-hal Dapat Berubah

Alat-alat build dan dependensinya berubah sepanjang waktu. Itu berita bagus bagi kami, tetapi membuat penerbitan informasi tentangnya menjadi tantangan! Bab ini berisi informasi terbaru berdasarkan praktik terbaik terkini (alias saat ini ditulis!), tetapi informasi ini dapat berubah. Jika Anda menemukan bahwa beberapa alat dan petunjuk tidak berfungsi sebagaimana mestinya, silakan periksa versi daring artikel ini (yang lebih sering diperbarui) di lokasi berikut: https://www.kirupa.com/react/setting_up_react_environment.htm

Kenali Alat-alatnya

Baiklah, sekarang saatnya untuk menjauh dari hal-hal umum (dan diagram yang menarik). Sekarang saatnya untuk serius—eh. Sekarang saatnya untuk mengenal alat-alat yang akan kita andalkan untuk menyiapkan lingkungan pengembangan kita dengan benar.

Node.js

Selama ini, JavaScript adalah sesuatu yang Anda tulis terutama untuk menjalankan sesuatu di browser Anda. Node.js mengubah semua ini. Node.js memungkinkan Anda menggunakan JavaScript untuk membuat aplikasi yang berjalan di server dan memiliki akses ke API dan sumber daya sistem yang bahkan tidak dapat diimpikan oleh browser Anda.

Pada dasarnya, ini adalah runtime pengembangan aplikasi lengkap yang aplikasinya (bukannya ditulis dalam Java, C#, C++, dll.) dibuat dan dijalankan sepenuhnya pada JavaScript. Untuk keperluan kita, kita akan mengandalkan Node.js (atau Node Package Manager, alias NPM) untuk mengelola dependensi dan menggabungkan langkah-langkah yang diperlukan untuk beralih dari JSX ke JavaScript. Anggap Node.js sebagai perekat yang membuat lingkungan pengembangan kita berfungsi.

Babel

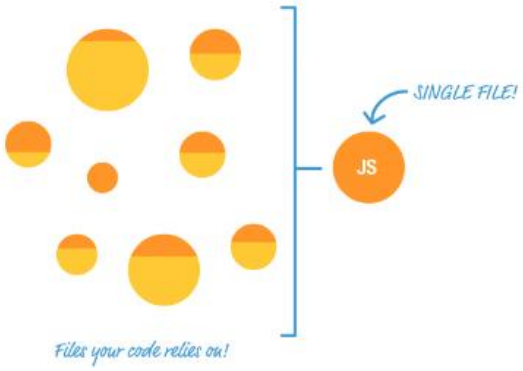
Yang ini seharusnya sudah tidak asing lagi bagi kita! Sederhananya, Babel adalah transpiler JavaScript. Ia mengubah JavaScript Anda menjadi... eh... JavaScript. Kedengarannya sangat aneh, jadi izinkan saya menjelaskannya. Jika Anda menggunakan fitur JavaScript terbaru, browser lama mungkin tidak tahu apa yang harus dilakukan saat menemukan fungsi atau properti baru. Jika Anda menulis JSX, yah... tidak ada browser yang akan tahu apa yang harus dilakukan dengan itu!

Yang dilakukan Babel adalah mengambil JS atau JSX baru Anda dan mengubahnya menjadi bentuk JS yang dapat dipahami oleh sebagian besar browser. Kami telah menggunakan versi dalam browser untuk mengubah JSX kami menjadi JavaScript selama ini. Dalam beberapa saat, Anda akan melihat bagaimana kami dapat mengintegrasikan Babel sebagai bagian dari proses pembuatan kami untuk menghasilkan file JS yang benar-benar dapat dibaca oleh browser dari JSX kami.

Webpack

Alat terakhir yang akan kita andalkan adalah webpack. Alat ini dikenal sebagai module bundler. Mengesampingkan judul yang menarik, banyak framework dan library yang disertakan aplikasi Anda memiliki banyak dependensi di mana berbagai bagian fungsionalitas yang Anda andalkan mungkin hanya merupakan bagian dari komponen yang lebih besar.

Anda mungkin tidak menginginkan semua kode yang tidak perlu itu, dan alat seperti webpack memainkan peran penting untuk memungkinkan Anda hanya menyertakan kode relevan yang diperlukan agar aplikasi Anda berfungsi. Alat ini sering kali menggabungkan semua kode yang relevan (bahkan jika berasal dari berbagai sumber) ke dalam satu file (lihat Gambar 15.3).

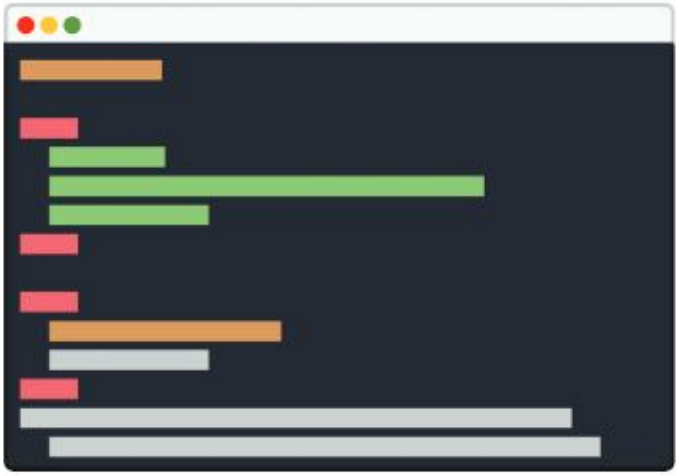


Gambar 15.3 File Yang Dikemas Ke Dalam Satu File

Kita akan mengandalkan webpack untuk menggabungkan bagian-bagian yang relevan dari library React, file JSX kita, dan JavaScript tambahan apa pun ke dalam satu file. Ini juga berlaku untuk file CSS (LESS/SASS) dan jenis aset lain yang digunakan aplikasi Anda, tetapi kita akan fokus pada sisi JavaScript saja di sini.

Editor Kode Anda

Tidak ada pembicaraan tentang lingkungan pengembangan Anda yang dapat terjadi tanpa membicarakan alat yang paling penting dalam semua ini, yaitu editor kode Anda (lihat Gambar 15.4).



Gambar 15.4 Editor Kode Anda

Tidak masalah apakah Anda menggunakan Sublime, Atom, VisualStudio Code, TextMate, Coda, atau alat lainnya. Anda akan menghabiskan banyak waktu di editor kode Anda tidak hanya untuk membangun aplikasi React Anda tetapi juga untuk mengonfigurasi berbagai berkas konfigurasi yang dibutuhkan Node, Babel, dan WebPack.

Saatnya Menyiapkan Lingkungan

Pada titik ini, Anda seharusnya sudah memiliki gambaran samar tentang apa yang ingin kita lakukan...mimpi yang ingin kita capai! Kita bahkan melihat berbagai alat yang akan berperan dalam mewujudkan mimpi ini. Sekarang, saatnya kerja keras untuk benar-benar mewujudkan semuanya.

15.2 MENYIAPKAN STRUKTUR PROYEK AWAL

Hal pertama yang akan kita lakukan adalah menyiapkan proyek kita. Buka Desktop Anda dan buat folder baru bernama MyTotallyAwesomeApp. Di dalam folder ini, buat dua folder lagi bernama dev dan output. Susunan folder Anda akan terlihat seperti Gambar 15.5. Apa yang kita lakukan di sini cukup sederhana. Di dalam folder dev, kita akan meletakkan semua JSX, JavaScript, dan konten terkait skrip lainnya yang belum dioptimalkan dan belum dikonversi.

Dengan kata lain, di sinilah kode yang sedang Anda tulis dan kerjakan akan berada. Di dalam folder output, kita akan meletakkan hasil dari menjalankan berbagai alat build pada file skrip yang ditemukan di dalam folder dev. Di sinilah Babel akan mengonversi semua file JSX kita menjadi JS. Di sinilah webpack akan menyelesaikan semua dependensi antara file skrip kita, dan meletakkan semua konten skrip penting ke dalam satu file JavaScript.

Gambar 15.5 Susunan Folder Kita Saat Ini

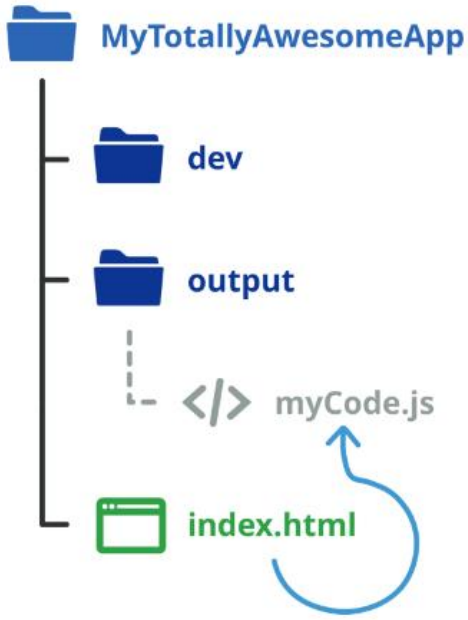
Hal berikutnya yang akan kita lakukan adalah membuat file HTML yang akan kita arahkan ke browser kita. Di dalam folder MyTotallyAwesomeApp, gunakan editor kode Anda untuk membuat file HTML baru bernama index.html dengan konten berikut:

```
<!DOCTYPE html>
<html>

<head>
<title>React! React! React!</title>
</head>

<body>
  <div id="container"></div>

  <script src="output/myCode.js"></script>
</body>

</html>
```

Pastikan untuk menyimpan berkas Anda setelah menambahkan konten ini. Sekarang, berbicara tentang konten, markup kita cukup sederhana. Isi dokumen kita hanyalah elemen div kosong dengan nilai id container dan tag skrip yang mengarah ke berkas JavaScript final (myCode.js) yang akan dihasilkan di dalam folder output:

```
<script src="output/myCode.js"></script>
```

Selain kedua hal tersebut, berkas HTML kita tidak memiliki banyak hal yang menarik. Jika kita harus memvisualisasikan hubungan dari semuanya sekarang, maka akan terlihat seperti Gambar 15.6.

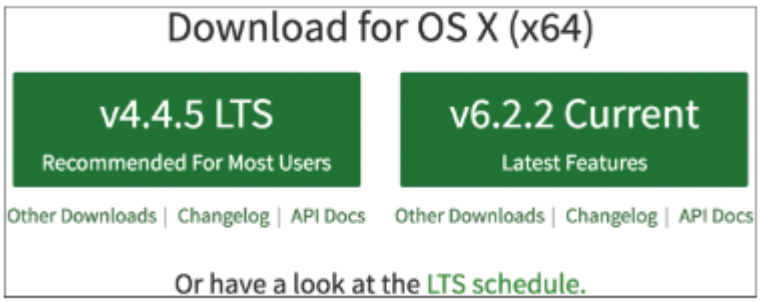


Gambar 15.6 Seperti Apa Struktur Proyek Anda Saat Ini

Saya telah memberi tanda titik pada berkas myCode.js di folder output kita karena berkas tersebut belum ada di sana. Kita menunjuk ke sesuatu di HTML kita yang saat ini tidak ada, tetapi tidak akan tetap seperti itu untuk waktu yang lama.

15.3 MENGINSTAL DAN MENGINISIALISASI NODE.JS

Langkah selanjutnya adalah menginstal Node.js. Kunjungi situs Node.js (https://nodejs.org/) untuk menginstal versi yang sesuai untuk sistem operasi Anda (lihat Gambar 15.7).



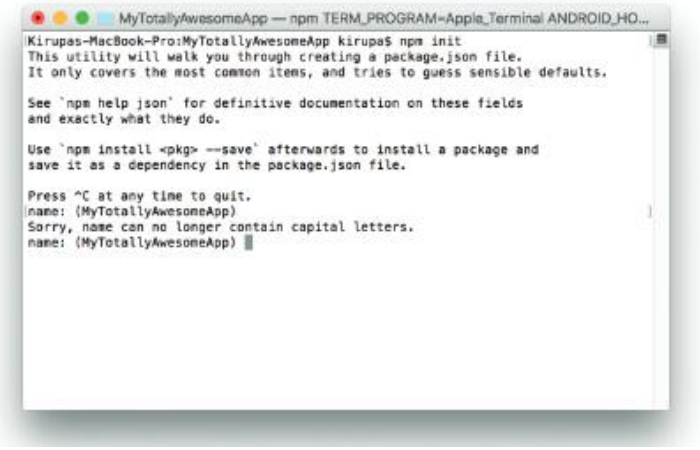
Gambar 15.7 Tombol Unduh Di Situs Node.Js

Saya cenderung selalu menginstal versi terbaru, jadi Anda juga harus menggunakannya. Prosedur pengunduhan dan instalasi tidak terlalu menarik. Setelah Node.js terinstal, uji untuk memastikannya benar-benar terinstal dengan meluncurkan Terminal (di Mac), Command Prompt (di Windows), atau alat setara pilihan Anda dan ketik perintah berikut dan tekan Enter: node -v Jika semuanya berjalan dengan baik, Anda akan melihat nomor versi yang ditampilkan yang biasanya sesuai dengan versi Node.js yang baru saja Anda instal.

Jika Anda mendapatkan kesalahan karena alasan apa pun, ikuti langkah-langkah pemecahan masalah yang tercantum di sini (https://github.com/npm/npm/wiki/ Pemecahan Masalah). Selanjutnya, kita akan menginisialisasi Node.js pada folder MyTotallyAwesomeApp. Untuk melakukannya, pertama-tama navigasikan ke folder MyTotallyAwesomeApp menggunakan Terminal atau Command Prompt. Pada OS X, tampilannya akan seperti Gambar 15.8.

Gambar 15.8 Navigasi Ke Folder Mytotallyawesomeapp

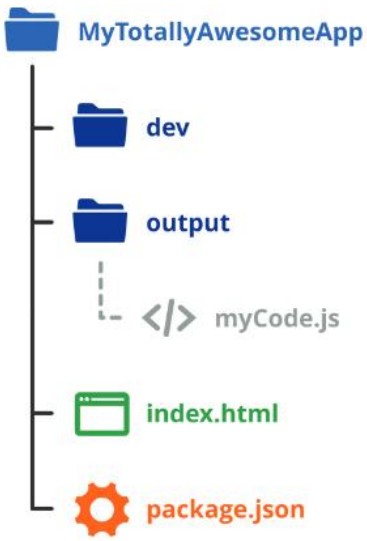
Sekarang, lanjutkan dan inisialisasi Node.js dengan memasukkan perintah berikut:
npm init



Gambar 15.9 Nama Folder Proyek Kita Mengandung Huruf Kapital, Yang Memicu Kesalahan

Ini akan memulai serangkaian pertanyaan yang akan membantu menyiapkan Node.js pada proyek kita. Pertanyaan pertama akan meminta Anda untuk menentukan nama proyek Anda. Menekan Enter akan memungkinkan Anda untuk menentukan nilai default yang telah dipilih untuk Anda. Itu semua bagus, tetapi nama default adalah folder proyek kita, yaitu MyTotallyAwesomeApp.

Jika Anda menekan Enter, karena mengandung huruf kapital, maka akan muncul kesalahan (lihat Gambar 15.9). Lanjutkan dan masukkan versi huruf kecil dari nama tersebut, mytotallyawesomeapp. Setelah Anda melakukannya, tekan Enter. Untuk pertanyaan yang tersisa, cukup tekan Enter untuk menerima semua nilai default. Hasil akhir dari semua ini adalah file baru bernama package.json yang akan dibuat di folder MyTotallyAwesomeApp Anda (lihat Gambar 15.10).



Gambar 15.10 File Package.Json Muncul Di Folder Anda

Jika Anda membuka konten package.json di editor kode, Anda akan melihat sesuatu yang mirip dengan berikut ini:

```
{
  "name": "mytotallyawesomeapp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

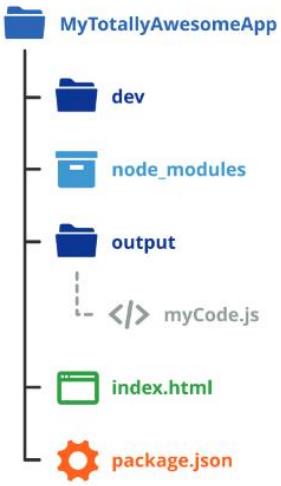
Jangan terlalu khawatir tentang isi berkas ini, tetapi ketahuilah bahwa salah satu hasil dari pemanggilan npm init adalah Anda memiliki berkas package.json yang dibuat dengan beberapa properti dan nilai aneh yang Node.js benar-benar tahu apa yang harus dilakukan dengannya.

15.4 MEMASANG DEPENDENSI REACT

Apa yang akan kita lakukan selanjutnya adalah memasang dependensi React kita sehingga kita dapat menggunakan pustaka React dan React DOM dalam kode kita. Jika Anda berasal dari latar belakang pengembangan web murni, ini akan terdengar aneh. Bersabarlah dengan saya. Di Terminal atau Command Prompt Anda, masukkan yang berikut untuk memasang dependensi React kita:

```
npm install react react-dom –save
```

Setelah Anda memasukkan ini, banyak hal aneh akan muncul di layar Anda. Anda bahkan mungkin melihat banyak peringatan, tetapi peringatan tersebut seharusnya aman untuk diabaikan. Yang terjadi adalah pustaka React dan React-DOM (dan hal-hal yang bergantung padanya) diunduh dari repositori raksasa paket Node.js yang ditemukan di sini: https://www.npmjs.com/



Gambar 15.11 Struktur Folder Yang Diperbarui

Jika Anda melihat folder MyTotallyAwesomeApp, Anda akan melihat folder bernama node_modules. Di dalam folder tersebut, Anda akan melihat sekumpulan berbagai modul (alias apa yang Node.js sebut sebagai pustaka, yang biasa kita sebut manusia biasa). Mari kita perbarui visualisasi struktur file/folder kita saat ini agar terlihat seperti Gambar 15.11. Daftar modul yang Anda lihat saat ini hanyalah permulaan. Kami akan menambahkan beberapa lagi saat Anda mencapai akhir, jadi jangan terlalu terpaku pada jumlah item yang Anda lihat di dalam folder node_modules :P

15.5 MENAMBAHKAN FILE JSX

Hal-hal akan menjadi lebih menarik. Sekarang setelah kami memberi tahu Node.js tentang minat kami pada React, kami selangkah lebih dekat untuk membangun aplikasi React. Kami akan lebih jauh memasuki area ini dengan menambahkan file JSX yang merupakan versi modifikasi dari contoh yang kami lihat di Bab 3 saat melihat Komponen. Di dalam folder dev, menggunakan editor kode, buat file bernama index.jsx dengan kode berikut sebagai isinya:

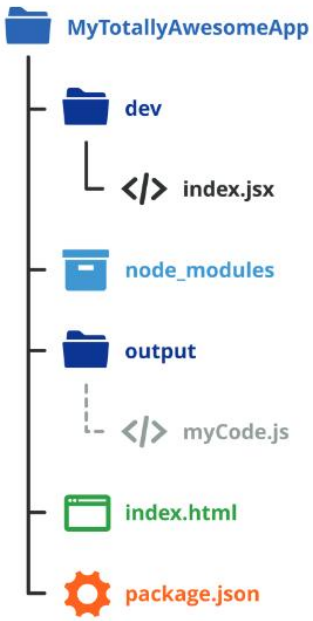
```
import React from "react";
import ReactDOM from "react-dom";

var HelloWorld = React.createClass({
  render: function() {
    return (
      <p>Hello, {this.props.greetTarget}!</p>
    );
  }
});

ReactDOM.render(
  <div>
    <HelloWorld greetTarget="Batman"/>
    <HelloWorld greetTarget="Iron Man"/>
    <HelloWorld greetTarget="Nicolas Cage"/>
    <HelloWorld greetTarget="Mega Man"/>
    <HelloWorld greetTarget="Bono"/>
    <HelloWorld greetTarget="Catwoman"/>
  </div>,
  document.querySelector("#container")
);
```

Perhatikan bahwa sebagian besar JSX yang kita tambahkan hampir tidak dimodifikasi dari apa yang kita miliki sebelumnya. Satu-satunya perbedaan adalah bahwa apa yang dulunya merupakan referensi skrip untuk memasukkan pustaka React dan React DOM ke dalam aplikasi kita kini telah diganti dengan pernyataan impor yang menunjuk ke paket Node.js react dan react-dom yang kita tambahkan beberapa saat yang lalu:

```
import React from "react";
```

```
import ReactDOM from "react-dom";
```

Sekarang, Anda mungkin penasaran kapan kita dapat membangun aplikasi dan menjalankannya di browser. Masih ada beberapa langkah lagi. Gambar 15.12 menunjukkan seperti apa visualisasi proyek kita saat ini.



Gambar 15.12 Proyek Saat Ini

File index.html kita mencari kode dari file myCode.js yang masih belum ada. Kita menambahkan file JSX, tetapi kita tahu bahwa browser tidak tahu apa yang harus dilakukan dengan JSX. Kita perlu berpindah dari index.jsx di folder dev ke myCode.js di folder output. Coba tebak apa yang akan kita lakukan selanjutnya?

Beralih Dari JSX Ke Javascript

Langkah yang hilang saat ini adalah mengubah JSX kita menjadi JavaScript yang dapat dipahami oleh peramban kita. Ini melibatkan webpack dan Babel, dan kita akan mengonfigurasi keduanya agar semuanya berfungsi.

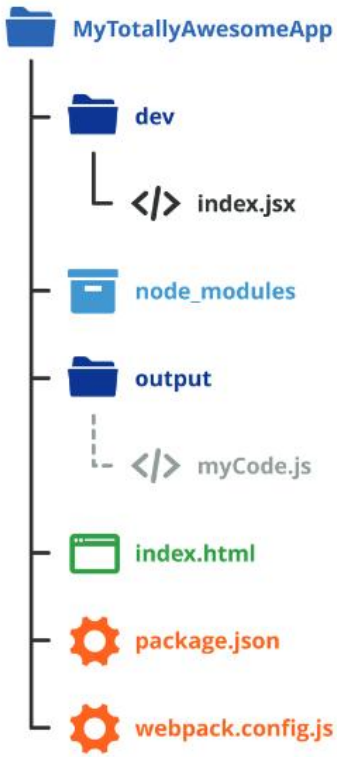
15.6 MENYIAPKAN WEBPACK

Karena kita berada di wilayah Node.js dan webpack dan Babel ada sebagai paket Node, kita perlu menginstal keduanya seperti kita menginstal paket terkait React. Untuk menginstal webpack, masukkan perintah berikut di Terminal/Command Prompt Anda:

```
npm install webpack –save
```

Ini akan memakan waktu beberapa saat sementara paket webpack (dan daftar dependensinya yang besar) diunduh dan ditempatkan ke dalam folder node_modules kita. Setelah Anda melakukan ini, kita perlu menambahkan file konfigurasi untuk menentukan bagaimana webpack akan bekerja dengan proyek kita saat ini. Dengan menggunakan editor kode Anda,

tambahkan file bernama webpack.config.js di dalam folder MyTotallyAwesomeApp kita (lihat Gambar 15.13).



Gambar 15.13 Menambahkan Webpack.Config.Js

Di dalam berkas ini, kita akan menentukan sekumpulan properti JavaScript untuk menentukan di mana berkas sumber asli kita yang tidak dimodifikasi berada dan di mana akan mengeluarkan berkas sumber akhir. Lanjutkan dan tambahkan JavaScript berikut ke webpack.config.js:

```
var webpack = require("webpack");
var path = require("path");

var DEV = path.resolve(__dirname, "dev");
var OUTPUT = path.resolve(__dirname, "output");

var config = {
   entry: DEV + "/index.jsx",
   output: {
      path: OUTPUT,
      filename: "myCode.js"
   }
};

module.exports = config;
```

Luangkan waktu sejenak untuk melihat apa yang dilakukan kode ini. Kami telah menetapkan

dua variabel yang disebut DEV dan OUTPUT yang merujuk ke folder dengan nama yang sama dalam proyek kami. Di dalam objek konfigurasi, kami memiliki dua properti yang disebut entry dan output yang menggunakan variabel DEV dan OUTPUT kami untuk membantu memetakan file index.jsx kami menjadi myCode.js.

15.7 MENYIAPKAN BABEL

Bagian terakhir dalam penyiapan kami saat ini adalah mengubah file index.jsx kami menjadi JavaScript biasa dalam bentuk myCode.js. Di sinilah Babel berperan. Untuk menginstal Babel, mari kembali ke Terminal/Command Prompt andalan kami dan masukkan perintah Node.js berikut:

```
npm install babel-loader babel-preset-es2015 babel-preset-react --save
```

Dengan perintah ini, kami menginstal paket babel-loader, babel-preset-es2015, dan babel-preset-react. Sekarang kami perlu mengonfigurasi Babel agar berfungsi dengan proyek kami. Ini adalah proses dua langkah. Langkah pertama adalah menentukan preset Babel mana yang ingin kita gunakan. Ada beberapa cara untuk melakukannya, tetapi cara yang saya sukai adalah memodifikasi package.json dan menambahkan konten yang disorot berikut:

```
{
     "name": "mytotallyawesomeapp",
     "version": "1.0.0",
     "description": "",
     "main": "index.js",
     "scripts": {
     "test": "echo \"Error: no test specified\" && exit 1"
},
   "author": "",
   "license": "ISC",
   "dependencies": {
     "babel-loader": "^6.2.4",
     "babel-preset-es2015": "^6.9.0",
     "babel-preset-react": "^6.5.0",
     "react": "^15.1.0",
     "react-dom": "^15.1.0",
     "webpack": "^1.13.1"
},
   "babel": {
     "presets": [
       "es2015",
       "react"
   ]
}
}
```

Pada baris yang disorot, kita tentukan objek babel kita dan tentukan nilai preset es2015 dan react. Langkah kedua adalah memberi tahu webpack tentang Babel. Pada berkas

webpack.config.js kita, lanjutkan dan tambahkan baris yang disorot berikut:

```
var webpack = require("webpack");
var path = require("path");

var DEV = path.resolve(__dirname, "Dev");
var OUTPUT = path.resolve(__dirname, "output");

var config = {
   entry: DEV + "/index.jsx",
   output: {
      path: OUTPUT,
      filename: "myCode.js"
   },
   module: {
      loaders: [{
         include: DEV,
         loader: "babel",
      }]
   }
};

module.exports = config;
```

Kami menambahkan objek modul dan loader yang memberi tahu webpack untuk meneruskan file index.jsx yang didefinisikan dalam properti entri kami untuk diubah menjadi JavaScript melalui Babel. Dengan perubahan ini, kami telah menyiapkan lingkungan pengembangan untuk membangun aplikasi React.

Membangun dan Menguji Aplikasi Kami

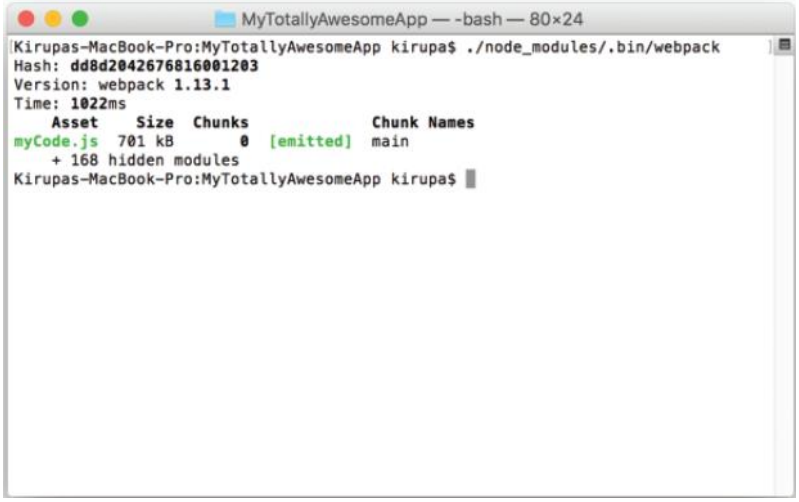
Langkah terakhir (dan semoga paling memuaskan) dalam semua ini adalah membangun aplikasi kami dan menjalankan alur kerja menyeluruh. Untuk membangun aplikasi kami, apa yang Anda ketik bervariasi, tergantung apakah Anda berada di Terminal atau di Command Prompt. Untuk Terminal di Mac, masukkan yang berikut:

```
./node_modules/.bin/webpack
```

Di Command Prompt di Windows, masukkan ini sebagai gantinya:

```
node_modules\.bin\webpack.cmd
```

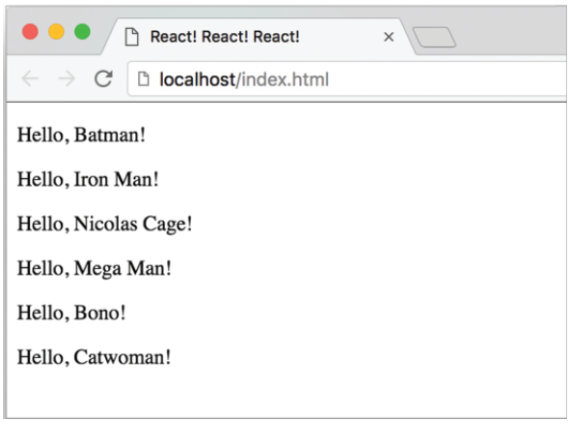
Perintah ini menjalankan webpack dan melakukan semua hal yang telah kami tentukan dalam file konfigurasi webpack.config.js dan package.json kami. Output Anda di Terminal/Command Prompt akan terlihat seperti Gambar 15-14.



Gambar 15.14 Output Webpack

Selain melihat sesuatu yang samar-samar tampak seperti hasil pembangunan yang berhasil ditampilkan dalam bentuk teks samar, buka folder MyTotallyAwesomeApp Anda. Buka file index.html di peramban Anda. Jika semuanya telah disiapkan dengan benar, Anda akan melihat aplikasi React sederhana kami ditampilkan (lihat Gambar 15.15). Jika Anda masuk ke folder Output dan melihat myCode.js, Anda akan melihat file yang cukup besar (~700Kb) dengan banyak JavaScript yang terdiri dari React, ReactDOM, dan kode aplikasi Anda yang relevan, semuanya terorganisasi di sana.

Dari titik ini, Anda dapat membangun aplikasi, menambahkan aset baru, dan membuat perubahan umum yang biasanya Anda lakukan. Satu-satunya perbedaan antara apa yang telah kita lakukan di seluruh buku ini dan apa yang kita lakukan sekarang adalah sederhana—apa yang menjadi perhatian browser Anda dibuat untuk Anda oleh berbagai alat dan pengemas build. Browser Anda tidak lagi mengambil semua React JSX/ES6/dll. ini dan mengubahnya menjadi HTML/CSS/JS normal saat halaman dimuat.



Gambar 15.15 Tampilan Aplikasi React Sederhana.

15.8 KESIMPULAN

Di beberapa bagian sebelumnya, kita mengikuti sejumlah langkah aneh dan tidak dapat dipahami untuk menyiapkan lingkungan build kita guna membangun aplikasi React kita. Apa

yang telah kita lihat hanyalah sebagian kecil dari semua yang dapat Anda lakukan saat menggabungkan Node, Babel, dan webpack. Sayangnya, mencakup semua itu jauh melampaui cakupan pembelajaran React, tetapi jika Anda tertarik dengan ini, Anda harus meluangkan waktu untuk mempelajari seluk-beluk semua alat bantu ini. Ada banyak hal menarik yang dapat Anda lakukan. Untuk informasi lebih lanjut tentang hal-hal menarik tersebut, lihat tautan berikut:

- Babel: https://babeljs.io/
- Dokumentasi npm: https://docs.npmjs.com/
- Penggabung modul webpack: https://webpack.github.io/
- Integrasi Perkakas React: https://facebook.github.io/react/docs/tooling-integration.html
- Bower: https://bower.io/

BAB 16
TRIVIA REACT NATIVE

16.1 MEMULAI DENGAN REACT NATIVE DAN APLIKASI RNTRIVIA

Setelah membaca empat bab pertama dan mulai membangun aplikasi nyata, saya harap Anda mulai merasakan kekuatan React Native. Anda sudah melihat bagaimana banyak hal bisa dilakukan dalam batasan satu perangkat seluler. Namun, dalam kenyataannya, aplikasi modern jarang beroperasi hanya pada satu perangkat saja. Saat ini, sebagian besar aplikasi memiliki tingkat konektivitas ke server atau perangkat lain. Aplikasi-aplikasi tersebut sering berinteraksi untuk tujuan seperti memesan hotel, mendapatkan petunjuk arah, menyimpan data ke server pusat, atau menghubungkan dua orang dalam permainan yang bersaing.

React Native tentu saja dapat menangani aplikasi yang terhubung dengan server ini, namun untuk melakukannya, ada berbagai pilihan yang harus dipertimbangkan. Misalnya, apakah Anda akan membangun API RESTful di server? Atau mungkin Anda akan menggunakan koneksi soket langsung? Mungkin Anda akan menggunakan protokol lain seperti FTP atau NNTP? Dalam bab ini, kita akan mengeksplorasi beberapa opsi ini dengan membangun aplikasi kedua dari tiga aplikasi yang akan kita buat bersama dalam buku ini.

Dalam proses membangun aplikasi ini, Anda tidak hanya akan mempelajari lebih lanjut tentang React Native, tetapi juga tentang Node.js dan beberapa metode komunikasi klien-server yang lebih baru, seperti WebSockets. Proyek ini melibatkan pembangunan komponen server dan menghubungkan aplikasi React Native ke server tersebut. Mari kita mulai!

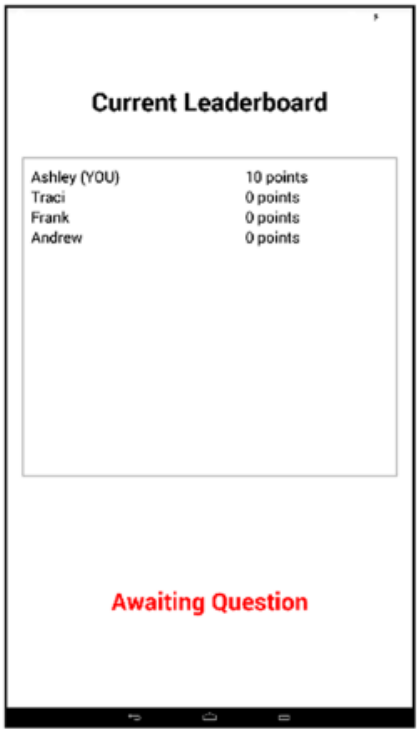
Apa yang Akan Kita Bangun?

Secara singkat (atau mungkin lebih tepatnya dalam tiga kata): kita akan membangun *RNTrivia*. Aplikasi ini memungkinkan kita untuk mengadakan kontes trivia untuk sekelompok orang. Satu orang, yang berperan sebagai administrator (atau "admin"), akan mengontrol kapan pertanyaan baru dikirim ke para pemain. Pemain akan menjawab pertanyaan-pertanyaan tersebut, dan papan peringkat akan diperbarui secara otomatis. Ketika admin memilih untuk mengakhiri permainan, permainan pun berakhir.

Aplikasi ini adalah proyek yang saya buat ketika memberikan presentasi di pustaka Webix pada tahun 2018, dan saya menggunakannya sebagai cara yang menyenangkan dan interaktif untuk memberikan hadiah. Saya memiliki banyak pertanyaan trivia bertema fiksi ilmiah (yang juga akan Anda temukan dalam bundel unduhan kode), dan para pemenang mendapat hadiah. Meskipun saya tidak menganggapnya sebagai *permainan* penuh (karena bab 7 dan 8 akan mencakup proyek permainan lainnya), saya lebih melihat aplikasi ini sebagai *alat* yang menyenangkan untuk digunakan.

RNTrivia memiliki beberapa layar, namun dua layar utamanya adalah layar papan peringkat dan layar pertanyaan. Layar papan peringkat, seperti yang dapat Anda lihat pada Gambar 5.1, adalah tempat pemain menunggu sampai admin memicu pertanyaan baru untuk dikirim ke semua pemain. Di layar ini, permainan sedang berlangsung dan pemain siap

menjawab ketika pertanyaan baru muncul.



Gambar 16.1 Layar papan peringkat ("permainan sedang berlangsung, menunggu pertanyaan")

Setelah pertanyaan dikirim, para pemain akan menemukan diri mereka di layar pertanyaan, seperti yang ditunjukkan pada Gambar 16.2. Di sini, pemain memilih jawaban mereka dan mengirimkannya. Semua ini memerlukan aplikasi klien, yang ditulis dengan React Native yang bagus, dan komponen server, yang akan kita tulis menggunakan Node. Server bertindak sebagai perantara antara admin dan semua pemain, tetapi bagaimana komunikasi ini terjadi cukup menarik, menurut saya, dan saya harap Anda juga akan berpikir demikian.

Itulah yang akan kita bahas sepanjang waktu di bab ini, menyimpan kode klien untuk bab berikutnya. Kode server ini akan menangani hampir semua logika inti permainan, termasuk beberapa logika yang mempertimbangkan berapa lama waktu yang dibutuhkan pemain untuk menjawab pertanyaan dalam menetapkan poin, sehingga mendapatkan jawaban yang benar tidak hanya penting tetapi juga penting untuk mendapatkannya secepat mungkin (ini juga memastikan bahwa hasil seri hampir tidak mungkin terjadi).

Gambar 16.2 Layar Pertanyaan

Sekarang mari kita bahas spesifikasi teknis dan selesaikan beberapa prasyarat, sehingga kita dapat masuk ke kode.

Klien

Cara membagi proyek ini menjadi dua bab, yang merupakan pola yang saya pilih untuk tiga proyek buku ini, sangat mudah. Bab ini membahas kode server, dan Bab 6 membahas kode klien. Seperti yang akan Anda lihat di sini, kode server tidak terlalu rumit atau banyak. Jelas ada lebih banyak hal di sisi klien, termasuk beberapa konsep baru, daripada yang ada di sisi server. Namun, saat kita menjelajahi kode server di sini, saya akan berusaha sebaik mungkin untuk memberi Anda beberapa konteks, sejauh menyangkut kode klien, yang cukup bagi Anda untuk memahami apa yang terjadi dalam kode server.

Server

Di Bab 1, saya telah menjelaskan secara singkat tentang Node, bagaimana cara menginstalnya, dan menulis kode sederhana untuk dijalankan dengan Node. Sejak saat itu, Anda sudah mulai menggunakan Node dan *Node Package Manager* (NPM), meskipun mungkin Anda tidak menyadarinya, karena alat seperti React Native dan Expo sudah menggunakannya di balik layar.

Namun sekarang, untuk memungkinkan aplikasi React Native berkomunikasi dengan server, kita perlu menulis kode Node yang lebih konkret. Dalam langkah ini, kita akan membangun server yang memungkinkan aplikasi React Native berinteraksi dengan backend, mengelola data, dan menjalankan logika yang diperlukan.

Jika Anda melakukan pencarian tentang cara menulis server di Node, hal pertama yang mungkin Anda temukan adalah kode yang terlihat seperti ini:

```
require(“http”).createServer((inRequest, inResponse) => {
  inResponse.end("Hello from my first Node server");
  }).listen(80);
```

Catatan Anda juga hampir pasti akan menemukan sesuatu yang disebut Express. Itu adalah pustaka yang berada di atas Node dan membuat pembuatan aplikasi server yang tidak sepele menjadi jauh lebih mudah. Beberapa baris kode di sini bagus dan bagus, tetapi seperti yang saya yakin dapat Anda tebak, membangun sesuatu yang lebih tangguh dengan cepat membengkak menjadi sedikit kode. Express mengabstraksikan sebagian besar dari itu, menghemat waktu dan tenaga Anda, sambil menggunakan pustaka yang telah teruji. Express tidak sesuai dengan kebutuhan kita di sini, karena apa yang akan saya bahas beberapa paragraf dari sekarang, tetapi saya ingin menyebutkan Express, jadi jika Anda menemukannya, Anda memiliki sedikit gambaran tentang apa itu.

Itu tidak diperlukan untuk menulis kode server Node, yang merupakan poin utama, meskipun, jika Anda harus melakukannya, dan apa yang digunakan untuk membangun kode server RNTrivia tidak sesuai dengan kebutuhan, cobalah Express. Sedikit kode yang sangat kecil itu adalah semua yang dibutuhkan di Node untuk menulis server. Jika Anda menjalankannya, lalu jalankan peramban web favorit Anda dan akses localhost, Anda akan mendapatkan balasan "Halo dari server Node pertama saya." Singkatnya, fungsi yang diteruskan ke createServer() menangani setiap permintaan HTTP yang masuk. Anda dapat melakukan apa pun yang Anda perlukan di sana, termasuk hal-hal seperti:

- Menginterogasi permintaan yang masuk untuk menentukan metode HTTP
- Mengurai jalur permintaan
- Memeriksa nilai header

Anda kemudian dapat melakukan beberapa logika percabangan pada salah satu atau semua ini, mungkin mengakses basis data atau mekanisme penyimpanan tahan lama lainnya, dan mengembalikan respons yang sesuai dan sepenuhnya dinamis untuk permintaan tertentu. Hanya dengan sedikit kode ini, Anda sebenarnya mengetahui dasar-dasar tentang apa yang Anda perlukan untuk menulis server untuk RNTrivia. Namun, jika Anda memikirkan tentang apa aplikasi RNTrivia ini, Anda akan segera menyadari kekurangannya: kita harus memiliki kemampuan agar server memulai komunikasi dengan klien, pemain kita. Server harus mengirimkan pertanyaan kepada para pemain. Itu adalah kebalikan dari cara kerja biasanya, dan, memang, kebalikan dari cara kerja contoh server sederhana ini.

Di sini, klien, melalui permintaan perambannya, yang memulai komunikasi dengan server, dan itu tidak akan memenuhi kebutuhan kita. Tentunya ada jawabannya, bukan? Sebenarnya ada lebih dari satu, tetapi yang akan kita gunakan adalah sesuatu yang relatif baru dalam pengembangan web, melalui pustaka kecil yang praktis: WebSocket dan socket.io. Catatan Jika Anda telah lama melakukan pengembangan web, Anda mungkin menyadari bahwa Anda dapat mencapai tujuan yang dinyatakan dengan cara lain, salah satunya adalah polling, di mana klien terus-menerus memanggil server untuk pembaruan status.

Meskipun itu akan berhasil untuk RNTrivia, tujuannya di sini adalah untuk memiliki

sesuatu yang lebih real-time (polling tidak, dengan asumsi Anda menggunakan interval polling yang wajar) dan juga sesuatu yang tidak akan menyumbat sumber daya terbatas yang dimiliki server mana pun, dalam hal kapasitas penanganan permintaan. Pada dasarnya, kami menginginkan sesuatu yang sedikit lebih berpikiran maju daripada polling atau teknik "hack-y" lainnya yang dapat kami gunakan di sini.

Menjaga Jalur Komunikasi Tetap Terbuka: Socket.Io

Web itu sendiri awalnya dirancang sebagai tempat di mana klien bertanggung jawab untuk meminta informasi dari server, tetapi hal itu menghilangkan sejumlah kemungkinan yang menarik, atau setidaknya membuatnya lebih sulit dan tidak optimal. Misalnya, jika Anda memiliki mesin yang menyediakan harga saham kepada klien untuk ditampilkan di dasbor, klien harus terus meminta harga yang diperbarui dari server. Ini adalah pendekatan polling yang umum. Kelemahannya, terutama, adalah memerlukan permintaan baru yang konstan dari klien ke server, dan harga hanya akan selalu baru selama interval polling, yang biasanya tidak ingin Anda lakukan terlalu sering, karena takut membebani server. Harga tidak real-time, sesuatu yang bisa sangat buruk, jika Anda seorang investor.

Dengan munculnya teknik AJAX, pengembang mulai menyelidiki cara untuk melakukan komunikasi dua arah, di mana server dapat mendorong harga saham baru ke klien. Salah satu metode tersebut adalah long-polling. Kadang disebut Comet, long-polling adalah teknik yang digunakan klien untuk membuka koneksi dengan server, seperti biasa. Namun sekarang, server menahan permintaan agar tetap terbuka, dengan tidak pernah mengirimkan sinyal penyelesaian respons HTTP. Kemudian, saat server memiliki sesuatu untuk dikirimkan ke klien, koneksi sudah terbentuk. Ini disebut sebagai "hanging-GET" atau "pending-POST," tergantung pada metode HTTP yang digunakan untuk membuat koneksi.

Ini bisa jadi sulit diterapkan karena banyak alasan, tetapi mungkin alasan utamanya adalah utas pemrosesan koneksi ditahan di server. Mengingat ini adalah koneksi HTTP, overhead sama sekali tidak tidak penting. Dalam waktu singkat, server Anda dapat kewalahan, tanpa harus menghubungkan banyak klien. Protokol WebSocket dibuat untuk memungkinkan koneksi persisten semacam ini tanpa semua masalah long-polling, atau pendekatan lainnya. WebSocket adalah standar Internet Engineering Task Force (IETF) yang memungkinkan komunikasi dua arah antara klien dan server. Ia melakukannya dengan jabat tangan khusus saat koneksi HTTP biasa terbentuk. Untuk melakukan hal ini, klien mengirimkan permintaan yang terlihat seperti ini:

```
GET ws://websocket.apress.com/ HTTP/1.1
 Origin: http://apress.com
 Connection: Upgrade
 Host: websocket.apress.com
 Upgrade: websocket
```

Perhatikan nilai header Upgrade? Itulah bagian ajaibnya. Saat server melihat ini, dan dengan asumsi mendukung WebSocket, server akan merespons dengan balasan seperti ini:

```
HTTP/1.1 101 WebSocket Protocol Handshake
Date: Mon, 21 Dec 2017 03:12:44 EDT
Connection: Upgrade
Upgrade: WebSocket
```

Server "setuju untuk peningkatan," dalam istilah WebSocket. Setelah jabat tangan ini selesai, permintaan HTTP diputus, tetapi koneksi TCP/IP yang mendasarinya tetap ada. Itulah koneksi persisten yang dapat digunakan klien dan server untuk berkomunikasi secara real time, tanpa harus membuat ulang koneksi setiap saat. WebSocket juga dilengkapi dengan API JavaScript yang dapat Anda gunakan untuk membuat koneksi dan mengirim serta menerima pesan (dan pesan adalah apa yang kami sebut data yang dikirimkan melalui koneksi WebSocket, di kedua arah). Namun, saya tidak akan membahas API itu, karena daripada menggunakannya secara langsung di RNTrivia, kami akan menggunakan pustaka yang ada di atasnya dan membuatnya lebih mudah digunakan, pustaka itu adalah socket.io.

Yang akan Anda temukan adalah bahwa pustaka ini ada untuk digunakan dalam kode server berbasis Node dan kode klien berbasis React Native dan memberi kita API yang lebih mudah dan konsisten di kedua tempat. Singkatnya, penggunaan socket.io, selain mengimpor pustaka, hanya memerlukan satu fungsi: io.on(). Aplikasi yang ditulis dengan socket.io akan memiliki satu atau beberapa panggilan seperti itu, satu untuk setiap pesan yang dibutuhkan aplikasi. Tidak masalah apakah pesan tersebut berasal dari klien dan masuk ke server, atau apakah pesan tersebut dimulai di server dan masuk ke klien. io.on() adalah semua yang diperlukan untuk menangani pesan di kedua sisi koneksi.

Metode ini menggunakan dua argumen. Pertama, metode ini menggunakan nama pesan, yang merupakan string acak yang dapat Anda buat agar bermakna dalam aplikasi Anda. Kedua, metode ini menggunakan fungsi panggilan balik yang menangani pesan tersebut. Panggilan balik ini diberikan objek yang merupakan data yang dikirimkan. Anda kemudian dapat melakukan apa pun yang harus Anda lakukan untuk menangani pesan tersebut. Ini bisa saja tidak ada (fungsi kosong), yang sepenuhnya valid. Dan, tidak ada yang mengatakan bahwa Anda harus memiliki pengendali sama sekali untuk pesan tertentu. Tidak ada yang akan rusak jika Anda mengirim pesan yang penerimanya tidak memiliki pengendali.s Jika pesannya adalah updateStock dan dikirim dari server, mungkin dalam kode klien Anda, Anda mungkin menulis

```
io.on("updateStock", function(inData) {
    console.log(‘Stock $’ + {inData.tickerSymbol} +  price is now  +
inData.newPrice}’);
});
```

Sekarang, setiap kali server mengirimkan harga updateStock, yang kami sebut memancarkan pesan, klien akan mengeluarkan harga baru ke konsol. Jika Anda ingin mengirim pesan clearPreferences dari klien ke server, maka di server, Anda dapat menulis

```
 io.on("clearPreferences", function(inData) {
  database.execute('delete from user_preferences where
```

```
userID=${inData.userID}');
});
```

Lihat? Tampilannya sama saja baik di klien maupun server. Nah, begitulah cara Anda menangani pesan, tetapi bagaimana Anda memancarkannya? Sekali lagi, tampilannya sama saja, terlepas dari asal pesan.

```
  io.emit("updateStock", { tickerSymbol: database.getSymbol(), newPrice:
database.getPrice()});
```

atau

```
io.emit("clearPreferences", {userID: "fzammetti" });
```

Seperti yang Anda lihat, API socket.io sangat sederhana tetapi juga sangat canggih. API ini juga menawarkan kemampuan yang lebih canggih, seperti namespace dan room, yang memungkinkan Anda untuk memisahkan pesan ke dalam pengelompokan logis, dan masih banyak lagi. Namun, untuk apa yang kita lakukan di RNTrivia, ini saja yang perlu Anda ketahui. Hanya ada sedikit hal yang lebih penting dari ini terkait dengan pembuatan koneksi, tetapi akan lebih mudah dijelaskan dalam konteks kode RNTrivia.

Memulai: Membangun Server

Sekarang saatnya untuk membedah beberapa kode server. Meskipun benar bahwa ini bukanlah buku tentang Node atau socket.io atau yang lainnya, kita tentu tidak dapat membangun aplikasi seperti ini tanpa membahas kodenya, untuk mendapatkan pandangan holistik tentang RNTrivia. Namun, Anda harus ingat bahwa meskipun saya akan berusaha sebaik mungkin untuk memberi Anda informasi yang cukup untuk memahami kode ini, bahkan jika Anda tidak memiliki pengalaman Node sebelumnya, Node memiliki lebih banyak hal daripada yang akan Anda lihat di sini. Jadi, jika Anda merasa ini menarik (dan saya harap demikian), maka Anda pasti ingin meluangkan waktu untuk menyelaminya lebih dalam.
Karena itu, mari kita bahas!

Masalah Non-Kode: questions.json

Tak satu pun kode yang akan saya bahas akan berguna jika kita tidak memiliki beberapa pertanyaan untuk diajukan kepada pemain kita. Daripada memasukkan pertanyaan ke dalam kode server, saya memilih untuk mengeksternalisasikannya ke dalam berkasnya sendiri, yang akan kita baca nanti. Berkas questions.json adalah daftar pertanyaan sederhana, beberapa di antaranya dapat Anda lihat di sini:

```
  {"questions": [
{"question": "What is the name of the Shadow's homeworld on Babylon 5?",
  "answer": "Z'ha'dum",
  "decoys": ["Galifrey", "Hoth", "Arrakis", "Tagora", "Nihil", "Daxam",
"Acheron", "Skaro", "Crematoria", "Qo'nos"]
},
{"question": "In Stargate SG-1, what galaxy is the lost city of Atlantis
```

```
discovered to reside in?",
      "answer": "Pegasus",
      "decoys": ["Andromeda", "Seraphia", "Triangulum", "Krell", "Virgo",
"Kaliem", "Ida", "Shi'ar", "Xeno", "Isop"]
    },
  ...
  ]}
```

Masih banyak lagi pertanyaan yang memiliki elipsis, tetapi bagian yang terpotong ini memberi Anda gambaran. Setiap pertanyaan adalah objek yang terdiri dari atribut pertanyaan yang merupakan pertanyaan sebenarnya, jawaban yang benar, dan serangkaian sepuluh umpan (jawaban yang salah) untuk setiap pertanyaan. Nanti, kami akan memilih enam dari sepuluh secara acak untuk disajikan kepada para pemain, hanya untuk memberikan sedikit variasi pada prosesnya.

Mengonfigurasi Server: package.json

Meskipun tidak diharuskan untuk menulis aplikasi Node, baik berbasis server atau tidak (sesuatu yang seharusnya jelas, mengingat contoh kode Node sederhana sebelumnya yang telah saya tunjukkan), sangatlah standar untuk memiliki file package.json di root aplikasi. Bahkan, jika Anda akan menggunakan pustaka pihak ketiga, seperti yang kita lakukan di RNTrivia, file ini menjadi sangat penting. (Bukan tidak mungkin untuk memasukkan dependensi tanpa file package.json, tetapi hal itu hampir tidak pernah terdengar.) Berikut adalah file package.json untuk RNTrivia:

```
{
"name": "com.etherient.mtrntrivia", "version": "1.0.0", "author": "Frank
Zammetti",
"description": "A trivia app written with React Native",
"private": true,"license": "MIT", "main": "server.js",
"dependencies": {"socket.io": "2.0.4", "lodash": "*"},
"scripts": {"start": "node server.js"
}
```

Tentu saja, ini bukan pertama kalinya Anda melihat berkas seperti itu, tetapi ini pertama kalinya Anda melihatnya dalam konteks aplikasi Node. Sebagian besar isinya seharusnya sudah jelas, dan beberapa di antaranya secara teknis bersifat opsional, tetapi atribut yang Anda lihat di sini adalah atribut yang diperlukan, dan atribut yang harus Anda berikan untuk menghindari peringatan apa pun dari NPM. Mungkin hal terpenting di sini adalah main, dependencies, dan scripts. Atribut main memberi tahu NPM dan Node berkas JavaScript utama untuk aplikasi kita. Atribut dependencies, tentu saja, adalah pustaka yang menjadi sandaran aplikasi kita. Seperti yang telah dibahas sebelumnya, socket.io seharusnya tidak mengejutkan.

Pustaka lodash, jika Anda belum pernah mendengarnya, adalah pustaka utilitas JavaScript serbaguna yang menyediakan beberapa fungsi generik dan sangat berguna, seperti helper pengurutan; helper untuk mengiterasi array, objek, dan string; helper untuk memanipulasi dan menguji nilai; dan helper untuk membuat fungsi komposit, di antara banyak

fungsi lainnya. Kita akan menggunakannya untuk fungsi-fungsi kecil namun penting nanti, tetapi jika ini adalah pengalaman pertama Anda dengan lodash, maka saya sangat menyarankan untuk meluangkan waktu untuk melihat apa yang ditawarkannya, karena ini adalah pustaka yang sangat membantu yang juga sangat sederhana, kecil, dan efisien, tiga atribut yang sangat saya sukai di pustaka JavaScript saya. Atribut skrip menyiapkan perintah NPM. Dengan kata lain, konfigurasi yang terlihat memungkinkan kita untuk mengeksekusi

```
npm start
```

Dengan konfigurasi yang disediakan, NPM tahu untuk mengeksekusi

```
node.server.js
```

atas nama kami. Mengapa Anda ingin melakukan ini? Nah, jika yang akan Anda lakukan hanyalah menjalankan satu file JavaScript dengan Node, maka mungkin tidak ada manfaat yang signifikan, tetapi menggunakan NPM seperti ini memungkinkan Anda untuk menjalankan perintah sembarang yang Anda suka. Ingin menjalankan Webpack pada kode Anda sebelum menjalankannya? Tidak masalah. Perlu menjalankan aplikasi di bawah akun pengguna alternatif? Anda dapat melakukannya dengan ini. Ditambah lagi, jika Anda melakukan ini untuk semua aplikasi Node Anda, itu berarti Anda tidak perlu memikirkan cara menjalankan aplikasi. Yang perlu dilakukan hanyalah npm start. Catatan Jangan lupa bahwa Anda harus menjalankan npm install sebelum Anda dapat melakukan npm start, karena begitulah cara semua dependensi kode sisi server diinstal. Saya yakin Anda sudah sangat menyadari hal ini sekarang, tetapi tidak ada salahnya untuk diingatkan.

Server.Js Pembukaan Volley: Impor Dan Variabel

File sumber utama (dan, pada kenyataannya, satu-satunya) untuk server adalah server.js yang diberi nama tepat, dan kode ini dimulai seperti kebanyakan kode Node, dengan beberapa impor.

```
const fs = require("fs");
const lodash = require("lodash");
```

Variabel fs akan menyimpan referensi ke Node File System API bawaan. Kita akan menggunakan ini untuk membaca berkas pertanyaan yang terlihat sebelumnya. Impor lodash, tentu saja, adalah pustaka lodash. Setelah impor, saatnya membangun server dan menghubungkan socket.io ke sana. Saya telah melakukannya dalam satu baris kode ini (sesuatu yang biasanya tidak saya rekomendasikan, tetapi variasi adalah bumbu kehidupan, jadi ini sedikit bumbu untuk Anda).

```
Const io
require("socket.io")(require("http").createServer(function(){}).listen(80));
```

Jika Anda menguraikannya, Anda akan melihat bahwa ia menciptakan server HTTP, seperti yang Anda lihat sebelumnya dalam contoh server sederhana. Namun, dalam kasus ini, server yang dibuat tidak akan menangani permintaan, tetapi kita masih harus menyediakan fungsi kosong untuk memenuhi kontrak createServer(). Server tersebut kemudian diteruskan ke konstruktor socket.io (yang diimpor secara anonim, karena, seperti impor http, ia tidak diperlukan di luar baris ini), yang merupakan hal yang mengaitkan socket.io ke server dan membuatnya berfungsi. Pada dasarnya, socket.io memanfaatkan server HTTP yang mendasarinya, memperluasnya untuk menangani koneksi WebSocket. Setelah itu muncul serangkaian variabel yang akan kita perlukan.

```
Const players = {};
```

Ini menyimpan objek, satu mewakili setiap pemain yang berpartisipasi. Objek ini diberi kunci dengan ID pemain unik yang akan dibuat saat pemain terhubung ke server.

```
let inProgress = false;
```

Diharapkan nama variabel dapat mendokumentasikan dirinya sendiri. Ini adalah tanda yang memberi tahu kode apakah permainan sedang berlangsung.

```
let questions = null;
let question = null;
let questionForPlayers = null;
```

Ketiga variabel ini menyimpan pertanyaan yang dibaca dari questions.json, pertanyaan saat ini, dan pertanyaan dalam bentuk yang sedikit berbeda untuk para pemain. Jangan terlalu khawatir tentang ini, dan mengapa pertanyaan tersebut tampaknya disimpan dua kali. Itu semua akan menjadi jelas dalam waktu dekat.

```
let questionStartTime = null;
```

Ingatkah saya mengatakan bahwa interval yang dibutuhkan pemain untuk menjawab pertanyaan menjadi faktor dalam skornya? Nah, variabel ini menyimpan waktu pertanyaan saat ini dikirim ke para pemain, dan menggunakannya, server dapat menentukan berapa lama waktu yang dibutuhkan setiap pemain untuk menjawab pertanyaan.

```
let numberAsked= 0;
```

Terakhir, numberAsked adalah berapa banyak pertanyaan dari total jumlah pertanyaan yang telah ditanyakan. Ini akan digunakan untuk memberi tahu admin ketika tidak ada lagi pertanyaan yang tersisa untuk ditanyakan. Daftar kecil variabel ini mewakili jumlah total status yang dibutuhkan kode server untuk melakukan tugasnya. Tidak banyak sama sekali, bukan?

16.2 FUNGSI UTILITAS

Selain beberapa variabel tersebut, ada dua fungsi utilitas yang akan kita perlukan di beberapa tempat berbeda. Ini adalah bagian kode berikutnya yang akan Anda temukan saat memeriksa berkas sumber ini.

Newgamedata()

Setiap kali pemain baru terhubung, atau permainan baru dimulai, kita harus menyetel ulang beberapa status untuk setiap pemain. Ini mewakili data tentang apa yang telah terjadi sejauh ini selama permainan saat ini (sebagian besar). Ini adalah objek sederhana, objek gameData, seperti yang saya sebut, dan fungsi newGameData() digunakan untuk membuatnya.

```
function newGameData(){
  return {right: 0,wrong: 0, totalTime: 0, fastest: 999999999, slowest: 0,
  average: 0, points: 0, answered: 0, playerName: null
  };
}
```

Atribut objek yang dibangun harus cukup jelas: berapa banyak pertanyaan yang dijawab dengan benar dan salah oleh pemain, total waktu yang dibutuhkan untuk menjawab, waktu tercepat dan paling lambat yang dijawab pemain, waktu rata-rata yang dibutuhkan untuk menjawab, berapa banyak poin yang dimiliki pemain, dan berapa banyak pertanyaan yang telah dijawabnya. playerName juga disimpan di sini, meskipun secara konseptual tidak sama dengan yang lain, dan itu dilakukan hanya untuk membuatnya lebih mudah diakses di beberapa tempat lain dalam kode nanti.

Calculateleaderboard()

Ingat kembali pada tangkapan layar sebelumnya bahwa ketika pemain sedang menunggu pertanyaan, mereka berada di layar papan peringkat yang menunjukkan poin dan peringkat pemain saat ini. Satu fungsi, calculateLeaderboard(), bertanggung jawab untuk menghasilkan data di balik tampilan itu.

```
function calculateLeaderboard(){
  const playersArray = [];
    for (const playerId in players){
    if (players.hasOwnProperty(playerID)) {
      const player = players[playerId];
      playersArray.push({playerId: playerId, playerName:
      player.playerName, points: player.points});
    }
  }

  playersArray.sort((inA, inB) => {
    const pointsA = inA.points;
    const pointsB = inB.points;
      if (pointsA > pointsB) {return -1;}
        else if (pointsA < pointsB) {return 1;}
        else {return 0;}
```

```
    });

    return playersArray;
}
```

Blok kode pertama, for loop, bertanggung jawab untuk mengambil objek pemain dan membuat array darinya. Karena papan peringkat adalah FlatList, dan FlatList didukung oleh array, itulah yang kita butuhkan (pemain adalah objek, karena membuat penulisan semua kode lainnya menjadi mudah, dan ini adalah satu-satunya waktu kita membutuhkannya sebagai array, jadi masuk akal untuk melakukan konversi di sini). Namun, sebagai bagian dari transformasi, kita hanya memerlukan beberapa informasi untuk merender papan peringkat, jadi daripada hanya mendorong objek yang ada ke array, objek minimal baru dibuat sebagai gantinya. Potongan kode kedua adalah pengurutan sederhana, berdasarkan poin, sehingga array sekarang dalam urutan poin menurun, persis seperti yang Anda harapkan dari daftar klasemen.

16.3 PENANGANAN PESAN PEMAIN

Hal berikutnya yang harus kita lakukan adalah menyediakan socket.io fungsi yang akan menangani berbagai pesan yang dapat dipancarkan ke server. Dan apa saja pesan tersebut, Anda bertanya? Berikut daftarnya:

- Validateplayer: Dikirimkan saat pemain pertama kali terhubung ke server
- Submitanswer: Dikirimkan saat pemain mengirimkan jawabannya ke pertanyaan saat ini
- Adminnewgame: Dikirimkan saat admin memulai permainan baru
- Adminnextquestion: Dikirimkan saat admin memicu pertanyaan berikutnya
- Adminendgame: Dikirimkan saat admin mengakhiri permainan

Seperti yang Anda lihat, tidak banyak pesan. Masing-masing adalah fungsi, dan fungsi tersebut harus disediakan ke objek io di dalam pengendali pesan koneksi. Pesan koneksi akan dikirim secara otomatis oleh klien saat terhubung ke server, dan API socket.io mengharuskan semua pengendali pesan didefinisikan di dalam pengendali untuk pesan tersebut. Jadi, kita punya ini:

```
io.on("connection", io => {
  ...
});
```

dan sebagai pengganti elipsis terdapat lima panggilan ke io.on(), yang masing-masing meneruskan nama pesan (dari daftar sebelumnya) dan kemudian fungsi yang menangani pesan tertentu tersebut. Namun, bahkan sebelum itu, ada satu pernyataan lain yang akan Anda temukan di dalam pengendali pesan koneksi:

```
io.emit("connected", {});
```

Ini memancarkan pesan yang terhubung kembali ke pemain. Jadi, urutannya adalah

- Aplikasi klien pemain terhubung, memancarkan pesan koneksi ke server. (Ini terjadi secara otomatis saat objek socket.io dibuat dalam kode klien, seperti yang akan Anda lihat nanti.)
- Penangan pesan koneksi memancarkan pesan yang terhubung ke pemain. (Perhatikan bahwa, secara umum, tidak diperlukan untuk memancarkan pesan seperti ini. socket.io tidak memerlukannya, tetapi alur startup aplikasi RNTrivia memerlukannya, yang akan dijelaskan di Bab 6.)
- Penangan pesan koneksi membuat lima panggilan io.on() untuk menghubungkan penangan untuk masing-masing dari lima pesan yang diperlukan agar seluruh kekacauan ini berfungsi.

Setelah penangan pesan koneksi selesai, server siap menangani semua pesan yang diperlukan dari pemain. Sekarang, mari kita lihat penangan untuk masing-masing pesan secara bergantian.

Validateplayer

Pesan pertama yang akan ditangani adalah pesan validatePlayer. Hal ini dipicu oleh klien yang menangani pesan terhubung yang dipancarkan oleh server sebagai respons terhadap pesan koneksi otomatis ke server. Hal ini berfungsi sebagai semacam jabat tangan: klien mengatakan "halo" saat objek socket.io dibuat di sana, yang memicu pesan koneksi, lalu server merespons dengan mengatakan "Oh, halo juga!" dengan memancarkan pesan terhubung, yang kemudian ditanggapi oleh klien dengan mengatakan "Hai, bisakah Anda memvalidasi pemutar ini untuk saya?" dengan memancarkan pesan validatePlayer, yang ditangani oleh kode ini:

```
io.on("validatePlayer", inData => {
  try {
    const responseObject = {inProgress: inProgress,
    gameData: newGameData(),leaderboard: calculateLeaderboard(), asked:
    numberAsked
  };
    responseObject.gameData.playerName = inData.playerName;
    responseObject.playerID = `pi_${new Date().getTime()}`;
    for (const playerID in players) {
      if (players.hasOwnProperty(playerID)) {
      if (inData.playerName === players[playerID].playerName) {
    responseObject.gameData.playerName += `_${new Date().getTime()}`;
      }
    }
  }
  players[responseObject.playerID] = responseObject.gameData;
  io.emit("validatePlayer", responseObject);
  } catch (inException) {
    console.log(`${inException}`);
  }
});
```

Pertama, Anda akan menemukan bahwa semua pengendali pesan dibungkus dalam blok try...catch seperti ini. Tujuannya adalah untuk memastikan bahwa server tetap berjalan, bahkan jika terjadi masalah yang melibatkan satu pemain. Tidak ada gunanya mematikan seluruh server untuk semua orang! Namun karena ini merupakan kondisi yang tidak terduga, hal-hal yang tidak dapat ditangani oleh kode secara wajar, maka yang dilakukan hanyalah mencatat pengecualian ke konsol server. Hal pertama yang harus dilakukan adalah membuat objek yang akan dikembalikan ke pemain.

Ini akan menyediakan semua informasi yang dibutuhkan saat ini. Ini termasuk apakah permainan sedang berlangsung (inProgress), contoh gameData baru (yang menyediakan status untuk entri baru ke dalam permainan), kedudukan papan peringkat saat ini, dan jumlah pertanyaan yang diajukan sejauh ini selama permainan ini. playerName juga ditambahkan ke objek gameData saat ini (seperti yang disebutkan sebelumnya, ini akan membuat penulisan kode klien sedikit lebih mudah, dengan membuat nama tersebut tersedia secara lebih langsung). Selanjutnya, ID unik untuk pemain ini dibuat, yang hanya berupa karakter pi_ diikuti oleh waktu saat ini dalam milidetik.

Bukan cara terbaik untuk menghasilkan nilai unik, saya setuju, tetapi cukup baik untuk kebutuhan aplikasi ini (artinya, tidak mungkin dua orang akan mendapatkan ID yang sama, kecuali Anda memiliki banyak pemain yang mencoba masuk pada waktu yang sama). Setelah itu, bagian "validasi" dari validatePlayer() dimulai, dengan memeriksa apakah nama yang diberikan klien sebagai bagian dari argumen inData sudah digunakan. Jika sudah, nama baru dibuat, dengan menambahkan waktu saat ini ke nama tersebut. Ini hanya memerlukan pengulangan atribut objek pemain dan memeriksa atribut playerName masing-masing, untuk mencari kecocokan.

Kemudian objek gameData yang dibuat sebelumnya ditambahkan ke objek pemain, menggunakan playerID sebagai kuncinya. Objek ini berisi semua informasi terkait yang perlu disimpan server tentang setiap pemain (ingat bahwa ID unik, yang jelas juga sangat relevan, adalah kunci atribut objek). Terakhir, pesan validatePlayer dipancarkan ke klien. Perhatikan bahwa tidak ada masalah menggunakan nama pesan yang sama di sini, karena yang satu dipancarkan oleh klien dan satunya lagi oleh server, dan yang lainnya menanggapinya (atau mengabaikannya, jika tidak ada pekerjaan yang harus dilakukan sebagai tanggapan).

Saya suka simetri melihat nama pesan yang sama berjalan ke kedua arah ketika pada dasarnya itu adalah model permintaan-tanggapan, seperti klien mengirim validatePlayer, dan sesuatu diharapkan sebagai balasannya (secara konseptual pesan tidak pernah "mengembalikan" nilai, per se, ketika berhadapan dengan socket.io), yang kebetulan adalah pesan validatePlayer, tetapi dari server kali ini. Bayangkan seluruh siklus dari klien ke server dan kembali lagi sebagai panggilan metode, dan menurut saya masuk akal untuk menamainya sama. (Tetapi tidak ada yang melampaui preferensi saya sendiri yang mengatakan bahwa itu harus terjadi. Untuk lebih jelasnya, socket.io tidak peduli, begitu pula Node.)

Submitanswer

Pesan berikutnya yang dapat dipancarkan oleh klien dan harus ditangani oleh server

adalah pesan submitAnswer, yang dikirimkan saat pengguna memilih jawaban atas pertanyaan. Ini mungkin merupakan penanganan pesan yang paling rumit dari semuanya, tetapi relatif terhadap yang lain, jadi masih belum terlalu rumit.

```
    io.on("submitAnswer", inData => {
    try {
        const gameData = players[inData.playerID];
        let correct = false;
        gameData.answered++;

        if (question.answer === inData.answer) {

            players[inData.playerID].right++;
            players[inData.playerID].wrong--;

            const time = new Date().getTime() - questionStartTime;
            gameData.totalTime = gameData.totalTime + time;
            if (time > gameData.slowest){
                gameData.slowest = time;
            }

    if (time < gameData.fastest) {
      gameData.fastest = time;
}

gameData.average = Math.trunc(gameData.totalTime / numberAsked);

const maxTimeAllowed = 15;
gameData.points = gameData.points + (maxTimeAllowed * 4);
gameData.points = gameData.points - Math.min(Math.max(
    Math.trunc(time / 250), 0), (maxTimeAllowed * 4)
);

gameData.points = gameData.points + 10;
correct = true;
}

    io.emit("answerOutcome", { correct: correct, gameData: gameData,
     asked: numberAsked, leaderboard: calculateLeaderboard()
  });
 } catch (inException) {
   console.log(`${inException}`);
  }
});
```

Langkah pertama adalah mendapatkan objek gameData dari koleksi pemain untuk pemain ini, menggunakan playerID yang dikirim sebagai bagian dari inData. Sekarang kita tahu pemain mana yang sedang kita hadapi. Selanjutnya, tanda correct ditetapkan ke false, karena kita berasumsi pemain tersebut salah sejak awal, hingga ditentukan sebaliknya. Jumlah pertanyaan yang telah dijawab pemain selanjutnya dinaikkan pada objek gameData untuk pemain ini.

Sekarang logika pertama muncul, dan itu adalah bagian yang sangat penting: apakah dia mendapatkan jawaban yang benar? Ini adalah perbandingan sederhana antara jawaban yang dia kirim dengan jawaban untuk objek pertanyaan saat ini.

Jika dia menjawab dengan benar, maka langkah pertama adalah menambah jumlah pertanyaan yang dia jawab dengan benar dan mengurangi jumlah yang dia jawab salah pada gameData. Ingat kembali bahwa ketika pertanyaan dikirim, jumlah yang salah bertambah. Dengan begitu, jika dia tidak menjawab pada saat pertanyaan berikutnya diajukan (atau permainan berakhir), maka pertanyaan ini dianggap sebagai jawaban yang salah. Namun, kita harus membatalkannya di sini, karena dia menjawab dengan benar, maka dari itu pengurangan jawaban yang salah.

Setelah itu, saatnya untuk melihat berapa lama dia menjawab. Itu adalah masalah sederhana dengan mengurangi waktu pertanyaan dimulai dari waktu saat ini. Ini ditambahkan ke totalTime pemain, dan kemudian pemeriksaan dilakukan untuk melihat apakah waktu ini adalah waktu tercepat atau terlambatnya, dan nilai waktu baru ini disimpan di gameData, jika demikian. Akhirnya, waktu rata-rata dihitung ulang dan diperbarui di gameData.Selanjutnya, jumlah poin yang didapat pemain untuk jawaban ini dihitung. Logika di sini relatif mudah dan dirancang agar seadil mungkin untuk semua pemain di semua tingkat pengetahuan.

Logika ini didasarkan pada jumlah waktu maksimum yang diizinkan untuk menjawab pertanyaan dan jumlah poin maksimum yang sesuai yang darinya sejumlah angka dikurangi, berdasarkan berapa banyak waktu yang dibutuhkan pemain untuk menjawab. Jawaban yang benar dimulai dengan memberikan pemain 60 poin, karena maxTimeAllowed adalah 15, dan nilai tersebut dikalikan dengan 4. Namun, untuk setiap seperempat detik yang dibutuhkan untuk menjawab, kami kurangi 1 poin, yang membatasi kerugian pada poin maksimal yang bisa didapatkan pemain. Jadi, pada akhirnya, jika ia membutuhkan lebih dari maxTimeAllowed detik untuk menjawab, ia tidak akan mendapatkan poin. Jika tidak, ia akan mendapatkan sesuatu yang kurang dari atau sama dengan maxTimeAllowed*4.

Namun, menurut saya agak tidak adil untuk mengambil semua poinnya jika ia menjawab dengan benar, jadi pada akhirnya, kami memberinya 10 poin, terlepas dari berapa lama ia menjawab. Jadi, dalam analisis akhir, semua ini berarti bahwa seorang pemain bisa mendapatkan 10–60 poin untuk jawaban yang benar, dan waktu yang dibutuhkan untuk menjawablah yang menentukan di mana dalam rentang itu ia berada. Hal itu juga memberi kesempatan kepada pemain yang kurang berpengetahuan, karena jika mereka mendapatkan satu atau dua pertanyaan yang langsung mereka ketahui jawabannya, mereka dapat mengejar pemain yang mungkin menjawab lebih banyak pertanyaan dengan benar secara keseluruhan tetapi membutuhkan waktu untuk menjawab masing-masing. Ini mungkin bukan algoritme yang sempurna, tetapi cukup adil untuk aplikasi ini.

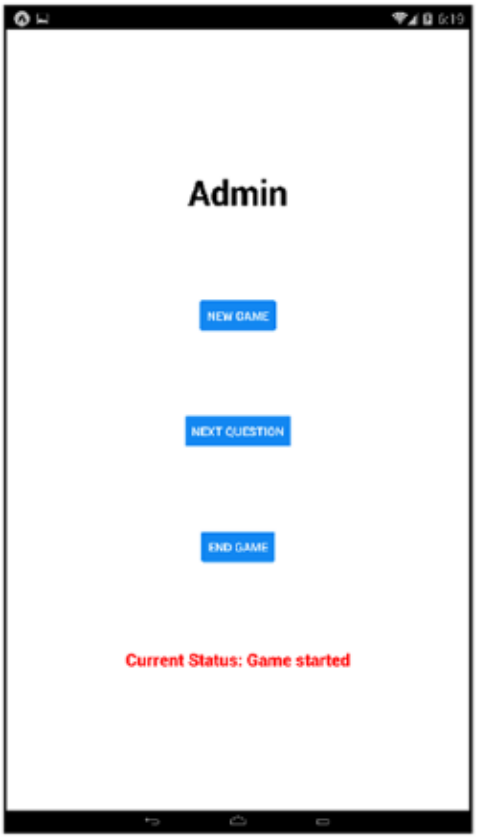
Penanganan Pesan Admin

Dua pesan yang dibahas di bagian terakhir adalah pesan yang dipancarkan pemain untuk masuk ke dalam permainan dan untuk mengirimkan jawaban. Namun, ada serangkaian pesan lain yang hanya dapat dikendalikan oleh pengguna admin untuk permainan. Saat pertama kali memulai aplikasi klien, Anda diminta untuk memasukkan nama Anda melalui

modal, seperti yang dapat Anda lihat pada Gambar 16.3. Anda akan melihat ada juga sakelar untuk menunjukkan bahwa Anda adalah pengguna admin.

Seperti yang akan Anda lihat di berikutnya, saat kita melihat kode klien, hasil dari membalik sakelar ini adalah serangkaian pengendali pesan socket.io yang berbeda diaktifkan, yang khusus untuk admin, dan yang lainnya untuk pemain diabaikan. Saat itu terjadi, pengguna melihat layar baru, dan hanya layar itu, bukan layar yang dilihat pemain. Layar admin ini tidak akan memenangkan kontes kecantikan, tetapi memang tidak harus demikian. Tujuannya hanya untuk memberi pengguna kendali atas permainan, jadi sangat sederhana, seperti yang ditunjukkan Gambar 16.4.

Gambar 16.3 Modal Prompt Nama (Versi Android)



Gambar 16.4 Layar Admin Yang Sama Sekali Tidak Mengesankan Ini Benar-Benar Menyelesaikan Tugasnya

Masing-masing tombol yang Anda lihat di sini memicu salah satu dari tiga pesan berikut untuk dipancarkan, jadi mari kita lihat pengendali sisi server untuk pesan-pesan tersebut.

Adminnewgame

Saat admin ingin memulai permainan baru, pesan adminNewGame yang diberi nama tidak kreatif dipancarkan, untuk melakukan semua pekerjaan yang terlibat dalam menyiapkan permainan baru.

```
io.on("adminNewGame", () => {

    try {
        question = null;
        questionForPlayers = null;
        numberAsked = 0;
        inProgress = true;

       questions=(JSON.parse(fs.readFileSync("questions.json", "utf8"))).
         questions;

        for (const playerID in players) {
            if (players.hasOwnProperty(playerID))
      {const playerName = players[playerID].playerName;
                players[playerID] = newGameData();
```

```
            players[playerID].playerName = playerName;
          }
        }
        const responseObject = {inProgress: inProgress, question: null,
        playerID: null, gameData: newGameData(), asked: numberAsked,
        leaderboard: calculateLeaderboard()
        };
        const gd = newGameData();
        gd.asked = 0;
        io.broadcast.emit("newGame", responseObject);
        io.emit("adminMessage", { msg: "Game started" });

        } catch (inException) {
          console.log(`${inException}`);
        }
});
```

Pertama, semua variabel "pelacakan" yang terkait dengan permainan disetel ulang, jadi kita mulai tanpa pertanyaan saat ini (dan questionForPlayers), nol numberAsked, dan tanda inProgress disetel ke true. Selanjutnya, file questions.json dibaca. Di sinilah Node File System API masuk, melalui variabel fs. API ini menyediakan berbagai macam metode, salah satu yang paling sederhana adalah readFileSync(). Ini akan membaca file yang ditentukan secara sinkron. (Ada juga metode readFile() yang akan melakukannya secara asinkron, tetapi kemudian Anda harus menyediakan panggilan balik dan kode yang sesuai, tetapi dalam kasus ini, kecepatan bukanlah masalah, jadi menahan utas dengan versi panggilan yang sinkron dapat diterima dan membuat kode lebih alami.)

File dibaca dan diteruskan ke JSON.parse(), untuk mendapatkan objek darinya, dan referensi ke sana disimpan dalam pertanyaan. Setelah itu, semua pemain yang saat ini dikenal server harus menyetel ulang objek gameData mereka, karena mereka mungkin telah berpartisipasi dalam permainan sebelumnya. Itu adalah iterasi sederhana atas kunci pemain dan panggilan ke newGameData() untuk masing-masing (ditambah dengan menambahkan playerName ke objek gameData tersebut). Selanjutnya, kita harus memberi tahu semua klien yang terhubung tentang permainan baru tersebut.

Jadi, responseObject dibangun, yang akan Anda lihat tampak sangat mirip dengan apa yang dilakukan saat pemain pertama kali terhubung. Itu memang sudah dirancang: keadaan setelah ini seharusnya tampak sangat mirip saat pemain pertama kali terhubung dan memasuki permainan, karena, secara konseptual, keduanya berada dalam situasi yang sama, sejauh menyangkut status server dan klien. Langkah terakhir adalah mengirimkan pesan adminMessage kembali ke klien, untuk memberi tahu bahwa permainan telah dimulai. Pesan yang diteruskan sebagai muatan pesan ini akan ditampilkan di layar, untuk mengonfirmasi permintaan permainan baru mereka.

AdminnextQuestion

Fungsi berikutnya yang dapat dilakukan admin adalah memicu pertanyaan untuk dikirim ke pemain. Ini mengirimkan pesan adminNextQuestion, yang ditangani oleh fungsi

pengendali ini:

```
io.on("adminNextQuestion", () => {
    try {
        if (!inProgress) {
            io.emit("adminMessage", { msg: "There is no game in progress" });
            return;
          }
        if (questions.length === 0) {
            io.emit("adminMessage", { msg: "There are no more questions" });
            return;
        }
        for (const playerID in players) {
            if (players.hasOwnProperty(playerID)) {
                players[playerID].wrong++;
          }
        }
      let choice = Math.floor(Math.random() * questions.length);
      question = questions.splice(choice, 1)[0];
      questionForPlayers = { question: question.question, answers: [] };

const decoys = question.decoys.slice(0);
for (let i = 0; i < 5; i++) {
    let choice = Math.floor(Math.random() * decoys.length);
    questionForPlayers.answers.push(decoys.splice(choice, 1)[0]);
}

      questionForPlayers.answers.push(question.answer);
      questionForPlayers.answers= lodash.shuffle(questionForPlayers.answers);
      numberAsked++;
      questionStartTime = new Date().getTime();

      io.broadcast.emit("nextQuestion", questionForPlayers);
      io.emit("adminMessage", { msg: "Question in play" });
} catch (inException) {
    console.log(`${inException}`);
 }
});
```

Selanjutnya, kita harus melakukan dua "pemeriksaan idiot." Pertama, jika tidak ada permainan yang sedang berlangsung, pesan yang sesuai akan dikirim ke klien, dengan mengirimkan pesan adminMessage. Demikian pula, jika tidak ada lagi pertanyaan yang tersisa untuk ditanyakan, admin juga akan diberi tahu tentang hal itu. Admin diharapkan akan mengirimkan pesan adminNewGame pada kasus pertama dan pesan adminEndGame pada kasus kedua, tetapi itu sepenuhnya tergantung pada admin. Selanjutnya, jumlah yang salah akan bertambah untuk semua pemain. Seperti yang Anda lihat sebelumnya, kita mulai dengan asumsi setiap pemain menjawab pertanyaan yang salah hingga server menentukan sebaliknya, dan jumlah ini akan berkurang jika pemain menjawab dengan benar, tetapi default ini memungkinkan server untuk menganggap tidak ada jawaban sama sekali sebagai jawaban yang salah.

Selanjutnya, sebuah pertanyaan dipilih secara acak dari deretan pertanyaan. Pertanyaan itu kemudian dihapus dari deretan tersebut. Dengan cara ini, kita tidak memiliki kode unik untuk melacak pertanyaan apa yang telah dan belum ditanyakan—jika masih ada dalam deretan tersebut, berarti pertanyaan itu belum ditanyakan. Sesederhana itu. Karena file questions.json dibaca ulang setiap kali permainan dimulai, array questions akan disusun ulang pada saat itu, jadi tidak perlu khawatir tentang fungsi pengaturan ulang khusus untuk permainan berikutnya. Sebagai bagian dari pilihan ini, objek questionForPlayers baru dibuat.

Itu karena objek dalam array questions berisi lebih banyak informasi daripada yang dibutuhkan aplikasi klien (termasuk, yang terpenting, jawaban yang benar). Bagaimanapun, kita tidak ingin ada yang mencoba memantau lalu lintas antara klien dan server dan berbuat curang dengan melihat jawaban yang benar. Jadi, kita akan membuat objek baru hanya dengan apa yang benar-benar dibutuhkan klien, daripada mengambil rute malas dengan hanya mengembalikan objek question yang ada. (Saya bisa saja menghapus dari objek yang ada apa yang tidak diperlukan dan mengirimkannya enam dari satu, setengah lusin dari yang lain, saya kira.) Tugas berikutnya adalah memilih lima dari sepuluh jawaban umpan.

Logika yang sama seperti "pilih satu lalu hapus" akan digunakan, tetapi dalam kasus ini, saya tidak ingin mengubah array asli, jadi array tersebut dikloning terlebih dahulu dengan panggilan slice(0). Kemudian, enam pilihan acak dibuat, dan setiap kali menghapus umpan dari array yang diklon. Setelah umpan ditambahkan ke questionForPlayers, kita juga harus menambahkan jawaban yang benar. Jika tidak, ini tidak akan menjadi permainan yang menarik. Namun, setelah melakukan push() pada array questionForPlayers.answers, tentu saja, itu adalah jawaban terakhir, dan akan selalu demikian, jadi akan cukup mudah bagi para pemain untuk mengetahuinya. Jadi, setelah itu, satu fungsi lodash yang saya sebutkan di awal yang kita butuhkan digunakan: shuffle(). Ini hanya mengacak array, jadi jawaban yang benar tidak selalu berada di slot yang sama.

Sekarang, kita memiliki beberapa tugas tata graha lagi yang harus diselesaikan. Pertama, variabel numberAsked dinaikkan, karena menentukannya dari panjang array pertanyaan agak sulit (kita perlu melacak berapa banyak pertanyaan yang harus dimulai, tetapi menurut saya lebih wajar untuk menghitungnya satu per satu, setiap kali pertanyaan baru dipilih). Kemudian waktu saat ini disimpan, sehingga kita dapat menentukan berapa lama waktu yang dibutuhkan setiap pemain untuk menjawab. Langkah terakhir adalah mengirimkan dua pesan, satu untuk pemain dan satu untuk admin.

Pesan nextQuestion dikirim ke pemain dan memicu aplikasi klien untuk menampilkan layar pertanyaan. Pesan adminMessage dikirim ke admin, tentu saja, dan mengonfirmasi kepada mereka bahwa pertanyaan sekarang sedang dimainkan. adminEndGame Penanganan pesan sisi server terakhir yang akan diperiksa, dan bagian kode terakhir di server.js, sebenarnya, adalah untuk menangani pesan adminEndGame yang dikirimkan pengguna admin untuk mengakhiri permainan saat ini.

```
io.on("adminEndGame", () => {
    try {
     if (!inProgress) {
```

```
            io.emit("adminMessage", { msg: "There is no game in progress" });
               return;
            }

            const leaderboard = calculateLeaderboard();
            io.broadcast.emit("endGame", { leaderboard: leaderboard });

            inProgress = false;
            questions = null;
            question = null;
            questionForPlayers = null;
            questionStartTime = null;
            numberAsked = 0;

            for (const playerID in players) {
                if (players.hasOwnProperty(playerID)) {
                    const playerName = players[playerID].playerName;
                    players[playerID] = newGameData();
                    players[playerID].playerName = playerName;
                }
            }
            io.emit("adminMessage", { msg: "Game ended" });

    } catch (inException) {
        console.log(`${inException}`);
    }
    });
```

Pertama, pemeriksaan idiot. Apakah sebenarnya ada permainan yang sedang berlangsung? Jika tidak, adminMessage dipancarkan untuk memberi tahu pengguna admin. Selanjutnya, papan peringkat dihitung ulang melalui panggilan ke calculateLeaderboard(). Ini menunjukkan kedudukan akhir untuk permainan, jadi ini dikirim ke semua pemain sebagai muatan untuk pesan endGame yang dipancarkan berikutnya. Setelah itu, semua variabel yang terkait dengan status permainan di server disetel ulang. Ini sebagian besar berlebihan (dan juga sebagian besar tidak perlu), karena sebagian besar dari ini akan dilakukan lagi, jika dan ketika permainan baru dimulai.

Namun, ini dilakukan di sini terutama agar jika pengguna admin mencoba mengirim pertanyaan baru atau mengakhiri permainan lagi, para pengendali akan dapat memberi tahu mereka tentang hal itu dengan benar. (Saya juga suka selalu memiliki status yang diketahui, jadi saya tidak keberatan sedikit menyetel ulang beberapa hal, untuk memastikannya.) Bagian dari itu adalah menyetel ulang gameData untuk semua pemain juga, di sinilah loop berperan. Akhirnya, adminMessage dipancarkan, memberi tahu admin bahwa permainan telah berakhir, sebagai konfirmasi. Dan, dengan itu, server selesai!

16.4 MEMBANGUN KLIEN RNTRIVIA DENGAN REACT NATIVE

Pada pembahasan awal bab ini, kita fokus membangun sisi server untuk aplikasi RNTrivia. React Native sendiri belum terlibat dalam proses tersebut, namun sekarang saatnya

untuk mengembangkan sisi klien, yaitu aplikasi React Native RNTrivia. Meskipun aplikasi ini memerlukan lebih banyak kode dibandingkan dengan server, kode yang ditulis tidak terlalu rumit, karena server sudah menangani sebagian besar proses yang lebih kompleks. Salah satu keuntungan dari kesederhanaan ini adalah memberi saya kesempatan untuk memperkenalkan beberapa konsep baru tanpa terasa berlebihan. Sebelum kita melangkah lebih jauh ke dalam kode, mari kita mulai dengan memahami bagaimana aplikasi dikonfigurasi untuk React Native dan Expo.

16.5 STRUKTUR APLIKASI DAN DESAIN KESELURUHAN

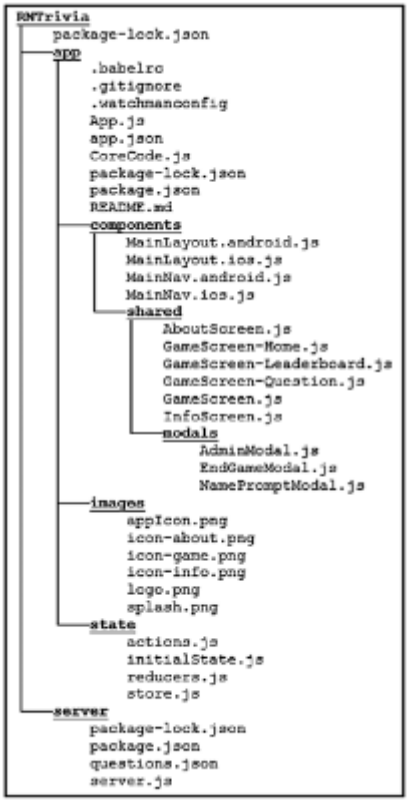
Pertama, mari kita bahas tentang struktur keseluruhan RNTrivia. Ini hadir dalam dua bentuk: tata letak kode sumber pada sistem file (struktur direktori, terutama) dan desain aplikasi, dalam hal layar dan semacamnya. Saya akan membahas yang pertama terlebih dahulu.

Tata Letak Sumber

Kode sisi server memiliki struktur yang sederhana: hanya satu berkas sumber (tidak termasuk berkas "dukungan", seperti package.json) dalam satu direktori. Namun, struktur kode sisi klien sedikit lebih rumit, seperti yang ditunjukkan Gambar 6.1. Di direktori root, Anda memiliki beberapa hal yang biasa: App.js sebagai titik masuk utama, app.json untuk menjelaskan aplikasi ke Expo, package.json (dan saudaranya package-lock.json) untuk manajemen dependensi (dan konfigurasi untuk keperluan NPM dan pengemasan), README.md (dihasilkan oleh react-native-create-app, meskipun tidak penting untuk keperluan kita di sini), dan beberapa file tersembunyi yang membawa konfigurasi untuk berbagai alat yang digunakan Expo dan React Native.

Anda juga akan melihat file CoreCode.js, dan seperti namanya, di situlah kode "inti" aplikasi ini akan berada (seperti yang akan Anda lihat sebentar lagi, App.js sangat jarang di aplikasi ini, bahkan lebih jarang daripada di Restaurant Chooser). Di luar direktori root, Anda akan melihat bahwa ada sejumlah subdirektori. Direktori gambar, sama seperti di Restaurant Chooser, adalah tempat penyimpanan gambar apa pun yang dibutuhkan aplikasi. Untuk aplikasi ini, saya telah meletakkan semua gambar di sana, termasuk ikon aplikasi dan gambar layar pembuka.

Saya sarankan untuk menjaga direktori root Anda sebersih mungkin, jadi tata letak ini, menurut saya, lebih baik daripada Restaurant Chooser, yang memiliki ikon dan gambar layar pembuka di dalamnya. Anda juga akan menemukan tiga gambar ikon di sana untuk tab yang akan menjadi model navigasi utama kita (yah, untuk iOS, tetapi saya akan membahasnya lebih lanjut).



Gambar 16.5 Struktur Direktori Aplikasi Klien RN trivia

Lalu ada direktori komponen, seperti di Restaurant Chooser, tempat semua kode komponen React Native berada. Namun, tidak seperti Restaurant Chooser, semua kode ada di sana. Komponen layar tidak dibagi ke dalam direktori mereka sendiri, tetapi ada struktur direktori yang lebih dalam yang perlu kita bicarakan sekarang. Di Restaurant Chooser, Anda akan ingat bahwa tidak banyak perbedaan antara iOS dan Android tidak banyak juga penargetan kode dan apa yang ada di sana ditangani menggunakan Platform API dan beberapa nilai info perangkat yang disediakan oleh React Native dan Expo. Untuk aplikasi sederhana, itu saja yang sering kali Anda butuhkan, karena tampilan, nuansa, dan fungsinya hampir seluruhnya sama di kedua platform, jadi tidak perlu membuat cabang kode apa pun.

Namun, bagaimana jika Anda ingin tata letak aplikasi, dan bahkan, mungkin, fungsinya, agar jauh lebih berbeda antara kedua platform? Jika Anda menduga akan menjadi monoton jika memiliki banyak logika kondisional menggunakan Platform API untuk membuat cabang di antara jalur kode, Anda benar. Terutama dalam kasus aplikasi yang perlu memiliki model navigasi yang berbeda antara keduanya (biasanya dilakukan untuk mematuhi pedoman desain aplikasi setiap platform), Anda mungkin menginginkan cara yang lebih baik, dan ternyata ada satu. Misalnya Anda memiliki komponen, sebut saja MyComponent, dan komponen tersebut ada dalam file MyComponent.js. Jadi, Anda

```
import MyComponent from "./MyComponent";
```

dalam beberapa file sumber untuk menggunakannya. Sekarang, misalnya Anda harus

membuat MyComponent terlihat sangat berbeda di iOS daripada di Android. Daripada memasukkan logika percabangan apa pun ke dalam kode tersebut, Anda dapat membuat salinan MyComponent.js, menamainya MyComponent.android.js, lalu mengganti nama aslinya menjadi MyComponent.ios.js, dan coba tebak apa yang terjadi? Saat pengemas React Native membuat bundel JavaScript untuk aplikasi Anda, pengemas tersebut akan memilih file yang sesuai secara otomatis, berdasarkan platform tempat aplikasi Anda berjalan. Sekarang, di dalam kedua file tersebut, Anda tidak perlu melakukan hal khusus apa pun, Anda cukup menulis komponen, seperti yang Anda lakukan pada komponen lainnya, tetapi khusus untuk setiap platform.

Saat aplikasi dibuat, versi yang benar akan digunakan. Cukup bagus, bukan? Nah, untuk RNTrivia, saya memutuskan sejak awal bahwa saya ingin menggunakan model navigasi tab yang sama seperti yang digunakan untuk Restaurant Chooser, tetapi hanya untuk iOS. Untuk Android, saya ingin menggunakan baki navigasi (alat yang memungkinkan Anda menggeser dari sisi layar untuk menampilkan menu navigasi), karena itu lebih umum di Android. Meskipun kedua model navigasi dapat berfungsi di kedua OS, menggunakan model yang lebih umum di masing-masing OS memberikan pengalaman yang lebih asli bagi pengguna, dan pemilihan file sumber secara otomatis ini adalah cara yang tepat untuk menerapkannya dengan rapi.

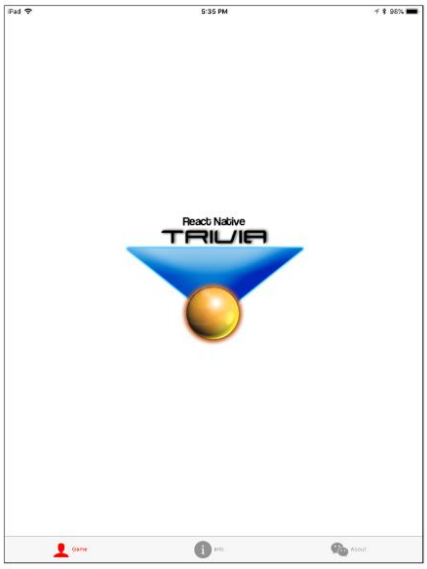
Jadi, Anda akan menemukan bahwa ada empat file di direktori komponen yang terkait dengan ini: MainLayout.android.js, MainLayout.ios.js, MainNav.android.js, dan MainNav.ios.js. Kita akan membahasnya nanti, tetapi sebagai pratinjau, file MainLayout* akan menjadi komponen tingkat atas aplikasi, dengan beberapa perbedaan antara setiap platform, dan file tersebut akan menggunakan file MainNav*, dengan pengemas memastikan versi yang benar dari setiap file digunakan, tanpa kita harus melakukan apa pun selain memberi nama file dengan tepat. Anda dapat menggunakan mekanisme pengalihan ini sebanyak atau sesedikit yang Anda perlukan, dan yang "diperlukan" hanyalah penggunaan nama file tertentu (dan perhatikan bahwa impor tidak menentukan platform; itu bagian penting lainnya).

Sekarang, setelah Anda memperkenalkan gagasan tentang komponen yang berbeda untuk platform yang berbeda, Anda mungkin ingin memisahkan komponen yang dibagikan di antara keduanya, dan itulah yang telah saya lakukan, dengan memperkenalkan direktori bersama. Baik React Native maupun Expo tidak memerlukan ini, tetapi masuk akal. Dalam kasus RNTrivia, semua kecuali file MainLayout* dan MainNav*, pada kenyataannya, dibagikan, jadi sisa kode aplikasi ada di sana. Kemudian, dalam direktori bersama ini, pada dasarnya saya memiliki dua hal: komponen layar dan komponen dialog modal, jadi saya telah membuat subdirektori modal di sana tempat kode untuk modal ditemukan, dan semua yang tidak ada di direktori itu adalah untuk layar tertentu, seperti yang ditunjukkan oleh nama file yang Anda lihat.

Direktori terakhir yang Anda lihat di root adalah status, dan di sinilah Anda akan menemukan beberapa kode, terbagi dalam empat file, yang terkait dengan status aplikasi. Namun, ini adalah percakapan yang lebih luas, yang akan kita bahas beberapa bagian dari sekarang. Jadi, mari kita beralih untuk melihat bagian pertama dari kode sumber, secara kasar, sekarang.

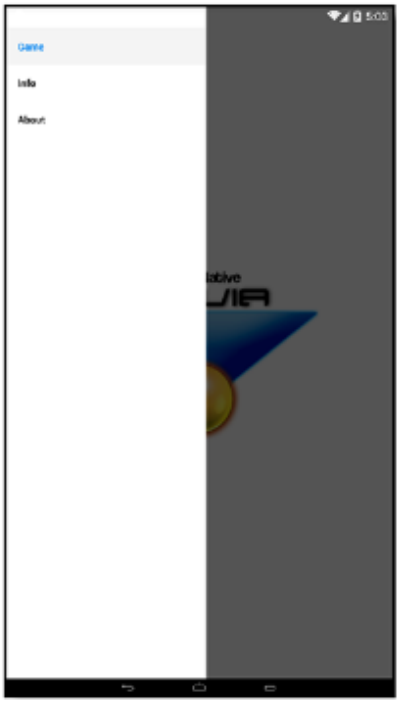
Navigasi Aplikasi

Pertimbangan lain, yang sangat disinggung di bagian sebelumnya, adalah bagaimana pengguna akan menavigasi aplikasi. Singkatnya, di iOS, ia akan menavigasi menggunakan antarmuka tab, seperti yang ditunjukkan pada Gambar 6-2.



Gambar 16.6 Navigasi Aplikasi Di Ios

Namun, di Android, kita akan menggunakan baki aplikasi untuk menavigasi, elemen UI yang hanya terlihat saat Anda menggeser dari sisi kiri layar, seperti yang dapat Anda lihat pada Gambar 16.7.



Gambar 16.7 Navigasi Aplikasi Di Android

Dalam kedua kasus tersebut, aplikasi terdiri dari tiga layar utama: layar tentang, layar info, dan layar permainan. Layar permainan selanjutnya dibagi menjadi tiga sublayar: layar beranda, layar papan peringkat, dan layar pertanyaan. Anda akan melihat masing-masing layar ini saat melihat kodenya, tetapi untuk memberi Anda gambaran umum:

- Layar tentang adalah layar tentang aplikasi yang umum, yang menyediakan informasi tentang aplikasi.
- Layar info menampilkan informasi tentang permainan saat ini yang diikuti pemain—hal-hal seperti skor, jumlah jawaban benar dan salah, dan sebagainya.
- Layar beranda permainan ➤ adalah layar yang pertama kali dilihat pengguna saat aplikasi dimulai, dengan asumsi tidak ada permainan yang sedang berlangsung (dalam hal ini, pengguna akan langsung melihat layar papan peringkat permainan sebagai gantinya). Layar beranda ini tidak lain hanyalah logo aplikasi.
- Layar papan peringkat permainan ➤ adalah layar yang dilihat pengguna saat permainan sedang berlangsung, dan pengguna sedang menunggu pertanyaan berikutnya. Layar ini menunjukkan kedudukan terkini untuk permainan yang sedang berlangsung.
- Layar pertanyaan permainan ➤ adalah layar tempat pertanyaan saat ini ditampilkan, dan pengguna mengirimkan jawabannya. Perhatikan bahwa ini adalah satu-satunya layar yang dapat digunakan pengguna untuk berinteraksi; yang lainnya hanya dapat dibaca.
- Selain layar dan sublayar ini, tiga modal dapat ditampilkan di aplikasi.
- NamePromptModal ditampilkan saat aplikasi dimulai dan digunakan untuk mendapatkan nama pengguna. Modal ini juga memiliki elemen untuk memungkinkan pengguna admin mengalihkan aplikasi ke mode admin.
- EndGameModal ditampilkan saat permainan berakhir dan digunakan untuk memberi tahu pemain hasil permainan (di posisi mana ia menyelesaikan permainan).
- AdminModal, tentu saja, adalah tempat kontrol admin berada dan hanya dapat dilihat oleh pengguna admin.

Pengguna dapat bergerak bebas di antara layar tentang, info, dan permainan (melalui tab di iOS atau tautan navigasi baki di Android), tetapi pergerakan di antara sublayar layar permainan dan tampilan modal sepenuhnya dikendalikan oleh logika dan peristiwa aplikasi.

16.6 MENGONFIGURASI APLIKASI

Kita tidak akan menghabiskan terlalu banyak waktu pada dua berkas konfigurasi, package.json, dan app.json, karena Anda sudah melihatnya, dan tidak banyak perbedaan di antara keduanya, dan sebagian besar sudah dibuat secara otomatis untuk kita, tetapi saya tidak ingin melewatkannya begitu saja, jadi kita akan melihatnya sebentar sekarang.

Package.Json

Seperti yang Anda ketahui, package.json menyediakan informasi untuk digunakan NPM, dan yang terpenting bagi kita sebagai pengembang adalah daftar dependensi.

```
{
  "name": "Intrivia", "version": "0.1.0", "private": true,
  "devDependencies": {
    "react-native-scripts": "1.14.0",
    "jest-expo": "~27.0.0",
    "react-test-renderer": "16.3.1"
  },
  "main": "./node_modules/react-native-scripts/build/bin/crna-entry.js",
  "scripts": {
    "start": "react-native-scripts start",
    "eject": "react-native-scripts eject",
    "android": "react-native-scripts android",
    "ios": "react-native-scripts ios",
    "test": "jest"
  },
  "jest": {
    "preset": "jest-expo"},
  "dependencies": {
   "expo": "^27.0.1",
  "react": "16.3.1",
  "react-native": "~0.55.2",
  "react-navigation": "^2.5.5",
  "react-redux": "^5.0.7",
  "redux": "^4.0.0",
  "socket.io-client": "^2.1.1",
  "native-base": "^2.6.1"
  }
}
```

Dalam aplikasi ini, beberapa dependensi di luar default yang ditambahkan secara otomatis saat create-react-native-app dijalankan harus ditambahkan. Kita harus menambahkan react-navigation, react-redux, redux, dan socket.io-client. Anda sudah tahu apa itu react-navigation. React-redux dan redux akan segera dibahas. socket.io-client tentu saja adalah socket.io, seperti yang dibahas di Bab 5, tetapi ini adalah versi klien dari pustaka tersebut, seperti yang Anda harapkan mengingat kita menulis aplikasi klien.

Catatan Versi React Native, Expo, dan NPM/Node yang telah Anda instal akan memengaruhi apa yang Anda lihat di sini saat menjalankan create-react-native-app. Mengingat sebagian besar ini adalah hal-hal standar yang biasanya tidak perlu Anda khawatirkan sebagai pengembang, dan juga karena apa yang Anda lihat saat membuat aplikasi bisa saja berbeda, saya belum membahas secara terperinci tentang setiap elemen di sini. Saya hanya membahas hal-hal yang secara khusus penting dalam konteks buku ini, dan Saya serahkan, sebagai latihan kepada Anda, untuk mengeksplorasi hal-hal lain, jika dan ketika hal itu menjadi relevan dalam pekerjaan pengembangan Anda sendiri.

App.Json

Konten file app.json juga sama seperti yang telah Anda lihat di Restaurant Chooser, hanya dengan beberapa perubahan nama, untuk sebagian besar.

```
{
  "expo": {
    "name": "RNTrivia","description": "React Native Trivia",
    "icon": "./images/appIcon.png",
    "splash": {"image": "./images/splash.png","resizeMode": "cover"},
    "version": "1.0.0", "slug": "rntrivia", "sdkVersion": "27.0.0",
    "ios": {"bundleIdentifier": "com.etherient.rntrivia"},
    "android": {"package": "com.etherient.rntrivia"
    }
  }
```

Perhatikan bahwa atribut ikon dan splash sekarang menggunakan jalur relatif untuk menunjuk ke gambar dalam direktori gambar, seperti yang dibahas di bagian sebelumnya.

Sebelum Kita Mulai, Catatan Tentang Impor

Agar bab ini tidak menjadi terlalu panjang, dalam upaya untuk menghemat ruang, saya telah menghapus impor dari setiap file sumber. Anda telah melihat banyak di antaranya, dan banyak yang hanya berupa file kode dari aplikasi itu sendiri. Namun, saya tidak ingin membiarkan Anda bertanya-tanya, jadi berikut adalah ikhtisar impor yang akan Anda temukan di setiap file yang akan dibahas dan apa saja kelas yang diimpor (dan, tentu saja, saya akan menjelaskan yang belum pernah Anda lihat sebelumnya, saat kita menemukannya):

- Diimpor dari react: React
- Diimpor dari react-native: Alert, Button, FlatList, Image, Modal, StyleSheet, Text, Vibration, View, WebView. Perhatikan bahwa Text diberi alias RNText dalam satu file (/components/shared/GameScreen- Question.js), untuk menghindari konflik nama, namun tetap merupakan komponen React Native Text. • Diimpor dari native-base: Body, Button, Card, CardItem, Input, Item, Label, Root, Switch, Text, Toast
- Diimpor dari react-navigation: createBottomTabNavigator, createDrawerNavigator, createSwitchNavigator
- Diimpor dari redux: combineReducers, createStore
- Diimpor dari react-redux: connect, provider
- Diimpor dari socket.io-client: io
- Diimpor dari /CoreCode.js: CoreCode
- Diimpor dari /state/store.js: store
- Diimpor dari /state/actions.js: answerButtonHighlight, resetAllButtons, setCurrentStatus, setEndGameMessage, setGameData, setIsAdmin, setPlayerID, setPlayerNamestate, setQuestion, showHideModal, updateAnswerButtonLabel, updateLeadboard, SOROTAN_TOMBOL_JAWAB, SETEL ULANG_SEMUA_TOMBOL, SET_STATUS_SAAT INI, SET_PESAN_AKHIR_PERMAINAN, SET_DATA_PERMAINAN, SET_IS_ADMIN, SET_ID_PEMAIN, SET_NAMA_PEMAIN, SET_PERTANYAAN, TUNJUKKAN_SEMBUNYIKAN_MODAL, PERBARUI_LABEL_TOMBOL_JAWABAN, PERBARUI_PAPAN_PERINGKAT
- Diimpor dari /state/initialState.js: initialState

- Diimpor dari /state/reducers.js: gameDataReducer, leaderboardReducer, modalsReducer, playerInfoReducer, questionReducer
- Diimpor dari /components/shared/modals/NamePromptModal.js: NamePromptModal
- Diimpor dari /components/shared/modals/EndGameModal.js: EndGameModal
- Diimpor dari /components/shared/modals/AdminModal.js: AdminModal
- Diimpor dari /components/shared/MainLayout.<android|ios>. js: MainLayout
- Diimpor dari /components/shared/MainNav.ios.js: Tab
- Diimpor dari /components/shared/MainNav.android.js: Laci
- Diimpor dari /components/shared/GameScreen.js: GameScreen
- Diimpor dari /components/shared/GameScreen-Home.js: GameHomeScreen
- Diimpor dari /components/shared/GameScreen-Leaderboard.js: GameLeaderboardScreen
- Diimpor dari /components/shared/GameScreen-Question.js: GameQuestionScreen
- Diimpor dari /components/shared/AboutScreen.js: AboutScreen
- Diimpor dari /components/shared/InfoScreen.js: InfoScreen

Titik Awal (Atau Kekurangannya?): App.js

Seperti halnya aplikasi React Native lainnya yang menggunakan Expo, titik awal kita adalah berkas App.js. Namun, bagi RNTrivia, apa yang Anda temukan dalam berkas itu, yah, tidak terlalu mengesankan.

```
constructor(inProps) {
  super(inProps);
}

render() {
  return (<Provider store={store}><Root><MainLayout /></Root></Provider>);
}

componentDidMount() {
  store.dispatch(showHideModal("namePrompt", true));
  };
}
```

Selain impor, komponen tingkat atas yang didefinisikan di sini untuk aplikasi hanya sedikit direnderer, karena sebagian besar pekerjaan didelegasikan ke komponen lain. Komponen Root, seperti yang Anda lihat di Restaurant Chooser, adalah komponen NativeBase yang harus kita bungkus di sekitar komponen lain, untuk merender pesan Toast, yang merupakan cara kita memberi tahu pengguna apakah dia menjawab pertanyaan dengan benar atau salah nanti. Menempatkannya di sini, di tingkat tertinggi, berarti kita dapat menggunakan pesan Toast tersebut dari mana saja di aplikasi dan tidak perlu mengingat untuk membungkus beberapa subkomponen dalam komponen Root.

Tapi apa masalahnya dengan komponen Provider itu? Itu adalah salah satu yang belum pernah Anda lihat sebelumnya. Nah, untuk menjelaskannya, saya harus berbicara tentang

sesuatu yang disebut Redux terlebih dahulu, dan itu ada di bagian berikutnya, jadi, untuk saat ini, mari kita lewati saja.Komponen MainLayout, seperti yang saya bicarakan belum lama ini, akan menjadi salah satu komponen yang didefinisikan dalam MainLayout.android.js atau MainLayout.ios.js, tergantung pada platformnya. Dengan risiko mencuri perhatian saya sendiri, Anda akan segera mengetahui bahwa satu-satunya perbedaan antara keduanya adalah file MainNav* mana yang digunakan.

Jika tidak, keduanya sama saja. Namun, karena keduanya dipisahkan berdasarkan platform seperti ini, mudah untuk memperluas aplikasi nanti, untuk menangani masalah khusus platform apa pun yang mungkin muncul. Itulah sebabnya hal itu dilakukan. Ya, itu, dan tentu saja saya dapat menunjukkan mekanisme peralihan platform ini kepada Anda. Dalam pengendali peristiwa componentDidMount(), NamePromptModal ditampilkan. Namun, cara menampilkannya adalah sesuatu yang baru, seperti halnya komponen Provider yang secara mencurigakan (tetapi sengaja) saya lewatkan untuk disebutkan dalam metode render(). Kedua hal ini terkait dengan sesuatu yang saya sebutkan sebelumnya yang akan saya bahas, yaitu, status aplikasi, dan itulah yang akan saya bahas sekarang.

Keadaan: Redux

Jika Anda pernah menghabiskan waktu online untuk meneliti React atau React Native, pasti Anda pernah menemukan sesuatu yang disebut Redux. Bahkan, Anda mungkin sering melihat "React+Redux", seolah-olah Anda harus menggunakan Redux dengan React dan React Native (apa pun itu). Sebelum memberi tahu Anda apa itu Redux, saya akan memberi tahu Anda bahwa itu sama sekali tidak diperlukan saat bekerja dengan React atau React Native (atau Expo). Namun, memang benar bahwa banyak pengembang merasa bahwa keduanya bekerja dengan sangat baik bersama-sama, bahkan sangat baik, sehingga Anda tidak selalu dapat menemukan tutorial dan artikel yang tidak menggunakan Redux! Namun, pada akhirnya, Redux hanyalah pustaka lain yang mungkin ingin atau tidak ingin Anda gunakan (dan Redux juga bukan satu-satunya opsi yang melakukan apa yang dilakukannya, tetapi mungkin yang paling populer dengan selisih yang cukup besar, setidaknya pada saat penulisan ini).

Bahkan, Anda bahkan dapat menggunakan Redux sendiri, baik dengan kerangka kerja lain sepenuhnya atau hanya sendiri, tetapi saya ngelantur. Apa sebenarnya Redux dan apa fungsinya? Nah, untuk menjawabnya, kita akan membahas konsep status aplikasi. Anda tentu telah melihat bahwa komponen React dapat memiliki status internalnya sendiri saat dibutuhkan, misalnya, untuk menyediakan data ke FlatList untuk ditampilkan. Tidak semua komponen memerlukan status, tetapi banyak yang memerlukannya. Namun, saat status tersebut terdapat di dalam komponen itu sendiri, terkadang Anda akan merasa bahwa menulis aplikasi menjadi rumit, karena, terkadang, komponen harus bekerja dengan status komponen lain, atau beberapa kode luar harus bekerja.

Anda melihat satu cara untuk melakukan pendekatan ini di Restaurant Chooser: simpan data yang mewakili status komponen di luar komponen tetapi "masukkan ke dalam komponen," menggunakan atribut data. Ingat daftar peserta, misalnya? Itu disimpan dalam array yang berada di luar salah satu komponen, tetapi kemudian beberapa komponen yang berbeda memanfaatkannya dengan mereferensikan array dalam status komponen (melalui

konstruktor yang menyalin referensi ke array di dalam komponen) dan kemudian atribut data itu, untuk menggunakannya dalam komponen (atau subkomponen). Ini adalah salah satu cara yang bisa dilakukan, dan cara ini cukup berhasil, jika aplikasinya kecil dan tidak terlalu rumit, karena mengelolanya tidaklah terlalu sulit.

Namun, saat aplikasi mulai berkembang dan menjadi lebih rumit, ceritanya mulai berubah. Pertimbangkan bahwa status tidak hanya berarti data yang saat ini ditampilkan oleh komponen Anda. Status juga dapat berarti hal-hal seperti respons server dan berbagai cache. Ada juga pertimbangan untuk memastikan integritas status aplikasi Anda setiap saat (sesuatu yang tidak dapat Anda lakukan dengan status yang hanya tersimpan dalam beberapa variabel JavaScript yang dapat diubah oleh kode di mana saja).

Saat aplikasi menjadi besar dan rumit, ada begitu banyak jalur melalui kode yang dapat mengubah status sehingga menjadi sulit untuk memahami apa yang terjadi, dan Anda dapat menemukan status Anda berubah dalam situasi yang bahkan tidak jelas bagaimana atau mengapa status itu berubah. Tidak bagus untuk debugging! Namun, dengan semua yang dikatakan, Anda mungkin ingin status Anda dapat diakses dari hampir di mana saja dalam kode Anda tetapi dengan cara yang aman. Untuk mencapai semua itu, Anda memerlukan mekanisme yang lebih tangguh daripada sekadar variabel JavaScript biasa (meskipun, hei, akan lebih baik jika memiliki kesederhanaan semacam itu juga), dan di situlah Redux berperan.
Redux dibangun berdasarkan tiga prinsip dasar.

- Ada "sumber kebenaran tunggal" yang kanonik untuk status aplikasi. Ini berarti bahwa ada satu penyimpanan data yang menangani semua hal di aplikasi Anda, semua komponen, dan semua kode. Data tersebut tidak tersebar di semua tempat, dan tidak tersimpan dalam variabel JavaScript "tidak terkelola" yang dapat diubah oleh kode apa pun kapan saja. Ini memiliki banyak manfaat, beberapa di antaranya mungkin tidak langsung terlihat. Misalnya, menjadi mudah untuk menyimpan seluruh status aplikasi Anda saat ini dan "menghidrasi ulang"-nya dengan status yang sama nanti. (Ini hanya menjadi penyimpanan status saat ini, mungkin ke Penyimpanan Lokal, lalu memuatnya kembali dan meneruskannya sebagai status awal saat penyimpanan dibuat, sesuatu yang akan saya bicarakan sebentar lagi.)
- Status bersifat hanya-baca. Prinsip ini sedikit keliru, karena, jelas, status tidak akan berfungsi jika Anda benar-benar tidak dapat mengubahnya. Bayangkan komponen TextInput. Jika saya ingin statusnya dikelola oleh Redux tetapi tidak dapat memperbarui status dengan cara tertentu, maka Redux tidak berguna bagi saya. Tidak, yang dimaksud prinsip ini adalah bahwa hanya ada satu cara untuk mengubah status, dan itu sangat terkontrol. Seperti yang akan Anda lihat, dengan Redux, itu berarti mengirimkan sesuatu yang disebut tindakan.
- Perubahan dilakukan di penyimpanan menggunakan fungsi murni. Ini berarti Anda tidak akan pernah dapat menyentuh data status secara langsung. Sebaliknya, suatu fungsi (disebut pereduksi di Redux) melihat status sebelumnya, melakukan beberapa tindakan, dan mengembalikan status baru.

Baiklah, itu saja teori di balik Redux, tetapi seperti apa praktiknya? Nah, semuanya dimulai

dengan membuat penyimpanan.

```
const store = createStore(function(){}, {});
```

Dua argumen untuk fungsi createStore() adalah fungsi pereduksi dan status awal, dalam urutan tersebut. Status awal adalah objek JavaScript biasa yang menyediakan data untuk disimpan pada awalnya. Objek kosong untuk status awal tidak banyak berguna, jadi mari kita berikan sesuatu yang lebih menarik.

```
const initialState = {
  kid: {firstName: "Bart", lastName: "Simpson"}
};
const store = createStore(function(){}, initialState);
```

Jika Anda menjalankan kode tersebut dan kemudian menjalankan console.log(store), Anda akan melihat bahwa penyimpanan telah dibuat, tetapi Anda tidak akan menemukan data di dalamnya. Untuk mendapatkan data, gunakan metode getState().

```
console.log(store.getstate());
```

Namun, meskipun Anda melakukannya, Anda tetap tidak akan melihat datanya; Anda akan melihat undefined. Apa yang terjadi? Untuk memperbaikinya, fungsi reducer yang saat ini kosong harus mengembalikan sesuatu. Itu karena reducer akan dipanggil oleh Redux saat penyimpanan dibuat, dan karena tidak mengembalikan apa pun saat ini, Redux memperlakukan hal-hal seolah-olah tidak ada status awal, meskipun kita meneruskannya ke createStore(). Untungnya, memperbaikinya mudah.

```
const store = createStore(function(inState) {return instate;}, initialState);
```

Jika Anda mencoba kode tersebut sekarang, Anda akan melihat data dalam initialState digemakan ke konsol. Sejauh ini, baik-baik saja! Kita punya penyimpanan, dan kita punya data di dalamnya. Kita belum bisa melakukan apa pun dengan penyimpanan ini, tetapi penyimpanan itu ada. Sekarang, mari kita bahas fungsi pereduksi itu. Tugas fungsi itu pada akhirnya adalah membuat baris kode ini melakukan sesuatu.

```
store.dispatch({ type: "update", payload: {firstName: "Lisa" } });
```

Metode dispatch() adalah satu-satunya cara untuk mengubah data di penyimpanan. Metode ini mengambil satu argumen: objek yang disebut tindakan. Objek ini memiliki atribut tipe yang memberi tahu reducer jenis tindakan apa yang harus dilakukan dan atribut payload yang merupakan objek dengan data apa pun yang harus diperbarui. Biasanya, Anda akan melihat fungsi tindakan yang digunakan, yang merupakan fungsi yang membuat objek tindakan. Ini menyediakan tingkat abstraksi dan cara untuk mengatur kode Anda dengan lebih baik dan

membuat penggunaan tindakan lebih kuat. Jadi, di sini kita akan menulis sesuatu seperti

```
const updateAction = function(inFirstName) {
  return {type: "update", payload: {firstName: inFirstName}};
};
```

Dan kemudian, panggilan dispatch() akan berubah untuk menggunakannya.

```
store.dispatch(updateAction("Lisa"));
```

Bagus sekali. Sekarang kita memberi tahu toko bahwa kita ingin melakukan tindakan pembaruan. Namun, tidak akan terjadi apa-apa jika kita melakukannya sekarang, karena kita juga harus menyediakan fungsi peredam yang benar-benar melakukan sesuatu.

```
const reducer = function(inState, inAction) {
  if (inAction.type === "update") {
    return inState.kid.firstName = inAction.payload.firstName;
  }
}
```

Argumen inState akan menjadi data status terkini, dan argumen inAction adalah objek tindakan yang diteruskan ke metode dispatch() (seperti yang dibuat oleh fungsi updateAction()). Sekarang, apa yang Anda lakukan di dalam fungsi tersebut tidak memiliki aturan nyata selain keluaran fungsi tersebut harus berupa status baru. Cara Anda mencapai status baru tersebut terserah Anda. Namun, ada masalah signifikan dalam kode ini, yaitu hal penting yang perlu diingat tentang reducer adalah Anda tidak pernah menyentuh status yang ada di dalamnya. Status terkini (di inState) tidak pernah dimodifikasi secara langsung seperti yang dilakukan di sini! Ingat: Reducer seharusnya menjadi fungsi "murni", yang berarti mereka tidak mengubah argumennya atau memiliki efek samping apa pun. Sebaliknya, Anda harus mengkloning inState dengan cara tertentu, mungkin seperti ini:

```
const reducer = function(inState, inAction) {
  const state = Object.assign({}, inState);
  state.kid.firstName = inAction.payload.firstName;
  return state;
}
```

Itu akan berhasil. Namun, cara yang lebih modern untuk melakukannya adalah dengan operator spread, dengan mengganti panggilan Object.assign() dengan ini:

```
cont state= {...instate};
```

Itu "menyebarkan" semua atribut dari inState ke objek baru yang direferensikan oleh status. Atau, Anda cukup membuat objek baru, jika payload berisi semua data yang Anda butuhkan. Itu juga bagus dan terkadang itulah yang ingin Anda lakukan. Apa pun itu, jangan pernah

sentuh status yang ada; itulah yang perlu diingat. Penting juga untuk dicatat bahwa reducer dapat (dan biasanya memang demikian, seperti yang akan Anda lihat nanti) menangani lebih dari satu jenis tindakan. Meskipun kita hanya mendefinisikan satu jenis tindakan pembaruan di sini, kita tetap memerlukan pernyataan if itu, karena ketika Redux membuat panggilan awal ke reducer, ia akan meneruskan jenis tindakan internal yang tidak akan diketahui cara menanganinya oleh kode kita.

Dalam hal itu, mengembalikan status yang ada adalah apa yang ingin kita lakukan (yang akan ditetapkan Redux menggunakan initialState), jadi kita tetap memerlukan logika di sana, karena tanpanya, kode akan menimpa data status awal itu, mencoba memperbaruinya dengan payload kosong, dan itu akan menjadi hal yang Sangat Buruk™. Singkatnya, itulah Redux. Penyimpanan, tindakan, dan pereduksi, satu sumber kebenaran yang hanya dapat dimodifikasi dengan metode dispatch(). Itulah intinya.

Masih banyak lagi yang perlu dilakukan saat tiba saatnya untuk menghubungkan React Native ke penyimpanan Redux, tetapi itu akan dibahas nanti. Untuk saat ini, mari kita lihat kode aktual di RNTrivia yang membahas Redux, yang, seperti yang akan Anda lihat, tidak lebih dari sekadar membangun konsep yang baru saja dijelaskan dan, sungguh, bahkan tidak terlalu membangun konsep tersebut. Catatan Bundel kode sumber berisi contoh ini dalam file HTML, jadi Anda tinggal memuatnya di browser pilihan Anda (yaitu browser yang relatif modern), dan Anda akan melihatnya beraksi.

initialState.js

Jadi, setelah pembahasan tentang apa itu Redux selesai, sekarang kita dapat melihat bagaimana Redux digunakan di RNTrivia. Pertimbangan pertama adalah seperti apa status awal aplikasi, dan file initialState.js menyediakannya.

```
export default initialState = {

      leaderboard: {listData: []},
      gameData: {
        asked:  "?????",  answered:  "?????",  points:  "?????",  right:
      "?????",wrong: "?????",
        totalTime:  "?????",fastest:  "?????",  slowest:  "?????",  average:
      "?????",
   },
question: {
      answerButtonPrimary: [true, true, true, true, true],
      answerButtonDanger: [false, false, false, false, false],
      answerButtonLabels: [null, null, null, null, null, null],
      currentQuestion: null, selectedAnswer: -1,
},
modals: {
  namePromptVisible: false, endGameVisible: false, adminVisible: false,
  endGameMessage: null,isAdmin: false, currentStatus: ""
},
  playerInfo: {id: null, name: null}
};
```

Seperti yang Anda lihat, ini hanyalah objek JavaScript. Data ini terdiri dari lima "cabang" data:

- Leaderboard: Ini berisi data yang ditampilkan FlatList pada layar leaderboard (listData).
- gameData: Ini berisi informasi tentang permainan saat ini untuk pemain ini, termasuk hal-hal seperti poinnya, berapa banyak pertanyaan yang dijawabnya dengan benar dan salah, dan statistik tentang kecepatannya dalam menjawab.
- Question: Ini berisi data yang digunakan pada layar permainan ➤ pertanyaan. Ini mencakup dua array (answerButtonPrimary dan answerButtonDanger) yang, Anda akan lihat nanti, digunakan untuk melacak dan memanipulasi status tombol (apakah salah satu dipilih atau tidak), label untuk tombol (answerButtonLabels), serta currentQuestion, dan tombol mana yang telah dipilih pengguna (selectedAnswer).
- modals: Ini berisi tanda yang digunakan untuk menentukan modal apa yang ditampilkan. Aha! Ingat baris di componentDidMount() di App.js? Jangan khawatir, kita akan membahasnya nanti, tetapi ini jelas terkait dengan ini.
- playerInfo: Ini menyimpan informasi tentang pemain, yaitu id dan namanya.

Masing-masing cabang ini akan dikontrol oleh reducer. Reducer tidak selalu harus menangani seluruh pohon status seperti contoh sederhana sebelumnya, dan, pada kenyataannya, reducer hampir tidak pernah menangani seluruh pohon status seperti itu. Sebaliknya, Anda akan menulis beberapa reducer, satu untuk setiap potongan logis data status. Setiap reducer akan memiliki satu atau beberapa tindakan terkait. Anda dapat mulai melihat semua itu bersatu di bagian berikutnya saat objek initialState ini digunakan.

Kiat Jika Anda bekerja dengan Redux di luar React, Anda akan menemukan bahwa atribut objek ini tidak perlu berupa objek, seperti di sini. Misalnya, jika Anda ingin playerInfo menjadi Boolean (yang tidak masuk akal, tetapi saya bisa melakukannya), itu tidak masalah. Namun, saat Anda mencoba melakukannya di React Native, Anda akan mendapatkan kesalahan yang mengatakan bahwa semua atribut status harus berupa objek. Itu tidak mengubah apa pun di sini, karena struktur ini pada dasarnya memang ideal, tetapi itu adalah sesuatu yang saya temukan dan ingin saya sampaikan kepada Anda.

Store.Js

Setelah Anda mendefinisikan objek status awal, langkah berikutnya adalah membuat toko itu sendiri, yang dilakukan di file store.js. Bagian dari pembuatan itu adalah memberi tahu reduktor apa yang akan digunakan. Namun, lazim untuk memiliki lebih dari satu reduktor dalam satu aplikasi, seperti yang disebutkan. Jadi, mengingat metode createStore() hanya menerima satu fungsi reduktor, bagaimana kita mengatasinya? Nah, Redux menyediakan fungsi combineReducers() yang sangat berguna yang dapat Anda gunakan seperti ini:

```
const rootReducer = combineReducers({
  leaderboard: leaderboardReducer,
  question: questionReducer,
  modals: modalsReducer,
  playerInfo: playerInfoReducer,
  gameData: gameDataReducer
});
```

Sekarang Anda memiliki satu rootReducer, tetapi di dalamnya terdapat lima reducer yang berbeda, masing-masing menangani cabang objek status yang berbeda, berdasarkan kunci di sini. Misalnya, leaderboardReducer bekerja dengan cabang leaderboard, dan seterusnya. Ada sedikit sihir hitam di sana, tetapi Redux menangani pemetaan itu di balik layar. Kita hanya perlu memastikan bahwa kuncinya cocok dengan apa yang ada di objek status. Perhatikan juga bahwa Anda akan mendapatkan kesalahan, jika ada sesuatu di status yang tidak memiliki reducer yang ditentukan, jadi pastikan Anda memilikinya untuk setiap cabang di pohon status. RootReducer ini kemudian dapat digunakan untuk membuat penyimpanan.

```
export default createStore(rootReducer, initialState);
```

Sekarang kita punya penyimpanan data Redux yang berisi status awal aplikasi kita. Setelah itu, saatnya membuat beberapa tindakan dan reducer. Kita akan mulai dengan tindakan.

Actions.Js

File actions.js adalah tempat kita menemukan semua tindakan yang diperlukan di seluruh RNTrivia. Sering kali, pengembang akan memiliki file sumber terpisah untuk setiap tindakan (serta setiap reducer, yang akan saya bahas di bagian berikutnya), tetapi tidak ada aturan yang pasti tentang itu. Gunakan organisasi apa pun yang paling masuk akal bagi Anda. Bagi saya, satu file saja sudah masuk akal, jadi itulah yang saya pilih. Potongan kode pertama dalam file ini mendefinisikan beberapa pseudo-konstanta yang mendefinisikan jenis tindakan.

```
exports.ANSWER_BUTTON_HIGHLIGHT = "abh";
exports.RESET_ALL_BUTTONS = "rab";
exports.SET_CURRENT_STATUS = "scs";
exports.SET_END_GAME_MESSAGE = "segm";
exports.SET_GAME_DATA = "sgd";
exports.SET_IS_ADMIN = "sia";
exports.SET_PLAYER_ID = "spi";
exports.SET_PLAYER_NAME = "spn";
exports.SET_QUESTION = "scq";
exports.SHOW_HIDE_MODAL = "shm";
exports.UPDATE_ANSWER_BUTTON_LABEL = "uabl";
exports.UPDATE_LEADERBOARD = "ul";
```

Nilai untuk bidang ini sepenuhnya acak; nilai tersebut hanya perlu unik. Saya hanya mengambil huruf pertama dari setiap jenis tindakan sebagai nilai. Saya berani bertaruh bahwa masing-masing hal ini dapat dijelaskan sendiri, tetapi saat kita melihat tindakan tersebut, Anda akan melihat apa arti masing-masing. Dan, untuk membantu Anda memahami tindakan tersebut, saya telah mengelompokkannya ke dalam kelompok logis, yang disusun berdasarkan tindakan pada tingkat tinggi.

Tindakan untuk Modal

Pertama, kita memiliki tindakan yang terkait dengan bekerja dengan modal. Dalam

kasus ini, sebenarnya hanya ada satu tindakan.

```
exports.showHideModal = (inModalName, inVisible) => {
  return {type: exports.SHOW_HIDE_MODAL,
    payload: {modalName: inModalName, visible: inVisible }
  };
};
```

Tindakan ini digunakan untuk menyembunyikan atau menampilkan salah satu modal. Seperti yang Anda lihat, nama modal dilewatkan, dan sebuah tanda yang menyatakan apakah modal tersebut terlihat atau tidak dan ini menjadi muatan, dengan tipe SHOW_HIDE_MODAL, salah satu pseudo-konstanta dari sebelumnya. Ini adalah bentuk umum yang akan diambil semua fungsi tindakan: menerima beberapa argumen (atau tidak sama sekali, seperti halnya untuk satu tindakan) dan mengembalikan objek yang terdiri dari atribut tipe, yang nilainya adalah salah satu pseudo-konstanta, dan atribut muatan (yang mungkin berupa objek kosong). Tidak ada yang mengatakan Anda tidak dapat menyertakan beberapa logika tambahan jika diperlukan, dan tidak ada yang mengatakan Anda tidak dapat membuat satu fungsi tindakan dan menggunakan percabangan di dalamnya untuk mengembalikan objek yang sesuai.

Namun, intinya adalah bahwa fungsi ini harus mengembalikan objek dengan kedua atribut tersebut; itulah yang dibutuhkan Redux (sebagai bagian dari panggilan dispatch()). Faktanya, mengingat kesamaan bentuk ini, saya tidak akan menunjukkan kode untuk tindakan lainnya; Saya akan menjelaskan fungsinya saja. (Saya akan menunjukkan nama fungsi, argumen yang dibutuhkan, tipe yang terkait dengannya dan, tentu saja, menjelaskan kegunaannya, meskipun saya rasa semuanya sudah sangat jelas.) Saya rasa ini akan menghilangkan beberapa redundansi dari percakapan ini. Jika Anda kembali ke metode componentDidMount() di komponen tingkat atas dari App.js, Anda akan menemukan satu baris di sana:

```
store.dispatch(showHideModal("namePrompt", true));
```

Itu seharusnya masuk akal sekarang. Metode dispatch() adalah cara kita selalu mengubah status, dan sekarang Anda dapat melihat bahwa objek yang diteruskan ke metode tersebut adalah hasil dari pemanggilan fungsi tindakan showHideModal(). Itu adalah objek yang dikembalikan oleh fungsi tersebut. Ini belum menjelaskan secara pasti bagaimana NamePromptModal ditampilkan karena pemanggilan ini, tetapi sekarang Anda tahu bahwa itu, entah bagaimana, dikendalikan oleh status yang dikelola oleh Redux, dan untuk saat ini, itu sudah cukup. Kita akan membahas bagian lain dari persamaan tersebut segera. Sampai saat itu, mari beralih ke fungsi tindakan lainnya.

Tindakan untuk Info Pemain

Ada dua fungsi tindakan untuk mengubah data di cabang playerInfo dari status.

- Setplayerid(inid): SET_PLAYER_ID: Digunakan untuk mengubah bidang id saja
- Setplayername(inname): SET_PLAYER_NAME: Digunakan untuk mengubah bidang
- Name saja

Perhatikan bahwa, dalam kasus ini, saya memiliki dua tindakan untuk mengubah dua bagian informasi di cabang data ini secara individual. Anda akan melihat kasus lain di mana seluruh cabang diubah sekaligus. Itu sepenuhnya bergantung pada kasus penggunaan Anda dan pendekatan mana yang Anda gunakan, tidak ada yang benar atau salah, atau lebih baik atau lebih buruk. Saya tidak melakukannya dengan cara ini untuk alasan tertentu selain membuat kode sedikit lebih mudah, tetapi terutama, itu hanya untuk menunjukkan bahwa Anda dapat melakukannya dengan cara ini vs. pendekatan sekaligus, yang dapat Anda lihat di fungsi tindakan berikutnya.

Tindakan untuk Data Game

Hanya ada satu fungsi tindakan tunggal untuk mengubah data di cabang gameData dari status. GameData(inGameData): SET_GAME_DATA: Digunakan untuk mengubah seluruh bidang gameData

Tindakan untuk Data Pertanyaan

Kami memiliki empat fungsi tindakan yang menangani manipulasi data di cabang pertanyaan:

- Answerbuttonhighlight(inbuttonnumber): ANSWER_BUTTON_ HIGHLIGHT: Digunakan untuk menyorot tombol saat pengguna mengetuknya (memutasi bidang answerbuttonprimary, answerbuttondanger, dan selectedanswer)
- Updateanswerbuttonlabel(inbuttonnumber, inlabel): UPDATE_ ANSWER_BUTTON_LABEL: Mengubah label tombol yang ditentukan (berdasarkan angka, 0–5), yang digunakan saat menampilkan kemungkinan jawaban kepada pengguna (mengubah kolom answerbuttonlabels)
- Resetallbuttons(): RESET_ALL_BUTTONS: Digunakan untuk mengatur ulang semua tombol ke status default, tidak dipilih, yang dilakukan saat pertanyaan pertama kali ditampilkan (mengubah kolom answerbuttonprimary dan answerbuttondanger)
- Setquestion(inquestion): SET_QUESTION: Digunakan untuk mengubah kolom currentquestion

Tindakan untuk Data Papan Peringkat

Ada dua fungsi untuk menangani data di cabang papan peringkat.

- SetEndGameMessage(inMessage): SET_END_GAME_MESSAGE: Digunakan untuk menunjukkan pesan kepada pemain saat permainan berakhir (mengubah bidang endGameMessage dari cabang modal. Tunggu, apa? Jangan khawatir, saya akan menjelaskannya.
- UpdateLeaderboard(inListData): UPDATE_LEADERBOARD: Digunakan untuk memperbarui data yang ditampilkan komponen FlatList pada layar papan peringkat permainan ➤ (mengubah bidang listData) Sekarang, mari kita bahas keanehan dalam setEndGameMessage(), yaitu, mengapa ia menyentuh apa pun di luar cabang papan peringkat data status. Memang benar bahwa, secara umum, fungsi tindakan (dan pereduksi, nanti) "terikat," dalam arti tertentu, ke cabang status tertentu. Namun, tidak ada aturan yang mengatakan bahwa mereka tidak dapat mengubah data lainnya. Ini murni penggambaran logis. Dalam kasus ini, pesan yang akan ditampilkan kepada

pengguna ditampilkan di EndGameModal, jadi fungsi ini secara logis, mungkin, seharusnya ada di grup modals. Namun, di sisi lain, EndGameModal hanya ditampilkan saat di layar papan peringkat permainan ➤ di akhir permainan, jadi masuk akal juga untuk berada di grup papan peringkat.

Bagaimana Anda memutuskan? Seperti banyak hal lain di dunia React Native, jawabannya adalah apa pun yang masuk akal bagi Anda. Namun, saya akan menyebutkan bahwa ada praktik terbaik yang diterima secara umum untuk mencoba dan tidak mengikat status Anda ke UI Anda. Dengan kata lain, cobalah untuk tidak memiliki tindakan dan pereduksi yang ditujukan untuk layar tertentu. Sebaliknya, rancang status Anda sedemikian rupa sehingga diabstraksikan dari UI itu sendiri. Meskipun demikian, itu tidak selalu praktis dan tidak selalu merupakan desain terbaik, dan terkadang, jika status dan UI tidak terlalu rumit, itu tidak membuat banyak perbedaan. Saya pikir ini adalah salah satu kasus di mana tidak ada jawaban yang benar atau salah atau bahkan "terbaik". Tetapi setidaknya sekarang Anda tahu apa yang menginformasikan pemikiran saya.

Tindakan untuk Data Admin

Kelompok fungsi tindakan terakhir khusus untuk layar admin (yang, tentu saja, bertentangan dengan praktik terbaik yang saya sebutkan di bagian sebelumnya!) dan mereka benar-benar mengubah data di cabang modal (untuk alasan dasar yang sama, seperti yang saya jelaskan di bagian sebelumnya, yaitu, ini hanya digunakan dari satu modal tertentu).

- Setisadmin(Inisadmin): SET_IS_ADMIN: Digunakan Saat Pengguna Membalik Tombol Pada Namepromptmodal (Mengubah Kolom Isadmin)
- Setcurrentstatus(Incurrentstatus): SET_CURRENT_STATUS: Digunakan Untuk Menampilkan Pesan Dari Server Ke Admin Yang Digunakan (Mengubah Kolom Currentstatus)

Reducers.Js

Reducers adalah pekerja keras Redux, karena mereka bereaksi terhadap objek yang dibuat oleh fungsi tindakan dan mengubah status. Nah, lebih tepatnya, mereka mengembalikan status baru; mereka tidak mengubah status yang ada. Nah, untuk lebih tepatnya, mereka mengembalikan bagian status baru, yang kemudian digabungkan Redux ke objek status yang sudah ada. Bagaimana pun cara Anda melihatnya, intinya adalah mereka bertanggung jawab untuk menyelesaikan pekerjaan, setidaknya sejauh menyangkut kode Anda sendiri. Untuk RNTrivia, saya telah menggunakan organisasi keseluruhan yang sama untuk reducer seperti yang saya lakukan untuk fungsi tindakan, yaitu, mereka dikelompokkan menurut cabang status yang mereka tangani dan semuanya berada dalam file reducers.js yang sama.

Reducer untuk Modals

Reducer pertama adalah untuk cabang modals status.

```
exports.modalsReducer = function(inState = {}, inAction) {
  switch (inAction.type) {
```

```
case SET_CURRENT_STATUS: {
  const modalsNode = {...inState};
  modalsNode.currentStatus = inAction.payload.currentStatus;
  return {...inState, ...modalsNode};
}

case SET_END_GAME_MESSAGE: {
  const modalsNode = {...inState};
  modalsNode.endGameMessage = inAction.payload.message;
  return {...inState, ...modalsNode};
}

case SET_IS_ADMIN: {
  const modalsNode = {...inState};
  modalsNode.isAdmin = inAction.payload.isAdmin;
  return {...inState, ...modalsNode};
}

case SHOW_HIDE_MODAL: {
  const modalsNode = {...inState };
  modalsNode[`${inAction.payload.modalName}Visible`]=
inAction.payload.visible;
  return {...inState, ...modalsNode};
}

    default: {return inState;}
   }
};
```

Tidak ada cara yang benar-benar standar untuk menulis reducer dan tidak ada aturan pasti tentang cara memecahnya, tetapi apa yang Anda lihat di sini adalah hal yang umum. Setiap reducer (yang biasanya diharapkan untuk menangani lebih dari satu jenis tindakan, dengan asumsi Anda memerlukan lebih dari satu jenis tindakan untuk melakukan pekerjaan aplikasi) adalah fungsi dengan pernyataan switch, dengan setiap case dari switch tersebut menjadi salah satu jenis yang ditentukan dalam actions.js.

Setiap case melakukan hal yang hampir sama: ia mendapatkan objek saat ini untuk cabang yang ditentukan (atau node, istilah mana pun yang Anda sukai) melalui kloning, menggunakan operator spread, lalu menetapkan nilai apa pun di dalamnya yang harus diubah. Dengan cara itu, kami mempertahankan nilai apa pun yang sudah ada di dalamnya dan hanya mengubah apa yang perlu kami ubah. Terkadang, kami dapat mengganti seluruh cabang, terkadang, hanya atribut tertentu, seperti yang dilakukan di sini. Anda dapat melakukan keduanya. Untuk jenis SET_CURRENT_ STATUS, misalnya, hanya atribut currentStatus yang diperbarui pada node modals. Anda akan melihat dua reducer nanti yang menggantikan seluruh cabang, tetapi semuanya tergantung pada apa yang masuk akal dalam kasus penggunaan spesifik Anda.

Catatan Ketika Anda memiliki satu reducer, seperti dalam contoh sederhana sebelumnya, reducer tersebut akan menerima seluruh pohon status, tetapi ketika Anda

menggabungkan beberapa reducer seperti ini, setiap reducer hanya akan menerima bagian dari pohon status yang menjadi tanggung jawabnya, berdasarkan kunci dalam objek yang diteruskan ke combineReducers().Apa pun itu, yang dikembalikan dari reducer adalah status baru, khususnya, cabang yang ditangani reducer.

Dalam semua kasus di sini, operator spread digunakan untuk mengkloning objek inState yang diteruskan (yang, ingat, hanyalah bagian dari status yang ditangani reducer ini, bukan seluruh objek status) dengan objek status yang dibuat oleh kode di setiap cabang. Perhatikan bahwa untuk menghindari kesalahan, kita memerlukan nilai default untuk inState, dan objek kosong sudah cukup. Anda dapat, daripada meneruskan status awal ke panggilan createStore(), membuat pohon status Anda sepotong-sepotong, dengan menentukan nilai default dengan cara ini. Serupa dengan itu, kasus default mencakup situasi apa pun yang mungkin tidak tercakup oleh blok switch reducer.

Seperti halnya dengan fungsi tindakan, saya tidak akan mencantumkan setiap reducer di sini, karena setelah melihat yang untuk modal, pada dasarnya Anda telah melihat semuanya. Saya hanya akan mencantumkan apa itu reducer, jenis apa yang ditanganinya, dan apa pun yang dapat melakukan sesuatu yang berbeda dari yang untuk modal.

Reducer untuk Info Pemain

Untuk cabang playerInfo, kita memiliki fungsi playerInfoReducer() yang sesuai. Fungsi ini menangani dua jenis: SET_PLAYER_ID dan SET_PLAYER_NAME. Kodenya hampir identik dengan modalsReducer(), jadi tidak ada yang perlu dilihat di sini.

Reducer untuk Data Game

Reducer untuk cabang gameData adalah sesuatu yang sedikit berbeda. Lihatlah

```
exports.gameDataReducer = function(inState = {}, inAction) {
  switch (inAction.type) {

    case SET_GAME_DATA: {
      return {...inState, ...inAction.payload.gameData };
    }
    default: {return inState;
    }
};
```

Ya, ini adalah salah satu kasus di mana seluruh cabang sedang dimutasi. Objek inAction.payload.gameData berisi semua data yang dibawa dalam cabang ini, jadi meskipun tentu saja memungkinkan untuk mengkloning cabang, lalu menyalin setiap atribut individual ke cabang tersebut, akan lebih ringkas untuk melakukan penggabungan sederhana dengan operator penyebaran seperti ini dan melanjutkan pekerjaan kita.

Pereduksi untuk Data Pertanyaan

Untuk bekerja dengan cabang pertanyaan status, kita harus melakukan beberapa hal yang sedikit berbeda. Dalam semua kasus, kita tetap mengkloning cabang yang ada, membuat perubahan, lalu menggabungkannya kembali, tetapi perubahan yang dibuat sedikit lebih rumit, setidaknya dalam beberapa kasus, daripada pereduksi lainnya, seperti yang dapat Anda

lihat sendiri.

```
exports.questionReducer = function(inState = {}, inAction) {
  switch (inAction.type) {
    case ANSWER_BUTTON_HIGHLIGHT: {
      const questionNode = { ...inState };
      questionNode.answerButtonPrimary = [true, true, true, true, true];
       questionNode.answerButtonDanger = [false, false, false, false,
      false];
];

      questionNode.selectedAnswer = inAction.payload.buttonNumber;
      if (inAction.payload.buttonNumber !== -1) {
      questionNode.answerButtonDanger[inAction.payload.buttonNumber] = true;
      }

      return { ...inState, ...questionNode };
    }
      case UPDATE_ANSWER_BUTTON_LABEL: {
        const questionNode = {...inState};
        questionNode.answerButtonLabels[inAction.payload.buttonNumber] =
      inAction.payload.label;
  return {...inState, ...questionNode};
}

      case RESET_ALL_BUTTONS : {
        const questionNode = {...inState };
        questionNode.answerButtonPrimary = [true, true, true, true, true];
        questionNode.answerButtonDanger = [false, false, false, false, false];
        return {...inState, ...questionNode };
      }
      case SET_QUESTION: {
        const questionNode = {...inState};
        questionNode.currentQuestion= inAction.payload.question;
        return {...inState, ...questionNode};
   }

default: {
  return inState;}

  }
};
```

ANSWER_BUTTON_HIGHLIGHT adalah perbedaan pertama. Pertama, array answerButtonPrimary dan answerButtonDanger disetel ulang ke nilai awal. Seperti yang akan Anda lihat nanti, array ini digunakan untuk menentukan gaya (warna, terutama) tombol, dan array ini digunakan untuk menyorot tombol saat pengguna mengetuk untuk memilihnya. Saat tombol diketuk, jenis tindakan ini dijalankan, dengan meneruskan nomor tombol yang diketuk. Jadi, setelah menyetel ulang array tersebut, dan menyimpan tombol mana yang diketuk dalam

atribut selectedAnswer, kita kemudian harus menyorot tombol yang ditentukan.

Jika -1 dimasukkan, kita tidak melakukan ini, tetapi, jika tidak, item dalam array answerButtonDanger yang terkait dengan tombol harus dibalik menjadi true. Setelah itu, sekali lagi, penggabungan langsung dilakukan untuk menyelesaikan penanganan jenis tersebut. Untuk tipe UPDATE_ANSWER_BUTTON_LABEL, kita hanya perlu memperbarui label untuk tombol yang ditentukan, jadi ini tidak lain hanyalah pembaruan ke dalam array answerButtonLabels, karena nilai inAction.payload.buttonNumber akan menjadi indeks ke dalam array untuk tombol tersebut.

Tipe RESET_ALL_BUTTONS melakukan pengaturan ulang yang sama seperti yang dilakukan ANSWER_BUTTON_HIGHLIGHT pada dua array penyorotan terkait tombol, tetapi hanya itu saja. Tidak perlu menangani jawaban yang dipilih. Tipe ini akan dikirimkan saat pertanyaan ditampilkan untuk memastikan semua tombol berada dalam status awal yang benar. Terakhir, tipe SET_QUESTION dikirimkan saat pertanyaan ditampilkan untuk mencatat pertanyaan dalam status, dan tidak ada yang baru dalam kasus tersebut.

Reducer untuk Data Papan Peringkat

Reducer untuk cabang papan peringkat pada dasarnya sama dengan reducer untuk cabang gameData, yang menggantikan seluruh cabang sebagai respons terhadap jenis tindakan UPDATE_LEADERBOARD. Jadi, tidak perlu membahasnya di sini. Mari kita bahas topik baru lainnya: cara yang lebih baik untuk menangani pengembangan multi-platform.

Pengembangan Multi-Platform yang Lebih Bersih

Saat kita melihat keseluruhan struktur aplikasi, saya memperkenalkan Anda pada mekanisme peralihan platform. Sekarang saatnya untuk melihat versi spesifik dari file yang menggunakan mekanisme ini, dimulai dengan file MainLayout*.

Versi Android

Tata letak utama menentukan tata letak keseluruhan untuk aplikasi, dan versi untuk Android, tentu saja, diberi nama MainLayout.android.js, untuk memanfaatkan mekanisme peralihan platform.

Mainlayout.Android.Js

Kita mulai (setelah impor yang tidak ditampilkan) dengan definisi StyleSheet yang sangat jarang.

```
const styles = StyleSheet.create({
  outerContainer: {flex: 1}
});
```

Ya, itu saja, dan ketika Anda melihat definisi komponen berikut, alasan untuk gaya ini seharusnya terlihat jelas.

```
export default class MainLayout extends React.Component {
  constructor(inProps) {
    super(inProps);
  }
```

```
  render() {
    return (
      <View style={styles.outerContainer}>
        <NamePromptModal />
        <EndGameModal />
        <AdminModal />
        <Drawer />
      </View>
    );
  }
}
```

Komponen View tingkat atas yang menggunakan gaya dengan flex:1 memastikan bahwa seluruh UI mengisi layar. Di dalam View tersebut terdapat empat komponen kustom: tiga modal, kode yang akan saya bahas nanti, dan komponen Drawer. Komponen Drawer merepresentasikan keseluruhan model navigasi aplikasi, dan ke sanalah kita akan menuju selanjutnya.

Mainnav.Android.Js

Seperti halnya berkas MainLayout.android.js, keseluruhan navigasi untuk aplikasi didefinisikan, untuk Android, dalam berkas MainNav.android.js dan, seperti yang disebutkan sebelumnya, kita akan menggunakan Drawer untuk ini.

```
      export default createDrawerNavigator(
      {
       GameScreen: {
     screen: GameScreen, navigationOptions: () => ({title: "Game"}),
  },
  InfoScreen: {
    screen: InfoScreen, navigationOptions: () => ({title: "Info" }),
  },
  AboutScreen: {
    screen: AboutScreen, navigationOptions: () => ({title: "About"}),
  },
},
{initialRouteName: "GameScreen",backBehavior: "none"}
);
```

NativeBase menyediakan komponen Drawer melalui fungsi createDrawerNavigation(). Konfigurasi yang diteruskan ke sana sebagai argumen pertama seharusnya terlihat familier, karena sangat mirip dengan TabNavigator yang digunakan di Restaurant Chooser. Setiap layar (rute) adalah entri dalam objek konfigurasi ini, dan kelas untuk masing-masing ditentukan oleh atribut layar. Untuk navigationOptions, kita hanya perlu menyediakan judul, karena tidak ada ikon di Drawer (bisa saja ada, tetapi saya memilih untuk tidak mencantumkannya di sini, tanpa alasan khusus selain untuk membuat konfigurasi sesederhana mungkin). Argumen kedua untuk createDrawerNavigation() adalah opsi untuk navigator itu sendiri. Sama seperti TabNavigator, initialRouteName menempatkan pengguna di GameScreen secara default, dan

opsi backBehavior memberi tahu navigator untuk tidak melakukan apa pun saat tombol kembali perangkat keras ditekan.

Versi iOS

Sekarang kita akan melihat sisi iOS, meskipun pada kenyataannya hanya ada satu file yang perlu diperiksa, dan tidak ada konsep baru di dalamnya, jadi ini tidak akan memakan waktu lama.

Mainlayout.Ios.Js

Lihat, berikut ini hal yang perlu diperhatikan tentang versi iOS dari berkas MainLayout.ios.js: berkas ini hampir sama dengan versi Android, kecuali dua perbedaan kecil.

- Alih-alih mengimpor Drawer dari MainNav, versi ini mengimpor Tabs dari MainNav (dan perhatikan bahwa MainNav tidak menentukan platform—pengemas React Native menanganinya untuk kita).
- Alih-alih komponen Drawer dalam metode render(), Tabs muncul sebagai gantinya.

Itulah, secara harfiah, semua perubahan yang diperlukan. Jadi, jangan berlama-lama. Mari beralih ke berkas MainNav.ios.js, yang memang berbeda dengan berkas MainNav untuk Android.

MainNav.Ios.Js

Alih-alih Drawer, untuk iOS, kita akan menggunakan TabNavigator yang sama seperti yang Anda lihat di Restaurant Chooser. Berikut kodenya, yang ditemukan dalam berkas MainNav.ios.js:

```
  export default createBottomTabNavigator(

{
 GameScreen: {
   screen: GameScreen,
   navigationOptions: {
     tabBarLabel: "Game",
     tabBarIcon: ({tintColor}) => (
       <Image source={require('../images/icon-game.png')}
              style={[styles.tabIcons, { tintColor }]}
       />
     )
   }
 },
 InfoScreen: {
   screen: InfoScreen,
   navigationOptions: {
     tabBarLabel: "Info",
     tabBarIcon: ({tintColor}) => (
       <Image source={require('../images/icon-info.png')}
              style={[styles.tabIcons, { tintColor }]}
       />
       )
     }
```

```
   },
  AboutScreen: {
    screen: AboutScreen,
    navigationOptions: {
      tabBarLabel: "About",
      tabBarIcon: ({tintColor}) => (
        <Image source={require('../images/icon-info.png')}
               style={[styles.tabIcons, { tintColor }]}
        />
      )
    }
  }
},
{
  initialRouteName: "GameScreen", animationEnabled: true, swipeEnabled: true,
  backBehavior: "none",lazy: false,tabBarPosition: "bottom",
  tabBarOptions: {activeTintColor: "#ff0000",showIcon: true
  }
);
```

Mungkin tidak ada gunanya mengulang kode ini, mengingat kode ini hampir identik dengan apa yang sudah Anda uraikan saat melihat Restaurant Chooser. Tentu saja, jika Anda tidak ingat pembahasan itu, Anda mungkin ingin meninjaunya kembali sekarang dan membandingkannya dengan kode ini, untuk melihat bahwa selain nama layar, tidak ada perubahan nyata. Kita dapat menghabiskan waktu untuk melihat komponen dan kode khusus untuk aplikasi ini.

16.7 KOMPONEN BERSAMA

Di direktori komponen, Anda akan menemukan file sumber yang menggunakan mekanisme peralihan platform untuk menentukan tata letak dan navigasi tingkat tinggi untuk aplikasi. Namun, semua hal lain di aplikasi tersebut dibagikan di antara kedua platform, jadi semua kode tersebut ada di direktori /components/shared (dan, sekali lagi, tidak ada aturan yang mengatakan harus seperti itu; itu hanya struktur yang menurut saya masuk akal). Kita akan mulai menjelajahi kode tersebut dengan melihat tiga modal.

NamePromptModal.js

Modal pertama yang harus dilihat juga merupakan yang pertama kali dilihat pengguna, dan yang akan selalu dilihatnya saat aplikasi dimulai: NamePrompt Modal. Ini adalah modal yang digunakan pengguna untuk memasukkan namanya atau menunjukkan bahwa ia adalah admin. Catatan Anda mungkin memperhatikan bahwa tidak ada kontrol yang diberikan pada siapa yang mengatakan bahwa mereka adalah admin. Siapa pun dan lebih dari satu orang dapat melakukannya. Ini adalah keputusan yang saya buat secara sadar, karena dua alasan. Yang pertama adalah modal mengurangi kerumitan dan membantu agar bab yang sudah panjang tidak bertambah panjang.

Alasan kedua adalah hal ini memberikan Anda kesempatan yang sangat baik untuk menguji pengetahuan yang telah Anda peroleh. Mungkin Anda harus meminta kata sandi, jika pengguna mengatakan bahwa ia adalah admin dan memvalidasinya di server. Atau, mungkin,

buatlah agar setelah pengguna mengatakan bahwa ia adalah admin, server tidak akan membiarkan orang lain mengaktifkan tombol tersebut (bahkan, mungkin menyembunyikan tombol tersebut sepenuhnya). Keduanya dapat digunakan, meskipun yang terakhir akan memerlukan perubahan yang lebih signifikan, dan itu bukanlah satu-satunya kemungkinan. Apa pun arah yang Anda pilih, menutup lubang tersebut merupakan latihan yang sangat baik.

Bagian kode pertama dalam berkas NamePromptModal.js ini, setelah impor, adalah definisi StyleSheet. Untuk menghemat ruang, saya akan menunjukkan kode ini di sini, dan daripada terus-menerus mengatakan "lalu gaya ini diterapkan," Anda harus merujuk kembali ke kode ini saat melihat salah satu gaya yang digunakan dalam kode komponen, untuk memahami cara penggunaan gaya tersebut. Jika ada gaya yang menurut saya tidak jelas atau sesuatu yang pernah Anda lihat sebelumnya, saya akan menunjukkannya secara spesifik, tetapi sebagian besarnya akan berupa flexbox dasar dan gaya font sederhana, jadi Anda seharusnya tidak akan kesulitan menguraikannya saat ini, karena sebagian besarnya akan sangat familiar.

```
const styles = StyleSheet.create({
  outerContainer: { flex: 1, alignItems: "center", justifyContent:
  "center", margin: 20},
  headingContainer: { height: 100, justifyContent: "center"},
  headingText: { fontSize: 20, fontWeight: "bold"},
  inputFieldContainer: { flex: 1, alignSelf: "stretch", justifyContent:
  "center"},
  switchContainer: { marginTop: 40, justifyContent: "center",
  flexDirection: "row"},
  buttonContainer: { height: 80, alignSelf: "stretch", justifyContent:
  "center"}
});
```

Satu hal menarik di sini adalah alignSelf yang diatur untuk meregang pada inputFieldContainer, yang merupakan wadah untuk kolom tempat pengguna memasukkan namanya. Hal ini diperlukan untuk memastikan bahwa kolom tersebut meregang di seluruh lebar modal. alignSelf menggantikan alignItems dari induk, yang di sini akan menjadi outerContainer dan pengaturan tengahnya.

Jika itu berlaku, kolom input akan dipusatkan, tetapi, tanpa menentukan lebar yang eksplisit, akan menggunakan beberapa nilai default yang kira-kira setengah dari lebar modal. Menggunakan alignSelf seperti ini memungkinkannya untuk mengembang agar sesuai dengan ruang yang tersedia, sementara tidak memengaruhi penyelarasan item lain dalam wadah tersebut. Sekarang kita dapat melihat kode untuk komponen ini, yang memiliki satu atau dua hal baru dan menarik untuk dilihat.

```
class NamePromptModal extends React.Component {

  constructor(inProps) {
    super(inProps);
}

render() {
```

```
    return (
      <Modal
        presentationStyle={"formSheet"}
        visible={this.props.isVisible}
        animationType={"slide"}
        onRequestClose={ () => {}}>

      <View style={styles.outerContainer}>
        <View style={styles.headingContainer}>
          <Text style={styles.headingText}>Hello, new player!</Text>
        </View>
      <View style={styles.inputFieldContainer}>
        <Item floatingLabel>
          <Label>Please enter your name</Label>
          <Input onChangeText={
            (inText) => store.dispatch(setPlayerName(inText))
          }
        />
      </Item>
      <View style={ styles.switchContainer}>
        <View>
          <Switch
            value={this.props.isAdmin}
            onValueChange={
              (inValue) => store.dispatch(setIsAdmin(inValue))
            }
          />
        </View>
          <View style={{ paddingLeft: 10}}>
            <Text>I am the admin</Text>
          </View>
        </View>
      </View>
      <View style={styles.buttonContainer}>
        <Button block onPress={CoreCode.startup}><Text>Ok</Text></Button>
      </View>
    </View>
  </Modal>
  );
  }
  }
```

Sebagian besar yang Anda lihat di sini seharusnya sudah biasa, karena Anda pernah melihat beberapa modal sebelumnya di Restaurant Chooser. Namun, yang mulai menarik adalah komponen Item yang membungkus komponen Input. Ini adalah komponen yang Anda lihat di Restaurant Chooser, tetapi yang baru adalah properti floatingLabel pada komponen Item, dan terkait dengan itu, komponen Label yang merupakan anak dari komponen Item. Fungsinya, bersama-sama, adalah menyediakan label di dalam komponen Input yang mengecil dan bergerak menjauh saat Input mendapat fokus. Ini adalah mekanisme luar biasa yang

memanfaatkan ruang secara efisien sekaligus informatif bagi pengguna.

Komponen baru lainnya di sini adalah komponen dari React Native itu sendiri, Switch. Ini memungkinkan pengguna untuk menentukan bahwa mereka adalah admin, dan setelah mereka mengklik tombol, mereka akan diarahkan ke AdminModal. Switch adalah widget yang bagus untuk membuat pilihan ya/tidak seperti ini. Saya juga perlu memberi label untuk komponen ini, agar pengguna tahu apa itu, jadi View yang berisi Switch dan komponen Text untuk label menggunakan flexDirection:row melalui gaya switchContainer, sehingga keduanya dapat berdampingan. Sedikit bantalan di sebelah kiri View yang berisi Text memastikan bahwa label tidak terjepit di Switch.

Hal lain yang sangat menarik di sini adalah ini adalah pertama kalinya Anda melihat Redux dikaitkan dengan komponen React Native, dan cara kerjanya kembali ke komponen Provider di file App.js. Komponen itulah yang memungkinkan kita untuk mengaitkan Redux ke kode lainnya. Komponen ini berasal dari paket baru: react-redux. Paket ini menyediakan kaitan ke Redux khusus untuk React (dan React Native sebagai ekstensi) dengan komponen Provider ini menjadi yang terpenting. Yang dilakukannya adalah membuat penyimpanan tersedia untuk semua komponen "terhubung" yang merupakan turunan dari komponen Provider. Dan apa artinya menjadi komponen "terhubung", Anda bertanya? Nah, di sinilah kode selanjutnya berperan.

```
const mapStateToProps = (inState) => {
  return {
    isVisible: inState.modals.namePromptVisible,
    isAdmin: inState.modals.isAdmin
};
};
```

Pengenalan status ke suatu komponen dilakukan melalui properti komponen. Untuk melakukannya, kita harus menyediakan fungsi yang mengambil status (lebih tepatnya, beberapa cabang dari pohon status) dan memetakan data darinya ke properti. Kemudian, Anda membungkus komponen dalam panggilan fungsi connect(), hal lain yang disediakan oleh paket react-redux, seperti ini:

```
export default connect(mapStateToProps)(NamePromptModal);
```

Hasilnya adalah komponen ini, yang sekarang kita anggap terhubung, akan memiliki dua properti baru yang tersedia untuknya, isVisible dan isAdmin, yang masing-masing sesuai dengan atribut status modals. namePromptVisible dan modals.isAdmin. Anda kemudian menggunakan properti tersebut seperti halnya Anda menggunakan properti lainnya, misalnya, sebagai properti value pada Switch, seperti yang dapat Anda lihat, atau sebagai nilai properti visible dari modal itu sendiri.

Nilai dalam status digunakan saat merender komponen, tidak berbeda dengan saat menggunakan variabel JavaScript yang tidak dikelola oleh Redux. Kemudian, setiap perubahan dalam komponen ditransfer kembali ke penyimpanan status. Lebih tepatnya, perubahan

tersebut dapat ditransfer kembali ke status, jika Anda menulis kode yang sesuai dalam event handler untuk melakukan dispatch() tindakan, seperti yang Anda lihat pada properti onTextChange komponen Input dan prop onValueChange Switch. Dan di sana Anda memiliki jawaban untuk pertanyaan dari awal bab: bagaimana modal ditampilkan atau disembunyikan.

Ketiga modal tersebut sebenarnya selalu ada, karena ketiganya merupakan turunan dari komponen tingkat atas dalam berkas MainLayout*, dan ditampilkan sebagai hasil dari atribut terkait di cabang modal pohon status yang ditetapkan ke true, atau disembunyikan saat ditetapkan ke false (yang, Anda akan perhatikan sekarang, merupakan nilai default di initialState.js). Ini hanya masalah pengiriman tindakan SHOW_HIDE_MODAL yang sesuai, meneruskan nama modal yang ingin kita tampilkan atau sembunyikan dan, tentu saja, apakah akan benar-benar menampilkan atau menyembunyikannya, dan saat Redux memperbarui pohon status, visibilitas modal juga akan diperbarui, tanpa kita harus melakukan hal lain, karena modal adalah komponen yang terhubung.

Cukup keren, bukan?

Terakhir, perhatikan bahwa pengendali onPress Button merujuk ke metode startup() dari beberapa objek yang disebut CoreCode. Ini adalah kode yang akan kita lihat setelah semua kode untuk komponen, tetapi hal yang menarik di sini adalah bahwa nilai dari prop onPress bukanlah fungsi anonim yang memiliki beberapa kode di dalamnya yang dipanggil di sana. Sebaliknya, ini adalah referensi ke fungsi yang ada, dan dalam kasus ini, ini adalah referensi ke metode suatu objek. Ini adalah sesuatu yang belum pernah Anda lihat sebelumnya, tetapi bisa dibilang ini adalah cara yang lebih baik untuk menulis kode React Native.

Beberapa pengembang menganjurkan untuk tidak pernah memiliki kode sebaris dalam komponen. Misalnya, alih-alih memiliki fungsi anonim yang memanggil dispatch() secara langsung pada komponen Input, seperti yang Anda lihat di sini, yang tertanam di dalamnya, mereka berpendapat bahwa Anda harus merujuk beberapa fungsi, mungkin updateInputValue(), di suatu tempat (baik metode atau objek) dan merujuknya sebagai nilai dari prop onTextChange.

Pemikirannya adalah bahwa ia memisahkan kode yang melakukan tindakan dari kode yang mendefinisikan komponen. Namun, beberapa pengembang lain berpendapat bahwa komponen sudah menjadi hal yang independen dan terenkapsulasi dan, oleh karena itu, memiliki kode yang tertanam di dalamnya adalah benar, tepat, dan tepat. Saya serahkan keputusan arsitektural/filosofis kepada Anda, tetapi saya tetap ingin menunjukkan kedua pendekatan tersebut.

EndGameModal.js

EndGameModal, yang ditemukan dalam berkas EndGameModal.js, tentu saja, adalah apa yang dilihat pengguna saat permainan berakhir. Ini adalah modal sederhana yang memberi tahu pemain apa hasilnya. Komponen ini menggunakan StyleSheet berikut:

```
const styles = StyleSheet.create({
  outerContainer: { flex: 1, alignItems: "center", justifyContent:
  "center", margin: 20},
  headingContainer: { height: 100, justifyContent: "center"},
```

```
  headingText: { fontSize: 20, fontWeight: "bold"},
  messageContainer: { flex: 1, alignSelf: "center", justifyContent:
  "center"},
  buttonContainer: { height: 80, alignSelf: "stretch", justifyContent:
  "center"},
  buttonText: { fontWeight: "bold", color: "white"}
});
```

The code for the component is as follows:

```
class EndGameModal extends React.Component {

    constructor(inProps) {
       super(inProps);
  }

render() {

   return (
     <Modal
     presentationStyle={"formSheet"}
     visible={this.props.isVisible}
     animationType={"slide"}
     onRequestClose={ () => {}}
   >
     <View style={styles.outerContainer}>
       <View style={styles.headingContainer}>
         <Text style={styles.headingText}>Game over</Text>
       </View>
       <View style={styles.messageContainer}>>
         <Text>{this.props.message}</Text>
       </View>
       <View style={styles.buttonContainer}>
         <Button block onPress={ () => {}}>
           <Text style={ styles.buttonText}>0k</Text>
         </Button>
       </View>
      </View>
    </Modal>
   );
}
}
```

Di dalam komponen View tingkat atas, yang memiliki beberapa gaya margin untuk memberi ruang di sekelilingnya, terdapat View yang berisi judul. View ini memiliki tinggi yang ditetapkan untuk membantu memastikan spasi yang tepat. Setelah itu ada View lain yang kemudian menyertakan komponen Text. Teks yang ditampilkan di sana diambil dari prop message, yang berasal dari status yang dikontrol Redux. Terakhir, View yang berisi komponen Button menyelesaikan semuanya.

Perhatikan bahwa prop onPress adalah fungsi kosong. Prop onPress diperlukan oleh React Native, tetapi tidak ada pekerjaan yang sebenarnya harus dilakukan; modal akan ditutup saat Button diketuk secara otomatis, oleh karena itu fungsinya kosong. Komponen ini juga terhubung, jadi memiliki fungsi mapStateToProps() dan menggunakan fungsi connect(). Pemetaan prop adalah isVisible yang dipetakan ke modals.endGameVisible dan message dipetakan ke modals.endGameMessage.

AdminModal.js

Seperti EndGameModal, AdminModal juga cukup sederhana, meskipun karena ada beberapa elemen interaktif, ada sedikit hal yang perlu dilihat. Dimulai dengan StyleSheet di file AdminModal.js, kita memiliki

```
const styles = StyleSheet.create({
   outerContainer:  {  flex:  1,  margin:  50,  justifyContent:  "center",
   alignItems: "center"},
   headingText: { fontSize: 40, fontWeight: "bold", margin: 50},
   buttonContainer: { margin: 50},
   currentStatusContainer: { margin: 50},
   currentStatusText: { fontSize: 20, fontWeight: "bold", color: "red"}
});
```

Seperti halnya EndGameModal, semuanya adalah gaya dasar yang sederhana. Setelah itu adalah kode komponen yang sebenarnya, seperti yang dapat Anda lihat di sini:

```
class AdminModal extends React.Component {

  constructor(inProps) {
   super(inProps);
}
  render() {

   return ()
     <Modal
      presentationStyle={"fullScreen"}
      visible={this.props.isVisible}
       animationType={"slide"}
       onRequestClose={ () => {}}
      >
       <View style={styles.outerContainer}>
        <Text style={styles.headingText}>Admin</Text>
        <View style={styles.buttonContainer}>
          <Button title="New Game"
            onPress={ () => {
<View style={styles.buttonContainer}>
  <Button title="Next Question"
    onPress={() => {
      CoreCode.io.emit("adminNextQuestion", {});
  }}
  />
```

```
</View>
<View style={styles.buttonContainer}>
  <Button title="End Game"
    onPress={() => {
      CoreCode.io.emit("adminEndGame", {});
  }}
  />
</View>
<View style={styles.currentStatusContainer}>
  <Text style={styles.currentStatusText}>
    Current Status: {this.props.currentStatus}
  </Text>
</View>
</View>
</Modal>
);
}
```

Intinya, modal ini tidak lain adalah komponen Teks untuk judul, diikuti oleh tiga Tombol, masing-masing di dalam Tampilan, yang ditata untuk memastikan jarak yang konsisten di antara keduanya. (Kami tidak ingin ketukan yang tidak disengaja hanya karena tombolnya terlalu berdekatan.) Setiap tombol memancarkan pesan socket.io ke server, yang tidak memerlukan muatan apa pun.

Setelah tombol, ada Tampilan lain dengan komponen Teks di dalamnya. Di sinilah pesan apa pun yang dikembalikan dari server akan ditampilkan. Anda dapat melihat di sini contoh lain tentang bagaimana komponen Teks menggunakan nilai dari status melalui referensi this.props. currentStatus, semua berkat paket redux-react yang menghubungkan berbagai hal dengan penyimpanan. Komponen ini juga terhubung, sehingga memiliki fungsi mapStateToProps() dan menggunakan fungsi connect(). Pemetaan prop adalah isVisible yang dipetakan ke modals. adminVisible dan currentStatus dipetakan ke modals.currentStatus.

AboutScreen.js

Setelah membahas tiga modal, sekarang kita akan membahas layar aplikasi, dimulai dengan yang mungkin paling sederhana, AboutScreen. Seperti modal, kita akan melihat StyleSheet terlebih dahulu, di berkas AboutScreen.js.

```
const styles = StyleSheet.create({
  outerContainer: {flex: 1, alignItems: "center", justifyContent: "center"},
  spacer: {flex: .2},
  textContainer: {flex: .15, justifyContent: "center", alignItems:
"center"},
  textTitle: {fontWeight: "bold", fontSize: 20},
  textVersion: {fontWeight: "bold", fontSize: 18},
  textSource: {fontWeight: "bold", fontSize: 16},
  textAuthor: {fontWeight: "bold", fontSize: 14}
});
```

Satu hal yang perlu diperhatikan di sini: saat Anda melihat kode untuk komponen tersebut,

Anda akan memahami bahwa ada enam komponen View yang terlibat yang merupakan anak dari View dengan outerContainer yang diterapkan. Ada satu untuk masing-masing dari empat bagian informasi yang akan ditampilkan (judul, versi, sumber, dan penulis), lalu ada dua yang tidak berisi konten sama sekali. Anda akan menemukan bahwa keduanya memiliki gaya spacer yang diterapkan. Saat Anda menambahkan nilai flex untuk keenam View tersebut (.2 + .15 + .15 + .15 + .15 + 2), Anda akan melihat bahwa totalnya menjadi 1, seperti yang diharapkan.

Melakukannya dengan cara ini, dengan flex, daripada padding atau margin atau tinggi statis, berarti bahwa konten layar ini akan meregang atau menyusut sesuai dengan ukuran layar. Yang pasti, sebagian besar layar lain yang Anda lihat di Restaurant Chooser atau aplikasi ini akan melakukan itu sampai tingkat tertentu juga, tetapi ini adalah pertama kalinya Anda melihat satu yang akan mengembang atau menyusut sepenuhnya, berdasarkan ukuran layar. Dengan kata lain, ini adalah tata letak yang sepenuhnya responsif. Tidak selalu memungkinkan atau bahkan disarankan untuk membuat tata letak seperti ini.

Sering kali, Anda akan memiliki beberapa kombinasi item fleksibel dan item berukuran tetap, tetapi jika Anda dapat melakukannya, seperti pada layar ini, itu bukan hal yang buruk. Setelah itu, mari kita lihat kodenya.

```
export default class AboutScreen extends React.Component {

   constructor(inProps) {
      super(inProps);
}

render() {
   return (
      <View style={styles.outerContainer}>
         <View style={styles.spacer} />
         <View style={styles.textContainer}>
            <Text style={styles.textTitle}>RNTrivia (React Native Trivia)</Text>
         </View>
         <View style={styles.textContainer}>
            <Text style={styles.textVersion}>v1.0</Text>
         </View>
         <View style={styles.textContainer}>
            <Text style={styles.textSource}>Published in the Apress
            book</Text>
            <Text style={styles.textSource}>Practical React Native
            Projects</Text>
            <Text style={styles.textSource}>in 2018</Text>
         </View>
         <View style={styles.textContainer}>
            <Text style={styles.textAuthor}>By Frank W. Zammetti</Text>
         </View>
         <View style={styles.spacer} />
      </View>
   );
}
}
```

Ya, seperti yang dijanjikan, ini lebih dari sekadar Tampilan kontainer luar dengan enam komponen Tampilan anak. Kecuali yang pertama dan terakhir, yang merupakan Tampilan pengatur jarak, masing-masing memiliki satu atau lebih komponen Y di dalamnya. Semua informasi yang ditampilkan dikodekan secara kaku. Tidak ada yang perlu ada di sini, jadi ini sejelas mungkin. Layar ini bahkan tidak perlu dihubungkan, jadi ini adalah kode lengkap kali ini.

InfoScreen.js

Berikutnya adalah InfoScreen, yang ditampilkan pada Gambar 6-4 dan ditempatkan di berkas InfoScreen.js, tentu saja.

Gambar 16.8 Layar Info Permainan

Ini adalah layar yang agak sederhana, seperti AboutScreen, dimulai dengan StyleSheet, seperti biasa.

```
const styles = StyleSheet.create({
  outerContainer:
    {justifyContent: "center", marginTop: 50, marginLeft: 20, marginRight:
    20},
  identificationCardContainer: {height: 150, marginBottom: 20},
  currentGameCardContainer: {height: 360},
  headerText: {fontWeight: "bold", fontSize: 20, color: "red"},
  fieldContainer: {flexDirection: "row"},
  fieldLabel: {width: 100, fontWeight: "bold"},
  fieldSpacing: {marginBottom: 12}
});
```

Dan, untuk melihat bagaimana gaya-gaya ini digunakan, kita harus memeriksa kode komponen (yang memiliki beberapa hal baru di dalamnya, meskipun gayanya sebenarnya tidak).

```
class InfoScreen extends React.Component {

   constructor(inProps) {
      super(inProps);
}
render() {
   return (
      <View style={styles.outerContainer}>
      <View style={styles.identificationCardContainer}>
         <Card>
            <CardItem header>
               <Text style={styles.headerText}>Identification</Text>
            </CardItem>
         <CardItem>
      <Body>
         <View style={styles.fieldContainer}>
            <Text style={styles.fieldLabel}>Player Name</Text>
            <Text>{this.props.playerName}</Text>
         </View>
         <View style={styles.fieldContainer}>
            <Text style={styles.fieldLabel}>Player ID</Text>
            <Text>{this.props.playerID}</Text>
         </View>
      </Body>
      </CardItem>
   </Card>
</View>
```

Daripada memasukkan semua kode dalam satu daftar besar, saya akan membaginya sedikit, berhenti sebentar di sini, sehingga saya dapat berbicara tentang apa yang baru. NativeBase, seperti yang telah Anda lihat, menyediakan banyak komponen yang apik di atas apa yang ditawarkan React Native secara langsung. Beberapa di antaranya mengambil inspirasi dari satu platform seluler atau lainnya, meskipun berfungsi di semua platform. Pada modal ini, kita memiliki sesuatu yang diambil dari buku pedoman Android: komponen Card.

Metafora Card umum di Android sebagai cara untuk membedakan satu set konten dari yang lain. Pada akhirnya, itu hanyalah wadah yang kuat untuk konten; ia tidak melakukan apa pun sendiri. Komponen Card NativeBase adalah wadah konten yang fleksibel dan dapat diperluas yang mencakup opsi untuk header dan footer, berbagai macam konten (hampir apa saja, sebenarnya), warna latar belakang kontekstual (opsional), dan serangkaian opsi tampilan yang kuat.

Namun, semua yang dikatakan, dalam bentuk yang paling murni, itu pada dasarnya hanyalah sebuah kotak, tetapi kotak dengan bayangan, sehingga sedikit menonjol dari layar, dan juga memastikan jarak yang tepat antara kartu lain, jika ada, tanpa Anda harus memperhitungkannya. Itu semua tanpa menggunakan opsi apa pun yang tersedia untuknya,

yang merupakan cara penggunaannya di RNTrivia pada layar ini. Semuanya dimulai dengan komponen Card, dan di dalamnya akan terdapat setidaknya satu komponen CardItem. Jika hanya ada satu CardItem, maka itu adalah konten untuk Card. Jika ada lebih dari satu, maka salah satunya harus memiliki properti header di atasnya, menjadikannya elemen header.

Demikian pula, Anda dapat memiliki CardItem dengan properti footer, menjadikannya footer. Anda masih dapat meletakkan hampir semua konten yang Anda suka di CardItem header atau footer, tetapi komponen card akan merendernya di bagian atas dan bawah, masing-masing, sesuai dengan namanya. Seperti yang dapat Anda lihat dari Kartu pertama ini, yang merupakan kartu identifikasi, kami memiliki informasi yang mengidentifikasi pemain, playerName, dan playerID miliknya.

Masing-masing terdapat dalam View dengan gaya fieldContainer yang diterapkan untuk mengaktifkan tata letak fleksibel berbasis baris, sehingga dua komponen Text, satu untuk label bidang dan satu untuk nilai, ditampilkan berdampingan. Komponen Text nilai merujuk pada beberapa properti (playerName dan playerID) yang ditambahkan ke komponen melalui fungsi mapStateToProps(), sehingga nilai dari penyimpanan status Redux adalah apa yang ditampilkan di dalamnya. Sekarang, saya dapat berbicara tentang sisa kode, yang sebenarnya hanya pengulangan dari apa yang baru saja dijelaskan, dengan beberapa bidang lagi yang terlibat.

```
<View style={styles.currentGameCardContainer}>
     <Card>
       <CardItem header>
         <Text style={styles.headerText}>Current Game</Text>
       </CardItem>
       <CardItem>
     <Body>
       <View style={[ styles.fieldContainer, styles.fieldSpacing ]}>
         <Text style={styles.fieldLabel}>Asked</Text>
         <Text>{this.props.asked}</Text>
       </View>
       <View style={[ styles.fieldContainer, styles.fieldSpacing ]}>
         <Text style={styles.fieldLabel}>Answered</Text>
         <Text>{this.props.answered}</Text>
       </View>
       <View style={[ styles.fieldContainer, styles.fieldSpacing ]}>
         <Text style={styles.fieldLabel}>Points</Text>
         <Text>{this.props.points}</Text>
       </View>
       <View style={[ styles.fieldContainer, styles.fieldSpacing ]}>
         <Text style={styles.fieldLabel}>Right</Text>
         <Text>{this.props.right}</Text>
       </View>
       <View style={[ styles.fieldContainer, styles.fieldSpacing ]}>
         <Text style={styles.fieldLabel}>Wrong</Text>
         <Text>{this.props.wrong}</Text>
       </View>
       <View style={[ styles.fieldContainer, styles.fieldSpacing ]}>
```

```
            <Text style={styles.fieldLabel}>Total Time</Text>
            <Text>{this.props.totalTime}</Text>
          </View>
          <View style={[ styles.fieldContainer, styles.fieldSpacing ]}>
            <Text style={styles.fieldLabel}>Slowest</Text>
            <Text>{this.props.slowest}</Text>
          </View>
          <View style={[ styles.fieldContainer, styles.fieldSpacing ]}>
            <Text style={styles.fieldLabel}>Fastest</Text>
            <Text>{this.props.fastest}</Text>
          </View>
          <View style={ styles.fieldContainer}>
            <Text style={styles.fieldLabel}>Average</Text>
            <Text>{this.props.average}</Text>
          </View>
            </Body>
          </CardItem>
        </Card>
      </View>
    </View>
    );
    }
    }
```

Lihat? Ini hanya Kartu lain, kali ini, dengan sembilan bidang, semua data berasal dari cabang gameData dari pohon status dan menyampaikan informasi kepada pengguna tentang permainan saat ini. Perhatikan bagaimana beberapa gaya diterapkan ke wadah bidang, gaya fieldContainer, seperti yang diharapkan, dan juga gaya fieldSpacing pada semua kecuali yang terakhir. Lihat gaya fieldSpacing. Gaya ini hanya memiliki spesifikasi marginBottom. Jadi, menerapkannya ke setiap bidang memberi kita beberapa ruang di antara bidang, tetapi, tentu saja, itu tidak diperlukan pada bidang terakhir. Komponen ini juga terhubung, jadi ia memiliki fungsi mapStateToProps() dan menggunakan fungsi connect().

Pemetaan prop adalah playerName dipetakan ke playerInfo.name, playerID dipetakan ke playerInfo.id, asked dipetakan ke gameData.asked, answered dipetakan ke gameData.answered, points dipetakan ke gameData.points, right dipetakan ke gameData.right, wrong dipetakan ke gameData.wrong, totalTime dipetakan ke gameData.totalTime, slowest dipetakan ke gameData.slowest, faster dipetakan ke gameData.fastest, dan average dipetakan ke gameData.average. GameScreen.js Sekarang saatnya beralih ke layar terakhir dari tiga layar, yang, seperti yang Anda ingat, terdiri dari tiga sublayar, jadi bisa dibilang begitu. GameScreen.js adalah induk dari ketiganya, dan sumber yang ditemukan di sana sangat kecil.

```
export default createSwitchNavigator(
  {
    GameHomeScreen: {screen: GameHomeScreen},
    GameLeaderboardScreen: {screen: GameLeaderboardScreen},
    GameQuestionScreen: {screen: GameQuestionScreen}
```

```
    },
    {headerMode: "none", initialRouteName: "GameHomeScreen"}
);
```

Di sini, kita menemukan navigator React Navigation yang baru. SwitchNavigator dalam banyak hal sama seperti StackNavigator, yaitu untuk mengelola sekumpulan layar yang hanya menampilkan satu layar pada satu waktu. Perbedaan utama antara keduanya adalah bahwa dengan StackNavigator, setiap transisi ke layar baru menempatkan layar tersebut di bagian atas tumpukan, tetapi riwayat navigasi pengguna melalui layar tetap dipertahankan.

Dengan begitu, pengguna dapat kembali dengan mudah melalui StackNavigator secara internal, cukup dengan mengeluarkan setiap layar dari tumpukan. Tentu saja, jika Anda menonaktifkan tombol kembali, seperti yang saya lakukan di Restaurant Chooser, perbedaan antara keduanya pada dasarnya tidak ada. SwitchNavigator tidak memiliki tumpukan seperti itu, jadi tidak ada riwayat navigasi untuk menelusurinya kembali.

Selain perbedaan itu, keduanya berfungsi sama dan, yang lebih penting, dikonfigurasikan sama, seperti yang ditunjukkan kode di sini, karena tampilannya hampir identik dengan konfigurasi StackNavigator di Restaurant Chooser. Perhatikan bahwa initialRouteName menetapkan bahwa GameHomeScreen, salah satu dari tiga sublayar, adalah yang pertama kali ditampilkan, dan itulah kode yang akan ditinjau berikutnya.

GameScreen-Home.js

GameHomeScreen dalam file GameScreen-Home.js adalah layar yang sangat sederhana yang hanya berupa logo, landasan pendaratan, jika Anda mau, bagi pengguna, saat aplikasi pertama kali dimulai (namun, jika permainan sedang berlangsung, mereka akan segera diarahkan ke layar papan peringkat, tetapi itu akan dibahas nanti). Kode dimulai dengan satu gaya.

```
const styles = StyleSheet.create({
   outerContainer: {flex: 1, alignItems: "center", justifyContent:
   "center"}
});
```

Satu-satunya tujuan adalah memastikan bahwa anak-anak wadah tempat gaya ini diterapkan dipusatkan baik secara vertikal maupun horizontal dan, seperti yang telah menjadi kebiasaan kita, wadah ini mengisi layar. Kode komponen setelah itu mudah dipahami.

```
export default class GameHomeScreen extends React.Component {

   constructor(inProps) {
     super(inProps);
     CoreCode.mainNavigator = inProps.navigation;
 }

render() {
  return (
    <View style={styles.outerContainer}>
```

```
      <Image source={require(" .. / .. /images/logo.png")} />
    </View>
  );
}
}
```

Dalam konstruktor, selain super() yang biasa, kita memiliki sesuatu yang akan kita perlukan nanti, yaitu, menyimpan referensi ke navigator yang mengelola layar ini. Ini akan menjadi penting ketika kode harus menavigasi dari satu layar ke layar lain secara otomatis. Ini hanya mungkin jika Anda memiliki referensi ke navigator, tetapi Anda hanya memiliki referensi ke navigator dari dalam kode komponen, karena secara otomatis diteruskan sebagai prop navigasi oleh React Navigation.

Jadi, referensi ke sana disimpan pada objek CoreCode, yang akan menjadi bit kode terakhir yang kita lihat dalam bab ini, sehingga membuatnya tersedia di luar kode komponen, yang persis seperti yang dibutuhkannya, seperti yang akan Anda lihat. Selain itu, seperti yang dinyatakan sebelumnya, layar ini hanyalah logo di dalam wadah View, dengan gaya outerContainer yang diterapkan. Tidakkah Anda suka ketika kode tersebut ternyata sesederhana yang dijelaskannya? Saya tahu saya suka.

GameScreen-Leaderboard.js

Layar papan peringkat, yang ditemukan dalam berkas GameScreen-Leaderboard.js, adalah layar kedua dari tiga sublayar layar permainan, dan diawali dengan StyleSheet, seperti biasa.

```
const styles = StyleSheet.create({
     outerContainer: {flex: 1, alignItems: "stretch", justifyContent:
  "center", marginTop: 50},
     headingContainer: {height: 150, justifyContent: "center", alignSelf:
  "center"},
     headingText: {fontSize: 34, fontWeight: "bold"},
     listContainer: {flex: .6, marginLeft: 20, marginRight: 20,
     marginBottom: 40, borderColor: "silver", borderWidth: 2, padding: 10},
     awaitingQuestionContainer: {flex: . 4},
     awaitingQuestionWebView: {backgroundColor: "transparent"}
});
```

Struktur keseluruhan layar ini adalah ada tiga bagian utama: bagian header, bagian daftar (tempat papan peringkat itu sendiri ditampilkan), dan bagian dengan beberapa teks yang memberi tahu pemain bahwa ia sedang menunggu pertanyaan. Di situlah gaya headingContainer, listContainer, dan awaitingQuestionContainer berperan, karena gaya tersebut adalah gaya untuk komponen Tampilan kontainer dari ketiga bagian tersebut, dan Anda dapat melihat bahwa nilai flex-nya berjumlah 1 (dengan headingContainer memiliki tinggi statis).

awaitingQuestionWebView dengan backgroundColor yang disetel ke transparan adalah sesuatu yang akan saya bahas setelah kita melihat kode komponen (sekali lagi, karena saya

pikir akan lebih masuk akal untuk meletakkannya dalam urutan ini). Ada potongan kode yang muncul setelah StyleSheet dan sebelum kode komponen, tetapi saya akan kembali ke sana, karena saya pikir akan lebih masuk akal jika Anda melihat kode komponen terlebih dahulu.

```
class GameLeaderboardScreen extends React.Component {
   constructor(inProps) {
     super(inProps);
 }
render() {

return (
   <View style={styles.outerContainer}>
    <View style={styles.headingContainer}>
<Text style={styles.headingText}>Current Leaderboard</Text>
</View>
<View style={styles.listContainer}>
  <FlatList
    data={this.props.listData}
    keyExtractor={(inItem) => inItem.playerID}
    renderItem={({ item}) => {
      return (
        <View style={{ flex: 1, flexDirection: "row"}}>
          <View style={{ flex: 0.6}}>
            <Text style={{ fontSize: 20}}>
              {item.playerName}
              {store.getState().playerInfo.id === item.playerID? " (You)": ""}
            </Text>
          </View>
          <View style={{ flex: 0.4}}>
            <Text style={{ fontSize: 20}}>{item.points} points</Text>
          </View>
        </View>
      );
  }}
  />
</View>
<View style={styles.awaitingQuestionContainer}>
  <WebView
    style={styles.awaitingQuestionWebView}
    source={{ html: awaitingQuestionHTML}}
  />
</View>
      />
    </View>
   </View>
  );
}
}
```

Oke, jadi kita melihat Tampilan luar dengan gaya outerContainer diterapkan, seperti biasa. Lalu

kita memiliki Tampilan dengan headingContainer diterapkan, jadi itu bagian pertama dari tiga bagian, dan di dalamnya ada komponen Teks dengan headingText diterapkan. Sejauh ini sederhana. Setelah itu adalah wadah Tampilan untuk FlatList yang merupakan papan peringkat itu sendiri. Data berasal dari penyimpanan Redux kita, ditransfer ke prop listData oleh fungsi mapStateToProps() yang muncul kemudian.

Perhatikan bahwa prop keyExtractor digunakan dan menggunakan playerID sebagai kunci unik untuk setiap item dalam daftar, yang menghindari React Native meneriaki kita tentang kunci yang hilang, meskipun kita tidak memerlukan kunci di sini. Prop renderItem merender Tampilan yang membentang di layar untuk setiap item dalam daftar, dan di dalam Tampilan itu ada dua komponen Tampilan lagi, satu untuk nama pemain dan satu untuk poin mereka saat ini. Jika pemain itu kebetulan Anda, seperti yang pasti akan terjadi pada salah satu dari mereka, maka teks "(Anda)" ditambahkan ke nama pemain.

Perhatikan bagaimana store.getState().playerInfo.id direferensikan secara langsung untuk melakukan pemeriksaan ini. Saya melakukannya dengan cara ini hanya untuk menunjukkan bahwa Anda dapat melakukannya, tetapi, dalam praktiknya, akan lebih baik untuk memetakannya ke properti dan mengaksesnya dengan cara itu. Setelah daftar, kita memiliki wadah View ketiga dari tiga wadah, yang satu ini berisi komponen WebView.

Komponen ini adalah salah satu yang dapat kita gunakan untuk menampilkan HTML sembarangan, dan alasan komponen ini digunakan di sini adalah agar saya dapat menunjukkan pesan kepada pengguna, memberi tahu bahwa ia sedang menunggu pertanyaan dan membuatnya berputar. Dengan WebView, Anda dapat memuat konten dari jaringan, atau dari sistem file, atau Anda dapat memasukkan HTML ke dalamnya secara langsung, seperti yang dilakukan di sini, dan HTML yang dimasukkan adalah ini:

```
const awaitingQuestionHTML = '
<style>${awaitingQuestionSpinStyles}</style>
<div class="spinText">Awaiting Question</div>
`;
```

Seperti yang Anda lihat, ini sebenarnya hanya sebuah fragmen HTML, bagian <style> yang berisi awaitingQuestionSpinStyles yang disisipkan ke dalamnya menggunakan interpolasi string, lalu dengan teks sebenarnya di dalamnya dan kelas gaya spinText yang diterapkan. Kelas gaya tersebut adalah sebagai berikut:

```
const awaitingQuestionSpinStyles = '
.spinText {
  animation-name: spin, depth;
  animation-timing-function: linear;
  animation-iteration-count: infinite;
  animation-duration: 3s;
  text-align: center;

   font-weight: bold;
   color: red;
```

```
   font-size: 24pt;
   padding-top: 100px;
}

@keyframes spin {
  from { transform: rotateY(odeg);}
  to { transform: rotateY(-360deg);}
}
@keyframes depth {
  0 { text-shadow: 0 0 black;}
  25% { text-shadow: 1px 0 black, 2px 0 black, 3px 0 black, 4px 0 black,
  5px 0 black;}
  50% { text-shadow: 0 0 black;}
  75% { text-shadow: -1px 0 black, -2px 0 black, -3px 0 black, -4px 0
  black, -5px 0 black;}
  100% { text-shadow: 0 0 black;}
}
 `;
```

Hasil dari semua ini adalah Anda memiliki WebView, dengan gaya awaitingQuestionWebView yang diterapkan, sehingga membuat latar belakangnya transparan, yang berarti bahwa yang akan Anda lihat hanyalah konten di dalamnya. (Tanpa gaya tersebut diterapkan, WebView akan menutupi apa yang ada di bawahnya, yang terlihat jelek dan sama sekali tidak mulus.) Kemudian HTML yang ditampilkan di dalamnya menggunakan beberapa animasi dan transformasi CSS untuk memutar teks. Ini mungkin bukan satu-satunya cara untuk mencapainya, bahkan mungkin bukan yang terbaik, tetapi ini menunjukkan bagaimana Anda dapat menggunakan komponen WebView dengan cara yang menarik.

Jika Anda pernah bekerja dengan React Native dan merasa apa yang ingin Anda capai membuat Anda kesulitan, tetapi itu akan mudah dilakukan dalam HTML biasa, ini adalah salah satu cara Anda dapat langsung melanjutkan dan mencampur HTML biasa ke dalam aplikasi Anda dan membuka kekuatan penuh HTML dan CSS. Tentu saja, Anda harus ingat bahwa apa pun yang Anda lakukan di sana tentu saja merupakan tampilan terpisah, semacam tag <iframe> dalam HTML, jadi Anda harus mengenali batasannya (hal-hal seperti konten dalam WebView tidak menyadari React Native dan sebaliknya) dan, seperti biasa, kinerjanya. Namun, untuk hal seperti ini, ini berfungsi dengan baik. Komponen ini juga terhubung, jadi ia memiliki fungsi mapStateToProps() dan menggunakan fungsi connect(). Pemetaan prop adalah peta listData ke leaderboard. listData dan, sebenarnya, hanya itu untuk komponen ini.

GameScreen-Question.js

Yang terakhir dari tiga sublayar layar permainan adalah layar pertanyaan, tempat pemain melihat pertanyaan saat ini dan menjawabnya. Kode ini ditempatkan dalam berkas GameScreen-Question.js dan dimulai dengan definisi StyleSheet.

```
  const styles= StyleSheet.create({
    outerContainer: {flex: 1, alignItems: "stretch", justifyContent: "center",
    marginTop: 50, marginLeft: 20, marginRight: 20},
    questionContainer: {flex: .2, justifyContent: "center", alignSelf:
```

```
    "center"},
    answerButtonsContainer: {flex: .8, alignItems:  "center", justifyContent:
    "center"},
    submitButtonContainer: {justifyContent: "center", height: 140},
    question: {fontWeight: "bold", fontSize: 26, color:  "red", textAlign:
  "center"},
    answerButton: {marginTop: 20},
    buttonText: {fontWeight: "bold", color: "white"}
    });
```

Pada layar ini, kita memiliki tiga bagian utama: Tampilan tempat pertanyaan ditampilkan (diberi gaya questionContainer), Tampilan tempat enam tombol jawaban berada (diberi gaya answerButtonsContainer), dan Kirim Jawaban di bagian bawah (diberi gaya submitButtonContainer). Dua wadah Tampilan pertama menggunakan fleksibilitas .2 dan .8, masing-masing, dengan wadah untuk tombol memiliki tinggi statis. Sekarang untuk kode komponen, dan saya akan menguraikannya sedikit untuk menjelaskannya dengan lebih baik, dimulai dengan ini:

```
class GameQuestionScreen extends React.Component {

   constructor(inProps) {
    super(inProps);
 }
   render() {

   return (
     <View style={styles.outerContainer}>
       <View style={styles.questionContainer}>
<RNText style={styles.question}>{this.props.question}</RNText>
     </View>
       <View style={styles.answerButtonsContainer}>
         <Button
          full
          style={styles.answerButton}
          primary={this.props.answerButtonPrimary[o]}
          danger={this.props.answerButtonDanger[o]}
onPress={ () => { store.dispatch(answerButtonHighlight(o))}}>
       <Text style={styles.buttonText}>
         {this.props.answerButtonLabels[o]}
       </Text>
     </Button>
   ..
```

Pertama adalah View dengan outerContainer yang diterapkan, seperti biasa. Di dalamnya terdapat View untuk pertanyaan. Perhatikan bahwa RNText digunakan. Seperti yang disebutkan sebelumnya, ini hanyalah komponen React Native Text, tetapi karena komponen ini juga akan menggunakan komponen NativeBase Text, akan ada konflik nama jika salah satunya tidak diberi alias, jadi saya memilih untuk memberi alias pada versi React Native tanpa

alasan tertentu. Pertanyaannya, tentu saja, disimpan dalam prop question, yang dipetakan dari penyimpanan status.

Setelah View pertanyaan, terdapat View yang berisi enam tombol jawaban, dan saya telah memotong lima lainnya, karena keduanya identik dengan yang pertama ini, kecuali indeks array yang digunakan (hanya bertambah satu dengan setiap tombol) dan argumen yang diteruskan ke fungsi tindakan answerButtonHighlight(). Setiap tombol, yang merupakan Tombol NativeBase, bukan Tombol React Native (karena versi NativeBase menyediakan beberapa fitur tambahan yang berguna), menggunakan prop lengkap, untuk memastikan bahwa tombol membentang di induknya, yang, dalam kasus ini, membuatnya menyebar di layar.

Setiap tombol memiliki gaya answerButton yang diterapkan, yang memberi beberapa ruang di atas setiap tombol, agar tetap terpisah dengan baik. Sekarang menjadi menarik. Setiap tombol memiliki properti utama dan properti danger. Properti tersebut menentukan tampilan tombol. Jika primary bernilai true, tombol akan memiliki latar belakang biru. Jika danger bernilai true, tombol akan memiliki latar belakang merah. Perhatikan bagaimana nilai untuk properti tersebut berasal dari properti answerButtonPrimary dan answerButtonDanger, yang Anda lihat adalah array dalam status yang dipetakan oleh properti tersebut. Alasan hal ini dilakukan adalah saat pengguna mengetuk salah satu tombol, kita ingin tombol tersebut berwarna merah, sementara yang lainnya berwarna biru.

Artinya, kita ingin semua elemen dalam array answerButtonPrimary bernilai true, dan semua kecuali yang terkait dengan tombol yang diketuk bernilai false, dalam array answerButtonDanger. Karena properti ini terikat dengan status, artinya saat kita mengubah nilai dalam status, tampilan tombol di layar akan berubah. Anda dapat melihat bahwa prop onPress mengirimkan pesan yang dapat Anda asumsikan mengubah array sesuai dengan logika tersebut, dan Anda akan melihatnya nanti di bagian CoreCode. Komponen Text di dalam komponen Button adalah komponen NativeBase Text kali ini, dan alasan komponen tersebut digunakan daripada komponen React Native Text adalah karena dokumentasi NativeBase mengatakan bahwa kita harus selalu menggunakan komponen ini dengan komponen Button. (Tidak ada alasan yang diberikan untuk ini dalam dokumentasi, tetapi siapa saya untuk membantahnya? Setelah enam tombol tersebut terdapat tombol Submit Answer.

```
<View style={styles.submitButtonContainer}>
   <Button
     block
     success
     onPress={
       () => {
        if (store.getState().question.selectedAnswer === -1) {
          Alert.alert("D'oh!", "Please select an answer",
            [ { text: "OK"} ], { cancelable: false}
          );
       } else {
          CoreCode.io.emit("submitAnswer", {
            playerID: store.getState().playerInfo.id,
```

```
          answer: store.getState().question.answerButtonLabels[
             store.getState().question.selectedAnswer
          ]
    });
           }
        }
      }
     >
]
          <Text style={styles.buttonText}>Submit Answer</Text>
        </Button>
      </View>
    </View>
  );
}
}
```

Untuk yang ini, saya menggunakan properti block, untuk memberi tombol tampilan yang sedikit berbeda dari tombol answer, dan properti success, untuk membuat tombol berwarna hijau. Handler onPress pertama-tama memeriksa apakah atribut status question.selectedAnswer, yang akan ditetapkan sebagai bagian dari panggilan dispatch() di handler onPress tombol answer, memiliki nilai selain -1, yang merupakan nilai "tidak ada jawaban yang dipilih". Jika demikian, pesan Alert akan ditampilkan. Jika tidak, pesan submitAnswer dipancarkan ke server, untuk menunjukkan jawaban yang dipilih.

playerID dikirim ke server, bersama dengan jawaban yang dipilih, dan perhatikan bahwa jawaban tekstual itu sendiri yang dikirim, bukan nomor yang disimpan dalam selectedAnswer, jadi kita mendapatkannya melalui array question.answerButtonLabels dalam status. Komponen ini juga terhubung, jadi ia memiliki fungsi mapStateToProps() dan menggunakan fungsi connect(). Pemetaan properti adalah answerButtonPrimary yang dipetakan ke question.answerButtonPrimary, answerButtonDanger dipetakan ke question. answerButtonDanger, answerButtonLabels dipetakan ke question.answerButtonLabel, dan question dipetakan ke question.currentQuestion.

Menemukan Inti Masalah: CoreCode.js

Mari kita bahas arsitektur sejenak. Ketika saya mulai menulis aplikasi ini, dengan cepat menjadi jelas bahwa akan ada beberapa kode yang tidak boleh (atau mungkin bahkan tidak bisa) ada di dalam masing-masing komponen React Native. Saya harus membuat pilihan: apakah saya meletakkan semua kode di beberapa tempat umum dan membiarkan komponen memanggilnya, atau apakah saya menyebarkannya sedikit, dengan meletakkan beberapa di lokasi pusat dan beberapa di komponen? Misalnya, ketika saya memiliki komponen Button, komponen tersebut akan memiliki pengendali acara onPress.

(Jika tidak, itu bukan Button, bukan?) Di mana kode yang dieksekusi sebagai respons terhadap acara tersebut ditempatkan? Sejauh ini, Anda telah melihatnya langsung masuk ke komponen, "dimasukkan" dengan prop onPress, dengan kata lain. Beberapa pengembang mengatakan bahwa ini adalah tempat yang tepat untuk itu, karena React Native, seperti React,

berbasis komponen, dan komponen seharusnya menjadi hal yang berdiri sendiri dan berdiri sendiri. Pengembang lain mengatakan bahwa Anda harus mencoba memisahkan kode presentasi dari kode tindakan Anda, jadi, sebagai gantinya, Anda harus memiliki beberapa objek yang memiliki metode, dan kemudian prop onPress merujuk ke metode tersebut.

Ada jalan tengah yang mengatakan bahwa karena komponen adalah modul JavaScript, Anda harus meletakkan fungsi "telanjang" di file sumber yang sama, tetapi tidak sejalan dengan markup komponen, dan merujuknya di markup komponen. Intinya adalah bahwa semua pendekatan ini berfungsi dan memiliki kelebihan dan kekurangan. Lokasi yang sentral memberi Anda pemisahan yang jelas yang, sebelum dunia komponen, merupakan sesuatu yang harus diperjuangkan. Sekarang kita berada di dunia komponen, mungkin itu bukan lagi tujuan yang valid. Bukankah arsitektur itu menyenangkan?

Namun, pada akhirnya, cara saya mengatur kode didasarkan pada pemahaman bahwa, secara konseptual, ada dua jenis kode: kode yang merespons interaksi pengguna dan kode yang tidak. Ini sedikit penyederhanaan yang berlebihan, dan beberapa kasus akan melewati batas, tentu saja, tetapi sebagai prinsip desain yang menyeluruh, ini berhasil. Dengan prinsip itu, keputusan menjadi relatif mudah. Setiap kode yang mewakili interaksi pengguna masuk ke kode komponen (dan, sebagai aturan, saya cenderung lebih suka kode tersebut sejalan dengan markup komponen, kecuali saya tahu kode tersebut akan dibagikan, dalam hal ini, saya akan menjadikannya fungsi yang berdiri sendiri dalam modul).

Setiap kode yang tidak dapat saya klasifikasikan sebagai interaksi pengguna akan masuk ke objek pusat yang saya sebut CoreCode. Objek CoreCode ini hanyalah objek JavaScript, dan karena saya tahu bahwa akan selalu hanya ada satu contoh kanoniknya, saya menggunakan pendekatan singleton, daripada menjadikannya kelas. Ini dimulai, dengan cukup polos, dengan beberapa properti.

```
const CoreCode = {

    serverIP: "127.0.0.1",
    io: null,
    mainNavigator: null,
```

serverIP adalah alamat IP server. Anda perlu memperbaruinya untuk lingkungan Anda dengan tepat (dan alamat loopback 127.0.0.1 tidak akan berfungsi, kecuali Anda entah bagaimana membuat aplikasi React Native berjalan pada mesin yang sama secara asli, yang, jika Anda mengetahui cara melakukannya, beri tahu saya karena itu akan luar biasa). Properti io adalah referensi ke objek klien socket.io, dan itu akan dibuat nanti. mainNavigator adalah referensi ke navigator utama untuk aplikasi, dan Anda melihat referensi ini terisi dalam konstruktor sublayar beranda layar game.

Fungsi pertama yang kita temukan dalam objek ini adalah startup(), yang, Anda ingat, adalah fungsi yang direferensikan oleh prop onPress dari Button pada NamePromptModal. (Jadi, berikut ini salah satu contoh di mana saya, dalam arti tertentu, melanggar aturan organisasi saya sendiri, tetapi hanya karena ini adalah satu kasus di mana kode benar-benar

melewati batas.)

```
startup: () => {
  if (!store.getState().modals.isAdmin &&
    (store.getState().playerInfo.name == null ||
    store.getState().playerInfo.name.trim()
    store.getState().playerInfo.name.length ===
) {
    return;
}
store.dispatch(showHideModal("namePrompt", false));
CoreCode.io = io('http://${CoreCode.serverIP}');
if (store.getState().modals.isAdmin) {
CoreCode.io.on("connected", function() { console.log("ADMIN CONNECTED");});
CoreCode.io.on("adminMessage", CoreCode.adminMessage);
store.dispatch(showHideModal("admin", true));
} else {
CoreCode.io.on("connected", CoreCode.connected);
CoreCode.io.on("validatePlayer", CoreCode.validatePlayer);
CoreCode.io.on("newGame", CoreCode.newGame);
CoreCode.io.on("nextQuestion", CoreCode.nextQuestion);
CoreCode.io.on("answerOutcome", CoreCode.answerOutcome);
CoreCode.io.on("endGame", CoreCode.endGame);

}
},
```

Tugas di sini adalah, pertama, memastikan, jika pengguna tidak menunjukkan bahwa ia adalah admin, bahwa mereka memasukkan nama dan panjangnya lebih dari satu karakter. Ed adalah nama asli terpendek yang dapat saya pikirkan, jadi, jika ada yang melihat ini kebetulan memiliki nama yang panjangnya hanya satu huruf, saya minta maaf, tetapi saya harus melihat akta kelahiran Anda sebelum saya mempercayainya. Setelah konfirmasi itu dilakukan, NamePromptModal disembunyikan, dengan mengirimkan pesan yang dikembalikan oleh fungsi tindakan showHideModal(). Setelah itu, saatnya membuat objek socket.io yang dirujuk oleh properti io.

Itu memerlukan alamat serverIP dari sebelumnya, dan perhatikan bahwa Anda juga harus menentukan protokol, menjadikannya URL yang tepat, atau koneksi tidak akan berfungsi. Setelah objek itu dibuat, saatnya untuk menghubungkan fungsi pengendali pesan, tetapi yang mana yang dihubungkan tergantung pada apakah pengguna adalah admin atau bukan. Jika dia adalah admin, maka satu-satunya yang perlu diperhatikan adalah pesan yang terhubung dan adminMessage. Bagi pemain, kode harus merespons pesan connected, validatePlayer, newGame, nextQuestion, answerOutcome, dan endGame. Semua fungsi pengendali hanyalah metode dari objek CoreCode, jadi bagian selanjutnya dari bagian ini akan membahas fungsi-fungsi tersebut, dimulai dengan connected.

```
connected: function(inData) {
   CoreCode.io.emit("validatePlayer", {
```

```
      playerName: store.getState().playerInfo.name
});
},
```

Setelah klien dan server terhubung, server memancarkan pesan yang terhubung, dan pada saat itu klien memancarkan pesan validatePlayer, yang meneruskan nama pemain ke server. Server kemudian melakukan validasi, seperti yang Anda lihat di bab sebelumnya, dan memancarkan pesan validatePlayer ke klien, yang ditangani seperti ini:

```
validatePlayer: function(inData) {
   store.dispatch(setPlayerID(inData.playerID));
   if (inData.inProgress) {
     inData.gameData.asked = inData.asked;
     store.dispatch(setGameData(inData.gameData));
     store.dispatch(updateLeadboard(inData.leaderboard));
     CoreCode.mainNavigator.navigate("GameLeaderboardScreen");

 }
},
```

Pertama, playerID yang dikembalikan oleh server disimpan dalam status, lalu pemeriksaan dilakukan, apakah permainan sedang berlangsung. Jika tidak ada permainan yang sedang berlangsung, pengguna saat ini akan melihat permainan ➤ layar beranda, menunggu permainan baru dimulai. (Ingat: Itu adalah rute default untuk SwitchNavigator, itulah sebabnya rute tersebut terlihat pada titik ini.) Jika permainan sedang berlangsung, jumlah pertanyaan yang diajukan ditransfer ke objek gameData pada objek inData, karena kita akan membutuhkannya di sana untuk ditampilkan di layar info, tetapi tidak akan ada di sana pada titik ini, dan data permainan dikirim ke penyimpanan, seperti halnya data papan peringkat saat ini yang akan dikirim server.

Terakhir, Anda dapat melihat mengapa mendapatkan referensi ke navigator utama diperlukan: sehingga panggilan ke navigation() dapat dilakukan, yang mengirim pengguna ke permainan ➤ layar papan peringkat. Pada titik ini, aplikasi berada dalam status awal apa pun yang diperlukan. Fungsi penanganan pesan lainnya menangani pesan-pesan lain yang dapat datang kapan saja, dimulai dengan penanganan pesan newGame.

```
newGame: function(inData) {
  store.dispatch(showHideModal("endGame", false));
  inData.gameData.asked = inData.asked;
  store.dispatch(setGameData(inData.gameData));
  store.dispatch(updateLeadboard(inData.leaderboard));
  CoreCode.mainNavigator.navigate("GameLeaderboardScreen");
},
```

Saat permainan baru dimulai, kita tahu bahwa pengguna berada di layar beranda permainan

atau layar papan peringkat permainan, dan jika pengguna berada di layar papan peringkat, modal akhir permainan mungkin akan ditampilkan. Jadi, pertama-tama kita mengirimkan pembaruan status untuk menyembunyikan modal tersebut. Selanjutnya, jumlah pertanyaan akan ditransfer, seperti yang dilakukan di validatePlayer (meskipun kita tahu dalam kasus ini akan menjadi nol).

Kemudian kita harus menyetel data permainan dan data papan peringkat awal di penyimpanan, jadi dua panggilan dispatch() akan menanganinya. Terakhir, layar papan peringkat permainan harus ditampilkan. Panggilan ke navigation() pada referensi mainNavigator dilakukan untuk menyelesaikannya. Pengguna sekarang berada di tempat yang seharusnya, dengan status yang bagus dan baru, menunggu pertanyaan. Saat admin menunjukkan bahwa sudah waktunya untuk pertanyaan baru, server memancarkan pesan nextQuestion, yang ditangani oleh metode ini:

```
nextQuestion: function(inData) {
  store.dispatch(answerButtonHighlight(-1));
  store.dispatch(setQuestion(inData.question));

  for (let i = 0; i < 6; i++) {
    store.dispatch(updateAnswerButtonLabel(i, inData.answers[i]));

  store.dispatch(resetAllButtons());
  CoreCode.mainNavigator.navigate("GameQuestionScreen");
},
```

Saat pesan ini diterima, aplikasi harus menampilkan pertanyaan dan enam kemungkinan jawaban, dan juga harus memastikan bahwa status tombol telah ditetapkan dengan benar (artinya tidak ada yang disorot). Kemudian layar permainan ➤ pertanyaan harus ditampilkan. Jadi, kita mulai dengan mengirimkan dua pembaruan ke status, yang pertama menggunakan fungsi tindakan answerButtonHighlight() dan memberikannya nilai -1. Seperti yang Anda lihat sebelumnya, itu adalah nilai khusus yang memberi tahu pereduksi bahwa tidak ada tombol yang harus disorot, jadi ia menangani pengaturan semua elemen array answerButtonPrimary ke true dan semua yang ada di array answerButtonDanger ke false.

Kemudian label untuk setiap tombol diperbarui. Ini memerlukan pengiriman tindakan updateAnswerButtonLabel(), mengirimkannya indeks tombol yang akan diperbarui dan label yang akan diberikan. Setelah itu, tindakan resetAllButtons() dikirim. Ini akan, jika Anda memperhatikan, tampak sangat aneh, karena jika Anda membandingkan apa yang dilakukan pereduksi yang terkait dengan tindakan ini, Anda akan melihat bahwa pada dasarnya semuanya sama. Masalah yang saya temukan adalah label pada tombol tidak akan diperbarui di layar, kecuali saya memperbarui jenis semua tombol untuk kedua kalinya.

Sejujurnya, saya tidak yakin mengapa demikian, dan mungkin saja ini hanya bug di NativeBase. Jawaban sederhananya adalah melakukannya dua kali, tetapi daripada menggunakan answerButtonHighlight() lagi, saya pikir lebih masuk akal untuk membuat tindakan terpisah untuknya. Dengan begitu, jika pembaruan menyebabkannya berfungsi

seperti yang saya harapkan sejak awal, saya dapat menghapus tindakan pengiriman yang tidak masuk akal ini dan huruf besar/kecil terkait di reducer dan tidak menyentuh apa pun lagi. Bagaimanapun, pesan berikutnya yang harus ditangani kode tersebut adalah pesan answerOutcome, yang dipancarkan server setelah klien mengirimkan jawabannya ke pertanyaan saat ini.

```
answerOutcome: function(inData) {
   let msg = "Sorry! That's not correct:(";
   let type = "danger";
   if (inData.correct) {
     msg = "Hooray! You got it right:)";
     type = "success";
}

inData.gameData.asked = inData.asked;
store.dispatch(setGameData(inData.gameData));
store.dispatch(updateLeadboard(inData.leaderboard));
CoreCode.mainNavigator.navigate("GameLeaderboardScreen");
Toast.show({ text: msg, buttonText: "Ok", type: type, duration: 3000
});
    Vibration.vibrate(1000);
},
```

Dimulai dengan asumsi bahwa pengguna salah, karena kode yang baik cenderung pesimis. Namun, jika inData.correct kembali dari server dengan nilai true, pesan tersebut diubah, begitu pula tipenya. Seperti semua pesan lainnya, asked disalin ke gameData, dan data game (ditampilkan di layar info) dan data papan peringkat (ditampilkan di layar game ➤ papan peringkat) dikirim ke penyimpanan. Kemudian layar game ➤ papan peringkat dinavigasi.

Selanjutnya, komponen Toast dari NativeBase digunakan, untuk menampilkan pesan toast dengan pesan yang benar atau salah dan menggunakan tipe untuk membuatnya menjadi hijau (benar) atau merah (salah). Terakhir, React Native Vibration API digunakan untuk menggetarkan perangkat. Kita tinggal memiliki dua metode lagi untuk diperiksa, dan hanya satu lagi yang terkait dengan pemain, dan itu adalah yang berikutnya, yang terkait dengan pesan endGame.

```
endGame: function(inData) {
  inData.gameData.asked = inData.asked;
  store.dispatch(setGameData(inData.gameData));
  store.dispatch(updateLeadboard(inData.leaderboard));
  CoreCode.mainNavigator.navigate("GameLeaderboardScreen");
  if (inData.leaderboard[o].playerID ===    store.getState().playerInfo.id)
{
  store.dispatch(setEndGameMessage("Congratulations! You were  the
  winner!"));
  store.dispatch(showHideModal("endGame", true));
} else {
   let place = "";
```

```
    let index = inData.leaderboard.findIndex((inPlayer) =>
      inPlayer.playerID === CoreCode.playerID);
    index++;
    const j = index % 10;
    const k = index % 100;
    if (j === 1 && k !== 11) {
        place = '${index}st';
    } else if (j === 2 && k !== 12) {
        place = `${index}nd';
    } else if (j === 3 && k !== 13) {
        place = '${index}rd';
    } else {
        place = `${index}th';
      store.dispatch(setEndGameMessage(
       Sorry, you didn't win. You finished in ${place} place. )
       );
     store.dispatch(showHideModal("endGame", true));
}
},
```

Saat permainan berakhir, kami memiliki dua tugas utama: menampilkan data permainan akhir dan papan peringkat, serta memberi tahu pengguna jika ia menang (dan jika tidak, di posisi mana ia finis). Empat baris pertama sama seperti yang telah Anda lihat di pengendali sebelumnya, yang berpuncak pada pengguna yang diarahkan ke layar permainan ➤ papan peringkat dengan data papan peringkat akhir yang ditampilkan. Setelah itu, kami melihat apakah pemain pertama dalam data papan peringkat adalah pemain saat ini, berdasarkan ID. Jika demikian, kami menampilkan EndGameModal, dengan pesan ucapan selamat yang sesuai di dalamnya.

Jika seorang pemain tidak menang, maka kami harus menunjukkan kepadanya pesan yang mengatakan di posisi mana ia finis. Indeksnya dalam array adalah posisi finisnya, dan berdasarkan itu, kami ingin melampirkan sufiks yang sesuai, baik "st," "nd," "rd," atau "th," mana pun yang sesuai untuk nilainya. Saat itu ditentukan, EndGameModal ditampilkan, tetapi dengan pesan "maaf" kali ini. Dengan semua penanganan pesan terkait pemain diperiksa, hanya ada satu metode pada objek CoreCode, yaitu untuk menangani pesan adminMessage.

```
adminMessage: function(inData) {
   store.dispatch(setCurrentStatus(inData.msg));
}
```

Ya, itu dia! Pesan yang dikembalikan oleh server dikirim ke toko, tempat pesan tersebut digunakan sebagai nilai komponen Teks di bagian bawah AdminModal. Dengan itu, kita telah menyelesaikan pembedahan aplikasi kedua dari tiga aplikasi. Saya harap Anda belajar banyak dari aplikasi ini dan bersenang-senang melakukannya.

16.8 KESIMPULAN

Kesimpulan dari bab ini adalah kita telah memulai pembangunan aplikasi RNTrivia, dimulai dengan pengembangan sisi server menggunakan Node, Web Sockets, dan socket.io. Di bab selanjutnya, kita akan melanjutkan dengan membangun sisi klien menggunakan React Native, menghubungkannya ke server, dan memastikan aplikasi berjalan sebagaimana mestinya. Dalam bab ini juga mengenalkan berbagai konsep baru, termasuk peralihan platform, komponen dari React Native, NativeBase, dan React Navigation, serta manajemen status dengan Redux. Kita juga akan belajar berkomunikasi dengan server secara real-time. Meskipun aplikasi ini tidak sepenuhnya dianggap sebagai permainan, React Native sangat cocok untuk membangun permainan yang sesungguhnya, yang akan kita bahas di bab-bab berikutnya. Proses ini akan memberikan pemahaman yang lebih dalam tentang bagaimana membangun aplikasi yang lebih kompleks dan interaktif.

BAB 17
MEMULAI APLIKASI REACT NATIVE DENGAN PEMILIH RESTORAN

17.1 MEMBANGUN APLIKASI PEMILIH RESTORAN DENGAN REACT NATIVE

Apa yang Kita Bangun?

Aplikasi Restaurant Chooser berupaya memecahkan masalah dunia nyata yang saya duga pernah dialami banyak orang: kontes yang terjadi saat Anda mengajukan pertanyaan sederhana kepada sekelompok orang, Di mana Anda ingin makan malam? Ini adalah masalah yang sering terjadi pada keluarga saya, karena seorang anak berkata, "Saya tidak suka restoran Meksiko!" dan yang lain mengakui, "Saya tidak tahu apa yang saya inginkan!" dan kemudian istri saya berkata, "Saya tidak peduli," yang tidak membantu untuk mencapai keputusan. Jadi, saya menulis aplikasi ini untuk memecahkan masalah tersebut.

Dalam istilah yang paling sederhana, Anda membuat orang dan restoran di aplikasi, lalu Anda membiarkan aplikasi memilih tempat makan untuk Anda, dengan beberapa peringatan. Pertama, Anda memilih orang yang akan makan, lalu kelompok yang ingin Anda beri makan dapat melakukan pra-penyaringan, jika diinginkan.

Jadi, misalnya, jika semua orang secara ajaib setuju bahwa mereka menginginkan makanan Cina, itu bagus, karena itu mempersempit pilihan (kecuali jika Anda hanya punya satu restoran Cina favorit, dalam hal ini, masih ada keputusan yang harus diambil). Mungkin Anda hanya ingin mengunjungi restoran ternama, jadi Anda lebih suka memfilter terlebih dahulu berdasarkan peringkat bintang. Setelah itu, dan aplikasi membuat pilihan, setiap orang dalam grup mendapat satu hak veto. Jadi, jika mayoritas memilih restoran Yunani, tetapi Tuan "Saya tidak suka makanan Yunani" berkeberatan, ia dapat menggunakan hak vetonya, dan aplikasi akan memilih restoran lain. Setelah restoran dipilih dan tidak ada yang memveto—atau ketika semua orang telah memveto sekali restoran itu adalah hasil akhirnya, suka atau tidak.

Ini bukan aplikasi yang rumit, tetapi dapat bermanfaat, dan terlepas dari kegunaannya, aplikasi ini akan berfungsi sebagai aplikasi yang sangat baik dan tidak terlalu rumit untuk mempelajari React Native. Jadi, seperti apa tampilannya? Semuanya dimulai dengan layar pembuka saat aplikasi dimulai, seperti yang ditunjukkan pada Gambar 17.1 (dengan versi iOS di sebelah kiri dan versi Android di sebelah kanan).

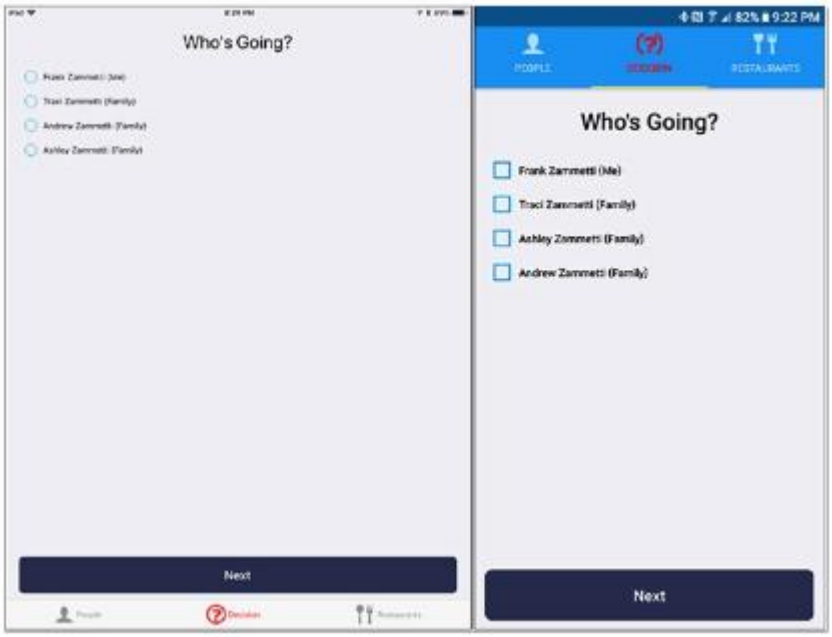
Gambar 17.1 Layar pembuka (iklan murah: dengan gambar masakan istri saya)

Setelah aplikasi dimuat, pengguna awalnya akan menemukan diri mereka di layar Saatnya Mengambil Keputusan, seperti yang ditunjukkan pada Gambar 17.2.

Gambar 17.2. Di mana semuanya dimulai

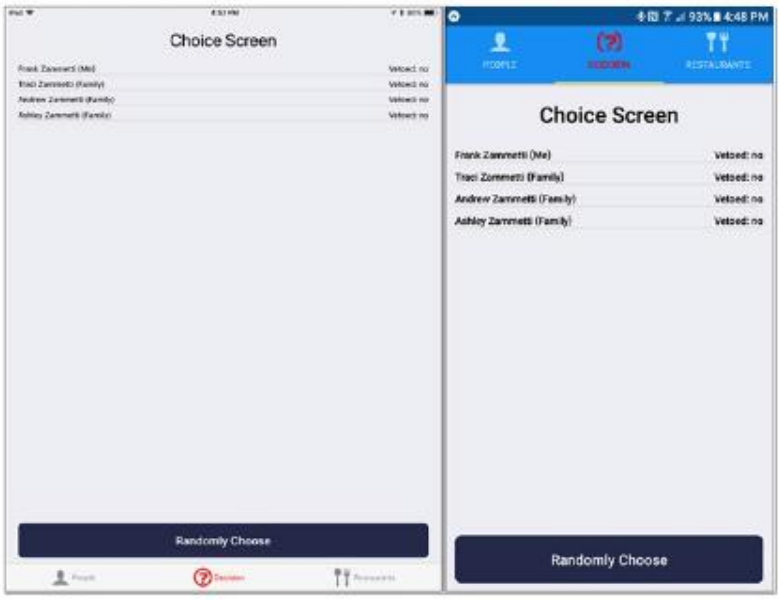
Anda akan segera melihat bahwa ada antarmuka bertab untuk melakukan ini, dan itu karena pada dasarnya ada tiga layar berbeda pada tingkat tinggi: layar Orang, yang merupakan tempat Anda membuat orang yang akan memenuhi syarat untuk terlibat dalam suatu keputusan; layar Keputusan, yang sebenarnya merupakan kumpulan sejumlah sublayar, seperti yang akan Anda lihat nanti; dan layar Restoran, yang seperti layar Orang tetapi untuk mengelola daftar restoran Anda.

Dengan asumsi orang dan restoran telah dibuat dan Anda siap untuk membuat keputusan, Anda hanya perlu mengetuk grafik besar, dan Anda akan berakhir di layar Siapa yang Akan Datang, seperti yang ditunjukkan pada Gambar 17.3.



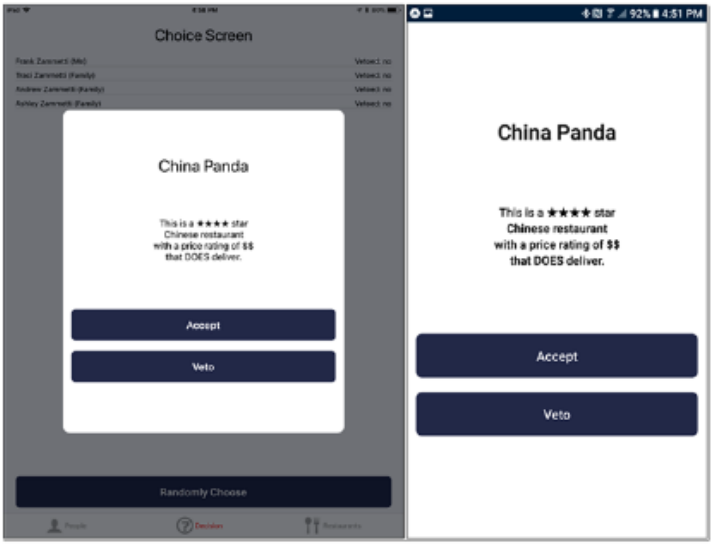
Gambar 17.3. Layar Siapa yang Akan Datang

Layar Siapa yang Akan Datang menampilkan daftar orang yang dapat Anda pilih. Setelah Anda memilih satu orang atau lebih, Anda menekan tombol Berikutnya, dan Anda akan diarahkan ke Layar Pilihan, yang ditunjukkan pada Gambar 17.4.



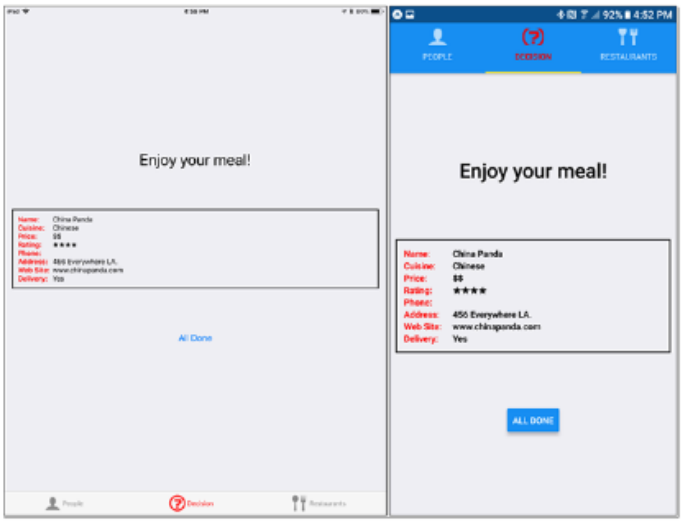
Gambar 17.4. Layar Pilihan

Tombol Pilih Acak yang besar di bagian bawah adalah tombol yang Anda ketuk untuk membuat pilihan, yang menghasilkan layar Keputusan yang ditunjukkan pada Gambar 17.5.



Gambar 17.5. Layar Keputusan

Presentasinya sedikit berbeda antara iOS dan Android, tetapi dalam kedua kasus tersebut, hal yang disampaikan sama: pilihan yang dibuat oleh aplikasi. Pada titik ini, Anda dapat mengetuk tombol Terima, jika semua orang setuju dengan pilihan ini, atau mengetuk tombol Tolak. Melakukan hal tersebut kemudian akan menampilkan pop-up berbeda yang mencantumkan orang-orang yang belum memveto, dan Anda dapat mengetuk satu untuk mendaftarkan veto. Namun, dengan asumsi Anda semua setuju, atau jika tidak ada yang memiliki hak veto yang tersisa, Anda akan menemukan diri Anda di layar yang ditunjukkan pada Gambar 17.6.



Gambar 17.6. Akhirnya, argumen yang berlarut-larut dapat dihindari!

Di sini, Anda telah memberikan informasi tentang pilihan terakhir, dan Anda siap untuk

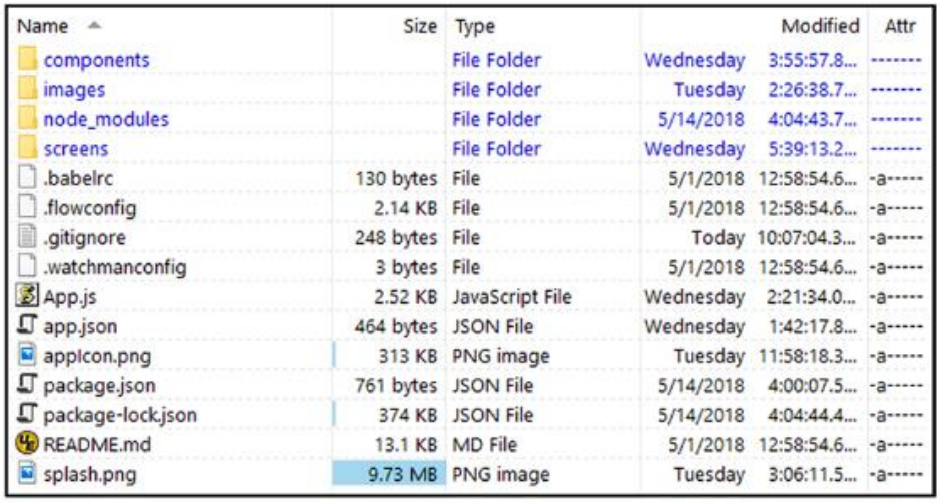
makan. Oke, jadi sekarang setelah kita tahu apa yang sedang kita bangun (sebagian besar—ada beberapa layar yang tidak ditampilkan di sini, tetapi saya akan membahasnya sebentar lagi), sekarang mari kita bahas bagaimana kita akan melakukannya, pada tingkat yang sangat tinggi, dalam hal bagaimana menyusun aplikasi.

17.2 STRUKTUR APLIKASI

Sebelum kita melangkah lebih jauh, kita harus berbicara sedikit tentang cara menyusun aplikasi React Native. Ini bisa menjadi pembicaraan singkat: tidak ada aturan.

Faktanya, React Native tidak benar-benar memaksakan struktur tertentu pada Anda, begitu pula Expo di atas React Native. Oh, tentu saja, ada beberapa hal yang harus Anda lakukan dan miliki. Pertama, Anda akan memerlukan file app.json untuk menjelaskan aplikasi Anda ke Expo, dan Anda akan memiliki file App.js yang merupakan titik masuk utama Anda ke kode aplikasi Anda. Seperti yang dibahas dalam Bab 1, saat Anda menjalankan create-react-native-app, Anda akan mendapatkan file-file ini, ditambah beberapa file lainnya, dan Anda akan mendapatkan direktori node-modules, tentu saja, tetapi di luar itu, Anda bebas menyusun kode sesuai keinginan Anda.

Jika Anda ingin meletakkan semua kode Anda di App.js, Anda tentu saja bisa, tetapi itu biasanya bukan pendekatan yang bagus, kecuali jika itu adalah aplikasi yang sangat sepele. Tidak, lebih sering daripada tidak, Anda ingin menambahkan beberapa direktori, untuk menjaga semuanya tetap teratur, dan itulah yang telah saya lakukan untuk Restaurant Chooser, seperti yang ditunjukkan Gambar 17.7.



Name	Size	Type	Modified	Attr
components		File Folder	Wednesday 3:55:57.8...	-------
images		File Folder	Tuesday 2:26:38.7...	-------
node_modules		File Folder	5/14/2018 4:04:43.7...	-------
screens		File Folder	Wednesday 5:39:13.2...	-------
.babelrc	130 bytes	File	5/1/2018 12:58:54.6...	-a-----
.flowconfig	2.14 KB	File	5/1/2018 12:58:54.6...	-a-----
.gitignore	248 bytes	File	Today 10:07:04.3...	-a-----
.watchmanconfig	3 bytes	File	5/1/2018 12:58:54.6...	-a-----
App.js	2.52 KB	JavaScript File	Wednesday 2:21:34.0...	-a-----
app.json	464 bytes	JSON File	Wednesday 1:42:17.8...	-a-----
appIcon.png	313 KB	PNG image	Tuesday 11:58:18.3...	-a-----
package.json	761 bytes	JSON File	5/14/2018 4:00:07.5...	-a-----
package-lock.json	374 KB	JSON File	5/14/2018 4:04:44.4...	-a-----
README.md	13.1 KB	MD File	5/1/2018 12:58:54.6...	-a-----
splash.png	9.73 MB	PNG image	Tuesday 3:06:11.5...	-a-----

Gambar 17.7. Struktur direktori aplikasi Restaurant Chooser

Selain app.json dan App.js, Anda juga akan menemukan .babelrc, .flowconfig.gitignore, dan .watchmanconfig, yang semuanya dapat Anda abaikan, karena merupakan berkas yang terkait dengan alat yang digunakan React Native dan Expo di balik layar. Berkas appIcon.png merupakan berkas tambahan yang akan dibahas nanti, yang juga berlaku untuk splash.png.

Berkas README.md dibuat untuk Anda tetapi tidak relevan dengan proyek ini (meskipun Anda bebas mengganti informasi berguna yang ada secara default dengan apa pun yang mungkin berguna bagi Anda). Berkas package.json dan package-lock.json juga merupakan berkas yang dapat Anda abaikan, karena ditujukan untuk penggunaan NPM.

Selain berkas, ada beberapa direktori. Direktori gambar Anda dapat menebaknya di mana saya meletakkan gambar, termasuk ikon pada tiga tab dan grafik untuk layar It's Decision Time. Anda sudah tahu apa itu node_modules, yang hanya menyisakan direktori komponen dan layar, yang keduanya merupakan direktori yang saya tambahkan dan berhubungan langsung dengan struktur aplikasi, seperti yang saya bahas di sini. Direktori komponen adalah tempat saya menyimpan dua komponen kustom yang akan Anda lihat nanti, dan ini merupakan praktik yang baik untuk dilakukan. Bekerja dengan React Native, seperti yang Anda ketahui sekarang, sepenuhnya didasarkan pada gagasan komponen.

Sering kali, Anda akan menggunakan komponen yang disediakan sendiri oleh React Native, terkadang, seperti yang akan Anda lihat nanti, komponen yang disediakan oleh pustaka pihak ketiga, dan terkadang Anda akan membuatnya sendiri. Yang terakhir adalah situasi di sini, dan di direktori tersebut, Anda akan menemukan dua file: CustomButton.js dan CustomTextInput.js. Seperti yang saya katakan, saya akan membahasnya nanti, tetapi saya berani bertaruh Anda dapat mengetahui apa saja file tersebut, berdasarkan namanya saja. Direktori layar mengisyaratkan bagaimana aplikasi tersebut pada dasarnya diatur: berdasarkan "layar". Sekarang, apa yang dimaksud dengan layar tidak selalu jelas, dan itulah yang terjadi dengan Restaurant Chooser. Ada tiga layar pada tingkat tinggi: layar People, layar Decision, dan layar Restaurants.

Namun, tangkapan layar sebelumnya menunjukkan bahwa ada lebih dari tiga layar unik di sini. Layar Decision memiliki beberapa yang mungkin Anda sebut sublayar. Namun, saya masih menganggap ini sebagai bagian dari layar Decision, jadi semuanya berada dalam file sumber yang sama di direktori screens, yang diberi nama DecisionScreen.js. Demikian pula, ada file PeopleScreen.js dan file RestaurantsScreen.js yang menampung kode untuk layar People dan Restaurants, masing-masing, dan yang nantinya akan Anda temukan berisi lebih dari satu sublayar, dalam arti tertentu. Anda mungkin menyebut sublayar sebagai tampilan, dan saya mempertimbangkan untuk melakukannya tetapi memutuskan untuk tidak melakukannya, untuk menghindari konflik dengan komponen View, tetapi pada akhirnya, yang terpenting adalah bahwa aplikasi tersebut dipecah menjadi sejumlah "layar," dan masing-masing berakhir dengan file sumbernya sendiri. Sublayar juga disertakan di dalamnya, meskipun seseorang dapat berargumen bahwa masing-masing sublayar juga harus disertakan dalam berkas sumbernya masing-masing sebagai komponen diskretnya sendiri, dan saya tidak akan membantahnya terlalu keras (meskipun itu akan memperumit beberapa hal, itulah sebabnya saya tidak melakukannya).

Namun, yang terpenting, ingat ketika saya mengatakan tidak ada aturan yang ditetapkan untuk struktur aplikasi React Native? Inilah yang saya maksud. Kita dapat memperdebatkan satu struktur vs. yang lain, tetapi Anda tidak diharuskan untuk memilih satu vs. yang lain, di luar beberapa persyaratan yang saya sebutkan sebelumnya. Apa pun yang

masuk akal bagi Anda dan apa pun yang mengatur kode Anda dengan cara yang logis bagi Anda, itu bagus. React Native tidak peduli, begitu pula Expo. Jika Anda memiliki berkas App.js untuk memulai, sisanya terserah Anda.

Catatan Seperti yang disebutkan sebelumnya, dalam buku ini, saya memilih untuk fokus pada pengembangan React Native menggunakan Expo, karena itu membuat sebagian besar hal lebih mudah dan menyediakan beberapa kemampuan tambahan (dan itu juga merupakan jalur yang secara default ditunjukkan oleh dokumen React Native sendiri). Namun, dalam membicarakan struktur aplikasi, perlu dicatat bahwa jika Anda tidak menggunakan Expo dan meminta React Native untuk membuatkan aplikasi untuk Anda, struktur yang Anda dapatkan akan sedikit berbeda. Jika Anda "mengeluarkan" aplikasi Anda dari Expo, yaitu, menyingkirkan Expo dari campuran dan menjadikannya aplikasi React Native "telanjang", Anda akan masuk ke dalam struktur itu juga. Saya tidak akan membahas struktur itu di sini, tetapi saya ingin menyebutkannya, karena jika Anda melihat contoh React Native di Web, Anda mungkin akan menemukan struktur itu, dan sekarang Anda akan memiliki sedikit pemahaman tentang mengapa demikian.

17.3 MEMULAI PROYEK

Memulai proyek ini semudah menjalankan yang berikut ini:

```
create-react-native-app RestaurantChoose
```

Itulah yang saya lakukan untuk memulai proyek ini, dan itu memberi kita kerangka aplikasi kita. Lebih tepatnya, itu memberi kita aplikasi yang berfungsi penuh yang dapat Anda jalankan segera. Tentu saja, itu belum menjadi Restaurant Chooser (meskipun itu akan sangat keren dan membuat kita semua kehilangan pekerjaan), jadi kita harus mulai menulis beberapa kode. Sementara file app.json dan App.js dibuat untuk kita, kita harus memulai pekerjaan kita dengan mengganti kontennya dengan kode yang lebih sesuai dengan kebutuhan kita. Jadi, mari kita mulai dengan app.json.

app.json

File app.json berkaitan dengan konfigurasi aplikasi Anda dan merupakan lokasi one stop shopping untuk memberi tahu React Native dan Expo tentang aplikasi Anda. File ini hanyalah file JSON biasa, seperti yang dapat Anda lihat di sini:

```
{
 }
 "expo": {
     "name": "Restaurant Chooser",
     "description": "Takes the pain out of making a decision!",
     "icon": "appIcon.png",
     "splash": { "image": "splash.png", "resizeMode": "cover" },
     "version": "1.0.0",
     "slug": "restaurantChooser",
     "sdkVersion": "23.0.0",
     "ios": { "bundleIdentifier": "com.etherient.restaurantChooser" },
```

```
    "android": { "package": "com.etherient.restaurantChooser" }
}
```

File ini memungkinkan banyak opsi konfigurasi di bawah kunci expo tingkat atas, tetapi hanya beberapa yang diperlukan: name, slug, dan sdkVersion.

Atribut name hanyalah nama aplikasi Anda. Atribut sdkVersion adalah versi Expo yang digunakan proyek, dan ini harus sesuai dengan versi yang ditentukan dalam package.json (yang dibuat create-react-native-app untuk Anda). Atribut slug adalah nama untuk aplikasi Anda yang akan digunakan untuk menerbitkan aplikasi Anda melalui Expo. URL yang akan dibuat Expo akan berbentuk expo.io/@<nama-pengguna-Anda>/<slug>, jadi nilai atribut slug harus sesuai untuk dimasukkan ke dalam URL (mis., tidak ada spasi yang diizinkan).

Semua atribut lain yang ditampilkan di sini bersifat opsional. Atribut description hanyalah deskripsi aplikasi Anda, yang bagus, jika Anda memiliki beberapa aplikasi yang diterbitkan, untuk memberi tahu orang lain (dan diri Anda sendiri) tentang aplikasi tersebut. Atribut ikon adalah ikon yang akan digunakan dalam peluncur aplikasi di iOS dan Android untuk mewakili aplikasi Anda. Ikon splash adalah gambar yang akan ditampilkan saat aplikasi Anda dimuat. Nilai atribut ini adalah nama file gambar, dan atribut resizeMode memberi tahu Expo cara mengubah ukuran gambar Anda. Nilai default, jika Anda tidak menentukan ini, adalah contains, yang akan mengakibatkan splash screen sepenuhnya termuat dalam layar fisik, artinya splash screen akan menutupi sebanyak mungkin layar, setelah diubah ukurannya dengan cara yang menjaga proporsinya tetap utuh, dan sisa layar akan tidak tertutup.

Nilai cover mengakibatkan splash screen diubah ukurannya cukup besar untuk memenuhi seluruh layar, sementara membiarkan bagian mana pun yang tidak "meluap" keluar layar. Atribut version adalah untuk penggunaan Anda sendiri dan merupakan versi aplikasi Anda. Atribut ios dan android masing-masing menentukan bundleIdentifier dan nama paket, yang menjadi penting saat Anda meminta Expo untuk membuatkan paket aplikasi nyata (yang akan dibahas di bab berikutnya). Nilai di sini harus dalam bentuk nama domain terbalik yang umum.

Ada beberapa atribut lain yang didukung di app.json dan Anda bahkan dapat membuatnya sendiri, jika Anda ingin menyertakan informasi dalam aplikasi Anda yang tidak boleh disematkan dalam kode tetapi atribut-atribut di sini adalah atribut yang harus Anda miliki dan yang kemungkinan besar Anda miliki, dan hanya itu yang kita perlukan untuk aplikasi ini.

17.4 BERALIH KE KODE

Setelah kerangka aplikasi dibuat dan konfigurasi yang diperlukan telah selesai, akhirnya tiba saatnya untuk membuat kode yang sebenarnya. Eksekusi dimulai di file App.js, jadi itu akan menjadi pemberhentian pertama kita.

App.js

Hal pertama yang perlu kita lakukan adalah mengimpor semua modul yang akan digunakan kode, baik yang berasal dari React Native itu sendiri, hal-hal yang disediakan Expo di atasnya, maupun kode aplikasi kita sendiri.

```
import React from "react";
import { Constants } from "expo";
import { Image, Platform } from "react-native";
import { TabNavigator } from "react-navigation";
import { PeopleScreen } from "./screens/PeopleScreen";
import { DecisionScreen } from "./screens/DecisionScreen";
import { RestaurantsScreen } from "./screens/RestaurantsScreen";
```

Anda akan selalu mengimpor React, karena React Native, tentu saja, dibangun di atasnya. Modul Constants adalah sesuatu yang disediakan Expo yang menyediakan beberapa informasi tambahan tentang perangkat yang tersedia untuk kode kita, seperti yang akan Anda lihat segera. Anda telah melihat komponen Image sebelumnya, serta modul Platform, meskipun itu hanya disinggung sebentar di Bab 2. Modul ini mirip dengan Constants tetapi disediakan oleh React Native sendiri, dan itu juga memberi kita informasi tambahan (dan beberapa metode, seperti yang Anda lihat di bab terakhir) tentang perangkat tempat kode berjalan. Tiga impor terakhir adalah tiga layar utama aplikasi. Tidak akan banyak yang bisa dilakukan tanpa itu, bukan?

Setelah impor, sedikit pembukaan logging dilakukan.

```
console.log("--------------------------------------------------");
console.log(`RestaurantChooser starting on ${Platform.OS}`);
```

Ini memberi kita beberapa informasi tentang OS yang kita jalankan di konsol dan menunjukkan bahwa aplikasi telah dimulai.

Setelah itu, ada baris kode sederhana.

```
const platformOS = Platform.OS.toLowerCase();
```

Ini adalah sesuatu yang bisa dibilang tidak perlu, tetapi menurut saya membuat hidup lebih mudah. Nanti, Anda akan melihat beberapa kode yang bercabang berdasarkan apakah aplikasi berjalan di iOS atau Android. Saya tidak ingin khawatir tentang huruf besar/kecil pada string tersebut, jadi dengan menggunakan huruf kecil, saya dapat menghindarinya. (Dengan begitu, jika React Native atau Expo memutuskan untuk mengubah huruf besar/kecil, kode tersebut tetap berfungsi. Akan menjadi cerita lain, jika mereka mengubah string itu sendiri, tentu saja.) Kode berikutnya mengharuskan saya untuk berbicara tentang sesuatu yang belum pernah saya bahas sebelumnya: navigasi melalui layar aplikasi. Jadi, saya akan membahasnya sekarang.

Navigasi React

Tidak umum untuk menemukan aplikasi seluler yang terdiri dari satu layar. Jadi, bagaimana Anda sebagai pengembang aplikasi menavigasi antarlayar? Nah, karena layar pada dasarnya hanyalah sebuah komponen, Anda dapat menulis beberapa kode yang menyembunyikan semua komponen layar, kecuali yang saat ini, dan terus melakukannya saat

pengguna berpindah melalui aplikasi Anda. Itu bisa berhasil, tetapi ada cukup banyak kode yang harus Anda tulis, dan terutama ketika Anda mulai memikirkan transisi animasi di antara keduanya, mendapatkan semua itu dengan benar bisa jadi sulit, terutama dalam cara lintas platform.

Dalam dunia React Native, dan dalam dunia pengembangan seluler secara umum, ada konsep navigator yang menangani semua detail tersebut untuk Anda. Secara sederhana, Anda memberi tahu navigator tentang layar Anda, lalu Anda memberi tahu layar mana yang ingin Anda tampilkan, dan navigator akan mengurus transisi dan semua hal yang terlibat dalam menyembunyikan layar saat ini dan menampilkan layar yang baru.

Ini bukan konsep yang rumit, meskipun detail implementasinya bisa rumit, tetapi React Native memiliki banyak kelebihan. Jika Anda mengetik "React Native navigation" di Google, Anda akan dihadapkan dengan hal-hal seperti React Navigation, NavigatorIOS, Ex-Navigation, Navigator, Native Navigation, React Native Router Flux, ExperimentalNavigation, React Router Native, dan React Native Navigation (dan mungkin masih banyak lagi). Mana yang Anda pilih? Bagaimana Anda membandingkannya untuk memutuskan? Nah, hal pertama yang perlu dipahami adalah bahwa semua navigator secara garis besar terbagi dalam dua kategori: navigator JavaScript, yang ditulis sepenuhnya dalam JavaScript, dan navigator asli, yang tentu saja ditulis dalam kode asli platform. Navigator JavaScript dapat memiliki kinerja yang lebih ruk, tetapi jika ditulis dengan baik, navigator tersebut hampir tidak dapat dibedakan dari navigator asli, dalam hal kinerja, sekaligus memiliki manfaat tambahan karena secara umum jauh lebih dapat disesuaikan. Pertimbangan lainnya adalah seberapa populer setiap opsi, karena akan selalu menyenangkan untuk memiliki banyak dukungan saat Anda mengalami masalah.

Untuk aplikasi Pemilih Restoran kami, keputusannya mudah, karena kami dapat langsung menghilangkan navigator asli, karena saat Anda menggunakan Expo, Anda tidak memiliki akses ke banyak bagian kode asli ekosistem React Native. (Itu sedikit berubah pada saat penulisan, tetapi masih benar, secara umum.) Setelah Anda menghilangkan opsi tersebut, pilihannya menjadi cukup sederhana, karena satu solusi berbasis JavaScript, untuk semua maksud dan tujuan, telah menang di pengadilan opini pengembang publik, dan itu adalah React Navigation.

React Navigation (https://reactnavigation.org) adalah pustaka yang berdiri sendiri, terpisah dari React Native (dan Expo). Untuk menambahkannya ke proyek ini, Anda harus menjalankan pm install --save react-navigation atau Anda harus menambahkannya secara eksplisit ke package.json lalu menjalankan npm install, untuk menginstalnya ke dalam proyek. Setelah selesai, kode Anda akan memiliki akses ke semua navigator yang disediakan React Navigation. Ada beberapa di antaranya, banyak di antaranya akan Anda lihat di proyek ini dan berikutnya, dan TabNavigator yang Anda lihat diimpor di sini adalah salah satunya.

Dengan React Navigation, navigator hanyalah komponen React Native, dan di antara yang sudah ada, Anda akan menemukan TabNavigator, yang menyediakan antarmuka tab, seperti yang terlihat pada tangkapan layar Restaurant Chooser; StackNavigator, yang sederhana, di mana beberapa layar "ditumpuk" dengan hanya satu yang ditampilkan pada satu

waktu; dan navigator tipe gambar untuk laci kontrol yang biasanya terlihat di aplikasi Android.

Anda juga dapat membuat navigator Anda sendiri atau menggunakan yang lain yang dibuat oleh komunitas. Manfaat dari semuanya adalah bahwa mereka akan menggunakan API umum, struktur konfigurasi umum, yang didasarkan pada default yang cerdas tetapi dapat disesuaikan yang seharusnya mengurangi jumlah kode yang harus Anda tulis. Berbicara tentang kode yang harus Anda tulis, mari kita lihat bagaimana Restaurant Chooser memanfaatkan React Navigation.

```
const tabs = TabNavigator({
   PeopleScreen : { screen : PeopleScreen,
       navigationOptions : { tabBarLabel : "People",
           tabBarIcon : ( { tintColor } ) => (
             <Image source={ require("./images/icon-people.png") }
                 style={{ width : 32, height : 32, tintColor : tintColor }}
              />
             )
        }
     },
```

Kita mulai dengan membuat komponen TabNavigator dan meneruskan objek konfigurasi yang menjelaskan masing-masing dari tiga layar tingkat atas aplikasi. Setiap objek memiliki atribut layar, yang merupakan komponen tingkat atas yang berisi layar (ini kebetulan adalah nama yang akan mewakili layar ini secara internal ke TabNavigator), serta atribut navigationOptions yang memberi tahu TabNavigator tentang tab layar tersebut, karena setiap layar, tentu saja, akan diwakili oleh tab. Ini termasuk tabBarLabel, yaitu teks yang ditampilkan pada tab, ditambah tabBarIcon, yang dibungkus dalam fungsi, sehingga kita dapat mengubah warna ikon saat ini (Anda akan melihat lebih lanjut tentang itu di bagian berikutnya). Setiap ikon yang ditentukan dibuat melalui komponen Gambar, dengan sumber yang ditentukan menggunakan pernyataan require() yang merujuk ke berkas gambar yang sesuai. Perhatikan bahwa atribut gaya merujuk ke tintColor yang diteruskan ke fungsi.

Itu menjadi penting saat Anda melihat objek terakhir yang diteruskan ke konstruktor TabNavigator.

```
{ initialRouteName : "DecisionScreen",animationEnabled : true,
swipeEnabled : true,
    backBehavior : "none", lazy : true,
    tabBarPosition : platformOS === "android" ? "top" : "bottom",
    tabBarOptions : { activeTintColor : "#ff0000",showIcon : true,
        style : { paddingTop : platformOS === "android" ? Constants.
        statusBarHeight : 0 }
    }
}
```

Alih-alih mendefinisikan layar individual, ini sekarang menjadi beberapa informasi yang mengonfigurasi TabNavigator itu sendiri. Pertama, initialRouteName "DecisionScreen" adalah

nama layar yang akan ditampilkan pertama kali (istilah rute sinonim dengan layar dalam konteks ini, serta navigator lain yang mungkin Anda temui, baik navigator React Navigation atau yang lainnya).

- ✓ Pengaturan animationEnabled: true menentukan apakah TabNavigator akan menganimasikan layar agar tidak terlihat atau hanya akan "muncul" di layar.
- ✓ Opsi swipeEnabled: true mengizinkan atau melarang pengguna menggeser ke kiri dan kanan untuk menavigasi antarlayar (jika tidak, mereka harus mengetuk ikon secara langsung, yang masih diaktifkan, meskipun penggeseran juga diaktifkan).
- ✓ Opsi backBehavior: "none" menentukan apa yang terjadi saat tombol kembali perangkat keras pada perangkat Android diketuk. Pengaturan none berarti tombol kembali tidak akan melakukan apa pun. Inilah yang saya inginkan dalam aplikasi ini; jika tidak, saat pengguna menavigasi antarlayar, tumpukan akan dibuat, dan saat pengguna menekan tombol kembali, mereka akan menavigasi melalui tumpukan itu, meskipun navigasinya tidak masuk akal.
- ✓ Atribut lazy: true memberi tahu TabNavigator untuk tidak membuat setiap layar hingga terlihat, yang membantu kinerja.
- ✓ Atribut tabBarPosition: memberi tahu TabNavigator apakah akan meletakkan tab di bagian atas atau bawah. Di sini, saya menggunakan variabel platformOS yang didefinisikan sebelumnya, untuk memastikan tab berada di platform yang sesuai: di atas untuk Android, di bawah untuk iOS. Terakhir, atribut tabBarOptions adalah objek yang menentukan tampilan tab. activeTintColor : "#ff0000" di dalamnya menentukan warna yang akan digunakan untuk ikon saat aktif, dalam kasus ini, merah (dan warna tersebut diteruskan ke fungsi yang membungkus komponen Image yang Anda lihat sebelumnya, jadi sekarang Anda tahu mengapa hal itu dilakukan dengan cara tersebut).
- ✓ Atribut showIcon : true harus disetel ke true agar ikon muncul.

Jika tidak, hanya teks yang akan ditampilkan. Atribut style menambahkan beberapa padding di bagian atas saat di Android, yang mencegah tab ditumpangkan di atas bilah status sistem (padding tersebut tidak diperlukan untuk iOS, oleh karena itu ada angka nol di ternary).

Konfigurasi tersebut adalah semua yang diperlukan agar TabNavigator berfungsi, dan masih banyak lagi opsi yang tersedia, terlalu banyak untuk dibahas di sini, tetapi ini memberi Anda gambaran dasar yang baik tentang apa yang dapat dilakukan TabNavigator. Ada satu baris kode terakhir dalam berkas sumber ini, setelah kode konfigurasi TabNavigator, yang sangat penting: export default tabs; Tanpa itu, React Native tidak akan tahu komponen apa yang harus dibuat saat aplikasi dimulai, dan Anda akan berakhir dengan layar kosong. Saya dengar itu tidak terlalu bagus untuk pengalaman pengguna, jadi kita mungkin harus mengekspor TabNavigator yang kita buat, melalui variabel tabs.

17.5 MENGGUNAKAN KOMPONEN KUSTOM

Jika Anda melihat kembali cuplikan layar Restaurant Chooser dari awal bab ini, Anda akan melihat bahwa tombol-tombol tersebut tampak sedikit berbeda dari tombol-tombol dalam proyek komponen. Dalam proyek tersebut, tombol-tombol tersebut adalah komponen

Tombol yang diberikan React Native kepada kita, yang merupakan komponen tombol khusus platform saat dirender. Mungkin saja untuk menggunakan gaya sederhana agar tombol-tombol tersebut tampak seperti yang Anda lihat di Restaurant Chooser, tetapi ada cara yang lebih mudah dan, dalam banyak hal, lebih baik, yaitu membuat komponen kustom yang dapat digunakan kembali di mana pun Anda membutuhkannya.

CustomButton.js

Tombol kustom tersebut ditempatkan di berkas CustomButton.js di direktori komponen. Dengan melakukan ini, kita akan dapat menggunakan tag <CustomButton> kapan pun kita membutuhkan salah satu tombol khusus ini.

```
import React, { Component } from "react";
import PropTypes from "prop-types";
import { TouchableOpacity, Text } from "react-native";
```

Pertama, kita harus mengimpor React, seperti biasa, dan kita juga harus mengimpor Component, karena kita akan memperluasnya untuk membuat komponen baru. Modul PropTypes adalah sesuatu yang kita perlukan, agar properti kustom tersedia di CustomButton kita.

Terakhir, untuk membuat tombol, kita akan menggunakan dua komponen React Native lainnya: TouchableOpacity dan Text. Anda pernah melihat Text sebelumnya, yang berfungsi untuk meletakkan teks di layar, tetapi TouchableOpacity adalah hal baru. Singkatnya, fitur ini memungkinkan kita membuat area layar yang bereaksi terhadap sentuhan dan memberikan umpan balik visual saat sentuhan terjadi melalui perubahan opasitas secara bertahap. Kedengarannya seperti sesuatu yang seharusnya dilakukan tombol, bukan?

```
class CustomButton extends Component {
 render() {
      const{text,onPress,buttonStyle, textStyle, width, disabled } =
      this.props;
```

Penugasan destrukturisasi ini bertanggung jawab untuk mengambil nilai beberapa properti yang dapat dimiliki komponen ini, properti yang tidak tersedia sebagai hasil dari perluasan Komponen, dan memasukkannya ke dalam beberapa variabel yang dapat kita gunakan di sisa kode.

Begini masalahnya: Anda dapat melampirkan properti sembarangan ke komponen yang Anda buat tanpa efek buruk, dan kode di dalam komponen dapat memperoleh akses ke properti tersebut melalui this.props. Namun, hal itu rawan kesalahan, karena pengguna komponen Anda tidak akan tahu jenis apa yang diharapkan oleh properti tertentu. Di situlah sesuatu yang disebut propTypes berperan. Berikut ini adalah potongan kode yang muncul setelah metode render():

```
CustomButton.propTypes = {
 text: PropTypes.string.isRequired,onPress:PropTypes.func.isRequired,
```

```
    buttonStyle: PropTypes.object, textStyle: PropTypes.object,
    width: PropTypes.string, disabled: PropTypes.string
  };
```

Anda melampirkan atribut propTypes ini ke komponen kustom Anda, dan di dalamnya, Anda mendefinisikan setiap prop tambahan yang didukung komponen Anda, dan untuk masing-masing, Anda menentukan fungsi yang akan memvalidasi prop tersebut. Di sini, saya menggunakan beberapa validator yang ada yang disediakan oleh modul PropTypes. Modul ini menyediakan beberapa validator, seperti string, array, bool, dan number, hanya untuk menyebutkan beberapa. Selain itu, beberapa varian menyertakan isRequired juga, jadi string.isRequired, misalnya seperti yang Anda lihat di sini, memberi tahu React Native bahwa prop text harus ada dan harus berupa string. Kita akan mendapatkan kesalahan yang sangat membantu jika validasi gagal untuk setiap prop, sehingga lebih mudah untuk menemukan masalah. Ini juga berfungsi sebagai bentuk dokumentasi diri, karena prop yang mewakili API dari komponen kustom tidak perlu ditebak; prop tersebut terdefinisi dengan baik, berkat propTypes.

Sekarang, saatnya mendefinisikan komponen.

```
return (
 <TouchableOpacity
      style={[
            {padding:10,height:60,borderRadius:8,margin:10,
            width
            : width,
                  backgroundColor :
                        disabled != null && disabled === "true" ? "#e0e0e0" :
                        "#303656",
                  },
                  buttonStyle
             ] }
            onPress={ () => { if (disabled == null || disabled === "false") {
            onPress() } } }
        >
           <Text style={ [
             {fontSize: 20, fontWeight: "bold", color:"#ffffff",
             textAlign: "center", paddingTop: 8
             },
             textStyle
             ] } >
             {text}
             </Text>
           </TouchableOpacity>
           );
      }
 }
```

Ini adalah ide dasar yang sama seperti saat kita membuat komponen: buat kelas yang diperluas dari beberapa komponen, di sini, kelas Komponen literal, dan buat metode render(). CustomButton kita hanyalah komponen y yang dibungkus di sekitar komponen y.

Seperti yang Anda lihat, ada beberapa gaya yang diterapkan pada TouchableOpacity, untuk memberinya bantalan, tinggi statis, dan sudut membulat. Sekarang, tombol juga memiliki lebar statis, setidaknya instance CustomButton kita memilikinya, tetapi Anda akan melihat bahwa nilai atribut gaya lebar diambil dari variabel lebar.

Atribut gaya terakhir adalah backgroundColor, dan di sini kita menggunakan beberapa logika untuk menentukan apakah tombol harus berwarna abu-abu (dinonaktifkan adalah benar) atau harus berwarna biru (aktif, dinonaktifkan adalah salah, atau tidak disertakan). Logika serupa digunakan selanjutnya dalam prop onPress, sehingga hanya tombol aktif yang merespons sentuhan.

Anda juga harus melihat bahwa atribut gaya sedikit berbeda dari apa pun yang pernah Anda lihat sebelumnya, karena tampaknya menggunakan notasi array, seperti yang ditunjukkan oleh penggunaan tanda kurung. Ya, Anda dapat menerapkan beberapa gaya ke komponen dengan cara ini, dan maksudnya melakukannya di sini adalah agar gaya tombol dapat diganti atau disesuaikan lebih lanjut oleh pengembang menggunakan CustomButton, dengan menyediakan properti buttonStyle.

Komponen Text berada di dalam TouchableOpacity dan tidak lebih dari sekadar gaya dasar, lagi-lagi menggunakan notasi array (yang dengannya textStyle dapat mengganti atau memperluas gaya dasar), sehingga gaya tersebut dapat diubah atau diperluas, sesuai kebutuhan, lalu teks itu sendiri, diambil dari properti text yang didefinisikan sebelumnya.

Ini sebenarnya bukan bagian kode yang canggih, tetapi menunjukkan cukup banyak tentang pembuatan komponen khusus. Namun, ada satu hal terakhir yang harus kita lakukan, dan saya rasa Anda tahu apa itu: mengekspor komponen. export default CustomButton; React Native akan menangani penambahan CustomButton ke registri internal komponennya saat Anda mengimpor modul ini ke modul lain, yang membuat <CustomButton> berfungsi, seperti yang akan Anda lihat segera.

CustomTextInput.js

Ada satu komponen kustom lain yang digunakan dalam Restaurant Chooser, yaitu CustomTextInput, yang kodenya, tentu saja, ditemukan dalam berkas CustomTextInput.js. Secara umum, komponen ini memiliki tujuan dasar yang sama dengan CustomButton, tetapi dengan sedikit tambahan, meskipun tidak banyak.

```
import React, { Component } from "react";
import PropTypes from "prop-types";
import { Platform, StyleSheet, Text, TextInput, View } from "react-native";
```

Kita mulai dengan dua baris yang sama seperti sebelumnya, mengimpor React, Component, dan PropTypes. Kemudian, kita juga harus mengimpor beberapa komponen lain yang akan digunakan untuk membangun komponen baru ini. Modul Platform akan memungkinkan kita untuk melakukan beberapa percabangan, berdasarkan OS yang menjalankan aplikasi.

StyleSheet, tentu saja, adalah cara kita mendefinisikan stylesheet yang akan digunakan komponen. Komponen Text akan diperlukan, sehingga kita dapat memiliki label yang dilampirkan ke TextInput, sehingga komponen tersebut juga diimpor. Kita juga akan memerlukan kontainer untuk semuanya, dan di situlah komponen View berperan.

Hal pertama yang harus dilakukan adalah mendefinisikan stylesheet.

```
const styles = StyleSheet.create({
   fieldLabel : { marginLeft : 10 },
   textInput : {
      height : 40, marginLeft : 10, width : "96%", marginBottom : 20,
       ...Platform.select({
       ios : { marginTop : 4, paddingLeft : 10, borderRadius : 8,
        borderColor : "#c0c0c0", borderWidth : 2
       },
       android : { }
     })
   }
 });
```

Gaya fieldLabel akan diterapkan pada komponen Text yang akan berfungsi sebagai label untuk kolom tersebut. Label memang merupakan alasan utama komponen kustom ini sejak awal. Tentu saja, saya bisa saja meletakkan komponen Text, diikuti oleh komponen TextInput, di layar mana pun tempat saya ingin meletakkan label dan kolom entri teks, tetapi masalahnya adalah komponen-komponen tersebut akan berakhir dengan posisi yang berbeda antara iOS dan Android.

Faktanya, itulah alasan logika percabangan dalam gaya textInput: gaya yang diperlukan untuk iOS agar kolom-kolom tersebut sejajar dengan label (serta beberapa gaya yang tidak penting tetapi bagus untuk menambahkan gaya visual, seperti sudut membulat dan pewarnaan) tidak diperlukan di Android. Saya bisa saja meniru gaya ini pada setiap TextInput dan menghindari pembuatan komponen kustom, tetapi dengan begitu saya tidak akan memiliki kesempatan untuk menunjukkan komponen kustom kepada Anda. Selain itu, saya biasanya mencoba dan mengikuti prinsip DRY (Don’t Repeat Yourself), jika memungkinkan, dan ini adalah tempat yang sangat baik untuk melakukannya. Saya pikir.

Dengan gaya yang ditentukan, kita dapat melanjutkan membangun komponen.

```
class CustomTextInput extends Component {
  render() {
    const {
     label, labelStyle, maxLength, textInputStyle, stateHolder,
     stateFieldName
 } = this.props;
 return (
      <View>
            <Textstyle={[styles.fieldLabel,labelStyle] }>{label}</Text>
```

```
            <TextInput maxLength={ maxLength }
                onChangeText={ (inText) => stateHolder.setState(
                   () => {
                     const obj = { };
                     obj[stateFieldName] = inText;
                     return obj;
                   }
             ) }
             style={ [ styles.textInput, textInputStyle ] }
           />
      </View>
   );
  }
 }
```

Sekali lagi, kami memiliki beberapa properti khusus yang tersedia: label adalah teks label; labelStyle adalah gaya tambahan yang ingin kami terapkan pada label; maxLength memungkinkan kami memiliki panjang teks maksimum yang dapat dimasukkan pengguna; textInputStyle memungkinkan pengembang mengganti atau memperluas gaya dasar TextInput.

Properti stateHolder dan stateFieldName adalah referensi ke objek yang menyimpan status untuk komponen TextInput, dan stateFieldName adalah nama bidang pada objek tersebut, masing-masing. Ini diperlukan agar kode dalam fungsi properti onChangeText berfungsi dengan benar dalam semua kasus, karena objek tersebut mungkin tidak selalu menjadi apa yang dirujuk oleh kata kunci this (jika kami tidak menggunakan notasi panah tebal) atau bahkan apa yang terikat pada fungsi tersebut jika kami mencoba melakukannya dengan fungsi bergaya klasik.

Menyediakan properti ini, dan menjadikannya wajib, memastikan bahwa komponen ini akan dapat digunakan dalam situasi apa pun, terlepas dari bagaimana data status disimpan dalam komponen mana pun yang menggunakannya. Konten yang dirender adalah komponen View di tingkat atas, komponen Text untuk label, dan komponen TextInput itu sendiri di dalam View. Anda dapat melihat bagaimana stateHolder dan stateFieldName digunakan dalam onChangeText untuk memperbarui nilai saat nilai dalam komponen TextInput berubah.

Tentu saja, komponen ini, seperti CustomButton, memiliki propTypes yang didefinisikan setelah metode render().

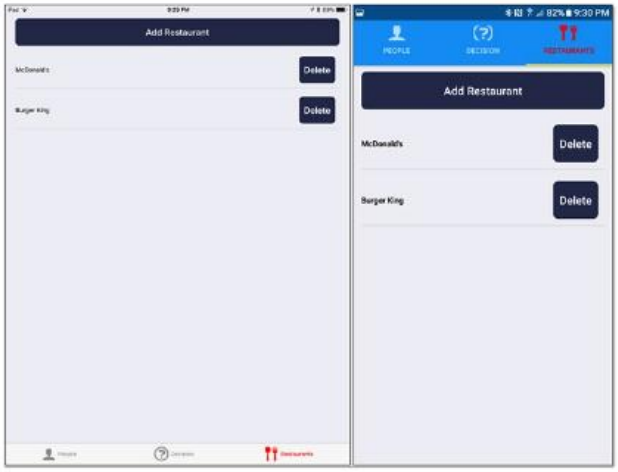
```
CustomTextInput.propTypes = {
 label: PropTypes.string.isRequired, labelStyle : PropTypes.object,
 maxLength: PropTypes.number, textInputStyle: PropTypes.object,
 stateHolder:PropTypes.object.isRequired, stateFieldName : PropTypes.
 string.isRequired
};
```

Kali ini, kita memerlukan teks label, serta stateHolder dan stateFieldName. Setelah itu, kita tinggal mengekspor export default CustomTextInput; dan itu adalah komponen kustom

lainnya, siap digunakan! Sekarang, mari kita lihat kode untuk layar Restoran, yang akan menyertakan penggunaan kedua komponen kustom ini.

Layar Pertama Kita: RestaurantsScreen.js

Layar Restoran adalah tempat pengguna memasukkan nama restoran yang akan dipilih nanti. Layar ini terdiri dari dua sublayar, yaitu: layar daftar dan layar tambah. Layar daftar ditunjukkan pada Gambar 17.8.



Gambar 17.8. Layar daftar restoran

Layarnya cukup sederhana, terdiri dari tombol di bagian atas yang mengarahkan pengguna ke layar tambah, untuk menambahkan restoran baru, dan daftar restoran yang ada, dengan kemampuan untuk menghapus satu restoran. Kemampuan untuk mengubah restoran tidak disediakan. Pengguna harus menghapus dan menambahkan lagi untuk membuat perubahan, tetapi, hei, mungkin ini akan menjadi tugas yang bagus untuk Anda lakukan sendiri, untuk mendapatkan sedikit pengalaman, petunjuk, petunjuk.

Kodenya juga cukup sederhana, dimulai dengan beberapa impor.

```
import React from "react";
import CustomButton from "../components/CustomButton";
import CustomTextInput from "../components/CustomTextInput";
import { Alert, AsyncStorage, BackHandler, FlatList, Picker, Platform,
ScrollView,
 StyleSheet, Text, View
} from "react-native";
import { StackNavigator } from "react-navigation";
import { Root, Toast } from "native-base";
import { Constants } from "expo";
```

React, StyleSheet, dan Platform sudah cukup familiar bagi Anda, begitu pula Text dan View. Seperti yang disebutkan di bagian sebelumnya, kita akan menggunakan komponen CustomButton dan CustomTextInput di layar ini. Ada daftar modul dari React Native itu sendiri

yang akan kita gunakan. Alert akan memungkinkan kita untuk menampilkan beberapa pesan kepada pengguna. AsyncStorage akan memberi kita akses ke mekanisme penyimpanan data sederhana yang sangat mirip dengan Local Storage di browser. BackHandler akan diperlukan untuk mengontrol apa yang dilakukan Android saat tombol kembali perangkat keras ditekan. FlatList, tentu saja, adalah cara kita menampilkan daftar restoran. Picker akan digunakan saat menambahkan restoran, untuk memberi pengguna daftar opsi yang dapat dipilih, seperti yang akan Anda lihat nanti. ScrollView akan berperan di layar tambah, untuk memastikannya bergulir ke kanan, karena akan ada terlalu banyak bidang entri untuk ditampilkan di satu layar. Ada juga navigator Navigasi react baru yang berperan di sini: StackNavigator. Itulah yang akan memungkinkan kita untuk beralih antara layar daftar dan layar tambah. Modul Constants memberi kita beberapa informasi tentang perangkat yang kita gunakan, mirip dengan modul Platform, tetapi Constants disediakan oleh Expo dan memberi kita beberapa informasi yang berbeda, seperti yang akan Anda lihat.

17.6 KOMPONEN PIHAK KETIGA: NATIVEBASE

Meskipun React Native menawarkan koleksi komponen yang sangat bagus untuk membangun aplikasi Anda (dan meskipun Anda belum melihatnya, Expo menyediakan beberapa lagi), dan meskipun Anda dapat membuat komponen kustom Anda sendiri, seperti yang Anda lihat di bagian sebelumnya, terkadang lebih baik untuk mencari komponen dari pustaka pihak ketiga. Jika Anda memilih pustaka yang bagus, Anda dapat memperoleh akses ke banyak komponen yang luar biasa dan solid yang akan memperluas palet tempat Anda membangun aplikasi secara signifikan.

Salah satu pustaka pihak ketiga yang paling populer yang tersedia adalah NativeBase (https://docs.nativebase.io). Ini adalah pustaka sumber terbuka dan gratis yang tidak hanya menyediakan beberapa widget baru dan praktis, tetapi pada intinya, dibangun dengan tujuan membuat komponen dapat ditata, tanpa mengubah kode di baliknya.

Seperti kebanyakan pustaka, untuk menggunakan NativeBase, Anda harus menginstalnya dengan NPM, baik dengan npm install --save native-base atau dengan menambahkannya ke package.json lalu melakukan instalasi NPM untuk mengunduhnya. Dengan cara apa pun, setelah selesai, Anda akan menemukan banyak komponen baru yang tersedia untuk Anda, termasuk (tetapi tidak terbatas pada)

1) Akordeon: Beberapa bagian konten yang visibilitasnya dapat diubah dengan mengeklik tajuk di atasnya
2) Lencana: Menampilkan ikon pemberitahuan, seperti jumlah email, pada ikon (atau elemen lain)
3) Kartu: Wadah konten yang biasanya terlihat pada perangkat Android
4) FAB: Tombol tindakan mengambang, biasanya tombol melingkar yang mengambang di atas UI yang memberi pengguna akses ke berbagai fungsi
5) Tata Letak: Saya membahas tata letak di React Native di Bab 4 (dan saya akan membahasnya sedikit di sini), tetapi komponen ini menyediakan cara yang bisa dibilang lebih mudah dan lebih fleksibel untuk membuat tata letak daripada yang Anda

dapatkan di React Native secara default (dan, sebagai pratinjau, itu adalah flexbox).

6) SearchBar: Bidang entri khusus dengan ikonografi yang berlaku untuk melakukan pencarian
7) Segment: Alternatif untuk tab, segment tampak seperti dua (atau lebih) tombol yang disatukan, yang, saat diketuk, akan disorot (sementara yang lain tidak disorot, jika sebelumnya disorot).
8) SwipableList: Daftar komponen yang memungkinkan pengguna untuk menggeser ke kiri dan/atau ke kanan pada item, untuk menampilkan tombol tindakan atau konten lainnya
9) Typography: Bukan komponen tertentu, tetapi sekelompok komponen yang memungkinkan Anda membuat judul seperti HTML, menggunakan tag H1, H2, dan H3

Ini hanyalah contoh komponen yang disediakan Native Base, dan, pada kenyataannya, Anda akan melihat dua komponen lainnya—yang diimpor dalam kode layar Restaurants, Root, dan Toast. Namun, jangan terlalu terburu-buru; masih banyak kode yang bisa dilihat.

Layar Daftar

Seperti yang saya sebutkan, layar Restoran (serta layar Orang dan Saatnya Mengambil Keputsan) sebenarnya terdiri dari sejumlah sublayar, dan yang pertama untuk Restoran adalah layar daftar. Jadi, untuk membuat layar ini, kita mulai, seperti biasa, dengan membuat komponen.

```
class ListScreen extends React.Component {
 constructor(inProps) {
      super(inProps);
      this.state = { listData : [ ] };
 }
```

Setiap kali Anda membangun komponen dan harus melakukan sesuatu pada waktu konstruksi (yang tidak diwajibkan tetapi mungkin lebih sering diperlukan), Anda akan mulai dengan meneruskan objek yang dirujuk oleh argumen inProps, yang berisi semua properti yang ditentukan pada tag untuk komponen dan diteruskan ke konstruktor oleh React Native, ke konstruktor superkelas. Faktanya, jika Anda tidak melakukan ini, Anda mungkin akan mengalami masalah dengan hal-hal yang tidak berfungsi, jadi sebagai aturan umum, Anda harus selalu melakukan ini. (Mungkin ada beberapa kasus di mana Anda tidak harus atau ingin melakukannya, tetapi kemungkinannya sangat sedikit dan jarang sehingga lebih baik untuk tidak mempertimbangkan kemungkinan tersebut, kecuali jika Anda benar-benar harus melakukannya.)

Konstruktor biasanya juga merupakan tempat Anda mendefinisikan atribut status pada komponen, jika Anda memerlukannya (ingat, tidak semua komponen memerlukan status). Di sini, objek status akan berisi larik objek yang akan menjadi data yang dirender oleh daftar.

Berbicara tentang daftar dan rendering, setelah konstruktor muncul metode render() yang

mudah dipahami. Lihat semuanya, lalu kita akan uraikan bersama.

```
render() { return (
  <Root>
   <View style={styles.listScreenContainer}>
     <CustomButton text="Add Restaurant" width="94%"
     onPress={ () => { this.props.navigation.navigate("AddScreen"); } } />
   <FlatListstyle={styles.restaurantList} data={this.state.listData}
     renderItem={ ({item}) =>
      <View style={styles.restaurantContainer}>
       <Text style={styles.restaurantName}>{item.name}</Text>
       <CustomButton text="Delete"
         onPress={ () => {
          Alert.alert("Please confirm",
            "Are you sure you want to delete this restaurant?",
            [
             {text : "Yes", onPress: () => {
               AsyncStorage.getItem("restaurants",
                 function(inError, inRestaurants) {
                  if (inRestaurants === null) {
                    inRestaurants = [ ];
                  } else {
                    inRestaurants = JSON.parse(inRestaurants);
                  }
                  for (let i = 0; i < inRestaurants.length; i++) {
                         const restaurant = inRestaurants[i];
                         if (restaurant.key === item.key) {
                            inRestaurants.splice(i, 1);
                            break;
                          }
                        }
                     AsyncStorage.setItem("restaurants",

                         JSON.stringify(inRestaurants), function() {
                                                  this.setState({
                         listData : inRestaurants });
                        Toast.show({ text : "Restaurant deleted",
                        position : "bottom", type : "danger",
                         duration : 2000
                       });
                     }.bind(this)
                    );
                  }.bind(this)
                 );
               } },
              {text:"No"}, { text : "Cancel", style : "cancel" }
             ],
             {cancelable : true }
            )
         } } />
       </View>
```

```
      }
    />
  </View>
  </Root>
); }
```

Pertama, kita memiliki sesuatu yang belum pernah Anda lihat sebelumnya: elemen Root. Ini adalah komponen NativeBase yang diperlukan agar komponen Toast, yang akan kita gunakan untuk menampilkan pesan, dapat berfungsi. Komponen ini menyediakan elemen kontainer yang dikontrol dan ditambah oleh NativeBase, sebagaimana diperlukan, agar Toast dapat berfungsi. (Komponen ini juga diperlukan untuk beberapa komponen NativeBase lainnya. Anda harus membaca dokumen untuk menentukan komponen mana yang memerlukannya.)

Di dalam elemen Root, kita memiliki View yang akan menyediakan kontainer yang dapat kita beri gaya dengan tepat untuk menerapkan tata letak yang diinginkan. Gaya seperti apa, Anda bertanya? Nah, gaya listScreenContainer di sini:

```
listScreenContainer:{flex:1,alignItems:"center", justifyContent :
"center",
 ...Platform.select({
    ios : { paddingTop : Constants.statusBarHeight },
    android : { }
  })
 }
```

Seperti yang saya sebutkan sebelumnya, tata letak adalah topik yang akan saya bahas secara terperinci di Bab 4, tetapi untuk saat ini, saya akan memberi tahu Anda bahwa konfigurasi gaya ini memastikan bahwa Tampilan ini mengisi seluruh layar (flex : 1) dan bahwa setiap anak di dalamnya dipusatkan baik secara horizontal (justifyContent : "center") maupun vertikal (alignItems : "center"). Metode Platform.select() digunakan untuk menyetel atribut paddingTop untuk iOS tetapi tidak untuk Android, sehingga wadah tidak tumpang tindih dengan bilah status.

Berikutnya adalah komponen Add Restaurant CustomButton. Ini adalah sedikit konfigurasi sederhana, tetapi pengendali onPress memberi Anda sesuatu yang baru untuk dilihat. Seperti yang akan Anda temukan menjelang akhir bab ini, layar daftar (serta layar tambah yang akan kita bahas selanjutnya) ditempatkan di dalam React Navigation StackNavigator. Navigator ini menyediakan cara untuk memiliki beberapa komponen, daftar dan sublayar tambah, yang ditumpuk di atas satu sama lain, sehingga hanya satu yang terlihat pada waktu tertentu, dan kita dapat memanggil beberapa metode untuk beralih di antara keduanya. React Navigation akan secara otomatis menambahkan atribut navigasi ke koleksi props dari komponen tingkat atas. Atribut tersebut adalah objek yang berisi beberapa metode yang dapat kita panggil, salah satunya adalah navigasi(). Yang kita berikan padanya adalah nama layar yang disertakan StackNavigator yang ingin kita tampilkan, dan navigator akan mengurus peralihan di antara keduanya.

Setelah itu muncul komponen FlatList. Kita telah membahasnya di Bab 2, tetapi sebagai

penyegaran, ini adalah komponen yang merender daftar item sederhana. Item yang akan dirender ditentukan oleh atribut data dan merujuk ke array listData di objek status untuk komponen ini. Jangan khawatir tentang bagaimana data masuk ke array tersebut; kita akan melihatnya setelah kita selesai dengan metode render().

FlatList juga memiliki gaya yang diterapkan, dan gaya tersebut adalah ini:

```
restaurantList : { width : "94%" }
```

Hal sebelumnya dilakukan untuk memastikan bahwa ada ruang di kedua sisi daftar, yang menurut saya tampak lebih menarik. Ini berfungsi karena gaya View induk memusatkan anak-anaknya, ingat, jadi kita akan berakhir dengan 3% lebar layar di kedua sisi FlatList.

Properti renderItem pada FlatList adalah fungsi yang Anda, sebagai pengembang, sediakan yang dipanggil FlatList untuk merender setiap item. Seperti yang Anda lihat, item tersebut diteruskan ke fungsi ini, dan Anda dapat mengembalikan hampir semua struktur yang Anda bisa dari metode render(), karena di balik layar, React Native membuat komponen dengan cepat dari apa yang Anda sediakan di sini. Dalam kasus ini, View dibuat, untuk memuat item, karena akan ada beberapa bagian di dalamnya. View ini menerapkan gaya berikut:

```
restaurantContainer : { flexDirection : "row", marginTop : 4,
marginBottom : 4,
 borderColor : "#e0e0e0", borderBottomWidth : 2, alignItems : "center"
 }
```

Jika Anda belum pernah melihat flexbox beraksi sebelumnya sekali lagi, kita akan membahasnya di bagian selanjutnya bahwa flexDirection, saat disetel ke baris, berarti anak-anak dari View ini akan ditata dalam satu baris, berdampingan di layar. Atribut lainnya adalah untuk memastikan bahwa ada ruang di atas dan di bawah setiap item, bahwa setiap item memiliki batas abu-abu muda setebal dua piksel, dan bahwa anak-anak dalam View dipusatkan secara horizontal.

Berbicara tentang anak-anak, yang pertama adalah komponen Teks yang merupakan nama restoran, diambil dari objek yang diteruskan ke metode prop renderItem. Ini juga memiliki gaya sederhana yang diterapkan.

```
restaurantName : { flex : 1 }
```

Ya, lebih banyak flexbox! Ini agar nama restoran akan mengambil ruang sebanyak mungkin, dikurangi ruang untuk Delete CustomButton, yang merupakan anak kedua di dalam View. Tombol akan secara otomatis menyesuaikan ukuran dengan teksnya, sehingga secara efektif memiliki lebar yang ditentukan, yang berarti bahwa komponen teks akan mengisi ruang horizontal yang tersisa setelah tombol dirender.

Sekarang, di dalam tombol tersebut terdapat pengendali onPress, dan ada beberapa

hal menarik yang terjadi di sana. Pertama, API Peringatan digunakan untuk meminta pengguna mengonfirmasi penghapusan. Ada tiga tombol: Ya, Tidak, dan Batal. Menekan salah satu dari tombol tersebut (atau mengetuk di luar pop-up di Android, berkat atribut yang dapat dibatalkan yang disetel ke benar) akan menutup pop-up tanpa terjadi apa pun. Kode di dalam pengendali tombol Ya yang melakukan semua pekerjaan, seperti yang Anda harapkan.

Pekerjaan itu dilakukan dalam dua bagian. Pertama, restoran harus dihapus. Ini dilakukan dengan menggunakan API AsyncStorage, untuk mengambil daftar restoran. AsyncStorage sangat mirip dengan Penyimpanan Lokal di peramban web karena merupakan penyimpanan data kunci-nilai sederhana. Anda hanya dapat menyimpan string di dalamnya, jadi Anda harus melakukan serialisasi dan deserialisasi apa pun ke dan dari string, seperti objek JavaScript, seperti yang terjadi di sini. Metode getItem() dipanggil untuk mendapatkan objek di bawah kunci restaurants. Jika belum ada, artinya pengguna belum membuat restaurants, array kosong akan dibuat. Jika tidak, string yang diambil akan dideserialisasi ke dalam objek menggunakan metode JSON.parse() yang terkenal (dan tersedia dalam kode React Native). Setelah itu, tinggal mengulang array dan menemukan restoran dengan kunci (yang dimiliki semua restoran) yang cocok dengan kunci item FlatList yang sedang dirender dan menghapusnya dari array.

Setelah dihapus dari array, langkah berikutnya adalah menulis array kembali ke penyimpanan, menggunakan AsyncStorage.setItem(), menggunakan JSON.stringify() untuk melakukan serialisasi array restaurants ke dalam string untuk penyimpanan. Perhatikan bahwa getItem() dan setItem() adalah metode asinkron, jadi Anda harus menyediakan penangan panggilan balik untuk masing-masing, dan penghapusan dari array dan panggilan ke setItem() dilakukan dalam panggilan balik untuk panggilan getItem().

Terakhir, dalam penangan panggilan balik untuk panggilan setItem(), API NativeBase Toast digunakan untuk menampilkan pesan yang menunjukkan bahwa penghapusan berhasil. Ini berbentuk spanduk kecil yang muncul di bagian bawah layar, yang ditentukan selama dua detik, yang akan dibaca karena telah menetapkan jenis ke danger.

Bagian terakhir dari persamaan adalah sesuatu yang saya singgung sebelumnya, yaitu, memasukkan data ke dalam FlatList sejak awal. Itu dilakukan dalam metode componentDidMount(), yang akan dipanggil React Native setelah komponen tingkat atas dibuat.

```
componentDidMount() {
    BackHandler.addEventListener( "hardwareBackPress", () => { return  true; }
    );
 AsyncStorage.getItem("restaurants",
      function(inError, inRestaurants) {
            if (inRestaurants === null) {
                  inRestaurants = [ ];
            } else {
                  inRestaurants = JSON.parse(inRestaurants);
            }
            this.setState({ listData : inRestaurants });
      }.bind(this)
```

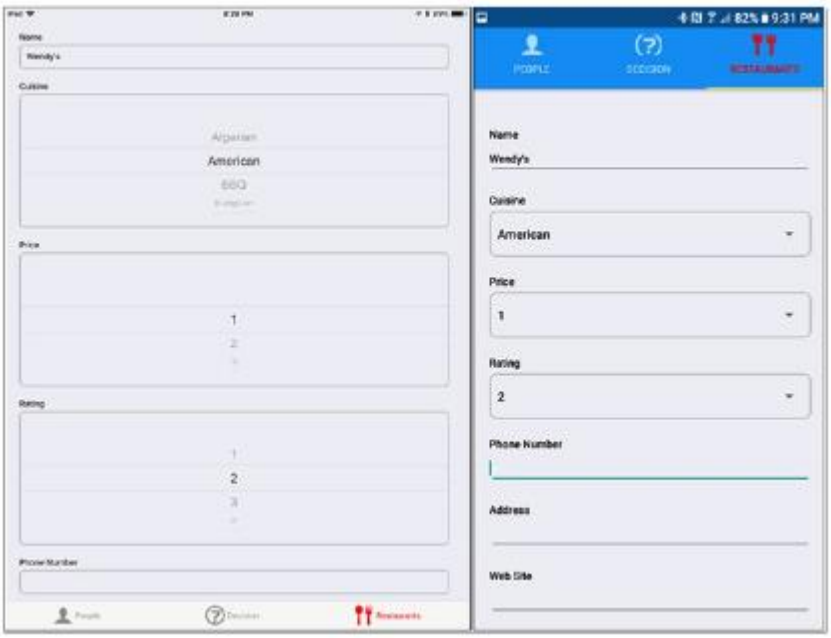
```
    );
  };
}
```

Pertama, kita harus mempertimbangkan apa yang seharusnya atau tidak seharusnya terjadi saat pengguna menekan tombol kembali pada perangkat keras di perangkat Android. Secara default, pengguna akan kembali ke setiap layar yang telah dinavigasinya dalam urutan terbalik (tumpukan dibuat saat Anda beralih dari satu layar ke layar lain, jadi menekan tombol kembali hanya akan memunculkan layar yang ditumpuk). Ini sering kali merupakan hal yang Anda inginkan, tetapi dalam aplikasi ini, menurut saya tidak berfungsi sebagaimana yang Anda harapkan secara logis, jadi saya ingin menonaktifkan fungsionalitas tersebut. Untuk melakukannya, Anda melampirkan event listener menggunakan API BackHandler, dan fungsi yang dijalankan hanya perlu mengembalikan true, dan navigasi yang biasanya terjadi akan dihentikan.

Setelah itu, saatnya memasukkan data ke dalam FlatList. Tentu saja, data tersebut tersimpan di AsyncStorage, jadi Anda tinggal menggunakan metode getItem() yang sama seperti yang Anda lihat beberapa saat yang lalu, melakukan pemeriksaan yang sama untuk null guna menghindari kesalahan, lalu memanggil setState() pada komponen dan meneruskan daftar restoran sebagai atribut listData. React Native akan menangani semua hal lainnya.

Layar Tambah

Kemampuan untuk mencantumkan restoran tidak akan berguna jika kita tidak dapat membuat restoran untuk dicantumkan. Di situlah layar tambah, yang dapat Anda akses dengan mengeklik tombol Tambah Restoran pada layar daftar, tentu saja, berperan. Gambar 17.9 menunjukkan layar tersebut.



Gambar 17.9. Layar tambah restoran

Setiap restoran dapat memiliki beberapa atribut, termasuk nama, jenis masakan, peringkat bintang, peringkat harga, dan informasi telepon serta alamat. Semua ini dimasukkan menggunakan berbagai bidang entri data, termasuk komponen CustomTextInput yang kita buat sebelumnya dan komponen Picker milik React Native sendiri.

Namun, sebelum kita membahasnya, mari kita lihat bagaimana komponen ini dimulai.

```
class AddScreen extends React.Component {
  constructor(inProps) {
      super(inProps);
      this.state = { name : "", cuisine : "", price : "", rating : "",
      phone : "", address : "", webSite : "", delivery : "",
      key : `r_${new Date().getTime()}`
     };
 }
```

Seperti pada layar daftar, konstruktor dengan panggilan ke konstruktor superkelas akan didahulukan, diikuti oleh pembuatan objek status. Atribut tersebut sesuai dengan kriteria yang dapat Anda masukkan tentang restoran, kecuali atribut kunci, yang merupakan kunci unik yang akan dimiliki restoran tambahan, yang hanya berupa nilai stempel waktu sederhana. Itu bukan cara yang paling kuat untuk menghasilkan kunci unik untuk suatu objek, tetapi itu akan sesuai dengan kebutuhan kita di sini.

Pada titik ini, perlu dicatat bahwa file RestaurantScreen.js yang telah kita lihat memiliki dua komponen yang didefinisikan (sejauh ini), satu untuk layar daftar dan sekarang yang satu ini untuk layar penambahan. Ini, tentu saja, baik-baik saja, karena Anda dapat membuat kelas sebanyak mungkin dalam satu modul (yang merupakan RestaurantScreen.js) sesuai keinginan, tetapi hanya satu yang akan diekspor, dan Anda akan melihatnya setelah kita selesai dengan kode layar ini. Berbicara tentang kode tersebut, mari kita lihat metode render() berikutnya, dan seperti pada layar daftar, saya akan membiarkan Anda membacanya, lalu saya akan menguraikannya (meskipun sekarang, saya yakin Anda dapat mengerjakannya sendiri).

```
render() { return (
 <ScrollView style={styles.addScreenContainer}>
```

Karena ada cukup banyak bidang entri, hampir dapat dipastikan bahwa layar fisik perangkat tidak akan cukup besar untuk menampilkan semuanya sekaligus, jadi kita harus menyediakan ruang untuk menggulir. Di situlah komponen ScrollView yang menampung semua komponen lainnya berperan. Ini adalah komponen kontainer yang memungkinkan pengguna untuk menyeretnya guna menggulir. Dalam arti tertentu, ini seperti FlatList, tetapi FlatList merender komponen tertentu, dan melakukannya sedikit demi sedikit saat komponen tersebut muncul, ScrollView merender semua anaknya sekaligus dan tidak melakukannya dengan fungsi yang ditentukan untuk merender masing-masing. Karena sangat sederhana, satu-satunya hal yang harus kita pertimbangkan adalah bahwa ScrollView akan tumpang tindih dengan bilah status

jika kita tidak mengatasinya, dan di situlah gaya yang diterapkan padanya berperan.

```
addScreenContainer : { marginTop : Constants.statusBarHeight }
```

API Expo Constants kembali digunakan untuk mendapatkan tinggi bilah status tersebut, dan gaya marginTop yang sederhana memberi kita jarak yang diperlukan.

```
<View style={styles.addScreenInnerContainer}>
      <View style={styles.addScreenFormContainer}>
```

Sekarang, jika Anda memikirkan layar ini, sebenarnya ada dua bagian di dalamnya: komponen entri data dan tombol Batal dan Simpan di bagian bawah. Untuk menata tampilan ini sebagai entitas unik, kita akan membuat Tampilan lain di dalam ScrollView (sehingga kita dapat menata kedua bagian secara keseluruhan juga) lalu membuat dua Tampilan lagi di dalam tampilan itu, satu untuk komponen entri dan satu untuk tombol. Jadi, ScrollView memiliki Tampilan sebagai anak pertamanya, dan tampilan itu memiliki gaya berikut yang disertakan:

```
addScreenInnerContainer: { flex: 1, alignItems: "center", paddingTop:20, width
: "100%" }
```

Ini memastikan bahwa anak-anaknya berada di tengah dan mengisi layar secara horizontal. Ada juga beberapa bantalan tambahan di bagian atas, untuk memastikan bahwa saat konten digulir, konten tidak akan menggulir melewati bilah status.

Kemudian, di dalam Tampilan tersebut ada yang lain, yang pertama dari dua yang saya sebutkan, yang ini untuk komponen entri. Gaya juga diterapkan di sini.

```
addScreenFormContainer : { width : "96%" }
```

Menggunakan lebar 96% menempatkan sejumlah ruang di sekitar sisi komponen, seperti yang dilakukan pada layar daftar.

```
        <CustomTextInput label="Name" maxLength={20}
          stateHolder={this} stateFieldName="name" />
        <Text style={styles.fieldLabel}>Cuisine</Text>
        <View style={styles.pickerContainer}>
          <Picker style={styles.picker} prompt="Cuisine"
            selectedValue={this.state.cuisine}
            onValueChange={ (inItemValue) => this.setState({ cuisine :
            inItemValue }) }
         >
            <Picker.Item label="" value="" />
            <Picker.Item label="Algerian" value="Algerian" />
            <Picker.Item label="American" value="American" />
             ...
             <Picker.Item label="Other" value="Other" />
```

```
          ...
        </Picker>
      </View>
```

Sekarang, kita telah memiliki tiga kontainer View yang mendalam di tata letak, dan pada titik ini, kita dapat mulai menambahkan komponen entri, yang pertama adalah komponen CustomTextInput. Ini untuk nama restoran, jadi kita menyediakan teks label yang sesuai melalui properti label, memberitahunya berapa panjang entri maksimum (20), dan memberitahunya objek apa yang menyimpan status komponen ini (this, yang merupakan referensi ke instans kelas AddScreen itu sendiri, dan React Native tahu untuk mencari atribut status di dalamnya) dan atribut pada objek status tersebut, name.

Setelah itu muncul entri jenis masakan restoran. Ini dilakukan melalui komponen Picker, yang disertakan dengan React Native. Ini adalah kontrol pemutar sederhana di iOS, dan dialog pop-up dengan daftar opsi yang dapat diklik yang dapat digulir di Android, yang intinya adalah memaksa pengguna untuk memilih opsi dari daftar opsi yang tersedia. Namun, hanya dengan meletakkan komponen Picker saja tidak akan cukup, karena pengguna tidak akan tahu kegunaannya secara pasti, jadi kita akan menambahkan komponen Text sebelum komponen tersebut sebagai label, dan pada komponen Text tersebut, kita akan menerapkan gaya ini:

```
fieldLabel : { marginLeft : 10 }
```

Sasaran di sini adalah untuk memastikan bahwa label sejajar dengan sisi kiri kotak Picker, yang tidak akan terjadi jika kita tidak menerapkan gaya ini. Picker itu sendiri dibungkus dalam komponen View, sehingga gaya berikut dapat diterapkan padanya dan agar memiliki efek yang diinginkan:

```
pickerContainer : {
 ...Platform.select({
   ios : { },
   android : { width : "96%", borderRadius : 8, borderColor : "#c0c0c0",
   borderWidth : 2,
     marginLeft : 10, marginBottom : 20, marginTop : 4
    }
  })
 }
```

Dan efek yang diinginkan utamanya adalah memberi Picker sebuah border. Selain itu, lebarnya ditetapkan menjadi 96%, yang akan diisi Picker, dan beberapa padding ditambahkan di sekitarnya, yang semuanya bertujuan untuk membuat Picker tampak bagus dan pas di layar. Namun, perhatikan bahwa Platform.select() digunakan di sini lagi, karena, di iOS, gaya ini tidak diperlukan.

Kebetulan, untuk membuat Picker tampak serupa di kedua platform, View diperlukan di sekitar Picker, tetapi untuk menyelesaikan tugas, kita juga harus menerapkan beberapa gaya pada Picker itu sendiri.

```
picker : {
   ...Platform.select({
     ios : { width : "96%", borderRadius : 8, borderColor : "#c0c0c0",
     borderWidth : 2,
       marginLeft : 10, marginBottom : 20, marginTop : 4
     },
     android : { }
 })
 }
```

Dalam kasus ini, iOS-lah yang membutuhkan gaya, sedangkan Android tidak. Ketika gaya ini diterapkan pada View dan Picker yang memuatnya, dan pernyataan Platform.select() dipertimbangkan, kita akan mendapatkan layar yang tampak hampir sama di kedua platform, yang merupakan tujuannya. Keduanya tidak dapat terlihat sama persis, hanya karena Picker diiOS pada dasarnya tampak dan berfungsi berbeda dari Picker di Android, tetapi gaya ini membuat keduanya tampak cukup mirip, yang merupakan yang saya inginkan.

Menurut saya, definisi Picker itu sendiri cukup jelas. Properti prompt hanya untuk Android, karena ketika Picker diklik, Android akan membuka pop-up yang dapat digunakan pengguna untuk membuat pilihan, dan properti ini memastikan bahwa label pada layar tambah direplikasi ke pop-up tersebut. Properti selectedValue mengikat Picker ke atribut objek status yang sesuai, dan onValueChange menangani pembaruan nilai tersebut ketika berubah. Kemudian Picker mendapatkan beberapa komponen anak yang didefinisikan di bawahnya, tepatnya komponen Picker.Item, yang masing-masing diberi label dan nilai, yang terakhir adalah apa yang akan ditetapkan dalam status. Dalam kode berikut, saya telah memotong item dalam daftar sedikit dengan elipsis, hanya untuk menghemat sedikit ruang, tetapi percayalah, item tersebut disertakan dalam kode sebenarnya.

```
    <Text style={styles.fieldLabel}>Price</Text>
     <View style={styles.pickerContainer}>
       <Pickerstyle={styles.picker} selectedValue={this.state.price}
      prompt="Price"
       onValueChange={ (inItemValue) => this.setState({ price :
      inItemValue }) }
    >
       <Picker.Item label="" value="" />
       <Picker.Item label="1" value="1" />
       <Picker.Item label="2" value="2" />
       <Picker.Item label="3" value="3" />
       <Picker.Item label="4" value="4" />
       <Picker.Item label="5" value="5" />
      </Picker>
     </View>
<Text style={styles.fieldLabel}>Rating</Text>
<View style={styles.pickerContainer}>
<Picker style={styles.picker} selectedValue={this.state.rating}
    prompt="Rating"
    onValueChange={ (inItemValue) => this.setState({ rating :
```

```
    inItemValue }) }
 >
    <Picker.Item label="" value="" />
    <Picker.Item label="1" value="1" />
    <Picker.Item label="2" value="2" />
    <Picker.Item label="3" value="3" />
    <Picker.Item label="4" value="4" />
    <Picker.Item label="5" value="5" />
 </Picker>
</View>
```

Setelah kolom masakan, muncul kolom entri untuk harga (yang menunjukkan seberapa mahal restoran tersebut) dan satu untuk pemeringkatan—keduanya adalah Pickers—dan keduanya pada dasarnya memiliki pola yang sama seperti yang Anda lihat pada masakan, jadi tidak ada gunanya menelusurinya lagi di sini.

```
  <CustomTextInput label="Phone Number" maxLength={20}
  stateHolder={this}
     stateFieldName="phoneNumber" />
  <CustomTextInput label="Address" maxLength={20}
  stateHolder={this}
     stateFieldName="address" />

  <CustomTextInput label="Web Site" maxLength={20}
  stateHolder={this}
     stateFieldName="webSite" />
```

Setelah dua komponen Picker, ada tiga kolom CustomTextInput lagi, masing-masing untuk nomor telepon, alamat, dan situs web restoran. Seperti halnya harga dan peringkat, kolom-kolom tersebut tidak jauh berbeda dari kolom nama yang kita bahas sebelumnya, jadi saya rasa aman untuk melewatinya di sini juga.

Namun, apakah Anda menyadari bahwa tidak satu pun dari kolom-kolom ini wajib diisi? Tidak! Anda sebenarnya dapat membuat restoran tanpa nama dan data, yang membuatnya hampir tidak berguna. Namun, saya sengaja melakukan ini, hanya agar saya dapat menyarankan ini: mengapa Anda tidak beristirahat sejenak di sini dan mencoba mencari tahu cara membuat kolom-kolom wajib diisi? Apakah React Native menawarkan semacam tanda "jadikan kolom ini wajib diisi"? Atau apakah Anda harus menulis beberapa kode, misalnya, di tombol Simpan, yang akan kita bahas sebentar lagi, untuk melakukan validasi dan menampilkan pesan jika ada yang hilang (yang mungkin atau mungkin bukan petunjuk)? Apakah sebenarnya ada beberapa cara untuk melakukannya yang dapat Anda pilih, mungkin termasuk beberapa pustaka pihak ketiga? Saya tinggalkan ini sebagai latihan untuk Anda, pembaca yang budiman.

```
<Text style={styles.fieldLabel}>Delivery?</Text>
 <View style={styles.pickerContainer}>
    <Picker style={styles.picker} prompt="Delivery?"
```

```
     selectedValue={this.state.delivery}
     onValueChange={ (inItemValue) => this.setState({ delivery :
    inItemValue }) }
 >
   <Picker.Item label="" value="" />
   <Picker.Item label="Yes" value="Yes" />
   <Picker.Item label="No" value="No" />
  </Picker>
 </View>
</View>
```

Melengkapi kolom informasi restoran, ada satu kolom entri data lainnya—Pemilih lainnya—dan ini untuk memasukkan apakah restoran tersebut menyediakan layanan pesan antar. Ini adalah Pemilih dasar lainnya, sama seperti yang lainnya, kecuali untuk opsi yang tersedia, tentu saja, jadi kita tidak akan berlama-lama di sini, karena ada satu hal penting lagi yang perlu diperhatikan, yaitu tombolnya.

```
<View style={styles.addScreenButtonsContainer}>
  <CustomButton text="Cancel" width="44%"
    onPress={()=>{this.props.navigation.navigate("ListScreen");}}/>
  <CustomButton text="Save" width="44%"
    onPress={ () => {
     AsyncStorage.getItem("restaurants",
         function(inError, inRestaurants) {
         if (inRestaurants === null) {
            inRestaurants = [ ];
            } else {
                  inRestaurants = JSON.parse(inRestaurants);
            }
            inRestaurants.push(this.state);
            AsyncStorage.setItem("restaurants",
            JSON.stringify(inRestaurants), function() {
            this.props.navigation.navigate("ListScreen");
            }.bind(this)
                  );
                  }.bind(this)
                );
               } }
            />
       </View>
    </View>
</ScrollView>
); }
}
```

Tombol-tombol tersebut terdapat di View kedua yang merupakan anak dari View utama yang kita buat sebelumnya (yang merupakan anak dari ScrollView, ingat?). View ini menerapkan gaya berikut:

```
addScreenButtonsContainer:{flexDirection:"row",justifyContent :"center" }
```

Sekarang, di sini, saya ingin tombol-tombol tersebut berdampingan, ditata dalam satu baris, maka nilai flexDirection adalah baris. Namun, saya tetap ingin tombol-tombol itu sendiri dipusatkan, jadi justifyContent ditetapkan ke tengah (dan sekali lagi, saya akan memberi tahu Anda bahwa kita akan membahas semua hal tentang flexbox dan tata letak ini secara lebih terperinci di bab berikutnya, tetapi sebagian besar, Anda telah melihat dasar-dasar yang paling penting di seluruh bab ini).

Di dalam View ini terdapat dua komponen CustomButton, untuk Cancel dan Save, masing-masing. Tombol-tombol tersebut berukuran 44% dari lebar layar, yang menyisakan 12% dari lebar untuk spasi. Karena View induk menatanya di tengah dalam satu baris, itu berarti 4% dari lebar layar akan berada di kedua sisi tombol dan juga di antara keduanya (totalnya harus 100%, bagaimanapun juga). Tombol Batal tidak memiliki banyak pekerjaan yang harus dilakukan: tombol tersebut hanya menavigasi kembali ke layar daftar, dengan melakukan panggilan navigasi() pada atribut props.navigation dari komponen tingkat atas, seperti yang Anda lihat di pengendali onPress tombol Tambahkan Restoran.

Namun, tombol Simpan memiliki beberapa pekerjaan lagi yang harus dilakukan, yaitu menyimpan restoran yang baru saja dimasukkan informasinya oleh pengguna. Untuk melakukannya, pertama-tama kita harus mengambil daftar restoran dari AsyncStorage, seperti yang Anda lihat di layar daftar. Setelah selesai, yang harus kita lakukan adalah mendorong objek status untuk komponen tingkat atas ke dalam larik restoran, karena objek tersebut berisi semua data yang kita simpan, dan menuliskannya kembali ke AsyncStorage. Terakhir, kita menavigasi kembali ke layar daftar, melalui StackNavigator kita, dan React Native akan mengurus pembaruan daftar, berdasarkan metode componentDidMount() dari komponen layar daftar yang diaktifkan lagi.

Ada satu bagian kode terakhir yang harus kita lihat, dan meskipun kodenya sangat sedikit, itu sangat penting. Ingat ketika saya mengatakan bahwa Anda hanya dapat mengekspor satu komponen dari modul? Dan ingat ketika saya mengatakan bahwa kita menggunakan StackNavigator untuk beralih di antara daftar dan menambahkan layar dengan memberikan nama ke metode navigasi()? Nah, di sinilah kedua pernyataan itu berperan:

```
const RestaurantsScreen = StackNavigator(
    {ListScreen : { screen : ListScreen }, AddScreen : { screen :
   AddScreen } },
    {headerMode : "none", initialRouteName : "ListScreen" }
);
exports.RestaurantsScreen = RestaurantsScreen;
```

Kode ini membuat komponen StackNavigator dan mengekspornya, dan konfigurasi yang diteruskan ke sana mendefinisikan dua layar, ListScreen dan AddScreen. Untuk masing-masing, kami mengarahkannya ke komponen untuk layar tersebut, yang hasilnya adalah navigator sekarang mengetahui layar-layar ini berdasarkan nama, dan begitulah cara kami dapat menavigasi di antara keduanya. Kami juga memberi tahu navigator bahwa kami tidak ingin

tajuk apa pun ditampilkan dan bahwa rute daftar (layar) adalah yang default untuk ditampilkan, persis seperti yang kami inginkan di Restaurant Chooser.

17.7 MENGANALISIS LAYAR KEPUTUSAN DAN TATA LETAK DENGAN FLEXBOX

Tata Letak dan Flexbox

Pada bab terakhir, dan juga pada dua bab pertama, Anda melihat beberapa contoh tata letak di React Native. Anda melihat atribut gaya seperti flex, justifyContent, dan alignItems. Pada satu titik, saya bahkan mengatakan bahwa Anda sudah cukup melihat untuk dapat membuat tata letak Anda sendiri, dan itu benar. Pada dasarnya, React Native menggunakan CSS Flexbox (kebanyakan) yang sama dengan yang Anda gunakan di Web, dan hanya beberapa atribut yang menyediakan sebagian besar kemampuan yang Anda butuhkan untuk membuat sebagian besar tata letak, baik yang sederhana maupun yang rumit.

Namun, itu bukan penjelasan, dan Anda berhak mendapatkan lebih banyak penjelasan. Jadi, saya akan membahas tata letak dan flexbox secara lebih rinci sekarang. Flexbox adalah algoritme tata letak yang diperkenalkan ke CSS hanya beberapa tahun yang lalu. Ia dirancang dengan mempertimbangkan praktik modern, yang berarti hal-hal seperti desain responsif dan tata letak yang "fleksibel" (karena itulah namanya), berdasarkan ukuran komponen yang ditata, bahkan ketika ukuran tersebut tidak diketahui atau dinamis. Saat berurusan dengan flexbox, Anda memperhatikan beberapa elemen wadah induk, komponen dalam React Native, dan turunan langsungnya, dan bagaimana mereka disusun dalam satu arah atau yang lain.

Komponen yang menggunakan flexbox untuk menata turunannya kemudian dapat meminta satu atau beberapa turunan itu sendiri menggunakan flexbox untuk menata turunannya, dan seterusnya, sejauh yang Anda perlukan. Dengan cara ini, dengan menumpuk komponen yang menggunakan flexbox, Anda dapat mencapai hampir semua tata letak yang dapat Anda bayangkan. Yang diperlukan untuk menggunakan flexbox untuk tata letak adalah menentukan beberapa atribut gaya pada komponen. Seperti yang disebutkan sebelumnya, atribut yang paling umum digunakan adalah flex, justifyContent, alignItems, dan flexDirection. Ada beberapa atribut lain yang penting, tetapi agak kurang penting, termasuk alignSelf, flexWrap, alignContent, position, dan zIndex.

Secara umum, jika Anda menemukan referensi tentang flexbox di Web, referensi tersebut juga akan berlaku untuk React Native, meskipun ada beberapa perbedaan yang perlu Anda ketahui. Pertama, nilai default-nya sedikit berbeda di React Native. Dengan flexbox di Web, nilai default untuk flexDirection adalah baris, tetapi di React Native, nilainya adalah kolom. Kedua, parameter flex hanya mendukung satu angka. Baiklah, itu semua baik dan bagus, tetapi apa fungsi atribut-atribut ini? Sekarang mari kita bahas masing-masing. Atribut pertama adalah flexDirection (nilai yang valid: baris, kolom, kolom terbalik, dan baris-terbalik, dengan kolom sebagai nilai default).

Ini menentukan arah sumbu tata letak utama atau primer. Dengan kata lain, ini menentukan apakah anak-anak dari wadah akan ditata secara horizontal di layar (ketika flexDirection diatur ke baris) atau vertikal (ketika diatur ke kolom). Default yang diubah (diubah dari flexbox di Web) masuk akal dalam konteks ini, karena sebagian besar komponen pada

perangkat seluler ditata secara vertikal di layar. Berikutnya adalah flex. Atribut ini memberi tahu flexbox bagaimana ruang yang tersedia di sepanjang sumbu utama akan dibagi kepada anak-anak. Ini bisa menjadi sedikit rumit, tetapi mari kita mulai dengan bagian yang sederhana: wadah utama Anda hampir selalu memiliki nilai flex 1, yang berarti akan menempati seluruh tinggi layar (atau seluruh lebar, jika Anda menata dalam mode baris). Kemudian, anak-anak di dalam wadah itu akan membagi ruang yang tersedia menurut nilai flex mereka. Katakanlah Anda memiliki kode berikut:

```
Import React from “react”;
Import {View} from “react-native”

export default class App extends React.Component {
     render() {
         return (
          <View style = {{flex: 1}}>
           <View style= {{flex: 5, backgroundrColor: “red”}} />
           <View style= {{flex: 2, backgroundColor: “green”}}/>
           <View style= {{flex: 3, backgroundColor: “blue”}}/>
          </View
         );
        }
       }
```

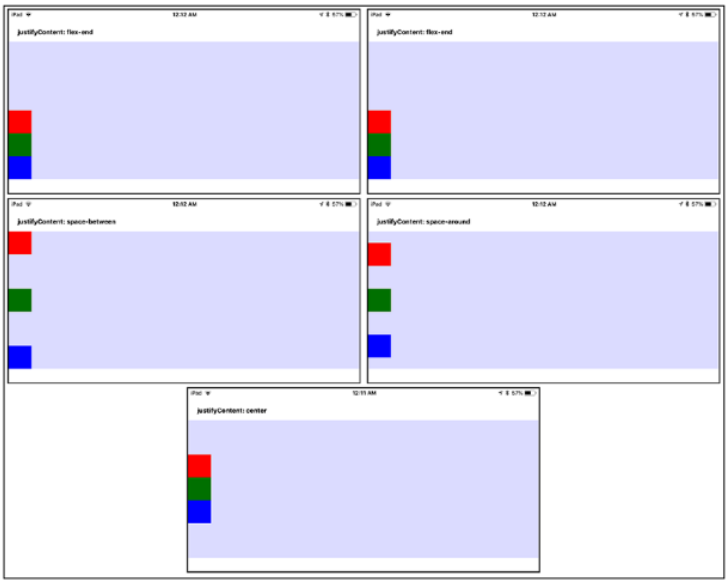
View pertama memiliki nilai flex 1, sehingga akan memenuhi layar. View tersebut memiliki tiga View anak, masing-masing dengan nilai flex yang berbeda. Berapa banyak ruang yang akan mereka gunakan? Nah, ketiganya jika digabungkan akan memenuhi View induk, yang berarti akan memenuhi layar, dan itu karena jika Anda tidak menentukan tinggi untuk komponen, View akan meregang untuk memenuhi ruang, berdasarkan nilai flex-nya, secara default. Namun, berapa banyak tinggi layar yang akan digunakan setiap View? Untuk mengetahuinya, Anda menjumlahkan nilai flex, sepuluh di sini, dan nilai flex masing-masing membentuk pecahan, menggunakan nilai tersebut sebagai penyebut.

Dengan kata lain, View anak pertama (merah) menempati 5/10 layar, yang berarti 1/2 (kurangi saja pecahan tersebut seperti yang Anda lakukan secara matematis). View anak kedua (hijau) menempati 2/10 layar, atau 1/5. View anak ketiga (biru) menempati 3/10 layar. Anggaplah sekarang kita mengubah nilai flex anak-anak menjadi 1, 2, dan 3, berturut-turut. Jumlahkan keduanya, dan kita dapatkan 6, jadi sekarang anak pertama menempati 1/5 layar, anak kedua 2/5, dan anak ketiga 3/5. Jika Anda menggunakan 4, 2, 4, sekarang masing-masing menempati 2/5, 1/5, dan 2/5. Lihat bagaimana cara kerjanya? Anda juga dapat mencampur nilai flex dengan nilai statis. Misalnya, ganti atribut flex anak pertama View dengan height:100 dan ubah nilai flex dua anak lainnya menjadi 1, dan Anda akan mendapatkan kotak merah setinggi 100 piksel, dan dua lainnya akan membagi ruang yang tersisa di antara keduanya secara merata.

Dalam kasus khusus ini, nilai flex yang sebenarnya Anda gunakan tidak penting, yang penting hanya keduanya sama. Saya harap Anda dapat melihat alasannya. Agar rangkaian ini

terus berjalan, selanjutnya kita akan membahas justifyContent (nilai yang valid: flex-start, flex-end, center, space-between, dan space-around, dengan flex-start sebagai nilai default). Atribut ini memberi tahu flexbox bagaimana anak-anak didistribusikan di sepanjang sumbu utama. Nilai flex-start berarti anak-anak akan "berkelompok" (artinya tanpa spasi di antara mereka) di bagian atas (atau kiri, tergantung pada sumbu utama) induknya. Nilai flex-end berarti kebalikannya; mereka akan berkumpul di kanan atau bawah.

Nilai center berarti mereka akan berkumpul di tengah wadah, dengan ruang yang tidak digunakan yang tersedia di atas dan di bawah (atau kiri dan kanan) anak-anak. Nilai space-between memastikan bahwa ruang yang tidak digunakan didistribusikan secara merata di antara anak-anak (tanpa spasi yang tidak digunakan sebelum dan sesudah anak pertama dan terakhir), dan space-around memastikan bahwa ruang yang tersedia dialokasikan di antara semua anak, termasuk sebelum dan sesudah anak pertama dan terakhir. Mereka mengatakan gambar bernilai seribu kata, jadi lihatlah Gambar 17.10, untuk melihat efek dari pengaturan ini dalam tindakan (ketika flexDirection kolom default digunakan).



Gambar 17.10 Berbagai Pengaturan Justifycontent

Berikutnya adalah atribut alignItems (nilai yang valid: flex-start, flex-end, center, dan stretch, dengan stretch sebagai nilai default). Ini menentukan bagaimana turunannya disejajarkan pada sumbu sekunder, atau sumbu "silang". Itu berarti bahwa, dalam kasus tata letak kolom default, alignItems menentukan bagaimana turunannya disejajarkan secara horizontal. Sekali lagi, mari kita lihat gambar, Gambar 4.2, untuk memperjelas pengaturan ini. Satu catatan penting adalah agar flex-start, center, dan flex-end berfungsi, turunan Anda harus memiliki lebar tertentu. Sebaliknya, turunan tersebut tidak dapat memiliki lebar tetap, jika Anda ingin menggunakan stretch. Atribut alignSelf (nilai yang valid: auto, flex-start, flex-end, center, stretch atau baseline, dengan auto sebagai nilai default) memungkinkan turunan untuk menentukan nilai alignItems-nya sendiri, dengan mengesampingkan nilai alignItems induknya.

Ini adalah atribut yang jarang digunakan, tetapi ada jika Anda membutuhkannya. Atribut flexWrap (nilai yang valid: wrap atau nowrap, dengan nowrap sebagai default) memberi tahu flexbox apakah anak-anak dipaksa masuk ke satu baris, bahkan jika itu berarti mereka akan mengalir keluar layar (nowrap) atau apakah mereka dapat berpindah ke baris kedua (wrap). Tentu saja, apakah sumbu utama Anda horizontal atau vertikal menentukan apakah Anda akan mendapatkan baris kedua anak-anak atau kolom kedua. Atribut alignContent (nilai yang valid: flex-start, center, flex-end, stretch, space-between, space-around, dengan flex-start sebagai nilai default) memungkinkan Anda untuk menyelaraskan baris anak saat flexWrap diatur untuk membungkus dan saat Anda berakhir dengan beberapa baris atau kolom. Misalnya, jika tata letak Anda mengalir untuk memiliki dua kolom anak, pengaturan alignContent ke tengah memastikan bahwa kedua kolom akan disejajarkan dengan bagian tengah wadah induk.

Dua atribut terakhir, position dan zIndex, berfungsi sebagaimana mestinya di Web: position dapat bersifat relatif atau absolut dan mengontrol apakah item memposisikan dirinya sendiri relatif terhadap saudara kandung sebelumnya (jika ada) atau apakah mereka diposisikan berdasarkan koordinat x/y absolut relatif terhadap sudut asal induk. Atribut zIndex memungkinkan Anda untuk menempatkan anak di atas yang lain (ketika position diatur ke absolut; jika tidak, tumpang tindih tidak akan terjadi saat position relatif). Secara default, position bersifat relatif, sama seperti di Web. Dengan informasi ini, Anda seharusnya dapat mencapai hampir semua tata letak di React Native yang Anda perlukan.

Namun, perhatikan bahwa ada lebih banyak atribut gaya terkait tata letak yang tersedia yang mungkin juga dapat Anda manfaatkan. Anda dapat melihatnya di sini (tetapi perhatikan bahwa dokumen React Native menyebutnya sebagai "Layout Props," meskipun pada akhirnya itu tetap saja merupakan atribut gaya): https://facebook.github.io/react-native/docs/layout-props.html. Catatan Bukan berarti penting untuk dapat melakukan tata letak dengan flexbox dan React Native, tetapi istilah justify berasal dari dunia percetakan (lihat, siapa bilang percetakan sudah mati?). Istilah ini merujuk pada cara baris-baris artikel surat kabar mengisi ruang yang tersedia untuk baris tersebut secara merata, sesuatu yang pada suatu saat sulit dilakukan, sebelum komputer hadir untuk membuatnya mudah. Hal ini menghasilkan tepi konten yang lurus di margin kiri dan kanan, yang dikatakan memberikan konten tata letak yang lebih teratur yang memungkinkan pembaca untuk bergerak lebih mudah di antara baris-baris teks.

Inti Masalah: DecisionScreen.js

Baiklah, setelah bagian intro selesai, saatnya untuk masuk ke kode yang terdapat dalam berkas DecisionScreen.js, yang merupakan sebagian besar kode Restaurant Chooser, tempat sebagian besar tindakan terjadi. Ingat di sini bahwa ada lima layar yang terdapat dalam berkas ini, atau sub-layar, seperti yang saya suka menyebutnya, tetapi bahkan sebelum itu, ada hal-hal klise yang biasa Anda ketahui dan sukai.

```
import React from “react”;
import CustomButton from “../components/CustomButton”;
import {Alert, AsyncStorage, BackHandler, Button, FlatList, Image, Modal,
```

```
Picker, ssPlatform, ScrollView, StyleSheet, Text, Touchable0pacity, View}
from “react-native”;
import {StackNavigator} from “react-navigation”;
import {CheckBox} from “native-base”;
import {Constans} from “expo”;
```

Kita tidak akan menggunakan komponen CustomTextInput di mana pun di sini, tetapi kita akan menggunakan CustomButton, jadi komponen tersebut diimpor. Semua komponen React Native adalah komponen yang sudah pernah Anda lihat dalam satu bentuk atau lainnya, jadi Anda seharusnya mengenalinya saat ini. StackNavigator dari React Navigation yang Anda lihat di layar Restaurants adalah cara kita beralih di antara berbagai sublayar. Komponen CheckBox dari NativeBase diimpor berikutnya, dan komponen ini akan digunakan di layar Who’s Going, untuk memilih orang-orang yang akan pergi makan.

Terakhir, Constants dari Expo akan digunakan sama seperti sebelumnya, yaitu untuk mendapatkan informasi ukuran header, sehingga beberapa padding dapat ditambahkan, jika perlu, seperti yang akan Anda lihat nanti.

Setelah impor, kita memiliki tiga variabel yang akan diperlukan dalam berkas sumber ini. Tentu saja, variabel-variabel tersebut bersifat global dalam modul ini, yang berarti bahwa semua kode sublayar akan dapat mengaksesnya, itulah sebabnya variabel-variabel tersebut didefinisikan di sini. Bila Anda harus berbagi data antar data dalam satu modul, ini adalah cara yang sangat baik untuk melakukannya. Kiat Meskipun cakupan global yang sebenarnya, di mana Anda dapat mengakses sesuatu dari modul mana pun, umumnya tidak disukai, itu adalah sesuatu yang dapat Anda lakukan di React Native, jika Anda harus melakukannya.

Ada variabel, yang namanya tidak terlalu kreatif, global, yang dapat Anda akses dari modul mana pun kapan saja, dan Anda dapat melampirkan data Anda sendiri ke dalamnya jika Anda mau. (Itu hanya objek JavaScript, jadi lampirkan atribut seperti yang Anda lakukan pada objek lainnya.) Namun, saya akan menyarankan Anda untuk menyimpan ini sebagai pilihan terakhir dan tidak menyalahgunakannya, jika Anda menggunakannya. Jika Anda harus melakukannya, lakukan dengan bijak, seperti melampirkan satu objek dengan nama yang Anda yakini unik (globalData_<nama aplikasi saya>, misalnya), untuk menghindari konflik dan masalah apa pun.

```
let participants = null;
let filteredRestaurants = null;
let chosenRestaurant = {};
```

Variabel peserta akan berisi serangkaian objek, satu untuk setiap orang yang akan berpartisipasi dalam pengambilan keputusan. Variabel filteredRestaurants akan berupa serangkaian objek, satu untuk setiap restoran yang mungkin dipilih secara acak oleh aplikasi. Sesuai namanya, daftar ini hanya akan berisi restoran yang lolos dari pilihan pra-filter yang dibuat oleh pengguna. Terakhir, chosenRestaurant persis seperti itu: objek dengan data tentang restoran yang dipilih secara acak. Ada satu bagian kode lagi sebelum kita masuk ke layar itu sendiri, yaitu fungsi pembantu kecil untuk memilih angka acak.

```
const getRandom = (inMin, inMax) => {
      inMin = Math.ceil(inMin);
      inMax = Math.floor(inMax);
      return Math.floor (Math.random() * (inMax – inMin + 1)) + inMin;
};
```

Ini cukup klise hanya generator angka acak biasa yang menerima nilai minimum dan nilai maksimum dan mengembalikan angka acak dalam rentang tersebut (inklusif). Karena mungkin harus dipanggil beberapa kali kemudian di dalam loop, masuk akal untuk mengekstrak kode ke dalam fungsi seperti ini. Sekarang, lanjut ke layar!

17.8 KOMPONEN DECISIONTIMESCREEN

Layar pertama yang harus dilihat adalah komponen DecisionTimeScreen, yang merupakan tempat pengguna memulai saat aplikasi pertama kali diluncurkan. Tidak ada apa-apa selain logo dan beberapa teks, yang semuanya dapat diketuk untuk memulai keputusan. Lihat kode di sini (dan, ya, ini semuanya).

```
class DecisionTimeScreen extends React.Component {

   render(){return(

      <View style={styles.decisionTimeScreenContainer}>
       <Touchable0Pacity style= {styles.decisionTimeScreenTouchable}
          onPress= {() => {

          AsyncStorage.getItem (“people”,
              function(inEror, inPeople) {
                if (inPeople === null) {
                   inPeople = [];
              } else {
                inPeople = JSON.parse (inPeople);
              }
             if (inPeople.length === 0){
                Alert.alert (“that ain’t gonna work, chief”,
                   “You haven’t added any people. “
                    “You should probably do that first, no?”,
                    [{text: “OK”} ], {cancellable : false}
                );
             }else
               AsyncStorage.getItem(“restaurants”,
                 function(inError, inRestaurants) {
                     if (inRestaurants === null) {
                      inRestaurants = [];
                    }else{
                     inRestaurants = JSON.parse (inRestaurants);
                    }
                    If (inRestaurants.length === 0) {
                     Alert.alert(“that ain’t gonna work, chief”,
```

```
                    “You haven’t added any restaurants. “+
                    “You should probably do that first, no?”,
                    [{text: “OK”}], {cancellable: false}
                  );
              }else{
          This.props.navigation.navigate(“WhosGoingScreen”);
              }
            }.bind(this)
        );
      }
    }.bind(this)
    );
    }}
    >
     <Image source= {require (“../images/its-decision-time.png)}/>
     <Text style= {{paddingTop:20}}>(click the food to get going)</Text>
    </Touchable0pacity>
  </View>
);}
```

Kita mulai dengan View kontainer, pola umum dalam React Native, dan pada View tersebut diterapkan gaya decisionTimeScreenContainer.

```
decisionTimeScreenContainer : { flex: 1, alignItems : “center”,
justifyContent: “center”}
```

Mengingat pembahasan yang mengawali bab ini, kini Anda tahu tentang apa ini. Ini adalah tata letak flexbox sederhana yang membuat View ini memenuhi seluruh layar dan memusatkan anak-anaknya baik secara vertikal maupun horizontal di atasnya. Pertama, View ini memiliki komponen TouchableOpacity. Anda akan ingat bahwa komponen ini adalah wadah generik, seperti View, tetapi yang merespons kejadian sentuh dan memungkinkan kita mengaitkan kode ke kejadian tersebut. Komponen ini juga mendapatkan gaya yang disertakan.

```
decisionTimeScreenTouchable:{alignItems:“center”,justifyContent: “center” }
```

Konfigurasi ini diperlukan, karena gaya pada View induk memusatkan komponen TouchableOpacity ini di dalamnya, tetapi anak-anak dalam TouchableOpacity juga harus dipusatkan. Jika tidak, Image dan Text tidak akan dipusatkan seperti yang kita harapkan. Yang terpenting, TouchableOpacity ini memiliki prop onPress yang terpasang, tetapi saya akan membahasnya sebentar lagi dan beralih ke bawah untuk melihat komponen Image yang merupakan anak pertamanya. Komponen Image merujuk pada gambar its-decision-time.png, menggunakan notasi jalur relatif, karena berada di direktori images, dan saat ini, konteks eksekusi file ini adalah direktori screens. Dengan kata lain, jika kita baru saja melakukannya

```
<Image source= {require(“its-decision-time.png”)}/>
```

maka React Native akan mencari file tersebut di direktori layar dan tentu saja tidak menemukannya, dan kita akan punya masalah. Anda harus selalu mempertimbangkan konteks eksekusi file sumber tempat kode Anda berada saat merujuk sumber daya yang merupakan bagian dari basis kode Anda, seperti ini. Tepat setelah komponen Gambar tersebut adalah komponen Teks yang memberi petunjuk kepada pengguna tentang apa yang harus dilakukan. Beberapa padding ditambahkan sebagai gaya sebaris, untuk memisahkan Gambar dari Teks. Saya membiarkannya sebagai gaya sebaris, pertama untuk mengingatkan Anda bahwa Anda dapat melakukannya, tetapi juga karena Anda harus selalu memutuskan apakah masuk akal untuk mengekstrak gaya Anda ke objek terpisah. Biasanya, memang demikian, tetapi terkadang gayanya sangat kecil dan secara konseptual terasa seolah-olah itu harus menjadi bagian dari objek yang sedang diberi gaya (dan itu bukanlah sesuatu yang ingin Anda ubah sama sekali atau secara global).

Ini adalah kasus di mana menambahkan gaya baru terasa agak berlebihan bagi saya, jadi saya tidak melakukannya. Sekarang, kembali ke pengendali onPress pada komponen TouchableOpacity. Tentu saja, di sinilah pekerjaan sebenarnya untuk layar ini, dan pekerjaan itu dimulai dengan menggunakan API AsyncStorage yang sekarang sudah cukup Anda kenal, untuk mengambil daftar orang yang diketahui aplikasi. Jika tidak ada, pengguna diberi tahu bahwa ia belum dapat membuat keputusan. Lagi pula, jika Anda sendiri tidak dapat memutuskan ke mana harus makan, maka tidak ada aplikasi di dunia ini yang akan membantu Anda! Tidak, aplikasi ini untuk membantu sekelompok orang membuat keputusan, dan dengan demikian, jelas harus ada orang yang dapat dipilih.

Dengan cara yang sama, jika pengguna belum membuat restoran apa pun, aplikasi juga tidak dapat melakukan apa pun, jadi restoran tersebut diambil berikutnya dari penyimpanan, dan jika tidak ada, pengguna juga diberi tahu tentang hal itu. Terakhir, jika setidaknya ada satu orang dan setidaknya satu restoran, kami memanggil metode this.props.navigation.navigate() dari React Navigation, dengan meneruskannya ke WhosGoingScreen, untuk menavigasi ke layar berikutnya, tempat pengguna dapat memilih siapa yang akan terlibat dalam pengambilan keputusan. Catatan Layar ini tidak perlu melakukan pekerjaan apa pun saat dibuat, itulah sebabnya tidak ada componentDidMount() di sini. Faktanya, sebagian besar sub-layar yang ada di file DecisionScreen.js seperti ini.

Ingat bahwa ini adalah metode opsional. Namun, ini adalah metode opsional yang dapat sangat berguna untuk membuat panggilan API jarak jauh awal guna mendapatkan nilai awal untuk mengisi layar atau melakukan segala jenis tugas pengaturan yang mungkin diperlukan. Anda akan melihat contohnya di seluruh buku ini, tetapi ini adalah salah satu metode yang kemungkinan besar akan sering Anda gunakan dalam pekerjaan Anda, jadi ingatlah ini.

17.9 KOMPONEN WHOSGOINGSCREEN

Setelah pengguna mengeklik grafik raksasa pada layar It’s Decision Time, layar berikutnya yang mereka lihat adalah layar Who’s Going, tempat mereka memilih orang-orang yang terlibat. Layar ini terdapat dalam komponen WhosGoingScreen (yang masih menjadi

bagian dari berkas DecisionScreen.js), dan dimulai seperti komponen React Native lainnya.

```
class WhosGoingScreen extends React.Component {

     constructor(inprops) {
      super(inProps);
      this.state= {people: [], selected : {} };
}
```

Properti diteruskan ke konstruktor superkelas, lalu objek status dilampirkan, untuk menyimpan status bagi komponen yang akan membentuk komponen ini. Kita memiliki dua informasi yang harus dilacak di sini: orang-orang yang dapat dipilih pengguna dan larik yang akan merekam orang-orang yang dipilih. Mungkin saja untuk menyimpan status yang dipilih itu di dalam objek-objek dalam larik people itu sendiri, tetapi saya merasa lebih mudah untuk tidak mengubah objek-objek itu, karena apakah mereka dipilih atau tidak adalah status sementara untuk layar ini, jadi memisahkan masalah-masalah itu tampaknya tepat. Larik people akan diisi dalam metode componentDidMount(), tetapi saya akan membahasnya setelah metode render(), dan berbicara tentang metode render(), ini dia sekarang:

```
      render() {return(
      <View style = {styles.listScreenContainer}>
      <Text style={styles.whosGoingHeadline}>whos's Going</Text>

      <FlatList style= {{width: "94%"}} data= {this.state.people}
        renderItem={({item}) =>
          <Touchable0pacity
            Style={styles.whosGoingItemTouchable}
          onPress={ function() {
            const selected = this.state.selected;
            selected[item.key= !selected[item.key];
            this.setState({ selected: selected});
            }.bind(this)}
        >
         <CheckBox style= {styles.whosGoingCheckBox}
            Checked={this.state.selected[item.key]}
          onPress= {function(){
            const selected = this.state.selected;
            selected[item.key]= !selected[item.key];
            this.setState ({selected:selected});
         }.bind(this)}/>
        <Text style= {style.whosGoingName}>
            {item.firstName}{item.lastName}({item.relationship})
        </Text>
      </Touchable0pacity>
     }
/>
```

Saya akan membagi diskusi ini menjadi dua bagian, bagian pertama dimulai dengan Tampilan

yang berisi seluruh layar dan menerapkan gaya ini:

```
listScreenContainer:{flex:1,alignItems:“center”, justifyContent :
“center”, ...Platform.select({ios:{paddingTop:Constants.statueBarHeight,
   android:{}})
}
```

Sesuai pembahasan sebelumnya tentang flexbox, Anda sekarang tahu bahwa atribut flex, alignItems, dan justifyContent bertanggung jawab untuk memastikan bahwa View mengisi layar dan anak-anaknya dipusatkan baik secara vertikal maupun horizontal. Ada juga kebutuhan untuk menambahkan beberapa padding di bagian atas, sehingga View tidak tumpang tindih dengan bilah status, dan di situlah penggunaan metode Platform.select() berperan, karena padding tersebut hanya diperlukan di iOS, bukan Android. Ketinggian bilah status diperoleh dengan Constants.statusBarHeight, seperti yang telah Anda lihat sebelumnya, dan itu menjadi nilai paddingTop. Anak pertama adalah komponen Text yang merupakan judul, atau tajuk utama, untuk layar. Ini adalah pola umum yang akan Anda lihat selanjutnya, dan agar teks tampak seperti judul, kita perlu menerapkan gaya berikut:

```
whosGoingHeadline:{fontSize:30, marginTop :20,marginBottom:20}
```

Ukuran font yang lebih besar memberi kita beberapa teks yang tampak seperti judul, dan beberapa margin di bagian atas dan bawah memastikan ada ruang di sekitarnya saat komponen lebih lanjut ditambahkan ke Tampilan wadah luar. Kebetulan, hanya ada dua anak lainnya, yang pertama adalah FlatList yang Anda lihat, dan, tentu saja, begitulah cara daftar orang ditampilkan. FlatList sendiri diberi lebar 94%, sehingga tidak membentur tepi layar, dan properti datanya menunjuk ke array people dalam status tersebut. Untuk setiap item, kami menyediakan properti renderItem yang merupakan fungsi yang akan dipanggil untuk merender setiap item.

Yang dirender di sini adalah komponen TouchableOpacity dan di dalamnya ada komponen CheckBox dari NativeBase dan komponen Text untuk menunjukkan nama. Keduanya bersarang di dalam TouchableOpacity, sehingga menyentuh teks akan memungkinkan kita untuk mencentang kotak juga. Jika tidak, pengguna harus mengetuk CheckBox secara khusus, membuatnya sedikit mengganggu untuk digunakan. Karena TouchableOpacity adalah wadah untuk komponen CheckBox dan Teks, kita perlu menatanya di dalamnya, jadi gaya ini digunakan:

```
whosGoingItemTouchable: {flexDirection: “row”, marginTop: 10,
marginBottom: 10}
```

Atribut flexDirection diatur ke baris, sehingga kedua komponen ditempatkan berdampingan, dan marginTop dan marginBottom memberikan ruang di antara setiap orang dalam daftar. Lihat properti onPress untuk TouchableOpacity dan CheckBox. Perhatikan sesuatu tentang keduanya? Ya, keduanya identik. Selain pertanyaan apakah ini harus atau tidak ditarik ke fungsi terpisah (mungkin ya, tetapi untuk sesuatu yang sepele seperti ini, menurut saya itu tidak

penting), keduanya identik, karena jika tidak, Anda akan menemukan bahwa menyentuh CheckBox tidak mengakibatkan perubahan status CheckBox. Secara konseptual, Anda akan membayangkan bahwa onPress dari TouchableOpacity adalah yang akan dipicu pada peristiwa sentuhan. Ini masuk akal, jika Anda membayangkannya dalam bentuk fisik. Bayangkan TouchableOpacity sebagai kotak plastik transparan, dan di dalam kotak itu ada Checkbox (mungkin Anda mencetak kotak centang sebenarnya dalam bentuk 3D).

Jika Anda mencoba menekan Checkbox, jari Anda sebenarnya akan menyentuh TouchableOpacity terlebih dahulu (dan sebenarnya tidak dapat menyentuh kotak centang sama sekali). Oleh karena itu, Anda akan meletakkan pengendali onPress pada TouchableOpacity, dan semuanya akan berjalan seperti yang Anda harapkan. Namun, bukan itu yang terjadi. Yang terjadi, atau setidaknya, cara kerjanya, adalah seolah-olah TouchableOpacity entah bagaimana mengetahui ada kotak centang di bawahnya dan mendelegasikannya ke pengendali onPress. Dalam versi fisik kami, seolah-olah kotak plastik secara ajaib memungkinkan jari Anda melewatinya untuk menyentuh kotak centang di bawahnya, tetapi hanya di tempat kotak centang berada. Jika Anda menekan di tempat lain pada kotak (atau pada TouchableOpacity), jari Anda tidak akan masuk, dan onPress dari TouchableOpacity aktif.

Sejujurnya, saya tidak yakin mengapa ini bekerja seperti ini. Saya tidak dapat menentukan jawabannya. Namun, pada akhirnya, solusi untuk masalah ini adalah dengan memasang pengendali yang sama ke TouchableOpacity dan CheckBox. Dengan begitu, pengguna dapat mengetuk di mana saja pada item dan mendapatkan efek yang diinginkan: mengaktifkan kotak centang. Pengendali itu sendiri mudah. Dapatkan referensi ke array yang dipilih dalam status (yang akan segera Anda lihat terisi), alihkan item dalam array yang dikaitkan dengan item yang diketuk pengguna berdasarkan atribut kuncinya, lalu lakukan setState(), untuk mencerminkan pembaruan. Karena properti CheckBox yang dicentang dikaitkan dengan entri dalam array yang dipilih untuk orang tersebut, status visual CheckBox diperbarui secara otomatis. Dan, berbicara tentang CheckBox itu, ia memiliki gaya sederhana yang diterapkan.

```
whosGoingCheckBox: {marginRight:20}
```

Hal ini dilakukan untuk memastikan bahwa ada spasi antara CheckBox dan nama orang tersebut, yang ditempatkan dalam komponen Teks setelah komponen CheckBox dan menerapkan gaya berikut:

```
whosGoingName: {flex:1}
```

Itu akan memaksa komponen Teks untuk mengisi ruang horizontal yang tersisa pada baris tersebut. Nilai komponen Uji hanyalah gabungan dari atribut firstName, lastName, dan relationship dari objek dalam array people. Satu hal yang saya abaikan adalah mengapa bind() terkadang digunakan pada event handler dan mengapa terkadang tidak. Jawaban sederhananya adalah saya melakukannya dengan kedua cara di berbagai tempat untuk

menunjukkan kepada Anda bahwa Anda sebenarnya dapat melakukannya dengan kedua cara.

Namun, Anda akan menemukan beberapa situasi di mana Anda tidak dapat menggunakan notasi fat arrow, atau Anda tidak akan memiliki referensi yang tepat ke komponen melalui this di dalam fungsi. Dalam kasus tersebut, Anda harus menggunakan notasi fungsi tradisional seperti yang dilakukan untuk handler onPress ini dan kemudian bind() fungsi tersebut ke komponen melalui this. Itu melengkapi kode FlatList. Berikutnya adalah tombol di bawah daftar tersebut.

```
      <CustomButton text=”Next” width=”94%”
         onPres={()=> {
           participants= [];
             for (const person of this.state.people){
              if(this.state.selected[person.key]){
              const participant= Object.assign({},person);
              participant.vetoed=”no”;
              participant.push(participant);
             }
        }
        If (participants.lengths === 0){
          Alert.alert(“Uhh, you awake?”
            “You didn’t select anyone to go , Wanna give it another try?”,
            [{text: “OK”}], {cancelabe:false}
      );
    }else{
      this.props.navigation.navigate(“PreFiltersScreen”);
     }
    }}
  />
  (/view>
  );}
```

Ini adalah prop onPress yang benar-benar kita pedulikan di sini dan yang bertanggung jawab untuk membuat array orang yang akan berpartisipasi dalam acara ini, maka dari itu nama variabelnya adalah peserta. Ini adalah masalah sederhana untuk mengulangi array orang dalam keadaan dan, untuk masing-masing, mencari dalam array yang dipilih untuk melihat apakah entri orang itu benar. Jika benar, orang itu disalin ke dalam array peserta, dan atribut yang diveto yang ditetapkan ke tidak ditambahkan (yang akan menjadi relevan pada layar berikut).

Sekarang, jika kita melakukan itu dan menemukan bahwa tidak ada entri dalam array peserta, maka, tentu saja, pengguna belum memilih apa pun, jadi peringatan muncul untuk memberi tahu mereka. Jika tidak, kita menggunakan metode React Navigation yang biasa untuk mentransfer pengguna ke layar Pre-Filters, yang akan kita lihat berikutnya. Namun, sebelum itu, kita memiliki satu bagian kode terakhir untuk dilihat pada layar ini, yaitu metode componentDidMount() yang saya janjikan akan segera Anda lihat.

```
componentDidMount() {
    BackHandler.addEventListener(“hardwareBackPress”,()=> {return true});
    AsyncStorage.getItem(“people”,

      if(inPeople === null) {
         inPeople= [];
      }else{
         inPeople = JSON.parse(inPeople);
      }
      const selected = {};
      for (const person of inPeople){selected[person.key] = false;}
      this.setState ({ people : inPeople,  selected : selected});
        }.bind(this)
     );
 };
```

Tugas di sini, karena metode ini aktif saat komponen dibuat (yang berarti saat layar ditampilkan), adalah mengisi daftar orang dalam status, menariknya dari AsyncStorage. Anda pernah melihat kode yang sama ini sebelumnya di layar Restaurants, meskipun kita memuat restoran di sana, tentu saja, jadi seharusnya terlihat familier. Setelah array tersebut dibuat, array yang dipilih kemudian dibuat, dengan entri untuk setiap orang dalam array inPeople, dengan nilai yang ditetapkan ke false, untuk menunjukkan bahwa mereka belum dipilih. Akhirnya, keduanya ditetapkan ke status, dan layar ini siap digunakan!

17.10 KOMPONEN PREFILTERSSCREEN

Layar Pre-Filters (yang merupakan komponen PreFiltersScreen) adalah layar yang dilihat pengguna setelah memilih siapa yang akan datang, dan memungkinkan mereka untuk memfilter restoran dari pertimbangan. Saya akan memecah kode menjadi beberapa bagian untuk Anda, agar sedikit lebih mudah dicerna, dimulai dengan konstruktor.

```
Class PreFilterScreen extends React.Component {

    Constructor(inProps) {
      super(inProps)
      this.state = {cuisine: “”, price : “”, rating : “”, delivery :“”};
    }
```

Tidak ada yang mengejutkan di sini. Cukup panggil konstruktor superkelas dan tentukan objek status. Ada empat kriteria yang dapat digunakan untuk memfilter restoran: jenis masakan, harga (kurang dari atau sama dengan), peringkat (lebih dari atau sama dengan), dan apakah restoran memiliki layanan pesan antar. Nilai-nilai tersebut direpresentasikan dalam objek status dan, tentu saja, akan ditetapkan oleh bidang entri data saat pengguna mengubahnya. Berikutnya adalah metode render().

```
render () {return (

   <ScrollView style={styles.preFiltersContainer}>
      <View style= {style.preFiltersInnerContainer}>
         <View style= {style.preFiltersScreenContainer}>

            <View style= {style.preFilterHeadlineContainer}>
                <Text style={styles.preFiltersHeadline}>Pre-Filters</Text>
            </View>
```

Pertama, seperti biasa, kita memiliki elemen kontainer, dalam hal ini, ScrollView. Ini agar kita memiliki komponen bergulir, karena komponen yang ditampilkan akan lebih panjang daripada layar perangkat apa pun yang ada saat ini, dan tanpa ini, pengguna tidak akan dapat menggulir ke bawah untuk melihat lebih banyak komponen. Gaya yang diterapkan pada komponen ini sederhana.

```
preFiltersContainer: {marginTop: Constants.statusBarHeight}
```

Sama seperti yang Anda lihat pada layar sebelumnya, sedikit ruang di bagian atas diperlukan, untuk menghindari tumpang tindih dengan bilah status. Sebuah View kemudian disarangkan di dalam View tersebut, yang dengannya kita dapat memperkenalkan beberapa tata letak. Meskipun dimungkinkan untuk menerapkan gaya yang diperlukan ke ScrollView itu sendiri, melakukannya sebagai anak dari ScrollView memberi kita kesempatan untuk memisahkan gaya di antara keduanya, memberi kita sedikit lebih banyak fleksibilitas jika layar ini suatu saat diperluas. Gaya yang diterapkan ke View anak ini juga cukup mendasar.

```
preFiltersInnerContainer: {flex:1, alignItems: “center”, paddingTop: 20,
width: “100%”}
```

Dengan flex:1, induknya akan terisi, dan anak-anaknya akan dipusatkan, berkat pengaturan alignItems. Sedikit tambahan padding ditambahkan. Ini diperlukan karena, saat pengguna menggulir, padding pada ScrollView akan bergulir keluar dari tampilan, dan item akan melapisi bilah status. Menambahkan padding di sini memastikan hal itu tidak terjadi. Terakhir, memberi View ini seluruh lebar tampilan memastikan bahwa kita memiliki ruang maksimum yang tersedia untuk mulai bekerja. Di dalam View ini terdapat View lain, wadah untuk kontrol entri data (atau "formulir" pra-filter kita). Ini dilakukan agar gaya berikut dapat diterapkan untuk menyediakan padding di kedua sisi layar.

```
preFiltersScreenFormContainer: {width: “96%”}
```

Di dalam Tampilan tingkat ketiga ini terdapat Tampilan lain, yang memiliki komponen Teks di dalamnya. Ini memberikan judul utama pada layar, sama seperti pada layar sebelumnya. Gaya untuk komponen Tampilan dan Teks ini, masing-masing, adalah

```
  preFiltersHeadlineContainer: {flex:1, alignItems: “center”,
  justifyContent: “center”}
  preFiltersHeadline: {fontSize: 30, marginTop: 20, marginBottom :20}
```

Gaya Tampilan diperlukan karena, secara default, wadah induknya, yang ditata dengan preFiltersScreenFormContainer, akan disejajarkan ke kiri (ingat, flex-start adalah default), tetapi kami menginginkannya di tengah, sehingga komponen Teks dibungkus dalam Tampilan dan perataan tengah ditambahkan padanya. Komponen Teks ditata dengan cara yang sama seperti judul pada layar sebelumnya. Sejauh ini, baik-baik saja. Sekarang, kita dapat mulai memasukkan komponen entri data untuk pemfilteran, dimulai dengan yang untuk jenis masakan.

```
<Text style= {styles.fieldLabel}>Cuisine</Text>
<View style= {styles.pickerContainer}>
    <Picker style={styles.picker) selectedValue={this.state.cuisine}
      prompt= “Cuisine”
      onValueChange={(inItemValue)=> this.setState({ cuisine:
      inItemValue})}>
           <Picker.Item label=”” value=””/>
           <Picker.Item label=”Algerian” value=”Algerian”/>
           <Picker.Item label=”American” value=”American”/>

           <Picker.Item label=”Other” value=”Other”/>
            ...
      </Picker>
</View>
```

Setiap bidang filter, pada kenyataannya, merupakan kombinasi dari komponen Teks, yang berfungsi sebagai label bidang, dan komponen entri data itu sendiri, Picker dalam kasus ini. Gaya yang diterapkan pada label (semuanya, bukan hanya yang ini) adalah

```
  fieldLabel: {marginLeft: 10}
```

Ini berfungsi untuk memberi sedikit ruang di sebelah kiri, sehingga label sejajar dengan tepi kotak Picker, seperti yang terlihat di layar Tambah Restoran. Picker itu sendiri dikaitkan dengan atribut masakan di objek status dan dibungkus dalam Tampilan dengan gaya yang diterapkan:

```
  pickerContainer: {
...Platform.select({
      ios: {},
      android: {width:”96%”, borderRadius: 8, borderColor: “#cOcOcO”,
      borderWidth: 2,
          marginLeft: 10, marginBottom: 20, marginTop: 4}
      })
   }
```

Anda melihat gaya yang sama sebelumnya di layar Tambah Restoran, jadi tidak perlu

membahasnya lebih lanjut di sini. Demikian pula, gaya pemilih yang diterapkan pada Pemilih itu sendiri sama seperti yang terlihat di layar Tambah Restoran, tetapi ini dia lagi, supaya Anda tidak perlu mempercayai apa yang saya katakan.

```
picker: {
...Platform.select({
      ios: {width:”96%”, borderRadius: 8, borderColor: “#cOcOcO”,
      borderWidth: 2,
          marginLeft: 10, marginBottom: 20, marginTop: 4}, android: {}
      })
   }
```

Perhatikan bahwa saya telah memangkas daftar elemen anak Picker.Item, hanya untuk menghemat sedikit ruang, tetapi semuanya ada di sana, seperti yang Anda harapkan. Penangan onChange cukup menyetel nilai baru Picker ke dalam objek status. Setelah Picker jenis masakan, muncul Picker untuk harga, peringkat, dan pengiriman. Namun, mengingat bahwa semuanya merupakan salinan langsung dari Picker jenis masakan, selain mendapatkan dan menyetel atribut harga, peringkat, dan pengiriman dari objek status, mari kita lewati semuanya. (Namun, ambil bundel kode unduhan, dan lihat semuanya, untuk memastikan.) Itu membawa kita ke tombol Berikutnya di bagian bawah layar yang diketuk pengguna saat mereka telah membuat pilihan pra-filter. Di situlah semua tindakan nyata untuk layar ini berada.

```
   <CustomButton text=”Next” width=”94%”
        onPress={() => {
      AsyncStorage.getItem (“restaurants”,
          function(inError, inRestaurants) {

if (inRestaurants === null) {
    inRestaurants= [];
}else{
    inRestaurants= JSON.parse(inRestaurants);
}
filteredRestaurants = [];
for (const restaurant of inRestaurants); {
    let passTests= true;
   if (this.state.cuisine !== “”) {
      if(Object.keys(this.state.cuisine).length>0) {
          if(restaurants.cuisine ! == this.state.cuisine) {
          passTests = false;
       }
    }
 }
if (this.state.price !== "") {
    if (restaurant.price>this.state.price) {passTests = false;}
}
if (this.state.rating !== "") {
```

```
if (restaurant.rating < this.state.rating) {
    passTests = false;
}
if (this.state.delivery !== "") {
    if (restaurant.delivery !== this.state.delivery) {
        passTests = false;}
}
if (this.state.cuisine.length === 0 && this.state.price === "" &&
    this.state.rating === "" && this.state.delivery === "") {
    passTests = true;
}
if (passTests) {filteredRestaurants.push(restaurant);}
}
if (filteredRestaurants.length === 0) {
    Alert.alert("Well, that's an easy choice",
        "None of your restaurants match these criteria. Maybe " +
        "try loosening them up a bit?",
        [{text: "OK"}], {cancelable: false }
    );
     } else {
         This.props.navigation.navigate(“ChoiceScreen”);
       }
   }.bind(this)
  );
}}
/>
</View>
</View>
</scrollView>

);}
```

Tentu saja, onPress handler adalah tempatnya, dan pekerjaan di sana dimulai dengan menarik daftar restoran dari AsyncStorage, seperti yang telah Anda lihat beberapa kali sebelumnya di berbagai tempat. Setelah kita memasukkannya ke dalam array inRestaurants, langkah berikutnya adalah membuat array filteredRestaurants yang kosong. Ingat bahwa ini adalah variabel global untuk modul ini, jadi kita hanya memastikannya sebagai array kosong pada titik ini. (Sebenarnya tidak boleh ada cara agar tidak kosong, tetapi sedikit pemrograman defensif tidak akan merugikan siapa pun.) Selanjutnya, kita mengulang daftar restoran yang diambil. Untuk masing-masing, tanda passTests ditetapkan ke true, jadi kita akan berasumsi, sebagai permulaan, bahwa setiap restoran disertakan dalam array akhir.

Kemudian pengujian dilakukan, berdasarkan kriteria filter yang dipilih, jika ada. Setiap bagian diperiksa untuk mengetahui apakah ada yang kosong, yang menunjukkan bahwa pengguna tidak menetapkan nilai untuk kriteria tertentu, dan untuk bagian yang tidak kosong, logika yang sesuai dijalankan dan passTests ditetapkan ke false untuk bagian yang gagal. Pada akhirnya, jika passTests bernilai true, restoran tersebut ditambahkan ke array filteredRestaurants. Terakhir, jika setelah iterasi tersebut array tersebut kosong, kami memberi tahu pengguna bahwa tidak ada yang dapat dilakukan aplikasi dan memperingatkan mereka

untuk mengubah kriteria pra-filter. Jika ada setidaknya satu, kami mengarahkan pengguna ke layar Choice, yang merupakan potongan kode berikutnya untuk kami lihat.

Komponen ChoiceScreen

Berikutnya dalam rangkaian kode yang populer adalah komponen ChoiceScreen. Di sinilah aplikasi memilih restoran dan menunjukkannya kepada pengguna. Layar ini menggunakan komponen Modal, jendela dialog pop-up, untuk menunjukkan restoran yang dipilih. Layar ini juga menggunakan Modal ketika seseorang dalam kelompok memutuskan untuk memveto pilihan tersebut, dan kedua Modal ini merupakan bagian dari kode ini. Sebelum kita membahasnya, mari kita lihat bagaimana komponen ini dimulai. Saat ini, ini bukan hal baru.

```
Class ChoiceScreen extends React.Component {

   Constructor (inProps) {
       Super(inProps);
            This.state = { participantsList: participants,
            participantsListRefresh : false,
            selectedVisible: false,vetoVisible: false, vetoDisabled: false,
          vetoText: “Veto”
      };
    }
```

Ya, cukup panggil konstruktor superkelas dan objek status seperti biasa. Atribut objek status ini adalah:

- ParticipantsList: Daftar orang yang berpartisipasi dalam keputusan. Ini digunakan untuk mencantumkan orang-orang di layar utama (bagian yang tidak ada dalam Modal) dan menunjukkan apakah ada yang telah melakukan veto.
- ParticipantsListRefresh: Tanda Boolean yang diperlukan agar daftar orang diperbarui setelah veto. (Jangan khawatir, saya akan menjelaskannya saat saya membahas kode untuk daftar tersebut.)
- SelectedVisible: Boolean yang memberi tahu React Native apakah
- Modal yang menampilkan restoran yang dipilih terlihat.
- VetoVisible: Boolean yang, seperti selectedVisible, memberi tahu React Native apakah Modal tempat pengguna memilih orang yang melakukan veto terlihat atau tidak.
- VetoDisabled: Boolean yang menentukan apakah tombol Veto pada Modal restoran yang dipilih dinonaktifkan atau tidak. (Jika tidak ada yang tersisa yang dapat memveto, tombol tersebut harus dinonaktifkan.)
- VetoText: Ini berisi teks untuk tombol Veto, yang akan diubah menjadi "No Vetoes Left" ketika tidak ada yang tersisa yang dapat memveto. (Ini lebih baik daripada hanya menonaktifkan tombol, karena dengan cara ini, pengguna tidak bertanya-tanya mengapa tombol tersebut dinonaktifkan.)

Sekarang, mari kita bahas metode render(). Saya akan menguraikannya menjadi beberapa

bagian kecil untuk kesenangan Anda dalam mengonsumsi kode, dimulai dengan bagian ini:

```
render(){return (

      <View style={styles.listScreenContainer}>
```

Seperti biasa, kita memiliki elemen kontainer, dan, seperti biasa, ini adalah komponen View. Gaya yang diterapkan sama dengan View kontainer pada layar Who's Going, jadi Anda dapat merujuk ke bagian tersebut, jika Anda tidak ingat. Setelah itu muncul komponen Modal pertama dari dua komponen. Berikut ini digunakan untuk menampilkan informasi tentang restoran yang dipilih secara acak:

```
<Modal presentationStyle={"formSheet"} visible={this.state.selectedVisible}

    animationType={"slide"} onRequestClose={() => { }} >
    <View style={styles.selectedContainer}>
        <View style={styles.selectedInnerContainer}>
            <Text style={styles.selectedName}>{chosenRestaurant.name}</Text>
            <View style={styles.selectedDetails}>
                <Text style={styles.selectedDetailsLine}>
                    This is a {"\u2605".repeat(chosenRestaurant.rating)} star
                </Text>
                <Text style={styles.selectedDetailsLine}>
                    {chosenRestaurant.cuisine} restaurant
                </Text>
                <Text style={styles.selectedDetailsLine}>
                    With a price rating of
{"$".repeat(chosenRestaurant.price)}
                </Text>
                   <Text style={style.selectedDetailsLine}>
      That {chosenRestaurants.delivery === "Yes"? "DOES" : "DOES" NOT"}
deliever
   </Text>
</View>
<CustomButton text="Accept" width= "94%"
  onPress={()=> {
        this.setState({ selectedVisible : false, vetoVisible:
        false});
      this.props.navigation.navigate("PostChoiceScreen");
      }}
      />
      <CustomButton text={this.state.vetoText} width="94%"
        disabled={this.state.vetoDisabled ? "true": "false"}
          onPress={() => {
          this.setState({ selectedVisible: false, vetoVisible: true});
      }}
      />
  </View>
 </View>
</Modal>
```

Properti presentation Style digunakan untuk mengontrol bagaimana Modal muncul. Untuk sebagian besar, ini hanya akan memiliki efek nyata pada perangkat yang lebih besar, seperti iPad, karena, pada perangkat lain, Modal akan muncul dalam layar penuh, terlepas dari pengaturannya (atau hanya dengan sedikit perbedaan visual). Pengaturan formSheet adalah salah satu dari empat yang menampilkan Modal sebagai tampilan lebar sempit yang dipusatkan pada layar. Pengaturan lainnya adalah:

- Fullscreen: Saya harap makna pengaturan ini jelas.
- Pagesheet: Ini sebagian menutupi tampilan yang mendasarinya dan dipusatkan.
- Overfullscreen: Ini sama dengan fullscreen, tetapi memungkinkan transparansi.

Properti visible untuk Modal ini terikat pada atribut selectedVisible dalam objek state, jadi cara untuk menampilkan Modal adalah dengan mengubah nilai tersebut dalam state. Ini juga berlaku untuk Modal lainnya, menggunakan atribut vetoVisible. Atribut animationType menentukan jenis animasi yang digunakan untuk menampilkan Modal. Nilai slide menyebabkan Modal meluncur dari bawah (nilai fade menyebabkannya memudar ke tampilan, dan nilai none menyebabkannya muncul tanpa animasi, dan ini adalah nilai default).

Properti onRequestClose memungkinkan Anda untuk mengeksekusi beberapa kode saat Modal ditutup. Dalam kasus ini, kita tidak memerlukan apa pun untuk terjadi; namun, properti ini diperlukan, dan kita akan mendapatkan peringatan YellowBox, jika kita tidak menyediakannya, maka fungsi ini kosong. (Kesalahan dan peringatan dibahas di bagian "Debugging dan Pemecahan Masalah" dari bab ini, jadi istilah peringatan YellowBox akan dibahas. Singkatnya, ini adalah peringatan yang muncul di layar saat Anda menjalankan aplikasi, dan, ya, dalam bentuk kotak kuning). Di dalam Modal, kita memulai kontennya dengan View, dengan gaya ini diterapkan:

```
selectedContainer: {flex: 1, justifyContent: “center”}
```

Modal sama seperti hal lain yang Anda lakukan di React Native, yaitu Anda harus menyediakan satu komponen, yang tentu saja dapat memiliki komponen anak, jadi ini adalah komponen tingkat atas. Komponen ini akan mengisi Modal dan memusatkan komponen anak-anaknya secara horizontal. Sumbu tata letak utama adalah vertikal, karena tata letak default adalah kolom, ingat. Di dalam Tampilan tingkat atas ini terdapat Tampilan lain, dan di sini kita menerapkan gaya untuk memusatkan komponen anak secara vertikal.

```
selectedInnerContainer: {alignItems: “center”}
```

Dalam Tampilan ini, pertama-tama terdapat komponen y yang nilainya adalah atribut nama dari objek chosenRestaurant. Kami ingin ini ditulis dalam teks besar, jadi gaya ini digunakan:

```
selectedName: {fontSize: 32}
```

Setelah itu muncul komponen View lainnya, yang ini dengan gaya yang diterapkan:

```
selectedDetails: {paddingTop: 80, paddingBottom: 80, alignItems:
“center”}
```

Hal ini memastikan bahwa ada cukup ruang di atas dan di bawah detail restoran, yang merupakan turunan tengah dari Tampilan ini. Setiap baris detail tersebut merupakan komponen Teks yang terpisah, dan karena saya ingin teksnya lebih besar dari biasanya, tetapi tidak sebesar nama restoran, gaya ini digunakan pada setiap:

```
selectedDetailsLine: {fontSize: 18}
```

Keempat komponen Teks kini memiliki beberapa hal menarik. Komponen Teks pertama memiliki nilai berikut:

```
This is a {“\u2605”.repeat(chosenRestaurant.rating)} star
```

Pertama, Anda dapat melihat bahwa Anda dapat menggunakan kode karakter Unicode dalam string, seperti yang telah saya lakukan, untuk menampilkan karakter bintang. Karena string dalam JavaScript memiliki metode repeat(), saya menggunakannya untuk menampilkan jumlah karakter bintang yang sesuai, berdasarkan atribut chosenRestaurant.rating. Komponen Teks ketiga melakukan hal yang sama untuk harga restoran, tetapi tidak perlu nilai Unicode di sini, karena tanda dolar sudah tersedia (meskipun, jika aplikasi ini diinternasionalkan dengan benar, kita mungkin menggunakan Unicode untuk menampilkan simbol denominasi yang sesuai untuk negara tempat perangkat berada).

Komponen Teks keempat menyertakan sedikit logika terner, untuk menampilkan baik BENAR-BENAR atau TIDAK BENAR-BENAR mengirim, sebagaimana ditentukan oleh nilai atribut pengiriman chosenRestaurant. Modal ini kemudian berisi dua komponen CustomButton, yang pertama digunakan saat pengguna menerima restoran ini, yang terakhir saat seseorang ingin memveto pilihan tersebut. Untuk tombol Terima, pengendali peristiwa onPress memperbarui atribut selectedVisible dan vetoVisible menjadi false dalam objek status, yang menyebabkan React Native menyembunyikan kedua Modal tersebut. (Ingat bahwa keduanya ada, baik saat ini terlihat atau tidak.) Kemudian, pengendali tersebut menavigasi aplikasi ke layar Post-Choice, yang dibahas nanti dalam bab ini.

CustomButton kedua mendapatkan teks labelnya dari atribut vetoText dari objek status dan menerima nilai dari prop yang dinonaktifkan dari atribut vetoDisabled dalam status. Anda akan melihat kode yang menetapkan nilai tersebut nanti, tetapi intinya adalah bahwa keduanya harus dinamis, sehingga mengikatnya ke atribut status. Pengendali onPress cukup menyembunyikan Modal ini dan menampilkan Modal berikutnya (untuk veto) dengan mengubah status. Modal kedua sedikit lebih rumit, dan kodenya adalah sebagai berikut:

```
<Modal presentationStyle={"formSheet"} visible={this.state.vetoVisible}
animationType={"slide"} onRequestClose={ () => { } } >
  <View style={styles.vetoContainer}>
    <View style={styles.vetoContainerInner}>
```

```
      <Text style={styles.vetoHeadline}>Who's vetoing?</Text>
      <ScrollView style={styles.vetoScrollViewContainer}>
        {participants.map((inValue) => {
          if (inValue.vetoed === "no") {
              return <Touchable0pacity key={inValue.key}
            style={styles.vetoParticipantContainer}
              onPress={() => {
              for (const participant of participants) {
                if (participant.key === inValue.key) {
                  participant.vetoed = "yes";
                  break;
                }
              }
            let vetoStillAvailable = false;
            let buttonLabel = "No Vetoes Left";
            for (const participant of participants) {
             if (participant.vetoed === "no") {
                    vetoStillAvailable = true;
                    buttonLabel = "Veto";
             break;
             }
            }

      for (let i = 0; i < filteredRestaurants.length; i++) {
     if (filteredRestaurants[i].key ===
    chosenRestaurant.key){

        filteredRestaurants.splice(i, 1);
        break;
    }
  }
      this.setState({ selectedVisible: false,
      vetoVisible: false,
        vetoText: buttonLabel,vetoDisabled:
        !vetoStillAvailable,
       participantsListRefresh:!this.state.
       participantsListRefresh
      });
      if (filteredRestaurants.length === 1) {
       this.props.navigation.navigate("PostChoiceScreen")
    }
  }}
  >
      <Text style={styles.vetoParticipantName}>
       {inValue.firstName + " " + inValue.lastName}
      </Text>
      </TouchableOpacity>;
      }
   }}
  }
<ScrollView>
  <View style={styles.vetoButtonContainer}>
```

```
    <CustomButton text="Never Mind" width="94%"
      onPress={() => {
        this.setState({ selectedVisible: true, vetoVisible: false });
      }}
    />
      </View>
   </View>
  </View>
</modal>
```

Modal itu sendiri didefinisikan seperti yang sebelumnya, jadi tidak ada yang baru untuk dilihat di sana. Modal juga dimulai seperti yang lain, dalam hal anak-anaknya, dengan View tingkat atas yang berfungsi sebagai wadah, dengan gaya berikut di atasnya:

```
vetoContainer: {flex: 1, justifyContent: “center”}
```

Hal ini memiliki tujuan yang sama dengan anak tingkat atas Modal sebelumnya, yaitu untuk mengisi Modal dan memusatkan anak-anaknya di sepanjang sumbu tata letak utama. Selain itu, seperti Modal sebelumnya, Tampilan kedua disarangkan di dalam yang pertama, sehingga kita dapat menerapkan beberapa konfigurasi tata letak lebih lanjut, seperti yang dapat Anda lihat dalam gaya ini:

```
vetoContainerInner: {justifyContent: “center”, alignItems : “center”
alignItems: “center”}
```

Kali ini, kami ingin anak-anak dipusatkan pada kedua sumbu, oleh karena itu pengaturan flexbox yang Anda lihat. Anak-anak tersebut dimulai dengan komponen Teks judul, menggunakan definisi gaya ini:

```
vetoHeadline: {fontsize: 32, fontweight: “bold”}
```

Ya, sama seperti pada Modal pertama. Sejauh ini, tidak banyak perbedaan, tetapi itu berubah dengan komponen anak berikutnya, yaitu ScrollView. Tujuannya di sini adalah untuk menyajikan daftar orang, daftar yang sama seperti yang terlihat di layar utama di bawah Modal ini, yaitu daftar orang yang berpartisipasi, lalu memungkinkan pengguna untuk mengetuk satu untuk menunjukkan bahwa mereka menolak. Karena daftar ini bisa sangat panjang, kita memerlukan area yang dapat digulir, dan ScrollView sederhana bisa menjadi solusinya. ScrollView ini menggunakan gaya berikut:

```
vetoScrollViewContainer: {height: “50%”}
```

Itu hanya ketinggian sembarangan yang saya tentukan melalui coba-coba yang sebagian besar mengisi area yang tersedia di Modal (setelah judul dan tombol dipertimbangkan). Nah, di sinilah hal yang menarik: ScrollView tentu saja harus memiliki anak, tetapi bagaimana Anda

mengambil array (peserta, dalam kasus ini) dan membuat daftar anak tersebut secara dinamis? Nah, salah satu cara untuk melakukannya adalah dengan menggunakan metode map() yang tersedia pada array JavaScript. Metode ini memungkinkan Anda mengambil setiap elemen array, menjalankannya melalui suatu fungsi, dan mengembalikan sesuatu. Dalam kasus ini, yang akan kita kembalikan adalah beberapa konfigurasi komponen React Native lama yang bagus.

Dengan membungkus panggilan map() dalam kurung kurawal, JSX mengetahui bahwa ini adalah suatu ekspresi, dan keluaran dari ekspresi tersebut akan disisipkan sebagai pengganti ekspresi tersebut. Dalam kasus ini, ekspresi tersebut adalah hasil dari menjalankan fungsi yang disediakan sekali untuk setiap anggota array. Oleh karena itu, kita berakhir dengan satu atau beberapa elemen anak untuk ScrollView. Apa yang dijalankan oleh fungsi map() untuk setiap item dalam array yang dikembalikan? Sebagai elemen tingkat atas, ia mengembalikan TouchableOpacity, yang pernah Anda lihat sebelumnya. Namun, di sini, Anda akan melihat bahwa ia memiliki properti key, yang nilainya diambil dari atribut key inValue, yang merupakan objek untuk orang berikutnya dalam array peserta.

Nilai key tersebut sebenarnya tidak diperlukan untuk melakukan pekerjaan dalam Modal ini, tetapi tanpanya, Anda akan mendapatkan peringatan bahwa setiap item dalam iterator harus memiliki key. Oleh karena itu, kita memiliki properti key, meskipun itu tidak diperlukan. Untuk setiap item dalam array peserta, kita memeriksa atribut yang diveto. Jika tidak, orang ini masih memiliki hak veto dan, oleh karena itu, akan dimasukkan dalam daftar. Jika tidak, orang tersebut tidak akan dimasukkan. Setelah kita menentukan bahwa orang tersebut akan dimasukkan, TouchableOpacity didefinisikan, dengan key dan gaya berikut:

```
vetoParticipantContainer: {paddingTop: 20, paddingBottom: 20}
```

Ini menyisipkan beberapa spasi di atas dan di bawah nama setiap orang dalam daftar. Ini juga berarti bahwa target sentuh bagi pengguna memiliki tinggi 40 piksel yang nyaman, jadi sebagian besar pengguna tidak akan mengalami masalah saat mengetuk nama yang benar dan tidak mengetuk nama lain secara tidak sengaja. onPress dari TouchableOpacity adalah tempat kerja yang sebenarnya terjadi, dan itu dimulai dengan menandai orang yang diketuk sebagai orang yang telah memveto. Ini berarti mengulangi melalui array peserta hingga kita menemukan item dengan kunci yang cocok dengan item yang diketuk dan menyetel atribut yang diveto ke ya.

Perhatikan bahwa itu disetel ke ya atau tidak, bukan Boolean benar atau salah, yang merupakan hal yang wajar Anda harapkan untuk disetel, untuk alasan yang sangat bagus, yang akan segera terlihat. Setelah langkah itu, kita harus melihat apakah masih ada orang yang dapat memveto. Ini dilakukan agar tombol Veto dapat dinonaktifkan saat tidak ada orang yang tersisa untuk memveto, serta mengubah teks labelnya ke string No Vetoes Left yang lebih tepat. Setelah itu, saatnya untuk menghapus restoran yang diveto dari pertimbangan. Ingat bahwa kita menyalin objek restoran ke array filteredRestaurants baru setelah layar Pre-Filter, sehingga kita dapat melakukan penghapusan langsung dari array, menggunakan metode

splice(), tanpa takut mengubah data permanen apa pun.

Array ini dan datanya hanya digunakan selama proses pengambilan keputusan, jadi jangan khawatir. Sebagai langkah kedua terakhir, kita harus memperbarui objek status untuk mencerminkan semua pekerjaan ini. Itu berarti menyetel selectedVisible ke false, untuk memastikan bahwa Modal disembunyikan (sebenarnya sudah disembunyikan, tetapi sekali lagi, sedikit pemrograman defensif bukanlah hal yang buruk) dan juga atribut vetoVisible. Label untuk tombol Veto disetel melalui atribut vetoText ke nilai buttonLabel yang ditentukan sebelumnya. Atribut vetoDisabled adalah kebalikan dari nilai variabel vetoStillAvailable, yang juga ditetapkan pada langkah sebelumnya.

Terakhir, kita memiliki atribut ParticipantsListRefresh yang diaktifkan. Apa maksudnya? Nah, untuk menjelaskannya, kita perlu melihat daftar orang di layar Pilihan utama, yang belum kita bahas, jadi mari kita tunda pembahasan itu sejenak. Namun, perlu diingat bahwa nilainya diaktifkan, apa pun yang terjadi. Nilainya berubah, dan itulah yang terpenting. Sebelum kita membahasnya, kita harus melihat komponen Test yang merupakan anak dari TouchableOpacity dan memiliki gaya berikut yang diterapkan:

```
vetoParticipanName: {fontSize: 24}
```

Nilai yang ditampilkan adalah gabungan atribut firstName dan lastName dari partisipan saat ini yang sedang dirender (seperti yang diteruskan ke fungsi yang diberikan ke map(), melalui argumen inValue). Komponen Text tersebut mengakhiri ScrollView, dan hanya menyisakan satu CustomButton untuk ditangani, yaitu tombol Never Mind yang memungkinkan pengguna untuk membatalkan veto, jika tombol Veto pada Selected Modal tidak sengaja ditekan. Ini harus menyetel selectedVisible ke true dan vetoVisible ke false dalam status untuk menampilkan kembali Selected Modal dan menyembunyikan Veto Modal. Omong-omong, CustomButton ini harus dipusatkan dan mengambil (hampir) seluruh lebar Modal, sehingga View yang memuatnya mendapatkan gaya ini:

```
vetoButtonContainer: {width: “100%”, alignItems: “center”, paddingTop: 40}
```

Sekarang, dengan Modal kedua yang telah dibahas, kita dapat membicarakan tentang layar Pilihan, yang merupakan apa yang Anda lihat saat tidak ada Modal yang ditampilkan (dan Anda melihat sebagiannya saat Modal ditampilkan juga, meskipun, hanya sebagian saja jika digunakan pada perangkat berlayar lebar).

```
<text style={styles.choiceSecreenHeadline}>Choice Screen</Text>
```

Pertama, kita punya judul lain, Teks, seperti yang sudah Anda lihat beberapa kali sebelumnya, dan gayanya juga sama seperti yang sudah Anda lihat sebelumnya, jadi mari kita bahas hal yang lebih menarik, yaitu, komponen FlatList yang muncul berikutnya.

```
<FlatList style={styles.choiceScreenListContainer}
          data={this.state.participantsList}
          extraData= {this.state.participantsListRefresh}

          rendetItem= {({item}) =>
         <View style={styles.choiceScreenListName}>
          <Text style={styles.choiceScreenListItemName}>
           {item.firstName} {item.lastName} ({item.relationship})
           </Text>
           <Text>Vetoed: {item.vetoed}</Text>
     </View>
    }
 />
```

Ini, tentu saja, bertanggung jawab untuk mencantumkan peserta dalam keputusan ini. Atribut data dikaitkan dengan atribut ParticipantsList dari objek status, dan diberi gaya berikut:

```
choiceScreenListContainer: {width: "94%"}
```

Anda sudah tahu aturannya sekarang: 94% memberi ruang di kedua sisi, karena anak dari wadah induk dipusatkan. Kemudian, properti renderItem adalah fungsi yang mengembalikan komponen untuk setiap item dalam larik data. Komponen tingkat atas yang dikembalikan fungsi ini adalah Tampilan, dengan gaya berikut diterapkan:

```
choiceScreenListItem: {flexDirection: "row", marginTop: 4, marginBottom: 4,
borderColor: "#e0e0e0", borderBottomWidth: 2, alignItems: "center"}
```

Di sini, setiap baris dalam FlatList akan terdiri dari dua komponen Teks, jadi kita harus menggunakan flexDirection baris untuk menempatkannya berdampingan. Ada beberapa ruang di bagian atas dan bawah, jadi item dalam FlatList tidak terlalu berdekatan (pilihan estetika), lalu batas abu-abu muda diletakkan tepat di bagian bawah setiap item (sekali lagi, hanya pilihan estetika). Dua komponen Teks tersebut adalah, pertama, nama dan hubungan orang tersebut, dan yang kedua adalah apakah mereka telah memveto. Komponen Teks pertama menggunakan gaya ini:

```
choiceScreenListItemName: {flex:1}
```

Itu untuk memastikan bahwa nama akan mengisi ruang apa pun yang tersedia (yang akan menjadi sebagian besar baris, karena komponen Teks kedua akan selalu memiliki lebar kecil). Berbicara tentang komponen Teks kedua itu, itu menjawab pertanyaan sebelumnya tentang mengapa atribut yang ditolak ditetapkan ke ya atau tidak dan bukan benar atau salah: itu ditampilkan secara harfiah di sini, dan ya atau tidak lebih ramah pengguna (dan itulah sebabnya itu akan selalu menjadi lebar kecil). Jika Anda telah memperhatikan, Anda pasti akan bertanya, Hei, tunggu sebentar, bagaimana dengan prop extraData itu? Nah, di situlah atribut status ParticipantsListRefresh yang saya lewati sebelumnya berperan.

Jadi, begini masalahnya: ketika React Native melihat nilai prop ini berubah, ia akan merender ulang daftar, terlepas dari apakah datanya berubah. Itu penting, karena ketika seseorang memveto pilihan restoran, kami memperbarui atribut yang diveto dari objek tersebut dalam array ParticipantsList di objek status, tetapi terkadang React Native tidak dapat melihat perubahan pada data dalam status ketika perubahan dilakukan pada atributnya. Jika Anda kembali dan melihat kode di tombol Veto, Anda akan melihat bahwa panggilan ke setState() tidak menyertakan pengaturan ParticipantsList. Melakukan hal itu tidak akan menyebabkan React Native melihat perubahan pada atribut yang diveto.

Pikirkan seperti ini: React Native hebat dalam memperhatikan perubahan pada atribut status yang secara langsung merupakan atribut status, tetapi tidak selalu hebat dalam memperhatikan perubahan pada atribut objek yang merupakan bagian dari koleksi yang secara langsung merupakan atribut status. Arahkan state.participantsList ke array yang sama sekali baru dalam kode pengendali onPress tombol Veto? React Native akan memperhatikannya dan merender ulang daftar tersebut. Ubah atribut objek di dalam array yang sudah ditunjuk oleh state.participantsList? React Native tidak akan memperhatikannya. Jadi, Anda harus memberikan sedikit dorongan, dengan kata lain, dengan prop extraData. Tidak masalah apa yang Anda simpan di prop, asalkan berubah. Itu cukup untuk memaksa React Native merender ulang daftar, dan itulah yang kita perlukan di sini. Terakhir, setelah FlatList, kita memiliki CustomButton sederhana yang memicu aplikasi untuk memilih restoran secara acak.

```
      <CustomButton text="Randomly Choose" width="94%"
         onPress={() => {
            const selectedNumber = getRandom(0, filteredRestaurants.length -
1);
            chosenRestaurant = filteredRestaurants[selectedNumber];
            this.setState({ selectedVisible: true });
        }}
      />
    </View>
);}
```

Akhirnya, kita dapat melihat di mana fungsi getRandom() dari awal Bab 3 berperan. Sebuah angka acak dipilih, lalu objek yang dikaitkan dengan indeks tersebut dalam array filteredRestaurants disimpan ke chosenRestaurant (salah satu variabel global modul dari sebelumnya, ingat), lalu panggilan setState() dilakukan, menyetel selectedVisible ke true, untuk menunjukkan Modal tersebut. Dan hanya itu yang ada di layar Choice dan Modal terkaitnya! Kita hanya memiliki satu layar lagi untuk dilihat, dan, tentu saja, itulah yang dilihat pengguna setelah menerima restoran ini.

Komponen PostChoiceScreen

Layar terakhir adalah layar Post-Choice, yang dilihat pengguna setelah menerima pilihan. Layar ini, selain layar It's Decision Time awal, bersifat langsung dan tidak berisi apa pun yang belum pernah Anda temui sebelumnya. Berikut kode untuk layar ini:

```
  Class PostChoiceScreen extends React.Component {
      constructor(inprops) {super(inProps);}
```

Konstruktor hanya memanggil konstruktor superkelas, meneruskan properti yang diteruskan kepadanya. Jika Anda kembali dan melihat layar It's Decision Time, Anda akan melihat bahwa tidak ada konstruktor. Saya melakukan ini dengan sengaja, untuk menunjukkan bahwa, secara tegas, Anda tidak harus memiliki konstruktor, dan Anda tidak harus meneruskan properti ke konstruktor superkelas. Namun, untuk lebih jelasnya, Anda hampir selalu harus melakukannya. Hanya karena kode layar It's Decision Time tidak menangani properti maka ia berfungsi (dan itu adalah layar yang sangat sederhana), dan hal yang sama berlaku di sini. Perhatikan bahwa kedua komponen kustom itu sama; mereka tidak memiliki konstruktor yang meneruskan properti ke konstruktor superkelas, namun semuanya berfungsi seperti yang diharapkan.

Namun karena React Native melakukan berbagai hal atas nama Anda, hal-hal yang mungkin tidak Anda sadari, dan mengingat bahwa pada suatu saat Anda mungkin mencoba menggunakan prop dan menemukan bahwa berbagai hal tidak berfungsi seperti yang Anda harapkan, akan selalu lebih aman untuk memiliki konstruktor dan, paling tidak, meneruskan prop tersebut ke konstruktor superkelas. Anda sering kali dapat menghindari hal ini, tetapi Anda mungkin menghadapi masalah di kemudian hari yang mungkin sulit diatasi, jadi saya sarankan untuk membiasakan diri selalu melakukan hal sebelumnya. Setelah konstruktor muncul metode render()

```
render() {return (
     <View style={styles.postChoiceScreenContainer}>

        <View><Text style={styles.postChoiceHeadline}>Enjoy your
meal!</Text></View>

      <View style={styles.postChoiceDetailsContainer}>

      <View style={styles.postChoiceDetailsRowContainer}>
     <Text style={styles.postChoiceDetailsLabel}>Name:</Text>
<Textstyle={styles.postChoiceDetailsValue}>{chosenRestaurant.name}</Text>

   </View>
      <View style={styles.postChoiceDetailsRowContainer}>
  <Text style={styles.postChoiceDetailsLabel}>Cuisine:</Text>
<Textstyle={styles.postChoiceDetailsValue}>{chosenRestaurant.cuisine}</Text>
  </View>
 <View style={styles.postChoiceDetailsRowContainer}>
    <Text style={styles.postChoiceDetailsLabel}>Price:</Text>
    <Text style={styles.postChoiceDetailsValue}>
        {"$".repeat(chosenRestaurant.price)}
</Text>

<View style={styles.postChoiceDetailsRowContainer}>
  <Text style={styles.postChoiceDetailsLabel}>Rating:</Text>
  <Text style={styles.postChoiceDetailsValue}>
```

```
    {"\u2605".repeat(chosenRestaurant.rating)}
  </Text>
</View>

<View style={styles.postChoiceDetailsRowContainer}>
  <Text style={styles.postChoiceDetailsLabel}>Phone:</Text>
  <Textstyle={styles.postChoiceDetailsValue}>{chosenRestaurant.phone}</Text>
</View>

<View style={styles.postChoiceDetailsRowContainer}>
  <Text style={styles.postChoiceDetailsLabel}>Address:</Text>
  <Text
style={styles.postChoiceDetailsValue}>{chosenRestaurant.adress}</Text>
</View>

 <View style={styles.postChoiceDetailsRowContainer}>
  <Text style={styles.postChoiceDetailsLabel}>Address:</Text>
  <Text
style={styles.postChoiceDetailsValue}>{chosenRestaurant.address}</Text>
</View>

<View style={styles.postChoiceDetailsRowContainer}>
  <Text style={styles.postChoiceDetailsLabel}>Web Site:</Text>
  <Text
style={styles.postChoiceDetailsValue}>{chosenRestaurant.website}</Text>

</View>

      <View style={styles.postChoiceDetailsRowContainer}>
        <Text style={styles.postChoiceDetailsLabel}>Delivery:</Text>
        <Text
      style={styles.postChoiceDetailsValue}>{chosenRestaurant.delivery}</Text
      >
       </View>
  </View>

  <View style={{ paddingTop: 80 }}>
  <Button title="All Done"
       onPress={() =>
      this.props.navigation.navigate("DecisionTimeScreen")}
 />
  </View>
</View>

);}
```

Ya, seluruh layar ini, pada umumnya, hanyalah serangkaian komponen Teks, yang digunakan untuk menampilkan detail tentang restoran. Dimulai dengan Tampilan kontainer, seperti yang dilakukan hampir setiap komponen React Native, yang menerapkan gaya berikut:

```
  postChoiceScreenContainer: {flex: 1, justifyContent: "center",
```

```
alignItems: "center",
alignContent: "center"}
```

Itu seharusnya terlihat sangat familiar bagi Anda sekarang, karena sama dengan yang digunakan pada layar It's Decision Time, dan untuk tujuan yang sama: mengisi seluruh layar (flex:1) dan memusatkan semua anak secara horizontal dan vertikal. Anak pertama dari View ini adalah komponen Text yang merupakan judul atau tajuk utama layar, dan menggunakan gaya berikut:

```
postChoiceHeadline: {fontSize: 32, paddingBottom: 80}
```

Jelas, idenya di sini adalah membuat teks lebih besar dan memastikan ada ruang di bawah judul, antara judul dan kotak yang berisi detail restoran. Berbicara tentang kotak itu, untuk itulah komponen View berikutnya setelah judul, yang menerapkan gaya berikut:

```
postChoiceDetailsContainer: {borderWidth:2, borderColor:"#000000", padding:
10, width: "96%"}
```

Ini memberi kita batas sisi hitam pekat dua piksel dan memastikan bahwa ada sepuluh piksel bantalan antara batas dan apa pun yang ada di dalam kotak. Saya juga memberinya lebar tidak cukup 100%, untuk memastikan bahwa ada ruang antara tepi layar dan kotak, hanya karena saya pikir itu terlihat lebih baik. Di dalam Tampilan itu terdapat serangkaian Tampilan lain, masing-masing berisi beberapa informasi tentang restoran. Tujuannya di sini adalah untuk memastikan bahwa jumlah ruang yang digunakan label bidang konsisten, terlepas dari label itu sendiri. Kedengarannya sangat mirip dengan tata letak yang memiliki beberapa kolom, dua, lebih tepatnya, dengan yang pertama berisi label dan yang kedua berisi data aktual. Untuk mencapai ini, masing-masing komponen Tampilan memiliki gaya berikut:

```
postChoiceDetailsRowContainer: {flexDirection:"row",justifyContent: "flex-
start", alignItems: "flex-start", alignContent: "flex-start"}
```

Menetapkan flexDirection ke baris akan menata anak-anak dalam satu baris, yang mencapai tujuan. Di sini, kita ingin konten setiap anak disejajarkan ke kiri, sehingga semua label sejajar (label akan "mengambang" jika kita memusatkannya dan tampak tidak sejajar dengan benar), jadi itulah mengapa flex-start digunakan untuk justifyContent, alignItems, dan alignContent. Sekarang, di dalam setiap komponen View ini terdapat dua komponen Text, satu untuk label dan satu untuk data bidang (nama restoran, jenis masakan, dll.). Komponen Text pertama memiliki gaya berikut:

```
postChoiceDetailsLabel: {width: 70, fontWeight: "bold", color: "#ff0000"}
```

Hal ini membuat kolom label, bisa dikatakan, memiliki lebar spesifik 70 piksel dan labelnya berwarna merah dan dicetak tebal. Lebarnya diatur sedemikian rupa sehingga terlepas dari

lebar teks label yang sebenarnya, komponen data setelah label akan semuanya sejajar dengan benar. (Membiarkan label berubah ukuran secara dinamis akan membuat data bergeser ke kiri dan kanan dan tidak sejajar dengan benar.) Terakhir, kita memiliki komponen Teks data yang sebenarnya, dengan masing-masing merujuk ke properti objek chosenRestaurant yang sebelumnya diisi untuk memberikan nilai yang akan ditampilkan. Komponen ini memiliki gaya sederhana yang diterapkan.

```
postChoiceDetailsValue: {width:300}
```

Itu menghindari pembungkusan nilai yang lebih panjang. Nilai-nilai itu akan terpotong sekarang. Namun, lebar ini cukup untuk menampung nilai-nilai "realistis" apa pun yang dapat saya pikirkan. Hanya ada satu bagian kode terakhir dalam berkas sumber ini, yaitu konfigurasi StackNavigator.

```
const DecisionScreen = StackNavigator(
    {DecisionTimeScreen: {screen: DecisionTimeScreen},
     WhosGoingScreen: {screen: WhosGoingScreen},
     PreFiltersScreen: {screen: PreFiltersScreen},
     ChoiceScreen: {screen: ChoiceScreen},
     PostChoiceScreen: {screen: PostChoiceScreen}
  },
  {headerMode: "none"}
);
```

Sama seperti yang Anda lihat pada kode layar Restoran, kita harus memberi tahu StackNavigator layar apa yang dikontrol tumpukan ini dan memberinya referensi ke komponen-komponennya. Selain itu, seperti pada layar Restoran, kita tidak menginginkan header, jadi, sekali lagi, headerMode ditetapkan ke none. Setelah itu, kita harus mengekspor StackNavigator, karena ini adalah komponen tingkat atas.

```
exports.DecisionScreen = DecisionScreen;
```

Dan selesai! Layar ini juga tidak memerlukan metode componentDidMount(), jadi dengan itu, Restaurant Chooser sekarang menjadi aplikasi yang lengkap, dan Anda telah menjelajahi semua kode yang menyusunnya. Bukankah itu menyenangkan?

Debugging dan Pemecahan Masalah

Menulis kode, tentu saja, hanya sebagian dari persamaan. Debugging kode tersebut adalah bagian besar lainnya dan sesuatu yang belum banyak saya bicarakan. Oh, yang pasti, Anda telah melihat bahwa Anda dapat menggunakan metode console.* untuk mengeluarkan pesan ke konsol tempat Anda menjalankan server Expo (sebagian besar metode yang mungkin Anda kenal dari pengembangan web bekerja sama dengan React Native), dan memang itu dapat bermanfaat. Namun, itu bukanlah satu-satunya fasilitas debugging saat bekerja dengan React Native dan Expo.

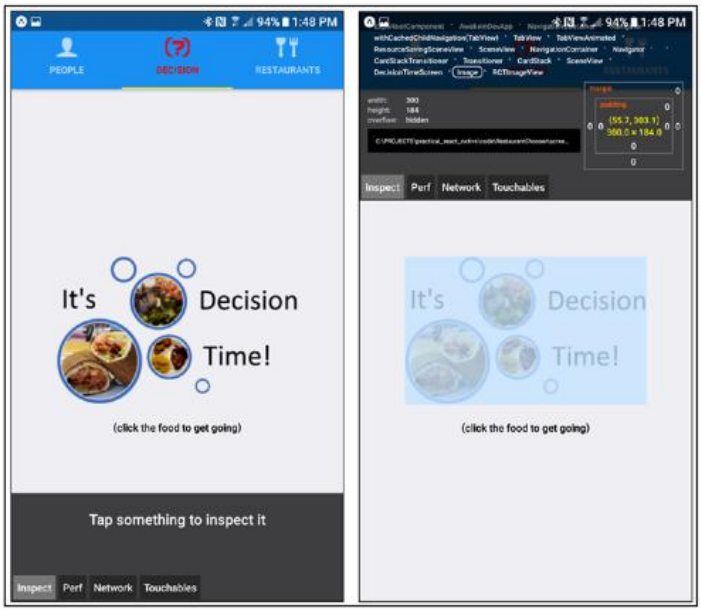
React Native secara otomatis menawarkan menu pengembang di dalam aplikasi Anda. Secara default, ini diakses dengan menggoyangkan perangkat Anda tetapi berhati-hatilah! Saya tidak ingin mendengar tentang pembaca yang menggoyangkannya terlalu keras dan menghancurkan ponsel pintar yang sangat mahal ke dinding bata. Anda dapat mengubah pemicu menu di aplikasi klien Expo, tetapi menu tersebut akan bergetar secara default. Saat Anda melakukannya, Anda akan melihat menu yang ditunjukkan pada Gambar 4.3.

Gambar 17.12 Menu Pengembang

Di sini, Anda memiliki beberapa opsi. Pertama, Anda dapat memuat ulang aplikasi dari server Expo. Kita akan melewatkan opsi Debug JS Remotely sebentar dan langsung beralih ke Disable Live Reload. Saat Anda membuat perubahan pada kode, dengan asumsi Anda telah membuka aplikasi di perangkat, aplikasi akan dimuat ulang secara otomatis (terkadang memerlukan waktu satu atau dua detik), dan opsi ini memungkinkan Anda untuk menonaktifkannya (berubah menjadi Enable Live Reload, jika Anda menonaktifkannya, sehingga berfungsi sebagai tombol alih).

Mirip dengan pemuatan ulang langsung, tetapi berbeda, adalah Enable Hot Reloading. Hot reload memungkinkan Anda untuk tetap menjalankan aplikasi saat versi baru file Anda dimasukkan ke dalam bundel JavaScript secara otomatis. Ini akan memungkinkan Anda untuk mempertahankan status aplikasi melalui pemuatan ulang. Opsi Toggle Inspector adalah opsi berikutnya, dan mengetuknya akan mengarah ke layar yang terlihat pada Gambar 17.13 (Pertama, Anda akan melihat layar di sebelah kiri, lalu saat Anda membuat pilihan, Anda akan

melihat layar di sebelah kanan.)



Gambar 17.13 Alat Inspector

Bila Anda memilih opsi ini, Anda akan dapat memilih elemen dalam aplikasi, seperti yang telah saya lakukan di sini, untuk gambar yang dapat diklik pada halaman awal, yang ditunjukkan dengan disorot. Anda akan melihat banyak informasi tentangnya di bagian atas, termasuk tempatnya dalam hierarki komponen (elemen yang dipilih adalah yang memiliki batas). Anda dapat melihat hal-hal seperti model kotaknya dan file tempat kodenya berada. Anda juga dapat memilih tab (Periksa, Perf, Jaringan, dan Yang Dapat Disentuh), untuk melihat informasi lebih lanjut tentang aplikasi Anda, termasuk hal-hal seperti statistik kinerja (Perf), permintaan yang masuk antara aplikasi Expo pada perangkat dan server Expo pada mesin pengembangan Anda (Jaringan), dan objek yang dapat disentuh yang tersedia pada saat itu (Yang Dapat Disentuh). Perhatikan bahwa Anda juga dapat mengeklik nama komponen dalam hierarki di bagian atas, untuk naik atau turun hierarki tersebut, sesuai kebutuhan. Alat lain yang tersedia untuk Anda adalah Perf Monitor, alat untuk memantau kinerja aplikasi Anda. Anda dapat melihatnya pada Gambar 17.14

Gambar 17.14 Alat Perf Monitor

Alat ini menunjukkan frame rate aplikasi Anda, berapa banyak gangguan visual yang terjadi, dll. Ini diperbarui secara real time, sehingga Anda dapat memantaunya saat menavigasi aplikasi dan menemukan titik masalah untuk diselidiki lebih lanjut. Sekarang, mengetahui tentang alat-alat tersebut sangatlah baik, tetapi apa yang terjadi ketika kesalahan terjadi pada kode Anda? Dalam kasus tersebut, React Native memiliki dua cara untuk melaporkan masalah: halaman kesalahan RedBox dan peringatan YellowBox. Lihat Gambar 17.15 untuk contoh halaman kesalahan RedBox.

Di sini, di bagian atas halaman, Anda dapat melihat apa yang memicu kesalahan (dalam kasus ini, saya telah mencoba memanggil metode AsyncStorage API yang tidak ada), dan Anda dapat melihat informasi jejak tumpukan, untuk membantu Anda menentukan masalahnya. Di bagian bawah, Anda memiliki beberapa opsi. Anda dapat mengabaikan kesalahan sepenuhnya, dalam hal ini aplikasi Anda mungkin berfungsi dengan baik atau tidak, tentu saja, atau Anda dapat memuat ulang aplikasi. Anda juga dapat menyalin informasi kesalahan, jika Anda membutuhkannya di tempat lain (waktu StackOverflow, mungkin?).

Gambar 17.15 Halaman Kesalahan Redbox

Di sisi lain, peringatan biasanya tidak cukup penting untuk menghentikan aplikasi Anda, tetapi itu adalah hal-hal yang perlu Anda ketahui. Gambar 17.16 menunjukkan seperti apa bentuknya. Anda kemudian dapat mengeklik peringatan untuk melihat pesan lengkapnya dalam bentuk YellowBox layar penuh, dan Anda juga akan memiliki akses ke jejak tumpukan di sana. Anda juga akan memiliki tiga tombol: Minimalkan (kembali ke layar dengan peringatan di bagian bawah), Abaikan peringatan (peringatan akan hilang), dan Abaikan Semua (jika ada lebih dari satu peringatan, seperti di sini). Perhatikan bahwa layar RedBox dan YellowBox ini dinonaktifkan dalam versi rilis (yang akan saya bahas di bagian berikutnya).

Topik terakhir yang terkait dengan debugging yang ingin saya bahas adalah opsi yang saya lewati sebelumnya saat membahas menu pengembang: Debug JS dari Jarak Jauh. Di sinilah semuanya menjadi sangat keren! Jika Anda menekan opsi itu, dengan asumsi Anda telah memasang Google Chrome di mesin pengembangan Anda, Anda akan menemukan bahwa tab baru terbuka di browser (konfigurasi yang diperlukan telah dilakukan untuk Anda oleh Expo), dan pada tab itu, Anda dapat membuka Alat Pengembang Chrome dan menggunakannya untuk men-debug aplikasi yang berjalan di perangkat Anda. Anda dapat mengatur breakpoint, memeriksa variabel, dan sebagainya, seperti halnya men-debug kode JavaScript lainnya di Chrome. Ini adalah cara yang elegan untuk memecahkan masalah kode Anda selama pengembangan yang akan membantu Anda, dan Anda akan mendapatkan sedikit detail lebih lanjut tentangnya.

Gambar 17.16 Peringatan YellowBox

Tips Ada beberapa kemampuan debugging lainnya yang tersedia bagi Anda sebagai pengembang React Native, tetapi saya menganggap yang dijelaskan di sini sebagai cara utama. Namun, jika Anda ingin melihat yang lainnya, lihat dokumen React Native di sini: https://facebook.github.io/react-native/docs/debugging.html.

Membungkus Semuanya

Di Bab sebelumnya, penulis menunjukkan kepada Anda cara menggunakan Expo untuk mengembangkan aplikasi, dengan memulai server Expo di mesin pengembangan Anda, lalu menggunakan klien Expo untuk menguji aplikasi. Saya harap Anda melakukan hal yang sama untuk aplikasi Komponen di Bab 2 dan Pemilih Restoran. Itu fantastis untuk pengembangan—dapat menjalankan aplikasi di perangkat nyata dengan mudah adalah hal yang luar biasa saat Anda mengutak-atik kode. Tetapi bagaimana jika Anda ingin menunjukkan aplikasi kepada orang lain? Jika Anda hanya menggunakan server Expo, mereka harus dapat mengakses mesin tempat server tersebut berada, dan itu mungkin tidak terlalu praktis, karena Anda harus terus menjalankan server Expo.

Catatan Menariknya, secara teori, orang dapat mengakses server Expo Anda, meskipun mereka tidak berada di jaringan area lokal Anda. Expo menggunakan domain unik, exp.direct, untuk tunneling. Ini memungkinkan siapa saja yang mengetahui URL aplikasi (yang ditampilkan di konsol saat Anda memulai server Expo) untuk mengakses Anda, meskipun Anda berada di balik firewall pada jaringan privat virtual. Saya perhatikan bahwa ini adalah kasus "secara teori", karena jika Anda harus menentukan URL pengemas seperti yang dibahas di Bab 1, tunneling ini tidak akan berfungsi. Selain itu, hal-hal lain dapat salah yang akan membuat mesin Anda tidak dapat diakses. Jika berhasil, itu bagus, tetapi saran saya jangan mengandalkannya dan, sebaliknya, carilah untuk menerbitkan kapan dan jika Anda perlu

membagikan aplikasi Anda dengan orang lain.

Dan itu bahkan belum mempertimbangkan langkah berikutnya: menerbitkan aplikasi ke Google Play Store atau Apple App Store. Ada dua jalur yang dapat Anda ambil untuk mendapatkan aplikasi Anda di komputer orang lain. Pertama, Anda dapat menerbitkan aplikasi melalui Expo, atau Anda dapat membuat paket asli untuk iOS dan/atau Android lalu mendistribusikannya (atau mengirimkannya ke toko). Mari kita bahas tentang penerbitan terlebih dahulu. Penerbitan, dalam konteks ini, berarti membuat aplikasi menjadi publik melalui situs web expo.io dan, dengan demikian, tersedia untuk orang lain. Penerbitan cukup mudah, tetapi pertama-tama, Anda harus membuat akun di expo.io dan masuk ke akun dengan memasukkan yang berikut:

```
exp login
```

Setelah selesai, yang harus Anda lakukan untuk menerbitkan adalah menjalankan perintah ini di dalam direktori aplikasi Anda, sama seperti saat Anda memulai aplikasi Anda:

exp publikasikan

Ini akan memicu proses (ini akan memakan waktu, jadi bersabarlah) yang akan mengambil kode sumber Anda, mengecilkannya, dan memanipulasinya, sebagaimana diperlukan, dan akan menghasilkan dua versi kode Anda, satu untuk iOS dan satu untuk Android. Di akhir proses, Anda akan diberikan URL yang kemudian dapat digunakan di aplikasi klien Expo untuk meluncurkan aplikasi Anda, sepenuhnya terlepas dari apakah server Expo berjalan di komputer Anda. Namun, aplikasi Anda tidak akan bersifat publik pada tahap ini, yang berarti seseorang harus mengetahui URL untuk mengaksesnya. Jika Anda ingin membuatnya tersedia untuk seluruh dunia, masuk ke akun Expo Anda di browser di expo.io dan klik tautan Lihat Profil. Di sana, Anda akan menemukan daftar proyek yang telah Anda terbitkan dan beberapa opsi untuk memanipulasi masing-masing, termasuk membuat satu proyek menjadi publik.

Penerbitan sangat bagus untuk memungkinkan orang melihat karya Anda, tetapi mengharuskan mereka memasang aplikasi klien Expo. Itu mungkin baik-baik saja selama pengembangan dan pengujian, tetapi, jelas, Anda tidak ingin memaksa pengguna untuk menggunakan aplikasi Anda secara nyata. Tidak, Anda hampir pasti ingin mereka masuk ke toko aplikasi untuk iOS dan Android sebagai aplikasi yang berdiri sendiri. Itu juga sangat mudah, berkat Expo! Pertama, Anda harus memastikan bahwa nilai-nilai dalam app.json sudah benar untuk membangun aplikasi, yang berarti memastikan bahwa Anda memiliki nilai untuk bundleIdentifier, name, icon, version, slug, dan sdkVersion. Opsi lain yang tersedia bersifat opsional, tetapi semuanya wajib, seperti yang dibahas dalam Bab 3. Dengan asumsi Anda sudah siap dengan file tersebut, seperti halnya Restaurant Chooser, yang harus Anda lakukan adalah menjalankan salah satu dari dua perintah, tergantung pada platform yang ingin Anda gunakan. Ini adalah

`exp build:android` atau `exp build:ios`

Anda akan ditanya satu atau dua pertanyaan, yang akan bervariasi, berdasarkan platform yang Anda targetkan. Untuk Android, Anda akan ditanya apakah Anda ingin

mengunggah keystore Anda sendiri atau menggunakan yang disediakan oleh Expo (yang digunakan untuk menandatangani paket aplikasi akhir secara digital). Kecuali Anda tahu apa yang Anda lakukan, saya sarankan agar Expo menanganinya untuk Anda. Jangan khawatir, jika Anda berubah pikiran nanti, Anda dapat menghapus keystore Anda saat ini dengan menjalankan

```
Exp Build:Android ClearCredentials
```

Kemudian Anda akan dapat mengunggah milik Anda sendiri, jika Anda mau. Untuk iOS, Anda akan ditanya pertanyaan serupa mengenai kredensial dan sertifikat distribusi (yang memiliki tujuan mendasar yang sama dengan keystore Android), dan Anda akan kembali memiliki pilihan untuk menanganinya sendiri atau membiarkan Expo mengerjakannya untuk Anda. Setelah itu, kode Anda akan diunggah ke infrastruktur cloud Expo, yang berisi semua perkakas yang diperlukan untuk membangun aplikasi Anda. Apakah Anda menyadari bahwa saya tidak menyebutkan sama sekali tentang keharusan memasang SDK iOS atau Android, tidak ada IDE seperti Xcode atau Android Studio, dan tidak ada persyaratan untuk melakukan apa pun pada satu OS vs.

OS lainnya? Semua itu tidak diperlukan saat Anda menggunakan Expo untuk melakukan pembangunan untuk Anda. Itulah salah satu cara menggunakan Expo yang membuat hidup Anda jauh lebih mudah sebagai pengembang. Sekarang, proses pembangunan ini akan memakan waktu yang cukup lama 15 menit bukanlah hal yang luar biasa, menurut pengalaman saya, tetapi biasanya akan memakan waktu sekitar 5 menit. Berapa pun lamanya waktu yang dibutuhkan, setelah selesai, Anda akan diperlihatkan URL yang sesuai dengan file IPA iOS atau file APK Android, dan Anda akan dapat mengunduh file di URL tersebut. Atau, jika Anda masuk ke akun Expo Anda di browser, Anda akan menemukan tautan Lihat pembangunan IPA/APK tempat Anda dapat mengunduhnya (jadi Anda tidak perlu mengingat URL tersebut).

Catatan Bagi pengguna Windows, Anda harus menginstal Windows Subsystem for Linux (WSL) agar build dapat berfungsi. Sebaiknya Anda menginstal Ubuntu dari Windows Store. Selain itu, Anda harus meluncurkan Ubuntu setidaknya satu kali sebelum mencoba melakukan build. Sekarang, mengunduh file hanyalah separuh dari perjuangan. Setelah itu, Anda harus memasukkannya ke perangkat (atau mungkin emulator atau simulator pada mesin pengembangan, karena itu adalah sesuatu yang benar-benar dapat Anda lakukan, jika Anda mau). Untuk Android, caranya mudah: Anda dapat menyalin file APK ke perangkat atau emulator dengan cara apa pun yang biasa Anda gunakan untuk menyalin file ke sana. Itu mungkin berarti menyalin APK ke berbagi jaringan lalu mengakses berbagi itu dari pengelola file di perangkat, atau mungkin mengirimkannya melalui Bluetooth, atau mungkin menggunakan perintah ADB (Android Developer Bridge) untuk menginstal atau mengirimkannya.

(Namun, ADB adalah bagian dari Android SDK, jadi Anda harus menginstalnya untuk menggunakan metode itu.) Anda tentu saja dapat mengakses URL yang sesuai dan mengunduh langsung dari sana, atau Anda dapat menggunakan cara lama dan mengirim file

itu ke diri Anda sendiri melalui email. Apa pun metodenya, setelah APK ada di perangkat, Anda harus memastikan bahwa Anda telah mengaktifkan opsi pengembang untuk mengizinkan instalasi aplikasi dari sumber yang tidak dikenal. Lokasi opsi ini berbeda-beda di setiap perangkat, tetapi biasanya ada di bawah opsi Keamanan di Setelan—dan, tentu saja, Google adalah teman Anda. Setelah menemukan dan menyetel opsi tersebut, Anda dapat "menjalankan" berkas tersebut. Itu akan memicu prosedur penginstalan Android yang biasa, dan tak lama kemudian, aplikasi akan siap digunakan.

Catatan Jika Anda telah menyiapkan emulator Android, penginstalan ke emulator tersebut semudah menyeret dan meletakkan berkas APK ke emulator tersebut. Jika itu tidak berhasil, Anda selalu dapat menginstalnya melalui ADB (dan jika Anda telah menyiapkan emulator, maka kemungkinan besar Anda telah menyiapkan alat pengembangan Android dan memiliki ADB). Untuk iOS, semuanya sedikit lebih rumit. Jika Anda kebetulan telah menginstal Xcode, itu berarti Anda memiliki simulator iOS yang siap digunakan. Untuk menjalankannya di Simulator iOS Anda, pertama-tama buat aplikasi Anda, dengan menambahkan tanda ke perintah build, seperti ini:

```
exp build:ios -t simulator
```

Kemudian, jalankan perintah berikut:

```
exp build:status
```

Anda dapat menjalankan perintah itu kapan saja Anda mau, dan sebanyak yang Anda mau, untuk melihat status build yang telah Anda kirimkan. Yang terpenting, dalam kasus ini, adalah, pada akhirnya, output dari perintah ini akan menunjukkan tarball yang tersedia dan tautan untuk mengunduhnya. Lakukan itu, lalu ekstrak file tar.gz dengan menjalankan perintah ini:

```
tar -xvzf your-app.tar.gz
```

Kemudian Anda dapat menjalankannya dengan memulai instance Simulator iOS dan menjalankan perintah ini:

`xcrun simctl install booted` <jalur aplikasi> diikuti oleh `xcrun simctl launch booted` <pengidentifikasi aplikasi>.

Pilihan lainnya, yang perlu Anda pertimbangkan untuk menjalankannya di perangkat sungguhan, adalah TestFlight Apple (https://developer.apple.com/testflight). Secara konseptual, ini mirip dengan klien Expo tetapi sedikit berbeda (Anda tidak perlu meluncurkannya untuk meluncurkan aplikasi yang diinstal dengannya, seperti yang Anda lakukan pada klien Expo) dan sedikit lebih rumit (dan juga mahal, karena Anda memerlukan akun pengembang Apple untuk menggunakannya). Ide dasarnya adalah Anda mengunduh file IPA yang dibuat Expo untuk Anda, mengunggahnya ke TestFlight, menambahkan anggota tim yang dapat mengakses aplikasi, dan kemudian mereka akan dapat melakukannya.

Catatan Sayangnya, meskipun menginstal dan menjalankan aplikasi di Android sangat mudah dan tidak memerlukan akun khusus, iOS adalah cerita yang sama sekali berbeda.

Seperti yang disebutkan, Anda memerlukan akun pengembang Apple (dengan biaya $99/tahun); Anda harus menyiapkan akun tersebut; dan Anda harus menyiapkan TestFlight. Prosedur ini dapat melibatkan banyak hal, jadi tidak dijelaskan secara rinci di sini. Selain tautan TestFlight, Anda perlu mengakses https://developer.apple.com, jika Anda akan menjalankan aplikasi di iOS.

Setelah Anda siap mengirimkan aplikasi ke toko iOS atau Android, Anda akan mengikuti prosedur yang diuraikan Google dan Apple untuk pengiriman aplikasi. Itu menjadi latihan bagi pembaca, karena berada di luar cakupan buku ini. Namun, saya ingin menunjukkan halaman ini dalam dokumentasi Expo: https://docs.expo.io/versions/latest/ distribution/app-stores.html. Di sini, Anda akan menemukan beberapa informasi bermanfaat yang seharusnya menjadi titik awal untuk membuat perjalanan Anda semulus mungkin. Setelah Anda membaca informasinya dan siap, Anda harus mendaftar untuk akun pengembang Apple, seperti yang dijelaskan sebelumnya, dan/atau akun pengembang Google (dengan biaya tetap $25), dan Anda siap memulai!

17.11 KESIMPULAN

Bab ini memberikan langkah awal dalam membangun aplikasi nyata menggunakan React Native, yang meliputi pemahaman tentang navigasi, penggunaan komponen khusus, serta pengenalan pustaka komponen pihak ketiga dan berbagai API. Selanjutnya, di bab berikutnya, kita akan melanjutkan pembangunan aplikasi Pemilih Restoran dengan fokus pada layar Keputusan, dan mendalami konsep seperti flexbox, tata letak, serta pengujian dan debugging.

DAFTAR PUSTAKA

Abdulloh, R. (2020). *Menguasai React JS Untuk Pemula: Panduan belajar JavaScript dari dasar hingga membuat aplikasi web modern* (Vol. 1). Rohi Abdulloh.

Aditya, M. D., & Susanty, M. (2022). Studi Komparasi Maintainability Antara Aplikasi yang Dikembangkan dengan Framework Flutter dan React Native. *Jurnal Informatika*, *9*(2), 159-171.

Aggarwal, S. (2018). Modern Web-Development using ReactJS. *International Journal of Recent Research Aspects*, *5*(1).

Arfani, M. Z., Pinandito, A., & Jonemaro, E. M. A. (2024). Studi Perbandingan Pengembangan Aplikasi Progressive Web App pada Proses Kompresi Gambar Menggunakan React Image File Resizer dan Native. *Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer*, *8*(6).

Arnesia, P. D., Pratama, N. A., & Sjafrina, F. (2022). Aplikasi Artificial Intelligence Untuk Mendeteksi Objek Berbasis Web Menggunakan Library Tensorflow Js, React Js Dan Coco Dataset. *JSiI (Jurnal Sistem Informasi)*, *9*(1), 62-69.

Banks, A., & Porcello, E. (2017). *Learning React: functional web development with React and Redux*. " O'Reilly Media, Inc.".

Bielak, K., Borek, B., & Plechawska-Wójcik, M. (2022). Web application performance analysis using Angular, React and Vue. js frameworks. *Journal of Computer Sciences Institute*, *23*, 77-83.

Bustamin, S. (2021). Aplikasi dekstop multi platform untuk redis client framework Electron JS dan React JS. *Dewantara Journal of Technology*, *2*(1), 21-25.

Chen, S., Thaduri, U. R., & Ballamudi, V. K. R. (2019). Front-end development in react: an overview. *Engineering International*, *7*(2), 117-126

Chen, S., Thaduri, U. R., & Ballamudi, V. K. R. (2019). Front-end development in react: an overview. *Engineering International*, *7*(2), 117-126..

Chinnathambi, K. (2018). *Learning React: a hands-on guide to building web applications using React and Redux*. Addison-Wesley Professional.

Davis, A. (2023). Creating a responsive, API-powered, PaaS-deployed, single-page application in React.

Domes, S. (2017). *Progressive Web Apps with React: Create lightning fast web apps with native power using React and Firebase*. Packt Publishing Ltd.

Fadilah, M. N. (2024). *Pembangunan Aplikasi Pengelolaan Data Pemesanan Dan Stok Barang Produksi Berbasis Web Menggunakan React. Js Di Percetakan Aliza* (Doctoral dissertation, Fakultas Teknik Unpas).

Ferreira, F., & Valente, M. T. (2023). Detecting code smells in React-based Web apps. *Information and Software Technology*, *155*, 107111.

Gackenheimer, C. (2015). *Introduction to React*. Apress.

Hayward, J., Fedosejev, A., Prusty, N., Horton, A., Vice, R., Holmes, E., & Bray, T. (2016). *React: Building Modern Web Applications*. Packt Publishing Ltd.

Hoque, S. (2018). *Full-stack react projects: Modern web development using react 16, node, express, and mongodb*. Packt Publishing Ltd.

Ismail, I., & AlBahri, F. P. (2019). Perancangan E-Kuisioner menggunakan CodeIgniter dan React-Js sebagai Tools Pendukung Penelitian. *J-SAKTI (Jurnal Sains Komputer dan Informatika)*, *3*(2), 337-347.

Iswari, L. (2021). Penerapan React JS Pada Pengembangan FrontEnd Aplikasi Startup Ubaform. *AUTOMATA*, *2*(2).

Jartarghar, H. A., Salanke, G. R., AR, A. K., Sharvani, G. S., & Dalali, S. (2022). React apps with Server-Side rendering: Next. js. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, *14*(4), 25-29.

Jihadi, H., & Syarabil, A. F. (2023). Perbandingan React Js Dan Vue Js Dalam Pengembangan Aplikasi Web Interaktif: Sebuah Studi Komparatif. *Jurnal Sistem Informasi Bisnis (JUNSIBI)*, *4*(2), 70-79.

Juliansyah, F., Utomo, S., Rachmanto, A., & Budiarto, S. (2021). Aplikasi Quiz dengan Konsep Gamification Berbasis Web Menggunakan Ruby On Rails & React. Js. *Jurnal Teknologi Informasi dan Komunikasi*, *11*(2).

Kainu, I. (2022). *Optimization in React. js: Methods, Tools, and Techniques to Improve Performance of Modern Web Applications* (Bachelor's thesis).

Karim, M. A., & Adriansyah, A. R. (2022). Analisis dan Perancangan Aplikasi Mobile untuk Donasi menggunakan Metode Hybrid berbasis React Native. *Jurnal Informatika Terpadu*, *8*(1), 26-34.

Khati Chhetri, A. (2024). Developing a Front-end web app using React.

Kroons, A. A., & Dewi, C. (2023). Pengembangan Dashboard Trivy Berbasis Website Menggunakan React Js dan Golang. *Jurnal Indonesia: Manajemen Informatika dan Komunikasi*, *4*(3), 1037-1049.

Le, T. (2022). A react responsive web application managing offers and configurations of projects.

Luong, Q. (2019). Web application development with Reactjs framework; Case: Web application for an association.

Maolana, Y., & Pambudi, A. (2023). Aplikasi Rekam Medis Imran Medical Center Menggunakan React Js Dengan Metode Prototype. *Infotech journal*, *9*(2), 626-636.

Mardan, A. (2017). *React quickly: painless web apps with React, JSX, Redux, and GraphQL*. Simon and Schuster.

Muhammad, F., Fitri, I., & Nuraini, R. (2022). Implementasi Customer Relationship Management (CRM) pada Sistem Informasi Pemasaran dengan Menggunakan Framework React. JS Berbasis Website. *Jurnal JTIK (Jurnal Teknologi Informasi dan Komunikasi)*, *6*(1), 93-101.

Mulyanto, Y., Haryanti, E., & Lazarus, L. (2024). Implementasi React Server Component Dan Server Action Untuk Meningkatkan Performa Aplikasi Web. *Teknimedia: Teknologi Informasi dan Multimedia*, *5*(1), 17-24.

Noviana, R., & Maulana, M. I. (2024). Shrimp-Detector App Aplikasi Website untuk Mendeteksi Udang Menggunakan Custom Model YOLOv8, Onnxruntime-web API, dan React Js. *Jurnal Ilmiah KOMPUTASI*, *23*(2), 201-2012.

Nursaid, F. F., Brata, A. H., & Kharisma, A. P. (2020). Pengembangan Sistem Informasi Pengelolaan Persediaan Barang Dengan ReactJS Dan React Native Menggunakan Prototype (Studi Kasus: Toko Uda Fajri). *Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer*, *4*(1), 46-55.

Pavić, F., & Brkić, L. (2021, September). Methods of Improving and Optimizing React Web-applications. In *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)* (pp. 1753-1758). IEEE.

Pradana, F., Yunianto, D. R., & Arief, S. N. (2024, June). Optimasi Performa React JS Menggunakan Code Splitting. In *Prosiding Seminar Nasional Teknik Elektro, Sistem Informasi, dan Teknik Informatika (SNESTIK)* (Vol. 1, No. 1, pp. 412-419).

Prasetiyo, S. M., Nugroho, M. I. P., Putri, R. L., & Fauzi, O. (2022). Pembahasan Mengenai Front-End Web Developer dalam Ruang Lingkup Web Development. *BULLET: Jurnal Multidisiplin Ilmu*, *1*(06), 1015-1020.

Purwanto, D. D., Honggara, E. S., Tjandra, S., Ardhi, S., & Tjoa, N. (2023). Pengembangan Aplikasi Human Resource Management pada PT. HJMB Menggunakan JS, React Native, dan GraphQL. *Journal of Information System, Graphics, Hospitality and Technology*, *5*(2), 95-101.

Putra, R. A. (2024). *Perancangan Aplikasi Layanan Pengaduan Customer Berbasis Web Menggunakan React Js: Studi Kasus Pada Pt Inovati F 78* (Doctoral dissertation, Sekolah Tinggi Teknologi Terpadu Nurul Fikri).

Rahmadhani, S., Wildana, D. W., Arumdanie, H. W., & Hakim, L. (2024). Penerapan React JS dan Axios untuk Pengembangan Front-end Aplikasi iCare. *Software Development, Digital Business Intelligence, and Computer Engineering*, *2*(02), 40-46.

Riva, M. (2022). *Real-World Next. js: Build scalable, high-performance, and modern web applications using Next. js, the React framework for production*. Packt Publishing Ltd.

Roldan, C. S. (2018). *React Cookbook: Create dynamic web apps with React using Redux, Webpack, Node. js, and GraphQL*. Packt Publishing Ltd.

Salim, A. (2021). Perancangan Frontend Aplikasi Pemandu Pariwisata Menggunakan Framework React. Js di Provinsi Jawa Barat. *TEMATIK*, *8*(1), 132-145.

Santoso, M. F. (2021). Teknik Single Page Application (Spa) Layout Web Dengan Menggunakan React Js Dan Bootstrap. *Jurnal Khatulistiwa Informatika*, *9*(2).

Sardjono, M. W., & Sanjaya, N. (2023). Aplikasi Konsultasi Dokter-Pasien Secara Online Berbasis Android Menggunakan React Native. *Jurnal Sistem Informasi dan Informatika (Simika)*, *6*(2), 153-164.

Sari, A. S., & Hidayat, R. (2022). Designing website vaccine booking system using golang programming language and framework react JS. *JISICOM (Journal of Information System, Informatics and Computing)*, *6*(1), 22-39.

Satyal, A. (2020). Designing and Developing a Website with ReactJS: Progressive Web Application.

Subramanian, V. (2019). *Pro Mern Stack: Full Stack Web App Development with Mongo, Express, React and Node*. Apress.

Tanna, M., & Singh, H. (2018). *Serverless Web Applications with React and Firebase: Develop real-time applications for web and mobile platforms*. Packt Publishing Ltd.

Tanudjaja, D., & Tanone, R. (2021). Analisis Penerapan Code Splitting Library React pada Aplikasi Penjualan Mebel Berbasis Website. *Jurnal Teknik Informatika dan Sistem Informasi*, *7*(2), 344-356.

Timalsina, S. S. (2019). Progressive Web Application with Reactjs.

Vipul, A. M., & Sonpatki, P. (2016). *ReactJS by Example-Building Modern Web Applications with React*. Packt Publishing Ltd.

Vo, L. T. T. (2020). Web application development with react and google firebase: data visualization.

Wiguna, P. D. A., Swastika, I. P. A., & Satwika, I. P. (2018). Rancang bangun aplikasi point of sales distro management system dengan menggunakan framework react native. *Jurnal Nasional Teknologi Dan Sistem Informasi*, *4*(3), 149-159.

Wijonarko, D., & Aji, R. F. (2018). Perbandingan Phonegap dan react Native sebagai framework pengembangan aplikasi mobile. *Jurnal Manajemen Informatika dan Sistem Informasi*, *1*(2), 1-7.

Zakaria, H. (2023). Implementasi Single Page Aplikasi (SPA) Pada Aplikasi Pengajuan Cuti Karyawan Berbasis Web Menggunakan React Js: Studi Kasus: PT Mitra Bisnis Sarana. *LOGIC: Jurnal Ilmu Komputer dan Pendidikan*, *1*(6), 1653-1661.

Zlatinov, Z., & Angelova, N. (2023). Web application of movie catalogue with React. In *SHS Web of Conferences* (Vol. 176, p. 02012). EDP Sciences.

Dr. Joseph Teguh Santoso, S.Kom M.Kom.

Aplikasi Web dengan REACT

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