CHAPTER 18

Creating Complete Applications

React provides an excellent set of features for presenting HTML content to the user and relies on third-party packages to provide the supporting functionality required to develop complete web applications. There are countless packages available for use with React, and in this part of the book, I introduce those that are most widely used and most likely to be needed by readers of this book. These packages are all open-source and freely available, and there are paid-for support options in some cases.

In this chapter, I build an example application using only the features described in Part 2 of this book. In the chapters that follow, I introduce the third-party packages and demonstrate the features they provide and explain the problems they solve. Table 18-1 provides a brief overview of the packages that are covered in this part of the book.

Table 18-1. *The Packages Described in This Part of the Book*

Name	Description
Redux	Redux provides a data store that manages data outside of an application's components. I use this package in Chapters 19 and 20.
React Redux	React Redux connects React components through its props to a Redux data store, allowing direct access to data without relying on prop threading. I use this package in Chapters 19 and 20.
React Router	React Router provides URL routing for React applications, allowing the components displayed to the user to be selected based on the browser's URL. I use this package in Chapters 21 and 22.
Axios	Axios provides a consistent API for making asynchronous HTTP requests. I use this package in Chapter 23 to consume a RESTful web service and in Chapter 25 to consume a GraphQL service.
Apollo Boost	Apollo is a client for consuming GraphQL services, which are more flexible than traditional RESTful web services. I use the Boost edition of this package, which provides sensible defaults for React applications, in Chapter 25 to consume a GraphQL service.
React Apollo	React Apollo connects React components to GraphQL queries and mutations, allowing a GraphQL service to be consumed through props.

A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_18

There are credible alternatives for each of the packages I have selected, and I make suggestions in each chapter in case you can't get along with the packages that are covered. Please e-mail me at `adam@adam-freeman.com` if there is a package that interests you that I have not covered in this part of the book. Although I make no promises, I will try to include commonly requested packages in the next edition of this book or, if there is sufficient demand, in updates posted to this book's GitHub repository.

Creating the Project

Open a new command prompt, navigate to a convenient location, and run the command shown in Listing 18-1.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from `https://github.com/Apress/pro-react-16`.

Listing 18-1. Creating the Example Project

```
npx create-react-app productapp
```

Run the commands shown in Listing 18-2 to navigate to the `productapp` folder to add the Bootstrap package.

Listing 18-2. Adding the Bootstrap CSS Framework

```
cd productapp
npm install bootstrap@4.1.2
```

To include the Bootstrap CSS stylesheet in the application, add the statement shown in Listing 18-3 to the `index.js` file, which can be found in the `productapp/src` folder.

Listing 18-3. Including Bootstrap in the index.js File in the src Folder

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';
import 'bootstrap/dist/css/bootstrap.css';

ReactDOM.render(<App />, document.getElementById('root'));

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: http://bit.ly/CRA-PWA
serviceWorker.unregister();
```

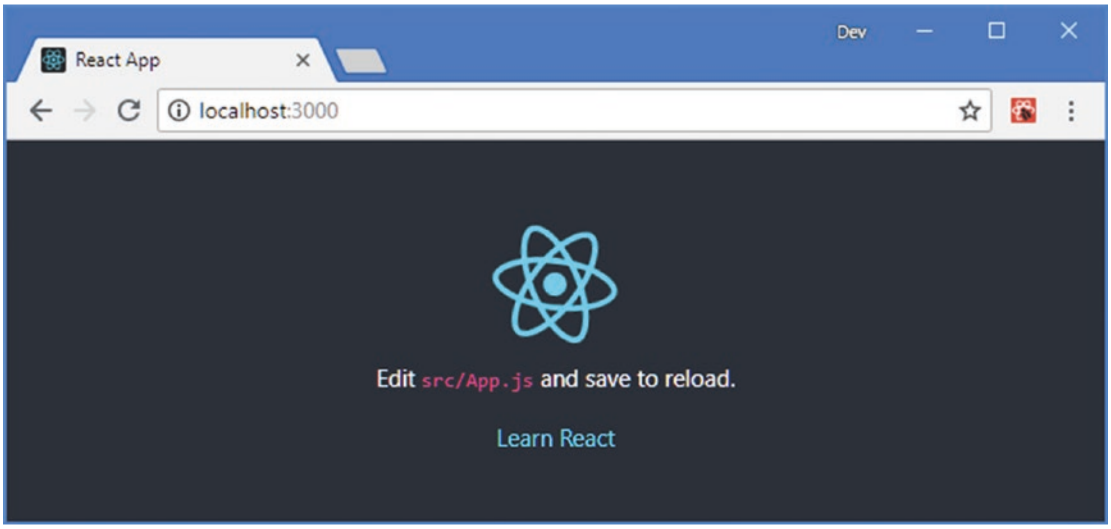
Starting the Development Tools

Using the command prompt, run the command shown in Listing 18-4 in the `productapp` folder to start the development tools.

Listing 18-4. Starting the Development Tools

```
npm start
```

Once the initial preparation for the project is complete, a new browser window will open and display the URL `http://localhost:3000`, which shows the placeholder content in Figure 18-1.

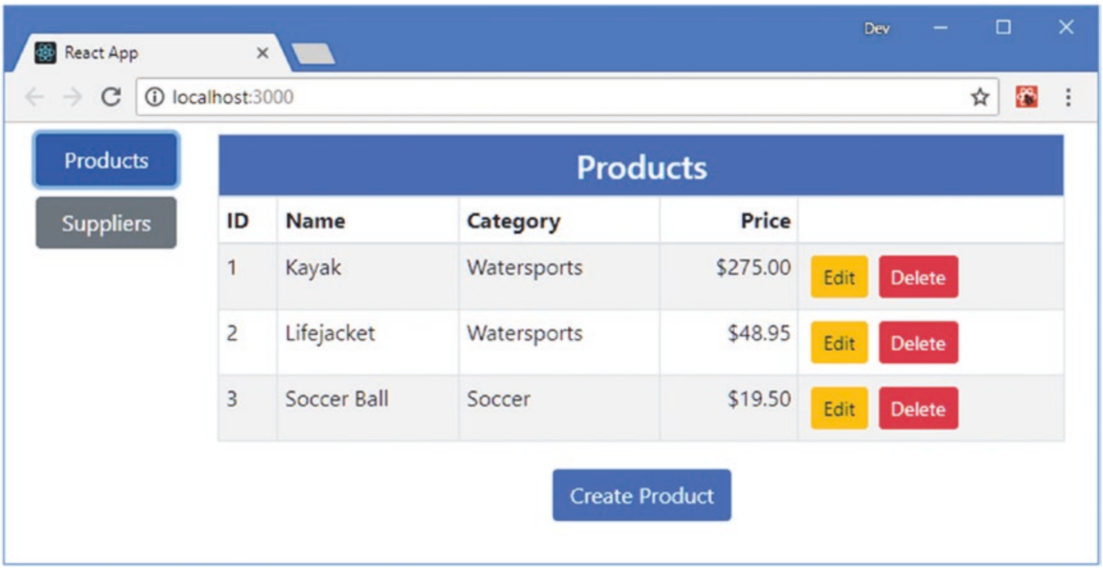


Edit `src/App.js` and save to reload.

Figure 18-1. *Running the example application*

Creating the Example Application

The application in this chapter is simple but representative of a typical project built using only the features provided by React. The application presents the user with create, read, update, and delete (CRUD) features for two types of data, products and suppliers, and the user can toggle between the data that is being managed. Figure 18-2 shows how the application will appear once the components defined in the following sections have been created.



Products

ID	Name	Category	Price	
1	Kayak	Watersports	$275.00	Edit Delete
2	Lifejacket	Watersports	$48.95	Edit Delete
3	Soccer Ball	Soccer	$19.50	Edit Delete

Figure 18-2. *The example application*

Example applications are contrived, of course, and my goal, in this case, is to show that the core React features are powerful but are not sufficient on their own to create complex web applications. Once the application has been defined, I highlight the problems that it contains, each of which I address using the tools and packages described in the following chapters.

Creating the Product Features

To get started with the application functionality, I added a file called `ProductTableRow.js` to the `src` folder and used it to define the component shown in Listing 18-5.

Listing 18-5. The Contents of the ProductTableRow.js File in the src Folder

```
import React, { Component } from "react";

export class ProductTableRow extends Component {

    render() {
        let p = this.props.product;
        return <tr>
            <td>{ p.id }</td>
            <td>{ p.name }</td>
            <td>{ p.category}</td>
            <td className="text-right">${ Number(p.price).toFixed(2) }</td>
            <td>
                <button className="btn btn-sm btn-warning m-1"
                    onClick={ () => this.props.editCallback(p) }>
                        Edit
                </button>
                <button className="btn btn-sm btn-danger m-1"
                    onClick={ () => this.props.deleteCallback(p) }>
                        Delete
                    </button>
            </td>
        </tr>
    }
}
```

This component renders a single row in a table, with columns for `id`, `name`, `category`, and `price` properties, which are obtained from a prop object called `product`. There is a further column that displays Edit and Delete buttons that invoke function props named `editCallback` and `deleteCallback`, passing the `product` prop as an argument.

Creating the Product Table

I added a file called `ProductTable.js` to the `src` folder and used it to define the component shown in Listing 18-6.

Listing 18-6. The Contents of the ProductTable.js File in the src Folder

```
import React, { Component } from "react";
import { ProductTableRow } from "./ProductTableRow";

export class ProductTable extends Component {

    render() {
        return <table className="table table-sm table-striped table-bordered">
                <thead>
                    <tr>
                        <th colSpan="5"
                                className="bg-primary text-white text-center h4 p-2">
                            Products
                        </th>
                    </tr>
                    <tr>
                        <th>ID</th><th>Name</th><th>Category</th>
                        <th className="text-right">Price</th>
                        <th></th>
                    </tr>
                </thead>
                <tbody>
                    {
                        this.props.products.map(p =>
                            <ProductTableRow product={ p }
                                key={ p.id }
                                editCallback={ this.props.editCallback }
                                deleteCallback={ this.props.deleteCallback } />)
                    }
                </tbody>
            </table>
    }
}
```

This component renders a table, whose body is populated with ProductTableRow components for each object in an array prop named products. This component passes on the deleteCallback and editCallback function props to the ProductTableRow instances.

Creating the Product Editor

To allow the user to edit a product or provide values for a new product, I added a file called ProductEditor.js in the src folder and added the code shown in Listing 18-7.

Listing 18-7. The Contents of the ProductEditor.js File in the src Folder

```
import React, { Component } from "react";

export class ProductEditor extends Component {

    constructor(props) {
        super(props);
        this.state = {
            formData: {
                id: props.product.id || "",
                name: props.product.name || "",
                category: props.product.category || "",
                price: props.product.price || ""
            }
        }
    }

    handleChange = (ev) => {
        ev.persist();
        this.setState(state => state.formData[ev.target.name] =  ev.target.value);
    }

    handleClick = () => {
        this.props.saveCallback(this.state.formData);
    }

    render() {
        return <div className="m-2">
            <div className="form-group">
                <label>ID</label>
                <input className="form-control" name="id"
                    disabled
                    value={ this.state.formData.id }
                    onChange={ this.handleChange } />
            </div>
            <div className="form-group">
                <label>Name</label>
                <input className="form-control" name="name"
                    value={ this.state.formData.name }
                    onChange={ this.handleChange } />
            </div>
```

```
                <div className="form-group">
                    <label>Category</label>
                    <input className="form-control" name="category"
                        value={ this.state.formData.category }
                        onChange={ this.handleChange } />
                </div>
                <div className="form-group">
                    <label>Price</label>
                    <input className="form-control" name="price"
                        value={ this.state.formData.price }
                        onChange={ this.handleChange } />
                </div>
                <div className="text-center">
                    <button className="btn btn-primary m-1" onClick={ this.handleClick }>
                        Save
                    </button>
                    <button className="btn btn-secondary"
                            onClick={ this.props.cancelCallback }>
                        Cancel
                    </button>
                </div>
            </div>
    }
}
```

The ProductEditor component presents the user with fields for editing the properties of an object. The initial values for the fields are received from a prop named product and used to populate state data. There is a Save button that invokes a function prop named saveCallback when it is clicked, passing the state data values so that can be saved. There is also a Cancel button that invokes a function callback named cancelCallback when it is clicked.

Creating the Product Display Component

Next, I need a component that will switch between the table of products and the product editor. I added a file called ProductDisplay.js to the src folder and used it to define the component shown in Listing 18-8.

Listing 18-8. The Contents of the ProductDisplay.js File in the src Folder

```
import React, { Component } from "react";
import { ProductTable } from "./ProductTable"
import { ProductEditor } from "./ProductEditor";

export class ProductDisplay extends Component {

    constructor(props) {
        super(props);
        this.state = {
            showEditor: false,
            selectedProduct: null
        }
    }
    startEditing = (product) => {
        this.setState({ showEditor: true, selectedProduct: product })
    }

    createProduct = () => {
        this.setState({ showEditor: true, selectedProduct: {} })
    }

    cancelEditing = () => {
        this.setState({ showEditor: false, selectedProduct: null })
    }

    saveProduct = (product) => {
        this.props.saveCallback(product);
        this.setState({ showEditor: false, selectedProduct: null })
    }

    render() {
        if (this.state.showEditor) {
            return <ProductEditor
                key={ this.state.selectedProduct.id || -1 }
                product={ this.state.selectedProduct }
                saveCallback={ this.saveProduct }
                cancelCallback={ this.cancelEditing } />
        } else {
            return <div className="m-2">
                <ProductTable products={ this.props.products }
                    editCallback={ this.startEditing }
                    deleteCallback={ this.props.deleteCallback } />
                <div className="text-center">
                    <button className="btn btn-primary m-1"
                        onClick={ this.createProduct }>
                        Create Product
                    </button>
                </div>
            </div>
        }
    }
}
```

This component defines state data to determine whether the data table or the editor should be shown and, if it is the editor, which product the user wants to modify. This component passes on function props to both the ProductEditor and ProductTable components, as well as introducing its own functionality.

Creating the Supplier Functionality

The part of the application that deals with supplier data follows a similar pattern to the components created in earlier sections. I added a file called SupplierTableRow.js to the src folder and used it to define the component shown in Listing 18-9.

Listing 18-9. The Contents of the SupplierTableRow.js File in the src Folder

```
import React, { Component } from "react";

export class SupplierTableRow extends Component {

    render() {
        let s = this.props.supplier;
        return <tr>
            <td>{ s.id }</td>
            <td>{ s.name }</td>
            <td>{ s.city}</td>
            <td>{ s.products.join(", ") }</td>
            <td>
                <button className="btn btn-sm btn-warning m-1"
                    onClick={ () => this.props.editCallback(s) }>
                        Edit
                </button>
                <button className="btn btn-sm btn-danger m-1"
                    onClick={ () => this.props.deleteCallback(s) }>
                        Delete
                    </button>
            </td>
        </tr>
    }
}
```

This component renders a table row with the `id`, `name`, `city`, and `products` properties of a prop object named `supplier`. There are also Edit and Delete buttons that invoke function props.

Creating the Supplier Table

To present a table of suppliers to the user, I added a file called `SupplierTable.js` to the `src` folder and added the code shown in Listing 18-10.

Listing 18-10. The Contents of the SupplierTable.js File in the src Folder

```
import React, { Component } from "react";
import { SupplierTableRow } from "./SupplierTableRow";

export class SupplierTable extends Component {

    render() {
        return <table className="table table-sm table-striped table-bordered">
                <thead>
                    <tr>
                        <th>ID</th><th>Name</th><th>City</th>
                        <th>Products</th><th></th>
                    </tr>
                </thead>
                <tbody>
                    {
                        this.props.suppliers.map(s =>
                            <SupplierTableRow supplier={ s }
                                key={ s.id }
                                editCallback={ this.props.editCallback }
                                deleteCallback={ this.props.deleteCallback } />)
                    }
                </tbody>
            </table>
    }
}
```

This component renders a table, mapping each object in the `suppliers` prop array into a `SupplierTableRow`. The props for the callbacks are received from the parent component and passed on.

Creating the Supplier Editor

To create the editor for suppliers, I added a file called `SupplierEditor.js` to the `src` folder and used it to define the component shown in Listing 18-11.

Listing 18-11. The Contents of the SupplierEditor.js File in the src Folder

```
import React, { Component } from "react";

export class SupplierEditor extends Component {

    constructor(props) {
        super(props);
        this.state = {
            formData: {
                id: props.supplier.id || "",
                name: props.supplier.name || "",
                city: props.supplier.city || "",
                products: props.supplier.products || [],
            }
        }
    }

    handleChange = (ev) => {
        ev.persist();
        this.setState(state =>
            state.formData[ev.target.name] =
                ev.target.name === "products"
                    ? ev.target.value.split(",") : ev.target.value);
    }

    handleClick = () => {
        this.props.saveCallback(
            {
                ...this.state.formData,
                products: this.state.formData.products.map(val => Number(val))
            });
    }

    render() {
        return <div className="m-2">
            <div className="form-group">
                <label>ID</label>
                <input className="form-control" name="id"
                    disabled
                    value={ this.state.formData.id }
                    onChange={ this.handleChange } />
            </div>
            <div className="form-group">
                <label>Name</label>
                <input className="form-control" name="name"
                    value={ this.state.formData.name }
                    onChange={ this.handleChange } />
            </div>
            <div className="form-group">
                <label>City</label>
                <input className="form-control" name="city"
                    value={ this.state.formData.city }
                    onChange={ this.handleChange } />
            </div>

            <div className="form-group">
                <label>Products</label>
                <input className="form-control" name="products"
                    value={ this.state.formData.products }
                    onChange={ this.handleChange } />
            </div>

            <div className="text-center">
                <button className="btn btn-primary m-1" onClick={ this.handleClick }>
                    Save
                </button>
                <button className="btn btn-secondary"
                        onClick={ this.props.cancelCallback }>
                    Cancel
                </button>
            </div>
        </div>
    }
}
```

Creating the Supplier Display Component

To manage the side of the application that deals with supplier data so that only the table or the editor is shown, I added a file called `SupplierDisplay.js` to the `src` folder and used it to define the component shown in Listing 18-12.

Listing 18-12. The Contents of the SupplierDisplay.js File in the src Folder

```
import React, { Component } from "react";
import { SupplierEditor } from "./SupplierEditor";
import { SupplierTable } from "./SupplierTable";

export class SupplierDisplay extends Component {

    constructor(props) {
        super(props);
        this.state = {
            showEditor: false,
            selected: null
        }
    }

    startEditing = (supplier) => {
        this.setState({ showEditor: true, selected: supplier })
    }

    createSupplier = () => {
        this.setState({ showEditor: true, selected: {} })
    }

    cancelEditing = () => {
        this.setState({ showEditor: false, selected: null })
    }

    saveSupplier= (supplier) => {
        this.props.saveCallback(supplier);
        this.setState({ showEditor: false, selected: null })
    }

    render() {
        if (this.state.showEditor) {
            return <SupplierEditor
                key={ this.state.selected.id || -1 }
                supplier={ this.state.selected }
                saveCallback={ this.saveSupplier }
                cancelCallback={ this.cancelEditing } />
        } else {
            return <div className="m-2">
                    <SupplierTable suppliers={ this.props.suppliers }
                        editCallback={ this.startEditing }
                        deleteCallback={ this.props.deleteCallback }
                    />
                    <div className="text-center">
                        <button className="btn btn-primary m-1"
                            onClick={ this.createSupplier }>
                                Create Supplier
                        </button>
                    </div>
            </div>
        }
    }
}
```

The SupplierDisplay component has its own state data for determining whether the editor or table component should be displayed.

Completing the Application

To allow the user to choose between the product or supplier features, I added a file called Selector.js to the src folder and added the code shown in Listing 18-13.

Listing 18-13. The Contents of the Selector.js File in the src Folder

```
import React, { Component } from "react";

export class Selector extends Component {

    constructor(props) {
        super(props);
        this.state = {
            selection: React.Children.toArray(props.children)[0].props.name
        }
    }

    setSelection = (ev) => {
        ev.persist();
        this.setState({ selection: ev.target.name});
    }

    render() {
        return <div className="container-fluid">
            <div className="row">
                <div className="col-2">
                    { React.Children.map(this.props.children, c =>
                        <button
                            name={ c.props.name }
                            onClick={ this.setSelection }
                            className={`btn btn-block m-2
                            ${this.state.selection === c.props.name
                                ? "btn-primary active": "btn-secondary"}`}>
                                    { c.props.name }
                        </button>
                    )}
                </div>
                <div className="col">
                    {
                        React.Children.toArray(this.props.children)
                            .filter(c => c.props.name === this.state.selection)
                    }
                </div>
            </div>
        </div>
    }
}
```

The Selector component is a container that renders a button for each of its children and displays only the one selected by the user. To provide the data that will be displayed by the application and implementation for the callback functions that operate on it, I added a file called `ProductsAndSuppliers.js` to the `src` folder and used it to define the component shown in Listing 18-14.

Listing 18-14. The Contents of the ProductsAndSuppliers.js File in the src Folder

```
import React, { Component } from 'react';
import { Selector } from './Selector';
import { ProductDisplay } from './ProductDisplay';
import { SupplierDisplay } from './SupplierDisplay';

export default class ProductsAndSuppliers extends Component {

    constructor(props) {
        super(props);
        this.state = {
            products: [
                { id: 1, name: "Kayak",
                category: "Watersports", price: 275 },
                { id: 2, name: "Lifejacket",
                    category: "Watersports", price: 48.95 },
                { id: 3, name: "Soccer Ball", category: "Soccer", price: 19.50 }
            ],
            suppliers: [
                { id: 1, name: "Surf Dudes", city: "San Jose", products: [1, 2] },
                { id: 2, name: "Field Supplies", city: "New York", products: [3] },
            ]
        }
        this.idCounter = 100;
    }

    saveData = (collection, item) => {
        if (item.id === "") {
            item.id = this.idCounter++;
            this.setState(state => state[collection]
                = state[collection].concat(item));
        } else {
            this.setState(state => state[collection]
                = state[collection].map(stored =>
                      stored.id === item.id ? item: stored))
        }
    }

    deleteData = (collection, item) => {
        this.setState(state => state[collection]
            = state[collection].filter(stored => stored.id !== item.id));
    }

    render() {
        return <div>
            <Selector>
                <ProductDisplay
                    name="Products"
                    products={ this.state.products }
                    saveCallback={ p => this.saveData("products", p) }
                    deleteCallback={ p => this.deleteData("products", p) } />
                <SupplierDisplay
                    name="Suppliers"
                    suppliers={ this.state.suppliers }
                    saveCallback={ s => this.saveData("suppliers", s) }
                    deleteCallback={ s => this.deleteData("suppliers", s) } />
            </Selector>
        </div>
    }
}
```

The component defines product and suppliers state data properties and defines methods that allow objects to be deleted or saved for each data category. The component renders a Selector and provides the category display components as its children.

The final step is to replace the contents of the App component so that the custom components defined in the previous sections are displayed to the user, as shown in Listing 18-15.

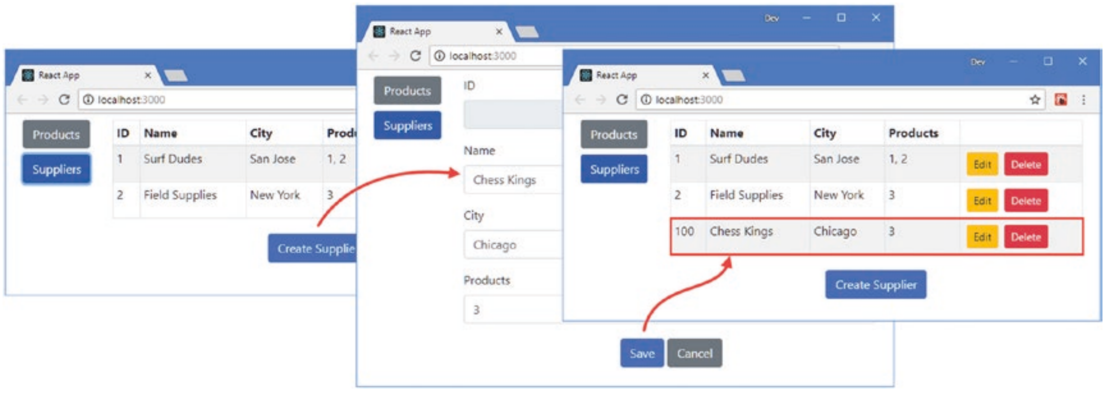
Listing 18-15. Adding Data and Methods to the App.js File in the src Folder

```
import React, { Component } from "react";
import ProductsAndSuppliers from "./ProductsAndSuppliers";

export default class App extends Component {

    render() {
        return <ProductsAndSuppliers/>
    }
}
```

Once you save the changes to the App component, the browser will display the completed example application. To make sure that everything works as it should, click the Suppliers button, click the Create Supplier button, and fill out the form. Click the Save button, and you should see a new entry in the table with the detail you entered, as shown in Figure 18-3.



ID	Name	City	Prod
1	Surf Dudes	San Jose	1, 2
2	Field Supplies	New York	3

ID	
Name	Chess Kings
City	Chicago
Products	3

ID	Name	City	Products	
1	Surf Dudes	San Jose	1, 2	Edit Delete
2	Field Supplies	New York	3	Edit Delete
100	Chess Kings	Chicago	3	Edit Delete

Figure 18-3. *Testing the example application*

Understanding the Limitations of the Example Application

The example application shows how React components can be combined to create applications—but it also shows the limitations of the features that React provides.

The biggest limitation of the example application is that it uses statically defined data that is hard-coded into the `App` component. The same data is displayed each time the application is started, and changes are lost when the browser is reloaded or closed.

The most common way of persisting data outside of a web application is to use a web service, although modern browsers provide support for storing limited amounts of data locally. React doesn't include integrated support for working with web services, but there are some good choices, both for simple web services, which I describe in Chapter 23, and those that present more complex data, which I describe in Chapters 24 and 25.

The next limitation is that the state data has been lifted all the way to the top of the application. As I explained in Part 2, state data can be used to coordinate between components, and that state data can be lifted up to the common ancestor of components that need to access the same data.

The example application shows the downside of this approach, such that the important data—the `products` and `suppliers` arrays, in this case—end up being pushed to the top level of the application. React destroys components when they are unmounted and their state data is lost, which means that any component that is below the `Selector` in the example application is unsuitable for storing the application's data. As a result, all of the application's data has been defined in the `App` component, along with the methods that operate on that data. I exacerbated this problem with the structure I chose for the application, but the underlying issue is that a component's state is perfect for keeping track of the data required to manage the content presented to the user—such as whether a data table or an editor should be displayed—but isn't well-suited for managing the data that relates to the purpose of the application, often known as the *domain data* or *model data*.

The best way to prevent the model data from being pushed up to the top-level component is to put it in a separate data store, which leaves the React components to deal with the presentation of the data without having to manage it. I explain the use of a data store and show you how to create one in Chapters 19 and 20.

The application is also limited in the way that it requires the user to work through a specific sequence of tasks to get to specific features. In many applications, especially those designed to support a specific corporate function, users have to perform a small set of tasks and want to be able to start them as easily as possible. The example application only presents its features in response to clicking particular elements. In Chapters 21 and 22, I add support for URL routing, which makes it possible for users to navigate to specific features directly.

Summary

In this chapter, I created the example application that I will enhance throughout this part of the book. In the next chapter, I start that process by introducing a data store, which will allow the model data to be removed from the `App` component and distributed directly to the parts of the application that need it.