

Fundamentals of Web Development

Third Edition by Randy Connolly and Ricardo Hoar



Chapter 11

React

In this chapter you will learn . . .

- What are frameworks and some of the most popular JavaScript frameworks
- What is React and how to create and use functional and class components
- How to use props, state, and behaviors in React
- Making use of a build tool chain in React
- How to extend React via its lifecycle methods and via component libraries

JavaScript Front-End Frameworks

A **software framework** is a reusable library of code that you can utilize to simplify, improve, and facilitate the process of developing an application.

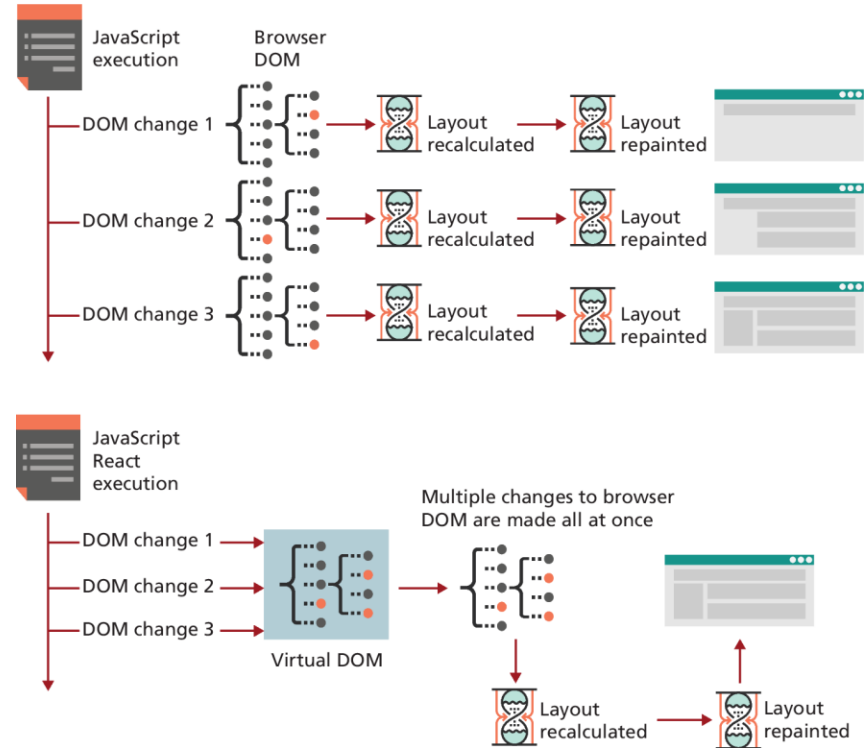
The ability to use a front-end framework has become an essential expected skill for most web developers. As such, this book could no longer simply provide just an overview of these frameworks and a simple Hello World style example. Instead, we have decided to provide an entire chapter on using a single framework;

Why Do We Need Frameworks?

Ideally a framework, simplifies the process of creating user interfaces.

React has a virtual DOM that your code manipulates, improving the execution speed of DOM manipulation and traversal.

They allow the developer to construct the user interface as a series of components.



React, Angular, and Vue

Angular is an “opinionated” framework in that it forces developers to adopt a known and well-regarded approach to structuring and implementing a web application.

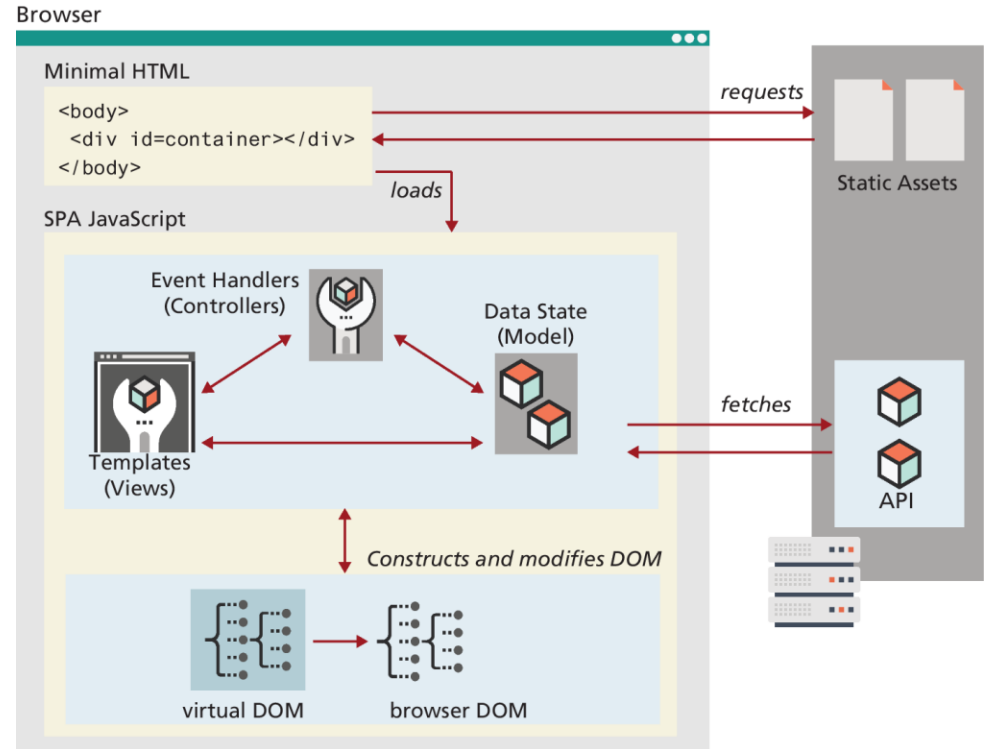
The **React** framework from Facebook is now the most popular JavaScript Framework today for constructing complex front-ends in JavaScript

Vue.js is similar to React in that it focuses on the view. While React uses its own JSX syntax that allows the developer to “inject” HTML into the JavaScript, Vue.js uses HTML templates with data and behavior “injected” via custom attributes and directives. Vue.js is fully open-source

All three frameworks are especially well suited to constructing a **Single-Page Application (SPA)**.

Single Page Application

An SPA constructed with a framework typically contains minimal HTML. Instead, JavaScript is used to populate the DOM with HTML elements using logic contained within the framework and data pulled from some API



Introducing React

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Chapter 11</title>
<script src="https://unpkg.com/react@16/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js">
</script>
```

The React JS libraries.

Use the Babel run-time library to convert JSX in the browser at run-time.

```
<script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
```

Note: language for this block is Babel.

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

```
  const link = React.createElement("a",
    {href: "http://www.reactjs.org"}, "Visit React");
```

```
  ReactDOM.render(link, document.querySelector("#react-container"));
</script>
```

```
</head>
```

```
<body>
```

```
<div id="react-container"></div>
```

```
</body>
```

```
</html>
```

The three parameters are:
element name, attribute
collection, and element's
content.

Inserts the React
content into an
existing element.

This React example is functionally equivalent to
the following regular DOM code:

```
const link = document.createElement("a");
link.setAttribute("href", "http://www.reactjs.org");
link.textContent = "Visit React";
document.querySelector("#react-container").appendChild(link);
```

JSX Syntax

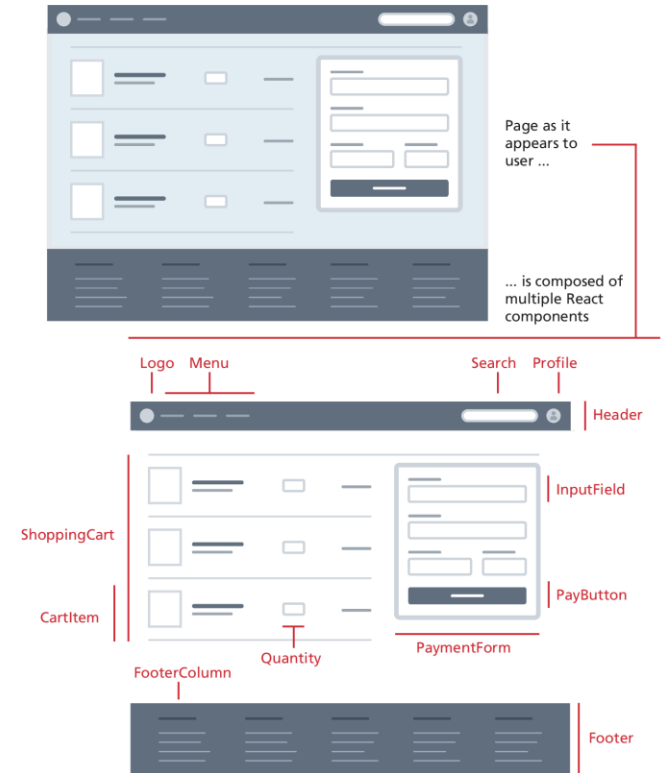
JSX follows the same syntactical rules as XML. These rules are quite straightforward:

- Element names are composed of any of the valid characters (most punctuation symbols and spaces are not allowed) in XML.
- Element names can't start with a number.
- There must be a single root element. A root element is one that contains all the other elements.
- All elements must have a closing element (or be self-closing).
- Elements must be properly nested.
- Elements can contain attributes.
- **Attribute values must always be within quotes.**
- Element and attribute names are case sensitive.
- since the keyword `class` has a different meaning in JavaScript, you cannot use it in JSX (remember that JSX eventually gets converted to JavaScript). Instead, you have to use `className`

React Components

A **component** in React is a block of user interface functionality. They allow the developer to break down the user interface into smaller pieces that can be reused, combined, and nested together.

As you work with React, you will compose your application by nesting multiple components, as shown in Figure 11.5.



React Components (ii)

The **Header** component might be added using the following JSX:

```
<Header>
  <Logo />
  <Menu />
  <Search initial="Find product"/>
  <Profile userId="34"/>
</Header>
```



If React components are *used* via markup, how are they *created*?

Essentially, a React component is simply function that returns a single React element. There are two types of components in React:

- **functional components** and
- **class components.**

Functional Components

The simplest way to create a component in React is to use functions. `const`

```
Logo = function(props) {  
  return ;  
};
```

Notice that the **Logo** function contains a single parameter named **props**. This parameter provides a mechanism for passing information into a component.

Once you have created a functional component, you can reference it via markup. For instance, to use it with `ReactDOM.render`

```
ReactDOM.render(<Logo />, document.querySelector('#react-container'));
```

Nested functional components

```
const Logo = function(props) {  
  return ;  
};  
const Title = function(props) {  
  return <h2>Site Title</h2>;  
};  
const Header = function(props) {  
  return (  
    <header>  
      <Logo />  
      <Title />  
    </header>  
  );  
}  
ReactDOM.render(<Header />,  
  document.querySelector('#react-container'));
```

This ability to reference components via markup allows you to compose your pages as a series of nested components, as shown in Listing 11.1.

LISTING 11.1 Nested functional components

Class Components

The (slightly) more complicated way to create a React component is to use the JavaScript class keyword.

You could rewrite the Header component from Listing 11.1 as →

The render function is required in all class components.

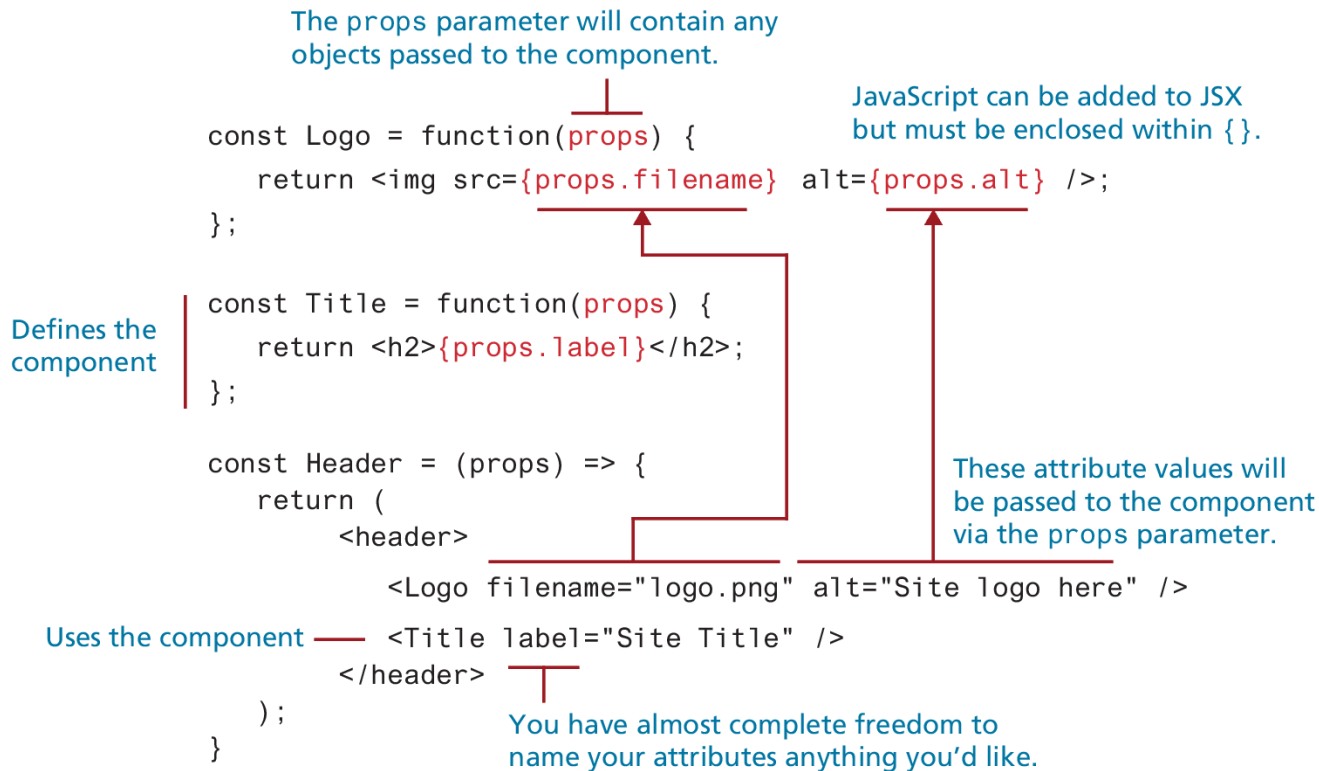
```
class Header extends React.Component {  
  render() {  
    return (  
      <header>  
        <Logo />  
        <Title />  
      </header>  
    );  
  }  
}
```

Props, State, Behavior, and Forms

In this section, you will learn how to make components do more via props, state, and behavior.

- **Props** provide a way to pass data into a component.
- **State** provides a way for a component to have its own internal data, and
- **Behavior** provides a way for a component to respond to events.

Props



Passing Complex Objects via Props

Imagine that you have the following array of movie objects:

```
const movieData = [  
  { id:17, title:"American Beauty", year:1999 },  
  { id:651, title:"Sense and Sensibility", year:1995 },  
  { id:1144, title:"Casablanca", year:1942 }  
];
```

You could pass this array into a component via props:

```
<MovieList movies={movieData} />
```


Passing Complex Objects via Props (ii)

How would MovieList then display this data? One way is to iterate through the array and create an array of `` items:

```
const MovieList = (props) => {  
  const items = [];  
  for (let m of props.movies) {  
    items.push( <li>{m.title}</li> );  
  }  
  return <ul>{ items }</ul>;  
}
```

A more concise approach is to use the **map()**

```
const MovieList = (props) => {  
  const items = props.movies.map( m => <li>{m.title}</li> );  
  return <ul>{ items }</ul>;  
}
```

Passing Complex Objects via Props (iii)

If you examined the result in the browser, you would see this React warning in the console:

Each child in a list should have a unique "key" prop.

This is easily fixed using an index provided by map()

```
props.movies.map( (m,indx) => <li key={indx}>{m.title}</li> );
```

We might decide to separate out the elements as their own component, as shown in Listing 11.2.

State

You need to use **state** instead of props whenever a component's data is going to be altered.

React *reacts* to changes in a component's data.

state can only be changed via the `setState()` function. The `setState()` function thus not only changes the data, but also communicates to React that data has been changed

State is available within class components.

```
class Box extends React.Component {  
  State is usually initialized in the class constructor.  
  constructor(props) {  
    super(props);  
    State data can be any number of property:value pairs.  
    this.state = { currentSeconds : Number(props.start) };  
    State belongs to the component it is defined within.  
  
    setInterval( () => {  
      let newSecs = this.state.currentSeconds + 1;  
      this.setState( { currentSeconds: newSecs } );  
    }, 1000);  
    State is modified by setState() function.  
  }  
  
  render() {  
    return (  
      <div className="box">  
        { this.state.currentSeconds } secs  
      </div>;  
    );  
    Individual state data properties can be referenced using this.state.  
  }  
}
```

Behaviors

Events are important in React components, but the technique for handling them is superficially similar to the old-fashioned inline handlers

To a button allowing the user to find out more about that particular movie from last example, you can define a handler method within *MovieListItem* and then reference it via an **onClick** attribute

```
const MovieListItem = (props) => {  
    const handleClick = () => {  
        alert('handling the click id=' + props.movie.id);  
    }  
    return (  
        <li>  
            {props.movie.title}  
            <button onClick={handleClick}>View</button>  
        </li>);  
    }  
}
```

Passing Data to Event Handlers

```
const App = (props) => {
  const movieData = [{ id: 17, title: "American Beauty", year: 1999 }, ... ];
  const handleClick = (movie) => {
    alert('handling the click for ' + movie.title);
  }
  return (
    <ul>
      { movieData.map( m =>
        <li><button onClick={ () => handleClick(m) }>
          View {m.title} </button></li>
        )}
    </ul> );
}
```

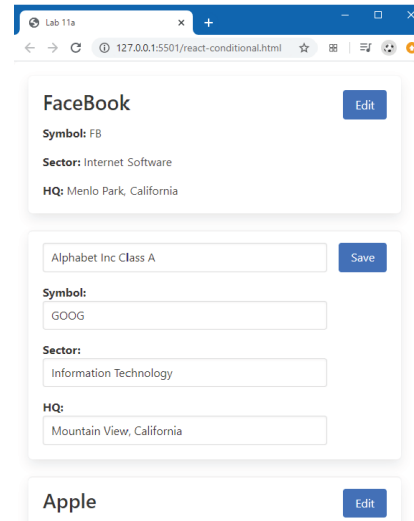
LISTING 11.3 Passing other data to event handler

Event-Driven Conditional Rendering

Rendering different content based on state data is known as **conditional rendering**.

```
...  
render() {  
  if (this.state.editing) {  
    return this.renderEdit();  
  }  
  else {  
    return this.renderNormal();  
  }  
}
```

LISTING 11.5 Conditional rendering



The screenshot shows a web browser window with the address bar displaying "127.0.0.1:5501/react-conditional.html". The page content includes three company entries, each with a form and an "Edit" button. The first entry is for "FaceBook" with fields for "Symbol" (FB), "Sector" (Internet Software), and "HQ" (Menlo Park, California). The second entry is for "Alphabet Inc Class A" with fields for "Symbol" (GOOG), "Sector" (Information Technology), and "HQ" (Mountain View, California). The third entry is for "Apple" with an "Edit" button. Red vertical lines on the right side of the browser window indicate that the top two entries are rendered using `renderNormal()` and the bottom entry is rendered using `renderEdit()`.

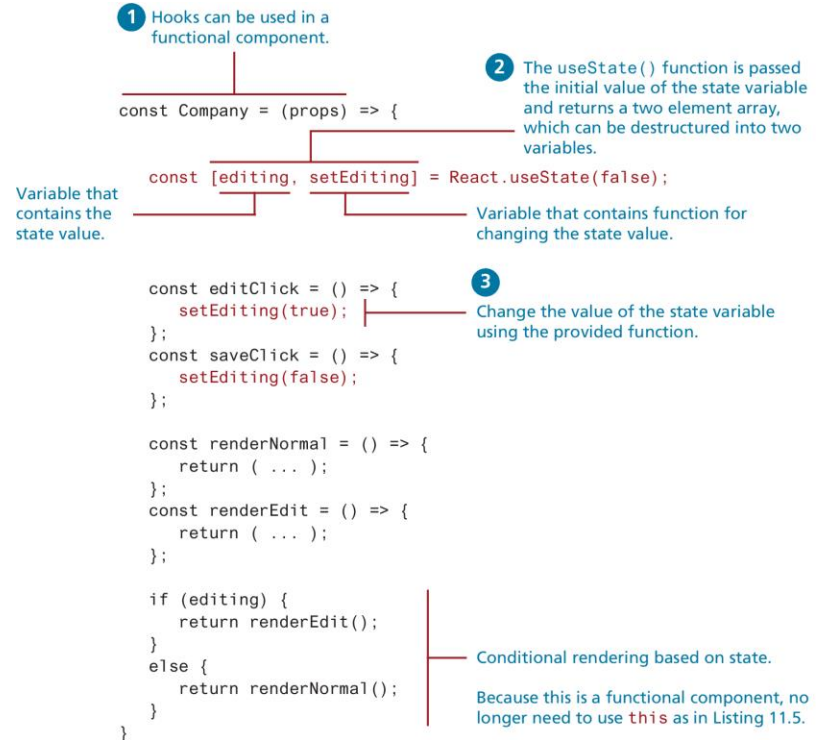
rendered using `renderNormal()`

rendered using `renderEdit()`

Using Hooks for State

Figure 11.9 illustrates the use of Hooks that is equivalent to that shown in Listing 11.5.

It is likely that functional components with Hooks will become the preferred approach amongst most developers moving forward.



Forms in React

With React, we can let the HTML form elements continue to maintain responsibility for their state (known as **uncontrolled form components**), or we can let the React components containing the form elements maintain the mutable state (these are known as **controlled form components**).

Controlled Form Components

You will set the form element's value using the value attribute and update the underlying state when the form element's value changes.

For instance, if you were using a class component, your render() function might have something similar to the following for one of the form elements:

```
<input type="text" name="sector" value={this.state.sector}  
onChange={this.handleSectorChange} />
```

The class component would then need to implement the handler method which would update the component state with the current value of the element:

```
class SampleForm extends React.Component {  
    constructor(props) {  
        super(props);  
        this.state = { sector: "", ... };  
    }  
    handleSectorChange = (e) => {  
        this.setState( {sector: e.target.value} );  
    }  
    ...  
}
```

Controlled form components using Hooks

```
const Company = (props) => {  
  ...  
  const renderEdit = () => {  
    return (  
      <article className="box media ">  
        <div className="media-content">  
          <h2><input type="text" value={name}  
            onChange={ (e) => setName( e.target.value ) } /></h2>  
          <p>Symbol: <input type="text" value={symbol}  
            onChange={ (e) => setSymbol( e.target.value ) } /></p>  
          <p>Sector: <input type="text" value={sector}  
            onChange={ (e) => setSector( e.target.value ) } /></p>  
          <p>HQ: <input type="text" value={hq}  
            onChange={ (e) => setHq( e.target.value ) } /></p>  
        </div>  
        ...  
      </article>  
    );  
  };  
  const [editing, setEditing] = React.useState(false);  
  
  // initialize the state variables to the data passed into the  
  // component  
  const [ name, setName] = React.useState( props.children );  
  const [ symbol, setSymbol] = React.useState( props.symbol );  
  const [ sector, setSector] = React.useState( props.sector );  
  const [ hq, setHq] = React.useState( props.hq );  
  ...  
}
```

LISTING 11.6 Controlled form components using Hooks

Uncontrolled Form Components

With uncontrolled form components, the form's state is managed by the DOM: you no longer need to create handler methods for form state updates

Instead, you retrieve the values from the form using a special ref object in React that allows you to access DOM nodes.

With the uncontrolled approach, you will generally create instance variables which will contain the references to the different DOM elements:

```
this.symbol = React.createRef();
```

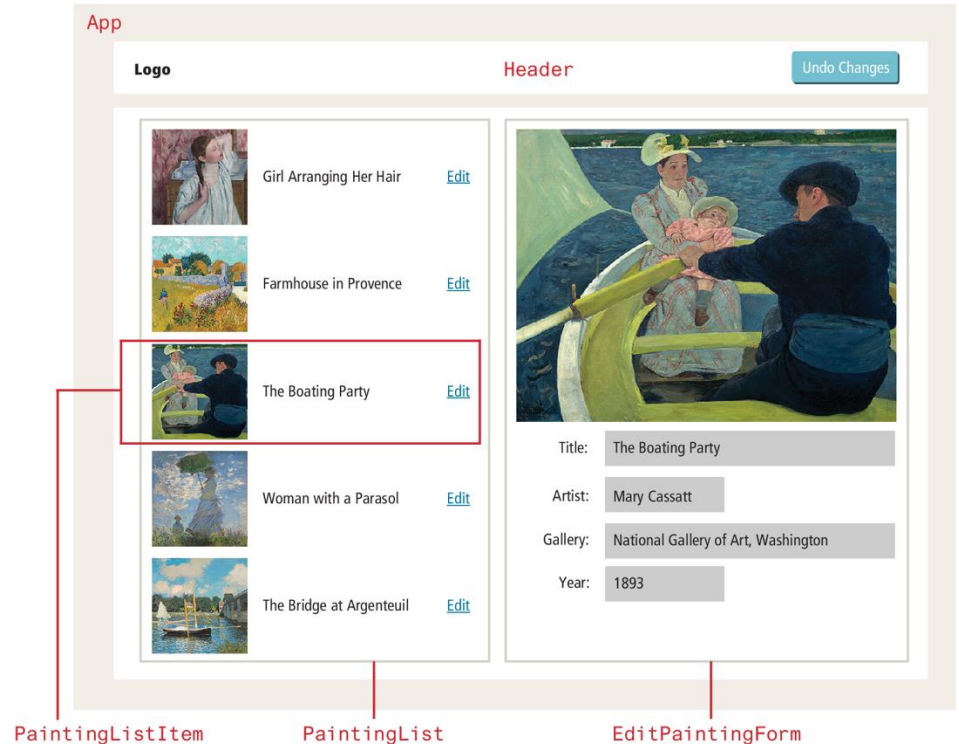
The current property of each ref object is used to reference the underlying DOM property or method.

Component Data Flow

In the page shown there are five components: App, Header, PaintingList, PaintingListItem, and EditPaintingForm. Which of these components appear to have data? Clearly, PaintingList, PaintingListItem, and EditPaintingForm.

Is the data they are displaying props data or state data?

Recall that props data is read-only

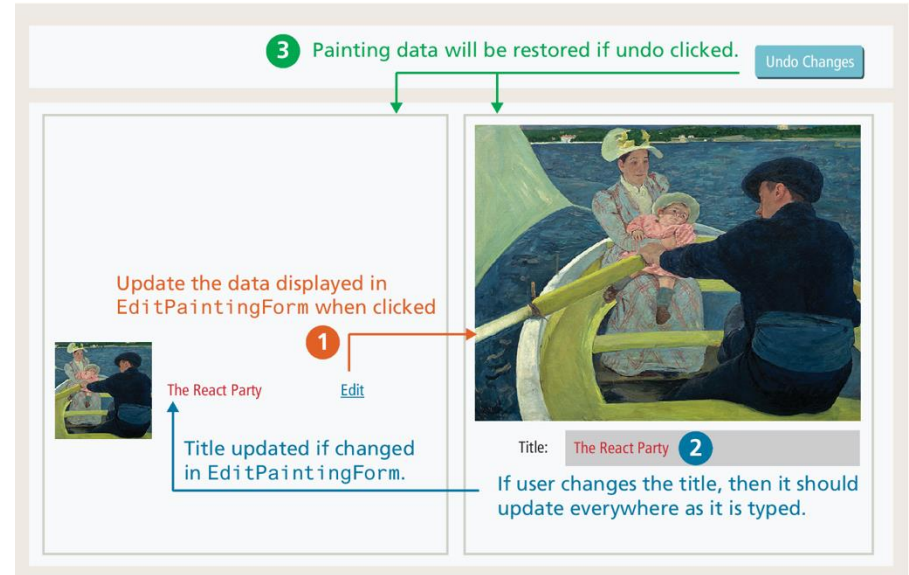


Data communication between components

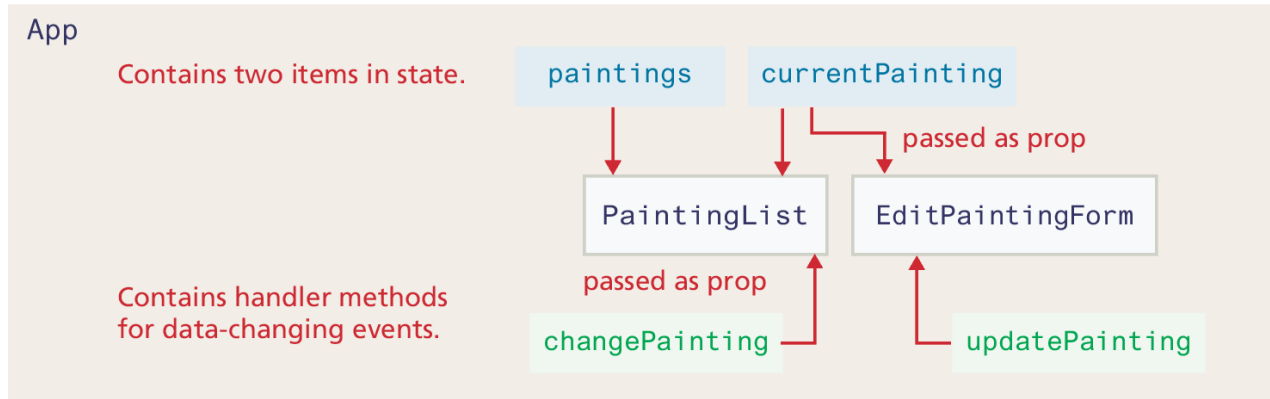
Different components need to communicate changes with each other. However, components can't access the state of their parents, siblings, or children.

The general solution in React is that changeable data will “belong” to the parent whose children will be using it.

Thus, for the page shown, the data will belong to the **App** component.



Implementing data flow between components



```
<article className="app">
  <Header undo={undoChanges} />
  <div className="box">
    <PaintingList paintings={paintings}
      current={currentPainting} change={changePainting} />
    <EditPaintingForm current={currentPainting} update={updatePainting} />
  </div>
</article>
```

React Build Approach

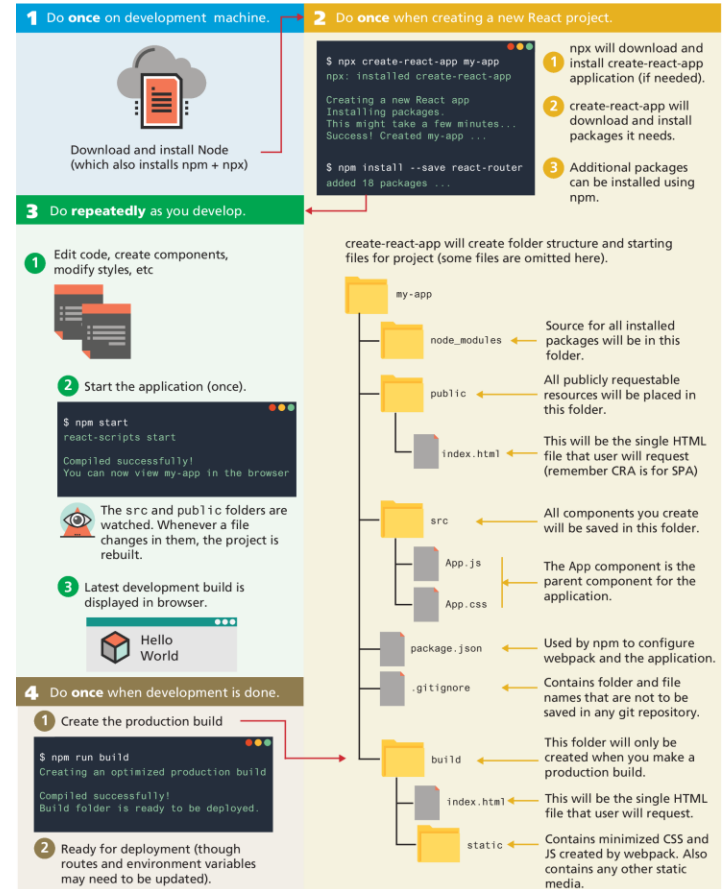
A **build tool** is used to build and manage your code base. It handles dependencies, but also handle a variety of other repetitive, complex, and tedious tasks including:

- Run transcompilers (e.g., JSX to JavaScript, TypeScript to JavaScript, SASS to CSS).
- Bundle files together into a single file or split a single large file into smaller files.
- Minify CSS and JavaScript.
- Create either development or production builds of projects.
- Run testing tools.
- Listen to folders for changes.
- Listen to folders for changes, and when one occurs, automatically run build tools and may even load file in browser.

Create React App

To make the process of creating React applications easier, a variety of pre-created starting solutions are available. **create-react-app** is from the React team (<https://github.com/facebook/create-react-app>).

Create React App (CRA) installs a CLI tool that installs boiler-plate starting files for a React project.



Other React Build Approaches

For creating a component library to be used by other developers a simpler tool chain such as Neutrino (<https://neutrinojs.org/>) or Parcel (<https://parceljs.org/>) might be a better choice.

To do JavaScript-to-HTML rendering on the server at design time you might use Next.js (<https://nextjs.org/>).

Gatsby (<https://www.gatsbyjs.org/>) is a framework for creating static websites using React. Behind the scenes, it uses GraphQL to access markdown files and external APIs for its data needs.

React Lifecycle

It is important to recognize that things happen in React in a certain order. You can programmatically interact with this order by working with the React lifecycle methods. Every component travels through a lifecycle of events.

- The **render()** event is triggered when the component is *mounted* (i.e., when it is created and inserted into the DOM) and when it is *updated*
- **componentDidMount()** is called after the first call to **render()**. This is typically where data would be fetched from an API.

React will call several events in a specific order; the most commonly used are **shouldComponentUpdate()**, **render()**, and then **componentDidUpdate()**.

Fetching Data

In React, we have to retrieve data at a specific point in the component's life cycle. For instance, to fetch a list of paintings from an API, you would write

```
class App extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { paintings: [] };  
  }  
  async componentDidMount() {  
    try {  
      const url  
      ="http://randyconnolly.com/funwebdev/3rd/api/art/paintings.  
      php";  
      const response = await fetch(url);  
      const jsonData = await response.json();  
      this.setState( {photos: jsonData} );  
    }  
    catch (error) {console.error(error);}  
  }  
  ...  
}
```

Fetching data with useEffect().

```
import React, { useEffect, useState } from 'react';
import HeaderApp from './components/HeaderApp.js';
import PhotoBrowser from
'./components/PhotoBrowser.js';

function App() {
  const [photos, setPhotos] = useState([]);
  useEffect( () => {
    const url = "...";
    fetch(url)
      .then( resp => resp.json() )
      .then( data => setPhotos(data))
      .catch( err => console.error(err));
  });

  return (
    <main>
      <HeaderApp />
      <PhotoBrowser photos={photos} />
    </main>
  );
}
```

LISTING 11.9 Fetching data with useEffect().

Extending React

Even if you limit yourself to just the basic features of React covered in this chapter so far, you can create reasonably complex single-page applications.

But what makes React such a popular development technology is the breadth and diversity of available component libraries.

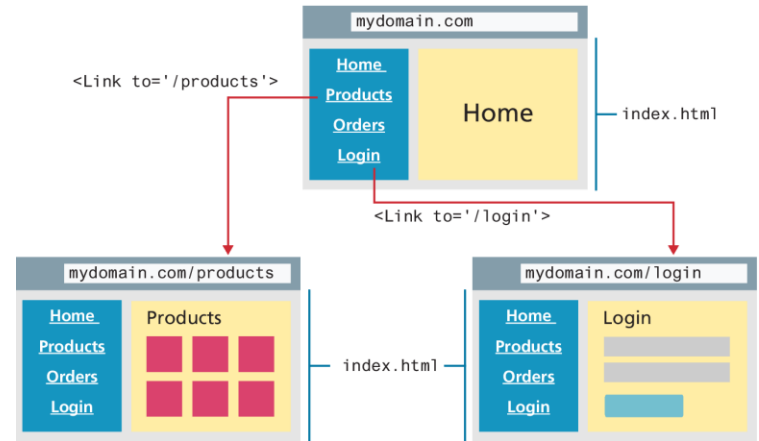
To make use of a library, the developer simply needs to install the package using npm.

React Routing

React Router is a package that allows a developer to configure routes.

What is a route? In normal HTML, you use hyperlinks (anchor tags) to jump from page to page in your application. But in a single-page application, there is only one page. Using React Router, you can turn links to different pages into links to display different React components.

Because users may still wish to make use of bookmarks, React Router will update the path in the browser address bar



Replacing hyperlinks with <Link> elements

```
import { Link } from 'react-router-dom';
const Menu = () => {
  return (
    <ul className="menu">
      <li><Link to="/">Home</Link></li>
      <li><Link to="/products">Products</Link></li>
      <li><Link to="/login">Login</Link></li>
    </ul>
  );
};
```

LISTING 11.10 Replacing hyperlinks with <Link> elements

Specifying components to render for different routes

```
import { Route } from 'react-router-dom';
import Home from './components/Home';
import Dialog from './components/Dialog';
import LoginForm from './components/LoginForm';
import ProductList from './components/ProductList';
const App = () => {
  ...

  return (
    <main>
      <Menu />
      <Route path="/" exact component={Home} />
      <Route path="/home" exact
        component={Home} />
      <Route path="/login" exact
        render= { () => <Dialog>
          <LoginForm />
        </Dialog> } />
      <Route path="/products" exact >
        <ProductList list={products} />
      </Route>
    </main>
  );
}
```

LISTING 11.11 Specifying components to render for different routes

CSS in React

You can continue working with CSS in the familiar manner of using an external CSS file brought in via a `<link>` element in the `<head>`.

It is also possible to define separate CSS files at the component level (e.g., `Header.css` used by `Header.js`) which are then imported, as shown in the following example:

```
import React from 'react';  
  
import './Header.css';  
  
function Header = props => { ... }
```

All the styles defined within `Header.css` will be merged by the build tool into a single global CSS file.

Using styled components

```
import React from 'react';
import styled from 'styled-components';
const ThumbImage = styled.img`
  width: 100px;
  height: 100px;
`;
const PhotoButton = styled.button`
  padding: 5px;
  font-size: 0.75em;
  border-radius: 3px;
  margin: 0 0.5em;
  min-width: 2.5em;
`;
```

```
const PhotoThumb = props => {
  const imgURL = '...';
  return (
    <div>
      <figure>
        <ThumbImage src={imgURL} alt={props.photo.title} />
      </figure>
      <div>
        <h3>{props.photo.title}</h3>
        <P>{props.photo.location.city},
          {props.photo.location.country}</P>
        <PhotoButton>View</PhotoButton>
        <PhotoButton>❤️</PhotoButton>
      </div>
    </div>
  );
}
export default PhotoThumb;
```

LISTING 11.12 Using styled components

Other Approaches to State

Using state in React typically requires the upper-most parent component to house the state variables and all behaviors that can modify this state. This prop-drilling tends to dramatically reduce the encapsulation of React child components, since they become dependent on their ancestors to pass in the data and behaviors they need as props.

The props-drilling approach can also be awkward when they are many components in an application that needs access to the same data.

As a consequence, many React developers decide to make use of some other React library for their application's state. This section will take a brief look at two approaches: the built-in **Context Provider** in React and the popular third-party **React Redux** library.

Context Provider

With the release of React Hooks in 2019, the **useContext()** hook provides a way to centralize data state into a single location known as a **context** which is available to both functional and class components.

The first step is to create a context provider: each bit of centralized state will have its own provider.

The context provider wraps any number of children components

Defining a context provider

```
import React, {useState, createContext } from 'react'  
// create the context object which will hold the state  
export const FavoriteContext = createContext();  
  
// create the object which will provide access to this context  
const FavoriteContextProvider = (props) => {  
  const [favorites, setFavorites] = useState([]);  
  return (  
    <FavoriteContext.Provider value={{favorites, setFavorites}} >  
      {props.children}  
    </FavoriteContext.Provider>  
  );  
};  
export default FavoriteContextProvider;
```

LISTING 11.13 Defining a context provider

Use a context provider

For instance, an App parent component would use this context as follows:

```
import FavoriteContextProvider from './contexts/FavoriteContext.js';  
...  
function App() {  
  ...  
    return (  
      <FavoriteContextProvider>  
        <Header />  
        <ArtBrowser paintings={paintings}/>  
      </FavoriteContextProvider>  
    );  
  }  
}
```

Context provider in action

1 When user clicks Add to Favorite button

Header

Art Store Browse Genres Galleries Artists

Favorites 2

ArtBrowser

PaintingCard

FavoriteList

2 Update favorites array stored in FavoriteContext ...

3 ... which will automatically update the displayed count.

The screenshot shows a web application interface for an art store. At the top is a blue header with the text 'Art Store' and navigation links 'Browse', 'Genres', 'Galleries', and 'Artists'. On the right of the header is a 'Favorites' button with a count of '2' and a user profile picture. Below the header is a grid of painting cards. Each card displays a painting image, its title, artist, and source (e.g., 'NATIONAL GALLERY OF ART' or 'J. PAUL GETTY MUSEUM'). At the bottom of each card are 'View' and 'Fav' buttons. A red arrow points from the 'Fav' button on the 'Portrait of Isabella of ...' card to the 'Add to Favorite' button in the 'Favorites' sidebar. The sidebar, titled 'Favorites', shows a list of favorite paintings and a count of '2'. The main grid contains eight painting cards: 'Madonna and Child' by Giotto, 'The Adoration of the ...' by Filippo Lippi, 'Portrait of Isabella of ...' by Rogier van der Weyden, 'Genevra de' Benci' by Leonardo da Vinci, 'Alba Madonna' by Raphael, 'Portrait of Bindo Alto...' by Raphael, 'Venus and Adonis' by Titian, and 'Christ on the Cross' by El Greco.

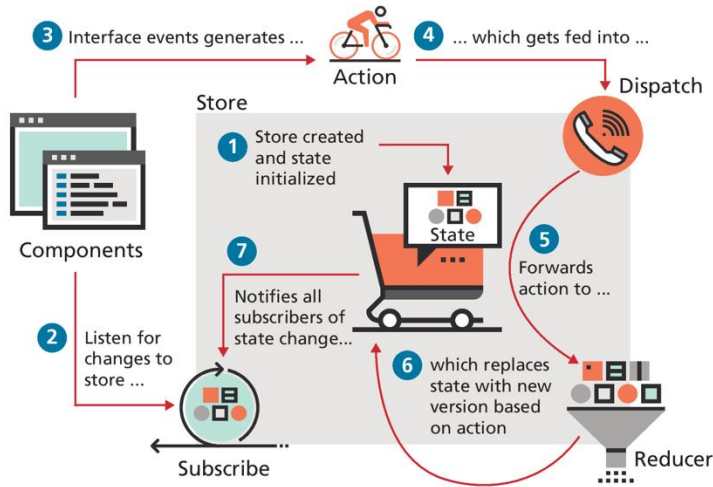
React Redux

The key idea of Redux is similar to that of the just-discussed Context Providers: namely, to store application state in a single data structure which is held in a storage area known as the **store**.

Unlike the Context Providers approach, in React the state is never mutated (changed) outside of the store. React follows functional programming principles in that functions never mutate state; instead, functions that create/return new versions of the state are used instead.

These functions are called **reducers** in Redux. A reducer is passed an **action** which is an object that contains a type property that indicates how the state should be changed.

Redux architecture



```
1 const initialState = {
  favorites: []
};
const store = createStore(reducer);
```

```
2 store.subscribe( () => {
  const state = store.getState();
  // update components with current state
  ...
});
```

```
3 AddFavClick( () => {
  // create favorite object to add to store
  const f = { id: ..., title: ... };
  // dispatch add-to-fav action with the data
4 store.dispatch( { type: 'ADD_TO_FAVS', payload: f });
});
```

```
6 const reducer = (state = initialState, action) => {
  if (action.type === 'ADD_TO_FAVS' ) {
    const newState = {...state};
    newState.favorites.push(action.payload);
    return newState;
  }
  else if (action.type === 'REMOVE_FROM_FAVS' ) {
    ...
  } else
    ...
};
```

Key Terms

action	create-react-app	prop-drilling	store
Angular	CSS-in-JS	pure functions	tagged template literal
build tool	currying	React	uncontrolled form
class components	dependencies	React Router	components
component	functional components	reducers	vanilla JavaScript
conditional rendering	functional composition	Single-Page Application	Vue.js
context	higher-order functions	(SPA)	webpack
controlled form components	jQuery	software framework	
	props	state	

Copyright



This work is protected by United States copyright laws and is provided solely for the use of instructors in teaching their courses and assessing student learning. Dissemination or sale of any part of this work (including on the World Wide Web) will destroy the integrity of the work and is not permitted. The work and materials from it should never be made available to students except by instructors using the accompanying text in their classes. All recipients of this work are expected to abide by these restrictions and to honor the intended pedagogical purposes and the needs of other instructors who rely on these materials.