React Components

The reader will learn how to use React and its component-based architecture in order to develop modern user interfaces. A new holistic way of thinking about UI development will be established throughout this book and the reader will discover the power of React components through many examples. After reading the book and following the example application, the reader has built a small to a mid-sized application with React using a component-based UI architecture. The book will take the reader on a journey to discover the benefits of component-based user interfaces over the classical MVC architecture. Throughout the book, the reader will develop a wide range of components and then bring them together to build a component-based UI. By the end of this book, readers will have learned several techniques of building powerful components and how component-based development is better than regular web development.

Who this book is written for
This book is ideal for developers who are familiar with the basics of React and are looking for a guide to building a wide range of components and who want to develop component-driven UIs.

What you will learn from this book
- Structure an app into components
- Work with nested components
- Set up communication across components
- Style the existing components
- Applying Material Design concepts to components
- Render components on the server
- Make the best of design patterns
- Make the app pluggable

Explore the power of React components for cutting-edge web development

Christopher Pitt

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Saving and Communicating Data'
- A synopsis of the book’s content
- More information on **React Components**
About the Author

Christopher Pitt is a principal developer for SilverStripe in Wellington, New Zealand. He usually works on open source software, though sometimes you'll find him building compilers and robots.
React is a fascinating new take on traditional frontend development. It has taken the JavaScript community by storm and has inspired sweeping changes in a number of existing JavaScript application frameworks and architectures.

Unfortunately, there still aren't many examples of great architecture. Most tutorials and books focus on small components and examples, leaving the question of larger applications and component hierarchies unanswered. That is what this book seeks to change.

**What this book covers**

*Chapter 1, Thinking in Components*, looks at the need to think of entire interfaces as a composition of small components and how to build them using modern ES6 JavaScript.

*Chapter 2, Working with Properties and State*, takes a comprehensive look at many aspects of property and state management, sharing a few more ES6 tricks along the way.

*Chapter 3, Saving and Communicating Data*, looks at reactive programming using event emitters and unidirectional flow of data.

*Chapter 4, Styling and Animating Components*, takes a look at how components can be styled and animated both inline and using stylesheets.

*Chapter 5, Going Material!*, explores material design and applies what you learn to our set of components.

*Chapter 6, Changing Views*, looks at ways of transitioning between different views with routing and animation.
Chapter 7, Rendering on the Server, takes a look at the process of rendering components through nodes and some ways of structuring server-side application code.

Chapter 8, React Design Patterns, explores different architectures such as Flux and Redux.

Chapter 9, Thinking of Plugins, looks at how to build components with dependency injection and extension points.

Chapter 10, Testing Components, explores various ways of ensuring that components are error-free and that changes to parts of an application don't have cascading effects on other parts.
In the previous chapter, we created complex component hierarchies. We created a list of pages and a way to edit those pages. Yet we stopped short of saving and reading any of that data to some kind of storage.

We could, for instance, send an edit through an Ajax request to be saved in a database server. In fact, that's what often happens in the applications we use these days. They always save our interactions, irrespective of whether we expect them to or not.

In this chapter, you will learn about local data stores and communicating with them. You'll also learn about event-based architecture and how it promotes the unidirectional flow of data.

There are many ways to save data. It's a rich and interesting topic that could fill scores of books. I could go so far as to say it is at the core of how businesses and applications work.

Furthermore, how data is communicated can often be different in a maintainable application and an unmaintainable application. It's up to us to figure out elegant ways of persisting data so that our applications remain maintainable.

We will only explore local storage in this chapter. You'll be able to see your stored data beyond page reloads, but nobody else will. You cannot build a practical website based on this chapter alone. You will have to wait until we explore React on the server.
Validating properties

Before we look at storing data, there is another habit I’d like to share with you. The components we created in the last chapter work well together, but our aim is to make each component self-contained. We want others to be able to reuse our components, but they will encounter problems if they don’t know which properties our components expect.

Consider what would happen if we used PageAdmin like this:

```javascript
ReactDOM.render(
    <PageAdmin backend="ajax" />,
    document.querySelector(".react")
);
```

Faced with this component, and no documentation, it might be tempting to substitute a Backend object with some other configuration data. This looks reasonable to someone unfamiliar with the component. And, without a careful study of all our components, we can’t expect others to know what those properties should be.

We can protect against this situation by adding property validation. Let's add some validation to PageEditor:

```javascript
PageEditor.propTypes = {
    "id": React.PropTypes.number.isRequired,
    "title": React.PropTypes.string.isRequired,
    "body": React.PropTypes.string.isRequired,
    "onUpdate": React.PropTypes.func.isRequired,
    "onCancel": React.PropTypes.func.isRequired
};
```

We have already imported the React object, which exposes a PropTypes object. This contains some validators. When we specify a few on PageEditor.propTypes, React checks the types of properties given to the component as it is rendered. If we give the incorrect property types or omit required properties, React will emit a warning.

The warnings look like this:

```
warning: Failed prop_type 'id' was not specified in 'PageEditor'.
react.js:18746
warning: Failed prop_type 'title' was not specified in 'PageEditor'.
react.js:18746
warning: Failed prop_type 'body' was not specified in 'PageEditor'.
react.js:18746
warning: Failed prop_type 'onUpdate' was not specified in 'PageEditor'.
react.js:18746
warning: Failed prop_type 'onCancel' was not specified in 'PageEditor'.
react.js:18746
```
There are many types to choose from, the simple ones being the following:

- `React.PropTypes.array`
- `React.PropTypes.bool`
- `React.PropTypes.func`
- `React.PropTypes.number`
- `React.PropTypes.object`
- `React.PropTypes.string`

If you need a property to be required (which is likely in most cases) then you can add `.isRequired` at the end. Let's follow this up with validators for `PageView`:

```javascript
PageView.propTypes = {
  "title": React.PropTypes.string.isRequired,
  "onEdit": React.PropTypes.func.isRequired,
  "onDelete": React.PropTypes.func.isRequired
};
```

This is even simpler, given that `PageView` uses fewer properties than `PageEditor`. Also, `Page` is relatively simple:

```javascript
Page.propTypes = {
  "id": React.PropTypes.number.isRequired,
  "onDelete": React.PropTypes.func.isRequired
};
```

We don't need to validate properties passed straight through components. For instance, `PageEditor` uses `onUpdate`. It's passed through `Page`, but `Page` doesn't use it, `PageEditor` does, so that's where we use validators for it.

However, what if we want to validate nested structures or more complex types? We can try the following:

```javascript
PageAdmin.propTypes = {
  "backend": function(props, propName, componentName) {
    if (props.backend instanceof Backend) {
      return;
    }
    return new Error(
      "Required prop `backend` is not a `Backend`.
      ");
  }
};
```
We expect the `backend` property to be an instance of the `Backend` class. If it is anything else, we return an `Error` describing why the property is invalid. We can also use `shape` to validate nested properties:

```js
Component.propTypes = {
  "page": React.PropTypes.shape({
    "id": React.PropTypes.number.isRequired,
    "title": React.PropTypes.string.isRequired,
    "body": React.PropTypes.string.isRequired
  })
};
```

The more specific we are about properties, the less chance there is for bad properties to break the interface. So, it's good to get in the habit of defining them all the time.

### Storing cookies

You must have heard of cookies before. They're a browser-based storage mechanism as old as the Internet, and they are often comically described in movies. Here's how we use them:

```js
document.cookie = "pages=all_the_pages";
document.cookie = "current=current_page_id";
```

The `document.cookie` parameter works as a temporary string store. You can keep adding new strings, where the key and value are separated by `=`, and they will be stored beyond a page reload, that is, until you reach the limit of how many cookies your browser will store per domain. If you set `document.cookie` multiple times, multiple cookies will be set.

You can read the cookies back again, with a function like this:

```js
var cookies = {};

function readCookie(name) {
  var chunks = document.cookie.split("; ");

  for (var i = chunks.length - 1; i >= 0; i--) {
    var parts = chunks[i].split("=");
    cookies[parts[0]] = parts[1];
  }

  return cookies[name];
}

export default readCookie;
```
The whole cookie string is read and split using semicolons. Then, each cookie is split into equals, leaving the key and value. These are stored in the local cookies object. Future requests just read the key from the local object. The cookies object can be inspected at any point to see the cookies that have been set.

Try http://browsercookielimits.squawky.net to test what your browser can handle. I'm running a modern version of Chrome, and I can probably store 180 cookies per domain, totaling 4096 bytes. 4096 bytes doesn't sound like a lot...

Cookies aren't typically used for the kinds of data we want to store. We'll have to look elsewhere.

If you want to learn more about how to use cookies, head over to https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie.

Using local storage

The next type of storage we will look at is a relatively recent addition to the browser toolset. It's called local storage, and it's been around for a while. You can add items to it as follows:

```javascript
localStorage.setItem("pages", "all_the_pages");
```

It's simpler than cookies to read items from:

```javascript
localStorage.getItem("pages");
```

This will persist the data beyond page reloads or the browser closing. You can store considerably more data than in cookies (anywhere from 3 MB to 10 MB, by default), and the interface is easier to use.

So, how can we use this to store our pages? Let's abstract local storage a bit:

```javascript
export default {
  "get": function(key, defaultValue) {
    var value = window.localStorage.getItem(key);

    var decoded = JSON.parse(value);

    if (decoded) {
      return decoded;
    }
  }
};
```
Saving and Communicating Data

```javascript
return defaultValue;
},

"set": function(key, value) {
  window.localStorage.setItem(key, JSON.stringify(value));
}
};

For once, we're exporting an object instead of a class. This object has a couple of methods both of which access window.localStorage. It's not ideal to reference this directly, but if we use this abstraction everywhere else, then I think it's OK.

The `get` method pulls a string value out of local storage and parses it as a JSON string. If the value parses to any non-false value, we return it, or else we return a default value.

The `set` method encodes a value as JSON, and stores it.

Then, we can use the following abstraction in the `Backend` class:

```javascript
import LocalStore from "local-store";

class Backend {
  constructor() {
    this.pages = LocalStore.get("pages", []);
  }

  getAll() {
    return this.pages;
  }

  update(id, property, value) {
    this.pages = this.pages.map((page) => {
      if (page.id == id) {
        page[property] = value;
      }
      return page;
    });

    LocalStore.set("pages", this.pages);
  }
}
```
delete(id) {
    this.pages = this.pages.filter(
        (page) => page.id !== id
    );
    
    LocalStore.set("pages", this.pages);
}
}

export default Backend;

We begin with a constructor that fetches any stored pages from localStorage. We provide a default empty array in case the pages key is missing in localStorage. We store that in this.pages so we can fetch and modify it later.

The getAll method is much simpler this time around. All it does is return this.pages. The update and delete methods become more interesting though. The update method uses the Array.map method to apply updates to the affected page objects. We have to store the updated pages array back in local storage so that the changes are persisted.

Similarly, delete modifies the pages array (this time with a short function syntax) and stores the modified array back in local storage. We have to see local storage with some initial data. You can do this in a developer console:

```javascript
localStorage.setItem("pages", JSON.stringify([{
    "id": 1,
    "title": "Home",
    "body": "...
  },
  {
    "id": 2,
    "title": "About Us",
    "body": "...
  },
  {
    "id": 3,
    "title": "Contact Us",
    "body": "...
  },
  {
    "id": 4,
    "title": "Products",
```

```javascript
]);
```

```javascript
LocalStore.set("pages", this.pages);
```
Saving and Communicating Data

"body": "...
}
]));

If you've made these changes, and you refresh the page, you should see the new backend code in action!

Using event emitters

Until now, our components have communicated with the backend through method calls. That's OK for tiny applications, but when things start to scale, we will forget to make some of those method calls.

Look at `onUpdate`, for instance:

```javascript
onUpdate(id, field, value) {
    this.props.backend.update(id, field, value);

    this.setState({
        "pages": this.props.backend.getAll()
    });
}
```

Every time we change the state of a page, we have to fetch an updated list of pages from the backend. What if multiple components send updates to the backend? How will our `PageAdmin` component know when to fetch a new list of pages?

We can turn to event-based architecture to solve this problem. We've already encountered and used events! Recollect what we did when we created the page edit form. There, we connected to input events so we could update pages when input values changed.

This kind of architecture moves us closer to a unidirectional flow of data. We can imagine our entire application like a tree of components, beginning with a single root component. When a component needs to update some application's state, we don't need to code the state change in relation to where that component is. In the past, we may have had to reference specific CSS selectors, or depend on the position of sibling elements, when updating state.

When we start to use events, then any component can trigger a change in the application. Also, multiple components can trigger the same kind of change. We'll explore this idea in more detail in later chapters.
We can use that same idea to notify components when the data changes. To begin with, we need to download an event emitter class:

```
$ npm install --save eventemitter3
```

Now, `Backend` can extend this, providing the event functionality that we are after:

```javascript
class Backend extends EventEmitter {
  constructor() {
    super();

    this.pages = LocalStore.get("pages", []);
  }

  getAll() {
    return this.pages;
  }

  update(id, property, value) {
    // ...update a page

    this.emit("update", this.pages);
  }

  delete(id) {
    // ...delete a page

    this.emit("update", this.pages);
  }
}
```

As each page is updated or deleted, the backend will emit an event on itself. This does nothing until we listen for these events in `PageAdmin`:

```javascript
constructor(props) {
  super(props);

  this.bind(
    "onUpdate",
    "onDelete"
  );

  this.state = {
    "pages": this.props.backend.getAll()
  };
```
this.props.backend.on("update",
    (pages) => this.setState({pages})
);}

onUpdate(id, field, value) {
    this.props.backend.update(id, field, value);
}

onDelete(id) {
    this.props.backend.delete(id);
}

Now we can remove the numerous calls to this.setState and replace them with a single event listener in the constructor. We are also doing something interesting with the setState call. It's called object destructuring, and it allows {pages} to become {"pages":pages}.

Now we can begin to use this backend for many different parts of the interface, and they'll all have accurate, real-time data. Open the page up in a few different windows and watch them all update at once!

**Summary**

In this chapter, we looked at how to protect our components from faulty properties. We also saw how easy it was to use cookies, although they are limited for what we need. Fortunately, we can use local storage and work it into our existing backend and components.

Finally, we explored using events to push state changes out to all interested components.

In the next chapter, we will start prettying up our components. We'll look at ways to style and animate them, bringing our interface to life!
Where to buy this book

You can buy React Components from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.