Functional React

SECOND EDITION

Quick start with
React Hooks, Redux and MobX

Cristian Salcescu

Functional React

SECOND EDITION

Quick start with
React Hooks, Redux and MobX

Cristian Salcescu

Functional React, 2nd Edition

Quick start with React Hooks, Redux and MobX

Cristian Salcescu

May 2019 First Edition
May 2020 Second Edition

Table of contents

Preface

React is one of the most popular JavaScript libraries for creating user interfaces.

The basic unit for a piece of the user interface is the component. With React Hooks we can express all the user interface using function components.

This is a book for beginners in React, but not for beginners in JavaScript. For a better understanding of JavaScript and functional programming consider reading Discover Functional JavaScript .

Through this book, we are going to build a shopping cart. We will build all components using functions. Classes will not be used to create components or any other objects. More than that, the `this` pseudo-parameter will not be used at all.

We are going to look at state management with React Hooks and with external libraries like Redux or MobX .

React embraces functional programming. Programming in a functional style means using concepts such as pure functions, immutability, closures, first-class functions, higher-order functions, or currying. We are going to explore how to apply these functional programming techniques in React.

Mastering the fundamentals makes us better at solving more complex tasks.

Source Code

The project files from this book are available at https://github.com/cristi-salcescu/functional-react-2 .

Feedback

I will be glad to hear your feedback. For comments, questions, or suggestions regarding this book send me an email to cristisalcescu@gmail.com .

Fast Development Environment

The first thing we need to do is to set-up our development environment.

Package Manager

A package manager is a tool used to track project dependencies in an easy to use manner. At the time of writing, Node.js package manager, in short `npm` , is the most popular. Let's start by installing Node.js .

The following commands can then be used in command prompt to check the Node.js and npm versions:

```
node --version
```

```
npm --v
```

NPM Packages

With `npm` we can install additional packages. These packages are the application dependencies.

For example, here is the command for installing the Redux package:

```
npm install --save redux
```

The installed packages can be found in the `node_modules` folder. The `--save` flag tells npm to store the package requirement in the `package.json` file.

The `package.json` file stores all the node packages used in the project. These packages are the application dependencies. The application can be shared with other developers without sharing all the node packages. Installing all the packages defined in the `package.json` file can be done using the `npm install` command.

Create React App

The easiest way to start with a React application is to use Create React App
.

To do that, run one of the following commands:

```
npm init react-app appname
```

```
npx create-react-app appname
```

`npx` can execute a package that wasn’t previously installed.

Once the application is created the following commands can be used:

- `npm start` : starts the development server.
- `npm test` : starts the test runner.
- `npm run build` : builds the app for production into the `build` folder.

IDE

For code editing, we need an Integrated Development Environment, IDE in short.

My favorite is Visual Studio Code .

To start the application, first, open the application folder in Visual Studio Code. Then open the terminal from Terminal→New Terminal and run: `npm start` . This launches the development server and opens the React application in the browser.

Chapter 01: Functional JavaScript

One great thing in JavaScript is functions. Functions are treated as values. We can simply define a variable, storing a function. Here is an example:

```
const sum = function(x, y){
 return x + y;
}

sum(1, 2);
//3
```

The same function can be written using the arrow syntax as follows:

```
const sum = (x, y) => x + y;
```

We favor declaring functions using the `const` keyword instead of the `let` keyword to prevent its modification.

Like other values, functions can be stored in objects, arrays, passed as arguments to other functions or returned from functions.

Pure Functions

A pure function is a computation. It computes the result using only its input. Given a specific input, it always returns the same output.

A pure function has no side-effects. It does not use variables from outside its scope that can change. It does not modify the outside environment in any way.

Pure functions have a big set of advantages. They allow focusing attention in one place, the function itself without its surrounding environment, and that makes it easier to read, understand, and debug.

Immutability

> An immutable object is an object that cannot be modified after it is created.

Immutable objects can be created using `Object.freeze()` .

```
const product = Object.freeze({
 name : "apple",
 quantity : 1
});

product.quantity = 2;
//Cannot assign to read only property 'quantity'
```

`Object.freeze()` does a shallow freeze and as such the nested objects can be changed. For a deep freeze, we need to recursively freeze each property of type object. Check the deepFreeze() implementation.

Changing an immutable value means creating a changed copy. For example, if we want to modify the `quantity` of the previous `product` , we create a new object.

```
const newProduct = Object.freeze({
 ...product,
 quantity: 2
});
```

The spread syntax is helpful when creating copies of objects.

In functional programming, all the data transferred between functions are immutable or at least treated as being immutable.

Functional Array

In the functional style, we get rid of the `for` statement and use the array methods.

Before going forward, note that primitives, objects, and functions are all values.

filter

> `filter()` selects values from a list using a predicate function deciding what values to keep.

```
const numbers = [1, 2, 3];

function isOdd(number){
 return number % 2 !== 0;
}

const oddNumbers = numbers.filter(n => isOdd(n))
//element=1, isOdd(1) is true  → result=[1]
//element=2, isOdd(2) is false → result=[1]
//element=3, isOdd(3) is true  → result=[1,3]
//[1, 3]
```

> A predicate function is a function that takes one input value and returns `true` or `false` based on whether the value satisfies the condition.

`isOdd()` is a predicate function.

Point-free style

> Point-free is a technique that improves readability by eliminating the unnecessary arguments.

Let’s reconsider the previous code :

```
const numbers = [1, 2, 3];
const oddNumbers = numbers.filter(n => isOdd(n))
```

In a point-free style, the same code will be written without arguments:

```
const oddNumbers = numbers.filter(isOdd)
```

map

`map()` transforms a list of values to another list of values using a mapping function.

```
function toStar(number){
 return '*'.repeat(number);
}

const numbers = [1, 2, 3];
const stars = numbers.map(toStar);

//element=1, map to '*'   → result=['*']
//element=2, map to '**'  → result=['*', '**']
//element=3, map to '***' → result=['*', '**', '***']
//['a', 'aa', 'aaa']
```

`toStar()` is a mapping function. It takes a number and returns the corresponding number of star `*` characters.

reduce

`reduce()` reduces a list of values to a single value.

`reduce(reducer, initialValue)` takes two arguments.

The first argument, the reducer, is a function that takes an accumulator and the current value and computes the new accumulator. `reduce()` executes the reducer function for each element in the array.

The second argument is the initial value for the accumulator. If no value is provided, it sets the first element in the array as the initial value and starts the loop from the second element.

```
function sum(total, number){
 return total + number;
}

const numbers = [1, 2, 3];
const total = numbers.reduce(sum, 0);
//element=1, total=0 → result=1
//element=2, total=1 → result=3
//element=3, total=3 → result=6
//6
```

Chaining

> Chaining is a technique used to simplify code by appling multiple methods on an object one after another.

Below is an example where the `filter()` and `map()` methods are both applied to an array:

```
const numbers = [1, 2, 3];

const stars = numbers
 .filter(isOdd)
 .map(toStar);

//["*", "***"]
```

Closures

> Closure is an inner function that has access to variables from outer function, even after the outer function has executed.

Consider the next example:

```
function createIncrement(){
 let count = 0;
 return function increment(){
  count += 1;
  return count;
 }
}

const doIncrement = createIncrement();
doIncrement(); //1
doIncrement(); //2
doIncrement(); //3
```

`increment()` is an inner function and `createIncrement()` is its parent. `increment()` has access to the variable `count` from its parent event after `createIncrement()` has been executed. `increment()` is a closure.

Higher-order functions

> A higher-order function is a function that takes another function as an input, returns a function, or does both.

`filter()` , `map()` , `reduce()` are higher-order functions.

Currying

> Currying is the process of transforming a function with several parameters into a series of functions each taking a single parameter.

Currying comes in handy when aiming for pure functions. Consider the next code:

```
const fruits = ['apple', 'mango', 'orange']

const newFruits = fruits.filter(function(name) {
return name.startsWith("a")
})
```

Now, let’s refactor out the filter callback to a pure function with an intention-revealing name.

```
const fruits = ['apple', 'mango', 'orange'];

function startsWith(text, name){
 return name.startsWith(text)
}

const newFruits = fruits.filter(name => startsWith('a', name))
```

As you can see, `startsWith()` has two parameters and `filter()` needs a callback expecting one parameter.

We can write `startsWith()` as a curried function:

```
function startsWith(text){
 return function(name){
  return name.startsWith(text)
 }
}
```

Then, it can be called providing only one parameter, the text to search for.

```
const fruits = ['apple', 'mango', 'orange'];

const newFruits = fruits.filter(startsWith('a'));
```

Final Thoughts

Functions are values and thus they can be treated as any other values.

A closure is a function that has access to the variables and parameters of its outer function, even after the outer function has executed.

Pure functions return a value using only on its input and have no side-effects.

An immutable value is a value that once created cannot be changed. Changing immutable values means creating changed copies.

`filter()` , `map()` and `reduce()` are the core methods for working with collections in a functional style.

Chapter 02: Functional Components

React is a library for building user interfaces. The basic unit in React is the component. The main idea is to split the page into small, reusable components and then combine them together to create the whole page.

The small components are easier to reason about and reusable.

Functional Components

Functional components are functions returning an HTML-like syntax called JSX .

React allows expressing the UI using functions.

The following `Header` component is a function that returns a `header` element.

```
import React from 'react';

function Header() {
 return (
  <header>
   <h1>A Shopping Cart</h1>
  </header>
 );
}

export default Header;
```

The returned HTML-like code is wrapped in parentheses to avoid the problems with automatic semi-colon insertion.

The `Header` component can then be used in another component called `App` .

```
import React from 'react';
import Header from './Header';

function App() {
 return (
  <div>
   <Header />
  </div>
 );
}

export default App;
```

At this point, we have established a relation between `App` and `Header` . `App` is the parent of `Header` .

Components are organized in a tree-like structure with a root component. In this case, the root component is `App` .

Props

Functional components can take properties.

The following `ProductItem` component takes a `product` and generates the user interface for it.

```
import React from 'react';

function ProductItem(props) {
 return (
  <div>{props.product.name}</div>
 );
}

export default ProductItem;
```

Functional components take a single parameter called “props” and return the user interface markup.

We can make the code cleaner by using the destructuring assignment syntax in the parameter list.

```
function ProductItem({product}) {
 return (
  <div>{product.name}</div>
 );
}
```

Note that `ProductItem(props)` was replaced with `ProductItem({product})`.

Props can be passed to components using JSX attributes. Below a `product` is created and passed to the `ProductItem` component:

```
import React from 'react';
import Header from './Header';
import ProductItem from './ProductItem';

const product = {
 id: 1,
 name : 'mango'
};

function App() {
 return (
  <div>
   <Header />
   <div>
  <ProductItem product={product} />
   </div>
```

```
    </div>
  );
}

export default App;
```

React passes all the component attributes in the “props” object.

Entry Point

The application has a single entry point. In our case, it is the `index.js` file. This is the place where the root component `App` is rendered. When the root component is rendered, all the other components are rendered.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

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

Virtual DOM and Rendering

React elements are plain objects describing the UI and are cheap to create.

Functional components are function returning React elements.

React elements are immutable, once created they cannot be changed. The only way to update the UI is by creating a new React element.

`ReactDOM` updates the DOM based on the React elements.

`ReactDOM.render(element, containerDOM)` renders a React element, including all its children. The first argument is the React element, the second argument is the target DOM element. It renders the component inside the root DOM element specified. Usually, there is a single root DOM element.

Consider for example the `root` DOM element:

```
<div id="root"></div>
```

The code in the `index.js` file renders the React element created by the `App` functional component into the `root` DOM node.

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

The Document Object Model, DOM, is an in-memory object representation of the HTML document. The DOM offers an application programming interface, that allows to read and manipulate the page's content, structure, styles and handle HTML events. The DOM is a tree-like structure.

DOM updates are expensive.

The Virtual DOM is an in-memory representation of the user interface. It is used by the React diff algorithm to identify only the necessary changes for updating the "real" DOM. Even if we create a React element describing the whole UI tree, only the necessary changes are applied to the real DOM.

The "real" DOM is made of DOM elements. The virtual DOM is made of React elements.

The virtual DOM can be inspected using the React Developer Tools extension.

Final Thoughts

Functional components are functions returning objects describing part o the user interface.

React makes it possible to split the page into small functions rendering parts of the user interface and combine these parts to create the whole page.

Functional components transform data into a visual user interface.

Chapter 03: A Brief Introduction to JSX

JSX describes how the user interface looks like. It offers a friendly HTML-like syntax for creating DOM elements.

JSX stands for JavaScript XML.

Transpiling

The JSX code is transpiled to JavaScript. Check the code below:

```
//JSX
ReactDOM.render(
 <header>
   <h1>A Shopping Cart</h1>
 </header>,
 document.getElementById('root')
)

//Transpiled to JavaScript
ReactDOM.render(
 React.createElement('header', null,
  React.createElement('h1', null, 'A Shopping Cart')
 ),
 document.getElementById('root')
);
```

As you can see, JSX is transpiled to `React.createElement()` calls, which create React elements.

Expressions

JSX accepts any valid JavaScript expression inside curly braces.

Here are some examples of expressions:

```
function Message(){
 const message = '1+1=2';
 return <div>{message}</div>
}

function Message(){
 return <div>1+1={1+1}</div>
}

function Message(){
 const sum = (a, b) => a + b;
 return <div>1+1={sum(1, 1)}</div>
}

ReactDOM.render(<Message/>, document.querySelector('#root'));
```

An expression is a valid unit of code that resolves to a value.

For example, the conditional ternary operator is an expression, but the `if` statement is not.

```
function Valid({valid}){
 return <span>{ valid ? 'Valid' : 'Invalid'}</span>
}

ReactDOM.render(<Valid valid={false} />,
document.getElementById('root'));
//<span>Invalid</span>
```

JSX is also an expression. It can be used inside statements, assigned to variables, passed as arguments.

Below, the previous component is rewritten with JSX inside the `if` statement.

```
function Valid({valid}){
 if(valid){
  return <span>Valid</span>
 } else {
  return <span>Invalid</span>
 }
}

ReactDOM.render(<Valid valid={false}/>,
document.querySelector('#root'));
```

Here is the same logic implemented by assigning JSX to a variable:

```
function Valid({valid}){
 let resultJSX;
 if(valid){
  resultJSX = <span>Valid</span>;
 } else {
  resultJSX = <span>Invalid</span>;
 }

 return resultJSX;
}

ReactDOM.render(<Valid valid={false}/>,
document.querySelector('#root'));
```

Escaping string expressions

JSX does automatic escaping of string expressions.

```
function Message(){
 const messageText = '<b>learn</b>';
 return <span>{messageText}</span>;
}

ReactDOM.render(<Message />,
 document.querySelector('#root'));
```

The text inside `messageText` is escaped, thus `<b>learn</b>` will be displayed as a text.

Below the previous code is rewritten with JSX instead of a string.

```
function Message(){
 const messageJSX = <b>learn</b>;
 return <span>{messageJSX}</span>;
}

ReactDOM.render(<Message />,
 document.querySelector('#root'));
```

This time, the `"learn"` text is displayed in bold.

Syntax

Function components can return a single tag. When we want to return multiple tags we need to wrap them into a new tag.

All tags need to be closed. For example, `<br>` must be written as `<br />` .

Since JSX is transpiled into `React.createElement()` calls, the React library must be available when using JSX.

```
import React from 'react';
```

Components must start with a capital letter. The following code doesn't work correctly:

```
function product(){
 return <div>A product</div>
}

ReactDOM.render(<product />, document.querySelector('#root'));
```

In JSX, lower-case tag names are considered to be HTML tags, not React components.

The automatic semi-colon insertion may create problems when using `return` . The following function component will not return a React element, but `undefined` .

```
function Product(){
 return
  <div>
   A product
  </div>
}
```

A common practice, to avoid this problem, is to open parentheses on the same line as `return` .

```
function Product(){
 return (
  <div>
   A product
  </div>
 );
}
```

Attributes

The attributes in JSX use camelCase. For example `tabindex` becomes `tabIndex` , `onclick` becomes `onClick` .

Other attributes have different names than in HTML: `class` becomes `className` , `for` becomes `htmlFor` .

Consider the `Valid` component:

```
function Valid({valid}){
 return <span>{ valid ? 'Valid' : 'Invalid'}</span>
}
```

Curly braces can be used to specify an expression in an attribute:

```
<Valid valid={false}/>
```

In this case, the `valid` property of the props object gets the value `false` .

Quotes can be used to specify a string literal in an attribute:

```
<Valid valid="false" />
```

Now, the `valid` property gets the string `"false"` .

Consider the next example:

```
<Valid valid="{false}" />
```

In this case, the `valid` property gets the string `"{false}"`

For attributes, we should use quotes to assign string values or curly braces to assign expressions, but not both.

Final Thoughts

JSX allows using markup directly in JS. In a sense, JSX is a mix of JS and HTML.

JSX is transpiled to `React.createElement()` calls, which create React elements.

JSX accepts expression inside curly braces. String expressions are escaped.

Attributes in JSX use camelCase.

Chapter 04: List Components

A functional component transforms a value into a React element.

A list component transforms a list of values into a list of React elements.

Let’s understand how. First, remember that `map()` transforms a list of values into another list of values using a mapping function.

ProductList Component

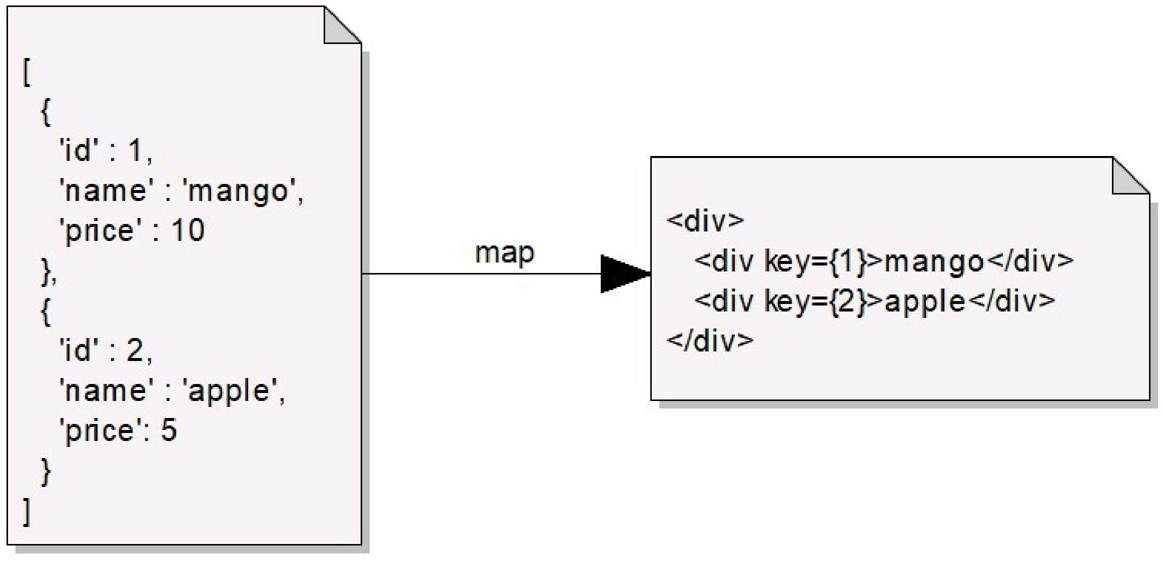
The `ProductItem` component creates the interface for a single product.

The `ProductList` component creates the interface for a list of products using `map()` and the `ProductItem` component.

```
import React from 'react';
import ProductItem from './ProductItem';

function ProductList({products}) {
 return (
  <div>
   {products.map(product =>
    <ProductItem
     key={product.id}
     product={product}
    />
   )}
  </div>
 );
}

export default ProductList;
```

`<ProductItem product={product} />` renders one product using the `ProductItem` component. It is the equivalent of the `ProductItem({product})` function call.

Here is how the list transformation can be imagined.



The same way we use `map()` to transform a list of values into another list of values, we can use `map()` to transform a list of values into a list of React elements.

List Item Key

React requires to uniquely identify each item in a list.

The `key` is a special attribute that is required in a list of elements. It helps the React diff algorithm to decide what DOM elements to update. Each item in a list should have a unique identity defined by the `key` attribute. Usually, we are going to use ids as keys. Note the use of `product.id` as the key to each `ProductItem`.

```
<ProductItem product={product} key={product.id} />
```

Keys are useful only when rendering a list. The `key` should be kept on the `<ProductItem />` elements in the array and not in the `ProductItem` itself.

Keys are not passed to components. When we need the same value in the component we need to pass it in a property with a different name.

Spread Attributes

Spread attributes make it easy to pass all the props to a child component.

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

Let’s consider that the `ProductItem` takes each property of a product instead of the product object.

```
import React from 'react';

function ProductItem({name}) {
 return (
  <div>{name}</div>
 );
}

export default ProductItem;
```

Here is how the spread attributes we can be used in `ProductList` to pass all properties of a product to the `ProductItem` component.

```
import React from 'react';
import ProductItem from './ProductItem';

function ProductList({products}) {
 return (
  <div>
   {products.map(product =>
    <ProductItem
     key={product.id}
     {...product}
```

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

export default ProductList;
```

App Component

The `App` root component takes a list of `products` and sends it to the `ProductList` component.

```
import React from 'react';

import Header from './Header';
import ProductList from './ProductList';

function App({products}) {
  return (
    <div>
      <Header />
      <div>
        <ProductList products={products} />
      </div>
    </div>
  );
}

export default App;
```

Entry Point

The entry point file `index.js` creates an array of `products` and sends it to the `App` component.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

const products = [
 {
  'id' : 1,
  'name' : 'mango',
  'price' : 10
 },
 {
  'id' : 2,
  'name' : 'apple',
  'price': 5
 }];

ReactDOM.render(<App products={products} />,
 document.getElementById('root'));
```

Final Thoughts

List components create the visual interface for a list of values.

It is a good practice to extract out each item from the list in its own component.

Chapter 05: Communication between Components

Splitting the page into small components makes things easier to manage and reuse, but it creates a new challenge, the communication between these components.

Components are organized in a tree-like structure, and so there are two communication directions between them, from parent to child and from child to parent.

Let's take a look at how this can be done.

Data as props

We have already seen how data is sent from parent to child using props. Consider the `ProductItem` functional component:

```
function ProductItem({product}) {
 return (
  <div>
   <div>{product.name}</div>
  </div>
 );
}
```

The `ProductItem` component takes a `product` object to display.

By simply using the parameter destructuring syntax to extract the `product` from props we are enabling the communication from parent to child. Note that `product` is a plain object, and that so far we have used only plain objects as props.

Functions as props

We can handle communication from child to parent using functions as props. Those functions will be used as event handlers.

React events are handled similar to DOM events. The attributes for React events are named using camelCase. `onClick` , `onChange` , `onSubmit` are examples of attributes for event handling. The callback functions for handling events are passed inside curly braces.

Let's take the case of adding a product to the cart.

The `ProductItem` component takes the `onAddClick` function to be called on the Add click.

```
function ProductItem({product, onAddClick}) {}
```

When the Add button is clicked, we need to execute the callback function with the current product as an argument.

```
<button onClick={() => onAddClick(product)} />
```

Here is the full `ProductItem` component:

```
import React from 'react';


function ProductItem({product, onAddClick}) {
 return (
  <div>
   <div>{product.name}</div>
   <div>
    <button
     type="button"
     onClick={() => onAddClick(product)}>
        Add
    </button>
   </div>
  </div>
 );
}

export default ProductItem;
```

It is important to remark that events expect functions as event handlers, so when we need to call the function with additional arguments we need to create a new function. This can be simply solved with an anonymous function that executes the callback with the additional arguments.

`ProductItem` expects the `onAddClick` callback. The callback function is called when the Add button is clicked.

The parent component should pass a callback function to handle the event. Let’s see how.

`ProductList` expects the `onAddClick` callback function. It does nothing more than assigning the `onAddClick` callback to the `onAddClick` event.

```
import React from 'react';
import ProductItem from './ProductItem';

function ProductList({products, onAddClick}) {
 return (
  <div>
   {products.map(product =>
    <ProductItem
     key={product.id}
     product={product}
     onAddClick={onAddClick}
    />
   )}
  </div>
 );
}

export default ProductList;
```

Child components communicate with their parents using callback functions. Parent components set the callback functions using attributes.

We define the callback function in the root component. `App` handles the `onAddClick` event by writing the `product` to console.

```
import React from 'react';

import Header from './Header';
import ProductList from './ProductList';

function App({products}) {
 function addToCart(product) {
  console.log(product);
 }

 return (
  <div>
   <Header />
   <div>
    <ProductList
     products={products}
     onAddClick={addToCart} />
   </div>
  </div>
 );
}

export default App;
```

Using Partial Application

Going back to the fact that events accept functions as event handlers, another option for calling a function with additional arguments is to use partial application.

> Partial application is the process of fixing a number of arguments to a function by producing another function with fewer arguments.

With the `partial()` utility function, from lodash from example, we can create an event handler with the `product` argument already applied.

```
import partial from 'lodash/partial';

<button onClick={partial(onAddClick, product)} />
```

Partial application can be implemented also using the `bind()` method. We should remember that functions are actually objects and as such they have methods.

```
import partial from 'lodash/partial';

<button onClick={onAddClick.bind(null, product)} />
```

In the next chapters, I will continue to use anonymous functions to wrap the call with additional arguments.

Data Flow

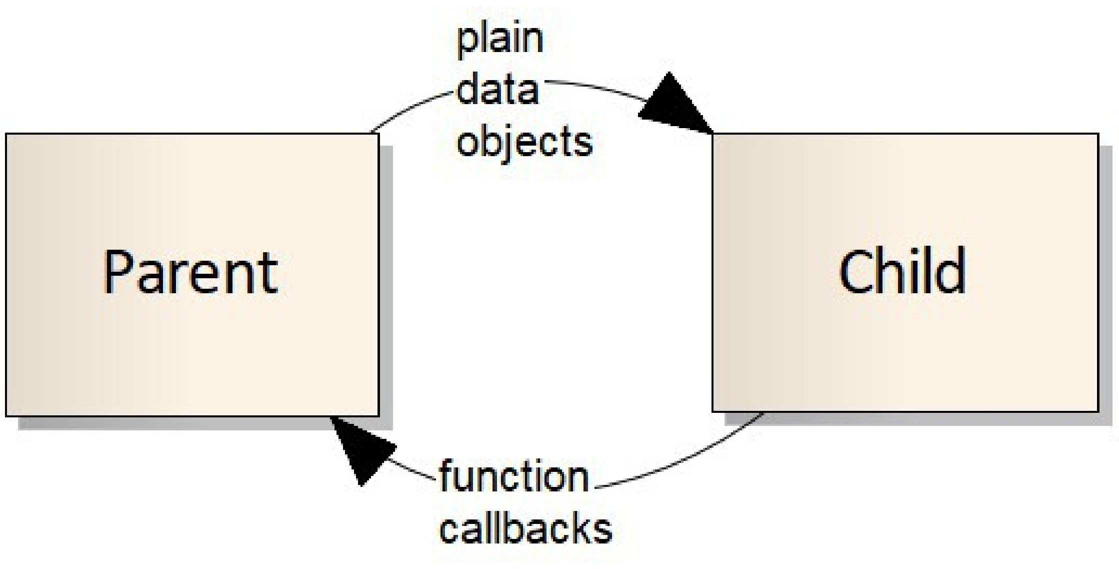
At this point, let’s analyze the data flow between components.

The entry point creates the `products` and passes them to `App` . From there, `App` sends them to `ProductList` . Then `ProductList` passes each `product` to the `ProductItem` components.

Data flow with plain objects as props: Entry point → `App` → `ProductList` → `ProductItem` .

When the Add button is clicked in `ProductItem` , it calls the `onAddClick()` provided by `ProductList` , which calls the `onAddClick()` callback provided by `App` . In a sense, data travels from children to parents using callbacks.

Data flow with functions as props : `ProductItem` → `ProductList` → `App` .



Final Thoughts

The communication between components can be done with props.

Parent-child communication is handled using plain objects as props.

Child-parent communication is handled using functions as props. On user interactions, child components invoke the function callbacks provided by the parent components with new data.

Chapter 06: Presentation Components

Presentation components transform data into a visual representation.

Presentation components communicate only through their own props.

All the components we have built so far `ProductItem` , `ProductList` , `ShoppingList` , `ShoppingItem` , `App` are presentation components.

Let’s build and analyze the `ShoppingItem` and `ShoppingCart` presentation components.

ShoppingItem Component

`ShoppingItem` gets a `product` and creates the JSX markup for one item in the shopping cart.

It takes a callback to handle the Remove `onClick` event.

```
import React from 'react';


function ShoppingItem({product, onRemoveClick}) {
 return (
  <div>
    <div>{product.name}</div>
    <div>{product.quantity}</div>
    <div>
     <button
        type="button"
        onClick={() => onRemoveClick(product)}>
         Remove
     </button>
    </div>
  </div>
 );
}

export default ShoppingItem;
```

ShoppingCart Component

`ShoppingCart` gets a `cart` object and creates the JSX markup for it.

`ShoppingCart` takes the `onRemoveClick` callback and assigns it to the `onRemoveClick` event.

```
import React from 'react';
import ShoppingItem from './ShoppingItem';

function ShoppingCart({cart, onRemoveClick}) {
 return (
  <div>
   <div>
    {cart.list.map(product =>
     <ShoppingItem
       key={product.id}
       product={product}
       onRemoveClick={onRemoveClick}
     />
    )}
   </div>
   <div>Total: {cart.total}</div>
  </div>
 );
}
```

```
export default ShoppingCart;
```

Presentation Components as Pure Functions

> A pure function is a function that given the same input always returns the same output and has no side-effects.

The `ProductItem` , `ProductList` , `ShoppingCart` , `ShoppingItem` , are not only presentation components but also pure functions.

`ShoppingItem` is a pure function no matter if the `onRemoveClick` callback is pure or impure. Note that `ShoppingItem` doesn’t execute the `onRemoveClick` . It creates a new function with the `product` already applied as the first argument and assigns it to the event.

Writing components as pure functions makes the code easier to reason about.

Props

Presentation components communicate only throw their own props. They take two kinds of inputs, plain data objects, and function callbacks.

The plain objects are the data to be transformed into a user interface.

The callback functions are used as event handlers.

Final Thoughts

Presentation components transform data into a visual interface.

Presentation components are best implemented as pure functions.

Chapter 07: Styling Components

Before going further, let’s apply a minimal styling to existing components.

A few CSS concepts

Visual HTML elements can be split into two, inline and block elements.

Block vs inline

Block-level elements include tags like `div` , `header` , `footer` . They act as containers for other inline or block elements. A block element starts on a new line and extends to the width of its parent. It can be sized using `width` and `height` .

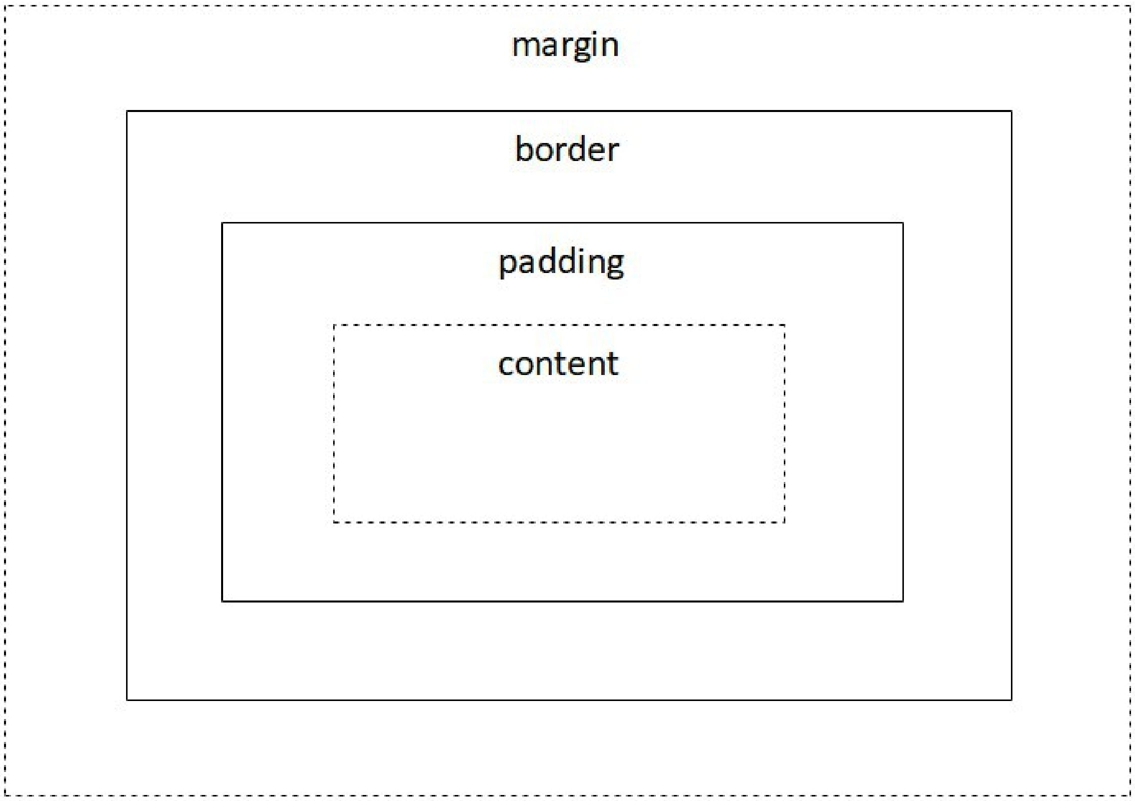
Inline elements include tags like `span` , `input` , `select` . They have the `width` and `height` of their content. Inline elements are displayed on the same line unless there is not enough space on that line. Inline elements can’t be sized using `width` and `height` .

Normal Document Flow

The normal document flow is the default way for displaying elements on a page. Block elements are put on new rows from top to bottom and inline elements are displayed one after the other, from left to right.

The Box Model

At the core of the page layout is the box model. Any visible HTML element is considered to be in a rectangular box defined by content (`width` , `height`), `padding` , `border` , and `margin` .



Selectors

Selectors select elements on the page. The most common are:

- The id selector `#id` selects an element with a specific id attribute. The id of an element should be unique on the page.
- The class selector `.className` selects all elements with a specific `class` attribute.
- The tag selector `tagName` selects all elements based on their tag name.

CSS Syntax

A CSS rule consists of a selector and a set of CSS properties. Each property has a name and a value.

The next rule selects the `header` tag and sets the `text-align` property to `center` .

```
header {
 text-align: center;
}
```

Flex Layout

`display: flex` defines the flex container. By default, all its direct children are displayed from left to right on a single line.

The `flex-direction` property defines the direction in which flex items are placed in the flex container. The `row` value displays the flex items horizontally, from left to right and the `column` value displays the flex items vertically, from top to bottom.

Styling Components

We will define a CSS file specific for each component. The CSS file will have the same name as the JSX file.

ProductItem CSS

The `.product-list-item` class defines the style for each product item.

```
.product-list-item {
 padding: 10px;
 margin: 10px;
 background-color: #CC6500;
 color: #FFF;
}
```

Create React App uses Webpack for handling all assets, including CSS files. Webpack extends the concept of `import` beyond JavaScript. To use a CSS file inside a component, we need to import it.

Here is how we can use the `ProductItem.css` file in `ProductItem.jsx` :

```
import React from 'react';
import './ProductItem.css';

function ProductItem({product, onAddClick}) {
 return (
  <div className="product-list-item">
   <div>{product.name}</div>
   <div>
    <button
     type="button"
     onClick={() => onAddClick(product)}>
       Add
    </button>
   </div>
  </div>
 );
}

export default ProductItem;
```

ShoppingItem CSS

The `.shopping-cart-item` class defines the style for each shopping item.

```
.shopping-cart-item {
 padding: 10px;
 margin: 10px;
 color: #FFF;
 background-color: #FF9C00;
 display: flex;
 flex-direction: column;
}
```

```
.shopping-cart-item div {
 margin: 5px;
}
```

ShoppingCart CSS

The `.shopping-cart-total` class defines the style for the total price.

```
.shopping-cart-total {
 padding: 10px;
 margin: 10px;
 color: #654321;
 border: 1px solid #FF9C00;
}
```

App CSS

The `App.css` defines the page layout and the default look for elements common to all components, like buttons.

```
button {
 background: #FAF1DD;
 border: 1px solid #FAF1DD;
 color: #654321;
 padding: 5px;
 cursor: pointer;
}

header {
 text-align: center;
}

//layout
.content {
 display: flex;
 width: 70%;
 margin: 0 auto;
}

.content > div{
 flex : 1;
}
```

Final Thoughts

CSS defines the look and feel of an HTML page.

In normal document flow block elements are displayed on new lines from top to bottom and inline elements are displayed on the same line from left to right.

Component styles can be defined in separate CSS files. Webpack allows importing CSS files inside components.

Chapter 08: Stateful Functional Components

So far, we have used pure functional components taking data as props and transforming that into a visual interface.

Next, we are going to explore how to define functional components that have state. That's it, they store data.

State

State is data that is stored and can be changed.

React Hooks enables to create functional components that store state. React Hooks is a collection of hook functions. They are easy to recognize as they must start with the word `"use"` .

Hook function calls should be done at the top of the functional component.

The first hook we are going to look at is the `useState()` function, that can be used to add local state to a functional component.

```
import React, { useState } from 'react';

function App({products}) {
 const [shoppingMap, setShoppingMap] = useState({});
 //...
}
```

`useState()` returns a variable holding the current state and a function to update it. `useState()` takes an argument as the initial state value. Changing the state triggers a re-render.

The `App` component stores a map with all products added to the shopping cart and their quantity.

A map is a dynamic collection of key-value pairs. The key, in this case, is the product id and the value is the product object.

`shoppingMap` gives the current map. `setShoppingMap()` updates the map and re-renders the component.

Here is the implementation of the `App` component:

```
import React, {useState} from 'react';

import Header from './Header';
import ProductList from './ProductList';
import ShoppingCart from './ShoppingCart';

import './App.css';

function App({products}) {
 const [shoppingMap, setShoppingMap] = useState({});

 function addToCart(product) {
  setShoppingMap(map => addProductToMap(product, map));
 }

 function removeFromCart(product) {
  setShoppingMap(map => removeProductFromMap(product, map));
 }

 return (
  <div>
   <Header />
   <div className="content">
    <ProductList
     products={products}
     onAddClick={addToCart} />
    <ShoppingCart
     cart={toCartView(shoppingMap)}
     onRemoveClick={removeFromCart} />
   </div>
  </div>
 );
}
```

`addToCart()` handles the Add click by adding the new product to the map.

`removeFromCart()` handles the Remove click by removing the product from the map.

`setShoppingMap()` can take a mapping function as an argument. The mapping function takes the previous state value and returns the new state value.

Pure Functions

We are going to do all state changes using pure functions.

`addProductToMap()` takes a product and a map and returns the new map with the product quantity updated.

```
function addProductToMap(product, map){
 const newMap = { ...map };
 const quantity = getProductQuantity(product, map) + 1;
 newMap[product.id] = { ...product, quantity };
 return Object.freeze(newMap);
}
```

The new product added to the map has the quantity updated. If the product does not exist it will be added with quantity one, otherwise, its current quantity will be incremented by one.

Pure functions don’t modify their input values. For this reason, the `map` is first cloned. The spread operator is used to create a shallow copy of the input map, `{ ...map }`.

`deleteProductFromMap()` takes a product and a map and returns the new map with the product removed.

```
function removeProductFromMap(product, map){
 const newMap = { ...map };
 delete newMap[product.id];
 return Object.freeze(newMap);
}
```

`getProductQuantity()` takes a product and a map and returns the current quantity of that product.

```
function getProductQuantity(product, map) {
 const existingProduct = map[product.id];
 return (existingProduct) ? existingProduct.quantity : 0;
}
```

`toCartView()` converts a map to a list of products plus the total price. The total price is computed using the `addPrice()` reducer function.

```
function toCartView(map) {
 const list = Object.values(map);
 return Object.freeze({
  list,
  total: list.reduce(addPrice, 0)
 });
}

function addPrice(totalPrice, line) {
 return totalPrice + line.price * line.quantity;
}
```

The pure functions can be extracted out from the component and exported to be tested.

At this point, all the logic managing the state is inside the `App` root component. All the other components are presentation components taking data as props.

Final Thoughts

Data can be available in a functional component either from props or from its internal state.

The `useState()` hook defines state inside a functional component.

State transformations can be done with pure functions taking the previous state as a parameter and returning the new state.

Chapter 09: Custom Store

Next, we are going to explore what it means to extract out the state and the state management logic in a separate object. We will create a new store object responsible for managing the shopping cart data.

First, we need to introduce the concept of the store.

Store

A store is an object whose main purpose is to store and manage data.

The store emits events every time its state changes. Components can subscribe to these events and update the user interface.

Event Emitter

The store object emits events. For this task, we will use a micro event emitter.

```
npm install micro-emitter --save
```

The emitter object is used to emit and register event handlers.

```
const eventEmitter = new MicroEmitter();
const CHANGE_EVENT = 'change';

//emit event
eventEmitter.emit(CHANGE_EVENT);

//register event handlers
eventEmitter.on(CHANGE_EVENT, handler);
```

The `ShoppingCartStore` will store and manage the current shopping cart. It is the single source of truth regarding the shopping cart.

Consider the following implementation of the store:

```
import MicroEmitter from 'micro-emitter';

//pure functions
//...

function ShoppingCartStore() {
 const eventEmitter = new MicroEmitter();
 const CHANGE_EVENT = "change";

 let shoppingMap = {};

 function addToCart(product) {
  shoppingMap = addProductToMap(product, shoppingMap);
  eventEmitter.emit(CHANGE_EVENT);
 }

 function removeFromCart(product) {
  shoppingMap = removeProductFromMap(product, shoppingMap);
  eventEmitter.emit(CHANGE_EVENT);
 }

 function onChange(handler) {
  eventEmitter.on(CHANGE_EVENT, handler);
 }

 function offChange() {
  eventEmitter.off(CHANGE_EVENT);
 }

 function get() {
  return toCartView(shoppingMap);
 }

 return Object.freeze({
  addToCart,
  removeFromCart,
  get,
  onChange,
  offChange
 });
}

export default ShoppingCartStore;
```

The `ShoppingCartStore` is a factory function. It builds encapsulated objects. The `shoppingMap` is hidden. Factory functions don't use the confusing `this` pseudo-parameter.

Entry Point

The store needs to be created and sent to components.

`index.js` is the application entry point. This is the place where the store is created. It is then sent to the `App` component using props.

```
import ShoppingCartStore from './ShoppingCartStore';

const shoppingCartStore = ShoppingCartStore();

ReactDOM.render(<App products={products}
 shoppingCartStore={shoppingCartStore} />,
 document.getElementById('root'));
```

Using the Store

The `App` root component communicates with the store.

React is a data-driven UI library. We need to update the data in order to modify the user interface. We can use the `useState()` hook function to define the `cart` object that is updated when the store changes. `setCart()` updates the `cart` .

When the store emits a change event, the component changes the local state and the UI is re-rendered.

```
import React, {useState, useEffect} from 'react';

import Header from './Header';
import ProductList from './ProductList';
import ShoppingCart from './ShoppingCart';

import './App.css';

function App({products, shoppingCartStore}) {
 const [cart, setCart] = useState({list: []});

 useEffect(subscribeToStore, []);

 function subscribeToStore() {
  shoppingCartStore.onChange(reload);

  return function cleanup(){
   shoppingCartStore.offChange();
  };
 }

 function reload() {
  const cart = shoppingCartStore.get();
  setCart(cart);
 }
```

```
  return (
   <div>
    <Header />
    <div className="content">
     <ProductList
      products={products}
      onAddClick={shoppingCartStore.addToCart} />

     <ShoppingCart
      cart={cart}
      onRemoveClick={shoppingCartStore.removeFromCart} />
    </div>
   </div>
  );
}

export default App;
```

Effect Hook

The effect hook performs side-effects in function components. Any communication with the outside environment is a side-effect.

The “effect” is a function that runs after React performs the DOM updates. The effect function has access to the state variables.

The effect function runs after every render.

`useEffect()` can skip running an effect if specific values haven’t changed between re-renders. For this, we need to pass an array as an optional second argument to the function. React compares the values in the array from the previous render with the values in the array from the next render and when all items in the array are the same React skips running the effect.

When we want to run the effect only once, we pass an empty array (`[]`) as a second argument. In this case, the props and state inside the effect function have their initial values.

The following code runs the effect only once. It subscribes to store events only once.

```
useEffect(loadAndSubscribe, []);

function loadAndSubscribe() {
 shoppingCartStore.onChange(reload);
}
```

Effects with Cleanup

The effect is a function. If this function returns another function, the returned function is then run to clean up.

The cleanup function is the return function from an effect.

When effects run more than once, React cleans up the effects from the previous render before running the next effects.

```
useEffect(subscribeToStore, []);

function subscribeToStore() {
 shoppingCartStore.onChange(reload);
 return function cleanup(){
  shoppingCartStore.offChange();
 };
}
```

Final Thoughts

Stores offer a way to encapsulate state and share the state management behavior between components.

The custom store object emits events when its state changes.

The effect hook allows side-effects inside components.

Chapter 10: MobX Store

MobX allows creating stores with observable state.

Let’s first install the MobX library.

```
npm install mobx --save
npm install mobx-react --save
```

Store

With MobX we can create the state in `ShoppingCartStore` as an observable map.

```
import { observable, action } from 'mobx';

//pure functions
//...

export default function ShoppingCartStore(){
 const shoppingMap = observable.map();

 function toCartView() {
  return toCartViewFromMap(shoppingMap);
 }

 //actions
 const addToCart = action(function(product){
  shoppingMap.set(product.id,
   incrementProductQuantity(shoppingMap, product));
 });

 const removeFromCart = action(function(product){
  shoppingMap.delete(product.id);
  });

  return Object.freeze({
    addToCart,
    removeFromCart,
    toCartView
  });
}
```

`observable.map()` defines an observable map.

`observable()` makes objects and arrays observable.

Once the state is observable, we can turn components into observers and react to changes by re-rendering.

In MobX actions are functions that modify the state. Action functions should be marked using the `action()` decorator. This will make sure that intermediate changes are not visible until the action has finished.

The `ShoppingCartStore` defines two actions for adding and removing products from the cart and exposes them as public methods. It also makes public the method for getting the current cart.

Container Components

Presentation components don't communicate with the outside environment using a publish-subscribe pattern. The `ShoppingCartStore` cannot be used in presentation components.

We want to keep presentation components as pure functions and get all the benefits of purity. We can use container components to connect stores to presentation components.

Container components provide data to presentation components and define handlers for events in presentation components. Container components may be impure, they may call methods on stores for example.

ProductList Container

The `ProductList` presentation component is a pure function. We will create a container that makes the connection between the presentation component and the store.

`inject()` can be used to get access to the MobX store.

> `inject()` creates a higher-order component that makes the stores available to the wrapped component. `inject()` can take a mapper function. The mapper function receives all stores as an argument and creates the object mapping properties to data and callbacks.
>
> A higher-order component is a function that takes a component and returns a new component.

Here is the container component that handles the `onAddClick` event.

```
import { inject } from 'mobx-react';

import ProductList from '../ProductList';

const withPropsMapped = inject(function(stores){
 return {
  onAddClick : stores.shoppingStore.addToCart
 };
});

export default withPropsMapped(ProductList);
```

ShoppingCart Container

Once the store is observable we can make the `ShoppingCart` component an observer. It means that when the state changes inside the store, the `ShoppingCart` component will re-render.

`observer()` takes a component an creates a new one that will re-render when state changes. Components become reactive.

```
import { inject, observer } from 'mobx-react';

import ShoppingCart from '../ShoppingCart';

const withPropsMapped = inject(function(stores){
 return {
  cart : stores.shoppingStore.toCartView(),
  onRemoveClick: stores.shoppingStore.removeFromCart
 };
});

export default withPropsMapped(observer(ShoppingCart));
```

Root Component

The `App` root component becomes very simple.

```
import React from 'react';

import Header from './Header';
import ProductList from './containers/ProductListContainer';
import ShoppingCart from './containers/ShoppingCartContainer';

import './App.css';

function App({products}) {
 return (
  <div>
   <Header />
   <div className="content">
```

```
    <ProductList products={products} />
    <ShoppingCart />
   </div>
  </div>
 );
}

export default App;
```

Entry Point

The `index.js` is the application single entry point. Here is where the store is created.

We can send the store down to components using `props` , but a simpler way is to use the [React Context](#) . React Context can be used to share data that is considered to be “global”.

The `Provider` component from the `mobx-react` package can be used to send the stores down to components. `Provider` uses the React Context. The root component `App` should be wrapped inside the `Provider` component.

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'mobx-react';

import ShoppingCartStore from './stores/ShoppingCartStore';
import App from './App';

const products = [
   //...
 ];

const shoppingStore = ShoppingCartStore();
```

```
ReactDOM.render(<Provider shoppingStore={shoppingStore}>
 <App products={products} />
 </Provider>,
 document.getElementById('root'));
```

Final Thoughts

MobX allows to defined observable state.

`observer()` turns components into reactive components. The observer components re-render when the state changes.

Container components can connect presentation components to stores.

Chapter 11: Redux Store

Next, we are going to do state management using the Redux library.

The Redux store does state management following functional principles. To start using the Redux store, we need to install the following packages:

```
npm install redux --save

npm install react-redux --save
```

Store

In Redux there is a single store that manages all the application state-tree.

The store has just a few methods:

- the `getState()` method for reading all the state as a read-only immutable value
- the `dispatch()` method for dispatching actions
- the `subscribe()` methods for registering to state changes

The state inside the store can be changed only by dispatching actions. There are no state setters on the store object.

Actions

Actions are plain data objects containing all the necessary information to make an action.

Let’s imagine what are the necessary information for adding a product to cart. First, the application needs to know that it is an `'ADD_TO_CART'` action, second, it needs to know the product to be added to the cart. Here is how the `'ADD_TO_CART'` action may look like:

```
{
 type: 'ADD_TO_CART',
 product
}
```

The action object needs the `type` property indicating the action to perform. `type` is usually a string.

Action objects should be treated as immutable.

Action Creators

A common practice is to encapsulate the code that creates the plain action objects in functions. These functions are pure functions called action creators.

```
function addToCart(product) {
 return {
  type: 'ADD_TO_CART',
  product
 };
}

function removeFromCart(product) {
 return {
  type: 'REMOVE_FROM_CART',
  product
 };
}

export default { addToCart, removeFromCart };
```

Reducers

The Redux store manages state using pure functions called reducers. These functions take the previous state and an action as parameters and return the new state.

The store applies reducers when an action is dispatched.

The Redux store does state changes, but these changes are encapsulated behind the library. We write only the pure functions and let the library apply them and do the state modifications.

Let’s write the reducer for managing the shopping cart.

```
function addToCart(map, product){
 const newMap = { ...map };
 const quantity = getProductQuantity(map, product) + 1;
 newMap[product.id] = { ...product, quantity };
 return Object.freeze(newMap);
}

function removeFromCart(map, product){
 const newMap = { ...map };
 delete newMap[product.id];
 return Object.freeze(newMap);
}

function getProductQuantity(map, product) {
 const existingProduct = map[product.id];
 return (existingProduct) ? existingProduct.quantity : 0;
}

export default function (shoppingMap = {}, action) {
 switch (action.type) {
  case 'ADD_TO_CART':
   return addToCart(shoppingMap, action.product);
  case 'REMOVE_FROM_CART':
   return removeFromCart(shoppingMap, action.product);
  default:
   return shoppingMap;
  }
}
```

The state is initialized with an empty object that will be used as a map.

When the `'ADD_TO_CART'` action is dispatched, the state changes to a new map with the quantity for the specified product updated. The product is found in `action.product` .

When the `'REMOVE_FROM_CART'` action is dispatched, the state changes to a new map with the product removed. The product to delete is taken from `action.product` .

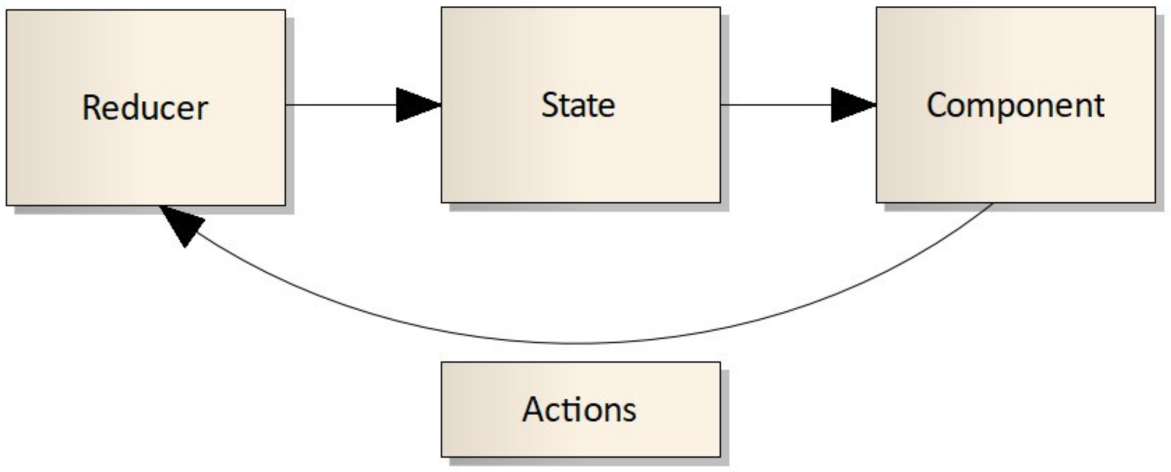
Reducers are used to change the state, not to get the state from the store.

Root Reducer

Redux requires one root reducer. We can create many reducers managing parts of the root state and then combine them together with the `combineReducers()` utility function and create the root reducer.

```
import { combineReducers } from 'redux';
import shoppingCart from './shoppingCart';

export default combineReducers({
 shoppingCart
});
```

Here is how the data flow with the Redux store looks like:



Selectors

Selectors are functions that take the state as input and return a part of that state or compute derived values from it. Selectors are pure functions.

The `toCartView()` selector extracts the `shoppingCart` map from the state and returns a list of all products and the total price.

```
function toCartView({shoppingCart}) {
 const list = Object.values(shoppingCart);
 return Object.freeze({
  list,
  total: list.reduce(addPrice, 0)
 });
}

function addPrice(totalPrice, line) {
 return totalPrice + line.price * line.quantity;
}

export { toCartView };
```

Entry Point

We create the store in the application entry point, the `index.js` file.

We can use the `Provider` component from the `react-redux` to send the store down to components. To send the store to all components we wrap the root component inside the `Provider` . The store is then available to components using the `connect()` function.

```
import React from 'react';
import ReactDOM from 'react-dom';
import { createStore } from 'redux';
import { Provider } from 'react-redux';

import rootReducer from './reducers';
import App from './App';

const products = [
   //...
 ];

const store = createStore(rootReducer);

ReactDOM.render(<Provider store={store}>
 <App products={products} /></Provider>,
 document.getElementById('root'));
```

Presentation Components

We will use the same presentation components.

Presentation components communicate only through their own props. They don’t have access to the store object.

Container Components

The `ProductList` is a presentation component that requires the list of `products` in order to render the user interface.

We need to take the data from the store, process it, and send it to the `ProductList` component. This is the role of the container component.

For this, we will use the `connect()` utility from the `react-redux` package.

> `connect()` connects a component to a Redux store.

The container component does the following:

- It takes the state from the store, processes it with a selector, and then sends the result to the presentation component as props. The `mapStateToProps()` does that.
- It defines the actions to be dispatched when events are triggered by user interaction. The `mapDispatchToProps()` does that.

ProductList Container

The `ProductListContainer` component dispatches the `'ADD_TO_CART'` action on the `onAddClick` event.

```
import { connect } from 'react-redux';
import { addToCart } from '../actions/ShoppingCartActions';
import ProductList from '../ProductList';

function mapDispatchToProps(dispatch) {
 return {
  onAddClick: function(product){
   dispatch(addToCart(product));
  }
 };
}

export default connect(
 null,
 mapDispatchToProps
)(ProductList);
```

ShoppingCart Container

The `ShoppingCartContainer` connects the `ShoppingCart` component to the store. It reads state from the store, transforms it using the `toCartView()` selector, and sends it to the presentation component. At the `onRemoveClick` event, it dispatches the `'REMOVE_FROM_CART'` action.

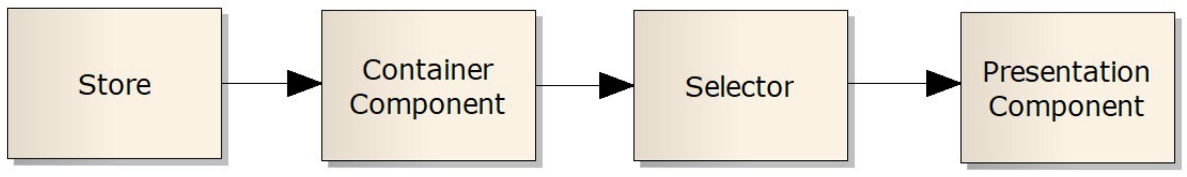
```
import { connect } from 'react-redux';

import { removeFromCart } from
'../actions/ShoppingCartActions';
import { toCartView } from
'../selectors/ShoppingCartSelectors';

import ShoppingCart from '../ShoppingCart';

function mapStateToProps(state) {
 return {
  cart: toCartView(state)
 };
}

function mapDispatchToProps(dispatch) {
 return {
  onRemoveClick: function(product){
   dispatch(removeFromCart(product));
  }
 };
}
```

```
export default connect(
 mapStateToProps,
 mapDispatchToProps
)(ShoppingCart);
```

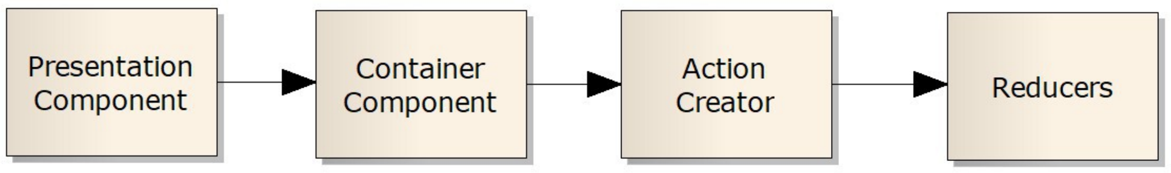
Data Flow

Let’s look at the data flow for reads and writes.

Reads from the store



Writes to the store



Final Thoughts

The Redux store manages state using pure functions.

A reducer is a pure function that takes the state and an action and returns the new state.

An action creator is a pure function that returns a plain action object.

A selector is a function that takes the state as input and extracts or computes new data from it.

Chapter 12: Redux Actions and Immutable.Js

In this chapter, we are going refactor the action creation and handling using helpers and then introduce immutable data structures.

Action Creators

Action creators tend to become repetitive in nature and can be easily created with a helper library like the one from Redux Actions .

```
import { createAction } from 'redux-actions';


const addToCart = createAction('ADD_TO_CART');

const removeFromCart = createAction('REMOVE_FROM_CART');


export default { addToCart, removeFromCart };
```

> The `createAction(type)` utility returns an action creator that creates an action with that type.

To install Redux Actions run the following command:

```
npm install redux-actions --save
```

You can imagine the `addToCart()` action creator defined with the helper as similar to the function below:

```
function addToCart(payload){
 return {
  type: 'ADD_TO_CART',
  payload
 }
}
```

As you see, all the additional data the action creator receives stays in the `payload` property of the action object.

Handling Actions in Reducer

Let’s look again at the reducer function handling several actions.

```
export default function (shoppingMap = {}, action) {
 switch (action.type) {
  case 'ADD_TO_CART':
   return addToCart(shoppingMap, action.product);
  case 'REMOVE_FROM_CART':
   return removeFromCart(shoppingMap, action.product);
  default:
   return shoppingMap;
 }
}
```

Functions should be small and do one thing. The `switch` statement leads to large code doing multiple things. I think that using a map is a better option for handling actions in a reducer.

The `handleActions()` utility function from the Redux Actions can be used to create a new function that handles several actions by calling a specific updater function. It does all this using a map. Below is how it looks like:

```
import { handleActions } from 'redux-actions';
import actions from '../actions/ShoppingCartActions';

function addToCart(map, action){}

function removeFromCart(map, action){}

export default handleActions({
  [actions.addToCart]: addToCart,
  [actions.removeFromCart]: removeFromCart
 },
 {}
);
```

Here is how we can understand the object literal used as a map:

- when the action `addToCart` comes in, the `addToCart()` updater function will be executed.
- when the action `removeFromCart` comes in, the `addToCart()` updater function will be executed.

We need to modify the updater functions handling the modification for an action to take the product from the `payload` property.

```
function addToCart(map, action){
 const product = action.payload;
 const newMap = { ...map };
 const quantity = getProductQuantity(map, product) + 1;
 newMap[product.id] = { ...product, quantity };
 return Object.freeze(newMap);
}

function removeFromCart(map, action){
 const product = action.payload;
 const newMap = { ...map };
 delete newMap[product.id];
 return Object.freeze(newMap);
}
```

Immutable Data Structures

An operation on immutable data structures results in a new updated data structure.

Immutable.js provides immutable data structures like `Map` and `List` . Let’s first install the library.

```
npm install immutable --save
```

Till now, we have used the object literal as a map. In order to create new value when changing the map, we used the spread syntax to make a copy of the map.

Next, we are going to use the Map data structures from Immutable.js .

> `Map()` creates an immutable map. `Map` is a factory function, it does not use the `new` keyword.
>
> `set(key, value)` returns a new Map containing the new key-value pair.
>
> `remove(key)` returns a new Map without the specified key.

Let’s refactor the previously updated functions.

```
function addToCart(map, action){
 const product = action.payload;
 const quantity = getProductQuantity(map, product) + 1;
 const newProduct = { ...product, quantity };
 return map.set(product.id, newProduct);
}

function removeFromCart(map, action){
 const product = action.payload;
 return map.remove(product.id);
}

export default handleActions({
  [actions.addToCart]: addToCart,
  [actions.removeFromCart]: removeFromCart
 },
 Map()
);
```

At this point, we need also to change the `toCartView()` selector extracting the list of products from the map. It now extracts the products as a sequence
.

```
function toCartView({shoppingCart}) {
 const list = shoppingCart.valueSeq();
 return Object.freeze({
  list,
  total: list.reduce(addPrice, 0)
 });
}
```

> A sequence returns the next value from the collection each time it is called. It efficiently chains methods like `map()` and `filter()` by not creating intermediate collections

Final Thoughts

Action creating can be simplified with helpers.

The mapping between actions and small updater functions can be also simplified with a helper.

Immutable data structures cannot be changed. Any of its methods, when called to do a modification will return a new data structure.

Chapter 13: Fetching with MobX Store

A common scenario is to get data from a REST API and display it on the screen. Let’s do just that.

Fake REST API

First, we need to create a REST API. We can simply create a fake API using JSON Server .

Start by installing the JSON Server.

```
npm install -g json-server
```

Then create a simple JSON file with a list of products in a `.json` file. Here is the `products.json` file:

```
{
  "fruits":[
    {
      "id":1,
      "name":"mango",
      "price":10
    },
    {
      "id":2,
      "name":"apple",
      "price":5
    }
  ]
}
```

In the end, start JSON Server.

```
json-server --watch products.json
```

Now, at `http://localhost:3000/fruits` we can get all the products.

API Utils

It is better to separate responsibilities and encapsulate the network calls in their own files. Functions doing network calls are impure.

`fetchProducts()` gets all products from the REST API.

```
const baseUrl = 'http://localhost:3000';

function toJson(response){
 return response.json();
}

function fetchProducts(){
 return fetch(`${baseUrl}/fruits/`)
  .then(toJson)
}

export default { fetchProducts };
```

The `fetch()` built-in function can be used to fetch resources across the network. `fetch()` is an asynchronous function that returns a promise containing the response.

> A promise is an object that represents a possible future result of an asynchronous operation.

The `json()` method extracts the JSON object from the response.

Product Store

MobX offers an object-oriented way of working with the state.

The store is responsible for managing state but also to keep it in sync with the server.

The store keeps an observable array of products. It delegates the network calls to the `api` object and then updates its internal state.

```
import { observable, action } from 'mobx';

function ProductStore(api){
 const products = observable([]);

 const fetchProducts = action(function(){
  return api.fetchProducts()
    .then(resetProducts);
 });

 function resetProducts(newProducts){
  products.replace(newProducts);
 }

 function getProducts(){
  return products.toJS();
 }

 return Object.freeze({
  getProducts,
  fetchProducts
 });
}

export default ProductStore;
```

Notice that we separate use from construction, and take the `api` object dependency as a parameter in `ProductStore` .

ProductsList Container

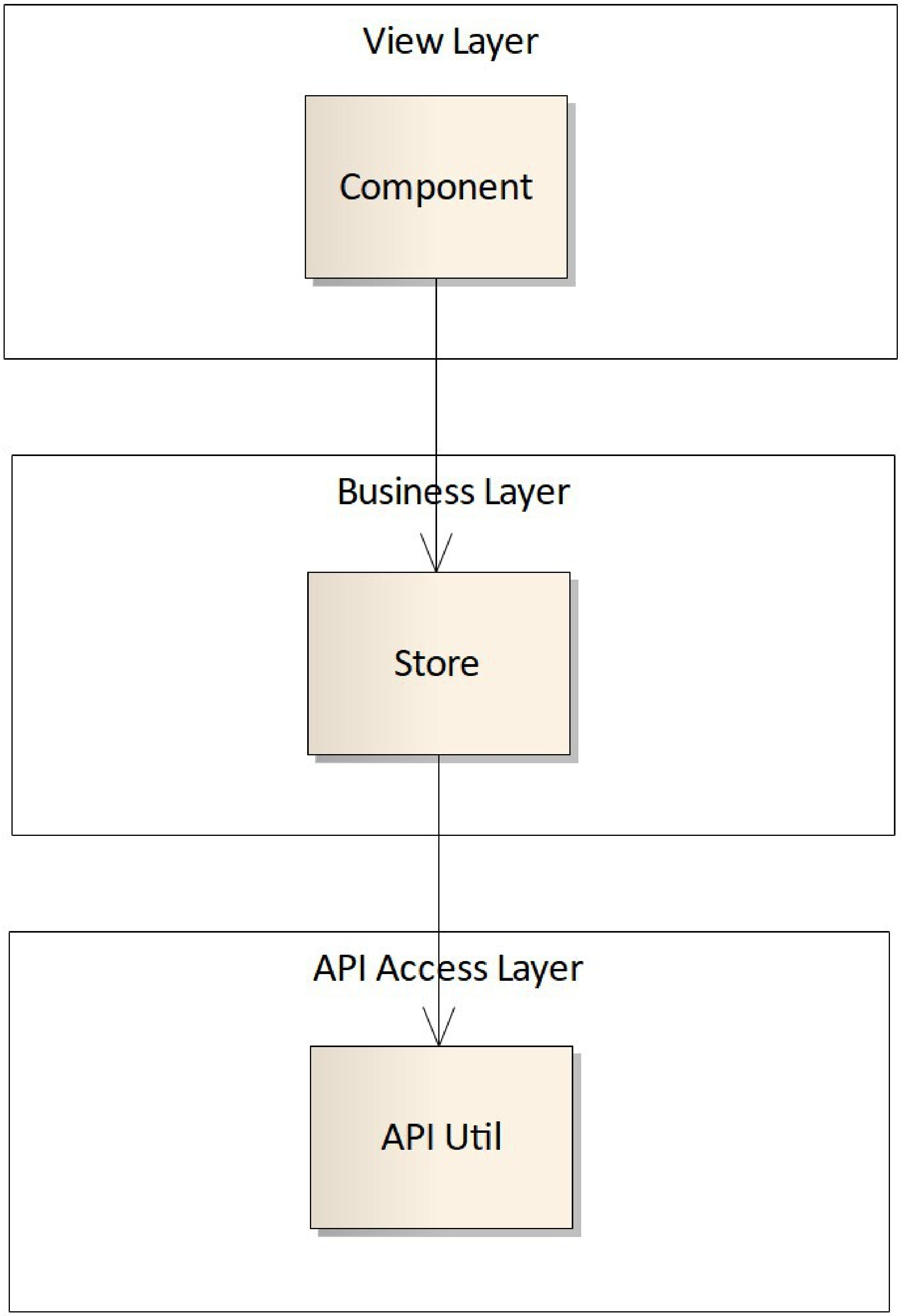
The container component turns the `ProductList` component into an observer. When products change in the store, the `ProductList` component re-renders.

```
import { inject, observer } from 'mobx-react';

import ProductList from '../ProductList';

const withPropsMapped = inject(function(stores){
 return {
  products : stores.productStore.getProducts(),
  onAddClick : stores.shoppingStore.addToCart
 };
});

export default withPropsMapped(observer(ProductList));
```

Layers

This approach leads to a layered architecture:

- The view layer is responsible for displaying data on the page, and for handling user interactions. The view layer is made up of components.
- The business layer is made of stores.
- The API access layer contains the objects communicating with the REST API.


View Layer
Component
Business Layer
Store
API Access Layer
API Util

The layered architecture offers a better separation of concerns.

Entry Point

The entry point `index.js` creates the two stores passing the `api` object as a dependency, and then sends all stores to components using the `Provider` .

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'mobx-react';

import api from './api/productsAPI';
import ProductStore from './stores/ProductStore';
import ShoppingCartStore from './stores/ShoppingCartStore';
import App from './App';

const shoppingStore = ShoppingCartStore();
const productStore = ProductStore(api);

const stores = {
 productStore,
 shoppingStore
};

ReactDOM.render(<Provider {...stores}><App /></Provider>,
 document.getElementById('root'));

productStore.fetchProducts();
```

Final Thoughts

It is a good practice to encapsulate network calls in their own files.

MobX takes an object-oriented approach to work with the state. Stores are responsible for managing the state and keep it in sync with the server.

To make things flexible and easier test we separate use from construction. For this reason, we can create all objects in the entry point file.

Chapter 14: Fetching with Redux Thunk

So far, all the functions written for the Redux store were pure functions. There are situations when we need to write impure core like when doing network calls.

In order to work with impure code in Redux, we need to introduce a new concept, the middleware .

A middleware enables us to write async logic that interacts with the store.

According to the documentation, the standard way to do it with Redux is to use the Redux Thunk middleware .

Redux Thunk

Thunks are the recommended middleware for side-effects logic like network requests.

The Redux Thunk middleware allows you to write action creators that return a function instead of a plain action object.

Let’s install the `redux-thunk` library.

```
npm install redux-thunk --save
```

Synchronous Action Creators

The action creators we defined until now built plain action objects. These plain action objects were dispatched and the state was updated immediately.

Now, we intend to make a network request and get all products from the REST API and display them on the page.

Redux architecture is a data-driven one, we need to update the state in order to update the UI.

In order to update the state, we need an action for resetting all products with the new ones. Here is the action creator for building it:

```
import { createAction } from 'redux-actions';

const resetProducts = createAction('resetProducts');

export default { resetProducts };
```

Asynchronous Action Creators

Let’s get back to the API call. We want to use the `fetchProducts()` function to take all products from the REST API and display them on the page.

Here is how we are going to do it. First, we make the network call to take the products and then we update the store by dispatching an action with the new products. We can do this orchestration logic using an asynchronous action creator.

These asynchronous action creators built with Redux Thunk are functions that return other functions. The returned functions are called “thunks”.

The following `fetchProducts()` asynchronous action creator takes the products from the REST API and then dispatches a plain action to change the state.

```
import api from '../api/productsAPI';
import actions from '../actions/productsActions';

function fetchProducts() {
 return function(dispatch) {
  return api.fetchProducts()
   .then(actions.resetProducts)
   .then(dispatch);
 };
}
```

```
export default { fetchProducts };
```

The inner function receives the store methods `dispatch` and `getState` as parameters.

Reducers

I order to update the state we need a new reducer that updates the previous products with the new products.

```
import { handleActions } from 'redux-actions';
import { List } from 'immutable';
import actions from '../actions/productsActions';

function resetProducts(products, action) {
 return List(action.payload);
}

export default handleActions({
  [actions.resetProducts]: resetProducts
 },
 List()
);
```

The list of products is stored as an immutable list using the List data structure.

Root Reducer

The root reducer should split the state management between the `shoppingCart` and `products` reducers, each with its own state.

`combineReducers()` needs to be updated with the new `products` reducer.

```
import { combineReducers } from 'redux';
import shoppingCart from './shoppingCart';
import products from './products';

export default combineReducers({
 shoppingCart,
 products
});
```

Entry Point

In the application entry point, when creating the store we need to enable the middleware library that intercepts the dispatch and runs the asynchronous actions.

```
const store = createStore(rootReducer, applyMiddleware(thunk));
```

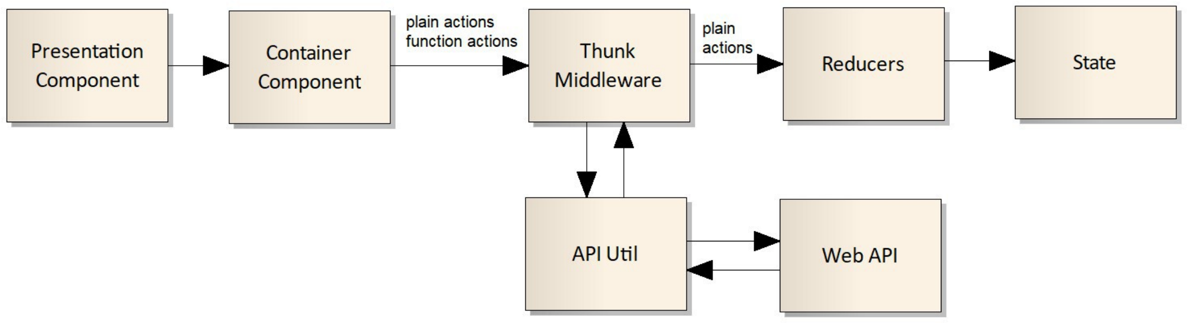
Here is the `index.js` file updated:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { createStore, applyMiddleware } from 'redux';
import { Provider } from 'react-redux';
import thunk from 'redux-thunk';

import rootReducer from './reducers';
import thunks from './thunks/productsThunks';
import App from './App';

const store = createStore(rootReducer, applyMiddleware(thunk));
store.dispatch(thunks.fetchProducts());

ReactDOM.render(<Provider store={store}><App /></Provider>,
document.getElementById('root'));
```

Data Flow

After adding the asynchronous actions, the data flow changes.

Here is a visual representation of it.



Some examples of asynchronous tasks are network calls, timer calls.

Final Thoughts

The Redux Thunk middleware enables two kinds of actions in an application, plain data actions and function actions. Dispatching a plain data action changes the store. Dispatching a function action runs the logic inside the function doing the asynchronous tasks and dispatching the plain object(s).

Synchronous action creators are functions that return plain objects.

Asynchronous action creators are functions that return functions. The returned functions are called thunks.

Middleware is an extension of the Redux store. It allows doing tasks after dispatching an action but before reaching the reducer. The middleware is the place for doing asynchronous tasks.

Chapter 15: Fetching with Redux Observable

In this chapter, we are going to look at an alternative way to orchestrate asynchronous actions in Redux.

Redux Observable is a middleware for Redux that introduces the observable streams from RxJS .

```
npm install --save redux-observable
```

Observables are streams of data.

We can subscribe to such a source of data using the `subscribe()` method. The `subscribe()` method decides how to react when receiving a new value.

```
import { Observable } from 'rxjs';

const observable = Observable.create(function(source) {
 source.next(1);
 source.next(2);
 setTimeout(() => {
  source.next(3);
  source.complete();
 }, 1000)
});

observable.subscribe(console.log);
//1
//2
//3 after 1 second
```

Operators

The operators that can be applied on an observable are inspired by the functional array methods like `filter()` , `map()` , or `reduce()` .

For example, to transform the emitted values from an observable source, we can use the `map()` operator.

> The `map()` operator returns a new observable of the transformed values using the mapping function.
>
> The `filter()` operator returns a new observable of the filtered values using the predicate function.

Here is an example using the `filter()` operator to select only the even numbers from the observable stream of data.

```
import { of } from 'rxjs';
import { filter } from 'rxjs/operators';

const observable = of(1, 2, 3, 4, 5, 6);

observable.pipe(
  filter(isEven)
 )
 .subscribe(console.log);
 //2
 //4
 //6

function isEven(n){
  return n % 2 == 0;
}
```

API

The easiest way to create an observable is through built-in functions. The `ajax.getJSON()` function makes a network call and returns an observable. This observable emits the response object returned from the request.

```
import { ajax } from 'rxjs/ajax';

const baseUrl = 'http://localhost:3000';

function fetchProducts(){
 return ajax.getJSON(`${baseUrl}/fruits/`)
}

export default { fetchProducts };
```

Actions

Our aim is to retrieve the products from the REST API and update the store. For this, we are going to need two actions.

```
import { createAction } from 'redux-actions';

const resetProducts = createAction('resetProducts');
const fetchProducts = createAction('fetchProducts');

export default { resetProducts, fetchProducts };
```

Epics

An epic is a core primitive of Redux Observable. It is a function that takes an observable stream of actions and returns a new observable stream of actions.

The epic for retrieving the products first filters out all the actions received and handles only the `fetchProducts` actions. Then for these actions, it makes an API calls using the `api.fetchProducts()` . The API call returns an observable whose data is then transformed into a new observable emitting the `resetProducts` action.

```
import api from '../api/productsAPI';
import actions from '../actions/productsActions';
import { ofType } from 'redux-observable';
import { map, mergeMap } from 'rxjs/operators';

function fetchProducts(action$){
 return action$.pipe(
  ofType(actions.fetchProducts),
  mergeMap(action$ =>
    api.fetchProducts().pipe(
     map(actions.resetProducts)
    )
  )
 );
}

export default { fetchProducts };
```

Entry Point

In the application entry point when creating the store we need to define the `epicMiddleware` using the `createEpicMiddleware()` utility.

The `epicMiddleware` needs one root epic. The root epic, containing all the other epics, is created using the `combineEpics()` utility function.

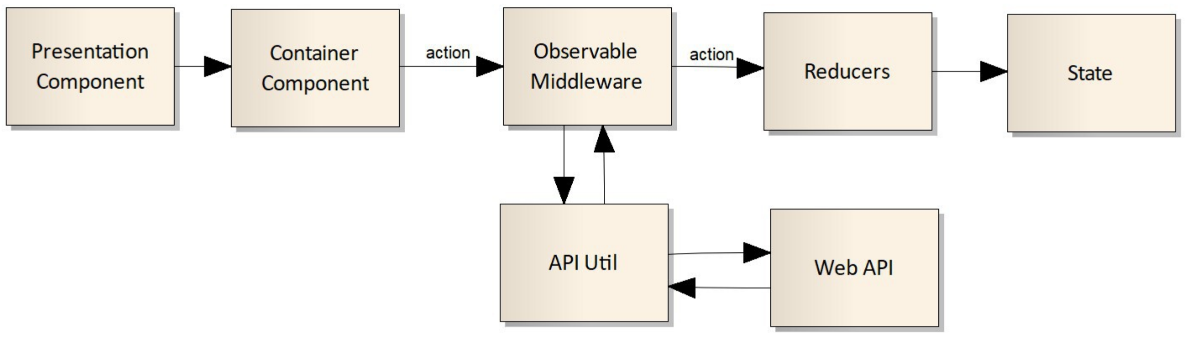
```
import { combineEpics, createEpicMiddleware } from 'redux-
observable';
import rootReducer from './reducers';
import actions from './actions/productsActions';
import epics from './async/productsEpics';

const rootEpic = combineEpics(epics.fetchProducts);
const epicMiddleware = createEpicMiddleware();
const store = createStore(rootReducer,
 applyMiddleware(epicMiddleware)
);

epicMiddleware.run(rootEpic);

store.dispatch(actions.fetchProducts());
```

At the start, the store dispatches the `fetchProducts` action.

The Data Flow

When the `fetchProducts` action is dispatched, it is intercepted by the epic middleware which runs the `fetchProducts()` epic.

Here is the data flow:



Final Thoughts

An observable represents a stream or source of data that can arrive over time.

The operators that can be applied on observables are similar to the functional array methods like `filter()` , `map()` , or `reduce()` .

An epic is a function taking an observable of all the actions, filtering them out, and returning an observable of new actions.

Chapter 16: Form Components

Form components are made of `input` form elements inside a `form` element.

Form elements like `input` , `textarea` and `select` usually have their own state that is updated based on user input.

An input element with both the value and the `onChange` properties controlled by React is a controlled component. The `value` property displays the state. The `onChange` event receives a handler to modify the state.

`input` , `textarea` and `select` all have the `value` and the `onChange` properties. The checkbox is different as the value is represented by the `checked` property.

From Component

We are going to build the `ProductSearch` form component. It lets the user write the search criteria and builds the search `query` object.

```
import React, { useState } from 'react';

import './ProductSearch.css';

function ProductSearch({ onSearch }) {
 const [text, setText] = useState('');

 function submitSearch(e){
  e.preventDefault();
  onSearch({text});
 }

 return (
 <form
  className="search-form"
```

```
    onSubmit={submitSearch}>

    <input
     value={text}
     onChange={e=> setText(e.target.value)}
     placeholder="Product"
     type="text"
     name="text"
    />

    <button
     type="search"
     className = "search-button">
     Search
    </button>
  </form>
  );
}

export default ProductSearch;
```

Let’s analyze the code. First, the state for the search textbox input is defined using the `useState()` hook.

```
const [text, setText] = useState('');
```

The state is then displayed in the textbox using the `value` property.

```
<input value={text} />
```

Input changes are captured with the `onChange` event. `onChange` fires on each keystroke, not only on lost focus. In the `event` object, the `event.target.value` property gives access to the current input value.

```
<input
```

```
value={text}
onChange={e=> setText(e.target.value)} />
```

In essence, we can say that controlled inputs have a bidirectional relationship with the associated state. When the text changes using the `setText()` function, the associated input is updated. When the value changes in the input textbox, the change event fires and updates the associated `text` state. At this point, if we need the current value form the textbox input in some other function, we have it in the `text` variable.

Form submission can be handled with the `onSubmit` event on the `form` element.

```
<form onSubmit={submitSearch}>
```

The default behavior of a form with a search button is to submit the form and reload the page. We don't want to reload the page. For this, we are going to call the event `preventDefault()` method in the event handler.

```
function submitSearch(e){
 e.preventDefault();
 onSearch({text});
}
```

`submitSearch()` builds the search `query` using the `text` variable storing the current input textbox value.

The `ProductSearch` component exposes the `onSearch` event property.

Custom Hook

Custom hooks allow us to extract out component logic into reusable functions.

A custom hook is a function whose name starts with `"use"` and that may call other hooks. Each hook use has its own state.

Let's simplify the input state management with a custom hook.

```
import { useState } from 'react';

function useInputState(initialValue){
 const [value, setValue] = useState(initialValue);

 function setValueFromEvent(e){
  setValue(e.target.value);
 }

 return [value, setValueFromEvent];
}

export { useInputState };
```

`useInputState()` is similar in concept with the `useState()` hook. The difference is that the set function returned by the custom hook expects to be called with a change event object. In short, the set function should be used as an event handler for the `onChange` event.

Here is how it can be used to handle the `onChange` event:

```
import React from 'react';
import { useInputState } from './hooks';
//...

function ProductSearch({ onSearch }) {
 const [text, setText] = useInputState('');

 //...

 <input
  value={text}
  onChange={setText}
```

```
    placeholder="Product"
    type="text"
    name="text" />

  //...
}
```

Handling the change event becomes simpler with the `useInputState()` custom hook.

App

In the `App` root component, we have access to the `query` object submitted by the `ProductSearch` component.

```
import React from 'react';
import ProductSearch from './ProductSearch';
//...

function App({products}) {
  function filterProducts(query){
    console.log(query);
  }

  //...
  <ProductSearch
   onSearch={filterProducts} />
  //...
}

export default App;
```

The `filterProducts()` function handles the `onSearch` event exposed by the `ProductSearch` component.

Final Thoughts

Form components typically have an internal state.

An input form element with the value and `onChange` properties controlled by React is a controlled component.

Custom Hooks allows the sharing of logic between components.

Chapter 17: UI State in MobX Store

The application state can contain data from the REST API and data related only to the user interface.

UI Store

The `UIStore` will manage the state related to the user interface. It is a simple store in the sense it just updates the state and triggers reactions.

In our case, the `UIStore` keeps the `query` criteria. The `query` is not something we want to save on the server, it is just the information we need for filtering the products.

```
import { observable, action } from 'mobx';

function UIStore(){
 const state = observable({
  query : {
   text : ''
  }
 });

 function getQuery(){
  return state.query;
 }

 const setQuery = action(function(query){
  state.query = query;
 });

 return Object.freeze({
  getQuery,
    setQuery
  });
}

export default UIStore;
```

The `UIStore` is simple. It just gets and sets the `query` .

ProductSearch Container

Once the `UIStore` is created, we need to make it available to the `ProductSearch` component. A new container component makes this connection. It uses `uiStore.setQuery()` to handle the `onSearch` event.

```
import { inject } from 'mobx-react';

import ProductSearch from '../ProductSearch';

const withPropsMapped = inject(function(stores){
  return {
    onSearch: stores.uiStore.setQuery
  };
});

export default withPropsMapped(ProductSearch);
```

ProductList Container

The `ProductListContainer` has already transformed `ProductList` into an observer. We just need to make it filter the products by the `query` criteria. This is accomplished by the `filterProducts()` pure function.

```
import { inject, observer } from 'mobx-react';

import ProductList from '../ProductList';
```

```
const withPropsMapped = inject(function(stores){
 return {
  products : filterProducts(stores.productStore.getProducts(),
      stores.uiStore.getQuery()),
  onAddClick : stores.shoppingStore.addToCart
 };
});

function filterProducts(products, query){
 return products.filter(isInQuery(query));
}

function isInQuery(query){
 return function(product){
  return product.name.includes(query.text);
 };
}

export default withPropsMapped(observer(ProductList));
```

Entry Point

In `index.js` all stores, including `uiStore` , are created and then passed to components.

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'mobx-react';

import api from './api/productsAPI';
import ProductStore from './stores/ProductStore';
import ShoppingCartStore from './stores/ShoppingCartStore';
import UIStore from './stores/UIStore';
import App from './App';

const shoppingStore = ShoppingCartStore();
const productStore = ProductStore(api);
const uiStore = UIStore();

const stores = {
  productStore,
  shoppingStore,
  uiStore
};

ReactDOM.render(<Provider {...stores}>
  <App />
  </Provider>,
 document.getElementById('root'));

productStore.fetchProducts();
```

Final Thoughts

Stores keep data from the backend and data related to the user interface.

The UI Store is a simple store. When state changes in the UI Store, the components using it will re-render.

Chapter 18: UI State in Redux Store

Let’s implement the search product functionality with Redux.

Searching by a query requires to actually store the query itself. The `query` in this case, is UI state.

Actions

Let’s start by defining the action we need for storing the `query` . Here is the action creator for it:

```
import { createAction } from 'redux-actions';

const setQuery = createAction('setQuery');

export default { setQuery };
```

Query Reducer

To store the `query` we need to define a new reducer managing this state.

```
import { handleActions } from 'redux-actions';
import actions from '../actions/queryActions';

function setQuery(state, action) {
 return action.payload;
}

export default handleActions({
  [actions.setQuery] : setQuery
 },
 { text: ''}
);
```

Root Reducer

The `query` reducer needs to be added to the `combineReducers()` utility creating the root reducer.

```
import { combineReducers } from 'redux';
import shoppingCart from './shoppingCart';
import products from './products';
import query from './query';

export default combineReducers({
 shoppingCart,
 products,
 query
});
```

ProductSearch Container

The `ProductSearch` component doesn’t change, it remains the same. What does change is the way it is connected to the store.

We will create a new container component that updates the store on the `onSearch` event.

When the Search button is clicked, the `setQuery` action is dispatched.

```
import { connect } from 'react-redux';
import actions from '../actions/queryActions';
import ProductSearch from '../ProductSearch';

function mapStateToProps(state) {
 return {
  query: state.search
 };
}

function mapDispatchToProps(dispatch) {
 return {
  onSearch: function(query){
   dispatch(actions.setQuery(query));
  }
 };
}

export default connect(
 mapStateToProps,
 mapDispatchToProps
)(ProductSearch);
```

Selector

The `filterProducts()` selector extracts the `products` and the current search `query` from the state and returns a new filtered list of products.

```
function filterProducts({products, query}){
 return products.filter(isInQuery(query));
}

function isInQuery(query){
 return function(product){
  return product.name.includes(query.text);
 };
}

export default {filterProducts}
```

`isInQuery()` is a manually curried function. It takes a query object and returns a function that takes a product and tests that product to see if it matches the query. As such it can be nicely used with the `filter()` method.

ProductList Container

The `ProductList` container reads the filtered list of products from the store using the `filterProducts()` selector.

```
import { connect } from 'react-redux';
import actions from '../actions/shoppingCartActions';
import selectors from '../selectors/productsSelectors';

import ProductList from '../ProductList';

function mapStateToProps(state) {
 return {
  products: selectors.filterProducts(state)
 };
}

function mapDispatchToProps(dispatch) {
 return {
  onAddClick: function(product){
   dispatch(actions.addToCart(product));
  }
 };
}

export default connect(
 mapStateToProps,
 mapDispatchToProps
)(ProductList);
```

Final Thoughts

UI state can be stored and changed in the Redux store the same way other data is stored and changed.

Chapter 19: Testing

Unit tests can identify bugs and confirm that functions work as expected.

We are going to make a few tests using the Jest testing framework. Jest is already installed in applications created with Create React App . Tests can be defined in files named with the `.test.js` suffix.

The `npm test` command executes the tests.

Testing Redux Reducers

The next test checks that the `product` reducer resets all products in the store when the `resetProducts` action is dispatched.

```
import actions from '../actions/productsActions';
import productsReducer from './products';

test('products() can reset all products', function() {
 //arrange
 const products = [];
 const newProducts = [
  {id:1, name: 'apple', price: 10},
  {id:2, name: 'mango', price: 5}
 ];

 //act
 const resetProductsAction =
actions.resetProducts(newProducts);
 const result = productsReducer(products, resetProductsAction);

 //assert
 expect(Array.from(result)).toEqual(newProducts);
});
```

We assert our expectations using the `expect()` function. `expect()` returns an expectation object.

The `toEqual()` method can be called on the expectation object to test the value for equality. `toEqual()` recursively checks every field of an object or array.

The following tests verify that the `shoppingCart` reducer can add products to cart, increment the quantity of an existing product, and remove a product from the cart.

```
import { Map } from 'immutable';
import actions from '../actions/shoppingCartActions';
import selectors from '../selectors/shoppingCartSelectors';
import shoppingCartReducer from './shoppingCart';

test('shoppingCart() can add products', function() {
 //arrange
 const cartMap = Map();

 //act
 const addToCartAction =
  actions.addToCart({id:1, title: 'apple', price: 10});
 const shoppingCart =
  shoppingCartReducer(cartMap, addToCartAction);

 //assert
 const cart = selectors.toCartView({shoppingCart});
 expect(cart.list.count()).toEqual(1);
});

test('shoppingCart() can increment quantity', function() {
 //arrange
```

```
  let cartMap = Map();
  cartMap = shoppingCartReducer(cartMap,
    actions.addToCart({id:1, title: 'apple', price: 10}));
  cartMap = shoppingCartReducer(cartMap,
    actions.addToCart({id:2, title: 'mango', price: 5}));

  //act
  const addToCartAction =
    actions.addToCart({id:1, title: 'apple', price: 10});
  const shoppingCart =
    shoppingCartReducer(cartMap, addToCartAction);

  //assert
  const cart = selectors.toCartView({shoppingCart});
  expect(cart.list.count()).toEqual(2);
  expect(cart.list.first().quantity).toEqual(2);
});

test('shoppingCart() can remove product', function() {
  //arrange
  let cartMap = Map();
  cartMap = shoppingCartReducer(cartMap,
    actions.addToCart({id:1, title: 'apple', price: 10}));
  cartMap = shoppingCartReducer(cartMap,
    actions.addToCart({id:2, title: 'mango', price: 5}));

  //act
  const removeFromCartAction =
    actions.removeFromCart({id:1, title: 'apple', price: 10});
  const shoppingCart =
   shoppingCartReducer(cartMap, removeFromCartAction);

  //assert
  const cart = selectors.toCartView({shoppingCart});
  expect(cart.list.count()).toEqual(1);
  expect(cart.list.first().id).toEqual(2);
});
```

We are using immutable data structures to store data in the store. The `cart.list` , in this case, is a sequence .

  `count()` gets the number of values in the sequence.

  `first()` gets the first value in the sequence.

The `query` UI reducer should be able to reset the `query` when the `setQuery` action is dispatched.

```
import actions from '../actions/queryActions';
import queryReducer from './query';

test('query() can set query', function() {
  //arrange
  const query = { text: '' };
  const newQuery = { text : 'apple'};

  //act
  const queryAction = actions.setQuery(newQuery);
  const result = queryReducer(query, queryAction);

  //assert
  expect(result).toEqual(newQuery);
});
```

Testing Selectors

The following test verifies that the `toCartView()` selector is able to transform the map of products into a cart object with all products and the total price.

```
import { Map } from 'immutable';
import selectors from './shoppingCartSelectors';

test('toCartView() can compute total price', function() {
 //arrange
 let shoppingCart = Map({
  1: {id:1, title: 'apple', price: 10, quantity: 1},
  2: {id:2, title: 'mango', price: 5, quantity: 1}
 });

 //act
 const cart = selectors.toCartView({shoppingCart});

 //assert
 expect(cart.total).toEqual(15);
});
```

Remember that we are using the Map immutable data structure to store the quantity in the cart for each product. The product `id` is the key in the map and the `product` itself with its quantity is the value in the map.

Next, we are going to check that the `filterProducts()` selector can filter the `products` by the `query` criteria.

```
import { List } from 'immutable';
import selectors from './productsSelectors';

test('filterProducts() can filter products by query',
function() {
 //arrange
 const products = List([
  {id:1, name: 'apple', price: 10},
  {id:2, name: 'mango', price: 5}
 ]);
 const query = {text: 'app'};

 //act
 const result = selectors.filterProducts({products, query});

 //assert
 expect(result).toEqual(List([
  {id:1, name: 'apple', price: 10}
 ]));
});
```

Products are kept in the store using the List immutable data structure.

Testing MobX Stores

Below are a few tests for the `ShoppingCartStore` .

```
import ShoppingCartStore from './ShoppingCartStore';

test('ShoppingCartStore can add products', function() {
 //arrange
 const store = ShoppingCartStore();

 //act
 store.addToCart({id:1, title: 'apple', price: 10});
 store.addToCart({id:2, title: 'mango', price: 5});

 //assert
 const cart = store.toCartView();
 expect(cart.list.length).toEqual(2);
});

test('ShoppingCartStore can increment quantity', function() {
 //arrange
 const store = ShoppingCartStore();
 store.addToCart({id:1, title: 'apple', price: 10});
 store.addToCart({id:2, title: 'mango', price: 5});

 //act
 store.addToCart({id:1, title: 'apple', price: 10});

 //assert
 const cart = store.toCartView();
 expect(cart.list.length).toEqual(2);
 expect(cart.list[0].quantity).toEqual(2);
});
```

Testing a store with a dependency requires to create a fake object and pass it as the dependency.

```
import ProductStore from './ProductStore';

test('ProductStore can reset all products', function() {
 //arrange
 const newProducts = [
  {id:1, title: 'apple', price: 10},
  {id:2, title: 'mango', price: 5}
 ];
 const apiFake = {
  fetchProducts : function(){
      return Promise.resolve(newProducts);
     }
   };
   const store = ProductStore(apiFake);

   //act
   return store.fetchProducts(newProducts)
     .then(function assert(){
       //assert
       const products = store.getProducts();
       expect(products.length).toEqual(2);
     });
});
```

`Promise.resolve(value)` returns a promise resolved with the specified `value`.

When testing an asynchronous code that returns a promise, we need to return the promise from the test. This way Jest waits for that promise to resolve. If we don’t return the promise, the test will complete before the promise is resolved or rejected and the test will not execute the assert callback.

Testing Application Start

We can create a smoke test verifying that the application can render the root component without errors.

Here is the start smoke test for the Redux application.

```
import React from 'react';
import ReactDOM from 'react-dom';
import { createStore, applyMiddleware } from 'redux';
import { Provider } from 'react-redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers';
import thunks from './thunks/productsThunks';
import App from './App';

it('App can start app without crashing', () => {
 const rootDiv = document.createElement('div');
 const store = createStore(rootReducer,
applyMiddleware(thunk));
 store.dispatch(thunks.fetchProducts());
 ReactDOM.render(<Provider store={store}><App /></Provider>,
  rootDiv);
});
```

Similarly, we can test the start of the MobX application.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import { Provider } from 'mobx-react';

import api from './api/productsAPI';
import ProductStore from './stores/ProductStore';
import ShoppingCartStore from './stores/ShoppingCartStore';
import UIStore from './stores/UIStore';

it('App can start app without crashing', () => {
 const rootDiv = document.createElement('div');

 const shoppingStore = ShoppingCartStore();
 const productStore = ProductStore(api);
 const uiStore = UIStore();
 const stores = {
  productStore,
  shoppingStore,
  uiStore
 };

 productStore.fetchProducts();

 ReactDOM.render(<Provider {...stores}><App /></Provider>,
  rootDiv);
});
```

Final Thoughts

Selectors and reducers are pure functions and we can create tests verifying the expected output for specific input.

Stores can be tested in isolation using fake dependencies.

A simple test rendering the root component can check if the application can start without errors.

Chapter 20: Redux Folder Structure

In this chapter, we are going to analyze and restructure the current folder organization. This will be a good way to understand the pros and cons of different options.

Type-First Folder Structure

When using this structure, each file type goes into the corresponding directory. For example, all actions are inside the `"actions"` directory, each reducer within the `"reducers"` directory, and so on.

```
actions/

api/

components/

containers/

reducers/

selectors/

thunks/
```

Next, inside the folder for a specific type, we will find the files for specific features. Here are the files in the `"actions"` folder:

```
|--productsActions.js

|--queryActions.js

|--shoppingCartActions.js
```

Inside the `"reducers"` folder, we find a reducer for each feature.

```
|--products.js

|--query.js

|--shoppingCart.js
```

The strong point of this folder strategy is its simplicity. No explanation is required. It works well when the application has a few features. When the number of features grows we still add the files into the same fixed number of folders and that can become a problem.

Feature-First Folder Structure

The alternative solution is to first group the files by the feature they belong and then split them by their type. This is called feature-first.

Let’s reorganize the files using this folder strategy.

```
products/
|-- actions.js
|-- reducer.js
|-- api.js
|-- selectors.js
|-- thunks.js
|-- components/
  |-- ProductItem.jsx
  |-- ProductItem.css
  |-- ProductList.jsx
  |-- ...
|-- containers/
  |-- ProductListContainer.jsx
  |-- ProductItem.css
  |-- ...
shopping-cart/
|-- actions.js
|-- reducer.js
|-- api.js
|-- selectors.js
|-- thunks.js
|-- components/
|-- containers/
```

Whenever we need to add a new feature, we add a new directory. This fits better in applications with several features. One possible weakness of this structure is the lack of certainty about the feature to which a file is linked.

One vs Multiple Files

Until now we have separated reducers, selectors, actions in their own file, but we can take a more evolutionary approach. That means we start out with a single file containing all actions, the reducer, and all sectors for a feature, and then, when it grows, we extract them out in their own files.

The same approach can be used for presentation and container components. We can put the code in the same file and then extract out the container logic in its own file when it grows.

Let’s look also at the reducers for the existing features.

```
products/
|--reducers/
  |--productsReducer
  |--queryReducer
shopping-cart/
  |--reducer
```

We can follow the same idea and keep the state management in one reducer file at start and then when it grows, we extract the code out in several files under the `"reducers"` sub-folder of the feature.

Final Thoughts

Type-first and feature-first are two options available for organizing files into directories.

We can take a more evolutionary approach and extract out the code from a file into several others when it grows.

Chapter 21: Router

All the functionality implemented until now happened on one page, the products page. In this chapter, we are going to introduce a new page, the product details page.

Let’s start implementing this new page.

Product Component

The `ProductDetails` functional component creates the visual interface displaying all the product details. In addition to the `name` and `price` , our products may have the `nutrients` property.

Here is an example of a detailed product:

```
{
 "id": 1,
 "name": "mango",
 "price": 10,
 "nutrients": [
  {
   "name": "Calories",
   "value": 99
  },
  {
   "name": "Protein",
   "value": "1.4 grams"
  }
 ]
}
```

The `ProductDetails` displays the nutrients property only when it is available. This is called conditional rendering .

```
import React from 'react';
import partial from 'lodash/partial';
import './ProductDetails.css';

function ProductDetails({ product, onAddClick }) {
 return (
  <div className="product-details">
   <div>{product.name}</div>
   <div>Price: {product.price}</div>
   <div>
    <div>
     {product.nutrients &&
      product.nutrients.map(n =>
       <div key={n.name}>
        {n.name}: {n.value}
       </div>
     )}
    </div>
   </div>
   <div>
    <button
     type="button"
     onClick={partial(onAddClick, product)}>
       Add
    </button>
   </div>
  </div>
 );
}
```

```
export default ProductDetails;
```

The logical `&&` operator can be handy for conditionally rendering an element. If the condition is `true` , the element after `&&` will be rendered, otherwise, it will not be displayed.

The associated container component dispatches the `addToCar` action on the Add click event.

```
import { connect } from 'react-redux';
import actions from '../shopping-cart/actions';
import ProductDetails from './ProductDetails';

function mapDispatch(dispatch) {
 return {
  onAddClick(product){
   dispatch(actions.addToCart(product));
  }
 };
}

export default connect(
 null,
 mapDispatch
)(ProductDetails);
```

State

The Redux flow is a data-driven one. In order to display the current product, first, we need this information in the store.

Actions

To save the current product in the store, we need a new action.

```
import { createAction } from 'redux-actions';

const setCurrentProduct = createAction('setCurrentProduct');
```

Reducer

A new reducer is required to manage the current product state. When `setCurrentProduct` is dispatched, it will update the current product with the new one.

```
import { handleActions } from 'redux-actions';
import actions from '../actions'

function setCurrentProduct(state, action) {
 return action.payload;
}

export default handleActions({
  [actions.setCurrentProduct] : setCurrentProduct
 },
 null
);
```

API Utils

A new API utility function is needed to fetch all product details by id.

```
const baseUrl = 'http://localhost:3000';

function toJson(response){
 return response.json();
}

function fetchProduct(id){
 return fetch(`${baseUrl}/fruits/${id}`)
  .then(toJson)
}
```

Thunks

The `fetchProduct` thunk uses the `api.fetchProduct()` utility to get the product details from the REST API, and then dispatches the `setCurrentProduct` action containing the product information.

```
import api from './api';
import actions from './actions';

function fetchProduct(id) {
 return function(dispatch) {
  return api.fetchProduct(id)
   .then(actions.setCurrentProduct)
   .then(dispatch);
 };
}
```

Root Reducer

The root reducer must be updated with the new reducer managing the current product.

```
import { combineReducers } from 'redux';
import shoppingCart from './shopping-cart/reducer';
import products from './products/reducers/productsReducer';
import query from './products/reducers/queryReducer';
import currentProduct from
'./products/reducers/currentProductReducer';

export default combineReducers({
  shoppingCart,
  products,
  query,
  currentProduct
});
```

Router

Navigating between several pages requires routing. We are going to use the React Router library to define the routes to these pages and create the links pointing to them.

```
npm install --save react-router-dom
```

Pages

We should at this point make a difference between pages and other reusable components. The page until now was the root App functional component.

Let’s create these pages.

Products Page

We are going to create the products page from the previous `App` component and do a few changes.

We are not loading the products when the application starts but when the products page is loaded. For this, we need to dispatch the `fetchProducts` thunk when the page is loaded, using the `useEffect()` hook.

```
import React, {useEffect} from 'react';


import ProductSearch from './products/ProductSearchContainer';
import ProductList from './products/ProductListContainer';
import ShoppingCart from './shopping-
cart/ShoppingCartContainer';
```

```
import './App.css';

function Products({onLoad}) {

 useEffect(() => {
  onLoad();
 },[]);

 return (
  <div className="content">
   <div>
    <ProductSearch />
    <ProductList />
   </div>
   <ShoppingCart />
  </div>
 );
}

export default Products;
```

The container component dispatch the `fetchProducts` thunk on the `onLoad` event.

```
import { connect } from 'react-redux';
import thunks from './products/thunks';
import Products from './Products';

function mapDispatch(dispatch) {
 return {
  onLoad(){
   dispatch(thunks.fetchProducts());
   }
  };
}

export default connect(
  null,
  mapDispatch
)(Products);
```

Product Details Page

The products page displays the details for the selected products. The `onLoad` callback is called with the product `id` when the page is loaded.

```
import React, {useEffect} from 'react';

import ProductDetails from
'./products/ProductDetailsContainer';
import ShoppingCart from './shopping-
cart/ShoppingCartContainer';

import './App.css';

function Product({ id, currentProduct, onLoad }){

  useEffect(() => {
   onLoad(id);
  },[id]);

  return (
   <div className="content">
    <div>
     { currentProduct &&
      <ProductDetails product={currentProduct} /> }
    </div>
    <ShoppingCart />
   </div>
  )
}

export default Product;
```

The container component reads the current `id` from the route using the `match.params` property and extracts the `currentProduct` from the store. It dispatches the `fetchProduct` thunk on the `onLoad` event.

```
import { connect } from 'react-redux';
import thunks from './products/thunks'
import Product from "./Product";

function mapProps({currentProduct}, {match}){
 const {id} = match.params;

 return {
  id,
  currentProduct
 }
}

function mapDispatch(dispatch) {
 return {
  onLoad(id){
   dispatch(thunks.fetchProduct(id));
  }
 };
}

export default connect(
 mapProps,
 mapDispatch
)(Product);
```

Header

Once we navigate to the details page, we need a way to get back. We can add a link the the `Hearder` component pointing to the products page.

The `<Link>` component from React Router allows navigating to different routes defined in the application. It can be imagined like an anchor link. Navigation links created with the `<Link>` component don’t make a page refresh. The `to` property specifies the navigating route.

```
import React from 'react';
import { Link } from 'react-router-dom';

import './Header.css'

function Header() {
 return (
  <header>
   <h1>A Shopping Cart</h1>
   <nav className="content">
     <Link to="/">Products</Link>
   </nav>
  </header>
 );
}
```

```
export default Header;
```

App Router

The `AppRouter` component defines all the routes using the `Route` component.

The `path` attribute declares the path used in the URL and the `component` attribute defines the component to be rendered when the route matches the URL.

```
import React from 'react';
import { BrowserRouter as Router, Route } from 'react-router-
dom';

import Header from './Header';
import Products from './ProductsContainer';
import Product from './ProductContainer';

function AppRouter(){
 return (
  <Router>
   <Header />
   <Route
    exact path="/"
    component={Products} />
   <Route
    path="/products/:id/"
    component={Product} />
  </Router>
 );
}
```

```
export default AppRouter;
```

Entry Point

In the `index.js` file, we are going to use the `AppRouter` as the root component.

```
import React from 'react';
import ReactDOM from 'react-dom';
import { createStore, applyMiddleware } from 'redux';
import { Provider } from 'react-redux';
import thunk from 'redux-thunk';
import { composeWithDevTools } from 'redux-devtools-extension';

import AppRouter from './AppRouter';
import rootReducer from './rootReducer';

const store = createStore(rootReducer, composeWithDevTools(
 applyMiddleware(thunk),
));

ReactDOM.render(<Provider store={store}><AppRouter />
</Provider>, document.getElementById('root'));
```

Final Thoughts

React Route enables rounding in an application.

The component with all the routes becomes the root component. The `<Link>` components define navigation to routes defined using the `<Route>` component.

The `useEffect()` hook can be used to detect the page load.

Chapter 22: Functional React

React embraces functional programming and that's why it is of great value to master the functional programming concepts when working in React.

In this chapter, we are going to take a look back and see how the functional programming principles were used in the application.

Functional Array

> `filter()` selects values from a list using a predicate function.
>
> `map()` transforms a list of values to another list of values using a mapping function.
>
> `reduce()` reduces a list of values to one value using a reducer function.

The basic array methods were used in all data transformations, but mostly we have seen them used in reducers and selectors.

The `map()` method was used in all list components.

Immutability

> An immutable value is a value that, once created, cannot be changed.

In functional components, props should be treated as immutable. That's it, we never modify the values in props and consider them read-only.

Even more, the state value should be treated as being immutable.

The transfer objects that move around the application from one component to another should be immutable.

Basically, we have two options when working with immutable values in JavaScript. One is to freeze all objects and arrays at creation with `Object.freeze()`. The other option is to use a library like Immutable.js that provides immutable data structures.

Pure functions

> A pure function is a function that given the same input always returns the same output and has no side-effects.
>
> A side-effect is a change of the outside environment in any way. Even more, reading data from the outside environment that can change is a side-effect.

First, we should aim to create components as pure functions. That’s it, they take data as props and return the markup defining the user interface.

Pure functional components are easier to read and understand.

Then we should aim at making the other functions pure. Reducers, selectors, action creators are pure functions.

Higher-order functions

> A higher-order function is a function that takes another function as argument, returns a function or does both.

The asynchronous action creator is a custom higher-order function that returns a function.

```
import api from '../api/productsAPI';
import actions from '../actions/productsActions';

function fetchProducts() {
 return function(dispatch) {
  return api.fetchProducts()
   .then(actions.resetProducts)
   .then(dispatch);
 };
}
```

```
export default { fetchProducts };
```

Higher-order components

> A higher-order component is a function that takes a component as input and returns a new component.

Reading data from the store and dispatching events are side-effects. We can keep components pure by encapsulating side effects in higher-order components, HoC in short.

The `observer()` utility from MobX is a higher-order component. The function created by the `inject()` utility is higher-order component.

```
import ProductList from '../ProductList';
//...

const withPropsMapped = inject(function(stores){
 return {
   //...
 };
});

export default withPropsMapped(observer(ProductList));
```

The function created by the `connect()` utility from Redux is a higher-order component.

```
import ProductList from '../ProductList';
//...

export default connect(
 mapStateToProps,
 mapDispatchToProps
)(ProductList);
```

Currying

> Currying transforms a function that takes multiple arguments into a sequence of functions where each takes one argument.

We used a manually curried function when filtering products by a query. `isInQuery()` is a curried function.

When calling `isInQuery()` with one argument it creates a new function taking the remaining argument. Currying can be useful when the function receives fewer arguments than necessary, like for example when it is used as a callback.

```
function filterProducts({products, query}){
 return products.filter(isInQuery(query));
}

function isInQuery(query){
 return function(product){
  return product.name.includes(query.text);
 };
}

export default {filterProducts}
```

Promises

> A promise is an object that represents a possible future result of an asynchronous operation.

API Util functions encapsulate the network calls that return promises. Asynchronous action creators work with promises.

Function Composition

> Function composition is a technique of passing the output of a function as the input for another function.

When creating the `ShoppingCart` container with MobX we used function composition.

`withPropsMapped()` takes as input the output of `observer()` .

```
import ShoppingCart from '../ShoppingCart';
//...

export default withPropsMapped(observer(ShoppingCart));
```

Closures

> A closure is a function that has access to the variables from its outer function, even after the outer function has executed.

Closure can encapsulate state. Multiple closures sharing the same private state can create encapsulated objects.

```
import { observable, action } from 'mobx';

//pure functions
//...

function ShoppingCartStore(){
 const shoppingMap = observable.map();

 function toCartView() {
  return toCartViewFromMap(shoppingMap);
 }

 //actions
 const addToCart = action(function(product){
  shoppingMap.set(product.id,
   incrementProductQuantity(shoppingMap, product));
 });

 const removeFromCart = action(function(product){
  shoppingMap.delete(product.id);
 });

 return Object.freeze({
  addToCart,
  removeFromCart,
  toCartView
 });
}
```

`ShoppingCartStore()` creates an object with private state. `addToCart()` , `removeFromCart()` , `toCartView()` are closures sharing the same private state `shoppingMap` . The `this` keyword is not used.

Final Thoughts

React makes it easy to build components with functions. Functional components are cleaner and easier to read. They are not using the `this` pseudo-parameter.

The best way to achieve clarity in a React application is to write as many components as pure functions, and also writing as many functions as pure functions.

Immutability avoids unexpected changes that create issues hard to understand.

`filter()` , `map()` , `reduce()` make data transformations easier to read.

Currying can help to create new functions with fewer arguments than the original functions. The new functions can be used as callbacks.

Higher-order components help to encapsulate impure code.

Multiple closures sharing the same private state can create flexible and encapsulated objects.

Chapter 23: Architecture

In this chapter, we are going to review and analyze the architectural decisions taken so far.

Props and State

Data can take two forms, props, and state.

Props

Functional components take a single object argument, called “props”, with all the JSX attributes.

There are two kinds of values passed as props, plain data objects, and function callbacks.

State

State is data that is stored and can be changed.

State can change, but props cannot.

Local and External State

State can be local to the component and is called local state. In this case, the local data can be sent to other components using props.

Reach Hooks enables functional components with local state, using the `useState()` hook. We explore the local state when building the `App` and `ProductSearch` components.

State can be external. It can be stored in a single store, like the Redux store, or in multiple stores, like the ones created with MobX.

The external state can be shared by multiple components.

With MobX we split the state between three stores: `ShoppingCartStore` , `ProductStore` , `UIStore` .

With Redux we split the state management between three reducers: `shoppingCart` , `products` and `query` .

Domain and UI State

The data we want to store can come either from an external Web API or from the user interface.

Domain state is taken from the backend.

UI state, or view state, is used in the user interface. It is not taken or stored on the backend.

The list of products in our case is the domain state. The shopping cart is domain state as it needs to be stored on the backend. The `query` search object is UI state.

Presentation Components

Presentation components take only plain data objects and function callbacks as props.

Presentation components can have a local UI state. For example, a component displaying data on multiple tabs can keep the currently selected tab as local UI state and still be a presentation component.

Presentation components are responsible for rendering the UI. Here are things presentation components don’t do:

- They don’t make network calls.
- They don’t have access to stores.
- They don’t dispatch actions.
- They don’t use objects with a publish-subscribe public interface.

We can best express presentation components as pure functions with no local state.

The presentation component encourages reusable and encapsulated code.

Presentation components transform data into a visual user interface and define the callbacks to run in response to the user interactions.

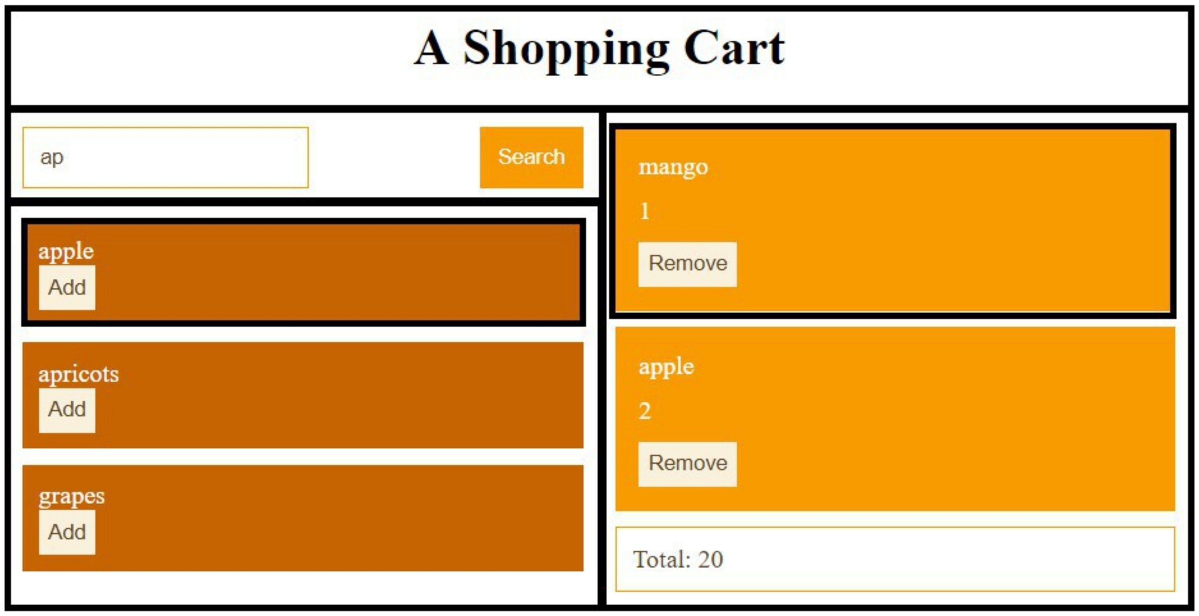
Container Components

Container components are connected to the external environment and can have side-effects.

Container components can read data from stores and dispatch actions.

We can use higher-order components to create containers.

UI Decomposition



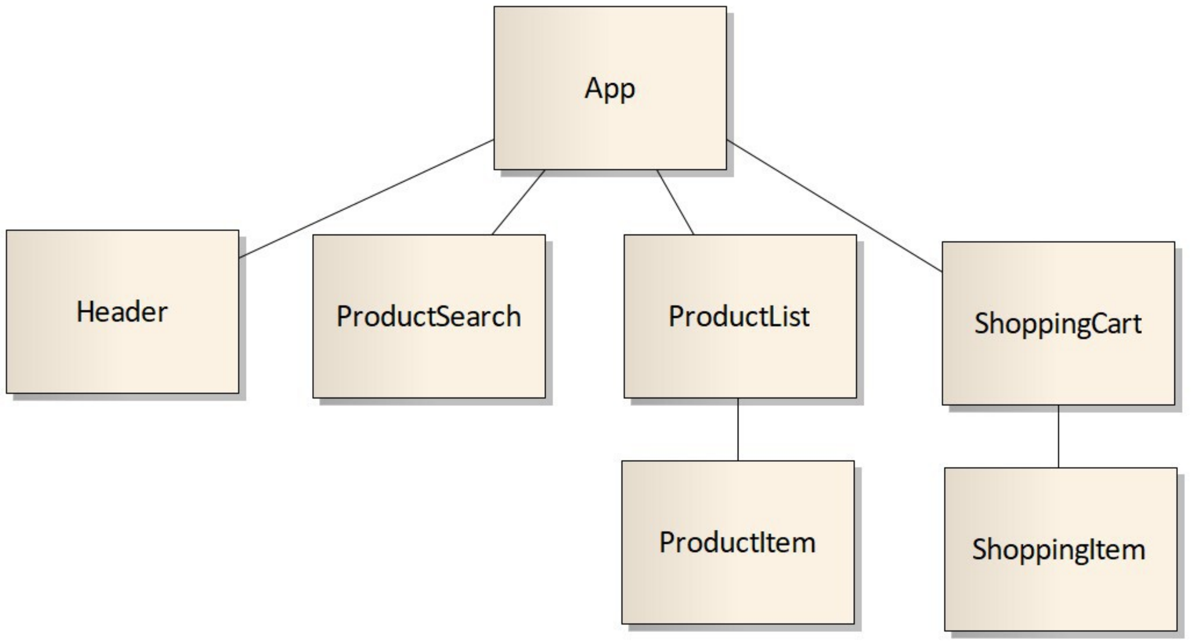
The complexity of a large UI can be handled by decomposing it into smaller pieces. Components are the basic unit for splitting the page into small parts easier to manage and reuse.

The process of decomposition is about assigning responsibilities and giving names. This makes it easy to talk and reason about.

We decomposed the products page into the following components:

- `App` the root component
- `Header` a pure function rendering the header information
- `ProductList` a pure function rendering the list of products
- `ProductItem` a pure function rendering one product from the list
- `ProductListContainer` a container component displaying the products using the `ProductList` component and communicating with the store(s)
- `ShoppingCart` a pure function rendering the shopping cart
- `ShoppingItem` a pure function rendering one product from the shopping cart
- `ShoppingCartContainer` a container component displaying the cart using the `ShoppingCart` component and communicating with the store(s)
- `ProductSearch` a presentation component rendering the search form and building the search query
- `ProductSearchContainer` a container component displaying the search form using the `ProductSearch` and updating the store with the search query

Components are organized in a tree structure. In our example, we created a tree with three levels. Let’s visualize it.



It is important to note that `ProductSearch` , `ProductList` , and `ShoppingCart` are the container components for the presentation components with the same name.

UI as Function of State

In a sense, React allows treating the UI as a function of state. There is an equation describing this relation:

```
UI = f(state)
```

`f` is a mapping function that transforms data into a UI.

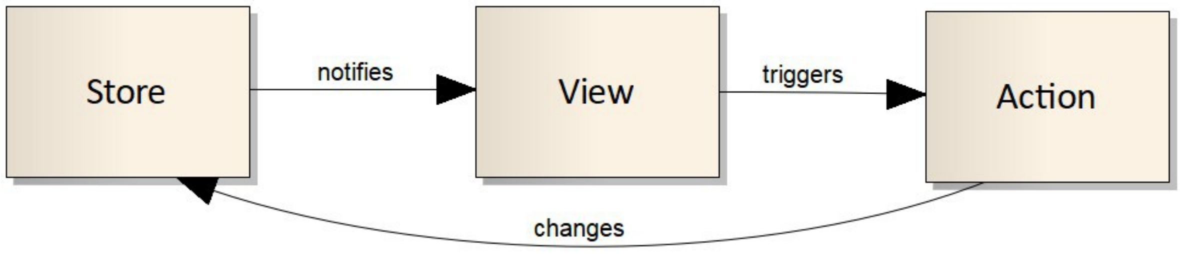
Note that this equation takes into consideration only the data props. The UI is not only about creating a visual representation of data but also about handling user interactions. The user interactions can be expressed using the concept of actions. Actions change state.

Unidirectional Data Flow

Views do not modify the data they received.

Views subscribe to store changes, so when the state changes views are notified to re-render. Views trigger actions on user interaction. These actions change the state. When the state is changed the associated views are re-render.

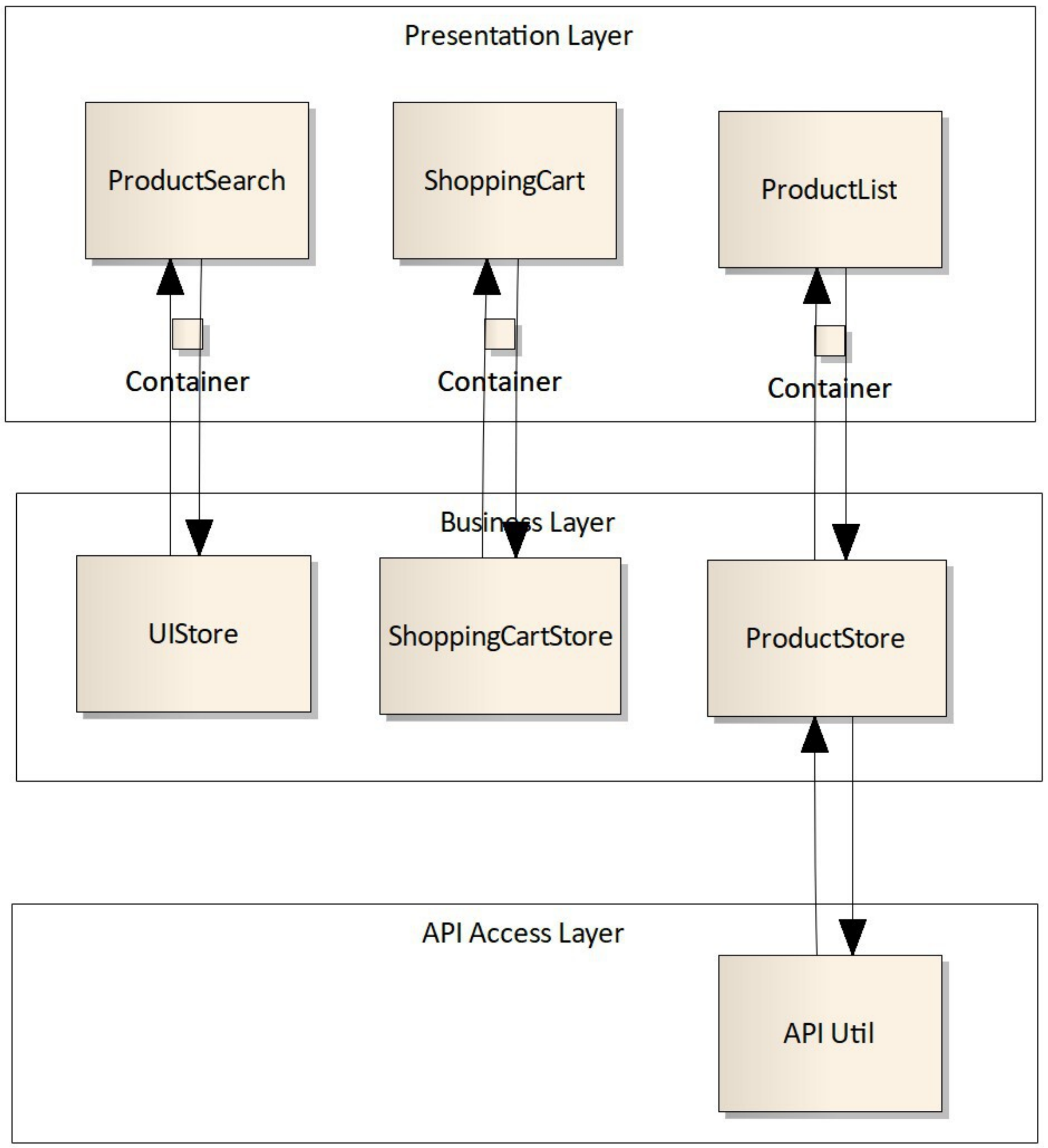
This data flow is known as the unidirectional data flow.



MobX Architecture

With MobX we tend to split the application into three layers:

- Presentation Layer: Presentation and Container Components
- Business Layer: Domain Stores and UI Stores
- API Access Layer: API Utils



Containers make presentation components observers of stores. When state changes in a store the presentation component using that data will re-render. Container components handle the presentation components events by calling methods on stores.

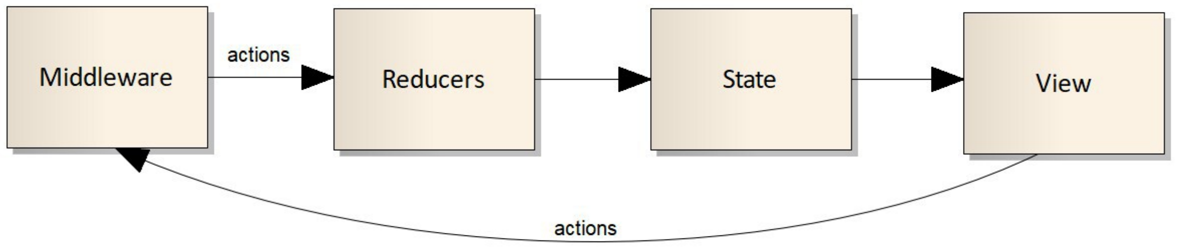
When in the entry point file `index.js` the `productStore.fetchProducts()` method is called, the store delegates the network call to the `api` object.

When the response gets back, the state is updated and that makes its observer components to re-render.

Redux Architecture

In a Redux Architecture we split the application into the following parts :

- Presentation and Container Components
- Reducers and Selectors
- Actions and Action Creators
- Asynchronous Action Creators
- API Utils



The action concept was different between the two middleware implementations. While the thunk middleware accepted plain data and function actions, the observable middleware accepted only plain action objects. Nevertheless, the actions that go from the middleware to reducers are only plain data objects in all implementations.

We express the UI using presentation components and aim for making them pure functions.

State management is split between reducers.

Container components are used to make the connection between the single store and presentation components. Container components read data from the store, use selectors to transform it, and send it to presentation components. Container components decide what actions to dispatch on events in the presentation components. When state changes inside the store the presentation components using that data will re-render.

The API Util objects make the network calls.

Asynchronous action creators coordinate asynchronous tasks and store updates. In our case, the entry point file `index.js` dispatches a thunk function, using `fetchProducts()` . The thunk function fetches data from the network and updates the store. Updating the store with new products triggers a re-render of the `ProductList` component.

Final Thoughts

The UI can be expressed as a tree of presentation components, where each component is a function.

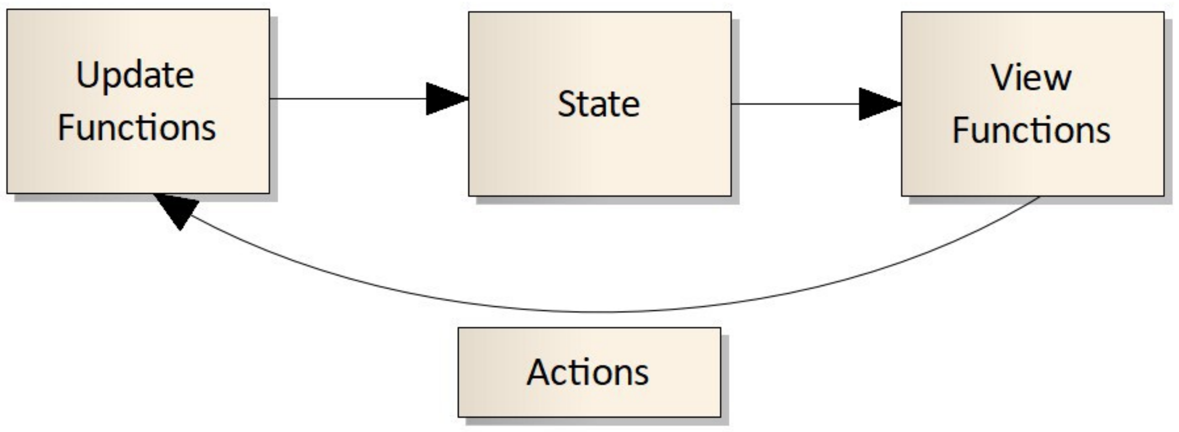
The application state can be split and managed in small parts. Container components connect the state to presentation components.

Redux takes a functional way of managing state, while MobX takes a more object-oriented approach. I favor Redux for working with the application state in a functional style.

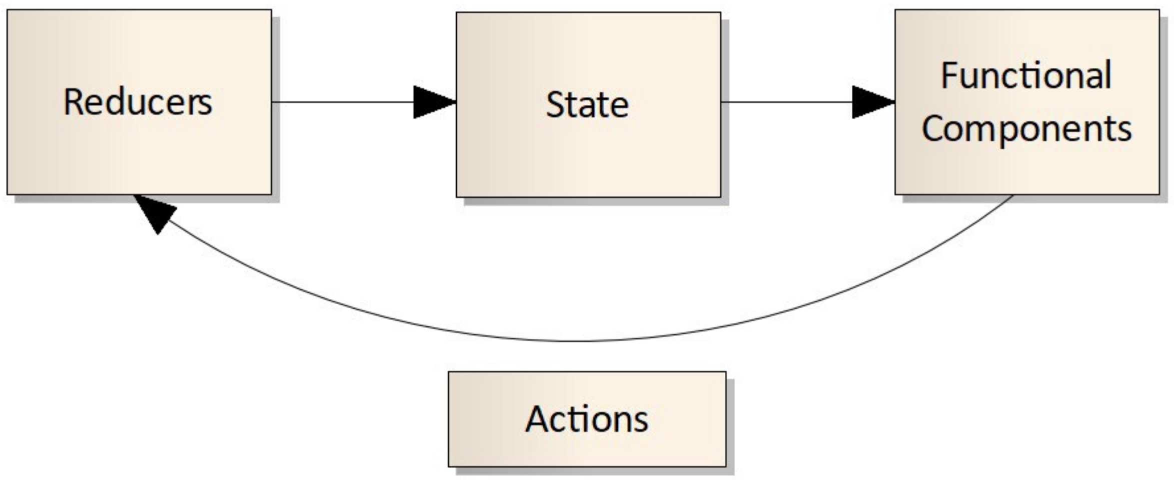
What’s next?

You can consider reading the Functional Architecture with React and Redux book, and put in practice what you learned by building several applications with an incremental level of complexity.

The functional architecture implies getting the initial state, showing it to the user using the view functions, listening for actions, updating the state based on those actions, and rendering the updated state back to the user again.



The update functions are the reducers, the view functions are the functional components.



Enjoy the book!

Table of Contents