CASS SDG Training

Release 1.0.0

CASS

May 09, 2018

Git Training

CHAPTER 1

Git

1.1 Training

This repository is designed to walk you through some of the more common Git operations you will need to know and use as a developer. Repository: https://github.com/osu-cass/GitTraining

1.2 Getting started

Clone this repository to your local machine. You may need to install Git if it isn't already installed. If you aren't sure if you have Git or not, simply run `git --help` from the command line. It will return a helpful message with some common commands if it is installed.

1.2.1 Git Basics

- What is source control?
- Learn You a Git
- Github Interactive Tutorial

1.2.2 Configure who you are

- Set your user name with `git config --global user.name "your name"`
- Set your email with `git config --global user.email "you@email.domain"`

1.2.3 Useful references

If you have questions or get stuck the following resources may help you.

- The official Git documentation
- Google and StackOverflow
- GitHub and Atlassian have some well written general Git documentation
- Your coworkers

1.3 Section 1: Tracking Changes

Topics: Checkout, commit, revert, merge, log, move, and remove.

Files for the section can be found in the `section1` directory.

1.3.1 Committing Changes

- Checkout the `dev` branch.
- Make a new branch and add your self to the list of people in `people.md`.
- Commit your changes.
- You will be asked to make a commit message. This will open a text editor. Save and quit the text editor to complete the commit. **Note:** Git uses the system default editor. On vanilla Git Bash in Windows the default text editor is Vim.
- If you need to edit the commit message for your last commit you can use the `git commit --amend` command.

1.3.2 Undoing Changes

Uncommitted Changes

- Make a change to any of the files in the repository.
- Run `git status` to see what files were changed.
- Stage the file using `git add`, but do not commit.
- Unstage the file with `git reset`.
- Use git to undo the changes you made and reset the file to the state it was in when it was committed. `git status` should report no modified files.

Committed Changes

Tony made 3 commits to the dev branch. He misunderstood the project requirements and the changes introduced with his two last commits need to be removed.

- Checkout the dev branch.
- Determine what changed between his first and last commit, then revert the two most recent of his commits. Git has diff and log tools which will help.
- There are a few ways to undo commits. It is best practice to preserve history whenever possible. The `git revert` command which will keep the commits in the repository history, but remove the changes introduced by the commit.

- **TIP:** If you are reverting multiple commits that modify one file it is easiest to start at the most recent commit you want to revert and work backwards.
- You will need to make a commit when you revert the changes.

Rewriting History

Sometimes you may want to undo a commit you have made or even erase it from existence.

WARNING: This is generally considered bad practice. Only do this for commits you have not pushed to a remote repository (more on those later).

- Make a change to a local file.
- Add and commit the file.

Undoing a commit:

- run `git reset HEAD~1`
- This will put you back 1 commit behind the latest but preserve the changes that were committed with it. Your changes will still be there and you can see the modified files with `git status`.

Removing a commit from existence:

- run `git reset --hard HEAD~1`
- This will put you back 1 commit behind the latest and nuke the previous from existence.

1.3.3 Moving, removing, and ignoring files

Someone checked in a temp file generated by their text editor. Stop git from tracking this file, remove it, and update the `.gitignore` file to prevent `.tmp` files from being tracked in the future.

Someone misnamed the `rename_me.md` file. Git has a command to move or rename a file while retaining its history. Use this to rename the file to `newname.md`.

1.4 Section 2: Working with multiple branches, and merging the work of others

Topics: Stash, diff, merge, merge conflicts

Files for this section can be found in the `section2` directory.

1.4.1 Merging

Sometimes two people will make changes to the same file on separate branches. When these branches are merged it can cause a merge conflict. As the developer performing the merge it is your job to decide what changes to keep.

You have been tasked with updating the installation instructions for your product.

- Checkout your branch again.

- Open the `installation.md` file and add install instructions (It doesn't matter what they actually say).
- Commit your changes.
- Andrew tells you he has been working on the installation instructions, and updated the uninstall section.
- Merge Andrew's branch into yours and resolve the conflict so the file contains both of your changes.
- After you have finished your changes your manager wants to see them in the dev branch. Merge your branch into dev. This time there should not be any conflicts.

1.4.2 Stashing changes

Git has a function to stash local changes without committing them.

You are working on your local branch, when your coworker Taylor comes over and asks you to try running their code. You are not yet ready to commit your code, but you want to look at Taylor's branch without losing your progress.

- Make some changes to the `stash_me.md` file.
- Use the `git stash` command to stash your work.
- Checkout Taylor's branch.
- Checkout your branch again.
- Use `git stash pop` to get your work back.
- You can discard these changes or commit them.

1.5 Section 3: Remotes and sharing your changes

Topics: Remote repositories, syncing changes, and forking a repository.

At the beginning you cloned this repository from GitHub. You have all of your changes stored locally, now you need to share them. Git uses the concept of remotes to track where you cloned a repository from. you may also hear these referred to as the upstream repository. Right now the remote for your repository is set to the repository you cloned from.

To send you changes to a remote repository so others can view and use them you need to push them using the `git push` command. You can see if any changes have been made to the remote repository by running `git fetch`. To pull changes for the current branch into your local copy of the repository use the `git pull` command. It is good practice to check for changes before pushing to a remote.

If you try to run `git push` right now you will get a message that you do not have permission to push to this repository. Most remote repositories have security in place to prevent just anyone from pushing their changes. You will make another remote copy of the repository that you can push your changes to.

Visit the repository on GitHub and fork it so you have a copy under your user account. Now update the remote for the copy of the repository you changed, and push your changes to the `dev` branch to your fork on GitHub. If a branch does not exist in the remote repository git will inform you. Try pushing your branch. What is the message? What do you need to do to push your new branch?

CHAPTER 2

Javascript

2.1 React Training

Complete the Tic Tac Toe tutorial here.

2.2 React/TypeScript Training

This document assumes you have basic familiarity with HTML, CSS and JavaScript.

Make sure Visual Studio, node.js, and the latest version of TypeScript for your Visual Studio version are installed.

2.2.1 Links

Typescript Deep Dive

2.2.2 User interfaces on the web

Since the early days of JavaScript, we've manipulated web pages through the Document Object Model (DOM). We would call functions like `document.getElementById` and `document.createElement` to do stuff like add table rows or change text content or alter the element's style. Various libraries like jQuery are used on top of these functions, but the basic idea is still the same: run a function that reaches in and changes the page in response to what the user did.

It's difficult to make this scale as the complexity of a web app grows. Actions in one area of the page frequently cause changes in other parts of the page. Consistency needs to be maintained. Sometimes whole sections of the document need to be replaced with new content that is completely different in structure.

React is a library for user interfaces from Facebook that allows for a simpler approach. It works by having you come up with a piece of data that describes the state of the user interface at a moment in time. Then you write a function called `render` which uses that piece of data to create the entire user interface, seemingly from scratch. To make changes in the user interface, you come up with a new piece of data and give it to React to re-render the UI with.

2.2.3 On the maintainability of JavaScript

JavaScript is what makes modern web apps possible. It makes it so the page can respond to user interaction by changing the content of parts of the page on the fly instead of making the web server generate an entirely new page and deliver it to the user's browser to be displayed. User interfaces made using JavaScript can be very fluid.

JavaScript has some problems that can make developing apps in it painful. Quirks like the `this` keyword in functions, unexpected behavior related to loose comparison operators `==` `!=`, and confusion around variable scope. The lack of static type checking can make it so remembering what properties are on objects, etc, and what types they actually are is hard, and your editor has limited ability to help you get it right.

TypeScript has arisen as an answer to many of these difficulties. At its heart, it's just JavaScript plus type annotations. A valid piece of JavaScript is generally a valid piece of TypeScript. The output from the compiler mainly just strips the type annotations away to make it JavaScript again.

Using TypeScript, we can develop and modify our apps more productively: when we make a change, we find out sooner if we've broken something somewhere else, and it's easy to tell what functions and variables are available to use in a given context.

2.2.4 Setting up your React/TypeScript project

We're going to create a simple to-do list app using React. Begin by opening Visual Studio and creating a new project. Search for "TypeScript" in the **Search Installed Templates** box and select the **HTML Application with TypeScript** project type. Name it something like ToDoList and create the project.

We want to modify this project so we can serve the TypeScript files we create to the browser! Add the following in your web.config file:

```xml
<configuration>
  <system.webServer>
    <staticContent>
      <remove fileExtension=".tsx" />
      <mimeMap fileExtension=".tsx" mimeType="application/javascript" />
    </staticContent>
  </system.webServer>
  <system.web>
    <compilation debug="true" targetFramework="4.5" />
    <httpRuntime targetFramework="4.5" />
  </system.web>
</configuration>
```

If you open `app.ts` in the Solution Explorer, you'll find a Greeter component that shows off some of the basic language features. If you hit the green play button, the app should start up and open a page in your browser with a ticking timestamp. Hit Stop when you're done looking at it.

NPM configuration

We're going to use npm (Node Package Manager) to add JavaScript libraries and TypeScript definitions files to the project. Definitions files allow us to have type checking and autocomplete on things provided to us by libraries like React.

1. Right click on the project in the Solution Explorer. Expand the **Add** menu and select **New Item**.

2. Type "npm" into the search bar and select **npm Configuration File**. Use the default name (package.json) and create the file.
3. Right click on your project file again and choose Open Folder in File Explorer.
4. Shift+right click in the background of the File Explorer window and choose **Open command window here**.
5. Run the following commands:

```
npm install --save react react-dom
```

```
npm install --save-dev @types/react @types/react-dom
```

Now we have the necessary libraries and type definitions installed, and they're saved as dependencies in `package.json`. If someone grabs this project in the future, they can run just `npm install` and fetch the dependencies specified in `package.json`.

TypeScript compiler configuration

We'll also need to add a `tsconfig.json` file to the project to configure the TypeScript compiler.

1. Go to the **Add->New Item** menu in the Solution Explorer, as you did for `package.json`.
2. Search for "TypeScript". Select TypeScript JSON Configuration File and create it with the name `tsconfig.json`.
3. Open the file you created and edit it so it looks like this:

```
{
  "compilerOptions": {
    "noImplicitAny": true,
    "strictNullChecks": true,
    "noEmitOnError": false,
    "removeComments": false,
    "sourceMap": true,
    "target": "es5",
    "jsx": "react"
  },
  "exclude": [
    "node_modules",
    "wwwroot"
  ],
  "compileOnSave": true
}
```

2.2.5 Your first React element

Now, let's try and use React in this app. Start by renaming `app.ts` to `app.tsx`. This allows us to use a language extension called JSX, which makes writing React code simpler.

Delete the Greeter class at the beginning of the file, and replace the content of the `window.onload` function with something like this:

```
window.onload = () => {
    const el = document.getElementById('content');
    const jsx = <div>Hello, world!</div>;
    ReactDOM.render(jsx, el);
};
```

Note the `<div>` inline in the middle of the TypeScript code. That's JSX, and we will be writing more of it.

Now let's tweak `index.html` to reference the scripts we need and try running it. Edit the content of `<head>` so it looks like this:

```html
<head>
    <meta charset="utf-8" />
    <title>TypeScript HTML App</title>
    <link rel="stylesheet" href="app.css" type="text/css" />
    <script src="node_modules/react/dist/react.js"></script>
    <script src="node_modules/react-dom/dist/react-dom.js"></script>
    <script src="app.js"></script>
</head>
```

Now run the app. If you see Hello, World! in your browser window, you succeeded! If not, you can try opening the console in Developer Tools with F12 to see what went wrong, or go over the instructions and compare your code to the examples to make sure everything was done right. In Chrome, you can also try opening the Developer Tools with F12, then right clicking the refresh button and selecting "Empty Cache and Hard Reload".

2.2.6 React Components

User interfaces in React are divided up into classes called Components. While it's possible to just ask React to render a straight JSX element, to create interfaces with dynamic behavior, it's a good idea to create components.

The most basic thing that all components need is a `render` function. All this function needs to do is return JSX. Add something like this to `app.tsx`:

```
class ToDoList extends React.Component<void, void> {
    render() {
        return (
            <div>It seems there is still much to do.</div>
        );
    }
}
```

Then, the usage site of the Component looks a little different:

```
window.onload = () => {
    const el = document.getElementById('content');
    const jsx = <ToDoList />;
    ReactDOM.render(jsx, el);
};
```

In essence, a Component is like a custom HTML element you've made up that can have complex behavior.

You probably noticed those two generic type arguments up at `React.Component`. Those are the two types of data the component will receive: the props and the state.

Using Props

Props is a piece of data received by a component which is used in `render`. You specify what the Props contain. Here's a simple example:

```
interface Props {
    name: string;
}
```

```
class ToDoList extends React.Component<Props, void> {
    render() {
        return (
            <div>
                It seems there is still much to do, {this.props.name}.
            </div>
        );
    }
}
```

Note the parenthesis by the `return` statement. These allow us to use a more readable indentation style. Omitting the parens and putting the JSX on the next line would cause the function to return `undefined`, which isn't what we want.

If you edit `app.tsx` to contain this, you'll see there is a red line under `<ToDoList />` at the usage site. TypeScript is letting us know that we've failed to provide a prop that we've said is part of the Props type. This kind of checking is something you'll be thankful for. Edit `window.onload` to fix it:

```
window.onload = () => {
    const el = document.getElementById('content');
    const jsx = <ToDoList name="CASS Student" />;
    ReactDOM.render(jsx, el);
};
```

As you can see, specifying props looks similar to specifying attributes like `<a href="google.com"></a>` on HTML elements.

When you run the program in your browser, you should now see the message "It seems there is still much to do, CASS Student."

Using State

Props are distinguished by the fact that they're passed in by whoever is creating a particular component. The component receiving the props has direct ability to modify those props.

State is managed by the component itself, and the component is able to modify its state. In our basic example, we're going to use state to track a list of to-do items that we can later add to dynamically.

```
interface Props {
    name: string;
}

interface State {
    items: string[];
}

class ToDoList extends React.Component<Props, State> {
    constructor(props: Props) {
        super(props);
        this.state = {
            items: ["Milk", "Eggs", "Bread"]
        };
    }

    render() {
        const itemsJSX = this.state.items.map(item => <li>{item}</li>);
        return (
            <div>
                <h3>{this.props.name}'s To-Do List</h3>
                <ul>
                    {itemsJSX}
                </ul>
            </div>
        );
    }
}
```

A few things to note in this code example:

- Adding state requires we implement a constructor to assign an initial state. Outside of the constructor, we **do not** assign to `this.state`, but instead use the `this.setState` function.
- Arrays are converted to JSX using `map`. `map` receives a function which converts an element from the array into some new value–in this case, an `<li>` element, and returns a new array with the result of the function applied to each element of the original array.
- JSX expressions are required to have a single root element– thus, we enclose the `<h3>` and `<ul>` in a `<div>`.

2.2.7 Putting it all together

Now, we'll add UI to add a new to-do item to the list. The approach we will take is roughly as follows:

1. Add an `<input>` element to type in the new to-do item's name.
2. Add a `<button>` to submit the new to-do item.
3. Call `setState` in this button's click handler with a new list of items.

First, we'll add the input element along with the necessary state to manage its content.

1. Add a property called `newItemName` of type `string` to `State`.
2. Add it to the initial state object in `constructor` and give it an initial value of `""` (the empty string).
3. Add the input element to `render`, specifying its `value` as `this.state.newItemName`.

If you run the app and try to type in the input, you'll notice that it stays blank. That's because we need to update its state in response to the user's keystrokes.

Props and state are immutable

Immutability in this case means that you don't change an object after you give it to React. That means you **never assign anything** to a member of `this.props` or `this.state`, and you don't assign to other variables referencing that same object, e.g.:

```
const props = this.props;
props.something = 42;
```

Instead, to change the state of our React app, we'll give it an entirely new state value.

Handling events

We will attach an `onChange` handler to the input to do this. Add a new function to `ToDoList` that looks like this:

```
// inside class ToDoList:
onInputChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    const newState = {
        ...this.state,
        newItemName: e.currentTarget.value
    };
    this.setState(newState);
}
```

Now we'll update the JSX to reference this newly defined function.

```
// inside ToDoList's render function:
<div>
    ... // the content that was already in the div
    <input value={this.state.newItemName} onChange={this.onInputChange} />
</div>
```

Key notes:

- We're defining `onInputChange` using an arrow function. This makes it so when we pass it off as a variable in JSX, `this` is guaranteed to be the instance of ToDoList that we're intending to use, instead of being determined by the caller. If we defined it the way we defined `render`, then `this` might point to something else that doesn't have a `setState` function on it. Check out this posting for more details.
- The type annotation on the change event `e` was determined using Go to Definition (F12) on the `onChange` attribute on the input.
- We're using the spread operator `...` to copy all the properties from `this.state` into a new object. This is syntactic sugar for `Object.assign()`.
- By specifying `newItemName` after `...this.state`, we're making it so `e.currentTarget.value` is used for the `newItemName` on the new state instead of the old `this.state.newItemName`.

If you go into your browser now, you can type and the contents of the input will update.

Adding an item

Now we just need to add a button to add the item name the user typed to the list of to-do items.

```
// inside class ToDoList
addItem = () => {
    const newItems = [...this.state.items, this.state.newItemName];
    const newState = {
        items: newItems,
        newItemName: ""
    };
    this.setState(newState);
};
```

This handler will make a new item with what the user typed into the input and clear the contents of the text field.

- The `...` spread operator is used here on the `this.state.items` array to create a new array consisting of items from the original array with a single new item at the end.

Now when you run the app, you should see a to-do list that you can add items to.

2.2.8 Looking forward

Now that you've learned some of the basic practices that go into making a React component, try adding the ability to check and uncheck individual items to mark them as completed. Here are some tips that should help:

- The items array should change from being a `string[]` to being some interface type you define that has `name: string` and `completed: boolean` attributes.
- Render each item slightly differently based on whether it's completed, e.g. add a ✓ character to its name when rendering, or a custom `className` attribute that changes the background color based on CSS.
- Come up with a function that toggles the item's completed state at a certain row and attach it to each row in the list. It might look something like this:

```
toggleCompleted = (index: number) => {
    const newItems = this.state.items.slice(); // Make a copy of the array
    // Make a copy of the item at `index` in the array
    // Modify the copied item to toggle the completed state and put it in the new␣
↪array at `index`
    // Make a new state object with the new array on it and `setState`
}
```

- Attach the handler when creating `<li>` elements using the optional second parameter to the `map` callback function:

```
this.state.items.map((item, i) => <li onClick={() => this.toggleCompleted(i)}>/* name␣
↪and check */</li>)
```

2.2.9 Growing complexity

As these functions for rendering subcomponents based on parameters change, it's a good idea to start factoring them out to separate functions like `renderTodoItem` or to create a new Component subclass like `ToDoItem`. If you choose the latter, you will want to pass the `onClick` handler in as a prop, because you won't have access to `toggleCompleted` in the other class.

A parent component's state is a child component's prop

One of the limitations of state is that it isn't visible to parent components. Consider the case of a component hierarchy with a ToDoList that contains ToDoItems. The completed attribute only affects the rendering of the particular ToDoItem in question. This makes it seem reasonable to keep it as state on the ToDoItem, but there are major drawbacks to that. The parent would have no way of knowing about the completed state of the ToDoItem. If you went to do a save of the user's to-do list at any point, you'd struggle to actually obtain what that latest state is.

Therefore, you should try to keep state in top-level components, or even above top-level components. The parent component contains the state, which will be accessible when you go to call "save" or "cancel" or whatever you want to do. The child component will just receive values inside the state as props– it won't have the ability to modify those props, but it may also receive event handler functions that cause the parent's state to be updated, and thus for the child to be re-rendered with new props.

State management

To end this training, we'll discuss a few commonly used techniques for state management, and mention a pattern we've already used in a few places at CASS.

Since we often try to limit the number of external libraries we pull into projects, we solve state management problems by just making a parent "controller" class which contains the top-level component of our React app. This has a few strengths:

1. It allows the top-level component to receive an event handler that makes it re-render with new props.
2. It allows for structured "interop" between React and non-React parts of the page. The fact that React doesn't demand to be in charge of the entire page is one of its biggest strengths.

The general structure of a controller looks like this:

```
class MyController {
    constructor(private props: SomeDataModel, private rootElement: HTMLDivElement) {}

    onSave = (newValue: string) => {
        ...
        this.props = ... // make a copy somehow using newValue
        this.render();
    }

    render() {
        ReactDOM.render(<MyComponent {...props} onSave={this.onSave} />, this.
↪rootElement);
    }
}

// This is a replacement for the window.onload handler function.
// In your page template or top-level script:
addEventListener("load", () => {
    const controller = new MyController(data, document.getElementById("root"));
    controller.render();
});
```

- Note that TypeScript has syntax sugar for instance variables: if you just throw a visiblity modifier (private, public...) onto a constructor parameter, it automatically gets assigned to the class instance.
- `SomeDataModel` is a type which has the data used to populate some component, but not its event handlers. `MyComponentProps` then is the union of `SomeDataModel` and a type which gives signatures for the event handlers (e.g `onSave`) on `MyComponent`.
- The `data` argument comes from either your template serializing a model to JSON or is given in a callback to some API method you have for populating the view.

To see another, more commonly used approach to state management, check out the Redux library.

2.3 Best Practices

A collection of best practice guidelines for ReactJS. Prepared by Rob Caldecott

2.3.1 Contents

2.3.2 JavaScript

A collection of JavaScript tips.

Stop using `var`

Use `const` and `let` instead. They have proper block scoping, unlike `vars` which are hoisted to the top of the function.

```
let name = "John Doe";
name = "Someone else";
...
const name = "John Doe";
// This will fail
name = "Someone else";
```

Use object shorthand notation

```
state = { name: "" };

onChangeName = e => {
  const name = e.target.value;
  this.setState({ name });
  // this.setState({ name: name });
};
```

Use string templates

```
const name = "John Doe";
const greeting = `Hello ${name}, how are you?`;
```

2.3.3 ESNext

You can take advantage of some next-generation JavaScript syntax right now, including:

- Async/await (ES7, ratified in June 2017)
  - Useful when using promises and `window.fetch`
- `Object` rest/spread (stage 3 proposal)
  - Destructure and make shallow copies of objects
- Class fields and `static` properties (stage 2 proposal)
  - Initialise component state
  - Auto-bound event handlers

async/await

Simplify your promise handling.

Before

```
const fetchIp = () => {
  window
    .fetch("https://api.ipify.org?format=json")
    .then(response => response.json())
    .then(({ ip }) => {
      // We're done, here in this handler
      this.setState({ ip });
    })
    .catch(({ message }) => {
      // Special catch handler syntax
      this.setState({ error: message })
    });
};
```

After

```
const fetchIp = async () => {
  try {
    const response = await window.fetch("https://api.ipify.org?format=json");
    const { ip } = await response.json();
    // We're done: the code looks synchronous
    this.setState({ ip });
  } catch ({ message }) {
    // Standard try/catch
    this.setState({ error: message });
  }
};
```

`Object` spread/rest

Use this to pull out useful properties from an object or make a shallow copy.

```
const { text, show } = this.props;
const { text, ...other } = this.props;
const copy = { ...data, additional: "value" };
```

Class fields and static properties

Make your React classes more readable:

```
class MyComponent extends React.Component {
  state = { name: "" };

  onChangeName = e => {
    this.setState({ name: e.target.value });
  }
}
```

Component class skeleton

A typical component class using ESNext syntax looks like this:

```
class MyComponent extends React.Component {
  static propTypes = { ... };

  static defaultProps = { ... };

  state = { ... };

  onEvent = () => { ... };

  classMethod() { ... }
}
```

2.3.4 prop-types

Always declare your props. Simply install the `prop-types` module from npm:

```
npm install --save prop-types
```

And then import the module into your component:

```
import PropTypes from "prop-types";
```

Using prop types ensures:

- Consumers of your component can see exactly what props are supported.
- Console warning are displayed when the wrong prop type is used.
- Props can be documented for use with `react-styleguidist`.

Example (class using ESNext static property support)

The following component supports two props: `onClick` which is a function and `name` which is a string and is a mandatory prop.

```
import React from "react";
import PropTypes from "prop-types";

export default class Greeting extends React.Component {
  static propTypes = {
    onClick: PropTypes.func,
    name: PropTypes.string.isRequired
  };

  render() {
    return (
      <h1 onClick={this.props.onClick}>
        Hello, {this.props.name}
      </h1>
    );
  }
}
```

Example (stateless functional component)

When using a stateless functional component you need to declare prop types on the function object itself:

```
import React from "react";
import PropTypes from "prop-types";

const Greeting = props =>
  <h1 onClick={props.onClick}>
    Hello,{props.name}
  </h1>;

Greeting.propTypes = {
  onClick: PropTypes.func,
  name: PropTypes.string.isRequired
};

export default Greeting;
```

Use destructuring to import specific prop types

You can also use destructuring to import just the prop types you need. This can save typing, especially when using props of the same type.

For example, here is the above stateless functional component example rewritten:

```
import React from "react";
import { func, string } from "prop-types";

const Greeting = props =>
  <h1 onClick={props.onClick}>
    Hello,{props.name}
  </h1>;
```

```
Greeting.propTypes = {
  onClick: func,
  name: string.isRequired
};

export default Greeting;
```

Using object shapes and arrays

You can also specify the *shape* of an object prop or the shape of arrays.

For example, the following component expects an array of objects. Each object requires `id` and `name` string properties.

```
import React from "react";
import { arrayOf, shape, string } from "prop-types";

const Stores = ({ stores }) =>
  <ul>
    {stores.map(store =>
      <li key={store.id}>
        {store.name}
      </li>
    )}
  </ul>;

Stores.propTypes = {
  stores: arrayOf(
    shape({
      id: string.isRequired,
      name: string.isRequired
    })
  )
};

export default Stores;
```

Custom prop types: sharing your shapes

You can easily share custom prop types by adding them to a file and exporting them for use in your project. For example:

```
// customProps.js
import { string, shape } from "prop-types";

export const store = shape({
  id: string.isRequired,
  name: string.isRequired
});

// Stores.js
import React from "react";
import { arrayOf } from "prop-types";
import { store } from "./customProps.js";
```

```
const Stores = ({ stores }) =>
  <ul>
    {stores.map(store =>
      <li key={store.id}>
        {store.name}
      </li>
    )}
  </ul>;

Stores.propTypes = {
  stores: arrayOf(stores).isRequired
};

export default Stores;
```

Specifying default prop values

You can also declare default values for props by declaring the `defaultProps` object on the component class or function. For example:

```
import React from "react";
import PropTypes from "prop-types";

export default class Heading extends React.Component {
  static propTypes = {
    backgroundColor: PropTypes.string,
    color: PropTypes.string,
    children: PropTypes.node.isRequired
  };

  static defaultProps = {
    backgroundColor: "black",
    color: "white"
  };

  render() {
    const style = {
      backgroundColor: this.props.backgroundColor,
      color: this.props.color
    };

    return (
      <h1 style={style}>
        {this.props.children}
      </h1>
    );
  }
}
```

This can be especially useful for `func` props as it stops a potential crash if an optional function prop is not supplied. For example:

```
import React from "react";
import { func } from "prop-types";
```

```
const Button = ({ onClick }) =>
  <div className="btn" onClick={onClick}>
    Button
  </div>;

Button.propTypes = {
  onClick: func
};

Button.defaultProps = {
  onClick: () => {}
};

export default Button;
```

Using default prop values in stateless functional components

Rather than declaring `defaultProps` for stateless functional components, you can use a combination of **destructuring** and **default parameter values** instead. For example:

```
import React from "react";
import { string, node } from "prop-types";

const Heading = ({ children, backgroundColor = "black", color = "white" }) => {
  const style = {
    backgroundColor,
    color
  };

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

Heading.propTypes = {
  backgroundColor: string,
  color: string,
  children: node.isRequired
};

export default Heading;
```

2.3.5 Stateless Functional Components

Stateless functional components are React components as JavaScript functions. They can be used for components that do not use any lifecycle methods other than render and do not use any state.

- Concerned with *how things look*.
- AKA as *presentational* or *dumb* components.
- Functional programming paradigm: **stateless function components are pure functions of their props**.
- Props passed as the first function parameter.

- Simply return the component JSX: the same as the class `render` method.
- No state, no lifecycle methods.
- Easy to test.
- Easy to re-use/share.
- Make no assumptions about application state or the data source.
- Can be combined with container components (which may have state and may know about the data source).

Example

A simple greeting component: it displays a name and calls a prop when clicked.

Note that ES6 arrow functions are preferred.

```
import React from "react";
import { func, string } from "prop-types";

const Greeting = ({ onClick, name }) =>
  <h1 onClick={onClick}>
    Hello, {name}
  </h1>;

Greeting.propTypes = {
  onClick: func,
  name: string.isRequired
};

export default Greeting;
```

Simple snapshot testing

You can quickly test a simple component like this using **snapshot testing**. For example:

```
import React from "react";
import renderer from "react-test-renderer";
import Greeting from "../Greeting";

it("renders", () => {
  expect(renderer.create(<Greeting name="The name" />)).toMatchSnapshot();
});
```

2.3.6 Containers

Containers are combinations of *state* and *presentational components*.

- Concerned with *how things work*.
- Usually ES6 class components with state.
- Render Re-usable stateless functional components.
- Knowledge about the data source and/or the application state.
- Commonly used with `react-redux`.

- Often generated using *higher order components* (HOCs).

Example

Here is a simple presentational component that renders a styled IP address:

```
import React from "react";
import { string } from "prop-types";

const IPAddress = ({ ip }) => {
  const styles = {
    container: {
      textAlign: "center"
    },
    ip: {
      fontSize: "20px",
      fontWeight: "bold"
    }
  };
  return (
    <div style={styles.container}>
      <div style={styles.ip}>
        {ip}
      </div>
    </div>
  );
};

IPAddress.propTypes = {
  ip: string
};

export default IPAddress;
```

And here is an example container for this component. The container knows about the data source (in this case how to fetch the current IP.)

Notice the following characteristics:

- It is an ES6 class.
- It has state.
- It is using component lifecycle (`componentDidMount`).
- It is acting as a wrapper for the `IPAddress` component.

```
import React from "react";
import IPAddress from "./IPAddress";

export default class IPAddressContainer extends React.Component {
  state = { ip: "" };

  componentDidMount() {
    window
      .fetch("https://api.ipify.org?format=json")
      .then(response => response.json())
      .then(response => {
        this.setState({ ip: response.ip });
      });
  }

  render() {
    return <IPAddress ip={this.state.ip} />;
  }
}
```

2.3.7 Higher Order Components

Higher Order Components (or HOCs) are used to transform a component into another component.

- A HOC **is a function that takes a component and returns a new component**.
- Made possible due to the compositional nature of React.
- Often used to inject additional props into an existing component.
- Useful for creating containers.
- A popular example is the `react-redux` `connect` function.
- Can be used to re-use code, hijack the `render` method and to manipulate existing props.

Example: IP address

The following HOC function will fetch the IP address and inject a prop called `ip` into **any** component. This is an example of using a `class` as the container.

```
import React from "react";

const withIPAddress = Component => {
  return class extends React.Component {
    state = { ip: "" };

    componentDidMount() {
      window
        .fetch("https://api.ipify.org?format=json")
        .then(response => response.json())
        .then(response => {
          this.setState({ ip: response.ip });
        });
    }

    render() {
      return <Component ip={this.state.ip} {...this.props} />;
    }
  };
};

export default withIPAddress;
```

Notice what's happening here: we are exporting a function that accepts a component as a parameter and returns an ES6 class.

To use this with an existing component we do something like this:

```
import React from "react";
import { string } from "prop-types";
import withIPAddress from "./withIPAddress";

const SimpleIPAddress = ({ ip, color = "black" }) =>
  <p style={{ color }}>
    {ip}
  </p>;

SimpleIPAddress.propTypes = {
  ip: string
};

export default withIPAddress(SimpleIPAddress);
```

We export the result of calling `withIPAddress`, passing in the component in which we want the `ip` prop injected.

Example: language

Here's another example of a HOC that injects the current browser language setting into any component as a prop called `language`. In this case we are using a stateless functional component as the container.

```
// withLanguage.js
import React from 'react';

const withLanguage = Component => props =>
  <Component {...props} language={navigator.language} />;

export default withLanguage;

// MyComponent.js
import React from "react";
import { string } from "prop-types";
import withLanguage from "./withLanguage";

const MyComponent = ({ language }) =>
  <div>
    Browser language: {language}
  </div>;

MyComponent.propTypes = {
  language: string
};

export default withLanguage(MyComponent);
```

2.3.8 Chaining HOCs

Note that you can also chain HOCs together to create a new component that combines them all. For example:

```
import React from "react";
import { string } from "prop-types";
import withLanguage from "./withLanguage";
import withIPAddress from "./withIPAddress";
```

```
const MyComponent = ({ language, ip }) =>
  <div>
    <div>
      Browser language: {language}
    </div>
    <div>
      IP address: {ip}
    </div>
  </div>;

MyComponent.propTypes = {
  language: string,
  ip: string
};

export default withLanguage(withIPAddress(MyComponent));
```

2.3.9 Functions as Children

An alternative pattern to HOCs is **functions as children** where you supply a function to call as the child of a container component: this is the equivalent of a **render callback**. Like HOCs you are decoupling your parent and child and it usually follows a similar pattern of a parent that has state you want to hide from the child.

This has some advantages over traditional HOCs:

- It does not pollute the `props` namespace. HOCs have an implicit contract they impose on the inner components which can cause prop name collisions especially when combining them with other HOCs.
- You do not need to use a function to create the container: you use simple composition instead.
- Developers do not need to call an HOC function to create a new wrapped component which can simplify the code: they simply export their child components as normal.

In order for this to work you need to use a function as the special `children` prop and have the outer container component call this function when rendering.

For example, here is a component that exposes the browser language:

```
import React from 'react';
import { func } from 'prop-types';

const Language = ({ children }) =>
  <div>
    {children(navigator.language)}
  </div>;

Language.propTypes = {
  children: func,
};

export default Language;
```

The component simply treats the `children` prop as a function and calls it. It can be used like this:

```
<Language>
  {language =>
```

```
    <p>
      Browser language: {language}
    </p>}
</Language>
```

The child node of `<Language>` is a function which returns the JSX to render.

Now let's imagine a component that calls an API and uses state to store the status, response and error.

```
import React from 'react';
import { string, func } from 'prop-types';

export default class CallAPI extends React.Component {
  static propTypes = {
    api: string,
    children: func,
  };

  state = {
    isFetching: false,
    data: {},
    error: '',
  };

  async componentDidMount() {
    this.setState({ isFetching: true });
    try {
      const response = await fetch(this.props.api);
      const data = await response.json();
      this.setState({ isFetching: false, data });
    } catch ({ message }) {
      this.setState({ isFetching: false, error: message });
    }
  }

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

The component makes an API call (specified with a prop) and maintains the state of the call. It renders by calling a function and passing through a copy of the state as an object.

It could be used like this:

```
<CallAPI api="https://api.ipify.org?format=json">
  {({ isFetching, data, error }) => {
    if (isFetching) {
      return <p>Loading...</p>;
    }
    if (error) {
      return <p>Error: {error}</p>;
    }
    return <p>Data: {JSON.stringify(data)}</p>;
  }}
</CallAPI>
```

And of course you can render normal components in the callback, for example:

```
<CallAPI api="https://api.ipify.org?format=json">
{
  props => <MyComponent {...props} />
}
</CallAPI>
```

> There is a caveat to using this pattern: they cannot be optimised by React because **they change on every render** (a new function is declared on every render cycle). This rules out using `shouldComponentUpdate` and `React.PureComponent` which may lead to performance issues. Use this pattern wisely.

2.3.10 Events

When using JavaScript DOM and `window` events we usually need `this` to point to our component instance.

Spot the bug in this code:

```
import React from "react";

export default class BindBug extends React.Component {
  state = { toggled: false };

  onClick(e) {
    this.setState({ toggled: !this.state.toggled });
  }

  render() {
    const style = {
      fontSize: "36px",
      color: this.state.toggled ? "white" : "black",
      backgroundColor: this.state.toggled ? "red" : "yellow"
    };

    return (
      <div style={style} onClick={this.onClick}>
        Click me
      </div>
    );
  }
}
```

When you click on the `<div>` the `onClick` function handler is called which then tries to call `this.setState`. But the handler has not bound `this` to the component instance and it ends up as `null` which causes the code to crash.

Binding events

To make this work we need to bind the `onClick` function to `this`. There are two ways to do this:

ESNext property initialize syntax (recommended)

The most readable way to do this is via an ESNext property initializer in conjunction with an arrow function. Arrow functions declared in this way are bound to `this` automatically:

```
import React from "react";

export default class BindClassMethod extends React.Component {
  state = { toggled: false };

  onClick = e => {
    this.setState({ toggled: !this.state.toggled });
  };

  render() {
    const style = {
      fontSize: "36px",
      color: this.state.toggled ? "white" : "black",
      backgroundColor: this.state.toggled ? "red" : "yellow"
    };

    return (
      <div style={style} onClick={this.onClick}>
        Click me
      </div>
    );
  }
}
```

Notice the syntax used here:

```
handlerName = (params) => { ... }
```

This is the best option: it is less code and even though this syntax is experimental it is used widely at Facebook.

Here's another example: a component that uses `window.setInterval` to update a counter every second:

```
import React from "react";

export default class Timer extends React.Component {
  state = { counter: 0 };

  componentDidMount() {
    this.timerId = window.setInterval(this.onTimer, 1000);
  }

  componentWillUnmount() {
    window.clearInterval(this.timerId);
  }

  onTimer = () => {
    this.setState(prevState => ({ counter: prevState.counter + 1 }));
  };

  render() {
    return (
      <p>
        Counter: {this.state.counter}
      </p>
    );
  }
}
```

Note the following:

- The timer ID is stored so it can be cleared when the component unmounts.
- The function version of `setState` is used.

Alternatively you could use an inline arrow function: this will ensure `this` has the correct context:

```
import React from "react";

export default class Timer extends React.Component {
  state = { counter: 0 };

  componentDidMount() {
    this.timerId = window.setInterval(() => {
      this.setState(prevState => ({ counter: prevState.counter + 1 }));
    }, 1000);
  }

  componentWillUnmount() {
    window.clearInterval(this.timerId);
  }

  render() {
    return (
      <p>
        Counter: {this.state.counter}
      </p>
    );
  }
}
```

Constructor binding

Another common way of binding is to add a constructor to your class and use `Function.prototype.bind`:

```
import React from "react";

export default class BindConstructor extends React.Component {
  state = { toggled: false };

  constructor() {
    super();
    this.onClick = this.onClick.bind(this);
  }

  onClick(e) {
    this.setState({ toggled: !this.state.toggled });
  }

  render() {
    const style = {
      fontSize: "36px",
      color: this.state.toggled ? "white" : "black",
      backgroundColor: this.state.toggled ? "red" : "yellow"
    };

    return (
      <div style={style} onClick={this.onClick}>
        Click me
      </div>
    );
  }
}
```

Although this method is not relying on any experimental syntax it suffers from the following issues:

- It requires you adding a constructor.
- You have to remember call `super` in the constructor before doing anything else.
- It is more code.

Sharing event handlers

Sometimes it is useful to share the share event handlers for your components and there is a simple trick to do this using the DOM `name` attribute (which is exposed as a prop for most React components):

```
import React from "react";

export default class DetailsForm extends React.Component {
  state = {
    name: "",
    email: "",
    phone: ""
  }

  onChange = e => {
    this.setState({
      [e.target.name]: e.target.value
    });
  }

  render() {
    return (
      <div>
        <input name="name" value={this.state.name} onChange={this.onChange} />
        <input name="email" value={this.state.email} onChange={this.onChange} />
        <input name="phone" value={this.state.phone} onChange={this.onChange} />
      </div>
    )
  }
}
```

This takes advantage of the JavaScript **computed property name syntax** to update the state. Notice how the `name` prop for each `<input>` matches the corresponding state property: this allows us to share a single `onChange` handler with all three components.

Although this looks like magic **it's just JavaScript**.

Handling the ENTER key in a form

If you want to let users press the ENTER key to submit a form then you will need to prevent the default `submit` behaviour of a HTML form. For example:

```
import React from "react";

export default class LoginForm extends React.Component {
  onSubmit = e => {
    // Don't actually submit!
    e.preventDefault();
    // Enter key was pressed
  };

  render() {
    return (
      <form onSubmit={this.onSubmit}>
        <input
          name="username"
          value={this.state.value}
          onChange={this.onChange}
        />
        <input
          name="password"
          type="password"
          value={this.state.password}
          onChange={this.onChange}
        />
        <input type="submit" value="Login" />
      </form>
    );
  }
}
```

This looks pretty much like a standard HTML form: the presence of the `<input type="submit" />` ensures the ENTER key works but by calling `preventDefault` on the submit event you can handle it yourself without the application reloading.

2.3.11 Conditional Rendering

Sometimes it is useful to render components based on your props or state and there are at least five different mechanisms available to you (remember: it's just JavaScript!)

Note that when you conditionally remove a component it **will be re-mounted when you put it back** which means `componentDidMount` and other lifecycle methods will be called again. So if you are, for example, fetching data when the component mounts, it will be called each time. To avoid this use some form of `show` prop and either return `null` from your `render` or use CSS to hide the content.

Store the JSX in a variable

You can declare a variable to hold the JSX you wish to render. If your condition is not met and an `undefined` variable is rendered, then React will simply ignore it.

```
let message;
if (someCondition) {
  message = <p>Hello, world!</p>;
}

return (
  <div>
    <p>Conditional rendering</p>
    {message}
  </div>
)
```

Ternaries

You can also use a **ternary**. Using `null` or `undefined` is enough to stop anything being rendered:

```
return (
  <div>
    <p>Conditional rendering</p>
    {someCondition ? <p>Hello, world!</p> : null}
  </div>
)
```

Logical && operator shortcut

This relies on the fact the JavaScript will stop evaluating an && condition if the preceding checks return `false`.

```
return (
  <div>
    <p>Conditional rendering</p>
    {someCondition && <p>Hello, world!</p>}
  </div>
)
```

So if `someCondition` is `true` then your JSX is rendered, but if it's `false` then your JSX will simply not be evaluated.

This is a very common method to conditionally render something in React.

Return null from your render method

Another common pattern seen in some 3rd-party component libraries is to conditionally render a component based on a boolean prop. For example, you may have a prop called `show` that determines if the component should display at all: if not then your `render` method can simply return `null`.

> The advantage of this is that the component will not be mounted multiple times each time the `show` prop changes which is useful if you are fetching data, setting timers, etc. in `componentDidMount`.

```
// MyComponent.js
const MyComponent = ({ show }) => {
  if (show) {
    return <p>Hello, world!</p>;
  }
  return null;
};

// SomeOtherComponent.js
...
return (
  <div>
    <p>Conditional rendering</p>
    <MyComponent show={someCondition} />
  </div>
)
```

Hide your component using CSS

A final way is to simply use CSS to hide your component. This also has the advantage of keeping your component mounted.

```
// MyComponent.js
const MyComponent = ({ show }) => {
  const style = {
    display: show ? "block" : "none"
  };

  return <p style={style}>Hello, world!</p>;
};

// SomeOtherComponent.js
...
return (
  <div>
    <p>Conditional rendering</p>
    <MyComponent show={someCondition} />
  </div>
)
```

2.3.12 Arrays

When dealing with arrays of JavaScript objects you can use `Array.prototype.map` to map from array elements to React components. This is a very common pattern.

The following example shows a component that is rendering an array of stores by mapping each array entry to a new `<li>` component.

```
import React from "react";
import { arrayOf, shape, number, string } from "prop-types";

const StoreList = ({ stores }) =>
  <ul>
    {stores.map(store =>
      <li key={store.id}>
```

```
        {store.name}
      </li>
    )}
  </ul>;

StoreList.propTypes = {
  stores: arrayOf(
    shape({
      id: number.isRequired,
      name: string.isRequired
    }).isRequired
  )
};

export default StoreList;
```

Keys

Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity:

```
<li key={store.id}>{store.name}</li>
```

Avoid using array indexes if array items can reorder.

```
stores.map((store, index) => <li key={index}>{store.name}</li>
```

2.3.13 Writing Components

For this section we will use an example of a simple button component but the technique is the same no matter what sort of component you are developing.

Designing a Button component

At first our button is very simple:

```
<Button text="Click me" />

const Button = ({ text }) =>
  <button className="btn">
    {text}
  </button>;
```

More requirements

Now we need support to render an icon:

```
<Button text="Click me!" iconName="paper-plane-o" />

const Button = ({ text, iconName }) =>
```

```
  <button className="btn">
    <i className={"fa fa-" + iconName} />
    {" " + text}
  </button>;
```

Even more requirements!

Now the button needs text formatting, icon positioning and icon size support. **The code is getting complicated**.

```
<Button text="Click me" textStyle="bold" iconName="paper-plane-o" iconPosition="top"␣
↪iconSize="2x" />

const Button = props => {
  const icon =
    props.iconName &&
    <i
      className={classnames("fa fa-" + props.iconName, {
        ["fa-" + props.iconSize]: props.iconSize
      })}
    />;

  return (
    <button className="btn">
      {icon &&
        (props.iconPosition === "top"
          ? <div>
              {icon}
            </div>
          : <span>
              {icon + " "}
            </span>)}
      {props.textStyle === "bold"
        ? <strong>
            {props.text}
          </strong>
        : <span>
            {props.text}
          </span>}
    </button>
  );
};
```

It is clear we cannot continue designing the component in this way for long before it becomes unmanageable.

Composition to the rescue

Instead of using lots of props and a single complicated `render` method split the component into smaller chunks and use composition to render it instead.

```
<Button>
  <FontAwesome
    name="paper-plane-o"
    size="2x"
    block />
  <strong>Click me</strong>
```

```
</Button>

const Button = ({ children }) =>
  <button className="btn">
    {children}
  </button>;

const FontAwesome = ({ name, size, block }) =>
  <i
    className={classnames("fa", "fa-" + name, {
      ["fa-" + size]: size,
      ["center-block"]: block
    })}
  />;
```

This takes advantage of the special `children` prop which is the cornerstone of composition using React.

Summary

You might need composition when:

- There are too many props
- There are props to target a specific part of the component (iconName, iconPosition, iconSize, etc.)
- There are props which are directly copied into the inner markup
- There are props which take a complex model object or an array

2.3.14 Unit Testing

Testing should be simple!

- React components are easy to test.
- Presentational components (stateless functional components) should be treated as pure functions.
- Two common testing patterns are:
  - DOM testing
    - Find DOM nodes, simulate events
  - Snapshot testing
    - Compares files of JSON output
    - Show the diff

Snapshot testing

Snapshot testing is a feature of **Jest** that can be used to test *any* JavaScript object. And thanks to a package called `react-test-renderer` you can convert a React component to an object to use with snapshot testing.

For example:

```
import React from "react";
import renderer from "react-test-renderer";
import IPAddress from "../IPAddress";

it("renders", () => {
  const component = renderer.create(<IPAddress ip="127.0.0.1" />);
  expect(component).toMatchSnapshot();
});
```

When the test runs for the first time a special snapshot file is created in a sub-folder containing the JSON output of the render. The next time you run the test it generates new output and compares it with the snapshot: if there are any differences then the test has failed and you are presented with the object diff. At this point you can decide to regenerate the snapshot.

DOM testing

Alternatively you can render your components into a in-memory DOM (`jsdom`).

- Use `react-dom/test-utils` or `enzyme`
- Find DOM nodes and check attributes
- Simulate events
- Mock event handlers

  Note that this does not work with stateless functional components unless you wrap them with a class (you can use a simple HOC for this.)

For example:

```
import React from "react";
import Greeting from "../Greeting";
import ReactTestUtils from "react-dom/test-utils";

it("renders", () => {
  const onClick = jest.fn();
  const instance = ReactTestUtils.renderIntoDocument(
    <Greeting name="The name" onClick={onClick} />
  );
  const h1 = ReactTestUtils.findRenderedDOMComponentWithTag(instance, "h1");
  ReactTestUtils.Simulate.click(h1);
  expect(onClick).toHaveBeenCalled();
});
```

In this example we use `ReactTestUtils` to render the component, look for the `<h1>` tag and simulate a click event. We then check that our `onClick` prop was called.

Wrapping stateless functional components for `ReactTestUtils`

`ReactTestUtils` does not play well with stateless functional components. To fix this simply wrap your component with a class when testing. You can use an HOC for this in your project:

```
const withClass = Component => {
  return class extends React.Component {
    render() {
      return <Component {...this.props} />;
    }
  };
};

const Component = withClass(Greeting);
const instance = ReactTestUtils.renderIntoDocument(
  <Component name="The name" onClick={onClick} />
);
```

2.3.15 State

State updates may be asynchronous!

React may batch multiple `setState()` calls into a single update for performance.

Because `this.props` and `this.state` may be updated asynchronously, you should not rely on their values for calculating the next state.

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment
});
```

Instead, you can use the **function version of `setState`**.

```
// Correct
this.setState((prevState, props) => ({
  counter: prevState.counter + props.increment
}));
```

Always use this version of `setState` if you need access to the previous state or props.

State update functions can be extracted and tested

Another benefit of using the function version of `setState` is you can extract them from your class, turn them into **thunks** and add tests for them. If you stick to using immutable data for your state then the update functions should be pure which makes them even easier to test. You can even share state update functions amongst your components.

For example:

```
// Stores.js
export const addStore = (id, name) => prevState => ({
  stores: [...prevState.stores, { id, name }]
});

export default class Stores extends React.Component {
  state = { stores: [] };

  onAddStore = () => {
    this.setState(addStore("ID", "NEW STORE NAME"));
  }
  ...
}

// Stores.test.js
```

```
import { addStore } from "./Stores";

it("adds a store to the state", () => {
  const prevState = {
    stores: [
      {
        id: "1",
        name: "Store 1"
      },
      {
        id: "2",
        name: "Store 2"
      }
    ]
  }
  expect(addStore("3", "Store 3")(prevState)).toMatchSnapshot();
});
```

Immutable data

- React tends to favour functional programming paradigms
- Mutable data can often be a source of bugs and unintended side effects
- Using immutable data can simplify testing
- Redux relies on immutable state to work correctly
- You don't necessarily need ImmutableJS: ES6 will usually suffice
- Immutable data can be used alongside `React.PureComponent` for a very simple performance boost

Immutable arrays

Here is an example of using `Array.prototype.map` to clone an array and modify a single element:

```
this.setState(prevState => ({
  items: prevState.items.map(item => {
    if (item.id === idToFind) {
      return { ...item, toggled: !item.toggled };
    }
    return item;
  })
}));
```

You can make a shallow copy of an array and add a new element at the same time using the **array spread operator**:

```
this.setState(prevState => ({
  items: [...prevState.items, { id: "3", name: "New store" }]
}));
```

You can remove elements from an array using `Array.prototype.slice`:

```
this.setState(prevState => ({
  // Remove the first element
  items: prevState.items.slice(0, 1)
}));
```

2.3.16 Props

Destructuring

You can increase code readability be destructuring props. For example:

```
render() {
  const { name, email } = this.props;

  return (
    <div>
      <p>{name}</p>
      <p>{email}</p>
    </div>
  )
}
```

Don't pass on unknown props

If you are wrapping components with another do not pass down any props that the wrapped component does not know about. This will generate a console warning in the browser. For example, this is wrong:

```
const Input = props => {
  const type = props.isNumeric ? "number" : "text";
  // <input> does not know about isNumeric
  // This will generate a console warning
  return <input {...props} type={type} />;
};

Input.propTypes = {
  isNumeric: PropTypes.bool
};
```

To fix this you can use the **object spread operator** to extract the props you care about and add the remaining ones to a single variable. For example:

```
const Input = props => {
  const { isNumeric, ...other } = props;
  const type = isNumeric ? "number" : "text";
  return <input {...other} type={type} />;
};

Input.propTypes = {
  isNumeric: PropTypes.bool
};
```

2.3.17 Pure Components

Premature optimization is the root of all evil.

Most of the time you are probably not going to worry about performance but there are times when you might to avoid potentially costly `render`s and this is where `React.PureComponent` can help.

When React needs to reconcile the virtual DOM it will call your component `render` method and compare it with an in-memory copy. If anything has changed then the real DOM is updated. Usually this is fast but if your `render` function is slow (perhaps it renders many components) then there could be a delay while reconciliation takes place.

However, there is a React lifecycle method you can override called `shouldComponentUpdate` and if you return `false` from this then your `render` method **will not be called**.

To make this easier to manage you can derive your component class from `React.PureComponent` which overrides `shouldComponentUpdate` and performs a simple (and fast) value comparison of your props and state: if there are no changes then the function returns `false` and no render will occur.

So if your `render` method renders exactly the same result given the same props and state then you can use `React.PureComponent` for a potential performance boost.

React performs a *value* comparison of your props and state and **not** a deep object comparison. Therefore you should use immutable data for all props and state to ensure this comparison works as expected: otherwise your component may not render when you expect it to.

For example:

```
import React from "react";
import PropTypes from "prop-types";

export default class MyList extends React.PureComponent {
  static propTypes = {
    items: PropTypes.arrayOf(
      PropTypes.shape({
        id: PropTypes.number,
        text: PropTypes.text
      })
    )
  };

  render() {
    // Only called if the props have changed
    return (
      <List>
        {this.props.items.map(item =>
          <ListItem key={item.id} primaryText={item.text} />
        )}
      </List>
    );
  }
}
```

If the `items` prop changes (is replaced with a new copy of the data) then the component will render.

2.3.18 Project Structure

There are numerous ways to structure your React project. One common layout for components:

- Components are located in `src/components/ComponentName.js`.
- Component-specific CSS is located in `src/components/ComponentName.css`.
- Component tests are located in `src/components/__tests__/ComponentName.test.js`.
- Component stories are located in `src/components/__stories__/ComponentName.stories.js`
- React Styleguidist component examples (if applicable) are located in `src/components/__examples__/ComponentName.md`

If you're using `redux`:

- Code to initialise your store is located in `src/store.js`
- Reducers are located in `src/reducers`
- Action creators are located in `src/actions`
- Selectors are located in `src/selectors`
- Action constants are located in `src/constants/actions.js`

Try and limit the number of files in the root `src` folder but be careful not to overdo your folder structure. There is nothing wrong with lots of files in one folder (Facebook use a monorepo: they have over 30,000 components in a single folder!)

An example layout may look like this:

```
src\
  index.js
  App.js
  setupTests.js
  components\
    __tests__\
      Button.test.js
    __stories__\
      Button.stories.js
    Button.js
    Button.css
  containers\
    __tests__\
      MainPage.test.js
    MainPage.js
  utils\
    __tests__\
      sharedStuff.test.js
    sharedStuff.js
    testUtils.js
```

Another layout involves a separate folder with each component containing the source code, CSS, tests, stories and any other component-specific files. For this to be manageable you need to also add an `index.js` that imports the component and this is not recommended for beginners.

2.3.19 Summary

- It's just JavaScript.
- Use functional programming patterns and techniques where possible.
- Use containers/presentational components.

- Always declare your prop types.
- Take advantage of ES6 and ESNext.
- Use immutable data.
- Use snapshot testing.
- Use the function form of `setState` if you need access to the previous state or props.
- Favour small components and composition when building your UI.
- Don’t ignore console warnings.

CHAPTER 3

.Net Core

3.1 .Net Core Movie Database Application

Complete the tutorial here

3.2 .Net Core People Application

Clone this repository

Refer to this Wiki

It is suggested that you use React and Typescript to complete the front end of this application.