Your First Week With React

Product Manager: Simon Mackie **Cover Designer:** Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

ISBN 978-0-9943469-6-4 (print)

ISBN 978-0-9943470-2-2 (ebook)
Printed and bound in the United States of America

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

Table of Contents

Preface

Preface Goes here

Who Should Read This Book

This book is for ....

Conventions Used

You'll notice that we've used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park. The birds
were singing and the kids were all back at school.</p>
```

If the code is to be found in the book's code archive, the name of the file will appear at the top of the program listing, like this:

0-1. example.css

```
.footer {
background-color: #CCC;
border-top: 1px solid #333;
}
```

If only part of the file is displayed, this is indicated by the word *excerpt*:

0-2. example.css *(excerpt)*

```
.footer {
background-color: #CCC;
border-top: 1px solid #333;
}
```

If additional code is to be inserted into an existing example, the new code will be displayed in bold:

```
 function animate() {
  new_variable = "Hello";
}
```

Where existing code is required for context, rather than repeat all of it, ⋮ will be displayed:

```
function animate() {
    ⋮
new_variable = "Hello";
```

```
}
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An ↪ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/
↪responsive-web-design-real-user-testing/?responsive1");
```

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Chapter

1

for Your Next Project

by Maria Antonietta Perna

Nowadays, users expect sleek, performant web applications that behave more and more like native apps. Techniques have been devised to decrease the waiting time for a website to load on a user's first visit. However, in web applications that expose a lot of interactivity, the time elapsing between a user action taking place and the app's response is also important. Native apps feel snappy, and web apps are expected to behave the same, even on less than ideal internet connections.

A number of modern JavaScript frameworks have sprung up that can be very effective at tackling this problem. React can be safely considered among the most popular and robust JavaScript libraries you can use to create fast, interactive user interfaces for web apps.

In this article, I'm going to talk about what React is good at and what makes it work, which should provide you with some context to help you decide if this library could be a good fit for your next project.

What Is React?

React is a Facebook creation which simply labels itself as being “a JavaScript library for building user interfaces”.

It's an open-source project which, to date, has raked in over 74,000 stars on GitHub.

React is:

- **Declarative**: you only need to design *simple views for each state in your application*, and React will efficiently update and render just the right components when your data changes.
- **Component-based**: you create your React-powered apps by assembling a

number of encapsulated components, each managing its own state.
- **Learn Once, Write Anywhere**: React is not a full-blown framework; it's just a library for rendering views.

How Does the Virtual DOM Work?

The **Virtual DOM** is at the core of what makes React fast at rendering user interface elements and their changes. Let's look closer into its mechanism.

The HTML Document Object Model or DOM is a

> programming interface for HTML and XML documents. ... The DOM provides a representation of the document as a structured group of nodes and objects that have properties and methods. Essentially, it connects web pages to scripts or programming languages. — MDN

Whenever you want to change any part of a web page programmatically, you need to modify the DOM. Depending on the complexity and size of the document, traversing the DOM and updating it could take longer than users might be prepared to accept, especially if you take into account the work browsers need to do when something in the DOM changes. In fact, every time the DOM gets updated, browsers need to recalculate the CSS and carry out layout and repaint operations on the web page.

React enables developers to make changes to the web page without having to deal directly with the DOM. This is done via the **Virtual DOM**.

The Virtual DOM is a lightweight, abstract model of the DOM. React uses the `render` method to create a node tree from React components and updates this tree in response to changes in the data model resulting from actions.

Each time there are changes to the underlying data in a React app, React creates a new Virtual DOM representation of the user interface.

Updating UI Changes with the Virtual DOM

When it comes to updating the browser's DOM, React roughly follows the steps below:

- Whenever something changes, React re-renders the entire UI in a Virtual DOM representation.
- React then calculates the difference between the previous Virtual DOM representation and the new one.
- Finally, React patches up the real DOM with what has actually changed. If nothing has changed, React won't be dealing with the HTML DOM at all.

One would think that such a process, which involves keeping two representations of the Virtual DOM in memory and comparing them, could be slower than dealing directly with the actual DOM. This is where efficient diff algorithms, batching DOM read/write operations, and limiting DOM changes to the bare minimum necessary, make using React and its Virtual DOM a great choice for building performant apps.

Is React Good for Every Project?

As the name itself suggests, React is great at making super reactive user interfaces — that is, UIs that are very quick at responding to events and consequent data changes. This comment about the name *React* made by Jordan Walke, engineer at Facebook, is illuminating:

> This API reacts to any state or property changes, and works with data of any form (as deeply structured as the graph itself) so I think the name is fitting. — Vjeux, "Our First 50,000 Stars"

Although some would argue that all projects need React, I think it's uncontroversial to say that React would be a great fit for web apps where you need to keep a complex, interactive UI in sync with frequent changes in the underlying data model.

React is designed to deal with stateful components in an efficient way (which doesn't mean devs don't need to optimize their code). So projects that would benefit from this capability can be considered good candidates for React.

Chris Coyier outlines the following, interrelated situations when reaching for React makes sense, which I tend to go along with:

- **Lots of state management and DOM manipulation**. That is, enabling and disabling buttons, making links active, changing input values, closing and expanding menus, etc. In this kind of project, React makes managing stateful components faster and easier. As Michael Jackson, co-author of React Router, aptly put it in a Tweet:

  > Point is, React takes care of the hard part of figuring out what changes actually need to happen to the DOM, not me. That's *invaluable*

- **Fighting spaghetti**. Keeping track of complex state by directly modifying the DOM could lead to spaghetti code, at least if extra attention isn't paid to code organization and structure.

Resources

If you're curious about how React and its Virtual DOM work, here's where you can learn more:

- React Videos from Facebook Engineers
- "The Real Benefits of the Virtual DOM in React.js", by Chris Minnick
- "The difference between Virtual DOM and DOM", by Bartosz Krajka
- "React is Slow, React is Fast: Optimizing React Apps in Practice", by François Zaninotto
- "How to Choose the Right Front-end Framework for Your Company", by Chris Lienert

Conclusion

React and other similar JavaScript libraries ease the development of snappy, event-driven user interfaces that are fast at responding to state changes. One underlying goal can be identified in the desire to bridge the gap between web apps and native apps: users expect web apps to feel smooth and seamless like native apps.

This is the direction towards which modern web development is heading. It's not by chance that the latest update of *Create React App*, a project that makes possible the creation of React apps with zero configuration, has shipped with the functionality of creating progressive web apps (PWAs) by default. These are apps that leverage service workers and offline-first caching to minimize latency and make web apps work offline.

Chapter

2

Beginner's Guide

by Michael Wanyoike

In this guide, I'll show you the fundamental concepts of React by taking you through a practical, step-by-step tutorial on how to create a simple Message App using React. I'll assume you have no previous knowledge of React. However, you'll need at least to be familiar with modern JavaScript and NodeJS.

React is a remarkable JavaScript library that's taken the development community by storm. In a nutshell, it's made it easier for developers to build interactive user interfaces for web, mobile and desktop platforms. One of its best features is its freedom from the problematic bugs inherent in MVC frameworks, where inconsistent views is a recurring problem for big projects. Today, thousands of companies worldwide are using React, including big names such as Netflix and AirBnB. React has become immensely popular, such that a number of apps have been ported to React --- including WhatsApp, Instagram and Dropbox.

Prerequisites

As mentioned, you need some experience in the following areas:

- functional JavaScript
- object-oriented JavaScript
- ES6 JavaScript Syntax

On your machine, you'll need:

- a NodeJS environment
- a Yarn setup (optional)

If you'd like to take a look first at the completed project that's been used in this guide, you can access it via GitHub.

What is React?

React is a JavaScript library for building UI components. Unlike more complete frameworks such as Angular or Vue, React deals only with the view layer. Hence, you'll need additional libraries to handle things such as data flow, routing, authentication etc. In this guide, we'll focus on what React can do.

Building a React project involves creating one or more React **components** that can interact with each other. A React component is simply a JavaScript class that requires the `render` function to be declared. The `render` function simply outputs HTML code, which is implemented using either JSX or JavaScript code. A React component may also require additional functions for handling data, actions and lifecyle events.

React components can further be categorized into **containers/stateful components** and **stateless components**. A stateless component's work is simply to display data that it receives from its parent React component. It can also receive events and inputs, which it passes up to its parent to handle. A React container or stateful component does the work of rendering one or more child components. It fetches data from external sources and feeds it to its child components. It also receives inputs and events from them in order to initiate actions.

Understanding the React DOM

Before we get to coding, you need to be aware that React uses a **Virtual DOM** to handle page rendering. If you're familiar with jQuery, you know that it can directly manipulate a web page via the **HTML DOM**. In a lot of use cases, this direct interaction poses little to no problems. However, for certain cases, such as the running of a highly interactive, real-time web application, performance often takes a huge hit.

To counter this, the concept of the Virtual DOM was invented, and is currently being applied by many modern UI frameworks including React. Unlike the HTML

DOM, the Virtual DOM is much easier to manipulate, and is capable of handling numerous operations in milliseconds without affecting page performance. React periodically compares the Virtual DOM and the HTML DOM. It then computes a diff, which it applies to the HTML DOM to make it match the Virtual DOM. This way, React does its best to ensure your application is rendered at a consistent 60 frames per second, meaning that users experience little or no lag.

Enough chitchat! Let's get our hands dirty ...

Start a Blank React Project

As per the prerequisites, I assume you already have a NodeJS environment setup. Let's first install or update npm to the latest version.

```
$ npm i -g npm
```

Next, we're going to install a tool, Create React App, that will allow us to create our first React project:

```
$ npm i -g create-react-app
```

Navigate to your project's root directory and create a new React project using the tool we just installed:

```
$ create-react-app message-app
```

```
…
Success! Created message-app at /home/mike/Projects/
github/message-app
Inside that directory, you can run several commands:

    yarn start
    Starts the development server.

    yarn build
    Bundles the app into static files for production.

    yarn test
    Starts the test runner.

    yarn eject
    Removes this tool and copies build dependencies,
configuration files
    and scripts into the app directory. If you do this,
you can't go back!

We suggest that you begin by typing:

    cd message-app
    yarn start

Happy hacking!
```

Depending on the speed of your internet connection, this might take a while to complete if this is your first time running the `create-react-app` command. A bunch of packages gets installed along the way, which are needed to set up a convenient development environment --- including a web server, compiler and

testing tools.

Navigate to the newly created `message-app` folder and open the `package.json` file.

```
{
    "name": "message-app",
    "version": "0.1.0",
    "private": true,
    "dependencies": {
    "react": "^15.6.1",
    "react-dom": "^15.6.1",
    "react-scripts": "1.0.12"
    },
    "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
    }
}
```

Surprise! You expected to see a list of all those packages listed as dependencies, didn't you? Create React App is an amazing tool that works behind the scenes. It creates a clear separation between your actual code and the development environment. You don't need to manually install Webpack to configure your project. Create React App has already done it for you, using the most common options.

Let's do a quick test run to ensure our new project has no errors:

```
$ yarn start

Starting development server…
Compiled successfully!

You can now view message-app in the browser.

    Local:            http://localhost:3000/
    On Your Network:  http://10.0.2.15:3000/

Note that the development build is not optimized.
To create a production build, use yarn build.
```

If you don't have Yarn, just substitute with npm like this: `npm start`. For the rest of the article, use `npm` in place of `yarn` if you haven't installed it.

Your default browser should launch automatically, and you should get a screen like this:

Create React App

One thing to note is that Create React App supports **hot reloading**. This means any changes we make on the code will cause the browser to automatically refresh. For now, let's first stop the development server by pressing `Ctrl + C`. This step isn't necessary, I'm just showing you how to kill the development server. Once the server has stopped, delete everything in the `src` folder. We'll create all the code from scratch so that you can understand everything inside the `src` folder.

Introducing JSX Syntax

Inside the `src` folder, create an `index.js` file and place the following code in it:

```
import React from 'react';
        import ReactDOM from 'react-dom';

        ReactDOM.render(<h1>Hello World</h1>,
document.getElementById('root'));
```

Start the development server again using `yarn start` or `npm start`. Your browser should display the following content:

Hello React

This is the most basic "Hello World" React example. The `index.js` file is the root of your project where React components will be rendered. Let me explain how the code works:

- Line 1: React package is imported to handle JSX processing
- Line 2: ReactDOM package is imported to render React components.
- Line 4: Call to render function
  - `<h1>Hello World</h1>`: a JSX element
  - `document.getElementById('root')`: HTML container

The HTML container is located in `public/index.html` file. On line 28, you should see `<div id="root"></div>`. This is known as the **root DOM** because everything inside it will be managed by the **React DOM**.

JSX (JavaScript XML) is a syntax expression that allows JavaScript to use tags such as `<div>`, `<h1>`, `<p>`, `<form>`, and `<a>`. It does look a lot like HTML, but there are some key differences. For example, you can't use a `class` attribute, since it's a JavaScript keyword. Instead, `className` is used in its place. Also, events such as `onclick` are spelled `onClick` in JSX. Let's now modify our Hello World code:

```
const element = <div>Hello World</div>;

ReactDOM.render(element,
document.getElementById('root'));
```

I've moved out the JSX code into a variable named `element`. I've also replaced the `h1` tags with `div`. For JSX to work, you need to wrap your elements inside a single parent tag. This is necessary for JSX to work. Take a look at the following example:

```
const element = <span>Hello,</span> <span>Jane</span>;
```

The above code won't work. You'll get a syntax error telling you must enclose adjacent JSX elements in an enclosing tag. Basically, this is how you should enclose your elements:

```
const element = <div>
  <span>Hello, </span>
  <span>Jane</span>
</div>;
```

How about evaluating JavaScript expressions in JSX? Simple, just use curly braces like this:

```
const name = "Jane";
        const element = <p>Hello, {name}</p>
```

... or like this:

```
const user = {
    firstName: "Jane",
    lastName: "Doe"
}
const element = <p>Hello, {user.firstName}
{user.lastName}</p>
```

Update your code and confirm that the browser is displaying "Hello, Jane Doe". Try out other examples such as `{ 5 + 2 }`. Now that you've got the basics of working with JSX, let's go ahead and create a React component.

Declaring React Components

The above example was a simplistic way of showing you how `ReactDOM.render()` works. Generally, we encapsulate all project logic within React components, which are then passed via the `ReactDOM.render` function.

Inside the `src` folder, create a file named `App.js` and type the following code:

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

```
class App extends Component {

    render(){
    return (
        <div>
        Hello World Again!
        </div>
    )
    }

}

export default App;
```

Here we've created a React Component by defining a JavaScript class that is a subclass of `React.Component`. We've also defined a render function that returns a JSX element. You can place additional JSX code within the `<div>` tags. Next, update `src/index.js` with the following code in order to see the changes reflected in the browser:

```
import React from 'react';
import ReactDOM from 'react-dom';

import App from './App';

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

First we import the `App` component. Then we render `App` using JSX format, like

this: <App/>. This is required so that JSX can compile it to an element that can be pushed to the React DOM. After you've saved the changes, take a look at your browser to ensure it's rendering the correct thing.

Next, we'll look at how to apply styling.

Styling JSX Elements

There are are two ways of styling JSX elements:

1. JSX inline styling
2. External Stylesheets

Below is an example of how we can implement JSX inline styling:

```
// src/App.js

…
    render() {
    const headerStyle = {
        color: '#ff0000',
        textDecoration: 'underline'
    }
    return (
        <div>
        <h2 style={headerStyle}>Hello World Again!</h2>
        </div>
    )
    }
…
```

React styling looks a lot like regular CSS but there are key differences. For example, `headerStyle` is an object literal. We can't use semicolons like we normally do. Also, a number of CSS declarations have been changed in order to make them compatible with JavaScript syntax. For example, instead of `text-decoration`, we use `textDecoration`. Basically, use camel case for all CSS keys except for vendor prefixes such as `WebkitTransition`, which must start with a capital letter. We can also implement styling this way:

```
// src/App.js

…
    <h2 style={{color:'#ff0000'}}>Hello World Again!</h2>
…
```

The second method is using external stylesheets. By default, external CSS stylesheets are already supported. If you want to use Sass or Less, please check the reference documentation on how to configure it. Inside the `src` folder, create a file named `App.css` and type the following code:

```
h2 {
    font-size: 4rem;
}
```

Add the following import statement to `src/App.js` on line 2 or 3:

```
// src/App.js
```

```
…
import './App.css';
…
```

After saving, you should see the text content on your browser dramatically change in size.

Styling in React

Now that you've learned how to add styling to your React project, let's go ahead and learn about stateless and stateful React components:

Stateless vs Stateful Components

In React, we generally deal with two types of data: **props** and **state**. `Props` are read-only and are set by a parent component. `State` is defined within a component and can change during the lifecyle of a component. Basically, stateless components (also known as **dumb** components) use `props` to store data, while stateful components (also known as **smart** components) use `state`. To gain a better understanding, let's examine the following practical examples. Inside the `src` folder, create a folder and name it `messages`. Inside that folder, create a file named `message-view.js` and type the following code to create a stateless component:

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

class MessageView extends Component {
    render() {
    return(
```

```
        <div className="container">
        <div className="from">
            <span className="label">From: </span>
            <span className="value">John Doe</span>
        </div>
        <div className="status">
            <span className="label">Status: </span>
            <span className="value"> Unread</span>
        </div>
        <div className="message">
            <span className="label">Message: </span>
            <span className="value">Have a great
day!</span>
        </div>
        </div>
    )
    }
}

export default MessageView;
```

Next, add some basic styling in the `src/App.css` with the following code:

```
container {
    margin-left: 40px;
}

.label {
    font-weight: bold;
```

```
    font-size: 1.2rem;
}

.value {
    color: #474747;
    position: absolute;
    left: 200px;
}

.message .value {
    font-style: italic;
}
```

Finally, modify `src/App.js` so that the entire file looks like this:

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

import './App.css';
import MessageView from './messages/message-view';

class App extends Component {
    render(){
    return (
        <MessageView />
    )
    }
}

export default App;
```

By now, the code should be pretty self explanatory, as I've already explained the basic React concepts. Take a look at your browser now, and you should have the following result:

Messages View

I'd also like to mention that it isn't necessary to use **object-oriented** syntax for stateless components, especially if you aren't defining a lifecycle function. We can rewrite `MessageView` using functional syntax like this:

```
// src/messages/message-view.js

import React from 'react';
import PropTypes from 'prop-types';

export default function MessageView({message}) {
    return (
    <div className="container">
        <div className="from">
        <span className="label">From: </span>
        <span className="value">{message.from}</span>
        </div>
        <div className="status">
        <span className="label">Status: </span>
        <span className="value">{message.status}</span>
        </div>
        <div className="message">
        <span className="label">Message: </span>
```

```
            <span className="value">{message.content}</span>
            </div>
        </div>
        );
}

MessageView.propTypes = {
    message: PropTypes.object.isRequired
}
```

Take note that I've removed the `Component` import, as this isn't required in the functional syntax. This style might be confusing at first, but you'll quickly learn it's faster to write React components this way.

You've successfully created a stateless React Component. It's not complete, though, as there's a bit more work that needs to be done for it to be properly integrated with a stateful component or container. Currently, the `MessageView` is displaying static data. We need to modify so that it can accept input parameters. We do this using `this.props`. We're going to assign a variable named `message` to props. We'll also mark the `message` variable as `required` using the `prop-types` package. This is to make it easier to debug our project as it grows. Update `message-view.js` with the following code:

```
// src/messages/message-view.js
…
import PropTypes from 'prop-types';

class MessageView extends Component {
    render() {
```

```
    const message = this.props.message;

    return(
        <div className="container">
        <div className="from">
            <span className="label">From: </span>
            <span className="value">{message.from}</span>
        </div>
        <div className="status">
            <span className="label">Status: </span>
            <span
className="value">{message.status}</span>
        </div>
        <div className="message">
            <span className="label">Message: </span>
            <span
className="value">{message.content}</span>
        </div>
        </div>
    )
    }

    // Mark message input parameter as required
    MessageView.propTypes = {
    message: PropTypes.object.isRequired
    }
}
…
```

Next, we'll create a stateful component which will act as a parent to

`MessageView` component. We'll make use of the `state` data type to store a message which we'll pass on to `MessageView`. To do this, create `message-list.js` file inside `src/messages` and type the following code:

```
// src/messages/message-list.js

import React, { Component } from 'react';
import MessageView from './message-view';

class MessageList extends Component {

    state = {
    message: {
        from: 'Martha',
        content: 'I will be traveling soon',
        status: 'read'
        }
    }

    render() {
    return(
        <div>
        <h1>List of Messages</h1>
        <MessageView message={this.state.message} />
        </div>
    )
    }
}

export default MessageList;
```

Next, update `src/App.js` such that `MessageList` gets rendered instead (which in turn renders its child component, `MessageView`).

```
// src/App.js
…
import MessageList from './messages/message-list';

class App extends Component {
    render(){
    return (
        <MessageList />
    )
    }
}
…
```

After saving the changes, check your browser to see the result.

Message List

Now let's see how we can display multiple messages using `MessageView` instances. First, we'll change `state.message` to an array and rename it to `messages`. Then, we'll use the map function to generate multiple instances of `MessageView` each corresponding to a message in the `state.messages` array. We'll also need to populate a special attribute named key with a unique value such as `index`. React needs this in order to keep track of what items in the list have been changed, added or removed. Update `MessageList` code as follows:

```
class MessageList extends Component {
```

```
    state = {
    messages:  [
        {
        from: 'John',
        message: 'The event will start next week',
        status: 'unread'
        },
        {
        from: 'Martha',
        message: 'I will be traveling soon',
        status: 'read'
        },
        {
        from: 'Jacob',
        message: 'Talk later. Have a great day!',
        status: 'read'
        }
    ]
    }

    render() {
    const messageViews =
this.state.messages.map(function(message, index) {
        return(
        <MessageView key={index} message={message} />
        )
    })
    return(
        <div>
        <h1>List of Messages</h1>
        {messageViews}
```

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

Check your browser to see the results:

Messages Loop

As you can see, it's easy to define building blocks to create powerful and complex UI interfaces using React. Feel free to add more styling such as putting spacing and dividers between each `MessageView` instance.

Chapter

3

Comparison

by Pavels Jelisejevs

Should I choose Angular or React? Today's bipolar landscape of JavaScript frameworks has left many developers struggling to pick a side in this debate. Whether you're a newcomer trying to figure out where to start, a freelancer picking a framework for your next project, or an enterprise-grade architect planning a strategic vision for your company, you're likely to benefit from having an educated view on this topic.

To save you some time, let me tell you something up front: this article won't give a clear answer on which framework is better. But neither will hundreds of other articles with similar titles. I can't tell you that, because the answer depends on a wide range of factors which make a particular technology more or less suitable for your environment and use case.

Since we can't answer the question directly, we'll attempt something else. We'll compare Angular (2+, not the old AngularJS) and React, to demonstrate how you can approach the problem of comparing any two frameworks in a structured manner on your own and tailor it to your environment. You know, the old "teach a man to fish" approach. That way, when both are replaced by a BetterFramework.js in a year's time, you'll be able to re-create the same train of thought once more.

Where to Start?

Before you pick any tool, you need to answer two simple questions: "Is this a good tool per se?" and "Will it work well for my use case?" Neither of them mean anything on their own, so you always need to keep both of them in mind. All right, the questions might not be that simple, so we'll try to break them down into smaller ones.

Questions on the tool itself:

- How mature is it and who's behind it?
- What kind of features does it have?
- What architecture, development paradigms, and patterns does it employ?
- What is the ecosystem around it?

Questions for self-reflection:

- Will I and my colleagues be able to learn this tool with ease?
- Does is fit well with my project?
- What is the developer experience like?

Using this set of questions you can start your assessment of any tool and we'll base our comparison of React and Angular on them as well.

There's another thing we need to take into account. Strictly speaking, it's not exactly fair to compare Angular to React, since Angular is a full-blown, feature-rich framework, while React just a UI component library. To even the odds, we'll talk about React in conjunction with some of the libraries often used with it.

Maturity

An important part of being a skilled developer is being able to keep the balance between established, time-proven approaches and evaluating new bleeding-edge tech. As a general rule, you should be careful when adopting tools that haven't yet matured due to certain risks:

- The tool may be buggy and unstable.
- It might be unexpectedly abandoned by the vendor.
- There might not be a large knowledge base or community available in case you need help.

Both React and Angular come from good families, so it seems that we can be confident in this regard.

React

React is developed and maintained by Facebook and used in their own products, including Instagram and WhatsApp. It has been around for roughly three and a half years now, so it's not exactly new. It's also one of the most popular projects on GitHub, with about 74,000 stars at the time of writing. Sounds good to me.

Angular

Angular (version 2 and above) has been around less then React, but if you count in the history of its predecessor, AngularJS, the picture evens out. It's maintained by Google and used in AdWords and Google Fiber. Since AdWords is one of the key projects in Google, it is clear they have made a big bet on it and is unlikely to disappear anytime soon.

Features

Like I mentioned earlier, Angular has more features out of the box than React. This can be both a good and a bad thing, depending on how you look at it.

Both frameworks share some key features in common: components, data binding, and platform-agnostic rendering.

Angular

Angular provides a lot of the features required for a modern web application out of the box. Some of the standard features are:

- Dependency injection
- Templates, based on an extended version of HTML
- Routing, provided by @angular/router
- Ajax requests by @angular/http
- @angular/forms for building forms
- Component CSS encapsulation

- XSS protection
- Utilities for unit-testing components.

Having all of these features available out of the box is highly convenient when you don't want to spend time picking the libraries yourself. However, it also means that you're stuck with some of them, even if you don't need them. And replacing them will usually require additional effort. For instance, we believe that for small projects having a DI system creates more overhead than benefit, considering it can be effectively replaced by imports.

React

With React, you're starting off with a more minimalistic approach. If we're looking at just React, here's what we have:

- No dependency injection
- Instead of classic templates it has JSX, an XML-like language built on top of JavaScript
- XSS protection
- Utilities for unit-testing components.

Not much. And this can be a good thing. It means that you have the freedom to choose whatever additional libraries to add based on your needs. The bad thing is that you actually have to make those choices yourself. Some of the popular libraries that are often used together with React are:

- React-router for routing
- Fetch (or axios) for HTTP requests
- A wide variety of techniques for CSS encapsulation
- Enzyme for additional unit-testing utilities.

We've found the freedom of choosing your own libraries liberating. This gives us the ability to tailor our stack to particular requirements of each project, and we didn't find the cost of learning new libraries that high.

Languages, Paradigms, and Patterns

Taking a step back from the features of each framework, let's see what kind higher-level concepts are popular with both frameworks.

React

There are several important things that come to mind when thinking about React: JSX, Flow, and Redux.

JSX

JSX is a controversial topic for many developers: some enjoy it, and others think that it's a huge step back. Instead of following a classical approach of separating markup and logic, React decided to combine them within components using an XML-like language that allows you to write markup directly in your JavaScript code.

While the merits of mixing markup with JavaScript might be debatable, it has an indisputable benefit: static analysis. If you make an error in your JSX markup, the compiler will emit an error instead of continuing in silence. This helps by instantly catching typos and other silly errors.

Flow

Flow is a type-checking tool for JavaScript also developed by Facebook. It can parse code and check for common type errors such as implicit casting or null dereferencing.

Unlike TypeScript, which has a similar purpose, it does not require you to migrate to a new language and annotate your code for type checking to work. In Flow, type annotations are optional and can be used to provide additional hints to the analyzer. This makes Flow a good option if you would like to use static code analysis, but would like to avoid having to rewrite your existing code.

- Further reading: Writing Better JavaScript with Flow

Redux

Redux is a library that helps manage state changes in a clear manner. It was inspired by Flux, but with some simplifications. The key idea of Redux is that the whole state of the application is represented by a single object, which is mutated by functions called reducers. Reducers themselves are pure functions and are implemented separately from the components. This enables better separation of concerns and testability.

If you're working on a simple project, then introducing Redux might be an over complication, but for medium- and large-scale projects, it's a solid choice. The library has become so popular that there are projects implementing it in Angular as well.

All three features can greatly improve your developer experience: JSX and Flow allow you to quickly spot places with potential errors, and Redux will help achieve a clear structure for your project.

Angular

Angular has a few interesting things up its sleeve as well, namely TypeScript and RxJS.

TypeScript

TypeScript is a new language built on top of JavaScript and developed by Microsoft. It's a superset of JavaScript ES2015 and includes features from newer versions of the language. You can use it instead of Babel to write state of the art JavaScript. It also features an extremely powerful typing system that can statically analyze your code by using a combination of annotations and type inference.

There's also a more subtle benefit. TypeScript has been heavily influenced by Java and .NET, so if your developers have a background in one of these languages, they are likely to find TypeScript easier to learn than plain JavaScript (notice how we switched from the tool to your personal environment). Although Angular has been the first major framework to actively adopt TypeScript, it's also possible to use it together with React.

- Further reading: An Introduction to TypeScript: Static Typing for the Web

RxJS

RxJS is a reactive programming library that allows for more flexible handling of asynchronous operations and events. It's a combination of the Observer and Iterator patterns blended together with functional programming. RxJS allows you to treat anything as a continuous stream of values and perform various operations on it such as mapping, filtering, splitting or merging.

The library has been adopted by Angular in their HTTP module as well for some internal use. When you perform an HTTP request, it returns an Observable instead of the usual Promise. Although this library is extremely powerful, it's also quite complex. To master it, you'll need to know your way around different types of Observables, Subjects, as well as around a hundred methods and operators. Yikes, that seems to be a bit excessive just to make HTTP requests!

RxJS is useful in cases when you work a lot with continuous data streams such as web sockets, however, it seems overly complex for anything else. Anyway, when working with Angular you'll need to learn it at least on a basic level.

- Further reading: Introduction to Functional Reactive Programming with RxJS

We've found TypeScript to be a great tool for improving the maintainability of our projects, especially those with a large code base or complex domain/business logic. Code written in TypeScript is more descriptive and easier to follow. Since TypeScript has been adopted by Angular, we hope to see even more projects

using it. RxJS, on the other hand, seems only to be beneficial in certain cases and should be adopted with care. Otherwise, it can bring unwanted complexity to your project.

Ecosystem

The great thing about open source frameworks is the number of tools created around them. Sometimes, these tools are even more helpful than the framework itself. Let's have a look at some of the most popular tools and libraries associated with each framework.

Angular

Angular CLI

A popular trend with modern frameworks is having a CLI tool that helps you bootstrap your project without having to configure the build yourself. Angular has Angular CLI for that. It allows you to generate and run a project with just a couple of commands. All of the scripts responsible for building the application, starting a development server and running tests are hidden away from you in `node_modules`. You can also use it to generate new code during development. This makes setting up new projects a breeze.

- Further reading: The Ultimate Angular CLI Reference

Ionic 2

Ionic 2 is a new version of the popular framework for developing hybrid mobile applications. It provides a Cordova container that is nicely integrated with Angular 2, and a pretty material component library. Using it, you can easily set up and build a mobile application. If you prefer a hybrid app over a native one, this is a good choice.

Material design components

If you're a fan of material design, you'll be happy to hear that there's a Material component library available for Angular. Currently, it's still at an early stage and slightly raw but it has received lots of contributions recently, so we might hope for things to improve soon.

Angular universal

Angular universal is a seed project that can be used for creating projects with support for server-side rendering.

@ngrx/store

@ngrx/store is a state management library for Angular inspired by Redux, being based on state mutated by pure reducers. Its integration with RxJS allows you to utilize the push change detection strategy for better performance.

- Further reading: Managing State in Angular 2 Apps with ngrx/store

Other Tools

There are plenty of other libraries and tools available in the Awesome Angular list.

React

Create React App

Create React App is a CLI utility for React to quickly set up new projects. Similar to Angular CLI it allows you to generate a new project, start a development server and create a bundle. It uses Jest, a relatively new test runner from Facebook, for unit testing, which has some nice features of its own. It also supports flexible application profiling using environment variables, backend proxies for local development, Flow, and other features. Check out this brief

introduction to Create React App for more information.

React Native

React Native is a platform developed by Facebook for creating native mobile applications using React. Unlike Ionic, which produces a hybrid application, React Native produces a truly native UI. It provides a set of standard React components which are bound to their native counterparts. It also allows you to create your own components and bind them to native code written in Objective-C, Java or Swift.

Material UI

There's a material design component library available for React as well. Compared to Angular's version, this one is more mature and has a wider range of components available.

Next.js

Next.js is a framework for the server-side rendering of React applications. It provides a flexible way to completely or partially render your application on the server, return the result to the client and continue in the browser. It tries to make the complex task of creating universal applications as simple as possible so the set up is designed to be as simple as possible with a minimal amount of new primitives and requirements for the structure of your project.

MobX

MobX is an alternative library for managing the state of an application. Instead of keeping the state in a single immutable store, like Redux does, it encourages you to store only the minimal required state and derive the rest from it. It provides a set of decorators to define observables and observers and introduce reactive logic to your state.

- Further reading: How to Manage Your JavaScript Application State with MobX

Storybook

Storybook is a component development environment for React. It allows you to quickly set up a separate application to showcase your components. On top of that, it provides numerous add-ons to document, develop, test and design your components. We've found it to be extremely useful to be able to develop components independently from the rest of the application. You can learn more about Storybook from a previous article.

Other Tools

There are plenty of other libraries and tools available in the Awesome React list.

Adoption, Learning Curve and Development Experience

An important criterion for choosing a new technology is how easy it is to learn. Of course, the answer depends on a wide range of factors such as your previous experience and a general familiarity with the related concepts and patterns. However, we can still try to assess the number of new things you'll need to learn to get started with a given framework. Now, if we assume that you already know ES6+, build tools and all of that, let's see what else you'll need to understand.

React

With React, the first thing you'll encounter is JSX. It does seem awkward to write for some developers. However, it doesn't add that much complexity --- just expressions, which are actually JavaScript, and a special HTML-like syntax. You'll also need to learn how to write components, use props for configuration and manage internal state. You don't need to learn any new logical structures or loops since all of this is plain JavaScript.

The official tutorial is an excellent place to start learning React. Once you're done with that, get familiar with the router. The react router v4 might be slightly complex and unconventional, but nothing to worry about. Using Redux will require a paradigm shift to learn how to accomplish already familiar tasks in a manner suggested by the library. The free Getting Started with Redux video course can quickly introduce you to the core concepts. Depending on the size and the complexity of your project you'll need to find and learn some additional libraries and this might be the tricky part, but after that everything should be smooth sailing.

We were genuinely surprised at how easy it was to get started using React. Even people with a backend development background and limited experience in frontend development were able to catch up quickly. The error messages you might encounter along the way are usually clear and provide explanations on how to resolve the underlying problem. The hardest part may be finding the right libraries for all of the required capabilities, but structuring and developing an application is remarkably simple.

Angular

Learning Angular will introduce you to more new concepts than React. First of all, you'll need to get comfortable with TypeScript. For developers with experience in statically typed languages such as Java or .NET this might be easier to understand than JavaScript, but for pure JavaScript developers, this might require some effort.

The framework itself is rich in topics to learn, starting from basic ones such as modules, dependency injection, decorators, components, services, pipes, templates, and directives, to more advanced topics such as change detection, zones, AoT compilation, and Rx.js. These are all covered in the documentation. Rx.js is a heavy topic on its own and is described in much detail on the official website. While relatively easy to use on a basic level it gets more complicated when moving on to advanced topics.

All in all, we noticed that the entry barrier for Angular is higher than for React. The sheer number of new concepts is confusing to newcomers. And even after you've started, the experience might be a bit rough since you need to keep in mind things like Rx.js subscription management, change detection performance and bananas in a box (yes, this is an actual advice from the documentation). We often encountered error messages that are too cryptic to understand, so we had to google them and pray for an exact match.

It might seem that we favor React here, and we definitely do. We've had experience onboarding new developers to both Angular and React projects of comparable size and complexity and somehow with React it always went smoother. But, like I said earlier, this depends on a broad range of factors and might work differently for you.

Putting it Into Context

You might have already noted that each framework has its own set of capabilities, both with their good and bad sides. But this analysis has been done outside of any particular context and thus doesn't provide an answer on which framework should you choose. To decide on that, you'll need to review it from a perspective of your project. This is something you'll need to do on your own.

To get started, try answering these questions about your project and when you do, match the answers against what you've learned about the two frameworks. This list might not be complete, but should be enough to get you started:

1. How big is the project?
2. How long is it going to be maintained for?
3. Is all of the functionality clearly defined in advance or are you expected to be flexible?
4. If all of the features are already defined, what capabilities do you need?
5. Are the domain model and business logic complex?
6. What platforms are you targeting? Web, mobile, desktop?
7. Do you need server-side rendering? Is SEO important?

8. Will you be handling a lot of real-time event streams?
9. How big is your team?
10. How experienced are your developers and what is their background?
11. Are there any ready-made component libraries that you would like to use?

If you're starting a big project and you would like to minimize the risk of making a bad choice, consider creating a proof-of-concept product first. Pick some of the key features of the projects and try to implement them in a simplistic manner using one of the frameworks. PoCs usually don't take a lot if time to build, but they'll give you some valuable personal experience on working with the framework and allow you to validate the key technical requirements. If you're satisfied with the results, you can continue with full-blown development. If not, failing fast will save you lot of headaches in the long run.

One Framework to Rule Them All?

Once you've picked a framework for one project, you'll get tempted to use the exact same tech stack for your upcoming projects. Don't. Even though it's a good idea to keep your tech stack consistent, don't blindly use the same approach every time. Before starting each project, take a moment to answer the same questions once more. Maybe for the next project, the answers will be different or the landscape will change. Also, if you have the luxury of doing a small project with a non-familiar tech stack, go for it. Such experiments will provide you with invaluable experience. Keep your mind open and learn from your mistakes. At some point, a certain technology will just feel natural and *right*.

Chapter 4

Ready Fast with Pre-configured Builds

by Pavels Jelisejevs

Starting a new React project nowadays is not as simple as we'd like it to be. Instead of instantly diving into the code and bringing your application to life, you have to spend time configuring the right build tools to set up a local development environment, unit testing, and a production build. But there are projects where all you need is a simple setup to get things running quickly and with minimal effort.

Create React App provides just that. It's a CLI tool from Facebook that allows you to generate a new React project and use a pre-configured Webpack build for development. Using it, you'll never have to look at the Webpack config again.

How Does Create React App Work?

Create React App is a standalone tool that should be installed globally via npm, and called each time you need to create a new project:

```
npm install -g create-react-app
```

To create a new project, run:

```
create-react-app react-app
```

Create React App will set up the following project structure:

```
.
├── .gitignore
├── README.md
├── package.json
├── node_modules
├── public
|   ├── favicon.ico
|   └── index.html
└── src
    ├── App.css
    ├── App.js
    ├── App.test.js
    ├── index.css
    ├── index.js
    └── logo.svg
```

It will also add a `react-scripts` package to your project that will contain all of the configuration and build scripts. In other words, your project depends `react-scripts`, not on `create-react-app` itself. Once the installation is complete, you can start working on your project.

Starting a Local Development Server

The first thing you'll need is a local development environment. Running `npm start` will fire up a Webpack development server with a watcher that will automatically reload the application once you change something. Hot reloading, however, is only supported for styles.

The application will be generated with a number of features built-in.

ES6 and ES7

The application comes with its own Babel preset, babel-preset-react-app, to support a set of ES6 and ES7 features. It even supports some of the newer features like async/await, and import/export statements. However, certain features, like decorators, have been intentionally left out.

Asset import

You can also import CSS files from your JS modules that allow you to bundle styles that are only relevant for the modules that you ship. The same thing can be done for images and fonts.

ESLint

During development, your code will also be run through ESLint, a static code analyzer that will help you spot errors during development.

Environment variables

You can use Node environment variables to inject values into your code at built-time. React-scripts will automatically look for any environment variables starting with `REACT_APP_` and make them available under the global `process.env`. These variables can be in a `.env` file for convenience:

```
REACT_APP_BACKEND=http://my-api.com
                REACT_APP_BACKEND_USER=root
```

You can then reference them in your code:

```
fetch(`${process.env.REACT_APP_SECRET_CODE}/endpoint`)
```

Proxying to a backend

If your application will be working with a remote backend, you might need to be able to proxy requests during local development to bypass CORS. This can be set up by adding a proxy field to your `package.json` file:

```
"proxy": "http://localhost:4000",
```

This way, the server will forward any request that doesn't point to a static file the given address.

Running Unit Tests

Executing `npm test` will run tests using Jest and start a watcher to re-run them whenever you change something:

```
PASS  src/App.test.js
    ✓ renders without crashing (7ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.123s, estimated 1s
Ran all test suites.
```

```
Watch Usage
    › Press p to filter by a filename regex pattern.
    › Press q to quit watch mode.
    › Press Enter to trigger a test run.
```

Jest is a test runner also developed by Facebook as an alternative to Mocha or Karma. It runs the tests on a Node environment instead of a real browser, but provides some of the browser-specific globals using jsdom.

Jest also comes integrated with your VCS and by default will only run tests on files changed since your last commit. For more on this, refer to "How to Test React Components Using Jest".

Creating a Production Bundle

When you finally have something you deploy, you can create a production bundle using `npm run build`. This will generate an optimized build of your application, ready to be deployed to your environment. The generated artifacts will be placed in the build folder:

```
.
├── asset-manifest.json
├── favicon.ico
├── index.html
└── static
    ├── css
    |   ├── main.9a0fe4f1.css
    |   └── main.9a0fe4f1.css.map
    ├── js
    |   ├── main.3b7bfee7.js
```

```
|   └── main.3b7bfee7.js.map
└── media
    └── logo.5d5d9eef.svg
```

The JavaScript and CSS code will be minified, and CSS will additionally be run through Autoprefixer to enable better cross-browser compatibility.

Deployment

React-scripts provides a way to deploy your application to GitHub pages by simply adding a homepage property to `package.json`. There's also a separate Heroku build pack.

Opting Out

If at some point you feel that the features provided are no longer enough for your project, you can always opt out of using react-scripts by running `npm run eject`. This will copy the Webpack configuration and build scripts from `react-scripts` into your project and remove the dependency. After that, you're free to modify the configuration however you see fit.

In Conclusion

If you're looking to start a new React project look no further. Create React App will allow you to quickly start working on your application instead of writing yet another Webpack config.

Chapter

5

CSS to Styled Components

by Chris Laughlin

While many aspects of building applications with React have been standardized to some degree, styling is one area where there are still a lot of competing options. Each has its pros and cons, and there's no clear best choice.

In this article, I'll provide a condensed overview of the progression in web application styling with regards to React components. I'll also give a brief introduction to styled-components.

Evolution of Styling in JavaScript

The initial release of CSS was in 1996, and not much has changed since. In its third major release, and with a fourth on the way, it has continued to grow by adding new features and has maintained its reputation as a fundamental web technology. CSS will always be the gold standard for styling web components, but how it's used is changing every day.

From the days when we could build a website from sliced up images to the times when custom, hand-rolled CSS could reflect the same as an image, the evolution of CSS implementation has grown with the movement of JavaScript and the web as a platform.

Since React's release in 2013, component-built web applications have become the norm. The implementation of CSS has, in turn, been questioned. The main argument against using CSS in-line with React has been the separation of concerns (SoC). SoC is a design principle that describes the division of a program into sections, each of which addresses a different concern. This principle was used mainly when developers kept the three main web technologies in separate files: styles (CSS), markup (HTML) and logic (JavaScript).

This changed with the introduction of JSX in React. The development team argued that what we had been doing was, in fact, *the separation of technologies,*

not concerns. One could ask, since JSX moved the markup into the JavaScript code, why should the styles be separate?

As opposed to divorcing styles and logic, different approaches can be used to merge them in-line. An example of this can be seen below:

```
<button style="background: red; border-radius: 8px;
color: white;">Click Me</button>
```

In-line styles move the CSS definitions from the CSS file. This thereby removes the need to import the file and saves on bandwidth, but sacrifices readability, maintainability, and style inheritance.

CSS Modules

button.css

```
.button {
    background: red;
    border-radius: 8px;
    color: white;
}
```

button.js

```
import styles from './button.css';
```

```
                document.body.innerHTML = `<button
class="${styles.button}">test</button>`;
```

As we can see from the code example above, the CSS still lives in its own file. However, when CSS Modules is bundled with Webpack or another modern bundler, the CSS is added as a script tag to the HTML file. The class names are also hashed to provide a more granular style, resolving the problems that come with cascading style sheets.

The process of hashing involves generating a unique string instead of a class name. Having a class name of `btn` will result in a hash of `DhtEg` which prevents styles cascading and applying styles to unwanted elements.

index.html

```
<style>
.DhtEg {
    background: red;
    border-radius: 8px;
    color: white;
}
</style>

…

<button class="DhtEg">test</button>
```

From the example above we can see the style tag element added by CSS

Modules, with the hashed class name and the DOM element we have that uses the hash.

Glamor

Glamor is a CSS-in-JS library that allows us to declare our CSS in the same file as our JavaScript. Glamor, again, hashes the class names but provides a clean syntax to build CSS style sheets via JavaScript.

The style definition is built via a JavaScript variable that describes each of the attributes using camel case syntax. The use of camel case is important as CSS defines all attributes in *train case*. The main difference is the change of the attribute name. This can be an issue when copying and pasting CSS from other parts of our application or CSS examples. For example `overflow-y` would be changed to `overFlowY`. However, with this syntax change, Glamor supports media queries and shadow elements, giving more power to our styles:

button.js

```
import { css } from 'glamor';

const rules = css({
    background: red;
    borderRadius: 8px;
    color: 'white';
});

const button = () => {
    return <button {...rules}>Click Me</button>;
};
```

styled-components

styled-components is a new library that focuses on keeping React components and styles together. Providing a clean and easy-to-use interface for styling both React and React Native, styled-components is changing not only the implementation but the thought process of building styled React components.

styled-components can be installed from npm via:

```
npm install styled-components
```

Imported as any standard npm package:

```
import styled from 'styled-components';
```

Once installed, it's time to start making styled React components easy and enjoyable.

Building Generic Styled React Components

Styled React components can be built in many ways. The styled-components library provides patterns that enable us to build well-structured UI applications. Building from small UI components — such as buttons, inputs, typography and tabs — creates a more unified and coherent application.

Using our button example from before, we can build a generic button using styled-components:

```
const Button = styled.button`
    background: red;
    border-radius: 8px;
    color: white;
`;

class Application extends React.Component {
    render() {
    return (
        <Button>Click Me</Button>
    )
    }
}
```

Codepen

As we can see, we're able to create our generic button while keeping the CSS in-line with the JavaScript. styled-components provides a wide range of elements that we can style. We can do this by using direct element references or by passing strings to the default function.

```
const Button = styled.button`
    background: red;
    border-radius: 8px;
    color: white;
`;

const Paragraph = styled.p`
    background: green;
```

```
`;

const inputBg = 'yellow';
const Input = styled.input`
    background: ${inputBg};
    color: black;
`;

const Header = styled('h1')`
    background: #65a9d7;
    font-size: 26px;
`

class Application extends React.Component {
    render() {
    return (
        <div>
        <Button>Click Me</Button>
        <Paragraph>Read ME</Paragraph>
        <Input
            placeholder="Type in me"
        />
        <Header>I'm a H1</Header>
        </div>
    )
    }
}
```

Codepen

The main advantage to this styling method is being able to write pure CSS. As

seen in the Glamor example, the CSS attribute names have had to be changed to camel case, as they were attributes of a JavaScript object.

styled-components also produces React friendly primitives that act like the existing elements. Harnessing JavaScript template literals allows us to use the full power of CSS to style components. As seen in the input element example, we can define external JavaScript variables and apply these to our styles.

With these simple components, we can easily build a style guide for our application. But in many cases, we'll also need more complicated components that can change based on external factors.

Customizable Styled React Components

The customizable nature of styled-components is the real strength. This could commonly be applied to a button that needs to change styles based on context. In this case, we have two button sizes — small and large. Below is the pure CSS method:

CSS

```
button {
    background: red;
    border-radius: 8px;
    color: white;
}

.small {
    height: 40px;
    width: 80px;
}
```

```
.medium {
    height: 50px;
    width: 100px;
}

.large {
    height: 60px;
    width: 120px;
}
```

JavaScript

```
class Application extends React.Component {
    render() {
    return (
        <div>
        <button className="small">Click Me</button>
        <button className="large">Click Me</button>
        </div>
    )
    }
}
```

Codepen

When we reproduce this using styled-components, we create a Button component that has the basic default style for the background. As the component acts like a React component, we can make use of props and change

the style outcome accordingly:

```
const Button = styled.button`
    background: red;
    border-radius: 8px;
    color: white;
    height: ${props => props.small ? 40 : 60}px;
    width: ${props => props.small ? 60 : 120}px;
`;

class Application extends React.Component {
    render() {
    return (
        <div>
        <Button small>Click Me</Button>
        <Button large>Click Me</Button>
        </div>
    )
    }
}
```

Codepen

Advanced Usage

styled-components provides the ability to create complex advanced components, and we can use existing JavaScript patterns to compose components. The example below demonstrates how components are composed — in the use case of notification messages that all follow a basic style, but each type having a different background color. We can build a basic, styled component and compose on top to create advanced components:

```
const BasicNotification = styled.p`
    background: lightblue;
    padding: 5px;
    margin: 5px;
    color: black;
`;

const SuccessNotification = styled(BasicNotification)`
    background: lightgreen;
`;

const ErrorNotification = styled(BasicNotification)`
    background: lightcoral;
    font-weight: bold;
`;

class Application extends React.Component {
    render() {
    return (
        <div>
        <BasicNotification>Basic
Message</BasicNotification>
        <SuccessNotification>Success
Message</SuccessNotification>
        <ErrorNotification>Error
Message</ErrorNotification>
        </div>
    )
    }
}
```

Codepen

As styled-components allows us to pass standard DOM elements and other components, we can compose advanced features from basic components.

Component Structure

From our advanced and basic example, we can then build a component structure. Most standard React applications have a components directory: we place our styled components in a `styledComponents` directory. Our `styledComponents` directory holds all the basic and composed components. These are then imported into the display components used by our application. The directory layout can be seen below:

```
src/
    components/
    addUser.js
    styledComponents/
        basicNotification.js
        successNotification.js
        errorNotification.js
```

Conclusion

As we've seen in this post, the ways in which we can style our components varies greatly — none of which is a clear winning method. This article has shown that styled-components has pushed the implementation of styling elements forward, and has caused us to question our thought processes with regards to our approach.

Every developer, including myself, has their favorite way of doing things, and it's

good to know the range of different methods out there to use depending on the application we're working on. Styling systems and languages have advanced greatly throughout the years, and there's no question that they are sure to develop further and change more in the future. It’s a very exciting and interesting time in front-end development.

Chapter 6

by Matt Burnett

When React was first introduced, one of the features that caught most people's attention (and drew the most criticism) was JSX. If you're learning React, or have ever seen any code examples, you probably did a double-take at the syntax. What is this strange amalgamation of HTML and JavaScript? Is this even real code?

Let's take a look at what JSX actually is, how it works, and why the heck we'd want to be mixing HTML and JS in the first place!

What is JSX?

Defined by the React Docs as an "extension to JavaScript" or “syntax sugar for calling `React.createElement(component, props, ...children)`”, JSX is what makes writing your React Components easy.

JSX is considered a domain-specific language (DSL), which can look very similar to a template language, such as Mustache, Thymeleaf, Razor, Twig, or others.

It doesn't render out to HTML directly, but instead renders to React Classes that are consumed by the Virtual DOM. Eventually, through the mysterious magic of the Virtual DOM, it will make its way to the page and be rendered out to HTML.

How Does it Work?

JSX is basically still just JavaScript with some extra functionality. With JSX, you can write code that looks very similar to HTML or XML, but you have the power of seamlessly mixing JavaScript methods and variables into your code. JSX is interpreted by a transpiler, such as Babel, and rendered to JavaScript code that the UI Framework (React, in this case) can understand.

Don't like JSX? That's cool. It's technically not required, and the React Docs

actually include a section on using "React Without JSX". Let me warn you right now, though, it's not pretty. Don't believe me? Take a look.

JSX:

```
class SitePoint extends Component {
    render() {
    return (
        <div>My name is
<span>{this.props.myName}</span></div>
    )
    }
}
```

React Sans JSX:

```
class SitePoint extends Component {
    render() {
    return React.createElement(
        "div",
        null,
        "My name is",
        React.createElement(
            "span",
            null,
            this.props.myName
        )
    )
    }
```

```
}
```

Sure, looking at those small example pieces of code on that page you might be thinking, "Oh, that's not so bad, I could do that." But could you imagine writing an entire application like that?

The example is just two simple nested HTML elements, nothing fancy. Basically, just a nested `Hello World`. Trying to write your React application without JSX would be extremely time consuming and, if you're like most of us other developers out here working as characters in DevLand™, it will very likely quickly turn into a convoluted spaghetti code mess. Yuck!

Using frameworks and libraries and things like that are meant to make our lives easier, not harder. I'm sure we've all seen the overuse and abuse of libraries or frameworks in our careers, but using JSX with your React is definitely not one of those cases.

What About Separation of Concerns?

Now, for those of us who learned to not mix our HTML with our JS --- how bad it was, and how if we did, our applications would be haunted by the Bug Gods of the Apocalypse --- mixing your HTML inside of your JavaScript probably seems like all kinds of wrong. After all, you have to maintain a Separation of Concerns at all costs! The world depends on it!

I know, I was there, **I was you**.

But, as weird as it may seem, it's really *not that bad*, and your concerns are still separated. As someone who's been working with React for long enough to be very comfortable with it, using small pieces of HTML/XML inside your JavaScript codebase is really kinda ... magical. It takes the management of HTML out of the

equation, and leaves you with nice, solid code. Just code. It helps to ease the pain of trying to use a markup language --- which was designed for building word documents --- for building applications, and brings it back to the control of the application's code.

Another issue of not separating out your JavaScript from your HTML: what if your JS fails to download or run, and the user is only left with HTML/CSS to render? If you're building your application the React way, then your HTML (and maybe even your CSS if you're using WebPack), is bundled in with your JavaScript itself. So the user really only has one small HTML file to download, then a large JavaScript payload that contains everything else.

Worrying about search engines and SEO is unfortunately still a legitimate concern. We all know that the major search engines now render JavaScript, but initial rendering with JavaScript can still be slower, which could have an effect on your overall ranking. To mitigate this, it's possible to do an initial render of your React on the server before sending it to the client. This will not only allow search engines to pull your site quicker, but also provide your users with a quicker First Meaningful Paint.

Doing this can become complicated quickly, but at the time of this writing, SitePoint itself actually uses this method. Brad Denver did a fantastic write-up of how it was done here: “Universal Rendering: How We Rebuilt SitePoint”.

Rendering server-side is still only going to help with failed loads (which are uncommon) or slow load times (much more common). It won't help with users who completely disable JavaScript. Sorry, but a React app is still a JavaScript-based application. You can do a lot behind the scenes, but once you start mixing in app functionality and state management (e.g. Flux, Redux, or MobX), your time investment vs potential payoff starts going negative.

Not Just for React

JSX with React is pretty great stuff. But what if you're using another framework

or library, but still want to use it? Well, you're in luck --- because JSX technically isn't tied to React. It's still just DSL or "syntax sugar", remember? Vue.js actually supports JSX out of the box, and there have been attempts to use it with other frameworks such as Angular 2 and Ember.

I hope this JSX quick introduction helped to give you a better understanding of just what JSX is, how it can help you, and how it can be used to create your applications. Now get back out there and make some cool stuff!

Chapter

7

Properties & State

Eric Greene

Managing data is essential to any application. Orchestrating the flow of data through the user interface (UI) of an application can be challenging. Often, today's web applications have complex UIs such that modifying the data in one area of the UI can directly and indirectly affect other areas of the UI. Two-way data binding via Knockout.js and Angular.js are popular solutions to this problem.

For some applications (especially with a simple data flow), two-way binding can be a sufficient and quick solution. For more complex applications, two-way data binding can prove to be insufficient and a hindrance to effective UI design. React does not solve the larger problem of application data flow (although Flux does), but it does solve the problem of data flow within a single component.

Within the context of a single component, React solves both the problem of data flow, as well as updating the UI to reflect the results of the data flow. The second problem of UI updates is solved using a pattern named Reconciliation which involves innovative ideas such as a Virtual DOM. The next article will examine Reconciliation in detail. This article is focused on the first problem of data flow, and the kinds of data React uses within its components.

Kinds of Component Data

Data within React Components is stored as either properties or state.

Properties are the input values to the component. They are used when rendering the component and initializing state (discussed shortly). After instantiating the component, properties should be considered immutable. Property values can only be set when instantiating the component, then when the component is re-rendered in the DOM, React will compare between the old and new property values to determine what DOM updates are required.

Here is a CodePen demonstration of setting the property values and updating

the DOM in consideration of updated property values, screenshot below.

State data can be changed by the component and is usually wired into the component's event handlers. Typically, updating the state triggers React components re-render themselves. Before a component is initialized, its state must be initialized. The initialized values can include constant values, as well as, property values (as mentioned above).

In comparison with frameworks such as Angular.js, properties can be thought of as one-way bound data, and state as two-way bound data. This is not a perfect analogy since Angular.js uses one kind of data object which is used two different ways, and React is using two data objects, each with their specific usages.

Properties

My previous React article covered the syntax to specify and access properties. The article explored the use of JavaScript and JSX with static as well as dynamic properties in various code demonstrations. Expanding on the earlier exploration, let's look at some interesting details regarding working with properties.

When adding a CSS class name to a component, the property name `className` must be used, rather than `class` must be used. React requires this because ES2015 identifies the word `class` as a reserved keyword and is used for defining objects. To avoid confusion with this keyword, the property name `className` is used. If a property named `class` is used, React will display a helpful console message informing the developer that the property name needs to be changed to `className`.

Observe the incorrect `class` property name, and the helpful warning message displayed in the Microsoft Edge console window.

Incorrect class property name
Changing the `class` property to `className`, results in the warning message not

being displayed.

class property changed to ClassName
When the property name is changed from `class` to `className` the warning message does not appear. Cehck out the complete CodePen demonstration, screenshot below.

In addition to property names such as `className`, React properties have other interesting aspects. For example, mutating component properties is an anti-pattern. Properties can be set when instantiating a component, but should not be mutated afterwards. This includes changing properties after instantiating the component, as well as after rendering it. Mutating values within a component are considered state, and tracked with the `state` property rather than the `props` property.

In the following code sample, `SomeComponent` is instantiated with `createElement`, and then the property values are manipulated afterwards.

JavaScript:

```
var someComponent = React.createElement(SomeComponent);

someComponent.props.prop1 = "some value";
someComponent.props.prop2 = "some value";
```

JSX:

```
var someComponent = &lt;SomeComponent /&gt;;

someComponent.props.prop1 = "some value";
```

```
someComponent.props.prop2 = "some value";
```

Manipulating the `props` of the instantiated component could result in errors that would be hard to trace. Also, changing the properties does not trigger an update to the component, resulting in the component and the properties could be out of sync.

Instead, properties should be set as part of the component instantiation process, as shown below.

JavaScript:

```
var someComponent = React.createElement(SomeComponent, {
    prop1: "some value",
    prop2: "some value"
});
```

JSX:

```
var someComponent = <SomeComponent prop1="some value"
prop2="some value" />
```

The component can then be re-rendered at which point React will perform its Reconciliation process to determine how the new property values affect the DOM. Then, the DOM is updated with the changes.

See the first CodePen demonstration at the top of this article for a

demonstration of the DOM updates.

State

State represents data that is changed by a component, usually through interaction with the user. To facilitate this change, event handlers are registered for the appropriate DOM elements. When the events occur, the updated values are retrieved from the DOM, and notify the component of the new state. Before the component can utilize state, the state must be initialized via the `getInitialState` function. Typically, the `getInitialState` function initializes the state using static values, passed in properties, or another data store.

```
var Message = React.createClass({

    getInitialState: function() {
        return { message: this.props.message };
    },
```

Once the state is initialized, the state values can be used like property values when rendering the component. To capture the user interactions which update the state, event handlers are registered. To keep the React components self-contained, event handler function objects can be passed in as properties or defined directly on the component object definition itself. See the Codepen demo, with screenshot below.

One of the benefits of React is that standard HTML events are used. Included with standard HTML events is the standard HTML Event object. Learning special event libraries, event handlers, or custom event objects is not needed. Because modern browsers are largely compatible, intermediary cross-browser libraries such as jQuery are not needed.

To handle the state changes, the `setState` function is used to set the new value on the appropriate state properties. Calling this function causes the component to re-render itself.

As shown below in the Visual Studio Code editor, the `setState` function is called from the `_messageChange` event handler.

Visual Studio code editor

Conclusion

React components provide two mechanisms for working with data: properties and state. Dividing data between immutable properties and mutable state more clearly identifies the role of each kind of data, and the component's relationship to it. In general, properties are preferred because they simplify the flow of data. State is useful for capturing data updates resulting from user interactions and other UI events.

The relationship between properties and state facilitate the flow of data through a component. Properties can be used to initialize state, and state values can be used to set properties when instantiating and rendering a component. New values from user interaction are captured via state, and then used to update the

properties.

The larger flow of data within an application is accomplished via a pattern named Flux.

Chapter 8

by Mark Brown

This article is for developers who are familiar with Angular 1.x and would like to learn more about React. We'll look at the different approaches they take to building rich web applications, the overlapping functionality and the gaps that React doesn't attempt to fill.

After reading, you'll have an understanding of the problems that React sets out to solve and how you can use the knowledge you have already to get started using React in your own projects.

Frameworks vs Libraries

Angular is a *framework*, whereas React is a *library* focused only on the view layer. There are costs and benefits associated with both using frameworks and a collection of loosely coupled libraries.

Frameworks try to offer a complete solution, and they may help organize code through patterns and conventions if you're part of a large team. However, having a large API adds cognitive load when you're writing, and you'll spend a lot more time reading documentation and remembering patterns --- especially in the early days when you're still learning.

Using a collection of loosely coupled libraries with small APIs is easier to learn and master, but it means that when you run into problems you'll need to solve them with more code or pull in external libraries as required. This usually results in you having to write *your own* framework to reduce boilerplate.

Out Of The Box

Angular gives you a rich feature set for building web applications. Among its features are:

- HTML templates with dynamic expressions in double curlies `{{ }}`
- built-in directives like `ng-model`, `ng-repeat` and `ng-class` for extending the capability of HTML
- controllers for grouping logic and passing data to the view
- two-way binding as a simple way to keep your view and controller in sync
- a large collection of modules like `$http` for communicating with the server and `ngRoute` for routing
- custom directives for creating your own HTML syntax
- dependency injection for limiting exposure of objects to specific parts of the application
- services for shared business logic
- filters for view formatting helpers.

React, on the other hand, gives you:

- JSX syntax for templates with JavaScript expressions in single curlies `{ }`
- components, which are most like Angular's element directives.

React is unopinionated when it comes to the rest of your application structure and it encourages the use of standard JavaScript APIs over framework abstractions. Rather than providing a wrapper like $http for server communication, you can use fetch() instead. You're free to use constructs like services and filters, but React won't provide an abstraction for them. You can put them in JavaScript modules and require them as needed in your components.

So, while Angular gives you a lot more abstractions for common tasks, React deliberately avoids this to keep you writing standard JavaScript more often and to use external dependencies for everything else.

Bootstrapping

Initializing Angular apps requires a module, a list of dependencies and a root element.

```
let app = angular.module('app', [])
let root = document.querySelector('#root');
angular.element(root).ready(function() {
    angular.bootstrap(root, ['app']);
});
```

The entry point for React is rendering a component into a root node. It's possible to have multiple root components, too:

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

Templates

The anatomy of an Angular view is complex and has many responsibilities. Your HTML templates contain a mix of directives and expressions, which tie the view and the associated controllers together. Data flows throughout multiple contexts via `$scope`.

In React, it's components *all the way down*, data flows in one direction from the top of the component tree down to the leaf nodes. JSX is the most common syntax for writing components, transforming a familiar XML structure into JavaScript. Whilst this does *resemble* a template syntax, it compiles into nested function calls.

```
const App = React.createClass({
    render: function() {
    return (
        <Component>
            <div>{ 2 + 1 }</div>
            <Component prop="value" />
            <Component time={ new Date().getTime() }>
```

```
                <Component />
            </Component>
        </Component>
        )
    }
})
```

The compiled code below should help clarify how the JSX expressions above map to `createElement(component, props, children)` function calls:

```
var App = React.createClass({
    render: function render() {
        return React.createElement(
            Component,
                null,
                React.createElement("div", null, 2 + 1),
                React.createElement(Component, { prop:
"value" }),
            React.createElement(
            Component,
            { time: new Date().getTime() },
            React.createElement(Component, null)
            )
        );
    }
});
```

Template Directives

Let's look at how some of Angular's most used template directives would be written in React components. Now, React doesn't have templates, so these examples are JSX code that would sit inside a component's `render` function. For

example:

```
class MyComponent extends React.Component {
    render() {
        return (
            // JSX lives here
        )
    }
}
```

ng-repeat

```
<ul>
                    <li ng-repeat="word in words">{ word
}</li>
                </ul>
```

We can use standard JavaScript looping mechanisms such as `map` to get an array of elements in JSX.

```
<ul>
    { words.map((word)=> <li>{ word }</li> )}
</ul>
```

ng-class

```
<form ng-class="{ active: active, error: error }">
                </form>
```

In React, we're left to our own devices to create our space-separated list of classes for the `className` property. It's common to use an existing function such as Jed Watson's classNames for this purpose.

```
<form className={ classNames({active: active, error:
error}) }>
</form>
```

The way to think about these attributes in JSX is as if you're setting properties on those nodes directly. That's why it's `className` rather than the `class` attribute name.

```
formNode.className = "active error";
```

ng-if

```
<div>
    <p ng-if="enabled">Yep</p>
</div>
```

`if … else` statements don't work inside JSX, because JSX is just syntactic sugar for function calls and object construction. It's typical to use ternary operators for this or to move conditional logic to the top of the render method, outside of the JSX.

```
// ternary
<div>
    { enabled ? <p>Enabled</p> : null }
</div>

// if/else outside of JSX
let node = null;
if (enabled) {
    node = <p>Enabled</p>;
}
<div>{ node }</div>
```

ng-show / ng-hide

```
<p ng-show="alive">Living</p>
<p ng-hide="alive">Ghost</p>
```

In React, you can set style properties directly or add a utility class, such as `.hidden { display: none }`, to your CSS for the purpose of hiding your elements (which is how Angular handles it).

```
<p style={ display: alive ? 'block' : 'none' }>Living</p>
<p style={ display: alive ? 'none' : 'block' }>Ghost</p>
```

```
<p className={ classNames({ hidden: !alive })
}>Living</p>
<p className={ classNames({ hidden: alive }) }>Ghost</p>
```

You've got the hang of it now. Instead of a special template syntax and attributes, you'll need to use JavaScript to achieve what you want instead.

An Example Component

React's *Components* are most like Angular's *Directives*. They're used primarily to abstract complex DOM structures and behavior into reusable pieces. Below is an example of a slideshow component that accepts an array of slides, renders a list of images with navigational elements and keeps track of its own `activeIndex` state to highlight the active slide.

```
<div ng-controller="SlideShowController">
    <slide-show slides="slides"></slide-show>
</div>
```

```
app.controller("SlideShowController", function($scope) {
    $scope.slides = [{
        imageUrl: "allan-beaver.jpg",
        caption: "Allan Allan Al Al Allan"
    }, {
        imageUrl: "steve-beaver.jpg",
        caption: "Steve Steve Steve"
    }];
});
```

```
app.directive("slideShow", function() {
    return {
        restrict: 'E',
        scope: {
            slides: '='
        },
        template: `
            <div class="slideshow">
                <ul class="slideshow-slides">
                <li ng-repeat="slide in slides" ng-class="{
active: $index == activeIndex }">
                    <figure>
                    <img ng-src="{{ slide.imageUrl }}" />
                    <figcaption ng-show="slide.caption">{{
slide.caption }}</figcaption>
                </figure>
                </li>
                </ul>
                <ul class="slideshow-dots">
                    <li ng-repeat="slide in slides"
ng-class="{ active: $index == activeIndex }">
                        <a ng-click="jumpToSlide($index)">{{
$index + 1 }}</a>
                    </li>
                </ul>
            </div>
        `,
        link: function($scope, element, attrs) {
            $scope.activeIndex = 0;

            $scope.jumpToSlide = function(index) {
                $scope.activeIndex = index;
            };
```

```
        }
    };
});
```

The Slideshow Component in Angular

See the Pen Angular Slideshow by SitePoint (@SitePoint) on CodePen.

This component in React would be rendered inside another component and passed the slides data via props.

```
let _slides = [{
    imageUrl: "allan-beaver.jpg",
    caption: "Allan Allan Al Al Allan"
}, {
    imageUrl: "steve-beaver.jpg",
    caption: "Steve Steve Steve"
}];

class App extends React.Component {
    render() {
        return <SlideShow slides={ _slides } />
    }
}
```

React components have a local scope in `this.state`, which you can modify by calling `this.setState({ key: value })`. Any changes to state causes the component to re-render itself.

```
class SlideShow extends React.Component {
    constructor() {
            super()
```

```
            this.state = { activeIndex: 0 };
        }
        jumpToSlide(index) {
            this.setState({ activeIndex: index });
        }
        render() {
            return (
                <div className="slideshow">
                    <ul className="slideshow-slides">
                        {
                            this.props.slides.map((slide,
index) => (
                                <li className={ classNames({
active: index == this.state.activeIndex }) }>
                                    <figure>
                                        <img src={
slide.imageUrl } />
                                        { slide.caption ?
<figcaption>{ slide.caption }</figcaption> : null }
                                    </figure>
                                </li>
                            ))
                        }
                    </ul>
                    <ul className="slideshow-dots">
                        {
                            this.props.slides.map((slide,
index) => (
                                <li className={ (index ==
this.state.activeIndex) ? 'active': '' }>
                                    <a onClick={ (event)=>
this.jumpToSlide(index) }>{ index + 1 }</a>
                                </li>
                            ))
```

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

Events in React look like old-school inline event handlers such as `onClick`. Don't feel bad, though: under the hood it does the right thing and creates highly performant delegated event listeners.

The Slideshow Component in React

See the Pen React SlideShow by SitePoint (@SitePoint) on CodePen.

Two-Way Binding

Angular's trusty `ng-model` and `$scope` form a link where the data flows back and forth between a form element and properties on a JavaScript object in a controller.

```
app.controller("TwoWayController", function($scope) {
    $scope.person = {
        name: 'Bruce'
    };
                });
```

```
<div ng-controller="TwoWayController">
    <input ng-model="person.name" />
    <p>Hello {{ person.name }}!</p>
</div>
```

React eschews this pattern in favor of a one-way data flow instead. The same types of views can be built with both patterns though.

```
class OneWayComponent extends React.Component {
    constructor() {
        super()
        this.state = { name: 'Bruce' }
    }
    change(event) {
        this.setState({ name: event.target.value });
    }
    render() {
        return (
            <div>
                <input value={ this.state.name } onChange={
(event)=> this.change(event) } />
                <p>Hello { this.state.name }!</p>
            </div>
        );
    }
}
```

The `<input>` here is called a "controlled input". This means its value is only ever changed when the `render` function is called (on every key stroke in the example above). The component itself is called "stateful" because it manages its own data. This isn't recommended for the majority of components. The ideal is to keep components "stateless" and have data passed to them via `props` instead.

See the Pen One-Way Data Flow in React by SitePoint (@SitePoint) on CodePen.

Typically, a stateful Container Component or Controller View sits at the top of the tree with many stateless child components underneath. For more information on this, read What Components Should Have State? from the docs.

Call Your Parents

Whilst data flows down in one direction, it's possible to call methods on the parent through callbacks. This is usually done in response to some user input. This flexibility gives you a lot of control when refactoring components to their simplest presentational forms. If the refactored components have no state at all, they can be written as pure functions.

```
// A presentational component written as a pure function
const OneWayComponent = (props)=> (
    <div>
        <input value={ props.name } onChange={ (event)=>
props.onChange(event.target.value) } />
        <p>Hello { props.name }!</p>
    </div>
);

class ParentComponent extends React.Component {
    constructor() {
        super()
        this.state = { name: 'Bruce' };
    }
    change(value) {
        this.setState({name: value});
    }
    render() {
        return (
            <div>
                <OneWayComponent name={ this.state.name }
onChange={ this.change.bind(this) } />
```

```
                <p>Hello { this.state.name }!</p>
            </div>
        )
    }
}
```

This might seem like a round-about pattern at first if you're familiar with two-way data binding. The benefit of having a lot of small presentational "dumb" components that just accept data as `props` and render them is that they are simpler by default, and simple components have *far* fewer bugs. This also prevents the UI from being in an inconsistent state, which often occurs if data is in multiple places and needs to be maintained separately.

Dependency Injection, Services, Filters

JavaScript Modules are a much better way to handle dependencies. You can use them today with a tool like Webpack, SystemJS or Browserify.

```
// An Angular directive with dependencies
app.directive('myComponent', ['Notifier', '$filter',
function(Notifier, $filter) {
    const formatName = $filter('formatName');

    // use Notifier / formatName

}]

// ES6 Modules used by a React component
import Notifier from "services/notifier";
import { formatName } from "filters";

class MyComponent extends React.Component {
```

```
    // use Notifier / formatName

}
```

Sounds Great. Can I Use Both!?

Yes! It's possible to render React components inside an existing Angular application. Ben Nadel has put together a good post with screencast on how to render React components inside an Angular directive. There's also ngReact, which provides a `react-component` directive for acting as the glue between React and Angular.

If you've run into rendering performance problems in certain parts of your Angular application, it's possible you'll get a performance boost by delegating some of that rendering to React. That being said, it's not ideal to include two large JavaScript libraries that solve a lot of the same problems. Even though React is just the view layer, it's roughly the same size as Angular, so that weight may be prohibitive based on your use case.

While React and Angular solve some of the same problems, they go about it in very different ways. React favors a functional, declarative approach, where components are pure functions free of side effects. This functional style of programming leads to fewer bugs and is simpler to reason about.

How About Angular 2?

Components in Angular 2 resemble React components in a lot of ways. The example components in the docs have a class and template in close proximity. Events look similar. It explains how to build views using a Component Hierarchy, just as you would if you were building it in React, and it embraces ES6 modules for dependency injection.

```
// Angular 2
```

```
@Component({
    selector: 'hello-component',
    template: `
        <h4>Give me some keys!</h4>
        <input (keyup)="onKeyUp($event)" />
        <div>{{ values }}</div>
    `
})
class HelloComponent {
    values='';
    onKeyUp(event) {
        this.values += event.target.value + ' | ';
    }
}

// React
class HelloComponent extends React.Component {
    constructor(props) {
    super()
    this.state = { values: '' };
}
onKeyUp(event) {
    const values = `${this.state.values + event.target.value}
| `;
    this.setState({ values: values });
}
    render() {
        return (
            <div>
                <h4>Give me some keys!</h4>
                <div><input onKeyUp={ this.onKeyUp.bind(this)
} /></div>
                <div>{ this.state.values }</div>
            </div>
```

```
        );
    }
}
```

A lot of the work on Angular 2 has been making it perform DOM updates a lot more efficiently. The previous template syntax and complexities around scopes led to a lot of performance problems in large apps.

A Complete Application

In this article I've focused on templates, directives and forms, but if you're building a complete application, you're going to require other things to help you manage your data model, server communication and routing at a minimum. When I first learned Angular and React, I created an example Gmail application to understand how they worked and to see what the developer experience was like before I started using them in real applications.

You might find it interesting to look through these example apps to compare the differences in React and Angular. The React example is written in CoffeeScript with CJSX, although the React community has since gathered around ES6 with Babel and Webpack, so that's the tooling I would suggest adopting if you're starting today.

- https://github.com/markbrown4/gmail-react
- https://github.com/markbrown4/gmail-angular

There's also the TodoMVC applications you could look at to compare:

- http://todomvc.com/examples/react/
- http://todomvc.com/examples/angularjs/

Chapter 9

Components

by Camile Reyes

React is a framework that has made headway within the JavaScript developer community. React has a powerful composition framework for designing components. React components are bits of reusable code you can wield in your web application.

React components are not tightly coupled from the DOM, but how easy are they to unit test? In this take, let's explore what it takes to unit test React components. I'll show the thought process for making your components testable.

Keep in mind, I'm only talking about **unit tests**, which are a special kind of test. (For more on the different kinds of tests, I recommend you read "JavaScript Testing: Unit vs Functional vs Integration Tests".)

With unit tests, I'm interested in two things: rapid and neck-breaking feedback. With this, I can iterate through changes with a high degree of confidence and code quality. This gives you a level of reassurance that your React components will not land dead on the browser. Being capable of getting good feedback at a rapid rate gives you a competitive edge --- one that you'll want to keep in today's world of agile software development.

For the demo, let's do a list of the great apes, which is filterable through a checkbox. You can find the entire codebase on GitHub. For the sake of brevity, I'll show only the code samples that are of interest. This article assumes a working level of knowledge with React components.

If you go download and run the demo sample code, you'll see a page like this:

The Great Apes Demo in React Components

Write Testable Components

In React, a good approach is to start with a hierarchy of components. The single responsibility principle comes to mind when building each individual component. React components use object composition and relationships.

For the list of the great apes, for example, I have this approach:

```
FilterableGreatApeList
|_ GreatApeSearchBar
|_ GreatApeList
    |_ GreatApeRow
```

Take a look at how a great ape list has many great ape rows with data. React components make use of this composition data model, and it's also testable.

In React components, avoid using inheritance to build reusable components. If you come from a classic object-oriented programming background, keep this in mind. React components don't know their children ahead of time. Testing components that descend from a long chain of ancestors can be a nightmare.

I'll let you explore the `FilterableGreatApeList` on your own. It's a React component with two separate components that are of interest here. Feel free to explore the unit tests that come with it, too.

To build a testable `GreatApeSearchBar`, for example, do this:

```
class GreatApeSearchBar extends Component {
    constructor(props) {
```

```
        super(props);

        this.handleShowExtantOnlyChange =
this.handleShowExtantOnlyChange.bind(this);
    }

    handleShowExtantOnlyChange(e) {

this.props.onShowExtantOnlyInput(e.target.checked);
    }

    render() {
        return(
            <form>
                <input
                    id="GreatApeSearchBar-showExtantOnly"
                    type="checkbox"
                    checked={this.props.showExtantOnly}

onChange={this.handleShowExtantOnlyChange}
                />

                <label
htmlFor="GreatApeSearchBar-showExtantOnly">Only show
extant species</label>
            </form>
        );
    }
}
```

This component has a checkbox with a label and wires up a click event. This

approach may already be all too familiar to you, which is a very good thing.

Note that with React, testable components come for free, straight out of the box. There's nothing special here – an event handler, JSX, and a render method.

The next React component in the hierarchy is the `GreatApeList`, and it looks like this:

```
class GreatApeList extends Component {
    render() {
        let rows = [];

        this.props.apes.forEach((ape) => {
            if (!this.props.showExtantOnly) {
                rows.push(<GreatApeRow key={ape.name}
ape={ape} />);

            return;
        }

            if (ape.isExtant) {
                rows.push(<GreatApeRow key={ape.name}
ape={ape} />);
            }
        });

        return (
            <div>
                {rows}
            </div>
        );
    }
```

```
}
```

It's a React component that has the `GreatApeRow` component and it's using object composition. This is React's most powerful composition model at work. Note the lack of inheritance when you build reusable yet testable components.

In programming, object composition is a design pattern that enables data-driven elements. To think of it another way, a `GreatApeList` has many `GreatApeRow` objects. It's this relationship between UI components that drives the design. React components have this mindset built in. This way of looking at UI elements allows you to write some nice unit tests.

Here, you check for the `this.props.showExtantOnly` flag that comes from the checkbox. This `showExtantOnly` property gets set through the event handler in `GreatApeSearchBar`.

For unit tests, how do you unit test React components that depend on other components? How about components intertwined with each other? These are great questions to keep in mind as we get into testing soon. React components may yet have secrets one can unlock.

For now, let's look at the `GreatApeRow`, which houses the great ape data:

```
class GreatApeRow extends Component {
    render() {
        return (
            <div>
                <img
                    className="GreatApeRow-image"
```

```
                    src={this.props.ape.image}
                    alt={this.props.ape.name}
                />

                <p className="GreatApeRow-detail">
                    <b>Species:</b> {this.props.ape.name}
                </p>

                <p className="GreatApeRow-detail">
                    <b>Age:</b> {this.props.ape.age}
                </p>
            </div>
        );
    }
}
```

With React components, it's practical to isolate each UI element with a laser focus on a single concern. This has key advantages when it comes to unit testing. As long as you stick to this design pattern, you'll find it seamless to write unit tests.

Test Utilities

Let's recap our biggest concern when it comes to testing React components. How do I unit test a single component in isolation? Well, as it turns out, there's a nifty utility that enables you to do that.

The Shallow Renderer in React allows you to render a component one level deep. From this, you can assert facts about what the render method does. What's remarkable is that it doesn't require a DOM.

Using ES6, you use it like this:

```
import ShallowRenderer from 'react-test-renderer/
shallow';
```

In order for unit tests to run fast, you need a way to test components in isolation. This way, you can focus on a single problem, test it, and move on to the next concern. This becomes empowering as the solution grows and you're able to refactor at will --- staying close to the code, making rapid changes, and gaining reassurance it will work in a browser.

One advantage of this approach is you think better about the code. This produces the best solution that deals with the problem at hand. I find it liberating when you're not chained to a ton of distractions. The human brain does a terrible job at dealing with more than one problem at a time.

The only question remaining is, how far can this little utility take us with React components?

Put It All Together

Take a look at `GreatApeList`, for example. What's the main concern you're trying to solve? This component shows you a list of great apes based on a filter.

An effective unit test is to pass in a list and check facts about what this React component does. We want to ensure it filters the great apes based on a flag.

One approach is to do this:

```
import GreatApeList from './GreatApeList';

const APES = [{ name: 'Australopithecus afarensis',
isExtant: false },
    { name: 'Orangutan', isExtant: true }];

// Arrange
const renderer = new ShallowRenderer();
renderer.render(<GreatApeList
    apes={APES}
    showExtantOnly={true} />);

// Act
const component = renderer.getRenderOutput();
const rows = component.props.children;

// Assert
expect(rows.length).toBe(1);
```

Note that I'm testing React components using Jest. For more on this, check out "How to Test React Components Using Jest".

In JSX, take a look at `showExtantOnly={true}`. The JSX syntax allows you to set a state to your React components. This opens up many ways to unit test components given a specific state. JSX understands basic JavaScript types, so a `true` flag gets set as a boolean.

With the list out of the way, how about the `GreatApeSearchBar`? It has this event handler in the `onChange` property that might be of interest.

One good unit test is to do this:

```
import GreatApeSearchBar from './GreatApeSearchBar';

// Arrange
let showExtantOnly = false;
const onChange = (e) => { showExtantOnly = e };

const renderer = new ShallowRenderer();
renderer.render(<GreatApeSearchBar
    showExtantOnly={true}
    onShowExtantOnlyInput={onChange} />);

// Act
const component = renderer.getRenderOutput();
const checkbox = component.props.children[0];

checkbox.props.onChange({ target: { checked: true } });

// Assert
expect(showExtantOnly).toBe(true);
```

To handle and test events, you use the same shallow rendering method. The `getRenderOutput` method is useful for binding callback functions to components with events. Here, the `onShowExtantOnlyInput` property gets assigned the callback `onChange` function.

On a more trivial unit test, what about the `GreatApeRow` React component? It displays great ape information using HTML tags. Turns out, you can use the shallow renderer to test this component too.

For example, let's ensure we render an image:

```
import GreatApeRow from './GreatApeRow';

const APE = {
    image: 'https://en.wikipedia.org/wiki/
File:Australopithecus_afarensis.JPG',
    name: 'Australopithecus afarensis'
};

// Arrange
const renderer = new ShallowRenderer();
renderer.render(<GreatApeRow ape={APE} />);

// Act
const component = renderer.getRenderOutput();
const apeImage = component.props.children[0];

// Assert
expect(apeImage).toBeDefined();
expect(apeImage.props.src).toBe(APE.image);
expect(apeImage.props.alt).toBe(APE.name);
```

With React components, it all centers around the `render` method. This makes it somewhat intuitive to know exactly what you need to test. A shallow renderer makes it so you can laser focus on a single component while eliminating noise.

Conclusion

As shown, React components are very testable. There's no excuse to forgo writing good unit tests for your components.

The nice thing is that JSX works for you in each individual test, not against you.

With JSX, you can pass in booleans, callbacks, or whatever else you need. Keep this in mind as you venture out into unit testing React components on your own.

The shallow renderer test utility gives you all you need for good unit tests. It only renders one level deep and allows you to test in isolation. You're not concerned with any arbitrary child in the hierarchy that might break your unit tests.

With the Jest tooling, I like how it gives you feedback only on the specific files you're changing. This shortens the feedback loop and adds laser focus. I hope you see how valuable this can be when you tackle some tough issues.

Chapter

10