

# 6 JAVASCRIPT PROJECTS



7.



8.



9.

COMPLETE PROJECT TUTORIALS

# 6 JavaScript Projects

Copyright © 2018 SitePoint Pty. Ltd.

**Cover Design:** Alex Walker

## Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

## Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

## Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.



Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood  
VIC Australia 3066  
Web: [www.sitepoint.com](http://www.sitepoint.com)  
Email: [books@sitepoint.com](mailto:books@sitepoint.com)

## 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, design, and more.

# Preface

There's no doubt that the JavaScript ecosystem changes fast. Not only are new tools and frameworks introduced and developed at a rapid rate, the language itself has undergone big changes with the introduction of ES2015 (aka ES6). Understandably, many articles have been written complaining about how difficult it is to learn modern JavaScript development these days. We're aiming to minimize that confusion with this set of books on modern JavaScript.

This book presents six complete JavaScript projects; each taking advantage of modern JavaScript and its ecosystem. You'll learn to build several different apps, and along the way you'll pick up a ton of useful advice, tips, and techniques.

## Who Should Read This Book?

This book is for all front-end developers who wish to improve their JavaScript skills. You'll need to be familiar with HTML and CSS and have a reasonable level of understanding of JavaScript in order to follow the discussion.



# Conventions Used

## 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>`

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");`

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

## 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!

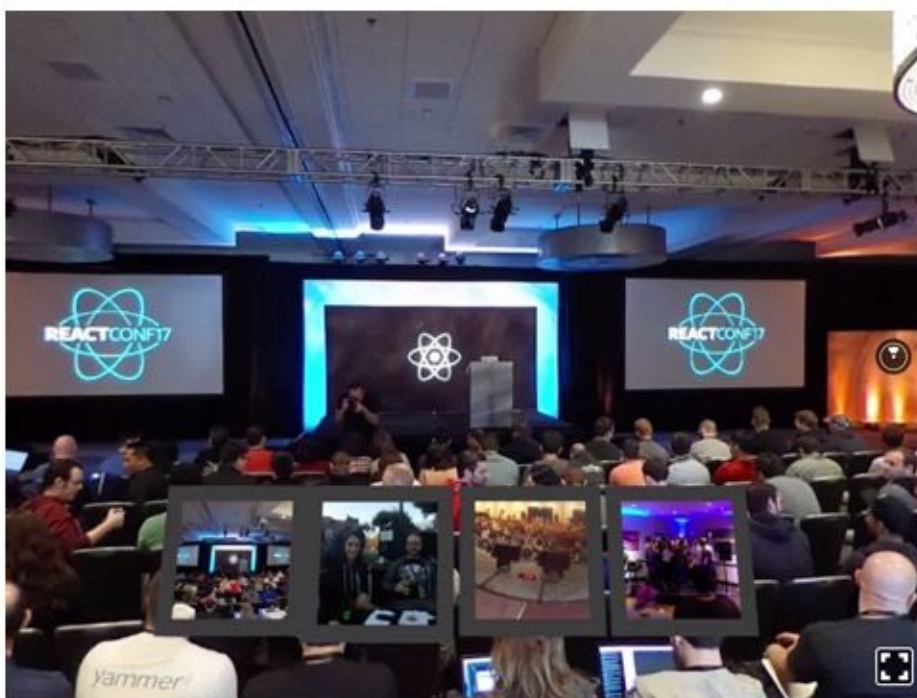
Warnings highlight any gotchas that are likely to trip you up along the way.

# Chapter 1: Build a Full-Sphere 3D Image Gallery with React VR

by Michaela Lehr

[React VR](#) is a JavaScript library by Facebook that reduces the effort of creating a [WebVR](#) application. You may compare React VR with [A-Frame](#) by Mozilla, but instead of writing HTML, with React VR we're using JavaScript to create a WebVR scene.

React VR is built on the WebGL library [three.js](#) and the React Native framework. This means that we're able to use JSX tags, React Native components, like `<View>` or `<Text>`, or React Native concepts, like the flexbox layout. To simplify the process of creating a WebVR scene, React VR has built-in support for 3D meshes, lights, videos, 3D shapes, or spherical images.



In this chapter, we want to use React VR to build a viewer for spherical images. For this, we'll use four [equirectangular](#) photos, which I shot at [React Conf 2017](#) with my [Theta S camera](#). The gallery will have four buttons to swap the images, which will work with the mouse and/or VR headset. You can download the equirectangular images as well as the button graphics [here](#). Last but not least, we'll take a look at how animations work with React VR by adding a simple button transition.

For development, we're using a browser like Chrome on the desktop. To check if the stereoscopic rendering for VR devices works, we're using a Samsung phone with Gear VR. In theory, any mobile browser capable of WebVR should be able to render our app in a stereoscopic way for the usage with GearVR, Google Cardboard, or even Google Daydream. But the library, as well as the API, are still under development, so the support may not be reliable. [Here's a good summary](#) of browsers currently supporting WebVR features.

## Development Setup and Project Structure

Let's start by installing the React VR CLI tool. Then create a new React VR project with all its dependencies in a new folder called GDVR\_REACTVR\_SITEPOINT\_GALLERY:

```
npm install -g react-vr-cli
react-vr init GDVR_REACTVR_SITEPOINT_GALLERY
cd GDVR_REACTVR_SITEPOINT_GALLERY
```

To start a local development server, we'll run an npm script and browse to <http://localhost:8081/vr/> in Chrome.

```
npm start
```

If you see a black and white room with stairs, pillars, and a “hello” text plane, everything's correct.



The most important files and folders scaffolded by the React VR CLI are:

- `index.vr.js`. This is the entry point of the application. Currently, the file

contains the whole source code of React VR's default scene, as we already saw in the browser.

- `static_assets`. This folder should contain all assets used in the application. We'll put the equirectangular images and the button graphics in this folder.

We want our project to have three components:

- a **Canvas** component, which holds the code for the full-sphere images
- a **Button** component, which creates a VR button to swap the images
- a **UI** component, which builds a UI out of four Button components.

The three components will each have their own file, so let's create a `components` folder to contain these files. Then, before we start creating the Canvas component, let's remove the scaffolded example code from the `index.vr.js` file so it looks like this:

```
/* index.vr.js */
import React from 'react';
import {
  AppRegistry,
  View,
} from 'react-vr';

export default class GDVR_REACTVR_SITEPOINT_GALLERY extends
React.Component {
  render() {
    return (
      <View>
      </View>
    );
  }
};

AppRegistry.registerComponent('GDVR_REACTVR_SITEPOINT_GALLERY', ()
=> GDVR_REACTVR_SITEPOINT_GALLERY);
```

## Adding a Spherical Image to the Scene

To add a spherical image to the scene, we'll create a new file `Canvas.js` in the `components` folder:

```
/* Canvas.js */
import React from 'react';
import {
  asset,
  Pano,
} from 'react-vr';

class Canvas extends React.Component {

  constructor(props) {
    super(props);

    this.state = {
      src: this.props.src,
    }
  }

  render() {
    return (
      <Pano source={asset(this.state.src)} />
    );
  }
};

export default Canvas;
```

In the first six lines of code, we import the dependencies. Then we declare our `Canvas` component and define how it renders by using the JSX syntax.

### More on JSX

If you want to learn more about JSX, I recommend you check out ["Getting Started with React and JSX"](#).

A look at the JSX code reveals that the `Canvas` component returns only one component, the React VR `<Pano>` component. It has a parameter, the `source` prop, that uses an `asset` function to load the image from the `static_assets` folder. The argument refers to a state, which we initialized in the constructor

function.

In our case, we don't want to define the path in the Canvas component itself, but use the `index.vr.js` file to define all image paths. This is why the `state.src` object refers to the component's props object.

## More on State and Props

Check out the ReactJS documentation for [React.Component](#) if you would like to know more about state and props.

Let's continue by modifying the `index.vr.js` file to use the Canvas component and render it to the scene:

```
/* index.vr.js */
import React from 'react';
import {
  AppRegistry,
  View,
} from 'react-vr';
import Canvas from '../components/Canvas';

export default class GDVR_REACTVR_SITEPOINT_GALLERY extends
React.Component {

  constructor() {
    super();

    this.state = {
      src: 'reactconf_00.jpg',
    };
  }

  render() {
    return (
      <View>
        <Canvas
          src={this.state.src}
        />
      </View>
    );
  }
};

AppRegistry.registerComponent('GDVR_REACTVR_SITEPOINT_GALLERY', ()
=> GDVR_REACTVR_SITEPOINT_GALLERY);
```



Besides the already used React VR dependencies, we need to import our custom Canvas component. Next, we declare the application class in line six:

```
/* index.vr.js */  
import Canvas from './components/Canvas';
```

Then, we add the <Canvas> component as a child component of the <View> component. We're using src as the component's prop because we're referring to it in the Canvas component. A look in the browser should now show the panoramic image, and we should already be able to interact with it.



## Create a UI Component to Hold Four Buttons

What we want to do now is to create four buttons that a user can trigger to swap the images. So we'll add two new components: a UI component, and its child component, a Button component. Let's start with the Button component:

```
/* Button.js */

import React from 'react';

import {

  asset,

  Image,

  View,

  VrButton,

} from 'react-vr';

class Button extends React.Component {
```

```
onButtonClick = () => {
```

```
  this.props.onClick();
```

```
}
```

```
render () {
```

```
  return (
```

```
    <View
```

```
      style={{
```

```
        alignItems: 'center',
```

```
        flexDirection: 'row',
```

```
        margin: 0.0125,
```

```
        width: 0.7,
```

```
      }}
    )
  )
}
```

```
>
```

```
<VrButton
```

```
  onClick={this.onButtonClick}
```

```
>
```

```
<Image
```

```
  style={{
```

```
    width: 0.7,
```

```
    height: 0.7,
```

```
  }}  
  source={asset(this.props.src)}
```

```
>
```

```
</Image>
```

```
</VrButton>
```

```

    </View>

    );

}

};

export default Button;

```

To build the button, we're using React VR's `<VrButton>` component, which we import in line six. Also, we're using an image component to add our asset images to each button, since the `<VrButton>` component itself has no appearance. Like before, we're using a prop to define the image source. Another feature we're using twice in this component is the `style` prop, to add layout values to each button and its image. The `<VrButton>` also makes use of an event listener, `onClick`.

To add four Button components to our scene, we'll use the UI parent component, which we'll add as a child in `index.vr.js` afterward. Before writing the UI component, let's create a config object defining the relation between the equirectangular images, the button images, and the buttons themselves. To do this, we declare a constant right after the import statements in the `index.vr.js` file:

```

/* index.vr.js */

const Config = [

```

```
{  
  
  key: 0,  
  
  imageSrc: 'reactconf_00.jpg',  
  
  buttonImageSrc: 'button-00.png',  
  
},  
  
{  
  
  key: 1,  
  
  imageSrc: 'reactconf_01.jpg',  
  
  buttonImageSrc: 'button-01.png',  
  
},  
  
{  
  
  key: 2,  
  
  imageSrc: 'reactconf_02.jpg',
```

```
      buttonImageSrc: 'button-02.png',

    },

    {

      key: 3,

      imageSrc: 'reactconf_03.jpg',

      buttonImageSrc: 'button-03.png',

    }

  ];
```

The UI component will use the values defined in the config to handle the gaze and click events:

```
/* UI.js */

import React from 'react';

import {

  View,
```

```
} from 'react-vr';
```

```
import Button from './Button';
```

```
class UI extends React.Component {
```

```
  constructor(props) {
```

```
    super(props);
```

```
    this.buttons = this.props.buttonConfig;
```

```
  }
```

```
  render () {
```

```
    const buttons = this.buttons.map((button) =>
```



```
<Button

  key={button.key}

  onClick={()=>{

    this.props.onClick(button.key);

  }}

  src={button.buttonImageSrc}

/>

);
```

```
return (
```

```
<View

  style={{

    flexDirection: 'row',
```

```
      flexWrap: 'wrap',

      transform: [

        {rotateX: -12},

        {translate: [-1.5, 0, -3]},

      ],

      width: 3,

    }}

  >

    {buttons}

  </View>

);

}

};
```

```
export default UI;
```

To set the source of an image, we're using the config values we already added to the `index.vr.js` file. We're also using the prop `onClick` to handle the click event, which we'll also add in a few moments to the `index.vr.js` file. Then we create as many buttons as defined in the button config object, to add them later in the JSX code that will be rendered to the scene:

```
/* UI.js */

const buttons = this.buttons.map((button) =>

  <Button

    key={button.key}

    onClick={()=>{

      this.props.onClick(button.key);

    }}

    src={button.buttonImageSrc}

  />
```

```
);
```

Now, all we have to do is add the UI component to the scene defined in the `index.vr.js` file. So we import the UI component right after importing the Canvas component:

```
/* index.vr.js */  
  
import UI from './components/UI';
```

Next, we add the `<Canvas>` component to the scene:

```
/* index.vr.js */  
  
<View>  
  
  <Canvas  
  
    src={this.state.src}  
  
  />  
  
  <UI  
  
    buttonConfig={Config}  
  
    onClick={(key)=>{
```

```
        this.setState({src: Config[key].imageSrc});

    }}

/>

</View>
```

When checking this code in the browser, you'll notice that the click doesn't trigger an image source swap at the moment. To listen for updated props, we'll have to add another function to the Canvas component right after the constructor function.

## Component Lifecycle

If you're interested in the lifecycle of a React component, you might want to read about [React.Component in the React docs](#).

```
/* Canvas.js */

componentWillReceiveProps(nextProps) {

    this.setState({src: nextProps.src});

}
```

A test in the browser should now be successful, and a click on a button image should change the spherical image.



## Add Animations for Button State Transitions

To make the buttons more responsive to user interactions, we want to add some hover states and transitions between the default idle and the hover state. To do this, we'll use the [Animated library](#) and [Easing functions](#), and then write to functions for each transition: `animateIn` and `animateOut`:

```
/* Button.js */

import React from 'react';

import {

  Animated,

  asset,

  Image,

  View,

  VrButton,

} from 'react-vr';

const Easing = require('Easing');
```

```
class Button extends React.Component {

  constructor(props) {

    super();

    this.state = {

      animatedTranslation: new Animated.Value(0),

    };

  }

  animateIn = () => {

    Animated.timing(
```



```
    this.state.animatedTranslation,  
  
    {  
  
      toValue: 0.125,  
  
      duration: 100,  
  
      easing: Easing.in,  
  
    }  
  
  ).start();  
  
}
```

```
animateOut = () => {  
  
  Animated.timing(  
  
    this.state.animatedTranslation,  
  
    {
```

```
        toValue: 0,  
  
        duration: 100,  
  
        easing: Easing.in,  
  
    }  
  
    ).start();  
  
}
```

```
onButtonClick = () => {  
  
    this.props.onClick();  
  
}
```

```
render () {  
  
    return (  

```

```
<Animated.View

  style={{

    alignItems: 'center',

    flexDirection: 'row',

    margin: 0.0125,

    transform: [

      {translateZ: this.state.animatedTranslation},

    ],

    width: 0.7,

  }}

>

<VrButton

  onClick={this.onButtonClick}
```

```
        onEnter={this.animateIn}

        onExit={this.animateOut}

    >

    <Image

        style={{

            width: 0.7,

            height: 0.7,

        }}

        source={asset(this.props.src)}

    >

    </Image>

    </VrButton>

</Animated.View>
```

```
    );  
  
    }  
  
};  
  
export default Button;
```

After adding the dependencies, we define a new state to hold the translation value we want to animate:

```
/* Button.js */  
  
constructor(props) {  
  
    super();  
  
    this.state = {  
  
        animatedTranslation: new Animated.Value(0),  
  
    };  
};
```

```
}
```

Next, we define two animations, each in a separate function, that describe the animation playing when the cursor enters the button, and when the cursor exits the button:

```
/* Button.js */

animateIn = () => {

  Animated.timing(

    this.state.animatedTranslation,

    {

      toValue: 0.125,

      duration: 100,

      easing: Easing.in,

    }

  ).start();

}
```

```
animateOut = () => {  
  
  Animated.timing(  
  
    this.state.animatedTranslation,  
  
    {  
  
      toValue: 0,  
  
      duration: 100,  
  
      easing: Easing.in,  
  
    }  
  
  ).start();  
  
}
```

To use the `state.animatedTranslation` value in the JSX code, we have to make the `<View>` component animatable, by adding `<Animated.view>`:

```
/* Button.js */
```

```
<Animated.View

  style={{

    alignItems: 'center',

    flexDirection: 'row',

    margin: 0.0125,

    transform: [

      {translateZ: this.state.animatedTranslation},

    ],

    width: 0.7,

  }}

>
```

We'll call the function when the event listeners `onButtonEnter` and `onButtonExit` are triggered:

```
/* Button.js */
```



```
<VrButton  
  
  onClick={this.onButtonClick}  
  
  onEnter={this.animateIn}  
  
  onExit={this.animateOut}  
  
>
```

A test of our code in the browser should show transitions between the position on the z-axis of each button:



## Building and Testing the Application

Open your app in a browser that supports WebVR and navigate to your development server, by using not `http://localhost:8081/vr/index.html`, but your IP address, for example, `http://192.168.1.100:8081/vr/index.html`. Then, tap on the **View in VR** button, which will open a full-screen view and start the stereoscopic rendering.



To upload your app to a server, you can run the npm script `npm run bundle`, which will create a new `build` folder within the `vr` directory with the compiled files. On your web server you should have the following directory structure:

```
Web Server
├── static_assets/
├── index.html
├── index.bundle.js
├── client.bundle.js
```

## Further Resources

### Full Project Code

This is all we had to do create a small WebVR application with React VR. You can find the [entire project code on GitHub](#).

This is all we had to do create a small WebVR application with React VR. React VR has a few more components we didn't discuss in this tutorial:

- There's a `Text` component for rendering text.
- Four different light components can be used to add light to a scene: `AmbientLight`, `DirectionalLight`, `PointLight`, and `Spotlight`.
- A `Sound` component adds spatial sound to a location in the 3D scene.
- To add videos, the `Video` component or the `VideoPano` component can be used. A special `VideoControl` component adds controls for video playback and volume.
- With the `Model` component we can add 3D models in the obj format to the application.
- A `CylindricalPanel` component can be used to align child elements to the inner surface of a cylinder — for example, to align user interface elements.
- There are three components to create 3D primitives: a sphere component, a plane component and a box component.

Also, React VR is still under development, which is also the reason for it running only in the Carmel Developer Preview browser. If you're interested in learning more about React VR, here are a few interesting resources:

- [React VR Docs](#)
- [React VR on GitHub](#)
- [Awesome React VR](#), a collection of React VR resources.

And if you'd like to dig deeper in WebVR in general, these articles might be right for you:

- [“A-Frame: The Easiest Way to Bring VR to the Web Today”](#)
- [“Embedding Virtual Reality Across the Web with VR Views”](#)

# Chapter 2: Build a WebRTC Video Chat Application with SimpleWebRTC

by Michael Wanyoike

With the advent of WebRTC and the increasing capacity of browsers to handle peer-to-peer communications in real time, it's easier than ever to build real-time applications. In this tutorial, we'll take a look at [SimpleWebRTC](#) and how it can make our lives easier when implementing WebRTC. Throughout the chapter, we'll be building a WebRTC video chat app with messaging features.

If you need a bit of a background regarding WebRTC and peer-to-peer communication, I recommend reading [The Dawn of WebRTC](#) and [Introduction to the getUserMedia API](#).

# What is SimpleWebRTC

Before we move on, it's important that we understand the main tool that we'll be using. [SimpleWebRTC](#) is a JavaScript library that simplifies WebRTC peer-to-peer data, video, and audio calls.

SimpleWebRTC acts as a wrapper around the browser's WebRTC implementation. As you might already know, browser vendors don't exactly agree on a single way of implementing different features, which means that for every browser there's a different implementation for WebRTC. As the developer, you'd have to write different code for every browser you plan to support. SimpleWebRT acts as the wrapper for that code. The API that it exposes is easy to use and understand, which makes it a really great candidate for implementing cross-browser WebRTC.

# Building the WebRTC Video Chat App

Now it's time to get our hands dirty by building the app. We'll build a single page application that runs on top of an Express server.

## Running the Examples

Please note that you can download the code for this tutorial from our [GitHub repo](#). To run it, or to follow along at home, you'll need to have Node and npm installed. If you're not familiar with these, or would like some help getting them installed, check out our previous tutorials:

- [Install Multiple Versions of Node.js using nvm](#)
- [A Beginner's Guide to npm — the Node Package Manager](#)

You also need a PC or laptop that has a webcam. If not, you'll need to get yourself a USB webcam that you can attach to the top of your monitor. You'll probably need a friend or a second device to test remote connections.

# Dependencies

We'll be using the following dependencies to build our project:

- [SimpleWebRTC](#) — the WebRTC library
- [Semantic UI CSS](#) — an elegant CSS framework
- [jQuery](#) — used for selecting elements on the page and event handling.
- [Handlebars](#) — a JavaScript templating library, which we'll use to generate HTML for the messages
- [Express](#) — NodeJS server.

## Project Setup

Go to your workspace and create a folder `simplewebrtc-messenger`. Open the folder in VSCode or your favorite editor and create the following files and folder structure:

```
simplewebrtc-messenger
├── public
│   ├── images
│   │   └── image.png
│   ├── index.html
│   └── js
│       └── app.js
├── README.md
└── server.js
```

Or, if you prefer, do the same via the command line:

```
mkdir -p simplewebrtc-messenger/public/{images,js}
cd simplewebrtc-messenger
touch public/js/app.js public/index.html .gitignore README.md
server.js
```

Open `README.md` and copy the following content:

```
# Simple WebRTC Messenger

A tutorial on building a WebRTC video chat app using SimpleWebRTC.
```

Add the line `node_modules` to the `.gitignore` file if you plan to use a git repository. Generate the `package.json` file using the following command:

```
npm init -y
```

You should get the following output:

```
{
  "name": "simplewebrtc-messenger",
  "version": "1.0.0",
  "description": "A tutorial on building a WebRTC video chat app using SimpleWebRTC.",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
  }
}
```



```
    "start": "node server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Now let's install our dependencies:

```
npm install express handlebars jquery semantic-ui-css simplewebrtc
```

As the installation proceeds, copy this code to `server.js`:

```
const express = require('express');

const app = express();
const port = 3000;

// Set public folder as root
app.use(express.static('public'));

// Provide access to node_modules folder from the client-side
app.use('/scripts', express.static(`${__dirname}/node_modules/`));

// Redirect all traffic to index.html
app.use((req, res) =>
  res.sendFile(`${__dirname}/public/index.html`));

app.listen(port, () => {
  console.info('listening on %d', port);
});
```

The server code is pretty standard. Just read the comments to understand what's going on.

Next, let's set up our `public/index.html` file:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="scripts/semantic-ui-
css/semantic.min.css">
  <title>SimpleWebRTC Demo</title>
```

```

<style>
  html { margin-top: 20px; }
  #chat-content { height: 180px; overflow-y: scroll; }
</style>
</head>
<body>
  <!-- Main Content -->
  <div class="ui container">
    <h1 class="ui header">Simple WebRTC Messenger</h1>
    <hr>
  </div>

  <!-- Scripts -->
  <script src="scripts/jquery/dist/jquery.min.js"></script>
  <script src="scripts/semantic-ui-css/semantic.min.js"></script>
  <script src="scripts/handlebars/dist/handlebars.min.js ">
</script>
  <script src="scripts/simplewebrtc/out/simplewebrtc-with-
adapter.bundle.js"></script>
  <script src="js/app.js"></script>
</body>
</html>

```

Next, let's set up our base client-side JavaScript code. Copy this code to public/js/app.js:

```

window.addEventListener('load', () => {
  // Put all client-side code here
});

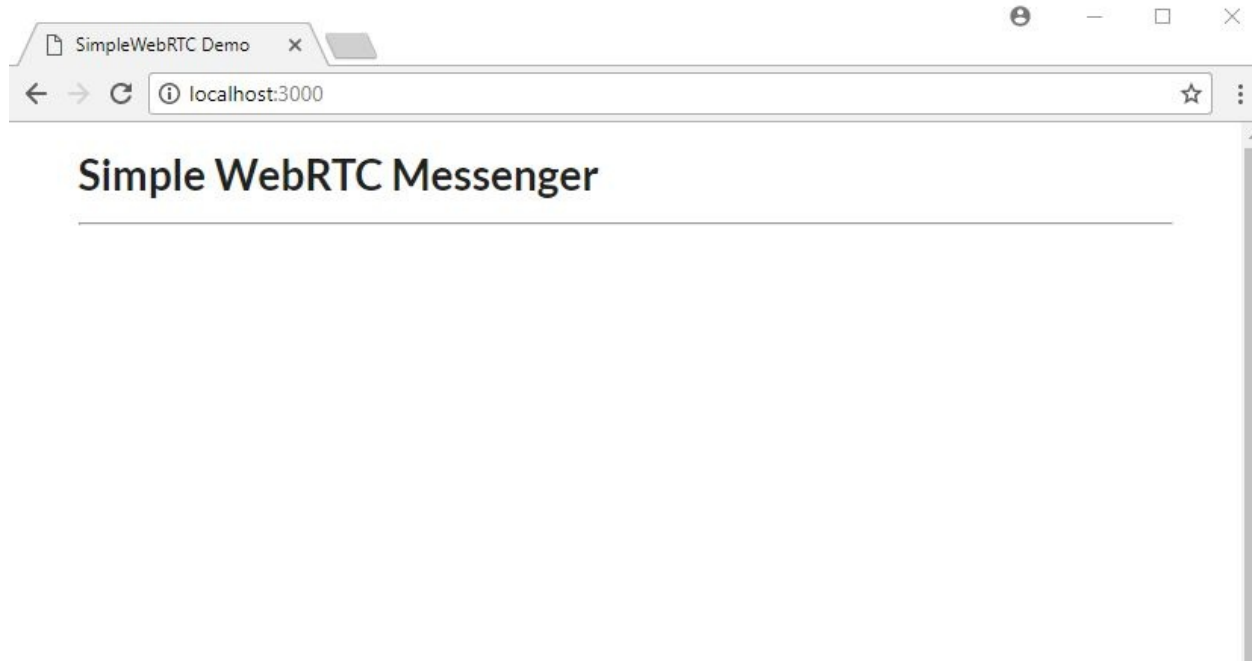
```

Finally, download this [image](#) from our GitHub repository and save it inside the public/images folder.

Now we can run our app:

```
npm start
```

Open the URL [localhost:3000](http://localhost:3000) in your browser and you should see the following:



## Markup

Let's now work on `public/index.html`. For the sake of simplicity (especially if you're already familiar with Handlebars) you can copy the entire markup code from our [GitHub repository](#). Otherwise, let's go through things step by step. First off, copy this code and place it after the `<hr>` tag within the `ui` container div:

```
<div class="ui two column stackable grid">

  <!-- Chat Section -->

  <div class="ui ten wide column"> <div class="ui segment"> <!-- Chat
  Room Form --> <div class="ui form">

    <div class="fields"> <div class="field"> <label>User Name</label>
    <input type="text" placeholder="Enter user name" id="username"
    name="username"> </div>

    <div class="field"> <label>Room</label> <input type="text"
    placeholder="Enter room name" id="roomName" name="roomName"> </div>

  </div>

  <br>

  <div class="ui buttons"> <div id="create-btn" class="ui submit
  orange button">Create Room</div> <div class="or"></div> <div
  id="join-btn" class="ui submit green button">Join Room</div> </div>

</div>

<!-- Chat Room Messages --> <div id="chat"></div> </div>

</div>

<!-- End of Chat Section -->

<!-- Local Camera -->

<div class="ui six wide column"> <h4 class="ui center aligned
header" style="margin:0;"> Local Camera

</h4>

 <video id="local-video" class="ui large
image hidden" autoplay></video> </div>
```

```
</div>
```

```
<!-- Remote Cameras -->
```

```
<h3 class="ui center aligned header">Remote Cameras</h3> <div  
id="remote-videos" class="ui stackable grid"> <div class="four wide  
column">  </div>
```

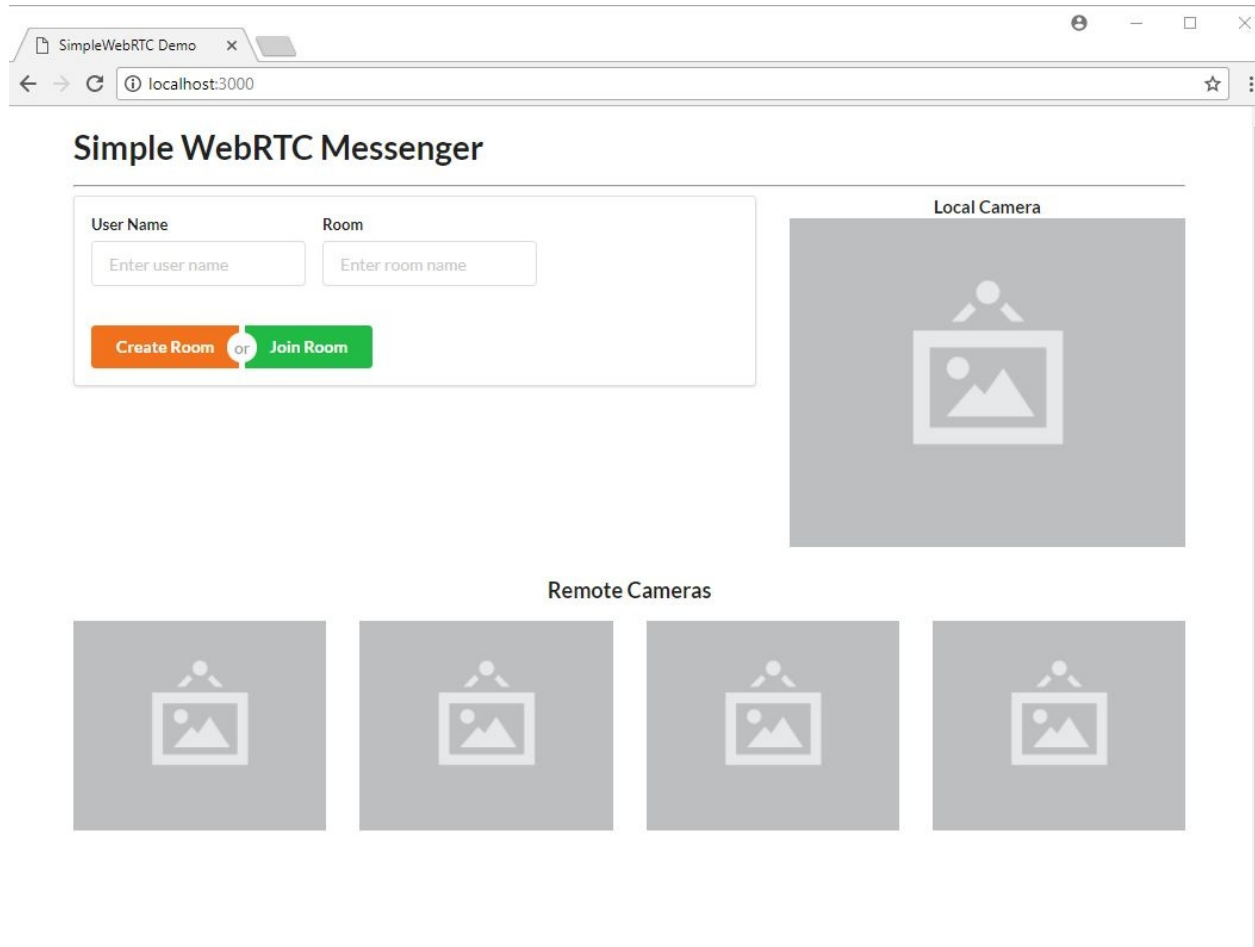
```
<div class="four wide column">  </div>
```

```
<div class="four wide column">  </div>
```

```
<div class="four wide column">  </div>
```

```
</div>
```

Go through the markup code and read the comments to understand what each section is for. Also check out the [Semantic UI](#) documentation if you're unfamiliar with the CSS library. Refresh your browser. You should have the following view:



We're using a blank image as a placeholder to indicate where the camera location will stream to on the web page. Take note that this app will be able to support multiple remote connections, provided your internet bandwidth can handle it.

# Templates

Now let's add the three Handlebar templates that will make our web page interactive.

Place the following markup right after the `ui container` div (although the location doesn't really matter). We'll start off with the chat container, which simply is made up of:

- Room ID
- empty chat messages container (to be populated later via JavaScript)
- input for posting messages.

```
<!-- Chat Template --> <script id="chat-template" type="text/x-handlebars-template"> <h3 class="ui orange header">Room ID ->
<strong>{{ room }}</strong></h3> <hr>

<div id="chat-content" class="ui feed"> </div> <hr>

<div class="ui form">

    <div class="ui field"> <label>Post Message</label> <textarea
id="post-message" name="post-message" rows="1"></textarea> </div>

    <div id="post-btn" class="ui primary submit button">Send</div>
</div>

</script>
```

Next, add the following template, which will be used to display user chat messages: <!-- Chat Content Template --> <script id="chat-content-template" type="text/x-handlebars-template"> {{#each messages}}

```
<div class="event">

<div class="label"> <i class="icon blue user"></i> </div>

<div class="content"> <div class="summary"> <a href="#"> {{
username }}</a> posted on <div class="date"> {{ postedOn }}

</div>

</div>

<div class="extra text"> {{ message }}

</div>

</div>

</div>

{{/each}}

</script>
```

Finally, add the following template, which will be used to display streams from a remote camera: <!-- Remote Video Template --> <script id="remote-video-template" type="text/x-handlebars-template"> <div id="{{ id }}" class="four wide column"></div> </script>

The markup code is hopefully pretty self-explanatory, so let's move on to writing the client-side JavaScript code for our application.



## Main App Script

Open the file `public/js/app.js` and add this code: // Chat platform

```
const chatTemplate = Handlebars.compile($('#chat-template').html());
const chatContentTemplate = Handlebars.compile($('#chat-content-template').html());
const chatEl = $('#chat');
const formEl = $('.form');
const messages = [];
let username; // Local Video
const localImageEl = $('#local-image');
const localVideoEl = $('#local-video');
// Remote Videos
const remoteVideoTemplate = Handlebars.compile($('#remote-video-template').html());
const remoteVideosEl = $('#remote-videos');
let remoteVideosCount = 0; // Add validation rules to Create/Join Room Form
formEl.form({ fields: { roomName: 'empty', username: 'empty', }, });
```

Here we're initializing several elements that we plan to manipulate. We've also added validation rules to the form so that a user can't leave either of the fields blank.

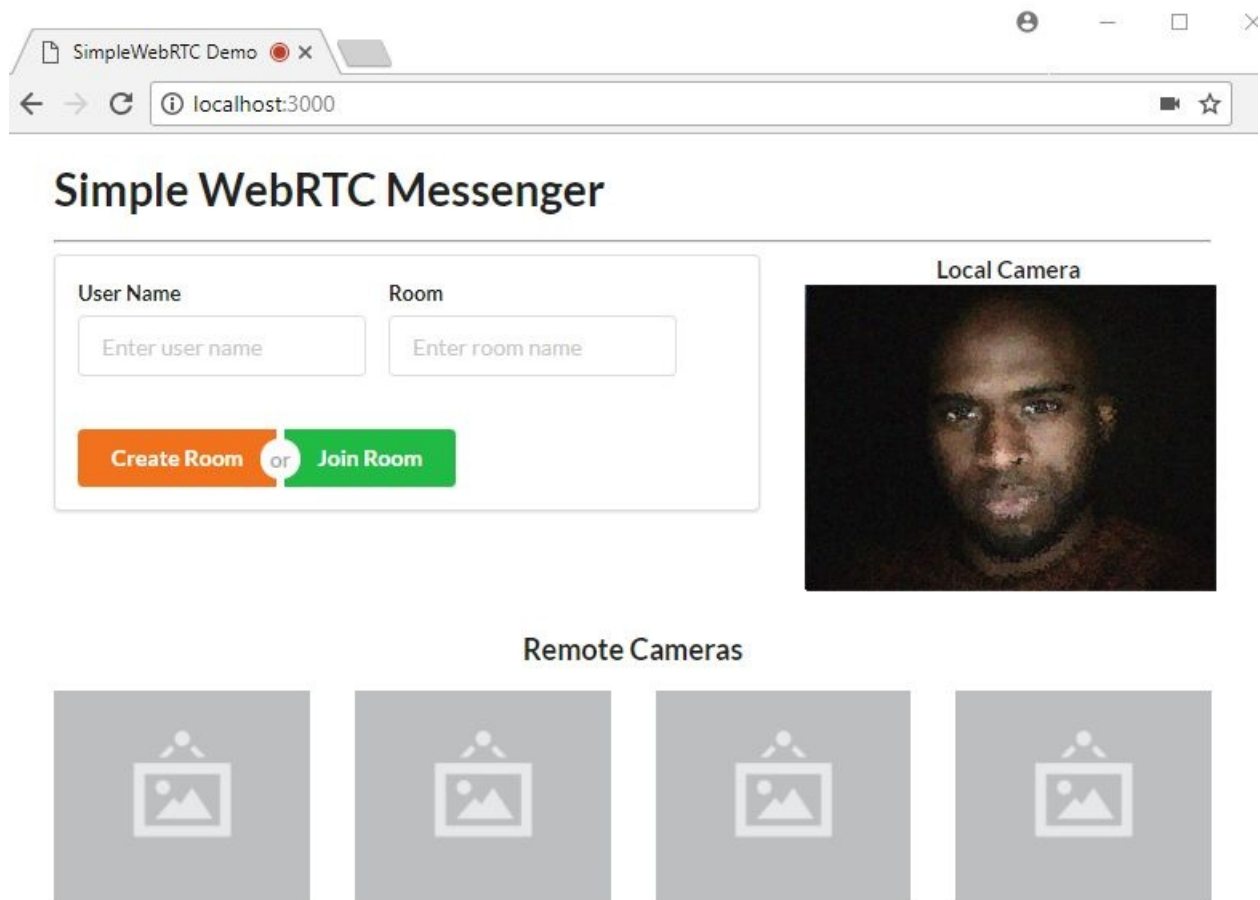
Next, let's initialize our WebRTC code:

```
// create our WebRTC connection
const webrtc = new SimpleWebRTC({
  // the id/element dom element that will hold "our" video
  localVideoEl: 'local-video',
  // the id/element dom element that will hold remote videos
  remoteVideosEl: 'remote-videos',
  // immediately ask for camera access
  autoRequestMedia: true,
});

// We got access to local camera
webrtc.on('localStream', () => {
  localImageEl.hide();
  localVideoEl.show();
});
```

Now you know why it's called SimpleWebRTC. That's all we need to do to initialize our WebRTC code. Noticed we haven't even specified any ICE servers or STUN servers. It just works. However, you can use other TURN services such as [Xirsys](#). You'll need to set up a local [SignalMaster](#) server for handling WebRTC signaling.

Let's do a quick refresh of the web page to confirm the new code is working:



The page should request access to your camera and microphone. Just click *Accept* and you should get the above view.

## Chat Room Script

Now let's make the form functional. We need to write logic for creating and joining a room. In addition, we need to write additional logic for displaying the chat room. We'll use the chat-room-template for this. Let's start by attaching click handlers to the form's buttons:

```
$('.submit').on('click', (event) => {  
  if (!formEl.form('is valid')) {  
    return false;  
  }  
  username = $('#username').val();  
  const roomName = $('#roomName').val().toLowerCase();  
  if (event.target.id === 'create-btn') {  
    createRoom(roomName);  
  } else {  
    joinRoom(roomName);  
  }  
  return false;  
});
```

Next, we need to declare the createRoom and joinRoom functions. Place the following code before the click handler code:

```
// Register new Chat Room  
const createRoom = (roomName) => {  
  console.info(`Creating new room: ${roomName}`);  
  webrtc.createRoom(roomName, (err, name) => {  
    showChatRoom(name);  
    postMessage(`${username} created chatroom`);  
  });  
};  
  
// Join existing Chat Room  
const joinRoom = (roomName) => {  
  console.log(`Joining Room: ${roomName}`);  
  webrtc.joinRoom(roomName);  
  showChatRoom(roomName);  
  postMessage(`${username} joined chatroom`);  
};
```

Creating or joining a room is as simple as that: just use [SimpleWebRTC's createRoom and joinRoom methods](#).

You may also have noticed that we have showChatroom and postMessage functions that we haven't defined yet. Let's do that now by inserting the following code before the calling code:

```
// Post Local Message
const postMessage = (message) => {
  const chatMessage = {
    username,
    message,
    postedOn: new Date().toLocaleString('en-GB'),
  };
  // Send to all peers
  webRTC.sendToAll('chat', chatMessage);
  // Update messages locally
  messages.push(chatMessage);
  $('#post-message').val('');
  updateChatMessages();
};

// Display Chat Interface
const showChatRoom = (room) => {
  // Hide form
  formEl.hide();
  const html = chatTemplate({ room });
  chatEl.html(html);
  const postForm = $('#form');
  // Post Message Validation Rules
  postForm.form({
    message: 'empty',
  });
  $('#post-btn').on('click', () => {
    const message = $('#post-message').val();
    postMessage(message);
  });
  $('#post-message').on('keyup', (event) => {
    if (event.keyCode === 13) {
      const message = $('#post-message').val();
      postMessage(message);
    }
  });
};
```

Take some time to go through the code to understand the logic. You'll soon come across another function we haven't declared, updateChatMessages. Let's add it now:

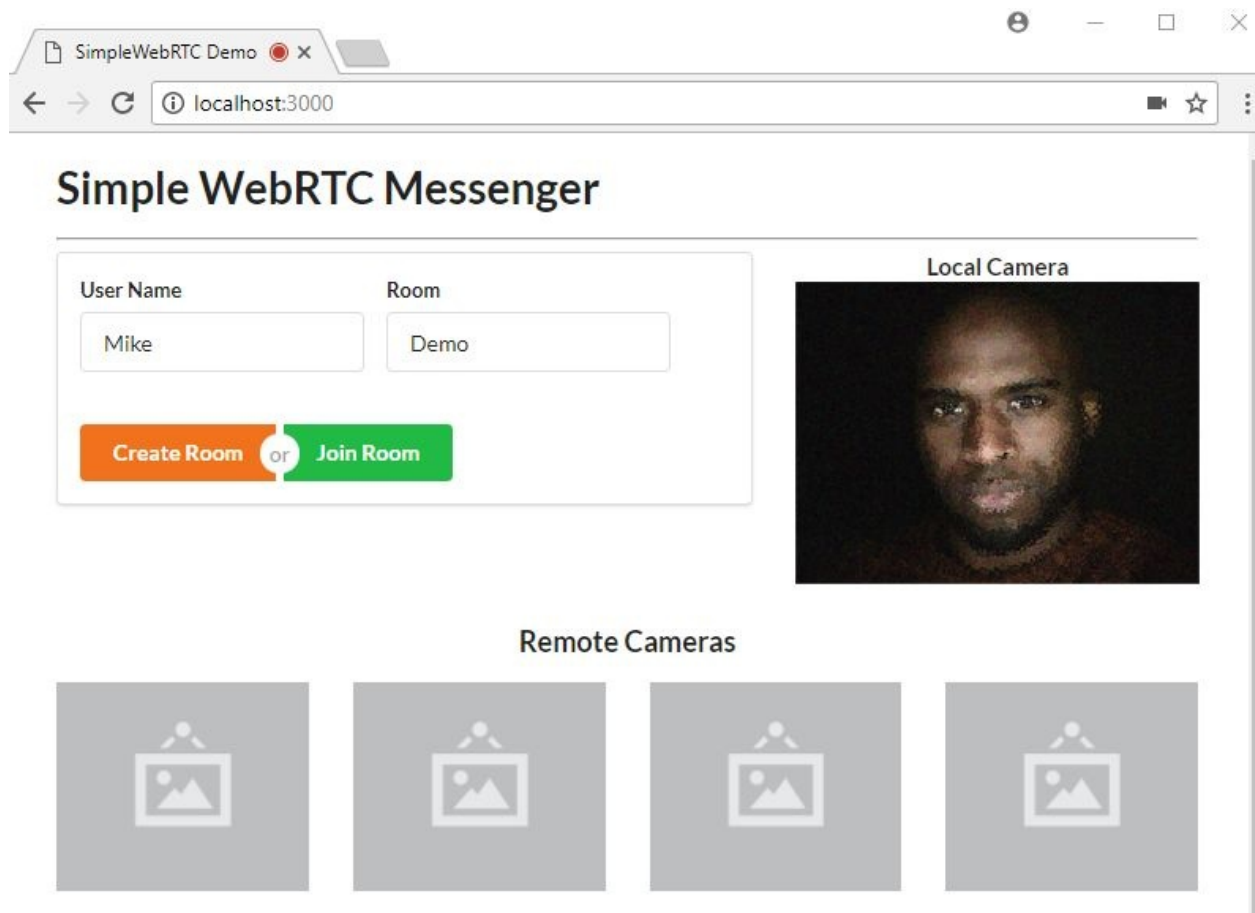
```
// Update Chat Messages
const updateChatMessages = () => {
```

```
const html = chatContentTemplate({ messages });
const chatContentEl = $('#chat-content');
chatContentEl.html(html);
// automatically scroll downwards
const scrollHeight = chatContentEl.prop('scrollHeight');
chatContentEl.animate({ scrollTop: scrollHeight }, 'slow');
};
```

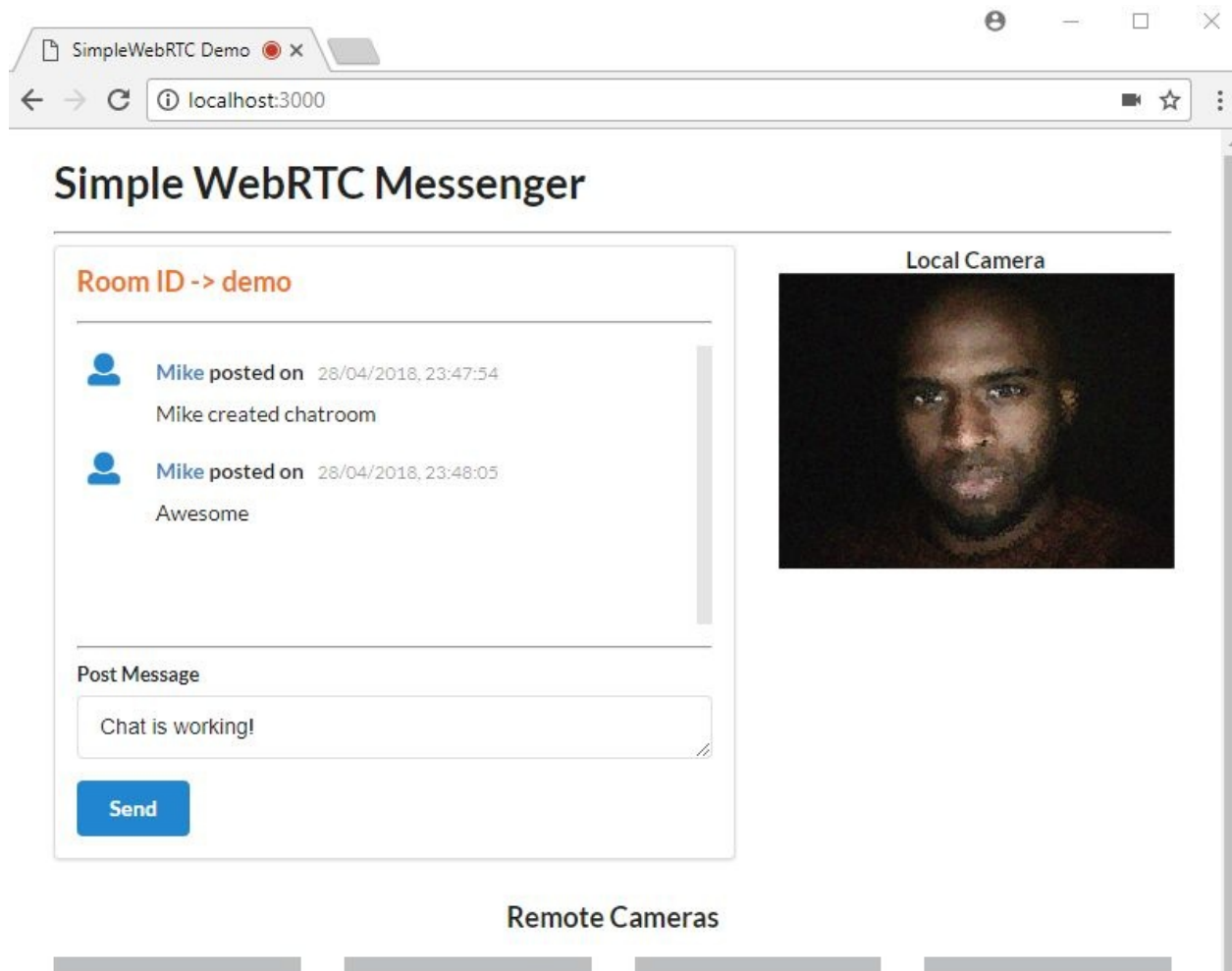
The purpose of this function is simply to update the Chat UI with new messages. We need one more function that accepts messages from remote users. Add the following function to `app.js`:

```
// Receive message from remote user
webRTC.connection.on('message', (data) => {
  if (data.type === 'chat') {
    const message = data.payload;
    messages.push(message);
    updateChatMessages();
  }
});
```

That's all the logic we need to have the chat room work. Refresh the page and log in:



Hit the *Create Room* button. You'll be taken to this view. Post some messages to confirm the chat room is working.



Once you've confirmed it's working, move on to the next task.

## Remote Video Camera

As mentioned earlier, SimpleWebRTC supports multiple peers. Here's the code for adding remote video streams when a new user joins a room: // Remote video was added webrtc.on('videoAdded', (video, peer) => { const id = webrtc.getDomId(peer); const html = remoteVideoTemplate({ id }); if (remoteVideosCount === 0) { remoteVideosEl.html(html); } else { remoteVideosEl.append(html); } \$('#\${id}`).html(video); \$('#\${id} video').addClass('ui image medium'); // Make video element responsive remoteVideosCount += 1; });

That's it. I'm sorry if you were expecting something more complicated. What we did is simply add an event listener for `videoAdded`, the callback of which receives a video element that can be directly add to the DOM. It also receives a peer object that contains useful information about our peer connection, but in this case, we're only interested in the DOM element's ID.

Unfortunately, testing this bit of code isn't possible without running it on an HTTPS server. Theoretically, you can generate a self-signed certificate for your Express server in order to run the app within your internal network. But the bad news is that browsers won't allow you to access the webcam if the certificate isn't from a trusted authority.

The simplest solution to testing the above code is to deploy it to a public server that supports the HTTPS protocol.



# Deployment

This method that we're about to perform is one of the easiest ways to deploy a NodeJS app. All we have to do is first register an account with [now.sh](https://now.sh).

Simply choose the free plan. You'll need to provide your email address. You'll also need to verify your email address for your account to activate. Next, install now CLI tool on your system:

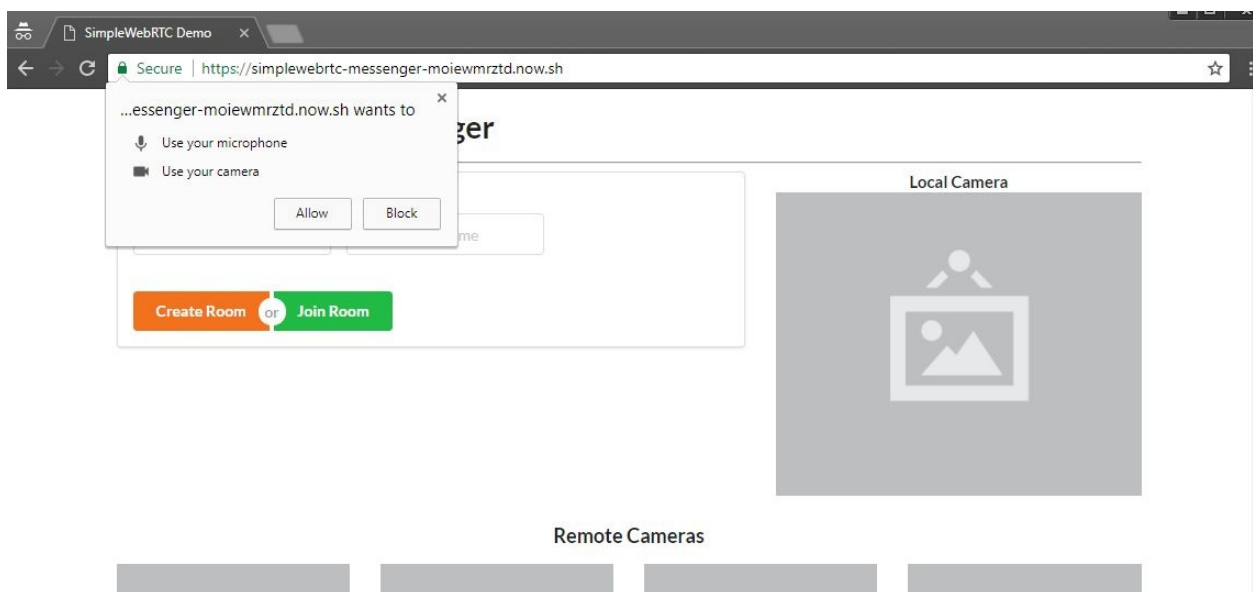
```
npm install -g now
```

After installation is complete, you can deploy the application. Simply execute the following command at the root of your project folder:

```
now --public
```

If this is the first time you're running the command, you'll be asked to enter your email address. You'll then receive an email that you'll need in order to verify your login. After verification has been done, you'll need to execute the command `now --public` again. After a few seconds, your app will be up and running at a specified URL that will be printed out on the terminal.




If you're using the VSCode integrated terminal, simply press ALT and click to open the URL in your browser.



You'll need to allow the page to access your camera and microphone. Next create a room just like before. After you've signed in, you need to access another device — such as another laptop or smartphone with a front-facing camera. You could also ask a friend with an internet connection to help you with this. Simply access the same URL, and enter a new username and the same room name. The remote user will have to hit the *Join Room* button. Within a few seconds, both devices should be connected to the chat room. If a device doesn't have a camera, that's okay, as the chat functionality will still work.

## Simple WebRTC Messenger

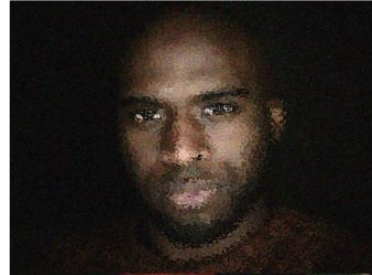
Room ID -> demo

-  **Mike** posted on 29/04/2018, 00:59:08  
Mike created chatroom
-  **Smith** posted on 29/04/2018, 00:59:55  
Hello
-  **Mike** posted on 29/04/2018, 01:00:16  
Hi smith

Post Message

Send

Local Camera



Remote Cameras



## Conclusion

In this tutorial, you've learned about SimpleWebRTC and how you can use it to create real-time apps. Specifically we've created a messaging application that allows the user to send text and make a video call to a remote peer.

SimpleWebRTC is a really great cross-browser library for painlessly implementing WebRTC in web applications.

### Full Code

Don't forget that the code used in this tutorial is available [on GitHub](#). Clone it, make something cool, and have fun!

# Chapter 3: Build a JavaScript Single Page App Without a Framework

by Michael Wanyoike

**Front-end frameworks are great. They abstract away much of the complexity of building a single-page application (SPA) and help you organize your code in an intelligible manner as your project grows.**

However, there's a flip side: these frameworks come with a degree of overhead and can introduce complexity of their own.

That's why, in this tutorial, we're going to learn how to build an SPA from scratch, without using a client-side JavaScript framework. This will help you evaluate what these frameworks actually do for you and at what point it makes sense to use one. It will also give you an understanding of the pieces that make up a typical SPA and how they're wired together.

Let's get started ...

## Prerequisites

For this tutorial, you'll need a fundamental knowledge of [modern JavaScript](#) and [jQuery](#). Some experience using [Handlebars](#), [Express](#) and [Axios](#) will come handy, though it's not strictly necessary. You'll also need to have the following setup in your environment:

- [Node.js](#)
- [Git](#) or [Git Bash](#) for Window users.

### Example Code

You can find the completed project on our [GitHub repository](#).

# Building the Project

We're going to build a simple currency application that will provide the following features:

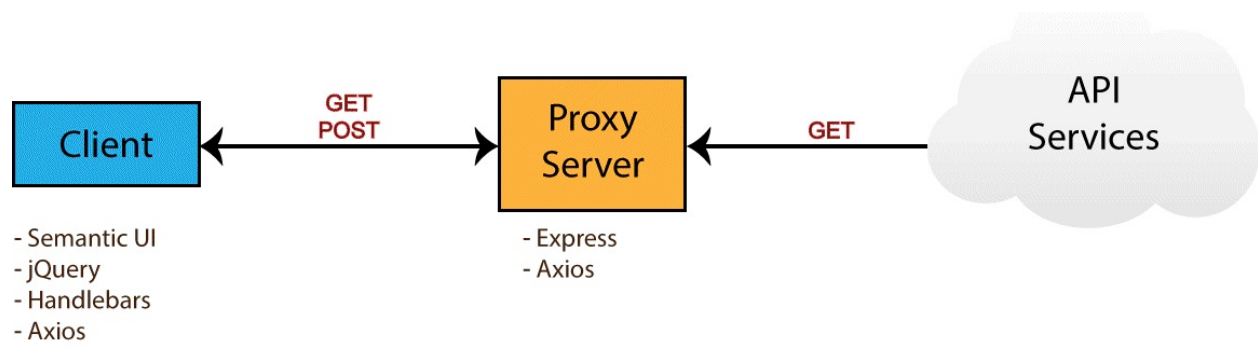
- display the latest currency rates
- convert from one currency to another
- display past currency rates based on a specified date.

We'll make use of the following free online REST APIs to implement these features:

- [fixer.io API](#)
- [Free Currency Converter API](#).

Fixer is a well-built API that provides a foreign exchange and currency conversion JSON API. Unfortunately, it's a commercial service and the free plan doesn't allow currency conversion. So we'll also need to use the Free Currency Converter API. The conversion API has a few limitations, which luckily won't affect the functionality of our application. It can be accessed directly without requiring an API key. However, Fixer requires an API key to perform any request. Simply [sign up on their website to get an access key for the free plan](#).

Ideally, we should be able to build the entire single-page application on the client side. However, since we'll be dealing with sensitive information (our API key) it won't be possible to store this in our client code. Doing so will leave our app vulnerable and open to any junior hacker to bypass the app and access data directly from our API endpoints. To protect such sensitive information, we need to put it in server code. So, we'll set up an [Express](#) server to act as a proxy between the client code and the cloud services. By using a proxy, we can safely access this key, since server code is never exposed to the browser. Below is a diagram illustrating how our completed project will work.



Take note of the npm packages that will be used by each environment — i.e. browser (client) and server. Now that you know what we'll be building, head over to the next section to start creating the project.



# Project Directories and Dependencies

Head over to your workspace directory and create the folder `single-page-application`. Open the folder in VSCode or your favorite editor and create the following files and folders using the terminal:

```
touch .env .gitignore README.md server.js
mkdir public lib
mkdir public/js
touch public/index.html
touch public/js/app.js
```

Open `.gitignore` and add these lines:

```
node_modules
.env
```

Open `README.md` and add these lines:

```
# Single Page Application

This is a project demo that uses Vanilla JS to build a Single Page Application.
```

Next, create the `package.json` file by executing the following command inside the terminal:

```
npm init -y
```

You should get the following content generated for you:

```
{
  "name": "single-page-application",
  "version": "1.0.0",
  "description": "This is a project demo that uses Vanilla JS to build a Single Page Application.",
  "main": "server.js",
  "directories": {
    "lib": "lib"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
}
```

```
"keywords": [],  
"author": "",  
"license": "ISC"  
}
```

See how convenient the npm command is? The content has been generated based on the project structure. Let's now install the core dependencies needed by our project. Execute the following command in your terminal:

```
npm install jquery semantic-ui-css handlebars vanilla-router  
express dotenv axios
```

After the packages have finished installing, head over to the next section to start building the base of the application.

## Application Base

Before we start writing our front-end code, we need to implement a server–client base to work from. That means a basic HTML view being served from an Express server. For performance and reliability reasons, we’ll inject front-end dependencies straight from the `node_modules` folder. We’ll have to set up our Express server in a special way to make this work. Open `server.js` and add the following: `require('dotenv').config(); // read .env files` `const express = require('express');`

```
const app = express();

const port = process.env.PORT || 3000;

// Set public folder as root
app.use(express.static('public'));

// Allow front-end access to node_modules folder
app.use('/scripts', express.static(`${__dirname}/node_modules/`));

// Listen for HTTP requests on port 3000
app.listen(port, () => {
  console.log('listening on %d', port); });
```

This gives us a basic Express server. I’ve commented the code, so hopefully this gives you a fairly good idea of what’s going on. Next, open `public/index.html` and enter: `<!DOCTYPE html> <html lang="en">`

```
<head>
```

```
<meta charset="UTF-8">
```

```
<meta name="viewport" content="width=device-width, initial-
scale=1.0"> <meta http-equiv="X-UA-Compatible" content="ie=edge">
<link rel="stylesheet" href="scripts/semantic-ui-
css/semantic.min.css"> <title>SPA Demo</title> </head>
```

```
<body>
```

```
<div class="ui container"> <!-- Navigation Menu --> <div class="ui
four item inverted orange menu"> <div class="header item"> <i
class="money bill alternate outline icon"></i> Single Page App
```

```

</div>

<a class="item" href="/"> Currency Rates
</a>

<a class="item" href="/exchange"> Exchange Rates
</a>

<a class="item" href="/historical"> Historical Rates
</a>

</div>

<!-- Application Root --> <div id="app"></div> </div>

```

```

<!-- JS Library Dependencies --> <script
src="scripts/jquery/dist/jquery.min.js"></script> <script
src="scripts/semantic-ui-css/semantic.min.js"></script> <script
src="scripts/axios/dist/axios.min.js"></script> <script
src="scripts/handlebars/dist/handlebars.min.js"></script> <script
src="scripts/vanilla-router/dist/vanilla-router.min.js"></script>
<script src="js/app.js"></script> </body>

</html>

```

We're using Semantic UI for styling. Please refer to the [Semantic UI Menu](#) documentation to understand the code used for our navigation bar. Go to your terminal and start the server: `npm start`

Open [localhost:3000](http://localhost:3000) in your browser. You should have a blank page with only the navigation bar showing:



Let's now write some view templates for our app.

## Front-end Skeleton Templates

We'll use [Handlebars](#) to write our templates. JavaScript will be used to render the templates based on the current URL. The first template we'll create will be for displaying error messages such as 404 or server errors. Place this code in `public/index.html` right after the the navigation section:

```
<!-- Error Template -->
<script id="error-template" type="text/x-handlebars-template">
  <div class="ui {{color}} inverted segment" style="height:250px;">
    <br>
    <h2 class="ui center aligned icon header">
      <i class="exclamation triangle icon"></i>
      <div class="content">
        {{title}}
        <div class="sub header">{{message}}</div>
      </div>
    </h2>
  </div>
</script>
```

Next, add the following templates that will represent a view for each URL path we specified in the navigation bar:

```
<!-- Currency Rates Template -->
<script id="rates-template" type="text/x-handlebars-template">
  <h1 class="ui header">Currency Rates</h1>
  <hr>
</script>

<!-- Exchange Conversion Template -->
<script id="exchange-template" type="text/x-handlebars-template">
  <h1 class="ui header">Exchange Conversion</h1>
  <hr>
</script>

<!-- Historical Rates Template -->
<script id="historical-template" type="text/x-handlebars-template">
  <h1 class="ui header">Historical Rates</h1>
  <hr>
</script>
```

Next, let's compile all theses templates in `public/js/app.js`. After compilation, we'll render the `rates-template` and see what it looks like:

```
window.addEventListener('load', () => {
  const el = $('#app');

  // Compile Handlebar Templates
  const errorTemplate = Handlebars.compile($('#error-
template').html());
  const ratesTemplate = Handlebars.compile($('#rates-
template').html());
  const exchangeTemplate = Handlebars.compile($('#exchange-
template').html());
  const historicalTemplate = Handlebars.compile($('#historical-
template').html());

  const html = ratesTemplate();
  el.html(html);
});
```

Take note that we're wrapping all JavaScript client code inside a load event. This is just to make sure that all dependencies have been loaded and that the DOM has completed loading. Refresh the page and see what we have:



We're making progress. Now, if you click the other links, except *Currency Rates*, the browser will try to fetch a new page and end up with a message like this: Cannot GET /exchange.

We're building a single page application, which means all the action should happen in one page. We need a way to tell the browser to stop fetching new pages whenever the URL changes.

## Client-side Routing

To control routing within the browser environment, we need to implement client-side routing. There are many client-side routing libraries that can help out with this. For our project, we'll use [vanilla router](#), which is a very easy-to-use routing package.

If you recall, we had earlier included all the JavaScript libraries we need in `index.html`. Hence we can call the Router class right away. Remove the last two statements you added to `app.js` and replace them with this code:

```
// Router Declaration
const router = new Router({
  mode: 'history',
  page404: (path) => {
    const html = errorTemplate({
      color: 'yellow',
      title: 'Error 404 - Page NOT Found!',
      message: `The path '${path}' does not exist on this site`,
    });
    el.html(html);
  },
});

router.add('/', () => {
  let html = ratesTemplate();
  el.html(html);
});

router.add('/exchange', () => {
  let html = exchangeTemplate();
  el.html(html);
});

router.add('/historical', () => {
  let html = historicalTemplate();
  el.html(html);
});

// Navigate app to current url
router.navigateTo(window.location.pathname);

// Highlight Active Menu on Refresh/Page Reload
const link = $('a[href$='${window.location.pathname}']');
link.addClass('active');
```

```

$('a').on('click', (event) => {
  // Block browser page load
  event.preventDefault();

  // Highlight Active Menu on Click
  const target = $(event.target);
  $('.item').removeClass('active');
  target.addClass('active');

  // Navigate to clicked url
  const href = target.attr('href');
  const path = href.substr(href.lastIndexOf('/'));
  router.navigateTo(path);
});

```

Take some time to go through the code. I've added comments in various sections to explain what's happening. You'll notice that, in the router's declaration, we've specified the `page404` property to use the error template. Let's now test the links:



The links should now work. But we have a problem. Click either the `/exchange` or `historical` link, then refresh the browser. We get the same error as before — `Cannot GET /exchange`. To fix this, head over to `server.js` and add this statement right before the `listen` code:

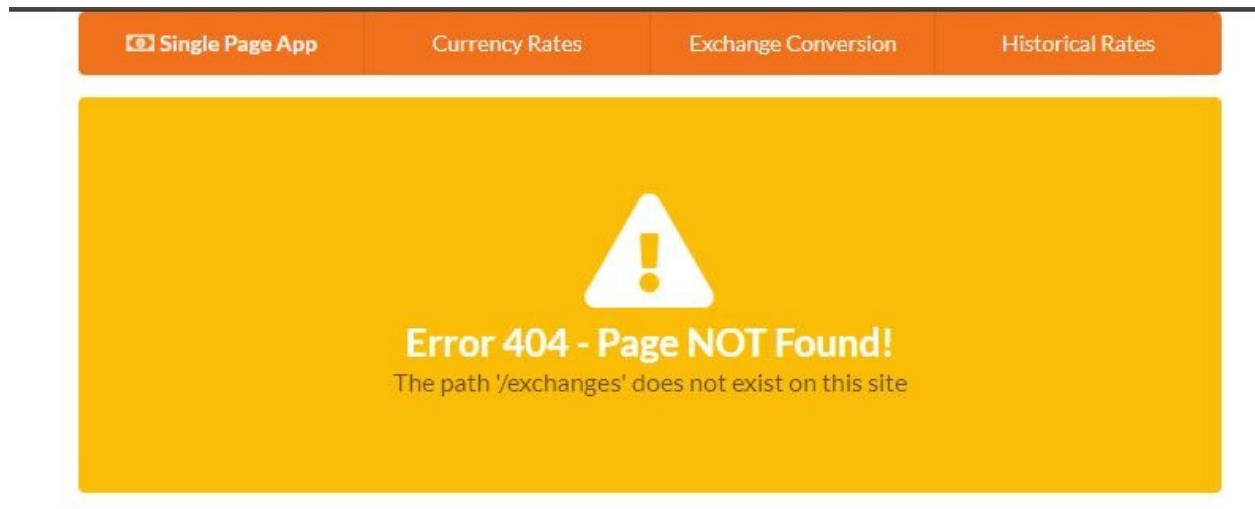
```

// Redirect all traffic to index.html
app.use((req, res) =>
  res.sendFile(`${__dirname}/public/index.html`));

```

You'll have to restart the the server using `Ctrl + C` and executing `npm start`. Go back to the browser and try to refresh. You should now see the page render correctly. Now, let's try entering a non-existent path in the URL like `/exchanges`. The app should display a 404 error message:





We've now implemented the necessary code to create our single-page-app skeleton. Let's now start working on listing the latest currency rates.

## Latest Currency Rates

For this task, we'll make use of the [Fixer Latest Rates Endpoint](#). Open the .env file and add your API key. We'll also specify the timeout period and the symbols we'll list on our page. Feel free to increase the timeout value if you have a slower internet connection: API\_KEY=<paste key here> PORT=3000

```
TIMEOUT=5000
```

```
SYMBOLS=EUR,USD,GBP,AUD,BTC,KES,JPY,CNY
```

Next create the file lib/fixer-service.js. This is where we'll write helper code for our Express server to easily request information from Fixer. Copy the following code: require('dotenv').config(); const axios = require('axios');

```
const symbols = process.env.SYMBOLS || 'EUR,USD,GBP';
```

```
// Axios Client declaration
```

```
const api = axios.create({
  baseURL: 'http://data.fixer.io/api', params: {
    access_key: process.env.API_KEY, },
  timeout: process.env.TIMEOUT || 5000, });
```

```
// Generic GET request function const get = async (url) => {
  const response = await api.get(url); const { data } = response;
  if (data.success) {
    return data;
  }
  throw new Error(data.error.type); };
```

```
module.exports = {
  getRates: () => get(`/latest&symbols=${symbols}&base=EUR`), };
```

Again, take some time to go through the code to understand what's happening. If you're unsure, you can also check out the documentation for [dotenv](#), [axios](#) and read up on [module exports](#). Let's now do a quick test to confirm the `getRates()` function is working.

Open `server.js` and add this code: `const { getRates } = require('./lib/fixer-service');`

```
...  
// Place this block at the bottom const test = async() => {  
  const data = await getRates(); console.log(data);  
}  
  
test();
```

Run `npm start` or `node server`. After a few seconds, you should get the following output: {

```
success: true,  
timestamp: 1523871848,  
base: 'EUR',  
date: '2018-04-16',  
rates: {  
  EUR: 1,  
  USD: 1.23732,  
  GBP: 0.865158,  
  AUD: 1.59169,  
  BTC: 0.000153,  
  KES: 124.226892,  
  JPY: 132.608498,  
  CNY: 7.775567
```

```
}  
}
```

If you get something similar to the above, it means the code is working. The values will of course be different, since the rates change every day. Now comment out the test block and insert this code right before the statement that redirects all traffic to index.html: // Express Error handler const errorHandler = (err, req, res) => {

```
  if (err.response) {  
    // The request was made and the server responded with a status code  
    // that falls out of the range of 2xx res.status(403).send({ title:  
    'Server responded with an error', message: err.message }); } else  
    if (err.request) {  
      // The request was made but no response was received  
      res.status(503).send({ title: 'Unable to communicate with server',  
      message: err.message }); } else {  
        // Something happened in setting up the request that triggered an  
        Error res.status(500).send({ title: 'An unexpected error occurred',  
        message: err.message }); }  
      };  
  
    // Fetch Latest Currency Rates app.get('/api/rates', async (req,  
    res) => {  
      try {  
        const data = await getRates(); res.setHeader('Content-Type',  
        'application/json'); res.send(data);  
      } catch (error) {  
        errorHandler(error, req, res); }  
      });
```

As we can see, there's a custom error handler function that's designed to handle different error scenarios, which can occur during execution of server code. When an error occurs, an error message is constructed and sent back to the client.

Let's confirm this bit of code is working. Restart the Express server and navigate

your browser to this URL: [localhost:3000/api/rates](http://localhost:3000/api/rates). You should see the same JSON result that was displayed in the console. We can now implement a view that will display this information in a neat, elegant table.

Open `public/index.html` and replace the `rates-template` with this code: `<!-- Currency Rates Template --> <script id="rates-template" type="text/x-handlebars-template"> <h1 class="ui header">Currency Rates</h1> <hr>`

```
<div class="ui loading basic segment"> <div class="ui horizontal list"> <div class="item"> <i class="calendar alternate outline icon"></i> <div class="content"> <div class="ui sub header">Date</div> <span>{{date}}</span> </div>
```

```
</div>
```

```
<div class="item"> <i class="money bill alternate outline icon"></i> <div class="content"> <div class="ui sub header">Base</div> <span>{{base}}</span> </div>
```

```
</div>
```

```
</div>
```

```
<table class="ui celled striped selectable inverted table"> <thead>
```

```
<tr>
```

```
<th>Code</th> <th>Rate</th> </tr>
```

```
</thead>
```

```
<tbody>
```

```
{{#each rates}}
```

```
<tr>
```

```
<td>{{@key}}</td> <td>{{this}}</td> </tr>
```

```
{{/each}}
```

```
</tbody>
```

```
</table>
```

```
</div>
```

```
</script>
```

Remember we're using Semantic UI to provide us with styling. I'd like you to pay close attention to the [Segment loading](#) component. This will be an indication to let users know that something is happening as the app fetches the data. We're also using the [Table UI](#) to display the rates. Please go through the linked documentation if you're new to Semantic.

Now let's update our code in `public/js/app.js` to make use of this new template. Replace the the first `route.add('/')` function with this code: `//`  
Instantiate api handler `const api = axios.create({`

```
  baseURL: 'http://localhost:3000/api', timeout: 5000,  
});
```

```
// Display Error Banner
```

```
const showError = (error) => {
```

```
  const { title, message } = error.response.data; const html =  
  errorTemplate({ color: 'red', title, message }); el.html(html);  
};
```

```
// Display Latest Currency Rates router.add('/', async () => {
```

```
  // Display loader first
```

```
  let html = ratesTemplate(); el.html(html);
```

```
  try {
```

```
    // Load Currency Rates
```

```
    const response = await api.get('/rates'); const { base, date, rates  
  } = response.data; // Display Rates Table
```

```
    html = ratesTemplate({ base, date, rates }); el.html(html);
```

```
  } catch (error) {
```

```
    showError(error);
```

```
  } finally {
```

```
// Remove loader status

$('.loading').removeClass('loading'); }

});
```

The first code block instantiates an API client for communicating with our proxy server. The second block is a global function for handling errors. Its work is simply to display an error banner in case something goes wrong on the server side. The third block is where we get rates data from the `localhost:3000/api/rates` endpoint and pass it to the `rates-template` to display the information.

Simply refresh the browser. You should now have the following view:

Single Page App	Currency Rates	Exchange Conversion	Historical Rates
Currency Rates			
DATE 2018-04-16 BASE EUR			
Code	Rate		
EUR	1		
USD	1.236554		
GBP	0.864524		
AUD	1.590827		
BTC	0.000153		
KES	124.149993		
JPY	132.603023		
CNY	7.769511		

Next we'll build an interface for converting currencies.

# Exchange Conversion

For the currency conversion, we'll use two endpoints:

- [Fixer's Symbols Endpoint](#)
- [Free Currency Converter Endpoint](#).

We need the symbols endpoint to get a list of supported currency codes. We'll use this data to populate the dropdowns that the users will use to select which currencies to convert. Open `lib/fixer-service.js` and add this line right after the `getRates()` function: `getSymbols: () => get('/symbols')`,

Create another helper file, `lib/free-currency-service.js`, and add the following code: `require('dotenv').config(); const axios = require('axios');`

```
const api = axios.create({
  baseURL: 'https://free.currencyconverterapi.com/api/v5', timeout:
  process.env.TIMEOUT || 5000, });

module.exports = {
  convertCurrency: async (from, to) => {
    const response = await api.get(`/convert?
    q=${from}_${to}&compact=y`); const key = Object.keys(response.data)
    [0]; const { val } = response.data[key]; return { rate: val };
  },
};
```

This will help us get the conversion rate from one currency to another for free. In the client code, we'll have to calculate the conversion amount by multiplying amount by rate. Now let's add these two service methods to our Express server



code. Open `server.js` and update accordingly: `const { getRates, getSymbols, } = require('./lib/fixer-service');` `const { convertCurrency } = require('./lib/free-currency-service');` ...

// Insert right after `get '/api/rates'`, just before the `redirect` statement

// Fetch Symbols

```
app.get('/api/symbols', async (req, res) => {  
  try {  
    const data = await getSymbols(); res.setHeader('Content-Type',  
    'application/json'); res.send(data);  
  } catch (error) {  
    errorHandler(error, req, res); }  
});
```

// Convert Currency

```
app.post('/api/convert', async (req, res) => {  
  try {  
    const { from, to } = req.body; const data = await  
    convertCurrency(from, to); res.setHeader('Content-Type',  
    'application/json'); res.send(data);  
  } catch (error) {  
    errorHandler(error, req, res); }  
});
```

Now our proxy server should be able to get symbols and conversion rates. Take note that `/api/convert` is a POST method. We'll use a form on the client side to build the currency conversion UI. Feel free to use the test function to confirm both endpoints are working. Here's an example: // Test Symbols Endpoint

```
const test = async() => {
```

```
  const data = await getSymbols(); console.log(data);  
}
```

```
// Test Currency Conversion Endpoint const test = async() => {
const data = await convertCurrency('USD', 'KES');
console.log(data);
}
```

You'll have to restart the server for each test. Remember to comment out the tests once you've confirmed the code is working so far. Let's now work on our currency conversion UI. Open public/index.html and update the exchange-template by replacing the existing code with this: <script id="exchange-template" type="text/x-handlebars-template"> <h1 class="ui header">Exchange Rate</h1> <hr>

```
<div class="ui basic loading segment"> <form class="ui form">

<div class="three fields"> <div class="field"> <label>From</label>
<select class="ui dropdown" name="from" id="from"> <option
value="">Select Currency</option> {{#each symbols}}

<option value="{{@key}}">{{this}}</option> {{/each}}

</select>

</div>

<div class="field"> <label>To</label> <select class="ui dropdown"
name="to" id="to"> <option value="">Select Currency</option>
{{#each symbols}}

<option value="{{@key}}">{{this}}</option> {{/each}}

</select>

</div>

<div class="field"> <label>Amount</label> <input type="number"
name="amount" id="amount" placeholder="Enter amount"> </div>

</div>

<div class="ui primary submit button">Convert</div> <div class="ui
error message"></div> </form>

<br>

<div id="result-segment" class="ui center aligned segment"> <h2
id="result" class="ui header"> 0.00
```

```
</h2>

</div>

</div>

</script>
```

Take your time to go through the script and understand what's happening. We're using [Semantic UI Form](#) to build the interface. We're also using Handlebars notation to populate the dropdown boxes. Below is the JSON format used by Fixer's Symbols endpoint: {

```
"success": true,

"symbols": {

"AED": "United Arab Emirates Dirham", "AFN": "Afghan Afghani",

"ALL": "Albanian Lek",

"AMD": "Armenian Dram",

}

}
```

Take note that the symbols data is in map format. That means the information is stored as key `{{@key}}` and value `{{this}}` pairs. Let's now update `public/js/app.js` and make it work with the new template. Open the file and replace the existing route code for `/exchange` with the following: // Perform POST request, calculate and display conversion results const

```
getConversionResults = async () => {

// Extract form data

const from = $('#from').val();

const to = $('#to').val();

const amount = $('#amount').val(); // Send post data to
Express(proxy) server try {

const response = await api.post('/convert', { from, to }); const {
rate } = response.data; const result = rate * amount;
$('#result').html(`${to} ${result}`); } catch (error) {

showError(error);
```

```

} finally {
$('#result-segment').removeClass('loading'); }
};

// Handle Convert Button Click Event const convertRatesHandler = ()
=> {
if ($('#ui.form').form('is valid')) {
// hide error message
$('#ui.error.message').hide(); // Post to Express server
$('#result-segment').addClass('loading'); getConversionResults();
// Prevent page from submitting to server return false;
}
return true;
};

router.add('/exchange', async () => {
// Display loader first
let html = exchangeTemplate();
el.html(html);
try {
// Load Symbols
const response = await api.get('/symbols'); const { symbols } =
response.data; html = exchangeTemplate({ symbols }); el.html(html);
$('#loading').removeClass('loading'); // Validate Form Inputs
$('#ui.form').form({
fields: {
from: 'empty',

```

```

to: 'empty',
amount: 'decimal',
},
});
// Specify Submit Handler
$('.submit').click(convertRatesHandler); } catch (error) {
showError(error);
}
});

```

Refresh the page. You should now have the following view:

Single Page App
Currency Rates
Exchange Conversion
Historical Rates

### Exchange Rate

From

To

Amount


United States Dollar
Kenyan Shilling
2500

Convert

0.00

Select some currencies of your choosing and enter an amount. Then hit the *Convert* button:

Single Page App
Currency Rates
Exchange Conversion
Historical Rates



**An unexpected error occurred**

Cannot destructure property 'from' of 'undefined' or 'null'.

Oops! We just hit an error scenario. At least we know our error handling code is working. To figure out why the error is occurring, go back to the server code and look at the `/api/convert` function. Specifically, look at the line that says `const`

```
{ from, to } = req.body;.
```

It seems Express is unable to read properties from the request object. To fix this, we need to install middleware that can help out with this: `npm install body-parser`

Next, update the server code as follows: `const bodyParser = require('body-parser'); ...`

```
/** Place this code right before the error handler function */
```

```
// Parse POST data as URL encoded data
app.use(bodyParser.urlencoded({
  extended: true,
}));
```

```
// Parse POST data as JSON
app.use(bodyParser.json());
```

Start the server again and refresh the browser. Try doing another conversion. It should now work.

---

Single Page App	Currency Rates	Exchange Conversion	Historical Rates
-----------------	----------------	---------------------	------------------

---

### Exchange Rate

---

From	To	Amount
<input type="text" value="United States Dollar"/>	<input type="text" value="Kenyan Shilling"/>	<input type="text" value="2500"/>
<input type="button" value="Convert"/>		
<div>KES 250874.995</div>		

Let's now focus on the final bit — historical currency rates. Let's start with the views.

## Historical Currency Rates

Implementing this feature will be like combining the tasks from the first and second pages. We're going to build a tiny form where the user will be expected to input a date. When the user clicks submit, the currency rates for the specified date will be displayed in table format. We'll use the [Historical Rates Endpoint](#) from Fixer API to achieve this. The API request looks like this:  
`https://data.fixer.io/api/2013-12-24`

```
? access_key = API_KEY
```

```
& base = GBP
```

```
& symbols = USD,CAD,EUR
```

And the response will look like this: {

```
"success": true,  
"historical": true,  
"date": "2013-12-24",  
"timestamp": 1387929599,  
"base": "GBP",  
"rates": {  
  "USD": 1.636492,  
  "EUR": 1.196476,  
  "CAD": 1.739516  
}  
}
```

Open `lib/fixer-service.js` and the Historical Rates Endpoint like this: ...

```
/** Place right after getSymbols */
```

```
getHistoricalRate: date =>  
get(`/ ${date}&symbols=${symbols}&base=EUR`), ...
```

Open `server.js` and add this code: ...

```
const { getRates, getSymbols, getHistoricalRate } =
require('./lib/fixer-service'); ...

/** Place this after '/api/convert' post function */

// Fetch Currency Rates by date app.post('/api/historical', async
(req, res) => {

try {

const { date } = req.body; const data = await
getHistoricalRate(date); res.setHeader('Content-Type',
'application/json'); res.send(data);

} catch (error) {

errorHandler(error, req, res); }

});

...

```

If you're in any doubt as to how the code is arranged, please refer to the complete `server.js` file on [GitHub](#). Feel free to write a quick test to confirm the historical endpoint is working: `const test = async() => {`

```
const data = await getHistoricalRate('2012-07-14');
console.log(data);

}
```

```
test();
```

Do remember to comment out the test block once you confirm everything's working. Now let's now work on the client code.

Open `index.html`. Delete the existing `historical-template` we used as a placeholder, and replace it with the following: `<script id="historical-template" type="text/x-handlebars-template"> <h1 class="ui header">Historical Rates</h1> <hr>`

```
<form class="ui form"> <div class="field"> <label>Pick Date</label>
```



```

<div class="ui calendar" id="calendar"> <div class="ui input left
icon"> <i class="calendar icon"></i> <input type="text"
placeholder="Date" id="date"> </div>

</div>

</div>

<div class="ui primary submit button">Fetch Rates</div> <div
class="ui error message"></div> </form>

<div class="ui basic segment"> <div id="historical-table"></div>
</div>

</script>

```

Take a look at the form first. One thing I'd like to point out is that Semantic UI doesn't officially have a date input. However, thanks to [Michael de Hoog's](#) contribution, we have the [Semantic-UI-Calendar](#) module available to us. Simply install it using npm: `npm install semantic-ui-calendar`

Go back to `public/index.html` and include it within the scripts section: ...

```

<script src="scripts/semantic-ui-css/semantic.min.js"></script>
<script src="scripts/semantic-ui-calendar/dist/calendar.min.js">
</script> ....

```

To display the historical rates, we'll simply reuse the `rates-template`. Next open `public/js/app.js` and update the existing route code for `/historical`:

```

const getHistoricalRates = async () => {

const date = $('#date').val(); try {

const response = await api.post('/historical', { date }); const {
base, rates } = response.data; const html = ratesTemplate({ base,
date, rates }); $('#historical-table').html(html); } catch (error)
{

showError(error);

} finally {

$('.segment').removeClass('loading'); }

};

```

```

const historicalRatesHandler = () => {
  if ($('#ui.form').form('is valid')) {
    // hide error message

    $('#ui.error.message').hide(); // Indicate loading status
    $('#segment').addClass('loading'); getHistoricalRates();

    // Prevent page from submitting to server return false;
  }
  return true;
};

router.add('/historical', () => {
  // Display form
  const html = historicalTemplate(); el.html(html);

  // Activate Date Picker
  $('#calendar').calendar({
    type: 'date',

    formatter: { //format date to yyyy-mm-dd date: date => new
      Date(date).toISOString().split('T')[0], },

  });

  // Validate Date input
  $('#ui.form').form({
    fields: {
      date: 'empty',
    },
  });

  $('#submit').click(historicalRatesHandler); });


```

Once again, take time to read the comments and understand the code and what it's doing. Then restart the server, refresh the browser and navigate to the `/historical` path. Pick any date before the year 1999 then click *Fetch Rates*. You should have something like this:

[Single Page App](#)[Currency Rates](#)[Exchange Conversion](#)[Historical Rates](#)



### Historical Rates

Pick Date

 2015-02-09

Fetch Rates


### Currency Rates

 DATE  
2015-02-09  BASE  
EUR

Code	Rate
EUR	1
USD	1.132543
GBP	0.744364
AUD	1.452675
BTC	0.005127
KES	103.663641
JPY	134.428049
CNY	7.054123

If you pick a date before the year 1999 or a date in the future, an error banner will be displayed when you submit the form.

[Single Page App](#)[Currency Rates](#)[Exchange Conversion](#)[Historical Rates](#)



**An unexpected error occurred**  
no\_rates\_available

## Summary

Now that we've come to the end of the tutorial, you should see that it's not that difficult to build a single-page application powered by REST APIs without using a framework. But there are a few things we should be concerned with:

- **DOM Performance.** In our client-side code, we're directly manipulating the DOM. This can soon get out of hand as the project grows, causing the UI to become sluggish.
- **Browser Performance.** There are quite a number of front-end libraries that we've loaded as scripts in `index.html`, which is okay for development purposes. For production deployment, we need a system for bundling all scripts such that the browsers use a single request for loading the necessary JavaScript resources.
- **Monolithic Code.** For the server code, it's easier to break down code into modular parts since it runs within a Node environment. However, for client-side code, it's not easy to organize in modules unless you use a bundler like [webpack](#).
- **Testing.** So far we've been doing manual testing. For a production-ready application, we need to set up a testing framework like Jasmine, Mocha or Chai to automate this work. This will help prevent recurring errors.

These are just a few of the many issues you'll face when you approach project development without using a framework. Using something such as Angular, React or Vue will help you alleviate a lot of these concerns. I hope this tutorial has been helpful and that it will aid you in your journey to becoming a professional JavaScript developer.

# Chapter 4: Build a To-do List with Hyperapp, the 1KB JS Micro-framework

by Darren Jones

**In this tutorial, we'll be using Hyperapp to build a to-do list app. If you want to learn functional programming principles, but not get bogged down in details, read on.**

Hyperapp is hot right now. It recently surpassed *11,000* stars on GitHub and made the 5th place in the Front-end Framework section of the [2017 JavaScript Rising Stars](#). It was also featured [on SitePoint](#) recently, when it hit version 1.0.

The reason for Hyperapp's popularity can be attributed to its pragmatism and ultralight size (1.4 kB), while at the same time achieving results similar to React and Redux out of the box.

## So, What Is HyperApp?

Hyperapp allows you to build dynamic, single-page web apps by taking advantage of a virtual DOM to update the elements on a web page quickly and efficiently in a similar way to React. It also uses a single object that's responsible for keeping track of the application's state, just like Redux. This makes it easier to manage the state of the app and make sure that different elements don't get out of sync with each other. The main influence behind Hyperapp was the [Elm architecture](#).

At its core, Hyperapp has three main parts:

- **State.** This is a single object tree that stores all of the information about the application.
- **Actions.** These are methods that are used to change and update the values in the state object.
- **View.** This is a function that returns virtual node objects that compile to HTML code. It can use JSX or a similar templating language and has access to the state and actions objects.

These three parts interact with each other to produce a dynamic application. Actions are triggered by events on the page. The action then updates the state, which then triggers an update to the view. These changes are made to the Virtual DOM, which Hyperapp uses to update the actual DOM on the web page.

## Getting Started

To get started as quickly as possible, we're going to use CodePen to develop our app. You need to make sure that the JavaScript preprocessor is set to *Babel* and the Hyperapp package is loaded as an external resource using the following link: <https://unpkg.com/hyperapp>

To use Hyperapp, we need to import the app function as well as the h method, which Hyperapp uses to create VDOM nodes. Add the following code to the JavaScript pane in CodePen: `const { h, app } = hyperapp;`

We'll be using JSX for the view code. To make sure Hyperapp knows this, we need to add the following comment to the code:

```
/** @jsx h */
```

The `app()` method is used to initialize the application:

```
const main = app(state, actions, view, document.body);
```

This takes the state and actions objects as its first two parameters, the `view()` function as its third parameter, and the last parameter is the HTML element where the application is to be inserted into your markup. By convention, this is usually the `<body>` tag, represented by `document.body`.

To make it easy to get started, I've created a boilerplate Hyperapp code template on CodePen that contains all the elements mentioned above. It can be forked by [clicking on this link](#).

# Hello Hyperapp!

Let's have a play around with Hyperapp and see how it all works. The `view()` function accepts the state and actions objects as arguments and returns a Virtual DOM object. We're going to use JSX, which means we can write code that looks a lot more like HTML. Here's an example that will return a heading:

```
const view = (state, actions) => (  
  <h1>Hello Hyperapp!</h1>  
);
```

This will actually return the following VDOM object:

```
{  
  name: "h1",  
  props: {},  
  children: "Hello Hyperapp!"  
}
```

The `view()` function is called every time the state object changes. Hyperapp will then build a new Virtual DOM tree based on any changes that have occurred. Hyperapp will then take care of updating the actual web page in the most efficient way by comparing the differences in the new Virtual DOM with the old one stored in memory.



# Components

Components are pure functions that return virtual nodes. They can be used to create reusable blocks of code that can then be inserted into the view. They can accept parameters in the usual way that any function can, but they don't have access to the state and actions objects in the same way that the view does.

In the example below, we create a component called `Hello()` that accepts an object as a parameter. We extract the `name` value from this object using destructuring, before returning a heading containing this value: `const Hello = ({name}) => <h1>Hello {name}</h1>;`

We can now refer to this component in the view as if it were an HTML element entitled `<Hello />`. We can pass data to this element in the same way that we can pass props to a React component:

```
const view = (state, actions) => (  
  <Hello name="Hyperapp" />  
);
```

Note that, as we're using JSX, component names must start with capital letters or contain a period.

## State

The state is a plain old JavaScript object that contains information about the application. It's the "single source of truth" for the application and can only be changed using actions.

Let's create the state object for our application and set a property called name:

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

The view function now has access to this property. Update the code to the following:

```
const view = (state, actions) => (  
  <Hello name={state.name} />  
);
```

Since the view can access the state object, we can use its name property as an attribute of the `<Hello />` component.

## Actions

Actions are functions used to update the state object. They're written in a particular form that returns another, curried function that accepts the current state and returns an updated, partial state object. This is partly stylistic, but also ensures that the state object remains immutable. A completely new state object is created by merging the results of an action with the previous state. This will then result in the view function being called and the HTML being updated.

The example below shows how to create an action called `changeName()`. This function accepts an argument called `name` and returns a curried function that's used to update the `name` property in the state object with this new name.

```
const actions = {  
  changeName: name => state => ({name: name})  
};
```

To see this action, we can create a button in the view and use an `onclick` event handler to call the action, with an argument of "Batman". To do this, update the view function to the following:

```
const view = (state, actions) => (  
  <div>  
    <Hello name={state.name} />  
    <button onclick={() => actions.changeName('Batman')}>I'm  
Batman</button>  
  </div>  
)
```

Now try clicking on the button and watch the name change!

You can see a [live example here](#).

# Hyperlist

Now it's time to build something more substantial. We're going to build a simple to-do list app that allows you to create a list, add new items, mark them as complete and delete items.

First of all, we'll need to start a new pen on CodePen. Add the following code, or simply fork my HyperBoiler pen:

```
const { h, app } = hyperapp;
```

```
/** @jsx h */
```

```
const state = {
```

```
};
```

```
const actions = {
```

```
};
```



```
$secondary-color: hotpink;
```

```
$bg-color: #222;
```

```
* {
```

```
    margin: 0;
```

```
    padding: 0;
```

```
    box-sizing: border-box;
```

```
}
```

```
body {
```

```
    padding-top: 50px;
```

```
    background: $bg-color;
```

```
    color: $primary-color;
```

```
display: flex;

height: 100vh;

justify-content: center;

font-family: $base-fonts;
}

h1 {

color: $secondary-color;

& strong{ color: $primary-color; }

font-family: $heading-font;

font-weight: 100;

font-size: 4.2em;

text-align: center;
```

```
}
```

```
a{
```

```
    color: $primary-color;
```

```
}
```

```
.flex{
```

```
    display: flex;
```

```
    align-items: top;
```

```
    margin: 20px 0;
```



```
input {  
  
    border: 1px solid $primary-color;  
  
    background-color: $primary-color;  
  
  
    font-size: 1.5em;  
  
    font-weight: 200;  
  
  
    width: 50vw;  
  
    height: 62px;  
  
  
    padding: 15px 20px;  
  
    margin: 0;  
  
    outline: 0;
```

```
&::-webkit-input-placeholder {  
  
    color: $bg-color;  
  
}
```

```
&::-moz-placeholder {  
  
    color: $bg-color;  
  
}
```

```
&::-ms-input-placeholder {  
  
    color: $bg-color;  
  
}
```

```
    &:hover, &:focus, &:active {  
  
        background: $primary-color;  
  
    }  
  
}  
  
button {  
  
    height: 62px;  
  
    font-size: 1.8em;  
  
    padding: 5px 15px;  
  
    margin: 0 3px;  
  
}  
  
}
```

```
ul#list {  
  
    display: flex;  
  
    flex-direction: column;  
  
    padding: 0;  
  
    margin: 1.2em;  
  
    width: 50vw;  
  
    li {  
  
        font-size: 1.8em;  
  
        vertical-align: bottom;  
  
        &.completed{  
  
            color: $secondary-color;  
  
            text-decoration: line-through;  
  
            button{
```

```
        color: $primary-color;

    }

}

button {

    visibility: hidden;

    background: none;

    border: none;

    color: $secondary-color;

    outline: none;

    font-size: 0.8em;

    font-weight: 50;

    padding-top: 0.3em;

    margin-left: 5px;
```

```
}

&:hover{

  button{

    visibility: visible;

  }

}

}

}
```

```
button {

  background: $bg-color;

  border-radius: 0px;

  border: 1px solid $primary-color;
```

```
color: $primary-color;
```

```
font-weight: 100;
```

```
outline: none;
```

```
padding: 5px;
```

```
margin: 0;
```

```
&:hover, &:disabled {
```

```
    background: $primary-color;
```

```
    color: #111;
```

```
}

&:active {

    outline: 2px solid $primary-color;

}

&:focus {

    border: 1px solid $primary-color;

}

}
```

These just add a bit of style and Hyperapp branding to the application.

Now let's get on and start building the actual application!



## Initial State and View

To start with, we're going to set up the initial state object and a simple view.

When creating the initial state object, it's useful to think about what data and information your application will want to keep track of throughout its lifecycle. In the case of our list, we'll need an array to store the to-dos, as well as a string that represents whatever is written in the input field where the actual to-dos are entered. This will look like the following:

```
const state = {  
  items: [],  
  input: '',  
};
```

Next, we'll create the `view()` function. To start with, we'll focus on the code required to add an item. Add the following code:

```
const view = (state, actions) => (  
  <div>  
    <h1><strong>Hyper</strong>List</h1>  
    <AddItem add={actions.add} input={actions.input} value=  
{state.input} />  
  </div>  
);
```

This will display a title, as well as an element called `<AddItem />`. This isn't a new HTML element, but a component that we'll need to create. Let's do that now:

```
const AddItem = ({ add, input, value }) => (  
  <div class='flex'>  
    <input type="text" value={value}  
      onkeyup={e => (e.keyCode === 13 ? add() : null)}  
      oninput={e => input({ value: e.target.value })}  
    />  
    <button onclick={add}></button>  
  </div>  
);
```

This returns an `<input>` element that will be used to enter our to-dos, as well as a `<button>` element that will be used to add them to the list. The component

accepts an object as an argument, from which we extract three properties: `add`, `input` and `value`.

As you might expect, the `add()` function will be used to add an item to our list of to-dos. This function is called, either if the `Enter` key is pressed (it has a `keyCode` of 13) or if the button is clicked. The `input()` function is used to update the value of the current item in state and is called whenever the text field receives user input. Finally, the `value` property is whatever the user has typed into the input field.

Note that the `input()` and `add()` functions are actions, passed as props to the `<AddItem />` component:

```
<AddItem add={actions.add} input={actions.input} value={state.input} />
```

You can also see that the `value` prop is taken from the state's `input` property. So the text that's displayed in the input field is actually stored in the state and updated every time a key is pressed.

To glue everything together, we need to add the `input` action:

```
const actions = {  
  input: ({ value }) => ({ input: value })  
}
```

Now if you start typing inside the input field, you should see that it displays what you're typing. This demonstrates the Hyperapp loop:

1. the `oninput` event is triggered as the user types text into the input field
2. the `input()` action is called
3. the action updates the `input` property in the state
4. a change in state causes the `view()` function to be called and the VDOM is updated
5. the changes in the VDOM are then made to the actual DOM and the page is re-rendered to display the key that was pressed.

Have a go and you should see what's typed appear in the input field.

Unfortunately, pressing `Enter` or clicking on the “+” button doesn't do anything at the moment. That's because we need to create an action that adds items to our list.

## Adding a Task

Before we look at creating a list item, we need to think how they're going to be represented. JavaScript's object notation is perfect, as it lets us store information as key-value pairs. We need to think about what properties a list item might have. For example, it needs a value that describes what needs to be done. It also needs a property that states if the item has been completed or not. An example might be:

```
{  
  value: 'Buy milk',  
  completed: false,  
  id: 123456  
}
```

Notice that the object also contains a property called `id`. This is because VDOM nodes in Hyperapp require a unique key to identify them. We'll [use a timestamp for this](#).

Now we can have a go at creating an action for adding items. Our first job is to reset the input field to be empty. This is done by resetting the `input` property to an empty string. We then need to add a new object to the `items` array. This is done using the `Array.concat` method. This acts in a similar way to the `Array.push()` method, but it returns a new array, rather than mutating the array it's acting on. Remember, we want to create a new state object and then merge it with the current state rather than simply mutating the current state directly. The `value` property is set to the value contained in `state.input`, which represents what's been entered in the input field:

```
add: () => state => ({  
  input: '',  
  items: state.items.concat({  
    value: state.input,  
    completed: false,  
    id: Date.now()  
  })  
})
```

Note that this action contains two different states. There's the *current* state that's represented by the argument supplied to the second function. There's also the *new* state that's the return value of the second function.

To demonstrate this in action, let's imagine the app has just started with an empty list of items and a user has entered the text "Buy milk" into the input field and pressed Enter, triggering the `add()` action.

Before the action, the state looks like this:

```
state = {  
  input: 'Buy milk',  
  items: []  
}
```

This object is passed as an argument to the `add()` action, which will return the following state object:

```
state = {  
  input: '',  
  items: [{  
    value: 'Buy milk',  
    completed: false,  
    id: 1521630421067  
  }]  
}
```

Now we can add items to the `items` array in the state, but we can't see them! To fix this, we need to update our view. First of all we need to create a component for displaying the items in the list:

```
const ListItem = ({ value, id }) => <li id={id} key={id}>{value}</li>;
```

This uses a `<li>` element to display a value, which is provided as an argument. Note also that the `id` and `key` attributes both have the same value, which is the unique ID of the item. The `key` attribute is used internally by Hyperapp, so isn't displayed in the rendered HTML, so it's useful to also display the same information using the `id` attribute, especially since this attribute shares the same condition of uniqueness.

Now that we have a component for our list items, we need to actually display them. JSX makes this quite straightforward, as it will loop over an array of values and display each one in turn. The problem is that the `state.items` doesn't include JSX code, so we need to use `Array.map` to change each item object in the array into JSX code, like so:

```
state.items.map(item => ( <ListItem id={item.id} value={item.value}
/> ));
```

This will iterate over each object in the `state.items` array and create a new array that contains `ListItem` components instead. Now we just need to add this to the view. Update the `view()` function to the code below:

```
const view = (state, actions) => (
  <div>
    <h1><strong>Hyper</strong>List</h1>
    <AddItem add={actions.add} input={actions.input} value=
{state.input} />
    <ul id='list'>
      { state.items.map(item => ( <ListItem id={item.id} value=
{item.value} /> )) }
    </ul>
  </div>
);
```

This simply places the new array of `ListItem` components inside a pair of `<ul>` tags so they're displayed as an unordered list.

Now if you try adding items, you should see them appear in a list below the input field!

## Mark a Task as Completed

Our next job is to be able to toggle the completed property of a to-do. Clicking on an uncompleted task should update its completed property to true, and clicking on a completed task should toggle its completed property back to false.

This can be done by using the following action:

```
toggle: id => state => ({
  items: state.items.map(item => (
    id === item.id ? Object.assign({}, item, { completed:
!item.completed }) : item
  ))
})
```

There's quite a bit going on in this action. First of all, it accepts a parameter called `id`, which refers to the unique ID of each task. The action then iterates over *all* of the items in the array, checking if the ID value provided matches the `id` property of each list item object. If it does, it changes the completed property to the opposite of what it currently is using the negation operator `!`. This change is made using the `Object.assign()` method, which creates a new object and performs a shallow merge with the old `item` object and the updated properties. Remember, we *never* update objects in the state directly. Instead, we create a new version of the state that overwrites the current state.

Now we need to wire this action up to the view. We do this by updating the `ListItem` component so that it has an `onclick` event handler that will call the `toggle` action we just created. Update the `ListItem` component code so that it looks like the following:

```
const ListItem = ({ value, id, completed, toggle, destroy }) => (
  <li class={completed && "completed"} id={id} key={id} onclick={e
=> toggle(id)}>{value}</li>
);
```

The eagle-eyed among you will have spotted that the component has gained some extra parameters and there's also some extra code in the `<li>` attributes list:

```
class={completed && "completed"}
```

This is a common pattern in Hyperapp that's used to insert extra fragments of code when certain conditions are true. It uses *short-circuit* or *lazy* evaluation to set the class as completed if the completed argument is true. This is because when using &&, the return value of the operation will be the second operand if both operands are true. Since the string "completed" is always true, this will be returned if the first operand — the completed argument — is true. This means that, if the task has been completed, it will have class of “completed” and can be styled accordingly.

Our last job is to update the code in the `view()` function to add the extra argument to the `<ListItem />` component:

```
const view = (state, actions) => (  
  <div>  
    <h1><strong>Hyper</strong>List</h1>  
    <AddItem add={actions.add} input={actions.input} value=  
{state.input} />  
    <ul id='list'>  
      {  
        state.items.map(item => (  
          <ListItem id={item.id} value={item.value} completed=  
{item.completed} toggle={actions.toggle} />  
        ))  
      }  
    </ul>  
  </div>  

```

Now if you add some items and try clicking on them, you should see that they get marked as complete, with a line appearing through them. Click again and they revert back to incomplete.

## Delete a Task

Our list app is running quite nicely at the moment, but it would be good if we could delete any items that we no longer need in the list.

Our first job is to add a `destroy()` action that will remove an item from the `items` array in `state`. We can't do this using the `Array.slice()` method, as this is a destructive method that acts on the original array. Instead, we use the `filter()` method, which returns a new array that contains all the item objects that pass a specified condition. This condition is that the `id` property doesn't equal the ID that was passed as an argument to the `destroy()` action. In other words, it returns a new array that doesn't include the item we want to get rid of. This new list will then replace the old one when the state is updated.

Add the following code to the `actions` object:

```
destroy: id => state => ({ items: state.items.filter(item =>
item.id !== id) })
```

Now we again have to update the `ListItem` component to add a mechanism for triggering this action. We'll do this by adding a button with an `onclick` event handler:

```
const ListItem = ({ value, id, completed, toggle, destroy }) => (
  <li class={completed && "completed"} id={id} key={id} onclick={e
=> toggle(id)}>
    {value}
    <button onclick={ () => destroy(id) }>x</button>
  </li>
);
```

Note that we also need to add another parameter called `destroy` that represents the action we want to use when the button is clicked. This is because components don't have direct access to the `actions` object in the same way as the view does, so the view needs to pass any actions explicitly.

Last of all, we need to update the view to pass `actions.destroy` as an argument to the `<ListItem />` component:

```
const view = (state, actions) => (
  <div>
```



```
<h1><strong>Hyper</strong>List</h1>
<AddItem add={actions.add} input={actions.input} value=
{state.input} />
<ul id='list'>
  {state.items.map(item => (
    <ListItem
      id={item.id}
      value={item.value}
      completed={item.completed}
      toggle={actions.toggle}
      destroy={actions.destroy}
    />
  ))}
</ul>
</div>
);
```

Now if you add some items to your list, you should notice the “x” button when you mouse over them. Click on this and they should disappear into the ether!

## Delete All Completed Tasks

The last feature we'll add to our list app is the ability to remove all completed tasks at once. This uses the same `filter()` method that we used earlier — returning an array that only contains item objects with a `completed` property value of `false`. Add the following code to the `actions` object:

```
clearAllCompleted: ({items}) => ({ items: items.filter(item => !item.completed) })
```

To implement this, we simply have to add a button with an `onclick` event handler to call this action to the bottom of the view:

```
const view = (state, actions) => (  
  <div>  
    <h1><strong>Hyper</strong>List</h1>  
    <AddItem add={actions.add} input={actions.input} value=  
{state.input} />  
    <ul id='list'>  
      {state.items.map(item => (  
        <ListItem  
          id={item.id}  
          value={item.value}  
          completed={item.completed}  
          toggle={actions.toggle}  
          destroy={actions.destroy}  
        />  
      )  
    )  
    </ul>  
    <button onclick={() => actions.clearAllCompleted({ items:  
state.items })}>  
      Clear completed items  
    </button>  
  </div>  
);
```

Now have a go at adding some items, mark a few of them as complete, then press the button to clear them all away. Awesome!

## Complete Example

See the Pen [Hyperlist](#).

## That's All, Folks

That brings us to the end of this tutorial. We've put together a simple to-do list app that does most things you'd expect such an app to do. If you're looking for inspiration and want to add to the functionality, you could look at adding priorities, changing the order of the items using drag and drop, or adding the ability to have more than one list.

I hope this tutorial has helped you to gain an understanding about how Hyperapp works. If you'd like to dig a bit deeper into Hyperapp, I'd recommend [reading the docs](#) and also having a peek at the [source code](#). It's not very long, and will give you a useful insight into how everything works in the background. You can also ask more questions on the [Hyperapp Slack group](#). It's one of the friendliest groups I've used, and I've been given a lot of help by the knowledgeable members. You'll also find that [Jorge Bucaran](#), the creator of Hyperapp, frequently hangs out on there and offers help and advice.

Using CodePen makes developing Hyperapp applications really quick and easy, but you'll eventually want to build your own applications locally and also deploy them online. For tips on how to do that, check out the next chapter on bundling a Hyperapp app and deploying it to GitHub Pages!

# Chapter 5: Use Parcel to Bundle a Hyperapp App & Deploy to GitHub Pages

by Darren Jones

**In the last chapter we met Hyperapp, a tiny library that can be used to build dynamic, single-page web apps in a similar way to React or Vue.**

In this chapter we're going to turn things up a notch. We're going to create the app locally (we were working on CodePen previously), learn how to bundle it using [Parcel](#) (a module bundler similar to webpack or Rollup) and deploy it to the web using [GitHub Pages](#).

Don't worry if you didn't complete the project from the first post. All the code is provided here (although I won't go into detail explaining what it does) and the principles outlined can be applied to most other JavaScript projects.

If you'd like to see what we'll be ending up with, you can [view the finished project here](#), or download the code from [our GitHub repo](#).

## Basic Setup

In order to follow along, you'll need to have both [Node.js and npm](#) installed (they come packaged together). I'd recommend using a version manager such as [nvm](#) to manage your Node installation ([here's how](#)), and if you'd like some help getting to grips with npm, then check out our [beginner-friendly npm tutorial](#).

We'll be using the terminal commands to create files and folders, but feel free to do it by just pointing and clicking instead if that's your thing.

To get started, create a new folder called `hyperlist`:

```
mkdir hyperlist
```

Now change to that directory and initialize a new project using npm:

```
cd hyperlist/  
npm init
```

This will prompt you to answer some questions about the app. It's fine to just press enter to accept the default for any of these, but feel free to add in your name as the author and to add a description of the app.

This should create a file called `package.json` inside the `hyperlist` directory that looks similar to the following:

```
{  
  "name": "hyperlist",  
  "version": "1.0.0",  
  "description": "A To-do List made with Hyperapp",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "author": "DAZ",  
  "license": "MIT"  
}
```

Now we need to install the Hyperapp library. This is done using npm along with the `--save` flag, which means that the `package.json` file will be updated to include it as a dependency:

```
npm install --save hyperapp
```

This might give some warnings about not having a repository field. Don't worry about this, as we'll be fixing it later. It should update the `package.json` file to include the following entry (there might be a slight difference in version number):

```
"dependencies": {  
  "hyperapp": "^1.2.5"  
}
```

It will also create a directory called `node_modules` where all the Hyperapp files are stored, as well as a file called `package-lock.json`. This is used to keep track of the dependency tree for all the packages that have been installed using npm.

Now we're ready to start creating the app!

## Folder Structure

It's a common convention to put all of your source code into a folder called `src`. Within this folder, we're going to put all of our JavaScript files into a directory called `js`. Let's create both of those now:

```
mkdir -p src/js
```

In the [previous post](#) we learned that apps are built in Hyperapp using three main parts: state, actions and view. In the interests of code organization, we're going to place the code for each part in a separate file, so we need to create these files inside the `js` directory:

```
cd src/js  
touch state.js actions.js view.js
```

Don't worry that they're all empty. We'll add the code soon!

Last of all, we'll go back into the `src` directory and create our “entry point” files. These are the files that will link to all the others. The first is `index.html`, which will contain some basic HTML, and the other is `index.js`, which will link to all our other JavaScript files and also our SCSS files:

```
cd ..  
touch index.html index.js
```

Now that our folder structure is all in place, we can go ahead and start adding some code and wiring all the files together. Onward!

## Some Basic HTML

We'll start by adding some basic HTML code to the `index.html` file. Hyperapp takes care of creating the HTML and can render it directly into the `<body>` tag. This means that we only have to set up the meta information contained in the `<head>` tag. Except for the `<title>` tag's value, you can get away with using the same `index.html` file for every project. Open up `index.html` in your favorite text editor and add the following code: `<!doctype html> <html lang='en'> <head> <meta charset='utf-8'> <meta name='viewport' content='width=device-width, initial-scale=1'> <title>HyperList</title> </head> <body> <script src='index.js'> </script> </body> </html>`

Now it's time to add some JavaScript code!



## ES6 Modules

[Native JavaScript modules](#) were introduced in ES6 (aka ES2015). Unfortunately, browsers have been slow to adopt the use of ES6 modules natively, although [things are now starting to improve](#). Luckily, we can still use them to organize our code, and Parcel will sort out piecing them all together.

Let's start by adding the code for the initial state inside the `state.js` file:

```
const state = {
  items: [],
  input: '',
  placeholder: 'Make a list..'
};

export default state;
```

This is the same as the object we used in [the previous chapter](#), but with the export declaration at the end. This will make the object available to any other file that imports it. By making it the default export, we don't have to explicitly name it when we import it later.

Next we'll add the actions to `actions.js`:

```
const actions = {
  add: () => state => ({
    input: '',
    items: state.items.concat({
      value: state.input,
      completed: false,
      id: Date.now()
    })
  }),
  input: ({ value }) => ({ input: value }),
  toggle: id => state => ({
    items: state.items.map(item => (
      id === item.id ? Object.assign({}, item, { completed:
!item.completed }) : item
    ))
  }),
  destroy: id => state => ({
    items: state.items.filter(item => item.id !== id)
  }),
  clearAllCompleted: ({ items }) => ({
```

```

    items: items.filter(item => !item.completed)
  })
};

export default actions;

```

Again, this is the same as the object we used in the previous chapter, with the addition of the export declaration at the end.

Last of all we'll add the view code to view.js:

```

import { h } from 'hyperapp'

const AddItem = ({ add, input, value, placeholder }) => (
  <div class='flex'>
    <input
      type="text"
      onkeyup={e => (e.keyCode === 13 ? add() : null)}
      oninput={e => input({ value: e.target.value })}
      value={value}
      placeholder={placeholder}
    />
    <button onclick={add}></button>
  </div>
);

const ListItem = ({ value, id, completed, toggle, destroy }) => (
  <li class={completed && "completed"} id={id} key={id} onclick={e
=> toggle(id)}>
    {value} <button onclick={ () => destroy(id) }>x</button>
  </li>
);

const view = (state, actions) => (
  <div>
    <h1><strong>Hyper</strong>List</h1>
    <AddItem
      add={actions.add}
      input={actions.input}
      value={state.input}
      placeholder={state.placeholder}
    />
    <ul id='list'>
      {state.items.map(item => (
        <ListItem
          id={item.id}
          value={item.value}
          completed={item.completed}

```

```

        toggle={actions.toggle}
        destroy={actions.destroy}
      />
    )}}
  </ul>
  <button onclick={() => actions.clearAllCompleted({ items:
state.items }) }>
    Clear completed items
  </button>
</div>
);
export default view;

```

First of all, this file uses the `import` declaration to import the `h` module from the Hyperapp library that we installed using `npm` earlier. This is the function that Hyperapp uses to create the Virtual DOM nodes that make up the view.

This file contains two components: `AddItem` and `ListItem`. These are just functions that return JSX code and are used to abstract different parts of the view into separate building blocks. If you find that you're using a large number of components, it might be worth moving them into a separate `components.js` file and then importing them into the `view.js` file.

Notice that only the `view` function is exported at the end of the file. This means that only this function can be imported by other files, rather than the separate components.

Now we've added all our JavaScript code, we just need to piece it all together in the `index.js` file. This is done using the `import` directive. Add the following code to `index.js`:

```

import { app } from 'hyperapp'

import state from './js/state.js'
import actions from './js/actions.js'
import view from './js/view.js'

const main = app(state, actions, view, document.body);

```

This imports the `app` function from the Hyperapp library, then imports the three JavaScript files that we just created. The object or function that was exported from each of these files is assigned to the variables `state`, `actions` and `view` respectively, so they can be referenced in this file.

The last line of code calls the `app` function, which starts the app running. It uses each of the variables created from our imported files as the first three arguments. The last argument is the HTML element where the app will be rendered — which, by convention, is `document.body`.

## Add Some Style

Before we go on to build our app, we should give it some style. Let's go to the `src` directory and create a folder for our SCSS:

```
mkdir src/scss
```

Now we'll create the two files that will contain the SCSS code that we used in part 1:

```
cd src/scss
```

```
touch index.scss _settings.scss
```

We're using a file called `_settings.scss` to store all the Sass variables for the different fonts and colors our app will use. This makes them easier to find if you decide to update any of these values in the future. Open up the `_settings.scss` file and add the following code:

```
// fonts
```

```
@import url("https://fonts.googleapis.com/css?family=Racing+Sans+One");
```

```
$base-fonts: Helvetica Neue, sans-serif;
```

```
$heading-font: Racing Sans One, sans-serif;
```

```
// colors

$primary-color: #00caff;

$secondary-color: hotpink;

$bg-color: #222;
```

The app-specific CSS goes in `index.scss`, but we need to make sure we import the `_settings.scss` file at the start, as the variables it contains are referenced later on in the file. Open up `index.scss` and add the following code:

```
@import 'settings';

* {

  margin: 0;

  padding: 0;

  box-sizing: border-box;

}
```

```
body {  
  
  padding-top: 50px;  
  
  background: $bg-color;  
  
  color: $primary-color;  
  
  display: flex;  
  
  height: 100vh;  
  
  justify-content: center;  
  
  font-family: $base-fonts;  
}
```

```
h1 {  
  
  color: $secondary-color;  
  
  & strong{ color: $primary-color; }
```

```
font-family: $heading-font;

font-weight: 100;

font-size: 4.2em;

text-align: center;

}
```

```
a{ color: $primary-color; }
```

```
.flex{

display: flex;

align-items: top;

margin: 20px 0;
```



```
input {  
  
    border: 1px solid $primary-color;  
  
    background-color: $primary-color;  
  
    font-size: 1.5em;  
  
    font-weight: 200;  
  
    width: 50vw;  
  
    height: 62px;  
  
    padding: 15px 20px;  
  
    margin: 0;  
  
    outline: 0;  
  
    &::-webkit-input-placeholder { color: $bg-color; }  
  
    &::-moz-placeholder { color: $bg-color; }
```

```
&::-ms-input-placeholder { color: $bg-color; }
```

```
&:hover, &:focus, &:active { background: $primary-color; }
```

```
}
```

```
button {
```

```
    height: 62px;
```

```
    font-size: 1.8em;
```

```
    padding: 5px 15px;
```

```
    margin: 0 3px;
```

```
}
```

```
}
```

```
ul#list {
```

```
display: flex;

flex-direction: column;

padding: 0;

margin: 1.2em;

width: 50vw;

li {

    font-size: 1.8em;

    vertical-align: bottom;

    &.completed{

        color: $secondary-color;

        text-decoration: line-through;

        button{

            color: $primary-color;
```

```
}

}

button {

    background: none;

    border: none;

    color: $secondary-color;

    outline: none;

    font-size: 0.8em;

    font-weight: 50;

    padding-top: 0.3em;

    margin-left: 5px;

}

}
```

```
}
```

```
button {
```

```
    background: $bg-color;
```

```
    border-radius: 0px;
```

```
    border: 1px solid $primary-color;
```

```
    color: $primary-color;
```

```
    font-weight: 100;
```

```
    outline: none;
```

```
    padding: 5px;
```

```
    margin: 0;
```

```
    &:hover, &:disabled {
```

```
background: $primary-color;

color: #111;

}

&:active { outline: 2px solid $primary-color; }

&:focus { border: 1px solid $primary-color; }

}
```

If your SCSS starts to get more complicated, you can break it up into separate files and then import them all into `index.scss`.

Now we need to link these files to our app. We don't actually place the link in our HTML file, as you usually do with CSS. Instead, we place it in the `index.js` file. This is because we're using SCSS and it needs to be pre-processed into CSS. Parcel will do this for us and also sort out linking the HTML file to the standard CSS file that it creates.

To import the SCSS files, we just need to update our `index.js` file to include the following line:

```
import './scss/index.scss'
```

Now that all our code's complete, it's time to start work on the build process!

# Babel

Babel will transpile the modern JavaScript code into code that most browsers can consume. It will also take care of rewriting the JSX code into pure JavaScript.

In order to be able to use Babel with JSX transforms, we need to install it along with the JSX plugin: `npm install --save babel-plugin-transform-react-jsx babel-preset-env`

We also need to create a `.babelrc` file that's used to tell Babel to use the `h` function from Hyperapp when processing the JSX. The following code will create the file with the relevant information: `echo '{ "plugins": [ ["transform-react-jsx", { "pragma": "h" } ] ] }' > .babelrc`

Note that this is a *hidden file*, so you might not be able to see it after it's been created!

# Parcel

Unfortunately, our code won't currently work in all browsers as it stands. We need to use a build process to transpile our ES6+ code to ES5 and merge all our JS files into a single file. Let's use [Parcel](#) to do that.

Parcel is a module bundler, similar to webpack or Rollup, that promises zero-configuration and is blazingly fast. It allows us to write modern JavaScript in separate files, and then bundles them together into a single, minified JavaScript file that most browsers will be able to consume. It also supports multiple CSS, SCSS and PostCSS files out of the box.

First of all, let's install Parcel:

```
npm install --save parcel-bundler
```

Parcel comes with its own built-in server. This means that you can continue to develop and make changes to the app and Parcel will build it in the background, so any changes are shown instantly!

To start the server running, enter the following command:

```
./node_modules/.bin/parcel src/index.html --out-dir docs
```

This specifies that the entry point is the `index.html` file. This is all Parcel needs to know about, as it will follow the link to `index.js` that's in this file and then follow the `import` directives in that file.

It also specifies that a folder called `docs` be used to output all of the static files to. By default, this is usually called `dist` — but, as you'll see later, we need it to be called `docs` so that we can integrate it with GitHub Pages.

You should also see a message that the app is being built in the terminal window. You might even notice that Parcel installs the npm module `node-sass` for you as it automatically notices that we've been using SCSS files, but also that we don't have `node-sass` installed. How cool is that?!

After a few seconds, you should see a message similar to the following:



```
Server running at http://localhost:1234  
Built in 3.15s.
```

The server is now running, and if you open up your browser and go to <http://localhost:1234>, you'll be able to see the app running. This will update on the fly, so any changes you make in your code will be reflected on the page straight away (or after a brief pause to rebuild the code). It also hotloads modules, so it will automatically install any npm modules that are required as they're needed, like it did with "node-sass". Awesome!

Once you're happy with how the site looks, it's time to build the static site. First of all, stop the server running by holding down `Ctrl` and `c` together. Then run the following command in the terminal:

```
./node_modules/.bin/parcel build src/index.html --out-dir docs --  
public-url ./
```

This will build the static files and place them inside the docs folder.

If you take a peak inside the docs folder, you should find a file called `index.html`. Open this in your browser and you should see the site running, using only the static files in the docs folder. Parcel has bundled all the relevant code together and used Babel to transpile our modern JavaScript into a single JavaScript file and used node-sass to pre-process our SCSS files into a single CSS file. Open them up and you can see that the code has also been minimized!

## npm Scripts

npm has a useful feature called *scripts* that allows you to run specific pieces of code with a single command. We can use this to create a couple of scripts that will speed up our use of Parcel.

Add the following to the “scripts” section of the `package.json` file:

```
"start": "parcel src/index.html --out-dir docs",  
"build": "parcel build src/index.html --out-dir docs --public-url  
./"
```

Now we can simply run the following commands to start the server:

```
npm start
```

And the following command will run the build process:

```
npm run build
```

If you’ve never used npm scripts, or would like a refresher, you might like to check out our [beginner-friendly tutorial on the subject](#).

# Deploying to GitHub Pages

GitHub is a great place for hosting your code, and it also has a great feature called [GitHub Pages](#) that allows you to host static sites on GitHub. To get started, you'll need to make sure you have a GitHub account and you have [git](#) installed on your local machine.

To make sure we don't commit unnecessary files, let's add a gitignore file to the hyperlist directory:

```
touch .gitignore
```

As the name suggests, this file tells git which files (or patterns) it should ignore. It's usually used to avoid committing files that aren't useful to other collaborators (such as the temporary files IDEs create, etc.).

I'd recommend adding the following items to make sure they're not tracked by git (remember that gitignore is a hidden file!):

```
# Logs
logs
*.log
npm-debug.log*

# Runtime data
pids
*.pid
*.seed

# Dependency directory
node_modules

# Optional npm cache directory
.npm

# Optional REPL history
.node_repl_history

# Cache for Parcel
.cache

# Apple stuff
.DS_Store
```

Now we're ready to initialize git in the `hyperlist` directory:

```
git init
```

Next, we add all the files we've created so far:

```
git add .
```

Then we commit those files to version control:

```
git commit -m 'Initial Commit'
```

Now that our important files are being tracked by git, we need to create a remote repository on GitHub. Just log into your account and click on the *New Repository* button and follow the instructions. If you get stuck, you can consult GitHub's documentation here: [Create A Repo](#).

After you've done this, you'll need to add the URL of your remote GitHub repository on your local machine:

```
git remote add origin https://github.com/<username>/<repo-name>.git
```

Be sure to replace `<username>` and `<repo-name>` with the correct values. If you'd like to check that you've done everything correctly, you can use `git remote -v`.

And, finally, we need to push our code to GitHub:

```
git push origin master
```

This will push all of your code to your GitHub repository, including the static files in the `docs` directory. GitHub Pages can now be configured to use the files in this directory. To do this, log in to the repository on GitHub and go to the *Settings* section of the repository and scroll down to the *GitHub Pages* section. Then under *Source*, select the option that says "master branch /docs folder", as can be seen in the screenshot below:

#### Source

Your GitHub Pages site is currently being built from the `/docs` folder in the `master` branch. [Learn more](#).

master branch /docs folder ▾

Save

This should mean that you can now access the app at the following address:  
`https://username.github.io/repo-name`.

For example, you can see ours at [sitepoint-editors.github.io/hyperlist/](https://sitepoint-editors.github.io/hyperlist/).

## Workflow

From now on, if you make any changes to your app, you can adhere to the following workflow:

1. start the development server: `npm start`
2. make any changes
3. check that the changes work on the development server
4. shut the server down by holding down `Ctrl + c`
5. rebuild the app: `npm run build`
6. stage the changes for commit: `git add .`
7. commit all the changes to git: `git commit -m 'latest update'`
8. push the changes to GitHub: `git push origin master.`

We can speed this process up by creating an npm script to take care of the last three steps in one go. Add the following to the “scripts” entry in `package.json`:

```
"deploy": "npm run build && git add . && git commit -a -m 'latest build' && git push origin master"
```

Now all you need to do if you want to deploy your code after making any changes is to run the following command:

```
npm run deploy
```

## That's All, Folks!

And that brings us to the end of this tutorial. I used the app we created in part 1 of this tutorial, but the principles remain the same for most JavaScript projects. Hopefully I've demonstrated how easy it is to use Parcel to build a static JS site and automatically deploy it to GitHub Pages with just a single command!

# Chapter 6: Interactive Data Visualization with Modern JavaScript and D3

by Adam Janes

In this chapter, I want to take you through an example project that I built recently — a *totally original* type of visualization using the D3 library, which showcases how each of these components add up to make D3 a great library to learn.

D3 stands for Data Driven Documents. It's a JavaScript library that can be used to make all sorts of wonderful data visualizations and charts.

If you've ever seen any of the [fabulous interactive stories](#) from the New York Times, you'll already have seen D3 in action. You can also see some cool examples of great projects that have been built with D3 [here](#).

The learning curve is pretty steep for getting started with the library, since D3 has [a few special quirks](#) that you probably won't have seen before. However, if you can get past the first phase of learning enough D3 to be dangerous, then you'll soon be able to build some really cool stuff for yourself.

There are three main factors that really make D3 stand out from any other libraries out there:

1. **Flexibility.** D3 lets you take any kind of data, and directly associate it with shapes in the browser window. This data can be *absolutely anything*, allowing for a huge array of interesting use cases to create completely original visualizations.
2. **Elegance.** It's easy to add interactive elements with *smooth transitions* between updates. The library is *written beautifully*, and once you get the hang of the syntax, it's easy to keep your code clean and tidy.
3. **Community.** There's a vast ecosystem of fantastic developers using D3 already, who readily share their code online. You can use sites like

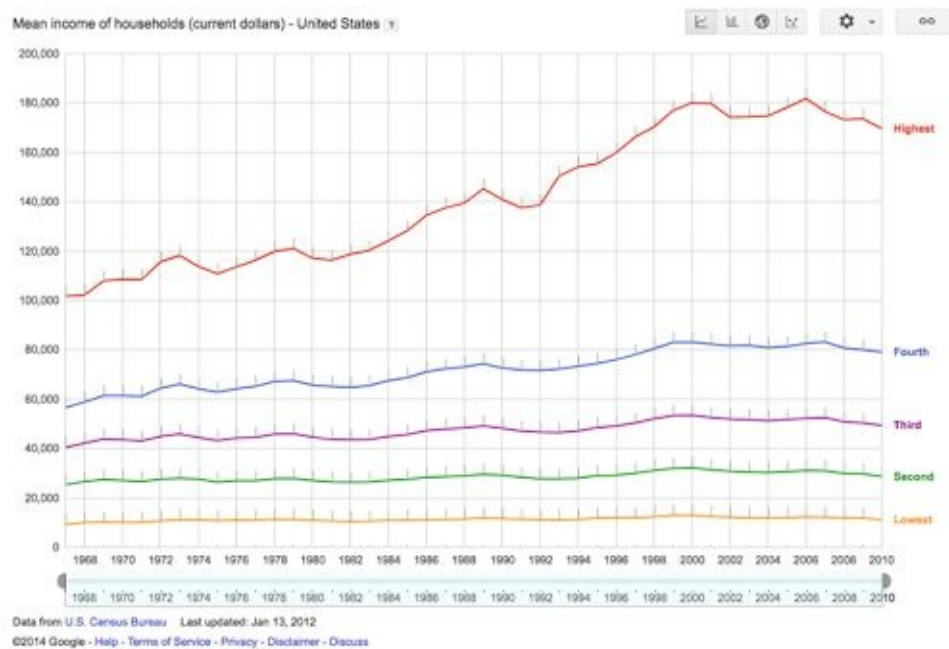


[bl.ocks.org](https://bl.ocks.org) and [blockbuilder.org](https://blockbuilder.org) to quickly find pre-written code by others, and copy these snippets directly into your own projects.

# The Project

As an economics major in college, I had always been interested in income inequality. I took a few classes on the subject, and it struck me as something that wasn't fully understood to the degree that it should be.

I started exploring income inequality using [Google's Public Data Explorer](#) ...



When you adjust for inflation, household income has *stayed pretty much constant* for the bottom 40% of society, although per-worker productivity has been skyrocketing. It's only really been *the top 20%* that have reaped more of the benefits (and within that bracket, the difference is even more shocking if you look at the top 5%).

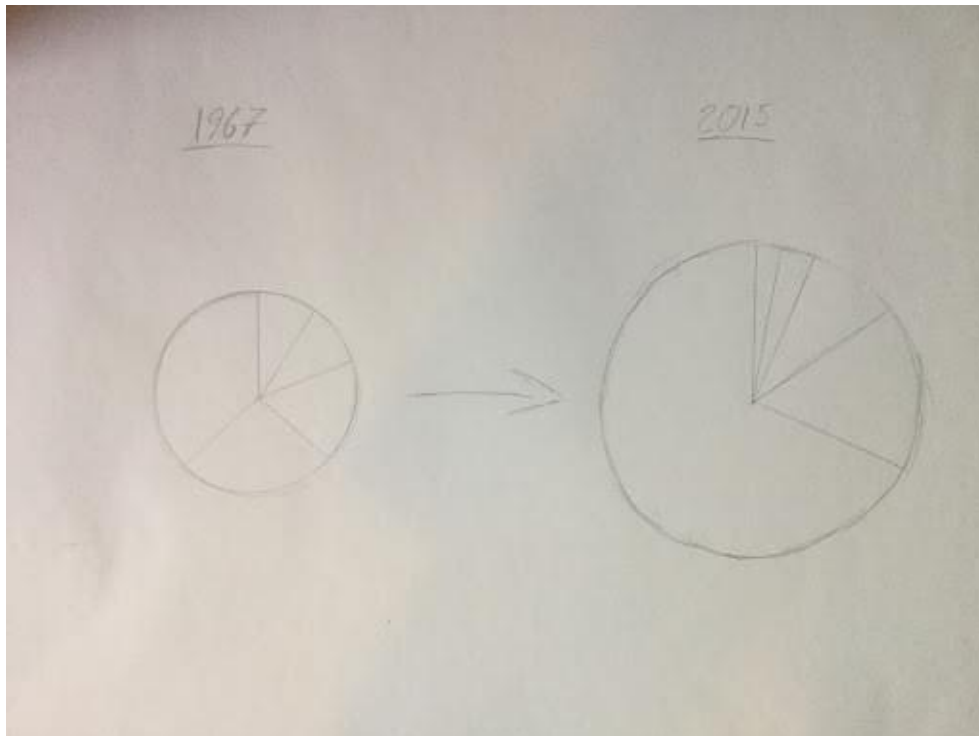
Here was a message that I wanted to get across in a convincing way, which provided a perfect opportunity to use some D3.js, so I started sketching up a few ideas.

## Sketching

Because we're working with D3, I could more or less just start sketching out *absolutely anything* that I could think of. Making a simple line graph, bar chart, or bubble chart would have been easy enough, but I wanted to make something different.

I find that the most common analogy that people tended to use as a counterargument to concerns about inequality is that “if [the pie gets bigger](#), then there's more to go around”. The intuition is that, if the total share of GDP manages to increase by a large extent, then even if some people are getting a *thinner slice* of pie, then they'll still be *better off*. However, as we can see, it's totally possible for the pie to get bigger *and* for people to be getting less of it overall.

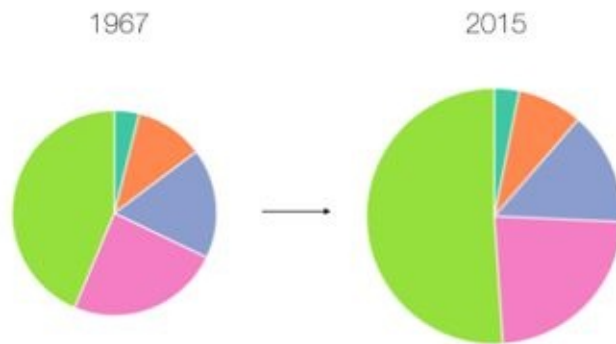
My first idea for visualizing this data looked something like this:



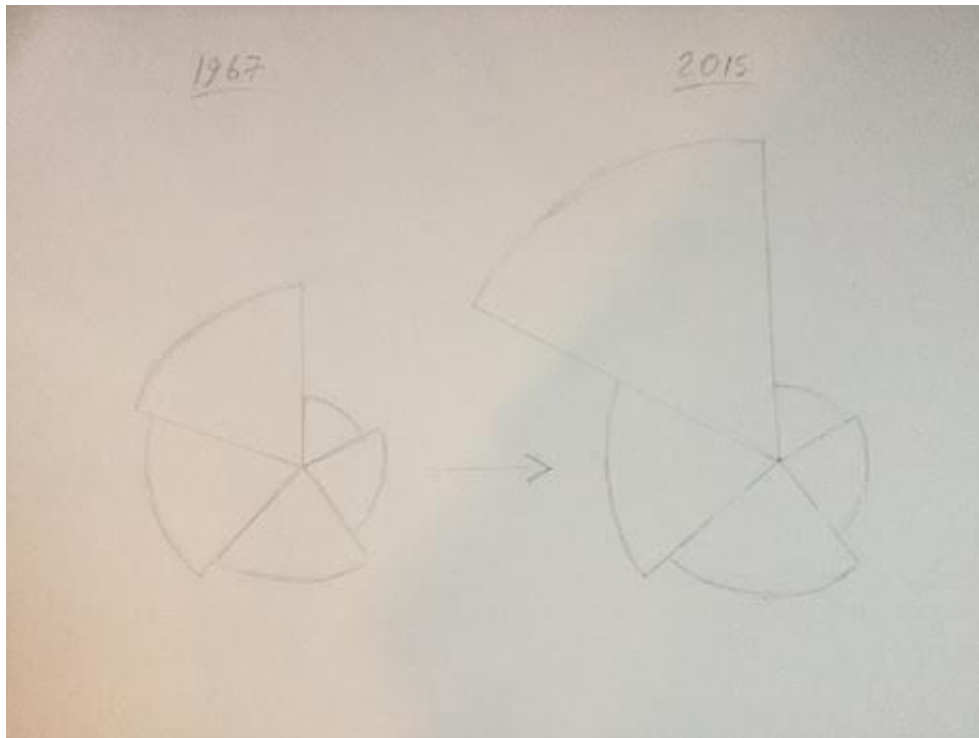
The idea would be that we'd have this pulsating pie chart, with each slice representing a fifth of the US income distribution. The area of each pie slice would relate to how much income that segment of the population is taking in,

and the total area of the chart would represent its total GDP.

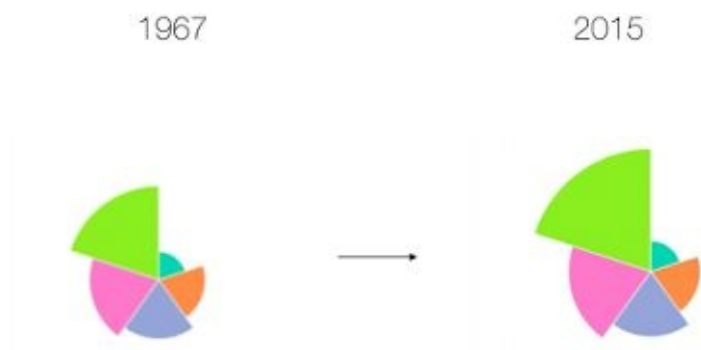
However, I soon came across a bit of a problem. It turns out that the human brain is *exceptionally poor at distinguishing between the size of different areas*. When I mapped this out more concretely, the message wasn't anywhere near as obvious as it should have been:



Here, it actually looks like the poorest Americans are getting *richer* over time, which confirms what seems to be intuitively true. I thought about this problem some more, and my solution involved keeping the angle of each arc constant, with the radius of each arc changing dynamically.



Here's how this ended up looking in practice:



I want to point out that this image still tends to understate the effect here. The effect would have been more obvious if we used a simple bar chart:



However, I was committed to making a unique visualization, and I wanted to hammer home this message that the *pie* can get *bigger*, whilst a *share* of it can get *smaller*. Now that I had my idea, it was time to build it with D3.

## Borrowing Code

So, now that I know what I'm going to build, it's time to get into the real meat of this project, and start *writing some code*.

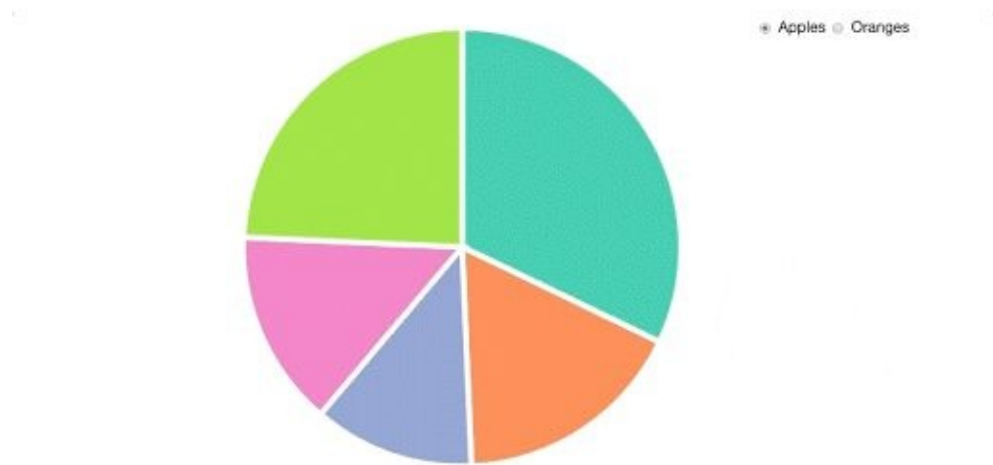
You might think that I'd start by writing my first few lines of code from scratch, but you'd be wrong. This is D3, and since we're working with D3, we can always find some pre-written code from the community to start us off.

We're creating something completely new, but it has a lot in common with a regular pie chart, so I took a quick look on [bl.ocks.org](http://bl.ocks.org), and I decided to go with this [classic implementation](#) by Mike Bostock, one of the creators of D3. This file has probably been copied thousands of times already, and the guy who wrote it is a real wizard with JavaScript, so we can be sure that we're starting with a nice block of code already.

This file is written in D3 V3, which is now two versions out of date, since version 5 was finally released last month. A big change in D3 V4 was that the library switched to using a flat namespace, so that scale functions like `d3.scale.ordinal()` are written like `d3.scaleOrdinal()` instead. In version 5, the biggest change was that data loading functions are now structured as [Promises](#), which makes it easier to handle multiple datasets at once.

To avoid confusion, I've already gone through the trouble of creating an updated V5 version of this code, which I've saved on [blockbuilder.org](http://blockbuilder.org). I've also converted the syntax to fit with ES6 conventions, such as switching ES5 anonymous functions to arrow functions.

Here's what we're starting off with already:



I then copied these files into my working directory, and made sure that I could replicate everything on my own machine. If you want to follow along with this tutorial yourself, then you can [clone this project from our GitHub repo](#). You can start with the code in the file `starter.html`. Please note that you will need a server (such as [this one](#)) to run this code, as under the hood it relies on the [Fetch API](#) to retrieve the data.

Let me give you a quick rundown of how this code is working.



## Walking Through Our Code

First off, we're declaring a few constants at the top of our file, which we'll be using to define the size of our pie chart:

```
const width = 540;
const height = 540;
const radius = Math.min(width, height) / 2;
```

This makes our code super reusable, since if we ever want to make it bigger or smaller, then we only need to worry about changing these values right here.

Next, we're appending an SVG canvas to the screen. If you don't know much about SVGs, then you can think about the canvas as the space on the page that we can draw shapes on. If we try to draw an SVG outside of this area, then it simply won't show up on the screen:

```
const svg = d3.select("#chart-area")
  .append("svg")
  .attr("width", width)
  .attr("height", height)
  .append("g")
  .attr("transform", `translate(${width / 2}, ${height / 2})`);
```

We're grabbing hold of an empty div with the ID of chart-area with a call to `d3.select()`. We're also attaching an SVG canvas with the `d3.append()` method, and we're setting some dimensions for its width and height using the `d3.attr()` method.

We're also attaching an SVG group element to this canvas, which is a special type of element that we can use to structure elements together. This allows us to shift our entire visualization into the center of the screen, using the group element's `transform` attribute.

After that, we're setting up a default scale that we'll be using to assign a new color for every slice of our pie:

```
const color = d3.scaleOrdinal(["#66c2a5", "#fc8d62",
"#8da0cb", "#e78ac3", "#a6d854", "#ffd92f"]);
```

Next, we have a few lines that set up D3's pie layout:

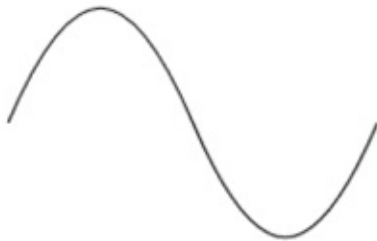
```
const pie = d3.pie()  
  .value(d => d.count)  
  .sort(null);
```

In D3, **layouts** are special functions that we can call on a set of data. A layout function takes in an array of data in a particular format, and spits out a *transformed array* with some automatically generated values, which we can then do something with.

We then need to define a [path generator](#) that we can use to draw our arcs. Path generators allow us to draw path SVGs in a web browser. All that D3 really does is to associate pieces of data with shapes on the screen, but in this case, we want to define a more complicated shape than just a simple circle or square. Path SVGs work by defining a route for a line to be drawn between, which we can define with its `d` attribute.

Here's what this might look like:

```
<svg width="190" height="160">  
  <path d="M10 80 C 40 10, 65 10, 95 80 S 150 150, 180 80"  
stroke="black" fill="transparent"/>  
</svg>
```



The `d` attribute contains a special encoding that lets the browser draw the path that we want. If you really want to know what this string means, you can find out about it in [MDN's SVG documentation](#). For programming in D3, we don't really need to know anything about this special encoding, since we have generators that will spit out our `d` attributes for us, which we just need to initialize with some simple parameters.

For an arc, we need to give our path generator an `innerRadius` and an `outerRadius` value in pixels, and the generator will sort out the complex maths

that goes into calculating each of the angles for us:

```
const arc = d3.arc()  
  .innerRadius(0)  
  .outerRadius(radius);
```

For our chart, we're using a value of zero for our `innerRadius`, which gives us a standard pie chart. However, if we wanted to draw a **donut chart** instead, then all we would need to do is plug in a value that's smaller than our `outerRadius` value.

After a couple of function declarations, we're loading in our data with the `d3.json()` function:

```
d3.json("data.json", type).then(data => {  
  // Do something with our data  
});
```

In D3 version 5.x, a call to `d3.json()` returns a [Promise](#), meaning that D3 will fetch the contents of the JSON file that it finds at the relative path that we give it, and execute the function that we're calling in the `then()` method once it's been loaded in. We then have access to the object that we're looking at in the `data` argument of our callback.

We're also passing in a function reference here — `type` — which is going to convert all of the values that we're loading in into numbers, which we can work with later:

```
function type(d) {  
  d.apples = Number(d.apples);  
  d.oranges = Number(d.oranges);  
  return d;  
}
```

If we add a `console.log(data);` statement to the top of our `d3.json` callback, we can take a look at the data that we're now working with:

```
{apples: Array(5), oranges: Array(5)}  
  apples: Array(5)  
    0: {region: "North", count: "53245"}  
    1: {region: "South", count: "28479"}  
    2: {region: "East", count: "19697"}  
    3: {region: "West", count: "24037"}  
    4: {region: "Central", count: "40245"}
```

```
oranges: Array(5)
  0: {region: "North", count: "200"}
  1: {region: "South", count: "200"}
  2: {region: "East", count: "200"}
  3: {region: "West", count: "200"}
  4: {region: "Central", count: "200"}
```

Our data is split into two different arrays here, representing our data for **apples** and **oranges**, respectively.

With this line, we're going to switch the data that we're looking at whenever one of our radio buttons gets clicked:

```
d3.selectAll("input")
  .on("change", update);
```

We'll also need to call the `update()` function on the first run of our visualization, passing in an initial value (with our "apples" array).

```
update("apples");
```

Let's take a look at what our `update()` function is doing. If you're new to D3, this might cause some confusion, since it's one of the most difficult parts of D3 to understand ...

```
function update(value = this.value) {
  // Join new data
  const path = svg.selectAll("path")
    .data(pie(data[value]));

  // Update existing arcs
  path.transition().duration(200).attrTween("d", arcTween);

  // Enter new arcs
  path.enter().append("path")
    .attr("fill", (d, i) => color(i))
    .attr("d", arc)
    .attr("stroke", "white")
    .attr("stroke-width", "6px")
    .each(function(d) { this._current = d; });
}
```

Firstly, we're using a default function parameter for `value`. If we're passing in an argument to our `update()` function (when we're running it for the first time), we'll use that string, or otherwise we'll get the value that we want from the

click event of our radio inputs.

We're then using the [General Update Pattern](#) in D3 to handle the behavior of our arcs. This usually involves performing a data join, exiting old elements, updating existing elements on the screen, and adding in new elements that were added to our data. In this example, we don't need to worry about exiting elements, since we always have the same number of pie slices on the screen.

First off, there's our data join:

```
// JOIN
const path = svg.selectAll("path")
  .data(pie(data[val]));
```

Every time our visualization updates, this associates a new array of data with our SVGs on the screen. We're passing our data (either the array for “apples” or “oranges”) into our `pie()` layout function, which is computing some start and end angles, which can be used to draw our arcs. This `path` variable now contains a special *virtual selection* of all of the arcs on the screen.

Next, we're updating all of the SVGs on the screen that still exist in our data array. We're adding in a [transition](#) here — a fantastic feature of the D3 library — to spread these updates over 200 milliseconds:

```
// UPDATE
path.transition().duration(200)
  .attrTween("d", arcTween);
```

We're using the `attrTween()` method on the `d3.transition()` call to define a custom transition that D3 should use to update the positions of each of its arcs (transitioning with the `d` attribute). We don't need to do this if we're trying to add a transition to most of our attributes, but we need to do this for transitioning between different paths. D3 can't really figure out how to transition between custom paths, so we're using the `arcTween()` function to let D3 know how each of our paths should be drawn at every moment in time.

Here's what this function looks like:

```
function arcTween(a) {
  const i = d3.interpolate(this._current, a);
  this._current = i(1);
  return t => arc(i(t));
}
```

```
}
```

We're using `d3.interpolate()` here to create what's called an **interpolator**. When we call the function that we're storing in the `i` variable with a value between 0 and 1, we'll get back a value that's somewhere between `this._current` and `a`. In this case, `this._current` is an object that contains the start and end angle of the pie slice that we're looking at, and `a` represents the new datapoint that we're updating to.

Once we have the interpolator set up, we're updating the `this._current` value to contain the value that we'll have at the end (`i(a)`), and then we're returning a function that will calculate the path that our arc should contain, based on this `t` value. Our transition will run this function on every tick of its clock (passing in an argument between 0 and 1), and this code will mean that our transition will know where our arcs should be drawn at any point in time.

Finally, our `update()` function needs to add in new elements that weren't in the previous array of data:

```
// ENTER
path.enter().append("path")
  .attr("fill", (d, i) => color(i))
  .attr("d", arc)
  .attr("stroke", "white")
  .attr("stroke-width", "6px")
  .each(function(d) { this._current = d; });
```

This block of code will set the initial positions of each of our arcs, the first time that this update function is run. The `enter()` method here gives us all the elements in our data that need to be added to the screen, and then we can loop over each of these elements with the `attr()` methods, to set the fill and position of each of our arcs. We're also giving each of our arcs a white border, which makes our chart look a little neater. Finally, we're setting the `this._current` property of each of these arcs as the initial value of the item in our data, which we're using in the `arcTween()` function.

Don't worry if you can't follow exactly how this is working, as it's a fairly advanced topic in D3. The great thing about this library is that you don't need to know all of its inner workings to create some powerful stuff with it. As long as you can understand the bits that you need to change, then it's fine to abstract some of the details that aren't completely essential.

That brings us to the next step in the process ...

## Adapting Code

Now that we have some code in our local environment, and we understand what it's doing, I'm going to switch out the data that we're looking at, so that it works with the data that we're interested in.

I've included the data that we'll be working with in the `data/` folder of our project. Since this new `incomes.csv` file is in a CSV format this time (it's the kind of file that you can open with Microsoft Excel), I'm going to use the `d3.csv()` function, instead of the `d3.json()` function:

```
d3.csv("data/incomes.csv").then(data => {  
  ...  
});
```

This function does basically the same thing as `d3.json()` — converting our data into a format that we can use. I'm also removing the `type()` initializer function as the second argument here, since that was specific to our old data.

If you add a `console.log(data)` statement to the top of the `d3.csv` callback, you'll be able to see the shape of the data we're working with:

```
(50) [{...}, {...}, {...}, {...}, {...}, {...}, {...} ... columns: Array(9)]  
  0:  
    1: "12457"  
    2: "32631"  
    3: "56832"  
    4: "92031"  
    5: "202366"  
    average: "79263"  
    top: "350870"  
    total: "396317"  
    year: "2015"  
    1: {1: "11690", 2: "31123", 3: "54104", 4: "87935", 5: "194277",  
year: "2014", top: "332729", average: "75826", total: "379129"}  
    2: {1: "11797", 2: "31353", 3: "54683", 4: "87989", 5: "196742",  
year: "2013", top: "340329", average: "76513", total: "382564"}  
    ...
```

We have an array of 50 items, with each item representing a year in our data. For each year, we then have an object, with data for each of the five income groups, as well as a few other fields. We could create a pie chart here for one of these



years, but first we'll need to shuffle around our data a little, so that it's in the right format. When we want to write a data join with D3, we need to pass in an array, where each item will be tied to an SVG.

Recall that, in our last example, we had an array with an item for every pie slice that we wanted to display on the screen. Compare this to what we have at the moment, which is an object with the keys of 1 to 5 representing each pie slice that we want to draw.

To fix this, I'm going to add a new function called `prepareData()` to replace the `type()` function that we had previously, which will iterate over every item of our data as it's loaded:

```
function prepareData(d){
  return {
    name: d.year,
    average: parseInt(d.average),
    values: [
      {
        name: "first",
        value: parseInt(d["1"])
      },
      {
        name: "second",
        value: parseInt(d["2"])
      },
      {
        name: "third",
        value: parseInt(d["3"])
      },
      {
        name: "fourth",
        value: parseInt(d["4"])
      },
      {
        name: "fifth",
        value: parseInt(d["5"])
      }
    ]
  }
}

d3.csv("data/incomes.csv", prepareData).then(data => {
  ...
});
```

For every year, this function will return an object with a values array, which we'll pass into our data join. We're labelling each of these values with a name field, and we're giving them a numerical value based on the income values that we had already. We're also keeping track of the average income in each year for comparison.

At this point, we have our data in a format that we can work with:

```
(50) [{...}, {...}, {...}, {...}, {...}, {...}, {...} ... columns: Array(9)]
  0:
    average: 79263
    name: "2015"
    values: Array(5)
      0: {name: "first", value: 12457}
      1: {name: "second", value: 32631}
      2: {name: "third", value: 56832}
      3: {name: "fourth", value: 92031}
      4: {name: "fifth", value: 202366}
  1: {name: "2014", average: 75826, values: Array(5)}
  2: {name: "2013", average: 76513, values: Array(5)}
  ...
```

I'll start off by generating a chart for the first year in our data, and then I'll worry about updating it for the rest of the years.

At the moment, our data starts in the year 2015 and ends in the year 1967, so we'll need to reverse this array before we do anything else:

```
d3.csv("data/incomes.csv", prepareData).then(data => {
  data = data.reverse();
  ...
});
```

Unlike a normal pie chart, for our graph, we want to fix the angles of each of our arcs, and just have the radius change as our visualization updates. To do this, we'll change the `value()` method on our pie layout, so that each pie slice always gets the same angles:

```
const pie = d3.pie()
  .value(1)
  .sort(null);
```

Next, we'll need to update our radius every time our visualization updates. To do this, we'll need to come up with a [scale](#) that we can use. A **scale** is a function in

D3 that takes an *input* between two values, which we pass in as the **domain**, and then spits out an *output* between two different values, which we pass in as the **range**. Here's the scale that we'll be using:

```
d3.csv("data/incomes.csv", prepareData).then(data => {  
  data = data.reverse();  
  const radiusScale = d3.scaleSqrt()  
    .domain([0, data[49].values[4].value])  
    .range([0, Math.min(width, height) / 2]);  
  ...  
});
```

We're adding this scale as soon as we have access to our data and we're saying that our input should range between 0 and the largest value in our dataset, which is the income from the richest group in the last year in our data (`data[49].values[4].value`). For the domain, we're setting the interval that our output value should range between.

This means that an input of zero should give us a pixel value of zero, and an input of the largest value in our data should give us a value of half the value of our width or height — whichever is smaller.

Notice that we're also using a *square root scale* here. The reason we're doing this is that we want the area of our pie slices to be proportional to the income of each of our groups, rather than the radius. Since  $\text{area} = \pi r^2$ , we need to use a square root scale to account for this.

We can then use this scale to update the `outerRadius` value of our arc generator inside our `update()` function:

```
function update(value = this.value) {  
  arc.outerRadius(d => radiusScale(d.data.value));  
  ...  
});
```

Whenever our data changes, this will edit the radius value that we want to use for each of our arcs.

We should also remove our call to `outerRadius` when we initially set up our arc generator, so that we just have this at the top of our file:

```
const arc = d3.arc()  
  .innerRadius(0);
```

Finally, we need to make a few edits to this `update()` function, so that everything matches up with our new data:

```
function update(data) {
  arc.outerRadius(d => radiusScale(d.data.value));

  // JOIN
  const path = svg.selectAll("path")
    .data(pie(data.values));

  // UPDATE
  path.transition().duration(200).attrTween("d", arcTween);

  // ENTER
  path.enter().append("path")
    .attr("fill", (d, i) => color(i))
    .attr("d", arc)
    .attr("stroke", "white")
    .attr("stroke-width", "2px")
    .each(function(d) { this._current = d; });
}
```

Since we're not going to be using our radio buttons anymore, I'm just passing in the year-object that we want to use by calling:

```
// Render the first year in our data
update(data[0]);
```

Finally, I'm going to remove the event listener that we set for our form inputs. If all has gone to plan, we should have a beautiful-looking chart for the first year in our data:



## Making it Dynamic

The next step is to have our visualization cycle between different years, showing how incomes have been changing over time. We'll do this by adding in call to JavaScript's `setInterval()` function, which we can use to execute some code repeatedly: `d3.csv("data/incomes.csv", prepareData).then(data => {`

```
...
```

```
function update(data) {
```

```
...
```

```
}
```

```
let time = 0;
```

```
let interval = setInterval(step, 200);
```

```
function step() {
```

```
  update(data[time]);
```

```
  time = (time == 49) ? 0 : time + 1; }
```

```
update(data[0]);
```

```
});
```

We're setting up a timer in this `time` variable, and every 200ms, this code will run the `step()` function, which will update our chart to the next year's data, and increment the timer by 1. If the timer is at a value of 49 (the last year in our data), it will reset itself. This now gives us a nice loop that will run continuously:

Year: 2009



Income Bracket	Household Income (2015 dollars)
Highest 20%	108,730
Second-Highest 20%	86,933
Middle 20%	64,720
Second-Lowest 20%	32,320
Lowest 20%	12,762
Average	75,093

To makes things a little more useful. I'll also add in some labels that give us the raw figures. I'll replace all of the HTML code in the body of our file with this:

```
<h2>Year: <span id="year"></span></h2>
```

```
<div class="container" id="page-main"> <div class="row">
<div class="col-md-7"> <div id="chart-area"></div> </div>
```

```
<div class="col-md-5"> <table class="table"> <tbody>
```

```
<tr>
```

```
<th></th> <th>Income Bracket</th> <th>Household Income (2015
dollars)</th> </tr>
```

```
<tr>
```

```
<td id="leg5"></td> <td>Highest 20%</td> <td class="money-cell">
<span id="fig5"></span></td> </tr>
```

```
<tr>
```

```
<td id="leg4"></td> <td>Second-Highest 20%</td> <td class="money-
cell"><span id="fig4"></span></td> </tr>
```

```
<tr>
```

```
<td id="leg3"></td> <td>Middle 20%</td> <td class="money-cell">
<span id="fig3"></span></td> </tr>
```

```
<tr>
```

```

<td id="leg2"></td> <td>Second-Lowest 20%</td> <td class="money-
cell"><span id="fig2"></span></td> </tr>

<tr>

<td id="leg1"></td> <td>Lowest 20%</td> <td class="money-cell">
<span id="fig1"></span></td> </tr>

</tbody>

<tfoot>

<tr>

<td id="avLeg"></td> <th>Average</th> <th class="money-cell"><span
id="avFig"></span></th> </tr>

</tfoot>

</table>

</div>

</div>

</div>

```

We're structuring our page here using [Bootstrap's grid system](#), which lets us neatly format our page elements into boxes.

I'll then update all of this with [jQuery](#) whenever our data changes: function

```

updateHTML(data) {

// Update title

$("#year").text(data.name);

// Update table values

$("#fig1").html(data.values[0].value.toLocaleString());
$("#fig2").html(data.values[1].value.toLocaleString());
$("#fig3").html(data.values[2].value.toLocaleString());
$("#fig4").html(data.values[3].value.toLocaleString());
$("#fig5").html(data.values[4].value.toLocaleString());
$("#avFig").html(data.average.toLocaleString()); }

```

```

d3.csv("data/incomes.csv", prepareData).then(data => {

```

```
...  
function update(data) {  
  updateHTML(data);  
  ...  
}  
...  
}
```

I'll also make a few edits to the CSS at the top of our file, which will give us a legend for each of our arcs, and also center our heading: <style> #chart-area  
svg {

```
margin:auto;  
display:inherit;  
}
```

```
.money-cell { text-align: right; }  
h2 { text-align: center; }
```

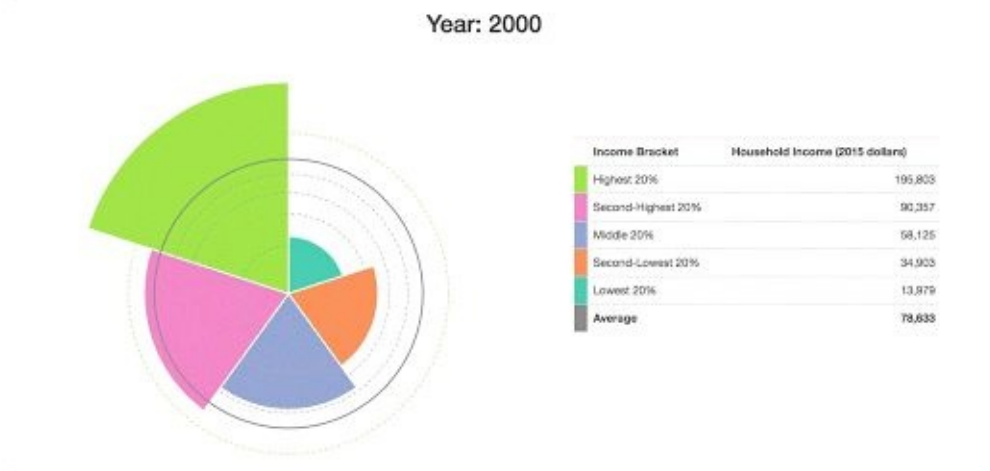
```
#leg1 { background-color: #66c2a5; }  
#leg2 { background-color: #fc8d62; }  
#leg3 { background-color: #8da0cb; }  
#leg4 { background-color: #e78ac3; }  
#leg5 { background-color: #a6d854; }  
#avLeg { background-color: grey; }
```

```
@media screen and (min-width: 768px) {  
  table { margin-top: 100px; }
```



```
}  
</style>
```

What we end up with is something rather presentable:



Since it's pretty tough to see how these arcs have changed over time here, I want to add in some grid lines to show what the income distribution looked like in the first year of our data: `d3.csv("data/incomes.csv", prepareData).then(data => {`

```
...
```

```
update(data[0]));
```

```
data[0].values.forEach((d, i) => {  
  svg.append("circle")  
    .attr("fill", "none")  
    .attr("cx", 0)  
    .attr("cy", 0)  
    .attr("r", radiusScale(d.value)) .attr("stroke", color(i))  
    .attr("stroke-dasharray", "4,4"); });  
});
```

I'm using the `Array.forEach()` method to accomplish this, although I could have also gone with D3's usual *General Update Pattern* again

(JOIN/EXIT/UPDATE/ENTER).

I also want to add in a line to show the average income in the US, which I'll update every year. First, I'll add the average line for the first time:

```
d3.csv("data/incomes.csv", prepareData).then(data => {  
  
  ...  
  
  data[0].values.forEach((d, i) => {  
    svg.append("circle")  
      .attr("fill", "none")  
      .attr("cx", 0)  
      .attr("cy", 0)  
      .attr("r", radiusScale(d.value)) .attr("stroke", color(i))  
      .attr("stroke-dasharray", "4,4"); });
```

```
    svg.append("circle")  
      .attr("class", "averageLine") .attr("fill", "none")  
      .attr("cx", 0)  
      .attr("cy", 0)  
      .attr("stroke", "grey")  
      .attr("stroke-width", "2px"); });
```

Then I'll update this at the end of our `update()` function whenever the year changes:

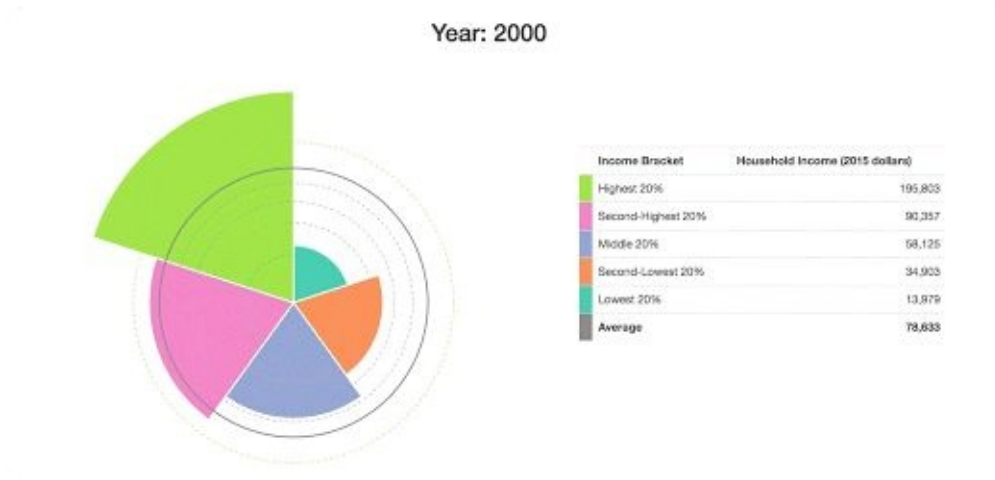
```
function update(data) {
```

```
  ...  
  
  svg.select(".averageLine").transition().duration(200) .attr("r",  
    radiusScale(data.average)); }
```

I should note that it's important for us to add each of these circles *after* our first call to `update()`, because otherwise they'll end up being rendered *behind* each of our arc paths (SVG layers are determined by the order in which they're added

to the screen, rather than by their z-index).

At this point, we have something that conveys the data that we're working with a bit more clearly:



## Making it Interactive

As a last step, I want us to add in some controls to let the user dig down into a particular year. I want to add in a *Play/Pause* button, as well as a year slider, allowing the user to pick a particular date to look at.

Here's the HTML that I'll use to add these elements onto the screen:

```
<div class="container" id="page-main">
  <div id="controls" class="row">
    <div class="col-md-12">
      <button id="play-button" class="btn btn-
primary">Play</button>
      <div id="slider-div">
        <label>Year: <span id="year-label"></span></label>
        <div id="date-slider"></div>
      </div>
    </div>
  </div>
  ...
</div>
```

We'll need to add some event listeners to both of these elements, to engineer the behavior that we're looking for.

First off, I want to define the behavior of our *Play/Pause* button. We'll need to replace the code that we wrote for our interval earlier to allow us to stop and start the timer with the button. I'll assume that the visualization starts in a "Paused" state, and that we need to press this button to kick things off.

```
function update(data) {
  ...

  let time = 0;
  let interval;

  function step() {
    update(data[time]);
    time = (time == 49) ? 0 : time + 1;
  }

  $("#play-button").on("click", function() {
    const button = $(this);
    if (button.text() === "Play"){
```

```

        button.text("Pause");
        interval = setInterval(step, 200);
    } else {
        button.text("Play");
        clearInterval(interval);
    }
});
...
}

```

Whenever our button gets clicked, our `if/else` block here is going to define a different behavior, depending on whether our button is a “Play” button or a “Pause” button. If the button that we’re clicking says “Play”, we’ll change the button to a “Pause” button, and start our interval loop going. Alternatively, if the button is a “Pause” button, we’ll change its text to “Play”, and we’ll use the `clearInterval()` function to stop the loop from running.

For our slider, I want to use the slider that comes with the [jQuery UI library](#). I’m including this in our HTML, and I’m going to write a few lines to add this to the screen:

```

function update(data) {
    ...
    $("#date-slider").slider({
        max: 49,
        min: 0,
        step: 1,
        slide: (event, ui) => {
            time = ui.value;
            update(data[time]);
        }
    });

    update(data[0]);
    ...
}

```

Here, we’re using the `slide` option to attach an event listener to the slider. Whenever our slider gets moved to another value, we’re updating our timer to this new value, and we’re running our `update()` function at that year in our data.

We can add this line at the end of our `update()` function so that our slider moves along to the right year when our loop is running:

```

function update(data) {

```

```
...  
  
// Update slider position  
$("#date-slider").slider("value", time);  
}
```

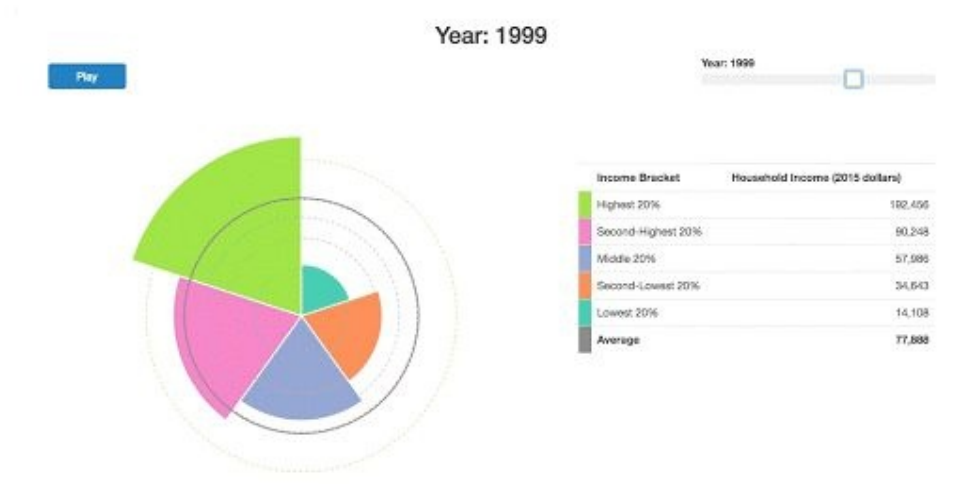
I'll also add in a line to our `updateHTML()` function (which runs whenever our visualization changes), which can adjust the value of the label based on the current year in the data:

```
function updateHTML(data) {  
  // Update title  
  $("#year").text(data.name);  
  
  // Update slider label  
  $("#year-label").text(data.name);  
  
  // Update table values  
  $("#fig1").html(data.values[0].value.toLocaleString());  
  ...  
}
```

I'll throw in a few more lines to our CSS to make everything look a little neater:

```
<style>  
...  
@media screen and (min-width: 768px) {  
  table { margin-top: 100px; }  
}  
  
#page-main { margin-top: 10px; }  
#controls { margin-bottom: 20px; }  
  
#play-button {  
  margin-top: 10px;  
  width: 100px;  
}  
  
#slider-div {  
  width: 300px;  
  float: right;  
}  
</style>
```

And there we have it — our finished product — a fully functioning interactive data visualization, with everything working as expected.



Hopefully, this tutorial demonstrated the real power of D3, letting you create absolutely anything you can imagine.

Getting started with D3 from scratch is always a tough process, but the rewards are worth it. If you want to learn how to create custom visualizations of your own, here are a few online resources that you might find helpful:

- An overview of [SitePoint's D3.js content](#).
- The [introduction to the library](#) on D3's homepage. This runs through some of the most basic commands, showing you how to make your first few steps in D3.
- “[Let's Make a Bar Chart](#)” by Mike Bostock — the creator of D3 — showing beginners how to make one of the simplest graphs in the library.
- [D3.js in Action by Elijah Meeks](#) (\$35), which is a solid introductory textbook that goes into a lot of detail.
- [D3's Slack channel](#) is very welcoming to newcomers to D3. It also has a “learning materials” section with a collection of great resources.
- [This online Udemy course](#) (\$20), which covers everything in the library in a series of video lectures. This is aimed at JavaScript developers, and includes four cool projects.
- The multitude of example visualizations that are available at [blocks.org](#) and [blockbuilder.org](#).
- The [D3 API Reference](#), which gives a thorough technical explanation of everything that D3 has to offer.

## Finished Code

Don't forget, if you want to see the finished version of the code that I was using in the chapter, then you can [find it on our GitHub repo](#).