

Build



5 Practical React Projects



5 Practical React Projects

Copyright © 2017 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
Email: 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

This book is a collection of in-depth tutorials, selected from SitePoint's [React Hub](#), that will guide you through some fun and practical projects. Along the way, you'll pick up lots of useful development tips.

Who Should Read This Book

This book is for developers with some React experience. If you're a novice, please read [Your First Week With React](#) before tackling this book.

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: How to Create a Reddit Clone Using React and Firebase

by Nirmalya Ghosh

In this article, we'll be using [Firebase](#) along with Create React App to build an app that will function similar to [Reddit](#). It will allow the user to submit a new link that can then be voted on.

Here's a [live demo](#) of what we'll be building.

Why Firebase?

Using Firebase will make it very easy for us to show real-time data to the user. Once a user votes on a link, the feedback will be instantaneous. Firebase's Realtime Database will help us in developing this feature. Also, it will help us to understand how to bootstrap a React application with Firebase.

Why React?

React is particularly known for creating user interfaces using a component architecture. Each component can contain internal [state](#) or be passed data as [props](#). State and props are the two most important concepts in React. These two things help us determine the state of our application at any point in time. If you're not familiar with these terms, please head over to the [React docs](#) first.

Using a State Container

Note: you can also use a state container like [Redux](#) or [MobX](#), but for the sake of simplicity, we won't be using one for this tutorial.

The whole project is [available on GitHub](#).

Setting up the Project

Let's walk through the steps to set up our project structure and any necessary dependencies.

Installing create-react-app

If you haven't already, you need to install **create-react-app**. To do so, you can type the following in your terminal:

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

Once you've installed it globally, you can use it to scaffold a React project inside any folder.

Now, let's create a new app and call it **reddit-clone**.

```
create-react-app reddit-clone
```

This will scaffold a new **create-react-app** project inside the **reddit-clone** folder. Once the bootstrapping is done, we can go inside **reddit-clone** directory and fire up the development server:

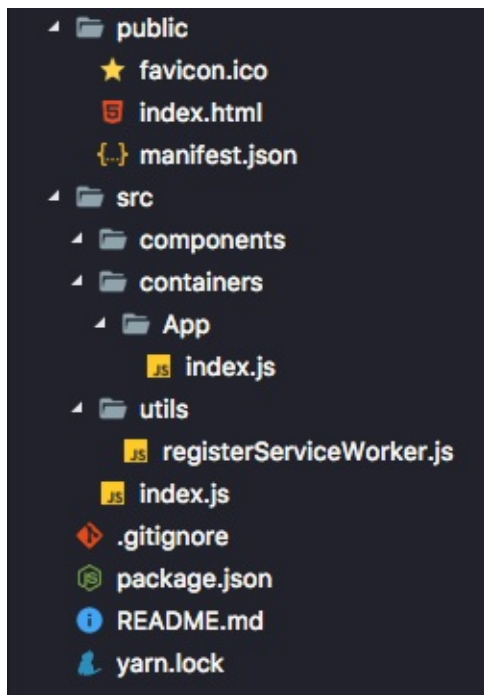
```
npm start
```

At this point, you can go to <http://localhost:3000/> and see your app skeleton up and running.

Structuring the app

For maintenance, I always like to separate my **containers** and **components**. Containers are the smart components that contain the business logic of our application and manage Ajax requests. Components are simply dumb presentational components. They can have their own internal state, which can be used to control the logic of that component (e.g. showing the current state of a [controlled](#) input component).

After removing the unnecessary logo and CSS files, this is how your app should look now. We created a components folder and a containers folder. Let's move App.js inside the containers/App folder and create registerServiceWorker.js inside the utils folder.



Your src/containers/App/index.js file should look like this:

```
// src/containers/App/index.js

import React, { Component } from 'react';

class App extends Component {
  render() {
    return (
      <div className="App">
        Hello World
      </div>
    );
  }
}

export default App;
```

Your src/index.js file should look like this:

```
// src/index.js

import React from 'react';
import ReactDOM from 'react-dom';
import App from './containers/App';
import registerServiceWorker from './utils/registerServiceWorker';

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



```
registerServiceWorker();
```

Go to your browser, and if everything works fine, you'll see **Hello World** on your screen.

You can check my [commit](#) on GitHub.

Adding react-router

[React-router](#) will help us define the routes for our app. It's very customizable and very popular in the React ecosystem.

We'll be using version **3.0.0** of **react-router**.

```
npm install --save react-router@3.0.0
```

Now, add a new file `routes.js` inside the `src` folder with the following code:

```
// routes.js

import React from 'react';
import { Router, Route } from 'react-router';

import App from './containers/App';

const Routes = (props) => (
  <Router {...props}>
    <Route path="/" component={ App }>
    </Route>
  </Router>
);

export default Routes;
```

The Router component wraps all the Route components. Based on the path prop of the Route component, the component passed to the component prop will be rendered on the page. Here, we're setting up the root URL (/) to load our App component using the Router component.

```
<Router {...props}>
  <Route path="/" component={ <div>Hello World!</div> }>
  </Route>
</Router>
```

The above code is also valid. For the path /, the `<div>Hello world!</div>` will be mounted.

Now, we need to call our `routes.js` file from our `src/index.js` file. The file should have the following content:

```
// src/index.js

import React from 'react';
import ReactDOM from 'react-dom';
import { browserHistory } from 'react-router';

import App from './containers/App';
import Routes from './routes';
import registerServiceWorker from './utils/registerServiceWorker';

ReactDOM.render(
  <Routes history={browserHistory} />,
  document.getElementById('root')
);

registerServiceWorker();
```

Basically, we're mounting our Router component from our `routes.js` file. We pass in the `history` prop to it so that the routes know how to handle [history tracking](#).

You can check my [commit](#) on GitHub.

Adding Firebase

If you don't have a [Firebase](#) account, create one now (it's free!) by going to their website. After you're done creating a new account, log in to your account and go to the [console](#) page and click on **Add project**.

Enter the name of your project (I'll call mine **reddit-clone**), choose your country, and click on the **Create project** button.

Now, before we proceed, we need to change the **rules** for the database since, by default, Firebase expects the user to be authenticated to be able to read and write data. If you select your project and click on the **Database** tab on the left, you'll be able to see your database. You need to click on the **Rules** tab on the top that will redirect us to a screen which will have the following data:

```
{
  "rules": {
    ".read": "auth != null",
    ".write": "auth != null"
  }
}
```

We need to change this to the following:

```
{
  "rules": {
    ".read": "auth === null",
    ".write": "auth === null"
  }
}
```

This will let users update the database without logging in. If we implemented a flow in which we had authentication before making updates to the database, we would need the default rules provided by Firebase. To keep this application simple, we *won't* be doing authentication.

You Must Make This Modification

If you don't make this modification, Firebase won't let you update the database from your app.

Now, let's add the `firebase` npm module to our app by running the following code:

```
npm install --save firebase
```

Next, import that module in your `App/index.js` file as:

```
// App/index.js
import * as firebase from "firebase";
```

When we select our project after logging in to Firebase, we'll get an option **Add Firebase to your web app**.

Welcome to Firebase! Get started here.



Add Firebase to
your iOS app



Add Firebase to
your Android app



Add Firebase to
your web app

If we click on that option, a modal will appear that will show us the config variable which we will use in our `componentWillMount` method.

Add Firebase to your web app



Copy and paste the snippet below at the bottom of your HTML, before other script tags.

```
<script src="https://www.gstatic.com/firebasejs/4.1.2/firebase.js"></script>
<script>
  // Initialize Firebase
  var config = {
    apiKey: "AIzaSyBRExKF0cHylh_wFLcd8Vxugj0UQRpq8oc",
    authDomain: "reddit-clone-53da5.firebaseio.com",
    databaseURL: "https://reddit-clone-53da5.firebaseio.com",
    projectId: "reddit-clone-53da5",
    storageBucket: "reddit-clone-53da5.appspot.com",
    messagingSenderId: "490290211297"
  };
  firebase.initializeApp(config);
</script>
```

COPY

Check these resources to
learn more about Firebase for
web apps:

[Get Started with Firebase for Web Apps](#)

[Firebase Web SDK API Reference](#)

[Firebase Web Samples](#)

Let's create the Firebase config file. We'll call this file `firebase-config.js`, and it will contain all the configs necessary to connect our app with Firebase:

```
// App/firebase-config.js

export default {
  apiKey: "AIzaSyBRExKF0cHylh_wFLcd8Vxugj0UQRpq8oc",
  authDomain: "reddit-clone-53da5.firebaseio.com",
  databaseURL: "https://reddit-clone-53da5.firebaseio.com",
  projectId: "reddit-clone-53da5",
  storageBucket: "reddit-clone-53da5.appspot.com",
  messagingSenderId: "490290211297"
};
```

We'll import our Firebase config into App/index.js:

```
// App/index.js

import config from './firebase-config';
```

We'll initialize our Firebase database connection in the constructor.

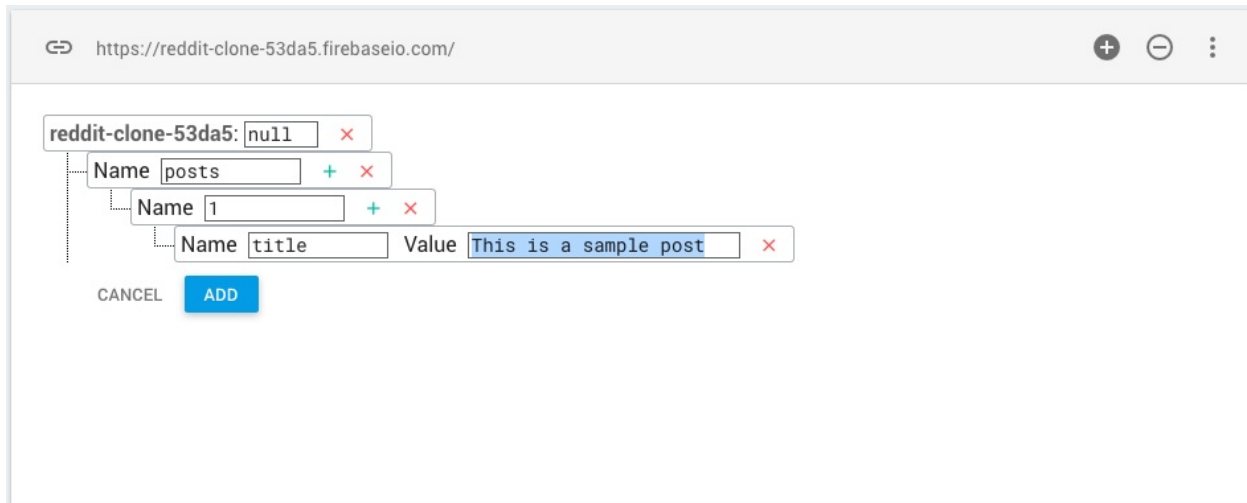
```
// App/index.js

constructor() {
  super();

  // Initialize Firebase
  firebase.initializeApp(config);
}
```

In the `componentWillMount()` lifecycle hook, we use the package `firebase` we just installed and call its `initializeApp` method and passed the `config` variable to it. This object contains all the data about our app. The `initializeApp` method will connect our application to our Firebase database so that we can read and write data.

Let's add some data to Firebase to check if our configuration is correct. Go to the *Database* tab and add the following structure to your database:



Clicking on *Add* will save the data to our database.



Now, let's add some code to our `componentWillMount` method to make the data appear on our screen:

```
// App/index.js

componentWillMount() {
  ...

  let postsRef = firebase.database().ref('posts');

  let _this = this;

  postsRef.on('value', function(snapshot) {
    console.log(snapshot.val());
  });
}
```

```

    _this.setState({
      posts: snapshot.val(),
      loading: false
    });
  });
}

```

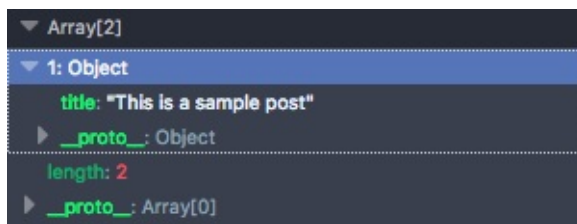
`firebase.database()` gives us a reference to the database service. Using `ref()`, we can get a specific reference from the database. For example, if we call `ref('posts')`, we'll be getting the `posts` reference from our database and storing that reference in `postsRef`.

`postsRef.on('value', ...)` gives us the updated value whenever there's any change in the database. This is very useful when we need a real-time update to our user interface based on any database events.

Using `postsRef.once('value', ...)` will only give us the data once. This is useful for data that only needs to be loaded once and isn't expected to change frequently or require active listening.

After we get the updated value in our `on()` callback, we store the values in our `posts` state.

Now we'll see the data appearing on our console.



Also, we'll be passing this data down to our children. So, we need to modify the render function of our `App/index.js` file:

```

// App/index.js

render() {
  return (
    <div className="App">
      {this.props.children && React.cloneElement(
        this.props.children, {
          firebaseRef: firebase.database().ref('posts'),
          posts: this.state.posts,

```

```
        loading: this.state.loading
      })}
    </div>
  );
}
```

The main objective here is to make the posts data available in all our children components, which will be passed through react-router.

We're checking if `this.props.children` exists or not, and if it exists we clone that element and pass all our props to all our children. This is a very efficient way of passing props to dynamic children.

Calling [cloneElement](#) will shallowly merge the already existing props in `this.props.children` and the props we passed here (`firebaseRef`, `posts` and `loading`).

Using this technique, the `firebaseRef`, `posts` and `loading` props will be available to all routes.

You can check my [commit](#) on GitHub.

Connecting the App with Firebase

Firebase can only store data as objects; [it doesn't have any native support for arrays](#). We'll store the data in the following format:



Add the data in the screenshot above manually so that you can test your views.

Add views for all the posts

Now we'll add views to show all the posts. Create a file `src/containers/Posts/index.js` with the following content:

```
// src/containers/Posts/index.js

import React, { Component } from 'react';

class Posts extends Component {
  render() {
    if (this.props.loading) {
      return (
        <div>
          Loading...
        </div>
      );
    }

    return (
      <div className="Posts">
        { this.props.posts.map((post) => {
          return (
            <div>
              { post.title }
            </div>
          );
        }) }
      </div>
    );
  }
}

export default Posts;
```

Here, we're just mapping over the data and rendering it to the user interface.

Next, we need to add this to our `routes.js` file:

```
// routes.js

...
<Router {...props}>
  <Route path="/" component={ App }>
    <Route path="/posts" component={ Posts } />
  </Route>
</Router>
```

...

This is because we want the posts to show up only on the /posts route. So we just pass the Posts component to the component prop and /posts to the path prop of the Route component of react-router.

If we go to the URL localhost:3000/posts, we'll see the posts from our Firebase database.

You can check my [commit](#) on GitHub.

Add views to write a new post

Now, let's create a view from where we can add a new post. Create a file `src/containers/AddPost/index.js` with the following content:

```
// src/containers/AddPost/index.js

import React, { Component } from 'react';

class AddPost extends Component {
  constructor() {
    super();

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  state = {
    title: ''
  };

  handleChange = (e) => {
    this.setState({
      title: e.target.value
    });
  }

  handleSubmit = (e) => {
    e.preventDefault();

    this.props.firebaseRef.push({
      title: this.state.title
    });

    this.setState({
```

```

        title: ''
      });
    }

    render() {
      return (
        <div className="AddPost">
          <input
            type="text"
            placeholder="Write the title of your post"
            onChange={ this.handleChange }
            value={ this.state.title }
          />
          <button
            type="submit"
            onClick={ this.handleSubmit }
          >
            Submit
          </button>
        </div>
      );
    }
  }

export default AddPost;

```

Here, the `handleChange` method updates our state with the value present in the input box. Now, when we click on the button, the `handleSubmit` method is triggered. The `handleSubmit` method is responsible for making the API request to write to our database. We do it using the `firebaseRef` prop that we passed to all the children.

```

this.props.firebaseRef.push({
  title: this.state.title
});

```

The above block of code sets the current value of the title to our database.

After the new post has been stored in the database, we make the input box empty again, ready to add a new post.

Now we need to add this page to our routes:

```

// routes.js

import React from 'react';

```

```

import { Router, Route } from 'react-router';

import App from '../containers/App';
import Posts from '../containers/Posts';
import AddPost from '../containers/AddPost';

const Routes = (props) => (
  <Router {...props}>
    <Route path="/" component={ App }>
      <Route path="/posts" component={ Posts } />
      <Route path="/add-post" component={ AddPost } />
    </Route>
  </Router>
);

export default Routes;

```

Here, we just added the /add-post route so that we can add a new post from that route. Hence, we passed the AddPost component to its component prop.

Also, let's modify the render method of our src/containers/Posts/index.js file so that it can iterate over objects instead of arrays (since Firebase doesn't store arrays).

```

// src/containers/Posts/index.js

render() {
  let posts = this.props.posts;

  if (this.props.loading) {
    return (
      <div>
        Loading...
      </div>
    );
  }

  return (
    <div className="Posts">
      { Object.keys(posts).map(function(key) {
        return (
          <div key={key}>
            { posts[key].title }
          </div>
        );
      })}
    </div>
  );
}

```

```
}
```

Now, if we go to localhost:3000/add-post, we can add a new post. After clicking on the **submit** button, the new post will appear immediately on the [posts page](#).

You can check my [commit](#) on GitHub.

Implement voting

Now we need to allow users to vote on a post. For that, let's modify the render method of our `src/containers/App/index.js`:

```
// src/containers/App/index.js

render() {
  return (
    <div className="App">
      {this.props.children && React.cloneElement(this.props.
        ↪children, {
          // https://github.com/ReactTraining/react-router/blob/v3/
          ↪examples/passing-props-to-children/app.js#L56-L58
          firebase: firebase.database(),
          posts: this.state.posts,
          loading: this.state.loading
        })}
    </div>
  );
}
```

We changed the `firebase` prop from `firebaseRef`: `firebase.database().ref('posts')` to `firebase: firebase.database()` because we'll be using Firebase's [set](#) method to update our voting count. In this way, if we had more Firebase refs, it would be very easy for us to handle them by using only the `firebase` prop.

Before proceeding with the voting, let's modify the `handleSubmit` method in our `src/containers/AddPost/index.js` file a little bit:

```
// src/containers/AddPost/index.js

handleSubmit = (e) => {
  ...
  this.props.firebase.ref('posts').push({
    title: this.state.title,
    upvote: 0,
  });
}
```

```

        downvote: 0
      });
      ...
    }
  }

```

We renamed our `firebaseRef` prop to `firebase` prop. So, we change the `this.props.firebaseRef.push` to `this.props.firebase.ref('posts').push`.

Now we need to modify our `src/containers/Posts/index.js` file to accommodate the voting.

The render method should be modified to this:

```

// src/containers/Posts/index.js

render() {
  let posts = this.props.posts;
  let _this = this;

  if (!posts) {
    return false;
  }

  if (this.props.loading) {
    return (
      <div>
        Loading...
      </div>
    );
  }

  return (
    <div className="Posts">
      { Object.keys(posts).map(function(key) {
        return (
          <div key={key}>
            <div>Title: { posts[key].title }</div>
            <div>Upvotes: { posts[key].upvote }</div>
            <div>Downvotes: { posts[key].downvote }</div>
            <div>
              <button
                onClick={ _this.handleUpvote.bind(this,
                  posts[key], key) }
                type="button"
              >
                Upvote
              </button>
              <button

```

```

        onClick={_this.handleDownvote.bind(this,
            ↪posts[key], key) }
        type="button"
      >
        Downvote
      </button>
    </div>
  </div>
);
  })}
</div>
);
}

```

When the buttons are clicked, the **upvote** or **downvote** count will be incremented in our Firebase DB. To handle that logic, we create two more methods: `handleUpvote()` and `handleDownvote()`:

```

// src/containers/Posts/index.js

handleUpvote = (post, key) => {
  this.props.firebase.ref('posts/' + key).set({
    title: post.title,
    upvote: post.upvote + 1,
    downvote: post.downvote
  });
}

handleDownvote = (post, key) => {
  this.props.firebase.ref('posts/' + key).set({
    title: post.title,
    upvote: post.upvote,
    downvote: post.downvote + 1
  });
}

```

In these two methods, whenever a user clicks on either of the buttons, the respective count is incremented in the database and is instantly updated in the browser.

If we open two tabs with localhost:3000/posts and click on the voting buttons of the posts, we'll see each of the tabs get updated almost instantly. This is the magic of using a real-time database like Firebase.

You can check my [commit](#) on GitHub.

In the [repository](#), I've added the /posts route to the IndexRoute of the application just to show the posts on [localhost:3000](#) by default. You can check that [commit](#) on GitHub.

Conclusion

The end result is admittedly a bit barebones, as we didn't try to implement any design (although [the demo](#) has some basic styles added). We also didn't add any authentication, in order to reduce the complexity and the length of the tutorial, but obviously any real-world application would require it.

Firebase is really useful for places where you don't want to create and maintain a separate back-end application, or where you want real-time data without investing too much time developing your APIs. It plays really well with React, as you can hopefully see from the article.

Further reading

- [Getting React Projects Ready Fast with Pre-configured Builds](#)
- [Build a React Application with User Login and Authentication](#)
- [Firebase Authentication for Web](#)
- [Leveling Up With React: React Router](#)

Chapter 2: Build a CRUD App Using React, Redux and FeathersJS

by Michael Wanyoike

Building a modern project requires splitting the logic into front-end and back-end code. The reason behind this move is to promote code re-usability. For example, we may need to build a native mobile application that accesses the back-end API. Or we may be developing a module that will be part of a large modular platform.

The popular way of building a server-side API is to use a library like Express or Restify. These libraries make creating RESTful routes easy. The problem with these libraries is that we'll find ourselves writing a *ton* of **repeating code**. We'll also need to write code for authorization and other middleware logic.

To escape this dilemma, we can use a framework like [Loopback](#) or [Feathers](#) to help us generate an API.

At the time of writing, Loopback has more GitHub stars and downloads than Feathers. Loopback is a great library for generating RESTful CRUD endpoints in a short period of time. However, it does have a slight learning curve and the [documentation](#) is not easy to get along with. It has stringent framework requirements. For example, all models must inherit one of its built-in model class. If you need real-time capabilities in Loopback, be prepared to do some additional coding to make it work.

FeathersJS, on the other hand, is much easier to get started with and has realtime support built-in. Quite recently, the Auk version was released (because Feathers is so modular, they use bird names for version names) which introduced a vast number of changes and improvements in a number of areas. According to a [post](#) they published on their blog, they are now the **4th most popular real-time web framework**. It has excellent [documentation](#), and they've covered pretty much

any area we can think of on building a real-time API.

What makes Feathers amazing is its simplicity. The entire framework is modular and we only need to install the features we need. Feathers itself is a thin wrapper built on top of [Express](#), where they've added new features – [services](#) and [hooks](#). Feathers also allows us to effortlessly send and receive data over WebSockets.

Prerequisites

Before you get started with the tutorial, you'll need to have a solid foundation in the following topics:

- How to write [ES6 JavaScript code](#)
- How to create [React](#) components
- [Immutability in JavaScript](#)
- How to [manage state with Redux](#)

On your machine, you'll need to have installed recent versions of:

- NodeJS 6+
- [Mongodb 3.4+](#)
- [Yarn](#) package manager (optional)
- Chrome browser

If you've never written a database API in JavaScript before, I'd recommend first taking a look at [this tutorial on creating RESTful APIs](#).

Scaffold the App

We're going to build a CRUD contact manager application using [React](#), [Redux](#), [Feathers](#) and [MongoDB](#). You can take a look at the completed project [here](#).

In this tutorial, I'll show you how to build the application from the bottom up. We'll kick-start our project using the [create-react-app](#) tool.

```
# scaffold a new react project create-react-app react-contact-manage

cd react-contact-manager

# delete unnecessary files

rm src/logo.svg src/App.css
```

Use your favorite code editor and remove all the content in index.css. Open App.js and rewrite the code like this: `import React, { Component } from 'react';`

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

```
return (  
  <div>  
    <h1>Contact Manager</h1>  
  </div>  
  );  
}
```

```
export default App;
```

Make sure to run `yarn start` to ensure the project is running as expected. Check the console tab to ensure that our project is running cleanly with no warnings or errors. If everything is running smoothly, use `Ctrl+C` to stop the server.

Build the API Server with Feathers

Let's proceed with generating the back-end API for our CRUD project using the `feathers-cli` tool.

```
# Install Feathers command-line tool npm install -g feathers-cli
```

```
# Create directory for the back-end code
```

```
mkdir backend
```

```
cd backend
```

```
# Generate a feathers back-end API server
```

```
feathers generate app
```

```
? Project name | backend
```

```
? Description | contacts API server
```

```
? What folder should the source files live in? | src
```

```
? Which package manager are you using (has to be installed ↪globally
```

```
? What type of API are you making? | REST, Realtime via Socket.io
```

```
# Generate RESTful routes for Contact Model
```

```
feathers generate service
```

```
? What kind of service is it? | Mongoose
```

```
? What is the name of the service? | contact
```

```
? Which path should the service be registered on? | /contacts ? What
```

```
→mongodb://localhost:27017/backend
```

```
# Install email field type
```

```
yarn add mongoose-type-email
```

```
# Install the nodemon package
```

```
yarn add nodemon --dev
```

Open backend/package.json and update the start script to use [nodemon](#) so that the API server will restart automatically whenever we make changes.

```
// backend/package.json
```

```
...
```

```
"scripts": {  
  
  ...  
  
  "start": "nodemon src/",  
  
  ...  
},  
  
...
```

Let's open `backend/config/default.json`. This is where we can configure MongoDB connection parameters and other settings. I've also increased the default paginate value to 50, since in this tutorial we won't write front-end logic to deal with pagination.

```
{  
  
  "host": "localhost",  
  
  "port": 3030,  
  
  "public": "../public/",  
  
  "paginate": {  
  
    "default": 50,
```

```
    "max": 50

  },

  "mongodb": "mongodb://localhost:27017/backend"

}
```

Open backend/src/models/contact.model.js and update the code as follows:
// backend/src/models/contact.model.js

```
require('mongoose-type-email');

module.exports = function (app) {
  const mongooseClient = app.get('mongooseClient');
  const contact = new mongooseClient.Schema({
    name : {
      first: {
        type: String,
        required: [true, 'First Name is required']
      },
      last: {
        type: String,
        required: false
      }
    },
    email : {
```



```

    type: mongooseClient.SchemaTypes.Email,
    required: [true, 'Email is required']
  },
  phone : {
    type: String,
    required: [true, 'Phone is required'],
    validate: {
      validator: function(v) {
        return /^\\+(?:[0-9] ?){6,14}[0-9]$/.test(v);
      },
      message: '{VALUE} is not a
        ↪valid international phone number!'
    }
  },
  createdAt: { type: Date, 'default': Date.now },
  updatedAt: { type: Date, 'default': Date.now }
});

return mongooseClient.model('contact', contact);
};

```

In addition to generating the contact service, Feathers has also generated a test case for us. We need to fix the service name first for it to pass: `// backend/test/services/contact.test.js`

```

const assert = require('assert');
const app = require('../src/app');

describe('\\'contact\\' service', () => {

```

```

it('registered the service', () => {
  const service = app.service('contacts'); // change
  ➔contact to contacts

  assert.ok(service, 'Registered the service');
});
});

```

Open a new terminal and inside the backend directory, execute `yarn test`. You should have all the tests running successfully. Go ahead and execute `yarn start` to start the backend server. Once the server has finished starting it should print the line: 'Feathers application started on localhost:3030'.

Launch your browser and access the url: <http://localhost:3030/contacts>. You should expect to receive the following JSON response:
`{"total":0,"limit":50,"skip":0,"data":[]}`

Now let's use Postman to confirm all CRUD restful routes are working. You can launch Postman using [this link](#).

If you're new to Postman, check out this [tutorial](#). When you hit the SEND button, you should get your data back as the response along with three additional fields – `_id`, `createdAt` and `updatedAt`.

Use the following JSON data to make a POST request using Postman. Paste this in the body and set content-type to `application/json`: {

```

"name": {
  "first": "Tony",
  "last": "Stark"
},
"phone": "+18138683770",
"email": "tony@starkenterprises.com"
}

```

Build the UI

Let's start by installing the necessary front-end dependencies. We'll use [semantic-ui css](#)/[semantic-ui react](#) to style our pages and [react-router-dom](#) to handle route navigation.

Where To Install

Make sure you are installing *outside* the backend directory.

```
// Install semantic-ui yarn add semantic-ui-css semantic-ui-react

// Install react-router

yarn add react-router-dom
```

Update the project structure by adding the following directories and files: | --
react-contact-manager | -- backend

```
| -- node_modules
| -- public
| -- src
| -- App.js
| -- App.test.js
| -- index.css
| -- index.js
| -- components
| | -- contact-form.js #(new)
```

```
| |-- contact-list.js #(new)
|-- pages
|-- contact-form-page.js #(new)
|-- contact-list-page.js #(new)
```

Let's quickly populate the JS files with some placeholder code.

For the component `contact-list.js`, we'll write it in this syntax since it will be a purely presentational component.

```
// src/components/contact-list.js

import React from 'react';

export default function ContactList(){

  return (

    <div>

      <p>No contacts here</p>

    </div>

  )

}
```

For the top-level containers, I use pages. Let's provide some code for the `contact-list-page.js`

```
// src/pages/contact-list-page.js

import React, { Component } from 'react';

import ContactList from '../components/contact-list';

class ContactListPage extends Component {

  render() {

    return (

      <div>

        <h1>List of Contacts</h1>

        <ContactList/>

      </div>

    )

  }

}
```

```
}  
  
}  
  
export default ContactListPage;
```

For the contact-form component, it needs to be smart, since it's required to manage its own state, specifically form fields. For now, we'll place this placeholder code.

```
// src/components/contact-form.js import React, { Component } from 'react'  
  
class ContactForm extends Component {  
  
  render() {  
  
    return (  
  
      <div>  
  
        <p>Form under construction</p>  
  
      </div>  
  
    )  
  
  }  
  
}
```

```
)  
  
}  
  
}  
  
export default ContactForm;
```

Populate the contact-form-page with this code: // src/pages/contact-form-page.js

```
import React, { Component } from 'react';  
import ContactForm from '../components/contact-form';  
  
class ContactFormPage extends Component {  
  render() {  
    return (  
      <div>  
        <ContactForm/>  
      </div>  
    )  
  }  
}  
  
export default ContactFormPage;
```

Now, let's create the navigation menu and define the routes for our App. App.js is often referred to as the 'layout template' for the Single Page Application.

```
// src/App.js

import React, { Component } from 'react';

import { NavLink, Route } from 'react-router-dom';

import { Container } from 'semantic-ui-react';

import ContactListPage from '../pages/contact-list-page'; import ContactFormPage from '../pages/contact-form-page';

class App extends Component {

  render() {

    return (

      <Container>

        <div className="ui two item menu">

          <NavLink className="item" activeClassName="active"

            to="/" exact to="/">

              Home

            </NavLink>

          <NavLink className="item" activeClassName="active"

            to="/contacts" exact to="/contacts">

              Contacts List

            </NavLink>

          <NavLink className="item" activeClassName="active"

            to="/contact-form" exact to="/contact-form">

              Add Contact

            </NavLink>

        </div>

        <Route>

          <Route path="/" exact component={ContactListPage} />

          <Route path="/contacts" exact component={ContactListPage} />

          <Route path="/contact-form" exact component={ContactFormPage} />

        </Route>

      </Container>

    );

  }

}
```



```
</NavLink>
```

```
<NavLink className="item" activeClassName="active"
```

```
  >exact to="/contacts/new">
```

```
    Add Contact
```

```
</NavLink>
```

```
</div>
```

```
<Route exact path="/" component={ContactListPage}/> <Route p
```

```
  >component={ContactFormPage}/>
```

```
</Container>
```

```
);
```

```
}
```

```
}
```

```
export default App;
```

Finally, update the `index.js` file with this code where we import semantic-ui CSS for styling and BrowserRouter for using the HTML5 history API that keeps our app in sync with the URL.

```
// src/index.js

import React from 'react';

import ReactDOM from 'react-dom';

import { BrowserRouter } from 'react-router-dom';

import App from './App';

import 'semantic-ui-css/semantic.min.css';

import './index.css';

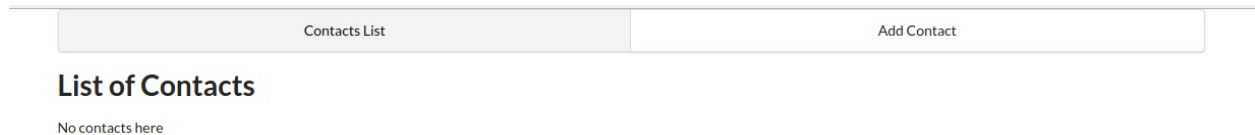
ReactDOM.render(

  <BrowserRouter>

    <App />
```

```
</BrowserRouter>,  
  
document.getElementById('root')  
  
);
```

Go back to the terminal and execute `yarn start`. You should have a similar view to the screenshot below:



Manage React State with Redux

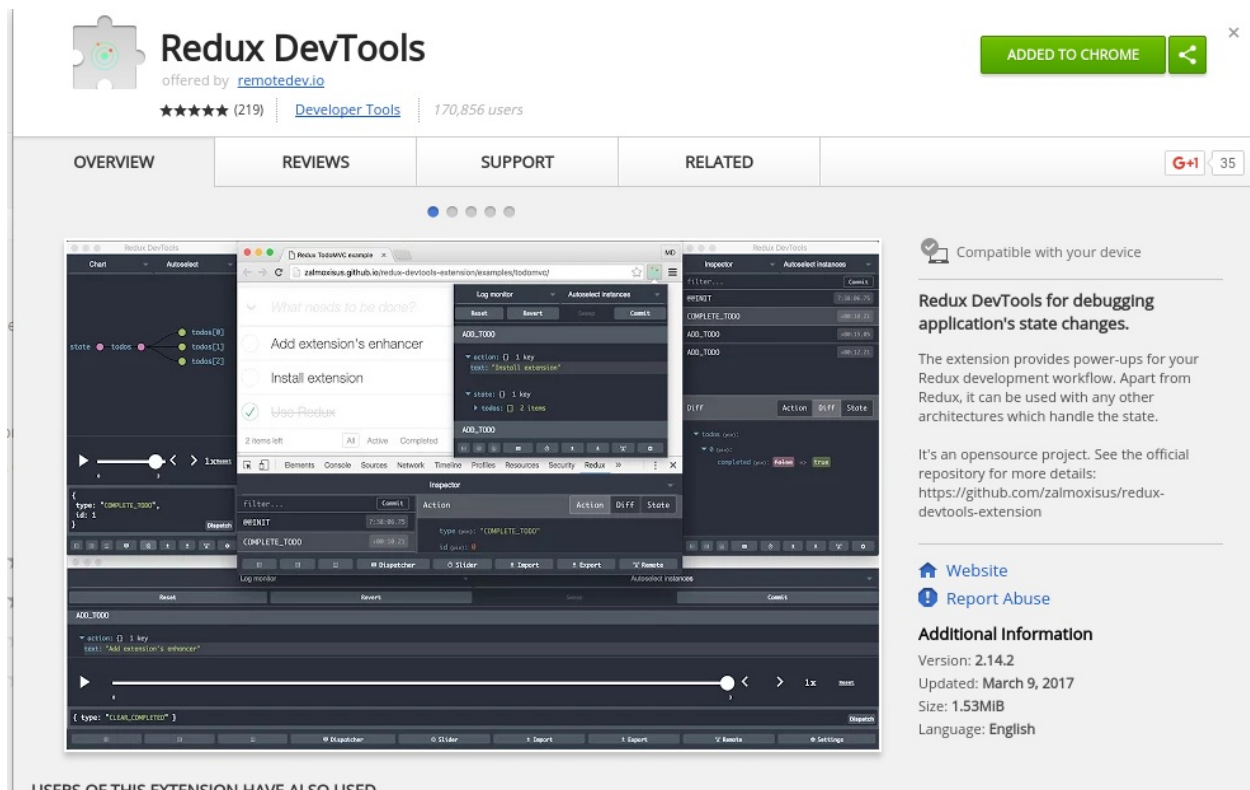
Stop the server with `ctrl+c` and install the following packages using yarn package manager: `yarn add redux react-redux redux-promise-middleware redux-thunk ➔redux-devtools-extension axios`

Phew! That's a whole bunch of packages for setting up Redux. I assume you are already familiar with Redux if you're reading this tutorial. [Redux-thunk](#) allows writing action creators as async functions while [redux-promise-middleware](#) reduces some Redux boilerplate code for us by handling dispatching of pending, fulfilled, and rejected actions on our behalf.

Feathers does include a light-weight client package that helps communicate with the API, but it's also really easy to use other client packages. For this tutorial, we'll use the [Axios](#) HTTP client.

The [redux-devtools-extension](#) an amazing tool that keeps track of dispatched

actions and state changes. You'll need to install its [chrome extension](#) for it to work.



Redux DevTools
offered by [remotedev.io](#)
★★★★★ (219) | [Developer Tools](#) | 170,856 users

ADDED TO CHROME

OVERVIEW | REVIEWS | SUPPORT | RELATED

Compatible with your device

Redux DevTools for debugging application's state changes.

The extension provides power-ups for your Redux development workflow. Apart from Redux, it can be used with any other architectures which handle the state.

It's an opensource project. See the official repository for more details:
<https://github.com/zalmoxisus/redux-devtools-extension>

[Website](#)
[Report Abuse](#)

Additional Information
Version: 2.14.2
Updated: March 9, 2017
Size: 1.53MiB
Language: English

Next, let's setup our Redux directory structure as follows: | -- react-contact-manager | -- backend

```
| -- node_modules
| -- public
| -- src
| -- App.js
| -- App.test.js
| -- index.css
| -- index.js
| -- contact-data.js #new
| -- store.js #new
| -- actions #new
```

```
|-- contact-actions.js #new
|-- index.js #new
|-- components
|-- pages
|-- reducers #new
|-- contact-reducer.js #new
|-- index.js #new
```

Let's start by populating `contacts-data.js` with some test data: `// src/contact-data.js`

```
export const contacts = [
{
  _id: "1",
  name: {
    first:"John",
    last:"Doe"
  },
  phone:"555",
  email:"john@gmail.com"
},
{
  _id: "2",
  name: {
    first:"Bruce",
    last:"Wayne"
  },
  phone:"777",
  email:"bruce.wayne@gmail.com"
```

```
}  
];
```

Define `contact-actions.js` with the following code. For now, we'll fetch data from the `contacts-data.js` file.

```
// src/actions/contact-actions.js  
  
import { contacts } from '../contacts-data';  
  
export function fetchContacts(){  
  
  return dispatch => {  
  
    dispatch({  
  
      type: 'FETCH_CONTACTS',  
  
      payload: contacts  
  
    })  
  
  }  
  
}
```

In `contact-reducer.js`, let's write our handler for the `'FETCH_CONTACT'` action. We'll store the contacts data in an array called `'contacts'`.

```
// src/reducers/contact-reducer.js

const defaultState = {

  contacts: []

}

export default (state=defaultState, action={}) => {

  switch (action.type) {

    case 'FETCH_CONTACTS': {

      return {

        ...state,

        contacts: action.payload

      }

    }

  }

}
```

```
    default:

      return state;

  }
}
```

In reducers/index.js, we'll combine all reducers here for easy export to our Redux store.

```
// src/reducers/index.js

import { combineReducers } from 'redux';

import ContactReducer from './contact-reducer';

const reducers = {

  contactStore: ContactReducer

}
```



```
const rootReducer = combineReducers(reducers);

export default rootReducer;
```

In `store.js`, we'll import the necessary dependencies to construct our Redux store. We'll also set up the `redux-devtools-extension` here to enable us to monitor the Redux store using the [Chrome extension](#).

```
// src/store.js

import { applyMiddleware, createStore } from "redux";

import thunk from "redux-thunk";

import promise from "redux-promise-middleware";

import { composeWithDevTools } from 'redux-devtools-extension'; import

const middleware = composeWithDevTools(applyMiddleware(promise(), ↵

export default createStore(rootReducer, middleware);
```

Open `index.js` and update the render method where we inject the store using Redux's Provider class.

```
// src/index.js

import React from 'react';

import ReactDOM from 'react-dom';

import { BrowserRouter } from 'react-router-dom';

import { Provider } from 'react-redux';

import App from './App';

import store from './store'

import 'semantic-ui-css/semantic.min.css';

import './index.css';

ReactDOM.render(

  <BrowserRouter>
```

```
<Provider store={store}>

  <App />

</Provider>

</BrowserRouter>,

document.getElementById('root')

);
```

Let's run `yarn start` to make sure everything is running so far.

Next, we'll **connect** our component `contact-list` with the Redux store we just created. Open `contact-list-page` and update the code as follows: `// src/pages/contact-list-page`

```
import React, { Component } from 'react';
import { connect } from 'react-redux';
import ContactList from '../components/contact-list';
import { fetchContacts } from '../actions/contact-actions';
class ContactListPage extends Component {

  componentDidMount() {
    this.props.fetchContacts();
  }
}
```

```

render() {
  return (
    <div>
      <h1>List of Contacts</h1>
      <ContactList contacts={this.props.contacts}/> </div>
    )
  }
}

```

```

// Make contacts array available in props
function mapStateToProps(state) {
  return {
    contacts : state.contactStore.contacts
  }
}

```

```

export default connect(mapStateToProps, {fetchContacts})
  <ContactListPage>;

```

We've made the contacts array in store and the fetchContacts function available to ContactListPage component via this.props variable. We can now pass the contacts array down to the ContactList component.

For now, let's update the code such that we can display a list of contacts.

```

// src/components/contact-list

import React from 'react';

```

```
export default function ContactList({contacts}){

  const list = () => {

    return contacts.map(contact => {

      return (

        <li key={contact._id}>{contact.name.first}

        {contact.name.last}</li>

      )

    })

  }

  return (

    <div>
```

```

      <ul>

        { list() }

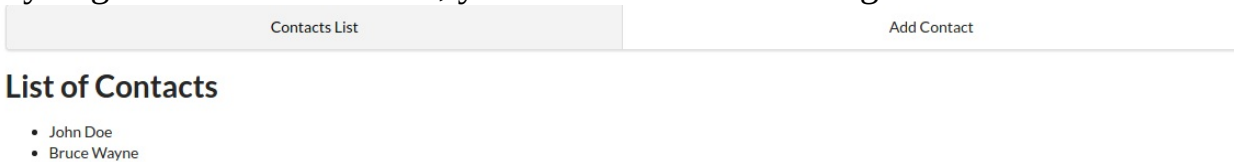
      </ul>

    </div>

  )
}

```

If you go back to the browser, you should have something like this:



Let's make the list UI look more attractive by using semantic-ui's *Card* component. In the components folder, create a new file `contact-card.js` and paste this code: `// src/components/contact-card.js`

```

import React from 'react';
import { Card, Button, Icon } from 'semantic-ui-react'

export default function ContactCard({contact, deleteContact}) {
  return (
    <Card>

```

```

<Card.Content>
<Card.Header>
<Icon name='user outline' /> {contact.name.first}
➔ {contact.name.last}
</Card.Header>
<Card.Description>
<p><Icon name='phone' /> {contact.phone}</p> <p><Icon name='mail
outline' /> {contact.email}</p> </Card.Description>
</Card.Content>
<Card.Content extra>
<div className="ui two buttons">
<Button basic color="green">Edit</Button> <Button basic
color="red">Delete</Button> </div>
</Card.Content>
</Card>
)
}

```

```

ContactCard.propTypes = {
contact: React.PropTypes.object.isRequired
}

```

Update contact-list component to use the new ContactCard component //

src/components/contact-list.js

```

import React from 'react';
import { Card } from 'semantic-ui-react';
import ContactCard from './contact-card';

```

```







export default function ContactList({contacts}){

  const cards = () => {
    return contacts.map(contact => {
      return (
        <ContactCard key={contact._id} contact={contact}/> )
      })
    }

    return (
      <Card.Group>
        { cards() }
      </Card.Group>
    )
  }
}

```

The list page should now look like this:

Contacts List	Add Contact
<h3>List of Contacts</h3> <div> <div> <p> John Doe</p> <p> 555</p> <p> john@gmail.com</p> <div> Edit Delete </div> </div> <div> <p> Bruce Wayne</p> <p> 777</p> <p> bruce.wayne@gmail.com</p> <div> Edit Delete </div> </div> </div>	

Server-side Validation with Redux-Form

Now that we know the Redux store is properly linked up with the React components, we can now make a real fetch request to the database and use the

data populate our contact list page. There are several ways to do this, but the way I'll show is surprisingly simple.

First, we need to configure an Axios client that can connect to the back-end server.

```
// src/actions/index.js import axios from "axios";

export const client = axios.create({

  baseURL: "http://localhost:3030",

  headers: {

    "Content-Type": "application/json"

  }

})
```

Next, we'll update the `contact-actions.js` code to fetch contacts from the database via a GET request using the Axios client.

```
// src/actions/contact-actions.js

import { client } from './';
```

```
const url = '/contacts';

export function fetchContacts(){

  return dispatch => {

    dispatch({

      type: 'FETCH_CONTACTS',

      payload: client.get(url)

    })

  }

}
```

Update `contact-reducer.js` as well since the action and the payload being dispatched is now different.

```
// src/reducers/contact-reducer.js

...
```

```
case "FETCH_CONTACTS_FULFILLED": {  
  
  return {  
  
    ...state,  
  
    contacts: action.payload.data.data || action.payload.data ➔  
  
  }  
  
}
```

After saving, refresh your browser, and ensure the back-end server is running at `localhost:3030`. The contact list page should now be displaying data from the database.

Handle Create and Update Requests using Redux-Form

Next, let's look at how to add new contacts, and to do that we need forms. At first, building a form looks quite easy. But when we start thinking about client-side validation and controlling when errors should be displayed, it becomes tricky. In addition, the back-end server does its own validation, which we also need to display its errors on the form.

Rather than implement all the form functionality ourselves, we'll enlist the help

of a library called [Redux-Form](#). We'll also use a nifty package called [Classnames](#) that will help us highlight fields with validation errors.

We need to stop the server with `ctrl+c` before installing the following packages:
`yarn add redux-form classnames`

We can now start the server after the packages have finished installing.

Let's first quickly add this css class to the `index.css` file to style the form errors:

```
/* src/index.css */
```

```
.error {  
  color: #9f3a38;  
}
```

Then let's add `redux-form`'s reducer to the `combineReducers` function in `reducers/index.js`

```
// src/reducers/index.js  
  
...  
  
import { reducer as formReducer } from 'redux-form';  
  
  
const reducers = {  
  
  contactStore: ContactReducer,  
  
  form: formReducer
```

```
}
```

```
...
```

Next, open `contact-form.js` and build the form UI with this code: `// src/components/contact-form`

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

import { Form, Grid, Button } from 'semantic-ui-react'; import {
Field, reduxForm } from 'redux-form';

import classNames from 'classnames';

class ContactForm extends Component {

  renderField = ({ input, label, type, meta: { touched, error }
    ↪ }) => (

    <Form.Field className={classNames({error:touched && error})}>
    <label>{label}</label>

    <input {...input} placeholder={label} type={type}/> {touched &&
    error && <span className="error"> ↪{error.message}</span>}

    </Form.Field>

  )

  render() {
    const { handleSubmit, pristine, submitting, loading } = this.
    ↪props;

    return (
      <Grid centered columns={2}>
```

```

<Grid.Column>

<h1 style={{marginTop:"1em"}}>Add New Contact</h1> <Form onSubmit=
{handleSubmit} loading={loading}> <Form.Group widths='equal'>

<Field name="name.first" type="text" component=
→{this.renderField} label="First Name"/> <Field name="name.last"
type="text" component=
→{this.renderField} label="Last Name"/> </Form.Group>

<Field name="phone" type="text" component=
→{this.renderField} label="Phone"/>

<Field name="email" type="text" component=
→{this.renderField} label="Email"/>

<Button primary type='submit' disabled={pristine ||
→submitting}>Save</Button>

</Form>

</Grid.Column>

</Grid>

)

}

}

```

```
export default reduxForm({form: 'contact'})(ContactForm);
```

Take the time to examine the code; there's a lot going on in there. See the [reference guide](#) to understand how redux-form works. Also, take a look at semantic-ui-react [documentation](#) and read about its elements to understand how they are used in this context.

Next, we'll define the actions necessary for adding a new contact to the database. The first action will provide a new contact object to the Redux form. While the second action will post the contact data to the API server.

Append the following code to `contact-actions.js`

```
// src/actions/contact-actions.js
```

```
...
```

```
export function newContact() {
```

```
  return dispatch => {
```

```
    dispatch({
```

```
      type: 'NEW_CONTACT'
```

```
    })
```

```
  }
```

```
}
```

```
export function saveContact(contact) {
```

```
  return dispatch => {
```

```
    return dispatch({

      type: 'SAVE_CONTACT',

      payload: client.post(url, contact)

    })

  }

}
```

In the contact-reducer, we need to handle actions for 'NEW_CONTACT', 'SAVE_CONTACT_PENDING', 'SAVE_CONTACT_FULFILLED', and 'SAVE_CONTACT_REJECTED'. We need to declare the following variables:

- **contact** - initialize empty object
- **loading** - update ui with progress info
- **errors** - store server validation errors in case something goes wrong

Add this code inside contact-reducer's switch statement: // src/reducers/contact-reducer.js

...

```
const defaultState = {
  contacts: [],
  contact: {name:{}},
```



```
loading: false,
errors: {}
}
...
case 'NEW_CONTACT': {
return {
...state,
contact: {name:{}}
}
}

case 'SAVE_CONTACT_PENDING': {
return {
...state,
loading: true
}
}

case 'SAVE_CONTACT_FULFILLED': {
return {
...state,
contacts: [...state.contacts, action.payload.data], errors: {},
loading: false
}
}
```

```

case 'SAVE_CONTACT_REJECTED': {
  const data = action.payload.response.data;

  // convert feathers error formatting to match client-side → error
  formatting

  const { "name.first":first, "name.last":last, phone, email →} =
  data.errors;

  const errors = { global: data.message, name: { first,last }, →
  phone, email };

  return {
    ...state,
    errors: errors,
    loading: false
  }
}
...

```

Open contact-form-page.js and update the code as follows: //

```

src/pages/contact-form-page

import React, { Component} from 'react';
import { Redirect } from 'react-router';
import { SubmissionError } from 'redux-form';
import { connect } from 'react-redux';
import { newContact, saveContact } from '../actions/
→contact-actions';
import ContactForm from '../components/contact-form';

```

```

class ContactFormPage extends Component {

```

```
state = {  
  redirect: false  
}
```

```
componentDidMount() {  
  this.props.newContact();  
}
```

```
submit = (contact) => {  
  return this.props.saveContact(contact)  
    .then(response => this.setState({ redirect:true })).catch(err => {  
      throw new SubmissionError(this.props.errors)  
    })  
}
```

```
render() {  
  return (  
    <div>  
      {  
        this.state.redirect ?  
        <Redirect to="/" /> :  
        <ContactForm contact={this.props.contact} loading=  
        {this.props.loading} onSubmit={this.submit} /> }  
      </div>  
    )  
  }  
}
```

```
}
```

```
function mapStateToProps(state) {  
  return {  
    contact: state.contactStore.contact,  
    errors: state.contactStore.errors  
  }  
}
```

```
export default connect(mapStateToProps, {newContact,  
  ↪ saveContact})(ContactFormPage);
```

Let's now go back to the browser and try to intentionally save an incomplete

Add New Contact

First Name	Last Name
<input type="text" value="First Name"/>	<input type="text" value="Parker"/>
First Name is required	
Phone	
<input type="text" value="Phone"/>	
Phone is required	
Email	
<input type="text" value="Email"/>	
Email is required	
<input type="button" value="Save"/>	







form

As you can see, server-side validation prevents us from saving an incomplete contact. We're using the **SubmissionError** class to pass `this.props.errors` to

the form, just in case you're wondering.

Now, finish filling in the form completely. After clicking save, we should be directed to the list page.

List of Contacts

<p> Tony Stark</p> <p> +18138683770</p> <p> tony@starkenterprises.com</p> <div>EditDelete</div>	<p> Peter Parker</p> <p> +1555555555</p> <p> peter@spiderman.com</p> <div>EditDelete</div>
---	--

Client-side Validation with Redux Form

Let's take a look at how client-side validation can be implemented. Open `contact-form` and paste this code outside the `ContactForm` class. Also, update the default export as shown: `// src/components/contact-form.js`

...

```
const validate = (values) => {
  const errors = {name:{}};

  if(!values.name || !values.name.first) {
    errors.name.first = {
      message: 'You need to provide First Name'
    }
  }

  if(!values.phone) {
    errors.phone = {
```

```

message: 'You need to provide a Phone number'
}
} else if(!/^+(?:[0-9] ?){6,14}[0-9]$/.test(values.phone)) {
errors.phone = {
message: 'Phone number must be
➔ in International format'
}
}
if(!values.email) {
errors.email = {
message: 'You need to provide an Email address'
}
} else if (!/^([A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4})$/i.
➔test(values.email)) {
errors.email = {
message: 'Invalid email address'
}
}
return errors;
}
...

```

```

export default reduxForm({form: 'contact', validate})
➔(ContactForm);

```

After saving the file, go back to the browser and try adding invalid data. This time, the client side validation blocks submitting of data to the server.

Add New Contact

First Name

First Name

You need to provide First Name

Last Name

Storm

Phone

555

Phone number must be in International format

Email










susan|

Invalid email address

Save

Now, go ahead and input valid data. We should have at least three new contacts by now.

List of Contacts

<p> Tony Stark</p> <p> +18138683770</p> <p> tony@starkenterprises.com</p> <p>Edit Delete</p>	<p> Peter Parker</p> <p> +1555555555</p> <p> peter@spiderman.com</p> <p>Edit Delete</p>	<p> Susan Storm</p> <p> +1231231</p> <p> susan@fantasticfour.com</p> <p>Edit Delete</p>
--	---	---

Implement Contact Updates

Now that we can add new contacts, let's see how we can update existing contacts. We'll start with the `contact-actions.js` file, where we need to define two actions – one for fetching a single contact, and another for updating the

contact.

```
// src/actions/contact-actions.js

...

export function fetchContact(_id) {

  return dispatch => {

    return dispatch({

      type: 'FETCH_CONTACT',

      payload: client.get(`${url}/${_id}`)

    })

  }

}

export function updateContact(contact) {

  return dispatch => {
```



```

    return dispatch({

      type: 'UPDATE_CONTACT',

      payload: client.put(`${url}/${contact._id}`

        ↩, contact)

    })

  }

}

```

Let's add the following cases to contact-reducer to update state when a contact is being fetched from the database and when it's being updated.

```

// src/reducers/contact-reducer.js

...

case 'FETCH_CONTACT_PENDING': {

  return {

    ...state,

```

```
    loading: true,

    contact: {name:{}}

  }

}

case 'FETCH_CONTACT_FULFILLED': {

  return {

    ...state,

    contact: action.payload.data,

    errors: {},

    loading: false

  }

}
```

```
case 'UPDATE_CONTACT_PENDING': {
```

```
  return {
```

```
    ...state,
```

```
    loading: true
```

```
  }
```

```
}
```

```
case 'UPDATE_CONTACT_FULFILLED': {
```

```
  const contact = action.payload.data;
```

```
  return {
```

```
    ...state,
```

```
    contacts: state.contacts.map(item => item._id ===
```

```

        ➔ contact._id ? contact : item),

        errors: {},

        loading: false

    }

}

case 'UPDATE_CONTACT_REJECTED': {

    const data = action.payload.response.data;

    const { "name.first":first, "name.last":last, phone, email }

    ➔ = data.errors;

    const errors = { global: data.message, name: { first,last }, ➔ pho

    return {

        ...state,

```

```
    errors: errors,

    loading: false

  }

}

...
```

Next, let's pass the new fetch and save actions to the `contact-form-page.js`. We'll also change the `componentDidMount()` and `submit()` logic to handle both create and update scenarios. Be sure to update each section of code as indicated below.

```
// src/pages/contact-form-page.js

...

import { newContact, saveContact, fetchContact, updateContact }

  from '../actions/contact-actions';

...
```

```
componentDidMount = () => {  
  
  const { _id } = this.props.match.params;  
  
  if(_id){  
  
    this.props.fetchContact(_id)  
  
  } else {  
  
    this.props.newContact();  
  
  }  
}
```

```
submit = (contact) => {  
  
  if(!contact._id) {  
  
    return this.props.saveContact(contact)  
  
  }  
}
```

```
        .then(response => this.setState({ redirect:true })) .catch(err => {

            throw new SubmissionError(this.props.errors)

        })

    } else {

        return this.props.updateContact(contact)

        .then(response => this.setState({ redirect:true })) .catch(err => {

            throw new SubmissionError(this.props.errors)

        })

    }

}

}
```

...

```
export default connect(

  mapStateToProps, {newContact, saveContact, fetchContact, ↪ updateC
```

We'll enable contact-form to asynchronously receive data from the `fetchContact()` action. To populate a Redux Form, we use its `initialize` function that's been made available to us via the props. We'll also update the page title with a script to reflect whether we are editing or adding new a contact.

```
// src/components/contact-form.js

...

componentWillReceiveProps = (nextProps) => { // Receive Contact ↪ da

  const { contact } = nextProps;

  if(contact._id !== this.props.contact._id) { // Initialize ↪ form

    this.props.initialize(contact)

  }

}

...

```



```

<h1 style={{marginTop:"1em"}}>{this.props.contact._id ?
  ➔ 'Edit Contact' : 'Add New Contact'}</h1>

...

```

Now, let's convert the **Edit** button in `contact-card.js` to a link that will direct the user to the form.

```

// src/components/contact-card.js

...

import { Link } from 'react-router-dom';

...

<div className="ui two buttons">

  <Link to={`/${contacts/edit/${contact._id}`} className="ui ➔ basic

  <Button basic color="red">Delete</Button> </div>

```

...

Once the list page has finished refreshing, choose any contact and hit the Edit button.

Edit Contact

First Name

Peter

Last Name

Parker

Phone

+1555555555










Email

peter@spiderman.com

Save

Finish making your changes and hit save.

List of Contacts

<p> Tony Stark</p> <p> +18138683770</p> <p> tony@starkenterprises.com</p> <p>Edit Delete</p>	<p> Spider Man</p> <p> +1555555555</p> <p> peter@spiderman.com</p> <p>Edit Delete</p>	<p> Susan Storm</p> <p> +1231231</p> <p> susan@fantasticfour.com</p> <p>Edit Delete</p>
--	---	---

By now, your application should be able to allow users to add new contacts and update existing ones.

Implement Delete Request

Let's now look at the final CRUD operation: delete. This one is much simpler to code. We start at the `contact-actions.js` file.

```
// src/actions/contact-actions.js

...

export function deleteContact(_id) {

  return dispatch => {

    return dispatch({

      type: 'DELETE_CONTACT',

      payload: client.delete(`${url}/${_id}`)

    })

  }

}
```

By now, you should have gotten the drill. Define a case for the `deleteContact()` action in `contact-reducer.js`.

```
// src/reducers/contact-reducer.js
```

```

...

case 'DELETE_CONTACT_FULFILLED': {

    const _id = action.payload.data._id;

    return {

        ...state,

        contacts: state.contacts.filter(item => item._id !== _id) }

    }

...

```

Next, we import the `deleteContact()` action to `contact-list-page.js` and pass it to the `ContactList` component.

```

// src/pages/contact-list-page.js

...

import { fetchContacts, deleteContact } from

➔ '../actions/contact-actions';

```

...

```
<ContactList contacts={this.props.contacts}
```

```
  ↪ deleteContact={this.props.deleteContact}/>
```

...

```
export default connect(mapStateToProps, {fetchContacts, ↪ deleteCont
```

The `ContactList` component, in turn, passes the `deleteContact()` action to the `ContactCard` component // `src/components/contact-list.js`

...

```
export default function ContactList({contacts, deleteContact}){
```

```
  ↪ // replace this line
```

```
  const cards = () => {
```

```
    return contacts.map(contact => {
```

```
      return (
```

```

<ContactCard
  key={contact._id}
  contact={contact}
  deleteContact={deleteContact} /> // and this one )
}))
}
...

```

Finally, we update **Delete** button in ContactCard to execute the `deleteContact()` action, via the `onClick` attribute.

```

// src/components/contact-card.js

...

<Button basic color="red" onClick={() =>
  deleteContact(contact._id)} >Delete</Button> ...

```

Wait for the browser to refresh, then try to delete one or more contacts. The delete button should work as expected.

Conclusion

By now, you should have learned the basics of creating a CRUD web app in JavaScript. It may seem we've written quite a lot of code to manage only one

model. We could have done less work if we had used an MVC framework. The problem with these frameworks is that they become harder to maintain as the code grows.

A Flux-based framework, such as Redux, allows us to build large complex projects that are easy to manage. If you don't like the verbose code that Redux requires you to write, then you could also look at [Mobx](#) as an alternative.

At least I hope you now have a good impression of FeathersJS. With little effort, we were able to generate a database API with only a few commands and a bit of coding. Although we have only scratched the surface in exploring its capabilities, you will at least agree with me that it is a robust solution for creating APIs.

Chapter 3: How to Build a Todo App Using React, Redux, and Immutable.js

by Dan Prince

The way [React](#) uses components and a one-way data flow makes it ideal for describing the structure of user interfaces. However, its tools for working with state are kept deliberately simple – to help remind us that React is just the View in the traditional [Model-View-Controller](#) architecture.

There's nothing to stop us from building large applications with just React, but we would quickly discover that to keep our code simple, we'd need to manage our state elsewhere.

Whilst there's no *official* solution for dealing with application state, there are some libraries that align particularly well with React's paradigm. In this post, we'll pair React with two such libraries and use them to build a simple application.

Redux

[Redux](#) is a tiny library that acts as a container for our application state, by combining ideas from [Flux](#) and [Elm](#). We can use Redux to manage any kind of application state, providing we stick to the following guidelines:

1. our state is kept in a single store
2. changes come from *actions* and not *mutations*

At the core of a Redux store is a function that takes the current application state and an action and combines them to create a new application state. We call this function a **reducer**.

Our React components will be responsible for sending actions to our store, and

in turn our store will tell the components when they need to re-render.

ImmutableJS

Because Redux doesn't allow us to mutate the application state, it can be helpful to enforce this by modeling application state with immutable data structures.

[ImmutableJS](#) offers us a number of immutable data structures with mutative interfaces, and they're implemented [in an efficient way](#), inspired by the implementations in Clojure and Scala.

Demo

We're going to use React with Redux and ImmutableJS to build a simple todo list that allows us to add todos and toggle them between complete and incomplete. You can [find a CodePen demo here](#), and the che code is available in a [repository on GitHub](#).

Setup

We'll get started by creating a project folder and initializing a `package.json` file with `npm init`. Then we'll install the dependencies we're going to need.

```
npm install --save react react-dom redux react-redux immutable
npm install --save-dev webpack babel-core babel-loader babel-preset-
→ babel-preset-react
```

We'll be using [JSX](#) and [ES2015](#), so we'll compile our code with [Babel](#), and we're going to do this as part of the module bundling process with [Webpack](#).

First, we'll create our Webpack configuration in `webpack.config.js`:

```
module.exports = {
  entry: './src/app.js',
  output: {
    path: __dirname,
    filename: 'bundle.js'
  },
  module: {
    loaders: [
```

```

    {
      test: /\.js$/,
      exclude: /node_modules/,
      loader: 'babel-loader',
      query: { presets: [ 'es2015', 'react' ] }
    }
  ]
}
};

```

Finally, we'll extend our `package.json` by adding an npm script to compile our code with source maps:

```

"script": {
  "build": "webpack --debug"
}

```

We'll need to run `npm run build` each time we want to compile our code.

React and Components

Before we implement any components, it can be helpful to create some dummy data. This helps us get a feel for what we're going to need our components to render:

```

const dummyTodos = [
  { id: 0, isDone: true, text: 'make components' },
  { id: 1, isDone: false, text: 'design actions' },
  { id: 2, isDone: false, text: 'implement reducer' },
  { id: 3, isDone: false, text: 'connect components' }
];

```

For this application, we're only going to need two React components, `<Todo />` and `<TodoList />`.

```

// src/components.js

import React from 'react';

export function Todo(props) {
  const { todo } = props;
  if(todo.isDone) {
    return <strike>{todo.text}</strike>;
  } else {
    return <span>{todo.text}</span>;
  }
}

```

```

    }
  }
}

export function TodoList(props) {
  const { todos } = props;
  return (
    <div className='todo'>
      <input type='text' placeholder='Add todo' />
      <ul className='todo__list'>
        {todos.map(t => (
          <li key={t.id} className='todo__item'>
            <Todo todo={t} />
          </li>
        ))}
      </ul>
    </div>
  );
}

```

At this point, we can test these components by creating an `index.html` file in the project folder and populating it with the following markup. (You can find a simple stylesheet [on GitHub](#)):

```

<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="style.css">
    <title>Immutable Todo</title>
  </head>
  <body>
    <div id="app"></div>
    <script src="bundle.js"></script>
  </body>
</html>

```

We'll also need an application entry point at `src/app.js`.

```

// src/app.js

import React from 'react';
import { render } from 'react-dom';
import { TodoList } from './components';

const dummyTodos = [
  { id: 0, isDone: true, text: 'make components' },
  { id: 1, isDone: false, text: 'design actions' },
  { id: 2, isDone: false, text: 'implement reducer' },
  { id: 3, isDone: false, text: 'connect components' }
]

```

```
];  
  
render(  
  <TodoList todos={dummyTodos} />,  
  document.getElementById('app')  
);
```

Compile the code with `npm run build`, then navigate your browser to the `index.html` file and make sure that it's working.

Redux and Immutable

Now that we're happy with the user interface, we can start to think about the state behind it. Our dummy data is a great place to start from, and we can easily translate it into ImmutableJS collections:

```
import { List, Map } from 'immutable';  
  
const dummyTodos = List([  
  Map({ id: 0, isDone: true, text: 'make components' }),  
  Map({ id: 1, isDone: false, text: 'design actions' }),  
  Map({ id: 2, isDone: false, text: 'implement reducer' }),  
  Map({ id: 3, isDone: false, text: 'connect components' })  
]);
```

ImmutableJS maps don't work in the same way as JavaScript's objects, so we'll need to make some slight tweaks to our components. Anywhere there was a property access before (e.g. `todo.id`) needs to become a method call instead (`todo.get('id')`).

Designing Actions

Now that we've got the shape and structure figured out, we can start thinking about the actions that will update it. In this case, we'll only need two actions, one to add a new todo and the other to toggle an existing one.

Let's define some functions to create these actions:

```
// src/actions.js  
  
// succinct hack for generating passable unique ids  
const uid = () => Math.random().toString(34).slice(2);
```

```

export function addTodo(text) {
  return {
    type: 'ADD_TODO',
    payload: {
      id: uid(),
      isDone: false,
      text: text
    }
  };
}

export function toggleTodo(id) {
  return {
    type: 'TOGGLE_TODO',
    payload: id
  }
}

```

Each action is just a JavaScript object with a type and payload properties. The type property helps us decide what to do with the payload when we process the action later.

Designing a Reducer

Now that we know the shape of our state and the actions that update it, we can build our reducer. Just as a reminder, the reducer is a function that takes a state and an action, then uses them to compute a new state.

Here's the initial structure for our reducer:

```

// src/reducer.js

import { List, Map } from 'immutable';

const init = List([]);

export default function(todos=init, action) {
  switch(action.type) {
    case 'ADD_TODO':
      // ...
    case 'TOGGLE_TODO':
      // ...
    default:
      return todos;
  }
}

```

Handling the `ADD_TODO` action is quite simple, as we can use the [.push\(\)](#) method, which will return a new list with the todo appended at the end:

```
case 'ADD_TODO':  
  return todos.push(Map(action.payload));
```

Notice that we're also converting the todo object into an immutable map before it's pushed onto the list.

The more complex action we need to handle is `TOGGLE_TODO`:

```
case 'TOGGLE_TODO':  
  return todos.map(t => {  
    if(t.get('id') === action.payload) {  
      return t.update('isDone', isDone => !isDone);  
    } else {  
      return t;  
    }  
  });
```

We're using [.map\(\)](#) to iterate over the list and find the todo whose `id` matches the action. Then we call [.update\(\)](#), which takes a key and a function, then it returns a new copy of the map, with the value at the key replaced with the result of passing the initial value to the update function.

It might help to see the literal version:

```
const todo = Map({ id: 0, text: 'foo', isDone: false });  
todo.update('isDone', isDone => !isDone);  
// => { id: 0, text: 'foo', isDone: true }
```

Connecting Everything

Now we've got our actions and reducer ready, we can create a store and connect it to our React components:

```
// src/app.js  
  
import React from 'react';  
import { render } from 'react-dom';  
import { createStore } from 'redux';  
import { TodoList } from './components';  
import reducer from './reducer';
```

```
const store = createStore(reducer);

render(
  <TodoList todos={store.getState()} />,
  document.getElementById('app')
);
```

We'll need to make our components aware of this store. We'll use the [react-redux](#) to help simplify this process. It allows us to create store-aware containers that wrap around our components, so that we don't have to change our original implementations.

We're going to need a container around our `<TodoList />` component. Let's see what this looks like:

```
// src/containers.js

import { connect } from 'react-redux';
import * as components from './components';
import { addTodo, toggleTodo } from './actions';

export const TodoList = connect(
  function mapStateToProps(state) {
    // ...
  },
  function mapDispatchToProps(dispatch) {
    // ...
  }
)(components.TodoList);
```

We create containers with the [connect](#) function. When we call `connect()`, we pass two functions, `mapStateToProps()` and `mapDispatchToProps()`.

The `mapStateToProps` function takes the store's current state as an argument (in our case, a list of todos), then it expects the return value to be an object that describes a mapping from that state to props for our wrapped component:

```
function mapStateToProps(state) {
  return { todos: state };
}
```

It might help to visualize this on an instance of the wrapped React component:

```
<TodoList todos={state} />
```

We'll also need to supply a `mapDispatchToProps` function, which is passed the store's dispatch method, so that we can use it to dispatch the actions from our action creators:

```
function mapDispatchToProps(dispatch) {  
  return {  
    addTodo: text => dispatch(addTodo(text)),  
    toggleTodo: id => dispatch(toggleTodo(id))  
  };  
}
```

Again, it might help to visualize all these props together on an instance of our wrapped React component:

```
<TodoList todos={state}  
  addTodo={text => dispatch(addTodo(text))}  
  toggleTodo={id => dispatch(toggleTodo(id))} />
```

Now that we've mapped our component to the action creators, we can call them from event listeners:

```
export function TodoList(props) {  
  const { todos, toggleTodo, addTodo } = props;  
  
  const onSubmit = (event) => {  
    const input = event.target;  
    const text = input.value;  
    const isEnterKey = (event.which === 13);  
    const isLongEnough = text.length > 0;  
  
    if(isEnterKey && isLongEnough) {  
      input.value = '';  
      addTodo(text);  
    }  
  };  
  
  const toggleClick = id => event => toggleTodo(id);  
  
  return (  
    <div className='todo'>  
      <input type='text'  
        className='todo__entry'  
        placeholder='Add todo'  
        onKeyDown={onSubmit} />  
      <ul className='todo__list'>  
        {todos.map(t => (  
          <li key={t.get('id')}>
```



```

        className='todo__item'
        onClick={toggleClick(t.get('id'))}>
        <Todo todo={t.toJS()} />
      </li>
    )}
  </ul>
</div>
);
}

```

The containers will automatically subscribe to changes in the store, and they'll re-render the wrapped components whenever their mapped props change.

Finally, we need to make the containers aware of the store, using the `<Provider />` component:

```

// src/app.js

import React from 'react';
import { render } from 'react-dom';
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import reducer from './reducer';
import { TodoList } from './containers';
// ^^^^^^^^^^^^^

const store = createStore(reducer);

render(
  <Provider store={store}>
    <TodoList />
  </Provider>,
  document.getElementById('app')
);

```

Conclusion

There's no denying that the ecosystem around React and Redux can be quite complex and intimidating for beginners, but the good news is that almost all of these concepts are transferable. We've barely touched the surface of Redux's architecture, but already we've seen enough to help us start learning about [The Elm Architecture](#), or pick up a ClojureScript library like [Om](#) or [Re-frame](#). Likewise, we've only seen a fraction of the possibilities with immutable data, but now we're better equipped to start learning a language like [Clojure](#) or [Haskell](#).

Whether you're just exploring the state of web application development, or you spend all day writing JavaScript, experience with action-based architectures and immutable data is already becoming a vital skill for developers, and *right now* is a great time to be learning the essentials.

Chapter 4: Building a Game with Three.js, React and WebGL

by Andrew Ray

I'm making a game titled "[Charisma The Chameleon](#)." It's built with Three.js, React and WebGL. This is an introduction to how these technologies work together using [react-three-renderer](#) (abbreviated R3R).

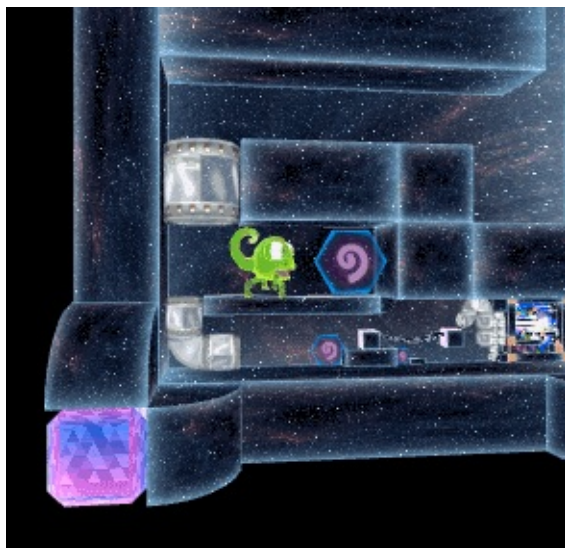
How It All Began

Some time ago, [Pete Hunt](#) made a joke about building a game using React in the #reactjs IRC channel:

I bet we could make a first person shooter with React! Enemy has <Head />
<Body> <Legs> etc.

I laughed. He laughed. Everyone had a great time. "Who on earth would do that?" I wondered.

Years later, that's exactly what I'm doing.



[Charisma The Chameleon](#) is a game where you collect power-ups that make you shrink to solve an infinite fractal maze. I've been a React developer for a few years, and I was curious if there was a way to drive Three.js using React. That's when R3R caught my eye.

Why React?

I know what you're thinking: **why?** Humor me for a moment. Here's some reasons to consider using React to drive your 3D scene:

- "Declarative" views let you cleanly separate your scene rendering from your game logic.
- Design easy to reason about components, like `<Player />`, `<Wall />`, `<Level />`, etc.
- "Hot" (live) reloading of game assets. Change textures and models and see them update live in your scene!
- Inspect and debug your 3D scene as markup with native browser tools, like the Chrome inspector.
- Manage game assets in a dependency graph using Webpack, eg `<Texture src={ require('../assets/image.png') } />`

Let's set up a scene to get an understanding of how this all works.

React and WebGL

I've created a [sample GitHub repository](#) to accompany this article. Clone the repository and follow the instructions in the README to run the code and follow along. It stars SitePointy the 3D Robot!



Watch Out!

R3R is still in beta. Its API is volatile and may change in the future. It only handles a subset of Three.js at the moment. I've found it complete enough to build a full game, but your mileage may vary.

Organizing view code

The main benefit of using React to drive WebGL is our view code is **decoupled** from our game logic. That means our rendered entities are small components that are easy to reason about.

R3R exposes a declarative API that wraps Three.js. For example, we can write:

```
<scene>
  <perspectiveCamera
    position={ new THREE.Vector3( 1, 1, 1 )
  />
</scene>
```

Now we have an empty 3D scene with a camera. Adding a mesh to the scene is as simple as including a `<mesh />` component, and giving it `<geometry />` and a `<material />`.

```
<scene>
  ...
  <mesh>
    <boxGeometry
      width={ 1 }
      height={ 1 }
      depth={ 1 }
    />
    <meshBasicMaterial
      color={ 0x00ff00 }
    />
  </mesh>
```

Under the hood, this creates a [THREE.Scene](#) and automatically adds a mesh with [THREE.BoxGeometry](#). R3R handles diffing the old scene with any changes. If you add a new mesh to the scene, the original mesh won't be recreated. Just as with vanilla React and the DOM, the 3D scene is **only updated with the differences**.

Because we're working in React, we can separate game entities into component files. The [Robot.js file](#) in the example repository demonstrates how to represent the main character with pure React view code. It's a "stateless functional" component, meaning it doesn't hold any local state:

```
const Robot = ({ position, rotation }) => <group
  position={ position }
  rotation={ rotation }
>
  <mesh rotation={ localRotation }>
    <geometryResource
      resourceId="robotGeometry"
    />
    <materialResource
      resourceId="robotTexture"
    />
  </mesh>
</group>;
```

And now we include the `<Robot />` in our 3D scene!

```
<scene>
  ...
  <mesh>...</mesh>
  <Robot
    position={...}
    rotation={...}
  />
</scene>
```

You can see more examples of the API on the [R3R GitHub repository](#), or view the complete example setup in [the accompanying project](#).

Organizing Game Logic

The second half of the equation is handling game logic. Let's give SitePointy, our robot, some simple animation.



How do game loops traditionally work? They accept user input, analyze the old "state of the world," and return the new state of the world for rendering. For convenience, let's store our "game state" object in component state. In a more mature project, you could move the game state into a Redux or Flux store.

We'll use the browser's [requestAnimationFrame](#) API callback to drive our game loop, and run the loop in [GameContainer.js](#). To animate the robot, let's calculate a new position based on the timestamp passed to `requestAnimationFrame`, then store the new position in state.

```
// ...
gameLoop( time ) {
  this.setState({
    robotPosition: new THREE.Vector3(
      Math.sin( time * 0.01 ), 0, 0
    )
  });
}
```

Calling `setState()` triggers a re-render of the child components, and the 3D scene updates. We pass the state down from the container component to the presentational `<Game />` component:

```
render() {
  const { robotPosition } = this.state;
  return <Game
    robotPosition={ robotPosition }
  />;
}
```

There's a useful pattern we can apply to help organize this code. Updating the

robot position is a simple time-based calculation. In the future, it might also take into account the previous robot position from the previous game state. A function that takes in some data, processes it, and returns new data, is often referred to as a **reducer**. We can abstract out the movement code into a reducer function!

Now we can write a clean, simple game loop that only has function calls in it:

```
import robotMovementReducer from './game-reducers/
  ↪robotMovementReducer.js';

// ...

gameLoop() {
  const oldState = this.state;
  const newState = robotMovementReducer(
    ↪ oldState );
  this.setState( newState );
}
```

To add more logic to the game loop, such as processing physics, create another reducer function and pass it the result of the previous reducer:

```
const newState = physicsReducer( robotMovementReducer( oldState ) );
```

As your game engine grows, organizing game logic into separate functions becomes critical. This organization is straightforward with the reducer pattern.

Asset management

This is still an evolving area of R3R. For textures, you specify a `url` attribute on the JSX tag. Using Webpack, you can require the local path to the image:

```
<texture url={ require( '../local/image/path.png' ) } />
```

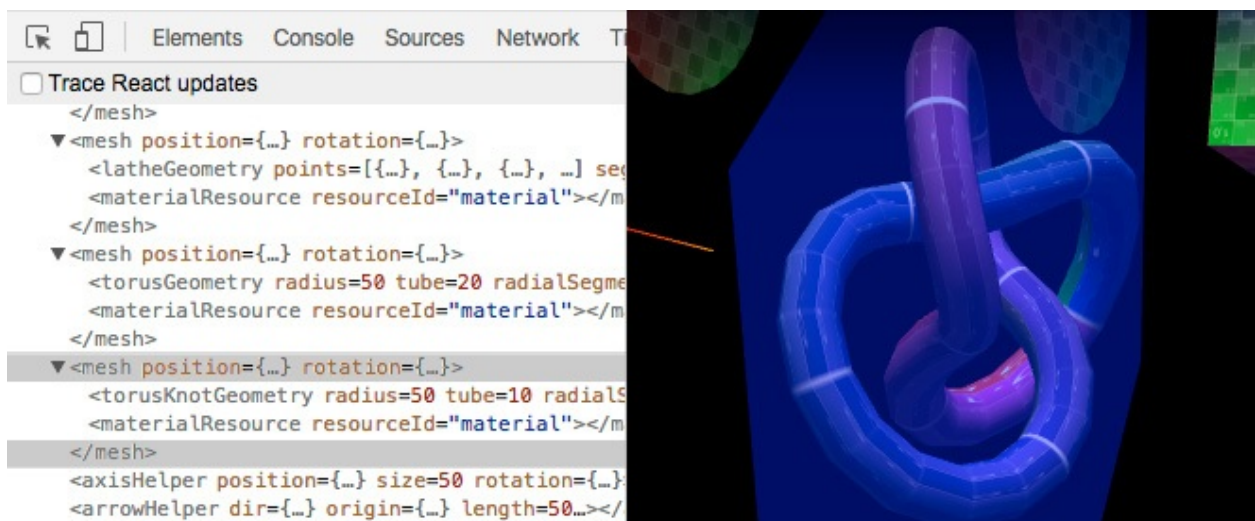
With this setup, if you change the image on disk, your 3D scene will live update! This is invaluable for rapidly iterating game design and content.

For other assets like 3D models, you still have to process them using the built-in loaders from Three.js, like the [JSONLoader](#). I experimented with using a custom Webpack loader for loading 3D model files, but in the end it was too much work for no benefit. It's easier to treat the model as binary data and load them with the

[file-loader](#). This still affords live reloading of model data. You can see this in action in [the example code](#).

Debugging

R3R supports the React developer tools extension for both [Chrome](#) and [Firefox](#). You can inspect your scene as if it were the vanilla DOM! Hovering over elements in the inspector shows their bounding box in the scene. You can also hover over texture definitions to see which objects in the scene use those textures.



You can also join us in the [react-three-renderer Gitter chat room](#) for help debugging your applications.

Performance Considerations

While building Charisma The Chameleon, I've run into several performance issues that are unique to this workflow.

- My **hot reload time** with Webpack was as long as thirty seconds! This is because large assets have to be re-written to the bundle on every reload. The solution was to implement [Webpack's DLLPlugin](#), which cut down reload times to below five seconds.
- Ideally your scene should only call **one `setState()`** per frame render. After profiling my game, React itself is the main bottleneck. Calling `setState()`

more than once per frame can cause double renders and reduce performance.

- Past a certain number of objects, **R3R will perform worse** than vanilla Three.js code. For me this was around 1,000 objects. You can compare R3R to Three.js under "Benchmarks" [in the examples](#).

The Chrome DevTools Timeline feature is an amazing tool for debugging performance. It's easy to visually inspect your game loop, and it's more readable than the "Profile" feature of the DevTools.

That's It!

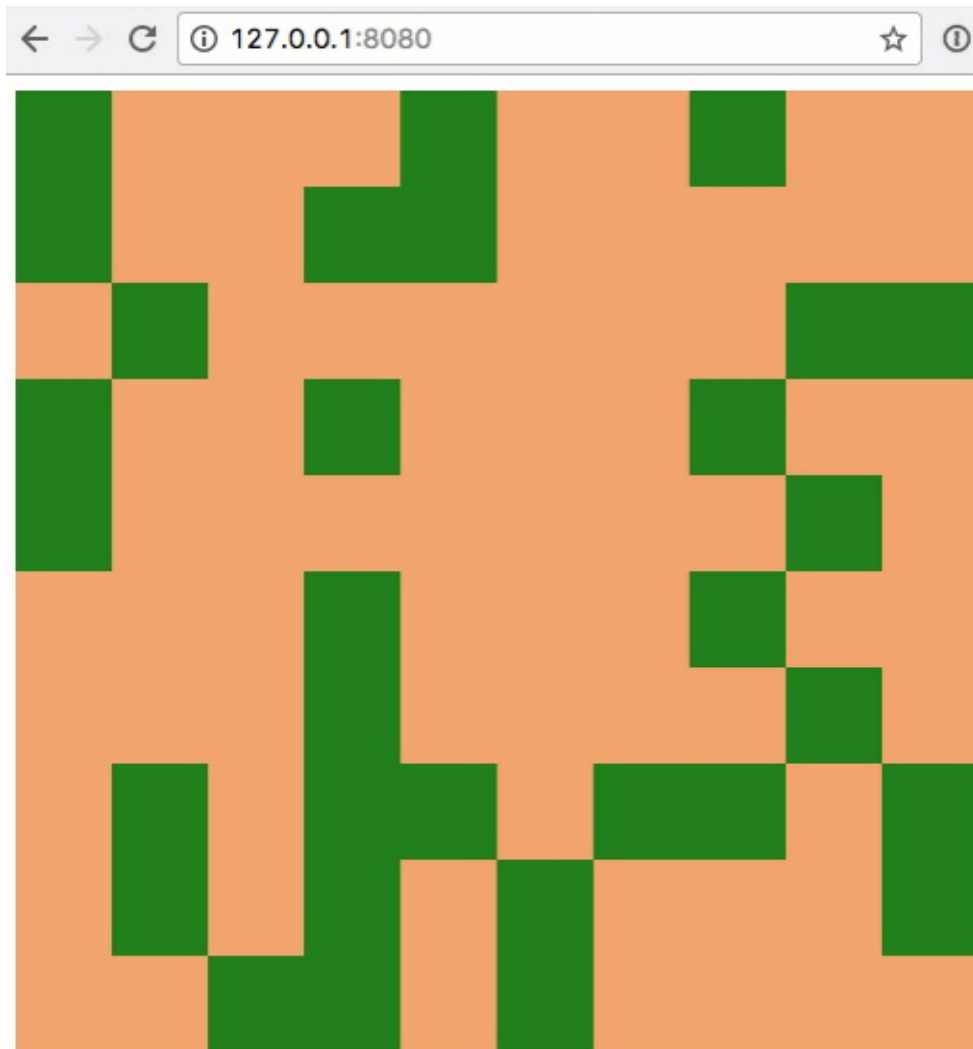
Check out [Charisma The Chameleon](#) to see what's possible using this setup. While this toolchain is still quite young, I've found React with R3R to be integral to organizing my WebGL game code cleanly. You can also check out the small but growing [R3R examples page](#) to see some well organized code samples.

Chapter 5: Procedurally Generated Game Terrain with React, PHP, and WebSockets

by Christopher Pitt

[Last time](#), I began telling you the story of how I wanted to make a game. I described how I set up the async PHP server, the Laravel Mix build chain, the React front end, and WebSockets connecting all this together. Now, let me tell you about what happened when I starting building the game mechanics with this mix of React, PHP, and WebSockets...

he code for this part can be found at github.com/assertchris-tutorials/sitepoint-making-games/tree/part-2. I've tested it with PHP 7.1, in a recent version of Google Chrome.



Making a Farm

"Let's start simple. We have a 10 by 10 grid of tiles, filled with randomly generated stuff."

I decided to represent the farm as a `Farm`, and each tile as a `Patch`. From `app/Model/FarmModel.pre`:

```
namespace App\Model;
```

```
class Farm

{

    private $width

    {

        get { return $this->width; }

    }


    private $height

    {

        get { return $this->height; }

    }


    public function __construct(int $width = 10,
```

```
int $height = 10)

{

    $this->width = $width;

    $this->height = $height;

}

}
```

I thought it would be a fun time to try out the [class accessors macro](#) by declaring private properties with public getters. For this I had to install `pre/class-accessors` (via `composer require`).

I then changed the socket code to allow for new farms to be created on request. From `app/Socket/GameSocket.pre`:

```
namespace App\Socket;

use Aerys\Request;

use Aerys\Response;

use Aerys\WebSocket;
```

```
use Aerys\Websocket\Endpoint;

use Aerys\Websocket\Message;

use App\Model\FarmModel;

class GameSocket implements Websocket
{

    private $farms = [];

    public function onData(int $clientId,

        Message $message)

    {

        $body = yield $message;
```

```
if ($body === "new-farm") {

    $farm = new FarmModel();

    $payload = json_encode([

        "farm" => [

            "width" => $farm->width,

            "height" => $farm->height,

        ],

    ]);

    yield $this->endpoint->send(

        $payload, $clientId

    );
```



```
        $this->farms[$clientId] = $farm;

    }

}

public function onClose(int $clientId,

    int $code, string $reason)

{

    unset($this->connections[$clientId]);

    unset($this->farms[$clientId]);

}

// ...
```

```
}
```

I noticed how similar this GameSocket was to the previous one I had – except, instead of broadcasting an echo, I was checking for new-farm and sending a message back only to the client that had asked.

"Perhaps it's a good time to get less generic with the React code. I'm going to rename component.jsx to farm.jsx."

From assets/js/farm.jsx:

```
import React from "react"

class Farm extends React.Component

{

  componentWillMount()

  {

    this.socket = new WebSocket(

      "ws://127.0.0.1:8080/ws"

    )
```

```
this.socket.addEventListener(  
  
  "message", this.onMessage  
  
)  
  
// DEBUG  
  
this.socket.addEventListener("open", () => {  
  
  this.socket.send("new-farm")  
  
  })  
  
}  
  
}
```

```
export default Farm
```

In fact, the only other thing I changed was sending new-farm instead of hello world. Everything else was the same. I did have to change the app.jsx code though. From assets/js/app.jsx:

```
import React from "react"

import ReactDOM from "react-dom"

import Farm from "../farm"

ReactDOM.render(

  <Farm />,

  document.querySelector(".app")

)ument.querySelector(".app")

)
```

It was far from where I needed to be, but using these changes I could see the class accessors in action, as well as prototype a kind of request/response pattern for future WebSocket interactions. I opened the console, and saw {"farm":

```
{"width":10,"height":10}}.
```

"Great!"

Then I created a Patch class to represent each tile. I figured this was where a lot of the game's logic would happen. From `app/Model/PatchModel.pre`:

```
namespace App\Model;

private $x

{

    get { return $this->x; }

}

private $y

{

    get { return $this->y; }

}
```

```
public function __construct(int $x, int $y)

{

    $this->x = $x;

    $this->y = $y;

}

}
```

I'd need to create as many patches as there are spaces in a new Farm. I could do this as part of FarmModel construction. From app/Model/FarmModel.pre:

```
namespace App\Model;

class FarmModel

{

    private $width
```

```
{  
  
    get { return $this->width; }  
  
}
```

```
private $height
```

```
{  
  
    get { return $this->height; }  
  
}
```

```
private $patches
```

```
{  
  
    get { return $this->patches; }  
  
}
```

```
public function __construct($width = 10, $height = 10)

{

    $this->width = $width;

    $this->height = $height;


    $this->createPatches();

}


private function createPatches()

{

    for ($i = 0; $i < $this->width; $i++) {

        $this->patches[$i] = [];
```



```

        for ($j = 0; $j < $this->height; $j++) {

            $this->patches[$i][$j] =

                new PatchModel($i, $j);

        }

    }

}

```

For each cell, I created a new `PatchModel` object. These were pretty simple to begin with, but they needed an element of randomness – a way to grow trees, weeds, flowers ... at least to begin with. From `app/Model/PatchModel.pre`:

```

public function start(int $width, int $height,

array $patches)

{

    if (!$this->started && random_int(0, 10) > 7) {

```

```
$this->started = true;

return true;

}

return false;

}
```

I thought I'd begin just by randomly growing a patch. This didn't change the external state of the patch, but it did give me a way to test how they were started by the farm. From `app/Model/FarmModel.pre`:

```
namespace App\Model;

use Amp;

use Amp\Coroutine;

use Closure;
```

```
class FarmModel

{

    private $onGrowth

    {

        get { return $this->onGrowth; }

    }


    private $patches

    {

        get { return $this->patches; }

    }


    public function __construct(int $width = 10,
```

```
int $height = 10, Closure $onGrowth)
```

```
{
```

```
    $this->width = $width;
```

```
    $this->height = $height;
```

```
    $this->onGrowth = $onGrowth;
```

```
}
```

```
public async function createPatches()
```

```
{
```

```
    $patches = [];
```

```
    for ($i = 0; $i < $this->width; $i++) {
```

```
        $this->patches[$i] = [];
```

```
for ($j = 0; $j < $this->height; $j++) {  
  
    $this->patches[$i][$j] = $patches[] =  
  
    new PatchModel($i, $j);  
  
}  
  
}
```

```
foreach ($patches as $patch) {  
  
    $growth = $patch->start(  
  
        $this->width,  
  
        $this->height,  
  
        $this->patches  
  
    );
```

```
if ($growth) {

    $closure = $this->onGrowth;

    $result = $closure($patch);

    if ($result instanceof Coroutine) {

        yield $result;

    }

}

}

}

// ...
```

```
}
```

There was a lot going on here. For starters, I introduced an `async` function keyword using a macro. You see, Amp handles the `yield` keyword by resolving Promises. More to the point: when Amp sees the `yield` keyword, it assumes what is being yielded is a Coroutine (in most cases).

I could have made the `createPatches` function a normal function, and just returned a Coroutine from it, but that was such a common piece of code that I might as well have created a special macro for it. At the same time, I could replace code I had made in the previous part. From `helpers.pre`:

```
async function mix($path) {

    $manifest = yield Amp\File\get(

        .."/public/mix-manifest.json"

    );

    $manifest = json_decode($manifest, true);

    if (isset($manifest[$path])) {

        return $manifest[$path];
    }
}
```

```
}

    throw new Exception("{ $path } not found");

}
```

Previously, I had to make a generator, and then wrap it in a new Coroutine:

```
use Amp\Coroutine;

function mix($path) {

    $generator = () => {

        $manifest = yield Amp\File\get(

            .."/public/mix-manifest.json"

        );

        $manifest = json_decode($manifest, true);
```



```
        if (isset($manifest[$path])) {  
  
            return $manifest[$path];  
  
        }  
  
        throw new Exception("{ $path } not found");  
  
    };  
  
    return new Coroutine($generator());  
}
```

I began the `createPatches` method as before, creating new `PatchModel` objects for each `x` and `y` in the grid. Then I started another loop, to call the `start` method on each patch. I would have done these in the same step, but I wanted my `start` method to be able to inspect the surrounding patches. That meant I would have to create all of them first, before working out which patches were around each other.

I also changed `FarmModel` to accept an `onGrowth` closure. The idea was that I

could call that closure if a patch grew (even during the bootstrapping phase).

Each time a patch grew, I reset the `$changes` variable. This ensured the patches would keep growing until an entire pass of the farm yielded no changes. I also invoked the `onGrowth` closure. I wanted to allow `onGrowth` to be a normal closure, or even to return a Coroutine. That's why I needed to make `createPatches` an async function.

A Note on the `onGrowth` Coroutines

Admittedly, allowing `onGrowth` coroutines complicated things a bit, but I saw it as essential for allowing other async actions when a patch grew. Perhaps later I'd want to send a socket message, and I could only do that if `yield` worked inside `onGrowth`. I could only `yield` `onGrowth` if `createPatches` was an async function. And because `createPatches` was an async function, I would need to `yield` it inside `GameSocket`.

"It's easy to get turned off by all the things that need learning when making one's first async PHP application. Don't give up too soon!"

The last bit of code I needed to write to check that this was all working was in `GameSocket`. From `app/Socket/GameSocket.pre`:

```
if ($body === "new-farm") {

    $patches = [];

    $farm = new FarmModel(10, 10,

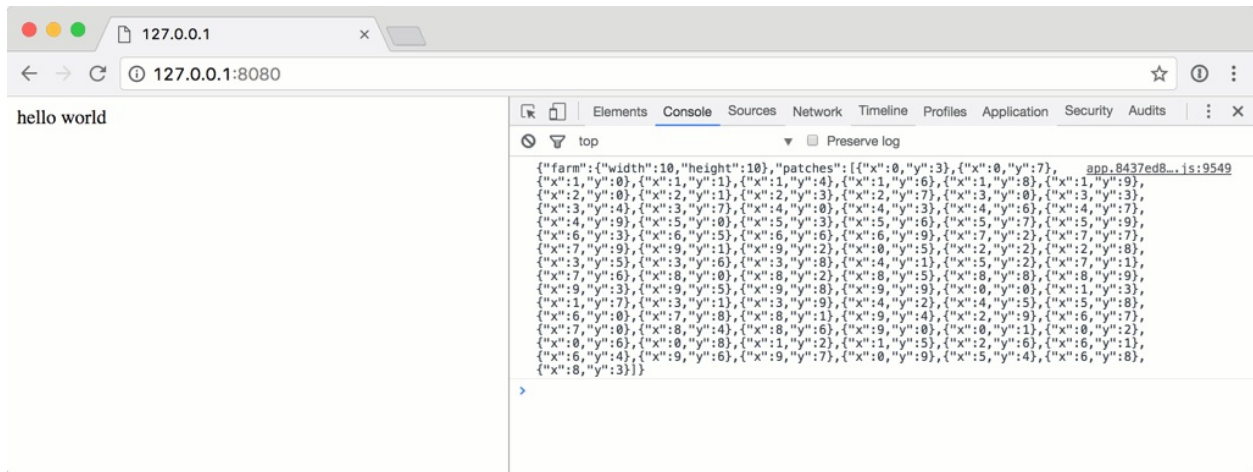
    function (PatchModel $patch) use (&$patches) {

        array_push($patches, [
```

```
        "x" => $patch->x,  
  
        "y" => $patch->y,  
  
    ]);  
  
}  
  
);  
  
yield $farm->createPatches();  
  
$payload = json_encode([  
  
    "farm" => [  
  
        "width" => $farm->width,  
  
        "height" => $farm->height,  
  
    ],
```

```
        "patches" => $patches,  
  
    ]);  
  
    yield $this->endpoint->send(  
  
        $payload, $clientId  
  
    );  
  
    $this->farms[$clientId] = $farm;  
  
}
```

This was only slightly more complex than the previous code I had. I needed to provide a third parameter to the `FarmModel` constructor, and yield `$farm->createPatches()` so that each could have a chance to randomize. After that, I just needed to pass a snapshot of the patches to the socket payload.



"What if I start each patch as dry dirt? Then I could make some patches have weeds, and others have trees ..."

I set about customizing the patches. From `app/Model/PatchModel.pre`:

```
private $started = false;

private $wet {

    get { return $this->wet ? : false; }

};

private $type {

    get { return $this->type ? : "dirt"; }
```

```
};
```

```
public function start(int $width, int $height,
```

```
array $patches)
```

```
{
```

```
    if ($this->started) {
```

```
        return false;
```

```
    }
```

```
    if (random_int(0, 100) < 90) {
```

```
        return false;
```

```
    }
```

```
$this->started = true;

$this->type = "weed";

return true;
}
```

I changed the order of logic around a bit, exiting early if the patch had already been started. I also reduced the chance of growth. If neither of these early exits happened, the patch type would be changed to weed.

I could then use this type as part of the socket message payload. From `app/Socket/GameSocket.pre`:

```
$farm = new FarmModel(10, 10,

function (PatchModel $patch) use (&$patches) {

    array_push($patches, [

        "x" => $patch->x,

        "y" => $patch->y,

        "wet" => $patch->wet,
```

```
"type" => $patch->type,  
  
]);  
  
}  
  
);
```

Rendering the Farm

It was time to show the farm, using the React workflow I had setup previously. I was already getting the width and height of the farm, so I could make every block dry dirt (unless it was supposed to grow a weed). From `assets/js/app.jsx`:

```
import React from "react"

class Farm extends React.Component

{

  constructor()

  {

    super()
```



```
this.onMessage = this.onMessage.bind(this)
```

```
this.state = {
```

```
  "farm": {
```

```
    "width": 0,
```

```
    "height": 0,
```

```
  },
```

```
  "patches": [],
```

```
};
```

```
}
```

```
componentWillMount()
```

```
{

  this.socket = new WebSocket(

    "ws://127.0.0.1:8080/ws"

  )


  this.socket.addEventListener(

    "message", this.onMessage

  )


  // DEBUG


  this.socket.addEventListener("open", () => {

    this.socket.send("new-farm")

  })

}
```

```
  })
```

```
}
```

```
onMessage(e)
```

```
{
```

```
  let data = JSON.parse(e.data);
```

```
  if (data.farm) {
```

```
    this.setState({"farm": data.farm})
```

```
  }
```

```
  if (data.patches) {
```

```
    this.setState({"patches": data.patches})
```

```
}
```

```
}
```

```
componentWillUnmount()
```

```
{
```

```
  this.socket.removeEventListener(this.onMessage)
```

```
  this.socket = null
```

```
}
```

```
render() {
```

```
  let rows = []
```

```
  let farm = this.state.farm
```

```
  let statePatches = this.state.patches
```

```
for (let y = 0; y < farm.height; y++) {  
  
  let patches = []  
  
  
  for (let x = 0; x < farm.width; x++) {  
  
    let className = "patch"  
  
  
  
    statePatches.forEach((patch) => {  
  
      if (patch.x === x && patch.y === y) {  
  
        className += " " + patch.type  
  
  
  
        if (patch.wet) {  
  
          className += " " + wet
```

```
}
```

```
}
```

```
})
```

```
patches.push(
```

```
  <div className={className}
```

```
    key={x + "x" + y} />
```

```
)
```

```
}
```

```
rows.push(
```

```
  <div className="row" key={y}>
```

```
    {patches}
```

```
        </div>

    )

}

return (

    <div className="farm">{rows}</div>

)

}

}

export default Farm
```

I had forgotten to explain much of what the previous Farm component was doing. React components were a different way of thinking about how to build interfaces. They changed one's thought process from "How do I interact with the DOM when I want to change something?" to "What should the DOM look like

with any given context?"

I was meant to think about the render method as only executing once, and that everything it produced would be dumped into the DOM. I could use methods like `componentWillMount` and `componentWillUnmount` as ways to hook into other data points (like WebSockets). And as I received updates through the WebSocket, I could update the component's state, so long as I had set the initial state in the constructor.

This resulted in an ugly, albeit functional set of divs. I set about adding some styling. From `app/Action/HomeAction.pre`:

```
namespace App\Action;

use Aerys\Request;

use Aerys\Response;

class HomeAction

{

    public function __invoke(Request $request,

        Response $response)

    {
```



```
$js = yield mix("/js/app.js");

$css = yield mix("/css/app.css");


$response->end("

<link rel='stylesheet' href='{ $css}' />

<div class='app'></div>

<script src='{ $js}'></script>

");

}

}
```

From assets/scss/app.scss:

```
.row {

    width: 100%;
```

```
height: 50px;

.patch {

  width: 50px;

  height: 50px;

  display: inline-block;

  background-color: sandybrown;


  &.weed {

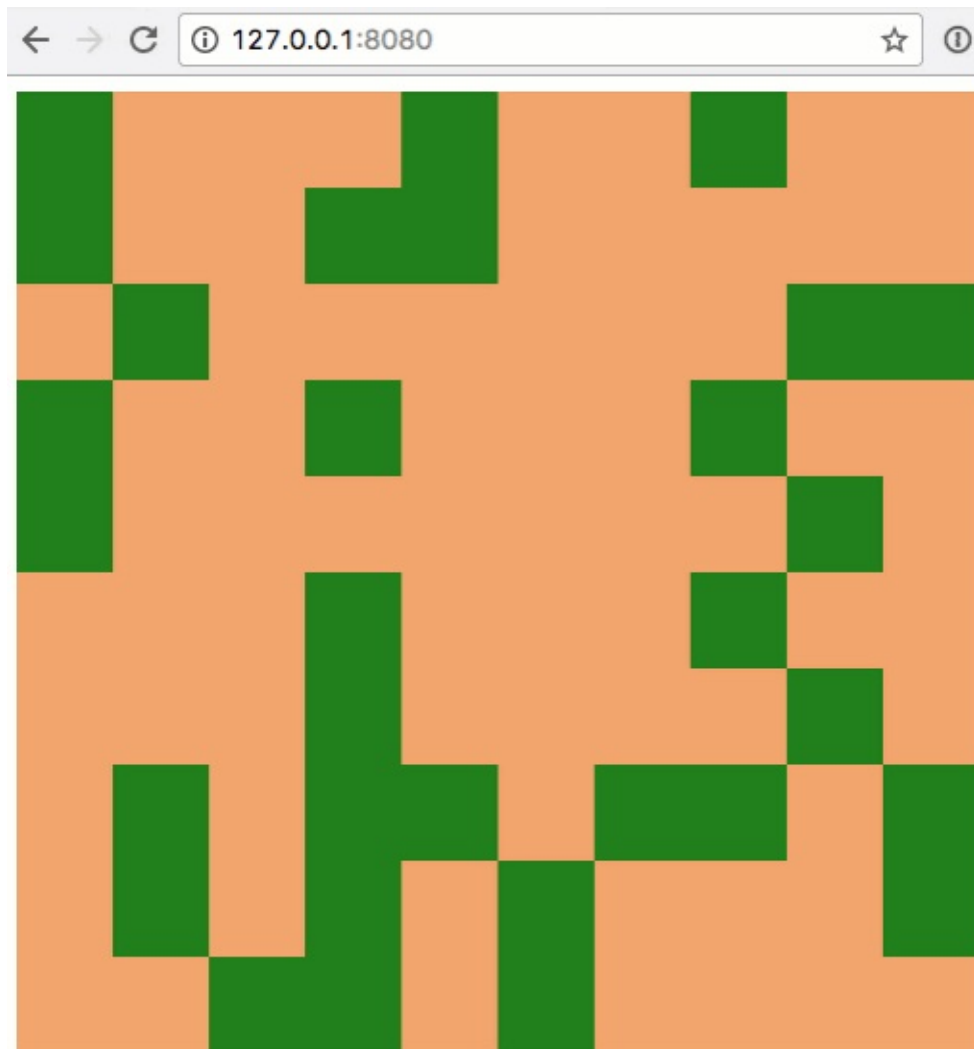
    background-color: green;

  }

}

}
```

The generated farms now had a bit of color to them:



Summary

This was by no means a complete game. It lacked vital things like player input and player characters. It wasn't very multiplayer. But this session resulted in a deeper understanding of React components, WebSocket communication, and preprocessor macros.

I was looking forward to the next part, wherein I could start taking player input, and changing the farm. Perhaps I'd even start on the player login system. Maybe one day!