5
Starting a React Application

The aims of this chapter are to formulate a plan, set up the environment, and scaffold the code. First, we'll work through the application design tasks described in the previous chapter. Then, we'll fetch development tools and configure the programming environment. In the end we'll have a running skeleton of our application and will have covered the following technical subjects:

- **Webpack**: This is the build automation and development server. We'll get a basic configuration working that will service ES6 and JSX compilation, code bundling, hot loading React components, and polyfilling.
- **React application structure**: Though there are many ways to arrange the parts of your application using scaffolding tools, such as **Yeoman**, they make a lot of assumptions during the process. In this chapter we'll arrange the structure ourselves.
- **React router**: The user experience starts at the address bar or a link to your application. There are many expectations that come with using an application or website hosted in a web browser, such as bookmarking and deep linking. Setting up the router early is prudent since this is where everything really begins.

Application design

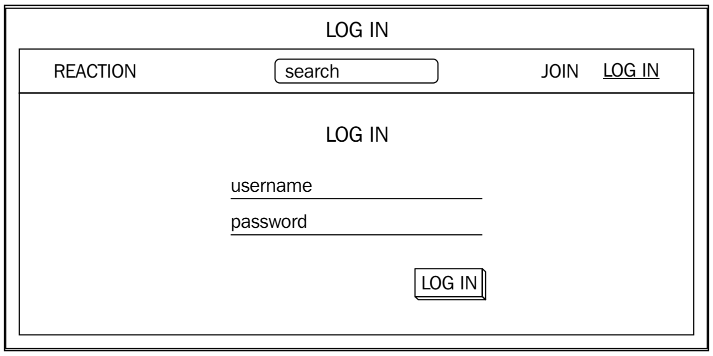
In *Chapter 4*, *Anatomy of a React Application*, we looked at a few design tasks that help to establish intent before coding in earnest. We will repeat those tasks for our blog application. The ultimate goal is to support blog entry for multiple users.

Creating wireframes

Starting with pen and paper is an approachable way to define a problem. We know we'll need a blog post entry screen to author rich text, a means to sign up a new user, a means for that user to log in, and ways to view the posts and the users.

User-related views

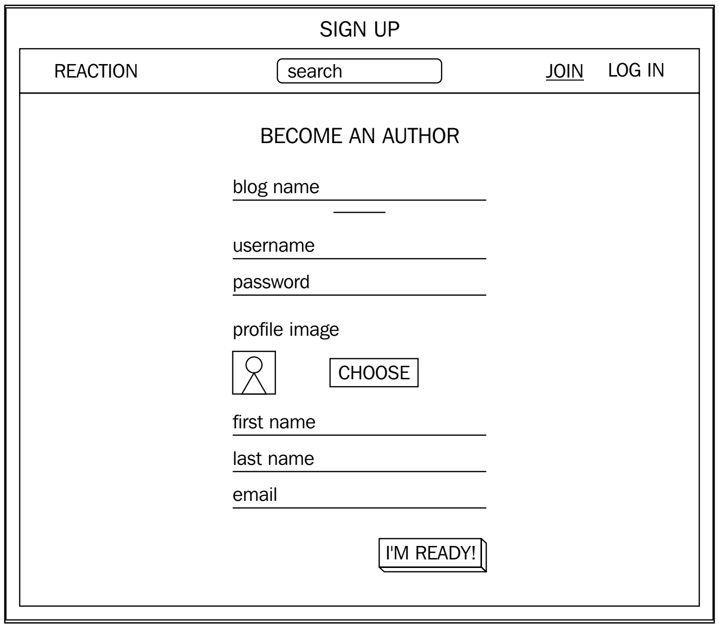
The user-related views include not only the ones that manage the user entity, but also those that manage the login session. The following figure shows the log in view:



Log in view

The log in screen is the simplest form in the application: just one heading, two fields, and a submit button.

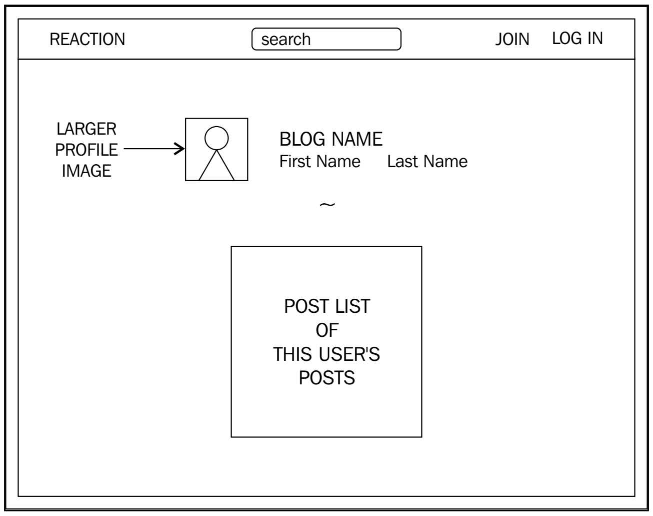
Here is the user sign-up screen. It is the largest and most complex input form in the application:



User create (sign-up) view

The sign-up screen could also be used to edit an existing user account. Using the same component for creation and editing is a common practice. When we implement this screen in the next chapter, we will use a trick to get a profile image from disk and persist it to the document database as text.

Finally, the user view displays a read-only form of the user profile and a filtered list of posts authored by the specific user:

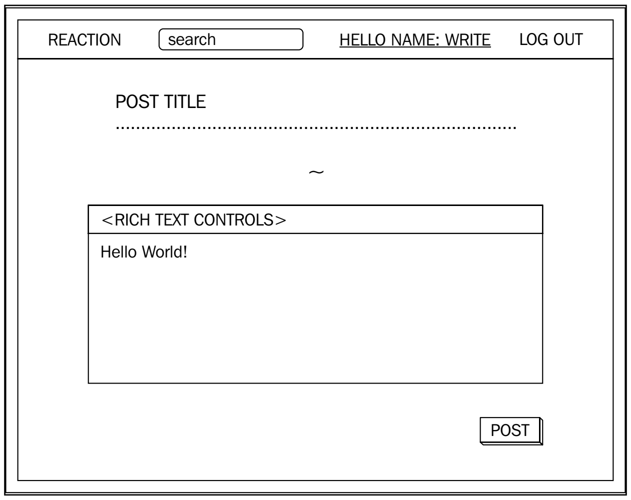


User view

Our first representation of a user includes the profile image, the name of their blog, and their name. The user's posts will be displayed beneath the profile information.

Post-related views

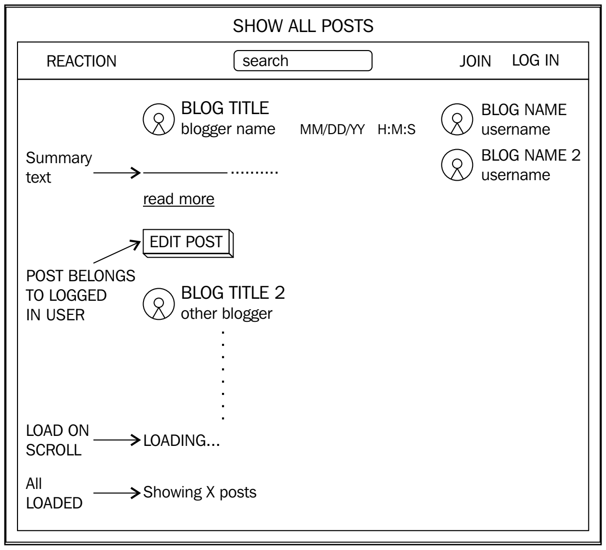
The wireframes in this section constitute the screens for creating and viewing posts. First, let's look at the wireframe for the post creation view:



Create post view

This is where we'll put our rich text editor, Quill. The post title is a large, prominent, input field followed by a separator.

The next wireframe depicts the default view for the application, the post list view:

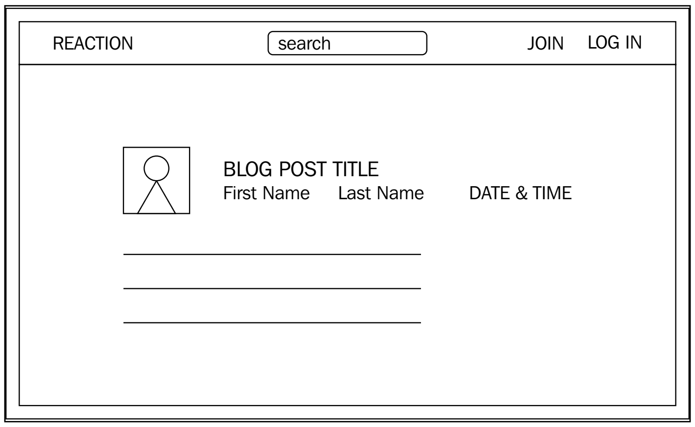


Post list view

This is the home view. It depicts the application header, the list of all posts, a loading message when the user scrolls, and a final **Showing X Posts** message with the total number of posts currently loaded. The user list appears on the right side of the screen. Clicking one of the users navigates to the user view screen shown in the previous figure.

There are two instances of obvious component repetition: a list of posts that now appears both on the user view and this post list view, and a representation of the user that has the photo, the name of their blog, and their name. When we set up the file structure, we'll account for these as reusable components.

Finally, the last post-related wireframe is for a single post entry:



View Post

The top portion of the post view is a mixture of user data and post information. The post title, date, and time are from the post data and the user photo and name are from the user data. The lines in the wireframe represent rendered markup created by the Quill editor from the create post view.

Data entities

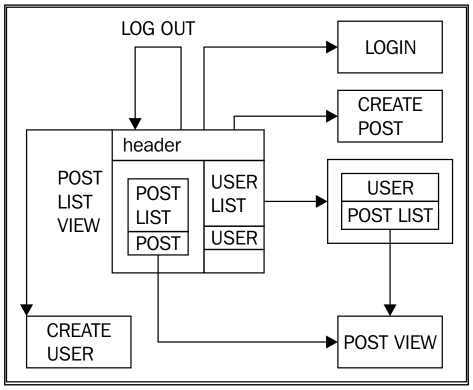
Users and posts are the primary data entities. Post deletion and user editing and deletion are left unimplemented for the sake of brevity, but could easily be added by you later. Other enhancement ideas are suggested at the end of the application tour in *Chapter 9*, *React Blog App Part 4 – Infinite Scroll and Search*.

Entity Name	**Data Members**	**Operations**
Posts	• post id • user id • body • date • summary	• create • edit
Users	• user id • blog name • user name • password • profile image • first name • last name	• create

Main views and the sitemap

Almost every main view is already sketched up from our wireframes. To capture the user workflows through the app, add the header bar component to that list. The header component is the first place an author will go to sign up, log in, or create a new post. It's the most explicit navigation in the application, resulting from direct user interaction. The navigation that occurs as a result of clicking on a post or user is subtler. The navigation that occurs after completing the create user or create post forms is automatic. The obvious destination choice is the read-only view version of the successfully submitted item.

All of the aforementioned navigation is represented in the next figure as an arrow in our final design task, the sitemap:



Sitemap

It is apparent from the wireframes that the header is in each main view, but it's only represented on the home view (all posts) of the sitemap in order to depict its navigation options just once.

Preparing the development environment

Before we start coding, we'll need to get our development tools installed and configured. This involves installing some Node modules and writing the Webpack configuration file.

Installing Node and its dependencies

Node.JS is needed to run Webpack automation, the Webpack dev server, and the JSON mock server. Head over to `Nodejs.org` and follow the installation instructions for your operating system. You'll also need a terminal to run commands and view the output of Webpack as it runs compilation steps. The Windows terminal is serviceable, but if you install Git for Windows it comes with a better shell called git-bash (which is part of MinGW, a minimalist GNU environment for Windows).

Initialize a Node project by running `npm init` and answering the prompts. The defaults will work for us, so you can just press *Enter* and accept each default. This will create the `package.json` file, which will contain a manifest of module versions for our development dependencies. If you haven't already, open a terminal, create a new directory, and execute the command:

```
npm init
```

Before installing and configuring Webpack, some Node packages must be fetched; these will be needed for the application code. If you need a refresher on why we've chosen these particular packages you can flip back to the previous chapter.

As mentioned before, Node Package Manager (npm) uses the configuration file named `package.json`. This file contains some metadata about your project, but the most important part is the manifests of modules and their respective versions. Node and `npm` use a pragmatic versioning scheme called **semver** (semantic versioning). Semver makes it clear when there's a breaking API change in a module revision. The version numbers consist of three components: major, minor, and patch. Different versions of the major component indicate API incompatibility. Different versions of the minor component denote backward-compatible API additions within the same major version. Changes to the patch component indicate defect fixes, and are always backward-compatible. The packages listed in `package.json` will each have a complete version, often with a tilde '~' in front of it. This means any non-breaking version near the stated version number will suffice when fetching updates or installing a fresh set of modules when there are none currently in the `node_modules` folder.

It can be tedious to manage the versions in `package.json`. Luckily, when you execute `npm install` you can ask `npm` to add version detail for a newly fetched module to the configuration in `package.json`. There are two kinds of dependencies: ones that are required for the app to run in production, and ones that are only needed for development. To make `npm` append the version detail to the file for application runtime dependencies, you can add the option `--save` to your `npm install` command. To get version information added to `package.json` for development dependencies, use `--save-dev`. Note that these web projects move very fast and change APIs from time to time. If you have problems with any of the code in this book, it's a good idea to reference the versions of modules referenced in the `package.json` file included in the respective chapter's code zip file.

To install our application dependencies, run the commands below. Alternatively you can just use the `package.json` file from one of the code zip files for this chapter.

```
npm install --save react
npm install --save react-dom
npm install --save react-addons-update
npm install --save react-router
npm install --save history
npm install --save reflux
npm install --save reflux-promise
npm install --save superagent
npm install --save classnames
npm install --save quill
npm install --save moment
```

Let's talk briefly here about Moment. If you code your own date manipulation math and formatting, you will have a bad time. It may seem simple on the surface, but it is definitely not. Moment is the premier date library for JS. Why aren't there loads of date libraries like everything else in JS? Because people don't want to (and usually shouldn't) code date and time math!

There is another item here that was not discussed in *Chapter 4*, *Anatomy of a React Application*. `Classnames` is a generic CSS class name string constructor module with semantics very similar to `ng-class` in Angular. It's a handy way to construct class names together based on the state of our React components. So handy, in fact, that it was part of the React addons for a while before it was broken out into a separate utility. You can find the documentation and source on GitHub under JedWatson's account.

The `react-dom` package is necessary for rendering and finding DOM elements. It was split from the main React package in version 0.14 in order to make things more modular. Similarly, React addons have also been split out. We'll use the `react-addons-update` module to create copies of objects.

The `history` module is used by react-router to manage browser history.

An update to the Reflux project split out the promise interface for asynchronous actions. So, `reflux-promise` is included here to add the promise interface back into those actions.

Installing and configuring Webpack

Install Webpack and the Webpack dev server. Here the Webpack dev server is installed globally so that the command is easily available from any command path.

```
npm install --save webpack
npm install -g webpack-dev-server
```

Within our Webpack configuration we are going to use the following: the Babel JS transpiler for ES6 and JSX, the React hot-loader so we'll have to refresh our browser less often, and the Webpack dev server to host our application locally. Reusable Webpack components are called **loaders**. You'll find that similar sorts of pluggable pieces are called tasks in **Gulp** or **Grunt**. They are transformations on source files performed in a pipeline fashion. Install each of these modules using the `npm` command:

```
npm install --save babel
npm install --save babel-core
npm install --save babel-polyfill
npm install --save babel-loader
npm install --save babel-preset-es2015
npm install --save babel-preset-react
npm install --save-dev react-hot-loader
```

A few more loaders and supporting modules are needed for Less CSS pre-processing:

```
npm install --save less
npm install --save less-loader
npm install --save style-loader
npm install --save css-loader
npm install --save autoprefixer-loader
```

Finally, install the mock dev server for our REST interface.

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

We'll need to restart `the webpack-dev-server` from time to time. To make this simple, add a script to the `package.json` file. Inside that file, there's a member called `scripts` with a default `test` target. Alongside the `test` member, add one called `start` with a dev server start string like this.

```
...
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "webpack-dev-server --progress --colors --watch"
}
```

Don't forget to add that comma after the `test` value!

The Webpack configuration

The listing for the Webpack configuration file is a bit lengthy. It is broken into sections in the text, but it's really just one JS object. Once this configuration file is written, it still won't be ready to run until we start scaffolding the entry points. You can follow along with the listing by looking at the `webpack.config.js` file included in `ch5-1.zip`.

Webpack configuration starts by including the path module and the Webpack module. We'll be specifying a lot of directory locations in this file. The path module is used to make definitions of file paths simple and safe across operating systems. The path module handles operating system differences, such as different slashes used as path separators. The entire Webpack configuration is a single object exported in `webpack.config.js`.

```
var path    = require('path')
,   webpack = require('webpack')
;

module.exports = {
```

Entry and output sections

The Webpack configuration file starts with the `entry` and `output` sections. We only have one entry point and one output file, but you could define multiple ones if desired. The `entry` modules are loaded in order and the last one is exported. The first item in `entry` enables the client portion of our dev server. The second provides an avenue for our react-hot loader to push updated React components to our application without a refresh.

We can get all the non-transformable code-backed polyfills for ES6 by including the babel-polyfill runtime. This includes features such as generators, promises, array map, and many more. Further, this Babel runtime polyfill technique doesn't pollute the global namespace as traditional ones can.

Finally, our app entry point is listed. The `output` value is the file name that will go into the `index.html` file. This final output file will include all of the transformed code.

```
    entry: [
      // WebpackDevServer host and port
      'webpack-dev-server/client?http://localhost:8080',
      'webpack/hot/only-dev-server',
      'babel-polyfill',
      './js/app' // Your app's entry point
    ],
    output: {
        filename: "js/bundle.js"
    },
```

The plugins section

Next is the plugins section. `HotModuleReplacementPlugin` allows the server to push changed JS modules into the browser execution context without a page refresh. Next, `NoErrorsPlugin` will prevent erroneously built code from propagating into our hot-loaded browser environment, where it would certainly cause exceptions. Error output from the build will still appear on the console where we execute our dev server via `npm start`, but the bundle `app.js` file will not be rebuilt and replaced until the compile errors are resolved.

I've left a small snippet in the plugins section for an array utilities polyfill. We won't need it since Babel will provide array utility functions (such as map, forEach, reduce, and so on) as part of the transpilation process. The commented out example is left in the listing to show how to include some global JavaScript in your Webpack bundle for any cases where global polyfilling is needed.

```
    plugins: [
      new webpack.HotModuleReplacementPlugin(),
      new webpack.NoErrorsPlugin(),

      // example of polyfilling with webpack
      // alternatively, just include the babel runtime option below
      // and get Promises, Generators, Map, and much more!
      // You can even get forward looking proposed features
      // for ES7 and beyond with the
      // stage query parameter below
      // https://babeljs.io/docs/usage/experimental/
      // welcome to the future of JavaScript! :)
      //new webpack.ProvidePlugin({
      //  'arrayutils': 'imports?this=>global!exports?global.
  arrayutils!arrayutils'
      //})
    ],
```

The resolve section

This section is for file location resolution. The extensions array is used when Webpack attempts to locate an imported code file in our application code. Alias is just what it sounds like, short names for paths during module search. Since Node has a handy `__dirname` variable we can make an alias for our application root. We'll use this throughout the app when we import modules instead of grappling with relative paths.

```
    resolve: {
      // require files in app without specifying extensions
      extensions: ['', '.js', '.json', '.jsx', '.less'],
      alias: {
        // convenient anchor point for nested modules
        'appRoot': path.join(__dirname, 'js'),
        'vendor': 'appRoot/vendor'
      }
    },
```

The module section

The module section contains Webpack loaders. The loaders look cryptic, but they are simply a transformation pipeline for text files. They are applied by testing filenames using regular expressions. We've added the Less autoprefixer plugin to the transformation pipeline for `.less` files in order to automatically inject CSS vendor prefixes for the browsers specified by the `browsers` parameter.

The most interesting loader section in our list is the one for `.js` and `.jsx` files. This loader pipeline runs Babel on our files; this does both the ES6 and JSX transformations. When a loader pipeline contains multiple items, as it does here, they are applied from right to left. The react hot loader is to the left of Babel so that it runs after it.

```
        module: {
          loaders: [
            {
              test: /\.less$/,
              loader: 'style-loader!css-
  loader!autoprefixer?browsers=last 2 version!less-loader'
            },
            {
              test: /\.css$/,
              loader: 'style-loader!css-loader'
            },
            {
              test: /\.(png|jpg)$/,
              // inline base64 URLs for <=8k images,
              // use direct URLs for the rest
              loader: 'url-loader?limit=8192'
            },
        {
          test: /\.jsx?$/,
          include: [
            // files to apply this loader to
            path.join(__dirname, 'js')
          ],
          // loaders process from right to left
          loaders: [
            'react-hot',
            'babel?presets[]=react,presets[]=es2015',
            'reflux-wrap-loader'
          ]
        }
      ]
    } // end module
  };
```

Babel is separated completely into submodules as of version 6. This means that, without these other modules, Babel will do nothing to transform the files. The Babel submodules are called **presets** in Babel parlance. Here we have included the react preset, which handles the JSX compilation, and the `es2015` preset, which provides all of our ES6 goodness.

Ahead of the Babel and react-hot loaders is the reflux-wrap-loader. The source for this loader should be located in the file `web_modules/reflux-wrap-loader/index.js`. Go ahead and make this directory structure, and use the code listing below for the `index.js` file inside the `reflux-wrap-loader` directory. Webpack automatically searches the `node_modules` and `web_modules` directories for loaders. The reflux-wrap-loader is a simple example of a loader.

```
module.exports = function (source) {
  this.cacheable && this.cacheable();
  var newSource;

  if (/reflux-core.*index.js$/.test(this.resourcePath)) {
    newSource = ";import RefluxPromise from 'reflux-promise';\n";
    newSource += source;
    newSource += ";\nReflux.use(RefluxPromise(Promise));";
  }
  return newSource || source;
};
```

As mentioned before, a loader simply transforms text files. Since the Reflux project split out promises into a separate package, it's necessary to call `Reflux.use` on the `reflux-promise` package. There's not a great place to do this in the application modules, since every module would have to do it to be sure it was done without worrying about execution order. So, instead it's done here in a loader by wrapping the Reflux module and adding the invocation for the import and the use method. The loader matches on the reflux-core file and frames it with the invocation text to include the promise interface. This loader will be run before the Babel transformation, so we can use the ES6 import syntax without generating a build error. The loader will run on every file so you must be sure to return the original source if you want the loader to do nothing to the file. The `cacheable()` invocation instructs Webpack that, after the transformation, the result for the reflux-core file can be cached indefinitely.

Considerations before starting

Whew! I think we're ready to start, but before we dive into the code I want to impart how I believe you should think about the render step in the React component lifecycle as well as a way to approach browser support and form validation.

React and rendering

The way you should think about the React render function is much like the definition of a mathematical function: `f(state) = UI`. React treats the DOM as a rendering target, much like a computer program or game treats the **Graphics Processing Unit** (**GPU**). State is kind of like the arrays of object data (vertices, and so on). That data is prepared for shipment to the GPU, and the rendering portion of the component lifecycle is kind of like the OpenGL rendering pipeline that consumes that data (state). The render function itself would be the shader code that processes the geometry and pixels in this analogy, and the virtual DOM would be the framebuffer. In the GPU analogy, object data and state go into the render pipeline and the result is an array of pixel data. In React, state goes into the render pipeline and a virtual DOM tree is the result.

This means that, when a potential render cascade begins (shouldComponentUpdate à componentWillUpdate à render), you shouldn't change the state. It's already too late for that sort of thing. That's worth saying again; the render function should be pure and never affect state or props. You are welcome to make intermediary variables in the render function to create projections (transformations) of state that are easier to consume by render logic, just don't change the state itself.

This application structure is quite powerful and is the reason that the React Native project can exist. The DOM isn't something that you typically interact with directly in React as you would with jQuery, for instance. Other native targets, such as iOS Cocoa, are just other rendering targets with their own specific render function. This one-way structure and focus on efficiency (for what is often the most expensive part of an application, getting pixels onto the screen) is what makes React special and, in some ways, more flexible for cross-platform development than other JavaScript libraries.

Browser support

For our application we want to use modern browsers: browsers that the vast majority of people use. This means being a bit choosy. Of course, in a real scenario you have to consider the target users who will actually use the application and support them, even if it means substantial additional effort. For example, if you make a government services website you may need to support an older IE browser if some users are citizens who use library computers that are often older and locked into running older browsers.

Conventional wisdom in the web community often asserts that we should start with the lowest common denominator (that is, the worst browser in terms of feature support) and work our way up to the fancier browsers. This is known as progressive enhancement. This is slightly at odds with a technique known as polyfilling in which newer features are back-ported to older browsers.

Polyfills are JavaScript code included to port features into browsers that do not yet support them natively. They enable you to write code as if the feature exists in an older browser. If it does already exist in the browser, the polyfill does nothing and the native code is used. They are often very cheap in terms of code size and performance. While not always on par with native speed, performance is acceptable (in some cases better!). A moment ago, I mentioned that this is slightly at odds with progressive enhancement because, with polyfills, you write code as if those features exist in browsers where they do not. However, it's not entirely at odds with the progressive enhancement strategy. One could argue that starting with the lowest supported browser set, then polyfilling, and then moving on to more complex features that will only be exhibited in newer browsers still fits the strategy. If you are supporting older browsers and following the progressive enhancement strategy, then significant user goals should still be achievable on those older browsers.

For our purposes (learning new stuff), we are focusing on interesting new technology and getting a prototype running easily. So, we'll start with our desired tech and fill it in using polyfills where we are compelled to do so, but we'll avoid writing two or more specific portions of code that have the same effect for two or more browsers.

Using the Babel runtime trick that you saw in the Webpack configuration section eliminates our immediate need for traditional polyfilling.

Form validation

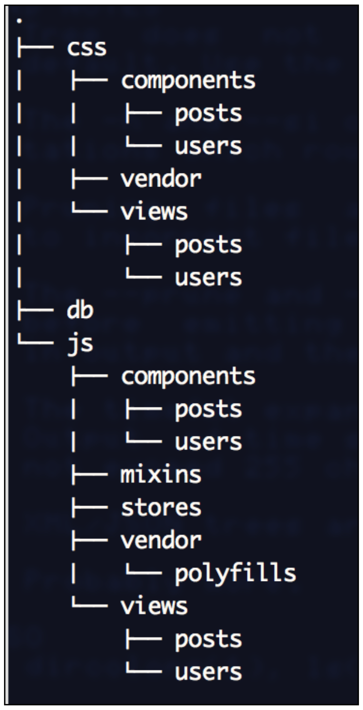
In *Chapter 3, Dynamic Components, Mixins, Forms, and More JSX* there was some discussion about validation. In that discussion, three means of immediate (field-level) validation were explored: view level, view model level, and model level. A conclusion there stated that, for system consistency, a model containing the constraints and a shared mechanism to exercise constraints was an ideal architecture. While that is still the contention of this text, in this chapter we want to focus on application structure, mostly views. So, we are going to cheat a little and do minimal validation in the view via a small helper. The constraints will reside inside their respective view components. This is, in part, due to the fact that we are using JSON Server as a mock back-end. To build a model of validation constraints while using a simple document store would take time away from exploring our primary application architecture components (views, stores, and actions) and their interconnections.

Starting the app

We will begin exploring the construction of the application in detail in the next chapter. To finish up here, we'll lay the groundwork for file structure, and stub out the main views. We'll boot up our dev server and mock back-end, then add some linking between views using React Router.

The directory structure

Make a directory structure that looks like this:



```
.
├── css
|   ├── components
|   |   ├── posts
|   |   └── users
|   ├── vendor
|   └── views
|       ├── posts
|       └── users
├── db
└── js
    ├── components
    |   ├── posts
    |   └── users
    ├── mixins
    ├── stores
    ├── vendor
    |   └── polyfills
    └── views
        ├── posts
        └── users
```

If you are using a POSIX shell (such as Bash, the default shell on Mac OSX), here are a couple of commands to quickly create the directory structure:

```
mkdir -p db css/{components,vendor,views} js/{components,mixins,stores,ve
ndor,views}
mkdir -p {css,js}/{components,views}/{users,posts} js/vendor/polyfills
```

Notice that the components and views subdirectories in `js` are mirrored in the `css` directory. Mirroring view and component structures in the style directories is an easy way to keep track of which styles belong to which JS constructs. Don't forget the directory made earlier for the reflux-wrap-loader.

The mock database

In the `db` directory, create a file called `db.json` with the following content:

```
{"posts":[], "users":[]}
```

That's it! That's our database for `json-server`. If you want to try running it, open a new terminal and execute this command from the root of your app:

```
json-server db/db.json
```

index.html

The `index.html` file is merely a shell to include the application as bundled by Webpack.

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript">
      WebFontConfig = {
        google: { families: [ 'Open+Sans:300italic,400italic,600italic
,700italic,800italic,400,300,600,700,800:latin' ] }
      };
      (function() {
        var wf = document.createElement('script');
        wf.src = ('https:' == document.location.protocol ? 'https' :
'http') +
        '://ajax.googleapis.com/ajax/libs/webfont/1/webfont.js';
        wf.type = 'text/javascript';
        wf.async = 'true';
        var s = document.getElementsByTagName('script')[0];
        s.parentNode.insertBefore(wf, s);
      })();
    </script>
  </head>
  <body>
    <div id="app"></div>
    <script src="js/bundle.js"></script>
  </body>
</html>
```

You may remember that our app bundle output file is `js/bundle.js` from the Webpack configuration. There is a `<div>` tag with the id app that we'll target to render the React app. The script tag at the top is font loading code taken from Google Fonts.

js/app.jsx

This is where it all starts. This main application file includes the React Router, primary views, and the router configuration.

```
import React     from 'react';
import ReactDom  from 'react-dom';
import { Router, Route, IndexRoute } from 'react-router';
import CSS       from '../css/app.less';
import AppHeader from 'appRoot/views/appHeader';
import Login     from 'appRoot/views/login';
import PostList  from 'appRoot/views/posts/list';
import PostView  from 'appRoot/views/posts/view';
import PostEdit  from 'appRoot/views/posts/edit';
import UserList  from 'appRoot/views/users/list';
import UserView  from 'appRoot/views/users/view';
import UserEdit  from 'appRoot/views/users/edit';

// Components must be uppercase - regular DOM is lowercase
let AppLayout = React.createClass({
render: function () {
    return (
      <div className="app-container">
        <AppHeader />
        <main>
          {React.cloneElement(this.props.children, this.props)}
        </main>
      </div>
    );
  }
});

let routes = (
<Route path="/" component={ AppLayout }>
  <IndexRoute component={ PostList } />
    <Route
      path="posts/:pageNum/?"
      component={ PostList }
      ignoreScrollBehavior
    />
    <Route
      path="/posts/create"
      component={ PostEdit }
    />
    <Route
```

```
        path="/posts/:postId/edit"
        component={ PostEdit }
      />
      <Route
        path="posts/:postId"
        component={ PostView }
      />
      <Route
        path="/users"
        component={ UserList }
      />
      <Route
        path="/users/create"
        component={ UserEdit }
      />
      <Route
        path="/users/:userId"
        component={ UserView }
      />
      <Route
        path="/users/:userId/edit"
        component={ UserEdit }
      />
      <Route
        path="/login"
        component={ Login }
      />
      <Route path="*" component={ PostList } />
    </Route>
  );

  ReactDom.render(<Router>{routes}</Router>, document.
  getElementById('app'));
```

React and React Router are imported because most of this file is router configuration. Next is a set of imports for our top-level views. Those are the views designed in the wireframes and sitemaps. After the imports, we see our first React component, which represents the app itself: `AppLayout`. The root of the router configuration will target this component.

After the application component, the JSX representing the routing table is defined as **routes**. This router configuration maps URL paths in a nested structure directly to the views that will be composed into the application component. These tags won't actually be rendered. They are used as configuration.

Each route has a path for the URL and a component, which is one of the top-level views imported at the top of the file. A special tag named `IndexRoute` could be considered the home view. Our home view is the `PostList` top-level view that shows all the blog posts. Finally, there's a `*` route that's like a 404 handler for the front-end. By putting `PostList` as the handler for this route we ensure that, if someone types in a random URL, the app will route to the home view.

> Be sure to capitalize the names of your components. In React, lower-case component names are reserved for built-in components that correspond to DOM tags.

Last, the Router component is rendered. This causes the router to begin listening to URL changes. The top route `/` uses the `AppLayout` component. The components that should be rendered by the routes are supplied as children. Those children are included in the `AppLayout` and any relevant props supplied through the router are applied to them through the `React.cloneElement` method within the `AppLayout` source.

Main views

If we try to run the code now, it will emit errors because our imports in `app.jsx` won't resolve to files on disk. So, go ahead and make some components for the main views. After that, we'll wire them up to exercise the links shown as arrows in our sitemap.

The main views all reside in the `js/views` directory. For each of the main views, `login.jsx` and `appHeader.jsx`, as well as `edit.jsx`, `list.jsx`, and `view.jsx` in both the `posts/` and `users/` directories, make a simple React component container. We'll start each of these by importing React. This goes at the top of each file.

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

Then, export a React component with the minimum requirement, a render function.

```
export default React.createClass({
  render: function () {
    return ();
  }
});
```

The render function needs to return React DOM. Using the following table for each of the files, add a JSX tag inside the return parentheses with a `className` and some content that has the name of the component. Each of these files, once again, is in the `js/views` directory.

Note: Your render function return value, and any other portion of JSX code you set on a variable, must always have only one root tag.

File	JSX
`login.jsx`	`<form className="login-form">login form</form>`
`appHeader.jsx`	`<header className="app-header">app header</header>`
`posts/edit.jsx`	`<form className="post-edit">post edit</form>`
`posts/list.jsx`	`<div className="post-list-view">post list view</div>`
`posts/view.jsx`	`<div className="post-view-full">post view</div>`
`users/edit.jsx`	`<form className="user-edit">user edit</form>`
`users/list.jsx`	`<ul className="user-list"><li>user list</li></ul>`
`users/view.jsx`	`<div className="user-view">user view</div>`

Before booting up the Webpack dev server, create the `app.less` file also referenced by an import in `app.jsx`. Just add an empty file for now.

Now, boot the dev server by executing `npm start` in another terminal. If you navigate to `localhost:8080` in your browser, you should see a pretty sparse page that says **app header** and **post list view**. You can also visit the routes we defined and see the stubbed views.

At this point your code should look like the code in `ch5-1.zip`.

Linking views with React Router

There are three main ways to transition views through React Router. The first simply changes the URL in the address bar of the browser. The second involves using the `Link` component supplied by the library. The third employs the `History` mixin, which will surface the `pushState` method on your component. We'll look at these usages for the app. The next few changes can be found in `ch5-2.zip`.

js/views/appHeader.jsx

The application header component needs a link to the login view. The `Link` component is imported from React Router. The `Log In` link is the simplest form of `Link` without any extra parameters. It is added to the `appHeader.jsx` file render function like this:

```
import React    from 'react';
import { Link } from 'react-router';

export default React.createClass({
  render: function () {
    return (
      <header className="app-header">
        app header
        <Link to="/login">Log In</Link>
      </header>
    );
  }
});
```

js/views/login.jsx

Now that we can get to the log in view, modify the `login.jsx` component with a `Log In` button.

```
import React       from 'react';
import { History } from 'react-router';

export default React.createClass({
  mixins: [ History ],
  logIn: function (e) {
    this.history.pushState('', '/');
  },
  render: function () {
    return (
      <form className="login-form" onSubmit={this.logIn}>
        <button type="submit">Log In</button>
      </form>
    );
  }
});
```

Here we've added the `History` mixin to get access to the `pushState` function. The component `logIn` function is triggered when the form is submitted. The `pushState` function forwards the user to the root route, `/`. In the next chapter, it will actually log the user in. If your dev server is still running, you should see your browser window update automatically as soon as you save the file.

Summary

In this chapter, we started the application not by jumping directly into the code, but by doing a bit of planning first. The design tasks we explored are a good way to get your priorities straight. We also laid the groundwork for the application by setting up the Webpack build pipeline and scaffolding out all of our main views.

In the next four chapters, the blog application will be built in earnest. This is done in four major parts:

Chapter 6, React Blog App Part 1 – Actions and Common Components

Chapter 7, React Blog App Part 2– Users

Chapter 8, React Blog App Part 3 – Posts

Chapter 9, React Blog App Part 4 – Infinite Scroll and Search