```
    });
  };
```

The objective of `onFormSubmit()` hasn't changed. It is still responsible for either adding a person to the list, or preventing that behavior if there are validation issues. To check for validation errors, we call `this.validate()`, and if there are any, we return early before adding the new person to the list.

Here's what the current version of `validate()` looks like:

```
  validate () {
    const person = this.state.fields;
    const fieldErrors = this.state.fieldErrors;
    const errMessages = Object.keys(fieldErrors).filter((k) => fieldErrors[k])

    if (!person.name) return true;
    if (!person.email) return true;
    if (errMessages.length) return true;

    return false
  },
```

Put simply, `validate()` is checking to make sure the data is valid at the form level. For the form to pass validation at this level it must satisfy two requirements: (1) neither field may be empty and (2) there must not be any field-level validation errors.

To satisfy the first requirement, we access `this.state.fields` and ensure that both `state.fields.name` and `state.fields.email` are truthy. These are kept up to date by `onInputChange()`, so it will always match what is in the text fields. If either `name` or `email` are missing, we return `true`, signaling that there is a validation error.

For the second requirement, we look at `this.state.fieldErrors`. `onInputChange()` will set any field-level validation error messages on this object. We use `Object.keys` and `Array.filter` to get an array of all present error messages. If there are any field-level validation issues, there will be corresponding error messages in the array, and its length will be non-zero and truthy. If that's the case, we also return `true` to signal the existence of a validation error.

`validate()` is a simple method that can be called at any time to check if the data is valid at the form-level. We use it both in `onFormSubmit()` to prevent adding invalid data to the list and in `render()` to disable the submit button, providing nice feedback in the UI.

And that's it. We're now using our custom `Field` component to do field-level validation on the fly, and we also use form-level validation to toggle the submit button in real-time.

Remote Data

Our form app is coming along. A user can sign up with their name and email, and we validate their information before accepting the input. But now we're going to kick it up a notch. We're going to explore how to allow a user to select from hierarchical, asynchronous options.

The most common example is to allow the user to select a car by year, make, and model. First the user selects a year, then the manufacturer, then the model. After choosing an option in one `select`, the next one becomes available. There are two interesting facets to building a component like this.

First, not all combinations make sense. There's no reason to allow your user to choose a 1965 Tesla Model T. Each option list (beyond the first) depends on a previously selected value.

Second, we don't want to send the entire database of valid choices to the browser. Instead, the browser only knows the top level of choices (e.g. years in a specific range). When the user makes a selection, we provide the selected value to the server and ask for next level (e.g. manufacturers available for a given year). Because the next level of options come from the server, this is an asynchronous.

Our app won't be interested in the user's car, but we will want to know what they're signing up for. Let's make this an app for users to learn more JavaScript by choosing a NodeSchool[59] workshop to attend.

A NodeSchool workshop can be either "core" or "elective". We can think of these as "departments" of NodeSchool. Therefore, depending on which department the user is interested in, we can allow them to choose a corresponding workshop. This is similar to the above example where a user chooses a car's year before its manufacturer.

If a user chooses the core department, we would enable them to choose from a list of core workshops like `learnyounode` and `stream-adventure`. If instead, they choose the elective department, we would allow them to pick workshops like `Functional JavaScript` or `Shader School`. Similar to the car example, the course list is provided asynchronously and depends on which department was selected.

The simplest way to achieve this is with two `select` elements, one for choosing the department and the other for choosing the course. However, we will hide the second `select` until: (1) the user has selected a department, and (2) we've received the appropriate course list from the server.

Instead of building this functionality directly into our form. We'll create a custom component to handle both the hierarchical and asynchronous nature of these fields. By using a custom component, our form will barely have to change. Any logic specific to the workshop selection will be hidden in the component.

Building the Custom Component

The purpose of this component is to allow the user to select a NodeSchool course. From now on we'll refer to it as our `CourseSelect` component.

[59] http://nodeschool.io

However, before starting on our new `CourseSelect` component, we should think about how we want it to communicate with its form parent. This will determine the component's `props`.

The most obvious `prop` is `onChange()`. The purpose of this component is to help the user make a department/course selection and to make that data available to the form. Additionally, we'll want to be sure that `onChange()` to be called with the same arguments we're expecting from the other field components. That way we don't have to create any special handling for this component.

We also want the form to be able to set this component's state if need be. This is particularly useful when we want to clear the selections after the user has submitted their info. For this we'll accept two props. One for `department` and one for `course`.

And that's all we need. This component will accept three `props`. Here's how they'll look in our new `CourseSelect` component:

forms/src/09-course-select.js

```
13  static propTypes = {
14    department: PropTypes.string,
15    course: PropTypes.string,
16    onChange: PropTypes.func.isRequired,
17  };
```

Next, we can think about the `state` that `CourseSelect` will need to keep track of. The two most obvious pieces of state are `department` and `course`. Those will change when the user makes selections, and when the form parent clears them on a submit.

`CourseSelect` will also need to keep track of available courses for a given department. When a user selects a department, we'll asynchronously fetch the corresponding course list. Once we have that list we'll want to store it in our `state` as `courses`.

Lastly, when dealing with asynchronous fetching, it's nice to inform the user that data is loading behind the scenes. We will also keep track of whether or not data is "loading" in our `state` as `_loading`.

The underscore prefix of `_loading` is just a convention to highlight that it is purely presentational. Presentational state is only used for UI effects. In this case it will be used to hide/show the loading indicator image.

Here's what our initial `state` looks like:

forms/src/09-course-select.js

```
19  state = {
20    department: null,
21    course: null,
22    courses: [],
23    _loading: false,
24  };
```

As mentioned above, this component's form parent will want to update the `department` and `course` `props`. Our `componentWillReceiveProps()` method will use the update to appropriately modify the `state`:

forms/src/09-course-select.js

```
26  componentWillReceiveProps(update) {
27    this.setState({
28      department: update.department,
29      course: update.course,
30    });
31  }
```

Now that we have a good idea of what our data looks like, we can get into how the component is rendered. This component is a little more complicated than our previous examples, so we take advantage of composition to keep things tidy. You will notice that our `render()` method is mainly composed of two functions, `renderDepartmentSelect()` and `renderCourseSelect()`:

forms/src/09-course-select.js

```
103 render() {
104   return (
105     <div>
106       { this.renderDepartmentSelect() }
107       <br />
108       { this.renderCourseSelect() }
109     </div>
110   );
111 }
```

Aside from those two functions, `render()` doesn't have much. But this nicely illustrates the two "halves" of our component: the "department" half and the "course" half. Let's first take a look at the "department" half. Starting with `renderDepartmentSelect()`:

forms/src/09-course-select.js

```
106         { this.renderDepartmentSelect() }
```

This method returns a `select` element that displays one of three options. The currently displayed option depends on the `value` prop of the `select`. The option whose `value` matches the `select` will be shown. The options are:

- "Which department?" (value: *empty string*)
- "NodeSchool: Core" (value: `"core"`)
- "NodeSchool: Electives" (value: `"electives"`)

The value of `select` is `this.state.department || ''`. In other words, if `this.state.department` is falsy (it is by default), the `value` will be an *empty string* and will match "Which department?". Otherwise, if `this.state.department` is either `"core"` or `"electives"`, it will display one of the other options.

Because `this.onSelectDepartment` is set as the `onChange` prop of the `select`, when the user changes the option, `onSelectDepartment()` is called with the change event. Here's what that looks like:

forms/src/09-course-select.js

```
33  onSelectDepartment = (evt) => {
34    const department = evt.target.value;
35    const course = null;
36    this.setState({ department, course });
37    this.props.onChange({ name: 'department', value: department });
38    this.props.onChange({ name: 'course', value: course });
39
40    if (department) this.fetch(department);
41  };
```

When the department is changed, we want three things to happen. First, we want to update `state` to match the selected department option. Second, we want to propagate the change via the `onChange` handler provided in the `props` of `CourseSelect`. Third, we want to fetch the available courses for the department.

When we update the `state`, we update it to the value of the event's `target`, the `select`. The value of the `select` is the value of the chosen `option`, either `''`, `"core"`, or `"electives"`. After the `state` is set with a new value, `render()` and `renderDepartmentSelect()` are run and a new option is displayed.

Notice that we also reset the course. Each course is only valid for its department. If the department changes, it will no longer be a valid choice. Therefore, we set it back to its initial value, `null`.

After updating `state`, we propagate the change to the component's change handler, `this.props.onChange`. Because we use the arguments as we have previously, this component can be used just like `Field` and can be given the same handler function. The only trick is that we need to call it twice, once for each input.

Finally, if a department was selected, we fetch the course list for it. Here's the method it calls, `fetch()`:

forms/src/09-course-select.js

```
49  fetch = (department) => {
50    this.setState({ _loading: true, courses: [] });
51    apiClient(department).then((courses) => {
52      this.setState({ _loading: false, courses: courses });
53    });
54  };
```

The responsibility of this method is to take a `department` string, use it to asynchronously get the corresponding course list, `courses`, and update the `state` with it. However, we also want to be sure to affect the `state` for a better user experience.

We do this by updating the `state` *before* the `apiClient` call. We know that we'll be waiting for the response with the new course list, and in that time we should show the user a loading indicator. To do that, we need our `state` to reflect our fetch status. Right before the `apiClient` call, we set the `state` of `_loading` to `true`. Once the operation completes, we set `_loading` back to `false` and update our course list.

Previously, we mentioned that this component had two "halves" illustrated by our `render()` method:

forms/src/09-course-select.js

```
103 render() {
104   return (
105     <div>
106       { this.renderDepartmentSelect() }
107       <br />
108       { this.renderCourseSelect() }
109     </div>
110   );
111 }
```

We've already covered the "department" half. Let's now take a look at the "course" half starting with `renderCourseSelect()`:

forms/src/09-course-select.js

```
108         { this.renderCourseSelect() }
```

The first thing that you'll notice is that `renderCourseSelect()` returns a different root element depending on particular conditions.

If `state._loading` is true, `renderCourseSelect()` only returns a single `img`: a loading indicator. Alternatively, if we're not loading, but a department has not been selected (and therefore `state.department` is falsy), an empty `span` is returned – effectively hiding this half of the component.

However, if we're not loading, and the user *has* selected a department, `renderCourseSelect()` returns a `select` similar to `renderDepartmentSelect()`.

The biggest difference between `renderCourseSelect()` and `renderDepartmentSelect()` is that `renderCourseSelect()` dynamically populates the `option` children of the `select`.

The first option in this `select` is "Which course?" which has an empty string as its value. If the user has not yet selected a course, this is what they should see (just like "Which department?" in the other `select`). The options that follow the first come from the course list stored in `state.courses`.

To provide all the child `option` elements to the `select` at once, the `select` is given a single array as its child. The first item in the array is an element for our "Which course?" option. Then, we use the spread operator combined with `map()` so that from the second item on, the array contains the course options from `state`.

Each item in the array is an `option` element. Like before, each element has text that it displays (like "Which course?") as well as a value `prop`. If the `value` of the `select` matches the `value` of the `option`, that `option` will be displayed. By default, the `value` of the `select` will be an empty string, so it will match the "Which course?" `option`. Once the user chooses a course and we are able to update `state.course`, the corresponding course will be shown.

This is a dynamic collection, we must also provide a `key prop` to each `option` to avoid warnings from React.

Lastly, we provide a change handler function, `onSelectCourse()` to the `select prop onChange`. When the user chooses a course, that function will be called with a related event object. We will then use information from that event to update the state and notify the parent.

Here's `onSelectCourse()`:

forms/src/09-course-select.js

```
43   onSelectCourse = (evt) => {
44     const course = evt.target.value;
45     this.setState({ course });
46     this.props.onChange({ name: 'course', value: course });
47   };
```

Like we've done before, we get the `value` of the `target` element from the event. This `value` is the `value` of whichever `option` the user picked in the courses `select`. Once we update `state.course` with this `value`, the `select` will display the appropriate `option`.

After the `state` is updated, we call the change handler provided by the component's parent. Same as with the department selection, we provide `this.props.onChange()` an object argument with the `name`/`value` structure the handler expects.

And that's it for our `CourseSelect` component! As we'll see in next part, integration with the form requires very minimal changes.

Adding CourseSelect

Now that our new `CourseSelect` component is ready, we can add it to our form. Only three small changes are necessary:

1. We add the `CourseSelect` component to `render()`.
2. We update our "People" list in `render()` to show the new fields (department and course).
3. Since department and course are required fields, we modify our `validate()` method to ensure their presence.

Because we were careful to call the change handler from within `CourseSelect` (`this.props.onChange`) with a `{name, value}` object the way that `onInputChange()` expects, we're able to reuse that handler. When `onInputChange()` is called by `CourseSelect`, it can appropriately update `state` with the new department and course information – just like it does with calls from the `Field` components.

Here's the updated `render()`:

forms/src/09-async-fetch.js

```
  render() {
    return (
      <div>
        <h1>Sign Up Sheet</h1>

        <form onSubmit={this.onFormSubmit}>

          <Field
            placeholder='Name'
            name='name'
            value={this.state.fields.name}
            onChange={this.onInputChange}
            validate={(val) => (val ? false : 'Name Required')}
          />

          <br />

          <Field
            placeholder='Email'
            name='email'
            value={this.state.fields.email}
            onChange={this.onInputChange}
            validate={(val) => (isEmail(val) ? false : 'Invalid Email')}
          />

          <br />

          <CourseSelect
            department={this.state.fields.department}
            course={this.state.fields.course}
            onChange={this.onInputChange}
          />

          <br />

          <input type='submit' disabled={this.validate()} />
        </form>

        <div>
          <h3>People</h3>
          <ul>
```



```
            { this.state.people.map(({ name, email, department, course }, i) =>
              <li key={i}>{[ name, email, department, course ].join(' - ')}</li>
            ) }
          </ul>
        </div>
      </div>
    );
  }
```

When adding `CourseSelect` we provide three `props`:

1. The current department from `state` (if one is present)
2. The current course from `state` (if one is present)
3. The `onInputChange()` handler (same function used by `Field`)

Here it is by itself:

```
<CourseSelect
  department={this.state.fields.department}
  course={this.state.fields.course}
  onChange={this.onInputChange} />
```

The other change we make in `render()` is we add the new department and course fields to the "People" list. Once a user submits sign-up information, they appear on this list. To show the department and course information, we need to get that data from `state` and display it:

```
<h3>People</h3>
<ul>
  { this.state.people.map( ({name, email, department, course}, i) =>
    <li key={i}>{[name, email, department, course].join(' - ')}</li>
  ) }
</ul>
```

This is as simple as pulling more properties from each item in the `state.people` array.

The only thing left to do is add these fields to our form-level validation. Our `CourseSelect` controls the UI to ensure that we won't get invalid data, so we don't need to worry about field-level errors. However, department and course are required fields, we should make sure that they are present before allowing the user to submit. We do this by updating our `validate()` method to include them:

forms/src/09-async-fetch.js

```
53   validate = () => {
54     const person = this.state.fields;
55     const fieldErrors = this.state.fieldErrors;
56     const errMessages = Object.keys(fieldErrors).filter((k) => fieldErrors[k]);
57
58     if (!person.name) return true;
59     if (!person.email) return true;
60     if (!person.course) return true;
61     if (!person.department) return true;
62     if (errMessages.length) return true;
63
64     return false;
65   };
```

Once `validate()` is updated, our app will keep the submit button disabled until we have both department and course selected (in addition to our other validation requirements).

Thanks to the power of React and composition our form was able to take on complicated functionality while keeping high maintainability.

Separation of View and State

Once we've received information from the user and we've decided that it's valid, we then need to convert the information to JavaScript objects. Depending on the form, this could involve casting input values from strings to numbers, dates, or booleans, or it could be more involved if you need to impose a hierarchy by corralling the values into arrays or nested objects.

After we have the information as JavaScript objects, we then have to decide how to use them. The objects might be sent to a server as JSON to be stored in a database, encoded in a url to be used as a search query, or maybe only used to configure how the UI looks.

The information in those objects will almost always affect the UI and in many cases will also affect your application's behavior. It's up to us to determine how to store that info in our app.

Async Persistence

At this point our app is pretty useful. You could imagine having the app open on a kiosk where people can come up to it and sign up for things. However, there's one big shortcoming: if the browser is closed or reloaded, all data is lost.

In most web apps, when a user inputs data, that data should be sent to a server for safe keeping in a database. When the user returns to the app, the data can be fetched, and the app can pick back up right where it left off.

In this example, we'll cover three aspects of persistence: saving, loading, and handling errors. While we won't be sending the data to a remote server or storing it in a database (we'll be using `localStorage` instead), we'll treat it as an asynchronous operation to illustrate how almost any persistence strategy could be used.

To persist the sign up list (`state.people`), we'll only need to make a few changes to our parent form component. At a high level they are:

1. Modify `state` to keep track of persistence status. Basically, we'll want to know if the app is currently loading, is currently saving, or encountered an error during either operation.
2. Make a request using our API client to get any previously persisted data and load it into our `state`.
3. Update our `onFormSubmit()` event handler to trigger a save.
4. Change our `render()` method so that the "submit" button both reflects the current save status and prevents the user from performing an unwanted action like a double-save.

First, we'll want to modify our `state` keep track of our "loading" and "saving" status. This is useful to both accurately communicate the status of persistence and to prevent unwanted user actions. For example, if we know that the app is in the process of "saving", we can disable the submit button. Here's the updated `state` method with the two new properties:

forms/src/10-remote-persist.js

```
16   state = {
17     fields: {
18       name: '',
19       email: '',
20       course: null,
21       department: null
22     },
23     fieldErrors: {},
24     people: [],
25     _loading: false,
26     _saveStatus: 'READY',
27   };
```

The two new properties are `_loading` and `_saveStatus`. As before, we use the underscore prefix convention to signal that they are private to this component. There's no reason for a parent or child component to ever know their values.

`_saveStatus` is initialized with the value `"READY"`, but we will have four possible values: `"READY"`, `"SAVING"`, `"SUCCESS"`, and `"ERROR"`. If the `_saveStatus` is either `"SAVING"` or `"SUCCESS"`, we'll want to prevent the user from making an additional save.

Next, when the component has been successfully loaded and is about to be added to the DOM, we'll want to request any previously saved data. To do this we'll add the lifecycle method `componentWillMount()` which is automatically called by React at the appropriate time. Here's what that looks like:

forms/src/10-remote-persist.js

```
29  componentWillMount() {
30    this.setState({ _loading: true });
31    apiClient.loadPeople().then((people) => {
32      this.setState({ _loading: false, people: people });
33    });
34  }
```

Before we start the fetch with `apiClient`, we set `state._loading` to `true`. We'll use this in `render()` to show a loading indicator. Once the fetch returns, we update our `state.people` list with the previously persisted list and set `_saveStatus` back to `false`.

`apiClient` is a simple object we created to simulate asynchronous loading and saving. If you look at the code for this chapter, you'll see that the "save" and "load" methods are thin async wrappers around `localStorage`. In your own apps you could create our own `apiClient` with similar methods to perform network requests.

Unfortunately, our app doesn't yet have a way to persist data. At this point there won't be any data to load. However, we can fix that by updating `onFormSubmit()`.

As in the previous sections, we'll want our user to be able to fill out each field and hit "submit" to add a person to the list. When they do that, `onFormSubmit()` is called. We'll make a change so that we not only perform the previous behavior (validation, updating `state.people`), but we *also* persist that list using `apiClient.savePeople()`:

forms/src/10-remote-persist.js

```
36  onFormSubmit = (evt) => {
37    const person = this.state.fields;
38
39    evt.preventDefault();
40
41    if (this.validate()) return;
42
43    const people = [ ...this.state.people, person ];
44
45    this.setState({ _saveStatus: 'SAVING' });
46    apiClient.savePeople(people)
47      .then(() => {
48        this.setState({
49          people: people,
50          fields: {
51            name: '',
52            email: '',
53            course: null,
54            department: null
55          },
56          _saveStatus: 'SUCCESS',
57        });
58      })
59      .catch((err) => {
60        console.error(err);
61        this.setState({ _saveStatus: 'ERROR' });
62      });
63  };
```

In the previous sections, if the data passed validation, we would just update our `state.people` list to include it. This time we'll also add the `person` to the `people` list, but we only want to update our `state` if `apiClient` can successfully persist. The order of operations looks like this:

1. Create a new array, `people` with both the contents of `state.people` and the new `person` object.
2. Update `state._saveStatus` to `"SAVING"`
3. Use `apiClient` to begin persisting the new `people` array from #1.
4. If `apiClient` is successful, update `state` with our new `people` array, an empty fields object, and `_saveStatus: "SUCCESS"`. If `apiClient` is *not* successful, leave everything as is, but set `state._saveStatus` to `"ERROR"`.

Put simply, we set the `_saveStatus` to `"SAVING"` while the `apiClient` request is "in-flight". If the request is successful, we set the `_saveStatus` to `"SUCCESS"` and perform the same actions as before. If not, the only update is to set `_saveStatus` to `"ERROR"`. This way, our local state does not get out of sync with our persisted copy. Also, since we don't clear the fields, we give the user an opportunity to try again without having to re-input their information.

For this example we are being conservative with our UI updates. We only add the new person to the list *if `apiClient` is successful*. This is in contrast to an optimistic update, where we would add the person to the list locally *first*, and later make adjustments if there was a failure. To do an optimistic update we could keep track of which `person` objects were added before which `apiClient` calls. Then if an `apiClient` call fails, we could selectively remove the particular `person` object associated with that call. We would also want to display a message to the user explaining the issue.

Our last change is to modify our `render()` method so that the UI accurately reflects our status with respect to loading and saving. As mentioned, we'll want the user to know if we're in the middle of a load or a save, or if there was a problem saving. We can also control the UI to prevent them from performing unwanted actions such as a double save.

Here's the updated `render()` method:

forms/src/10-remote-persist.js

```
  render() {
    if (this.state._loading) {
      return <img alt='loading' src='/img/loading.gif' />;
    }

    return (
      <div>
        <h1>Sign Up Sheet</h1>

        <form onSubmit={this.onFormSubmit}>

          <Field
            placeholder='Name'
            name='name'
            value={this.state.fields.name}
            onChange={this.onInputChange}
            validate={(val) => (val ? false : 'Name Required')}
          />

          <br />

```



```
          <Field
            placeholder='Email'
            name='email'
            value={this.state.fields.email}
            onChange={this.onInputChange}
            validate={(val) => (isEmail(val) ? false : 'Invalid Email')}
          />

          <br />

          <CourseSelect
            department={this.state.fields.department}
            course={this.state.fields.course}
            onChange={this.onInputChange}
          />

          <br />

          {{
            SAVING: <input value='Saving...' type='submit' disabled />,
            SUCCESS: <input value='Saved!' type='submit' disabled />,
            ERROR: <input
              value='Save Failed - Retry?'
              type='submit'
              disabled={this.validate()}
            />,
            READY: <input
              value='Submit'
              type='submit'
              disabled={this.validate()}
            />,
          }[this.state._saveStatus]}

        </form>

        <div>
          <h3>People</h3>
          <ul>
            { this.state.people.map(({ name, email, department, course }, i) =>
              <li key={i}>{[ name, email, department, course ].join(' - ')}</li>
            ) }
          </ul>
```

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

First, we want to show the user a loading indicator while we are loading previously persisted data. Like the previous section, this is done on the first line of `render()` with a conditional and an early return. While we are loading (`state._loading` is truthy), we won't render the form, only the loading indicator:

```
  if (this.state._loading) return <img src='/img/loading.gif' />
```

Next, we want the submit button to communicate the current save status. If no save request is in-flight, we want the button to be enabled if the field data is valid. If we are in the process of saving, we want the button to read "Saving..." to be disabled. The user will know that the app is busy, and since the button is disabled, they won't be able to submit duplicate save requests. If the save request resulted in an error, we use the button text to communicate that and indicate that they can try again. The button will be enabled if the input data is still valid. Finally, if the save request completed successfully, we use the button text to communicate that. Here's how we render the button:

```
{{
  SAVING: <input value='Saving...' type='submit' disabled />,
  SUCCESS: <input value='Saved!' type='submit' disabled/>,
  ERROR: <input value='Save Failed - Retry?' type='submit' disabled={this.valida\
te()}/>,
  READY: <input value='Submit' type='submit' disabled={this.validate()}/>
}[this.state._saveStatus]}
```

What we have here are four different buttons – one for each possible `state._saveStatus`. Each button is the value of an object keyed by its corresponding status. By accessing the key of the current save status, this expression will evaluate to the appropriate button.

The last thing that we have to do is related to the `"SUCCESS"` case. We want to show the user that the addition was a success, and we do that by changing the text of the button. However, "Saved!" is not a call to action. If the user enters another person's information and wants to add it to the list, our button would still say "Saved!". It should say "Submit" to more accurately reflect its purpose.

The easy fix for this is to change our `state._saveStatus` back to `"READY"` as soon as they start entering information again. To do this, we update our `onInputChange()` handler:

forms/src/10-remote-persist.js

```
  onInputChange = ({ name, value, error }) => {
    const fields = this.state.fields;
    const fieldErrors = this.state.fieldErrors;

    fields[name] = value;
    fieldErrors[name] = error;

    this.setState({ fields, fieldErrors, _saveStatus: 'READY' });
  };
```

Now instead of just updating `state.fields` and `state.fieldErrors`, we also set `state._saveStatus` to `'READY'`. This way after the user acknowledges their previous submit was a success and starts to interact with the app again, the button reverts to its "ready" state and invites the user to submit again.

At this point our sign-up app is a nice illustration of the features and issues that you'll want to cover in your own forms using React.

Redux

In this section we'll show how you we can modify the form app we've built up so that it can work within a larger app using Redux.

Chronologically we haven't talked about Redux in this book. The next two chapters are all about Redux in depth. If you're unfamiliar with Redux, hop over to those chapters and come back here when you need to deal with forms in Redux.

Our form, which used to be our entire app, will now become a component. In addition, we'll adapt it to fit within the Redux paradigm. At a high level, this involves moving state and functionality from our form component to Redux reducers and actions. For example, we will no longer call API functions from within the form component – we use Redux async actions for that instead. Similarly, data that used to be held as `state` in our form will become read-only `props` – it will now be held in the Redux store.

When building with Redux, it is very helpful to start by thinking about the "shape" your state will take. In our case, we have a pretty good idea already since our functionality has been built. When using Redux, you'll want to centralize state as much as possible – this will be the `store`, accessible by all components in the app. Here's what our `initialState` should look like:

forms/src/11-redux-reducer.js

```
const initialState = {
  people: [],
  isLoading: false,
  saveStatus: 'READY',
  person: {
    name: '',
    email: '',
    course: null,
    department: null
  },
};
```

No surprises here. Our app cares about the list of people who have signed up, the current person being typed in the form, whether or not we're loading, and the status of our save attempt.

Now that we know the shape of our state, we can think of different actions that would mutate it. For example, since we're keeping track of the list of people, we can imagine one action to retrieve the list from the server when the app starts. This action would affect multiple properties of our state. When the request to the server returns with the list, we'll want to update our state with it, and we'll also want to update `isLoading`. In fact, we'll want to set `isLoading` to `true` when we *start* the request, and we'll want to set it to `false` when the request finishes. With Redux, it's important to realize that we can often split one objective into multiple actions.

For our Redux app, we'll have five action types. The first two are related to the objective just mentioned, they are `FETCH_PEOPLE_REQUEST` and `FETCH_PEOPLE_SUCCESS`. Here are those action types with their corresponding action creator functions:

forms/src/11-redux-actions.js

```
/* eslint-disable no-use-before-define */
export const FETCH_PEOPLE_REQUEST = 'FETCH_PEOPLE_REQUEST';
function fetchPeopleRequest () {
  return {type: FETCH_PEOPLE_REQUEST};
}

export const FETCH_PEOPLE_SUCCESS = 'FETCH_PEOPLE_SUCCESS';
function fetchPeopleSuccess (people) {
  return {type: FETCH_PEOPLE_SUCCESS, people};
}
```

When we start the request we don't need to provide any information beyond the action type to the reducer. The reducer will know that the request started just from the type and can update `isLoading`

to `true`. When the request is successful, the reducer will know to set it to `false`, but we'll need to provide the people list for that update. This is why `people` is on the second action, `FETCH_PEOPLE_SUCCESS`.

We skip `FETCH_PEOPLE_FAILURE` only for expediency, but you'll want to handle fetch failures in your own app. See below for how to do that for saving the list.

We can now imagine dispatching these actions and having our state updated appropriately. To get the people list from the server we would dispatch the `FETCH_PEOPLE_REQUEST` action, use our API client to get the list, and finally dispatch the `FETCH_PEOPLE_SUCCESS` action (with the people list on it). With Redux, we'll use an asynchronous action creator, `fetchPeople()` to perform those actions:

forms/src/11-redux-actions.js

```
export function fetchPeople () {
  return function (dispatch) {
    dispatch(fetchPeopleRequest())
    apiClient.loadPeople().then((people) => {
      dispatch(fetchPeopleSuccess(people))
    })
  }
}
```

Instead of returning an action object, asynchronous action creators return functions that dispatch actions.

Asynchronous action creators are not supported by default with Redux. To be able to dispatch functions instead of action objects, we'll need to use the `redux-thunk` middleware when we create our store.

We'll also want to create actions for *saving* our list to the server. Here's what they look like:

forms/src/11-redux-actions.js

```
export const SAVE_PEOPLE_REQUEST = 'SAVE_PEOPLE_REQUEST';
function savePeopleRequest () {
  return {type: SAVE_PEOPLE_REQUEST};
}

export const SAVE_PEOPLE_FAILURE = 'SAVE_PEOPLE_FAILURE';
function savePeopleFailure (error) {
  return {type: SAVE_PEOPLE_FAILURE, error};
}
```

```
20
21 export const SAVE_PEOPLE_SUCCESS = 'SAVE_PEOPLE_SUCCESS';
22 function savePeopleSuccess (people) {
23   return {type: SAVE_PEOPLE_SUCCESS, people};
```

Just like the fetch we have `SAVE_PEOPLE_REQUEST` and `SAVE_PEOPLE_SUCCESS`, but we also have `SAVE_PEOPLE_FAILURE`. The `SAVE_PEOPLE_REQUEST` action happens when we start the request, and like before we don't need to provide any data besides the action type. The reducer will see this type and know to update `saveStatus` to `'SAVING'`. Once the request resolves, we can trigger either `SAVE_PEOPLE_SUCCESS` or `SAVE_PEOPLE_FAILURE` depending on the outcome. We will want to pass additional data with these though – `people` on a successful save and `error` on a failure.

Here's how we use those together within an asynchronous action creator, `savePeople()`:

forms/src/11-redux-actions.js

```
36 export function savePeople (people) {
37   return function (dispatch) {
38     dispatch(savePeopleRequest())
39     apiClient.savePeople(people)
40       .then((resp) => { dispatch(savePeopleSuccess(people)) })
41       .catch((err) => { dispatch(savePeopleFailure(err)) })
42   }
43 }
```

Notice that this action creator delegates the 'work' of making the API request to our API client. We can define our API client like this:

forms/src/11-redux-actions.js

```
45 const apiClient = {
46   loadPeople: function () {
47     return {
48       then: function (cb) {
49         setTimeout( () => {
50           cb(JSON.parse(localStorage.people || '[]'))
51         }, 1000);
52       }
53     }
54   },
55
56   savePeople: function (people) {
57     const success = !!(this.count++ % 2);
```



```
58
59     return new Promise(function (resolve, reject) {
60       setTimeout( () => {
61         if (!success) return reject({success});
62
63         localStorage.people = JSON.stringify(people);
64         resolve({success});
65       }, 1000);
66     })
67   },
68
69   count: 1
70 }
```

Now that we've defined all of our action creators, we have everything we need for our reducer. By using the two asynchronous action creators above, the reducer can make all the updates to our state that our app will need. Here's what our reducer looks like:

forms/src/11-redux-reducer.js

```
6  const initialState = {
7    people: [],
8    isLoading: false,
9    saveStatus: 'READY',
10   person: {
11     name: '',
12     email: '',
13     course: null,
14     department: null
15   },
16 };
17
18 export function reducer (state = initialState, action) {
19   switch (action.type) {
20     case FETCH_PEOPLE_REQUEST:
21       return Object.assign({}, state, {
22         isLoading: true
23       });
24     case FETCH_PEOPLE_SUCCESS:
25       return Object.assign({}, state, {
26         people: action.people,
27         isLoading: false
28       });
```

```
    case SAVE_PEOPLE_REQUEST:
      return Object.assign({}, state, {
        saveStatus: 'SAVING'
      });
    case SAVE_PEOPLE_FAILURE:
      return Object.assign({}, state, {
        saveStatus: 'ERROR'
      });
    case SAVE_PEOPLE_SUCCESS:
      return Object.assign({}, state, {
        people: action.people,
        person: {
          name: '',
          email: '',
          course: null,
          department: null
        },
        saveStatus: 'SUCCESS'
      });
    default:
      return state;
  }
}
```

By just looking at the actions and the reducer you should be able to see all the ways our state can be updated. This is one of the great things about Redux. Because everything is so explicit, state becomes very easy to reason about and test.

Now that we've established the shape of our state and how it can change, we'll create a store. Then we'll want to make some changes so that our form can connect to it properly.

Form Component

Now that we've created the foundation of our app's data architecture with Redux, we can adapt our form component to fit in. In broad strokes, we need to remove any interaction with the API client (our asynchronous action creators handle this now) and shift dependence from component-level `state` to `props` (Redux state will be passed in as `props`).

The first thing we need to do is set up `propTypes` that will align with the data we expect to get from Redux:

forms/src/11-redux-form.js

```
  static propTypes = {
    people: PropTypes.array.isRequired,
    isLoading: PropTypes.bool.isRequired,
    saveStatus: PropTypes.string.isRequired,
    fields: PropTypes.object,
    onSubmit: PropTypes.func.isRequired,
  };
```

We will require one additional `prop` that is not related to data in our Redux store, `onSubmit()`. When the user submits a new person, instead of using the API client, our form component will call this function instead. Later we'll show how we hook this up to our asynchronous action creator `savePeople()`.

Next, we limit the amount of data that we'll keep in `state`. We keep `fields` and `fieldErrors`, but we remove `people`, `_loading`, and `_saveStatus` – those will come in on `props`. Here's the updated `state`

forms/src/11-redux-form.js

```
  state = {
    fields: this.props.fields || {
      name: '',
      email: '',
      course: null,
      department: null
    },
    fieldErrors: {},
  };
```

`state.fields` will be initialized to `props.fields` (or the default fields, if not provided). Additionally, if a new `fields` object comes in on `props`, we will update our `state`:

forms/src/11-redux-form.js

```
  componentWillReceiveProps(update) {
    console.log('this.props.fields', this.props.fields, update);

    this.setState({ fields: update.fields });
  }
```

Now that our `props` and `state` are in order, we can remove any usage of `apiClient` since that will be handled by our asynchronous action creators. The two places that we used the API client were in `componentWillMount()` and `onFormSubmit()`.

Since the only purpose of `componentWillMount()` was to use the API client, we have removed it entirely. In `onFormSubmit()`, we remove the block related to the API and replace it with a call to `props.onSubmit()`:

forms/src/11-redux-form.js

```
35  onFormSubmit = (evt) => {
36    const person = this.state.fields;
37
38    evt.preventDefault();
39
40    if (this.validate()) return;
41
42    this.props.onSubmit([ ...this.props.people, person ]);
43  };
```

With all of that out of the way, we can make a few minor updates to `render()`. In fact, the only modifications we have to make to `render()` are to replace references to `state._loading`, `state._saveStatus`, and `state.people` with their counterparts on `props`.

forms/src/11-redux-form.js

```
69  render() {
70    if (this.props.isLoading) {
71      return <img alt='loading' src='/img/loading.gif' />;
72    }
73
74    const dirty = Object.keys(this.state.fields).length;
75    let status = this.props.saveStatus;
76    if (status === 'SUCCESS' && dirty) status = 'READY';
77
78    return (
79      <div>
80        <h1>Sign Up Sheet</h1>
81
82        <form onSubmit={this.onFormSubmit}>
83
84          <Field
85            placeholder='Name'
86            name='name'
```



```
87            value={this.state.fields.name}
88            onChange={this.onInputChange}
89            validate={(val) => (val ? false : 'Name Required')}
90          />
91
92          <br />
93
94          <Field
95            placeholder='Email'
96            name='email'
97            value={this.state.fields.email}
98            onChange={this.onInputChange}
99            validate={(val) => (isEmail(val) ? false : 'Invalid Email')}
100         />
101
102         <br />
103
104         <CourseSelect
105           department={this.state.fields.department}
106           course={this.state.fields.course}
107           onChange={this.onInputChange}
108         />
109
110         <br />
111
112         {{
113           SAVING: <input value='Saving...' type='submit' disabled />,
114           SUCCESS: <input value='Saved!' type='submit' disabled />,
115           ERROR: <input
116             value='Save Failed - Retry?'
117             type='submit'
118             disabled={this.validate()}
119           />,
120           READY: <input
121             value='Submit'
122             type='submit'
123             disabled={this.validate()}
124           />,
125         }[status]}
126
127       </form>
128
```

```
        <div>
          <h3>People</h3>
          <ul>
            {this.props.people.map(({ name, email, department, course }, i) =>
              <li key={i}>{[ name, email, department, course ].join(' - ')}</li>
            ) }
          </ul>
        </div>
      </div>
    );
  }
```

You may notice that we handle `saveStatus` a bit differently. In the previous iteration, our form component was able to control `state._saveStatus` and could set it to `'READY'` on a field change. In this version, we get that information from `props.saveStatus` and it is read-only. The solution is to check if `state.fields` has any keys – if it does, we know the user has entered data and we can set the button back to the "ready" state.

Connect the Store

At this point we have our actions, our reducer, and our streamlined form component. All that's left is to connect them together.

First, we will use Redux's `createStore()` method to create a `store` from our `reducer`. Because we want to be able to dispatch asynchronous actions, we will also use `thunkMiddleware` from the `redux-thunk` module. To use middleware in our `store`, we'll use Redux's `applyMiddleware()` method. Here's what that looks like:

forms/src/11-redux-app.js

```
const store = createStore(reducer, applyMiddleware(thunkMiddleware));
```

Next, we will use the `connect()` method from `react-redux` to optimize our form component for use with Redux. We do this by providing it two methods: `mapStateToProps` and `mapDispatchToProps`.

When using Redux, we want our components to subscribe to the `store`. However, with `react-redux` it will do that for us. All we need to do is provide a `mapStateToProps` function that defines the mapping between data in the `store` and `props` for the component. In our app, they line up neatly:

forms/src/11-redux-app.js

```
function mapStateToProps(state) {
  return {
    isLoading: state.isLoading,
    fields: state.person,
    people: state.people,
    saveStatus: state.saveStatus,
  };
}
```

From within our form component, we call `props.onSubmit()` when the user submits and validation passes. We want this behavior to dispatch our `savePeople()` asynchronous action creator. To do this, we provide `mapDispatchToProps()` to define the connection between the `props.onSubmit()` function and the dispatch of our action creator:

forms/src/11-redux-app.js

```
function mapDispatchToProps(dispatch) {
  return {
    onSubmit: (people) => {
      dispatch(savePeople(people));
    },
  };
}
```

With both of those functions created, we use the `connect()` method from `react-redux` to give us an optimized `ReduxForm` component:

forms/src/11-redux-app.js

```
const ReduxForm = connect(mapStateToProps, mapDispatchToProps)(Form);
```

The final step is to incorporate the `store` and the `ReduxForm` into our app. At this point our app is a very simple component with only two methods, `componentWillMount()` and `render()`.

In `componentWillMount()` we dispatch our `fetchPeople()` asynchronous action to load the people list from the server:

forms/src/11-redux-app.js

```
17   componentWillMount() {
18     store.dispatch(fetchPeople());
19   }
```

In `render()` we use a helpful component `Provider` that we get from `react-redux`. `Provider` will make the store available to all of its child components. We simply place `ReduxForm` as a child of `Provider` and our app is good to go:

forms/src/11-redux-app.js

```
21   render() {
22     return (
23       <Provider store={store}>
24         <ReduxForm />
25       </Provider>
26     );
27   }
```

And that's it! Our form now fits neatly inside a Redux-based data architecture.

After reading this chapter, you should have a good handle on the fundamentals of forms in React. That said, if you'd like to outsource some portion of your form handling to an external module, there are several available. Read on for a list of some of the more popular options.

Form Modules

formsy-react

https://github.com/christianalfoni/formsy-react[60]

`formsy-react` tries to strike a balance between flexibility and reusability. This is a worthwhile goal as the author of this module acknowledges that forms, inputs, and validation are handled quite differently across projects.

The general pattern is that you use the `Formsy.Form` component as your form element, and provide your own input components as children (using the `Formsy.Mixin`). The `Formsy.Form` component has handlers like `onValidSubmit()` and `onInvalid()` that you can use to alter state on the form's parent, and the mixin provides some validation and other general purpose helpers.

react-input-enhancements

http://alexkuz.github.io/react-input-enhancements[61]

[60] https://github.com/christianalfoni/formsy-react
[61] http://alexkuz.github.io/react-input-enhancements

`react-input-enhancements` is a collection of five rich components that you can use to augment forms. This module has a nice demo to showcase how you can use the `Autosize`, `Autocomplete`, `Dropdown`, `Mask`, and `DatePicker` components. The author does make a note that they aren't quite ready for production and are more conceptual. That said, they might be useful if you're looking for a drop-in datepicker or autocomplete element.

tcomb-form

http://gcanti.github.io/tcomb-form[62]

`tcomb-form` is meant to be used with `tcomb` models (https://github.com/gcanti/tcomb[63]) which center around Domain Driven Design. The idea is that once you create a model, the corresponding form can be automatically generated. In theory, the benefits are that you don't have to write as much markup, you get usability and accessibility for free (e.g. automatic labels and inline validation), and your forms will automatically stay in sync with changes to your model. If `tcomb` models seem to be a good fit for your app, this `tcomb-form` is worth considering.

winterfell

https://github.com/andrewhathaway/winterfell[64]

If the idea of defining your forms and fields entirely with JSON, `winterfell` might be for you. With `winterfell`, you sketch out your entire form in a JSON schema. This schema is a large object where you can define things like CSS class names, section headers, labels, validation requirements, field types, and conditional branching. `winterfell` is organized into "form panels", "question panels", and "question sets". Each panel has an ID and that ID is used to assign sets to it. One benefit of this approach is that if you find yourself creating/modifying lots of forms, you could create a UI to create/modify these schema objects and persist them to a database.

react-redux-form

https://github.com/davidkpiano/react-redux-form[65]

If Redux is more your style `react-redux-form` is a "collection of action creators and reducer creators" to simplify "building complex and custom forms with React and Redux". In practice, this module provides a `modelReducer` and a `formReducer` helper to use when creating your Redux `store`. Then within your form you can use the provided `Form`, `Field`, and `Error` components to help connect your `label` and `input` elements to the appropriate reducers, set validation requirements, and display appropriate errors. In short, this is a nice thin wrapper to help you build forms using Redux.

[62] http://gcanti.github.io/tcomb-form
[63] https://github.com/gcanti/tcomb
[64] https://github.com/andrewhathaway/winterfell
[65] https://github.com/davidkpiano/react-redux-form

Using Webpack with Create React App

In most of our earlier projects, we loaded React with `script` tags in our apps' `index.html` files:

```
<script src='vendor/react.js'></script>
<script src='vendor/react-dom.js'></script>
```

Because we've been using ES6, we've also been loading the Babel library with `script` tags:

```
<script src='vendor/babel-standalone.js'></script>
```

With this setup we've been able to load in any ES6 JavaScript file we wanted in `index.html`, specifying that its type is `text/babel`:

```
<script type='text/babel' src='./client.js'></script>
```

Babel would handle the loading of the file, transpiling our ES6 JavaScript to browser-ready ES5 JavaScript.

If you need a refresher on our setup strategy so far, we detail it in Chapter 1.

We began with this setup strategy because it's the simplest. You can begin writing React components in ES6 with little setup.

However, this approach has limitations. For our purposes, the most pressing limitation is the lack of support for **JavaScript modules**.

JavaScript modules

We saw modules in earlier apps. For instance, the time tracking app had a Client module. That module's file defined a few functions, like `getTimers()`. It then set `window.client` to an object that "exposed" each function as a property. That object looked like this:


```
// `window.client` was set to this object
// Each property is a function
{
  getTimers,
  createTimer,
  updateTimer,
  startTimer,
  stopTimer,
  deleteTimer,
};
```

This Client module only exposed these functions. These are the Client module's **public methods**. The file `public/js/client.js` also contained other function definitions, like `checkStatus()`, which verifies that the server returned a 2xx response code. While each of the public methods uses `checkStatus()` internally, `checkStatus()` is kept **private**. It is only accessible from within the module.

That's the idea behind a module in software. You have some self-contained component of a software system that is responsible for some discrete functionality. The module exposes a limited interface to the rest of the system, ideally the minimum viable interface the rest of the system needs to effectively use the module.

In React, **we can think of each of our individual components as their own modules**. Each component is responsible for some discrete part of our interface. React components might contain their own state or perform complex operations, but the interface for all of them is the same: they accept inputs (props) and output their DOM representation (`render`). Users of a React component need not know any of the internal details.

In order for our React components to be truly modular, we'd ideally have them live in their own files. In the upper scope of that file, the component might define a styles object or helper functions that only the component uses. But we want our component-module to only expose the component itself.

Until ES6, modules were not natively supported in JavaScript. Developers would use a variety of different techniques to make modular JavaScript. Some solutions only work in the browser, relying on the browser environment (like the presence of `window`). Others only work in Node.js.

Browsers don't yet support ES6 modules. But ES6 modules are the future. The syntax is intuitive, we avoid bizarre tactics employed in ES5, and they work both in and outside of the browser. Because of this, the React community has quickly adopted ES6 modules.

If you look at `time_tracking_app/public/js/client.js`, you'll get an idea of how strange the techniques for creating ES5 JavaScript modules are.

However, due to the complexity of module systems, we can't simply use ES6's import/export syntax and expect it to "just work" in the browser, even with Babel. More tooling is needed.

For this reason and more, the JavaScript community has widely adopted **JavaScript bundlers**. As we'll see, JavaScript bundlers allow us to write modular ES6 JavaScript that works seamlessly in the browser. But that's not all. Bundlers pack numerous advantages. Bundlers provide a strategy for both organizing and distributing web apps. They have powerful toolchains for both iterating in development and producing production-optimized builds.

While there are several options for JavaScript bundlers, the React community's favorite is **Webpack**.

However, bundlers like Webpack come with a significant trade-off: They add complexity to the setup of your web application. Initial configuration can be difficult and you ultimately end up with an app that has more moving pieces.

In response to setup and configuration woes, the community has created loads of boilerplates and libraries developers can use to get started with more advanced React apps. But the React core team recognized that as long as there wasn't a core team sanctioned solution, the community was likely to remain splintered. The first steps for a bundler-powered React setup can be confusing for novice and experienced developers alike.

The React core team responded by producing the **Create React App** project.

Create React App

The create-react-app[66] library provides a command you can use to initiate a new **Webpack-powered React app**:

```
$ create-react-app my-app-name
```

The library will configure a "black box" Webpack setup for you. It provides you with the benefits of a Webpack setup while abstracting away the configuration details.

Create React App is a great way to get started with a Webpack-React app using standard conventions. Therefore, we'll use it in all of our forthcoming Webpack-React apps.

In this chapter, we'll:

- See what a React component looks like when represented as an ES6 module
- Examine the setup of an app managed by Create React App
- Take a close look at how Webpack works
- Explore some of the numerous advantages that Webpack provides for both development and production use
- Peek under the hood of Create React App
- Figure out how to get a Webpack-React app to work alongside an API

[66] https://github.com/facebookincubator/create-react-app

The idea of a "black box" controlling the inner-workings of your app might be scary. This is a valid concern. Later in the chapter, we'll explore a feature of Create React App, `eject`, which should hopefully assuage some of this fear.

Exploring Create React App

Let's install Create React App and then use it to initialize a Webpack-React app. We can install it globally from the command line using the `-g` flag. You can run this command anywhere on your system:

```
$ npm i -g create-react-app@0.5.0
```

The `@0.5.0` above is used to specify a version number and is important.

create-react-app is a very new project. It is moving quickly in its early stages. To avoid possible discrepancies, be sure to use the same version we have specified here.

Now, anywhere on your system, you can run the `create-react-app` command to initiate the setup for a new Webpack-powered React app.

Let's create a new app. We'll do this inside of the code download that came with the book. From the root of the code folder, change into the directory for this chapter:

```
$ cd webpack
```

That directory already has three folders:

```
$ ls
es6-modules/
food-lookup/
heart-webpack-complete/
```

The completed version of the code for this next section is available in `heart-webpack-complete`.

Run the following command to initiate a new React app in a folder called `heart-webpack`:

```
$ create-react-app heart-webpack
```

This will create the boilerplate for the new app and install the app's dependencies. This might take a while.

When Create React App is finished, `cd` into the new directory:

```
$ cd heart-webpack
```

Before exploring, we want to ensure your `react-scripts` version is the same as the one we're using here in the book. We'll see what the `react-scripts` package is in a moment. Run this command inside `heart-webpack` to lock in version `0.7.0`:

```
npm install --save-dev --save-exact react-scripts@0.7.0
```

Now let's take a look at what's inside:

```
$ ls
README.md
node_modules/
package.json
public/
src/
```

Inside `src/` is a sample React app that Create React App has provided for demonstration purposes. Inside of `public/` is an `index.html`, which we'll look at first.

public/index.html

Opening `public/index.html` in a text editor:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">
    <!--
      ... comment omitted ...
    -->
    <title>React App</title>
  </head>
  <body>
    <div id="root"></div>
    <!--
      ... comment omitted ...
    -->
  </body>
</html>
```

The stark difference from the `index.html` we've used in previous apps: there are no `script` tags here. That means this file is not loading any external JavaScript files. We'll see why this is soon.

package.json

Looking inside of the project's `package.json`, we see a few dependencies and some script definitions:

webpack/heart-webpack-complete/package.json

```
{
  "name": "heart-webpack",
  "version": "0.1.0",
  "private": true,
  "devDependencies": {
    "react-scripts": "0.9.5"
  },
  "dependencies": {
    "react": "15.5.4",
    "react-dom": "15.5.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}
```

Let's break it down.

react-scripts

`package.json` specifies a single development dependency, `react-scripts`:

webpack/heart-webpack-complete/package.json

```
  "devDependencies": {
    "react-scripts": "0.9.5"
  },
```

Create React App is just a boilerplate generator. That command produced the folder structure of our new React app, inserted a sample app, and specified our `package.json`. It's actually the `react-scripts` package that makes everything work.

`react-scripts` specifies all of our app's development dependencies, like Webpack and Babel. Furthermore, it contains scripts that "glue" all of these dependencies together in a conventional manner.

Create React App is just a boilerplate generator. The `react-scripts` package, specified in `package.json`, is the engine that will make everything work.

Even though `react-scripts` is the engine, throughout the chapter we'll continue to refer to the overall *project* as Create React App.

`react` and `react-dom`

Under `dependencies`, we see `react` and `react-dom` listed:

webpack/heart-webpack-complete/package.json

```
  "dependencies": {
    "react": "15.5.4",
    "react-dom": "15.5.4"
  },
```

In our first two projects, we loaded in `react` and `react-dom` via script tags in `index.html`. As we saw, those libraries were not specified in this project's `index.html`.

Webpack gives us the ability to use npm packages in the browser. We can specify external libraries that we'd like to use in `package.json`. This is incredibly helpful. Not only do we now have easy access to a vast library of packages. We also get to use npm to manage **all** the libraries that our app uses. We'll see in a bit how this all works.

Scripts

`package.json` specifies four commands under `scripts`. Each executes a command with `react-scripts`. Over this chapter and the next we'll cover each of these commands in depth, but at a high-level:

- `start`: Boots the Webpack development HTTP server. This server will handle requests from our web browser.
- `build`: For use in production, this command creates an optimized, static bundle of all our assets.
- `test`: Executes the app's test suite, if present.
- `eject`: Moves the innards of `react-scripts` into your project's directory. This enables you to abandon the configuration that `react-scripts` provides, tweaking the configuration to your liking.

For those weary of the black box that `react-scripts` provides, that last command is comforting. You have an escape hatch should your project "outgrow" `react-scripts` or should you need some special configuration.

In a `package.json`, you can specify which packages are necessary in which environment. Note that `react-scripts` is specified under `devDependencies`.

When you run `npm i`, npm will check the environment variable `NODE_ENV` to see if it's installing packages in a production environment. In production, npm only installs packages listed under `dependencies` (in our case, `react` and `react-dom`). In development, npm installs all packages. This speeds the process in production, foregoing the installation of unneeded packages like linters or testing libraries.

Given this, you might wonder: Why is `react-scripts` listed as a development dependency? How will the app work in a production environment without it? We'll see why this is after taking a look at how Webpack prepares production builds.

`src/`

Inside of `src/`, we see some JavaScript files:

```
$ ls src
App.css
App.js
App.test.js
index.css
index.js
logo.svg
```

Create React App has created a boilerplate React app to demonstrate how files can be organized. This app has a single component, `App`, which lives inside `App.js`.

`App.js`

Looking inside `src/App.js`:

webpack/heart-webpack-complete/src/App.js

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <div className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h2>Welcome to React</h2>
        </div>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to reload.
        </p>
      </div>
    );
  }
}

export default App;
```

There are a few noteworthy features here.

The import statements

We import `React` and `Component` at the top of the file:

webpack/heart-webpack-complete/src/App.js

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

This is the ES6 module import syntax. Webpack will infer that by `'react'` we are referring to the npm package specified in our `package.json`.

If ES6 modules are new to you, check out the entry in Appendix B."

The next two imports may have surprised you:

webpack/heart-webpack-complete/src/App.js

```
import logo from './logo.svg';
import './App.css';
```

We're using `import` on files that aren't JavaScript! Webpack has you specify *all* your dependencies using this syntax. We'll see later how this comes into play. Because the paths are relative (they are preceded with `./`), Webpack knows we're referring to local files and not npm packages.

`App` is an ES6 module

The `App` component itself is simple and does not employ state or props. Its return method is just markup, which we'll see rendered in a moment.

What's special about the `App` component is that it's an ES6 module. Our `App` component lives inside its own dedicated `App.js`. At the top of this file, it specifies its dependencies and at the bottom it specifies its export:

webpack/heart-webpack-complete/src/App.js

```
export default App;
```

Our React component is entirely self-contained in this module. Any additional libraries, styles, and images could be specified at the top. Any developer could open this file and quickly reason what dependencies this component has. We could define helper functions that are private to the component, inaccessible to the outside.

Furthermore, recall that there is another file in `src/` related to `App` besides `App.css`: `App.test.js`. So, we have three files corresponding to our component: The component itself (an ES6 module), a dedicated stylesheet, and a dedicated test file.

Create React App has suggested a powerful organization paradigm for our React app. While perhaps not obvious in our single-component app, you can imagine how this modular component model is intended to scale well as the number of components grows to the hundreds or thousands.

We know where our modular component is defined. But we're missing a critical piece: Where is the component written to the DOM?

The answer lies inside `src/index.js`.

`index.js`

Open `src/index.js` now:

webpack/heart-webpack-complete/src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import './index.css';

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

Stepping through this file, we first import both `react` and `react-dom`. Because we specified `App` as the default export from `App.js`, we can import it here. Note again that the relative path (`./App`) signals to Webpack that we're referring to a local file, not an npm package.

At this point, we can use our `App` component just as we have in the past. We make a call to `ReactDOM.render()`, rendering the component to the DOM on the `root` div. This div tag is the one and only div present in `index.html`.

This layout is certainly more complicated than the one we used in our first couple of projects. Instead of just rendering `App` right below where we define it, we have another file that we're importing `App` into and making the `ReactDOM.render()` call in. Again, this setup is intended to keep our code modular. `App.js` is restricted to only defining a React component. It does not carry the additional responsibility of rendering that component. Following from this pattern, we could comfortably import and render this component anywhere in our app.

We now know where the `ReactDOM.render()` call is located. But the way this new setup works is still opaque. `index.html` does not appear to load in any JavaScript. How do our JavaScript modules make it to the browser?

Let's boot the app and then explore how everything fits together.

Why do we import `React` at the top of the file? It doesn't apparently get referenced anywhere.

`React` actually *is* referenced later in the file, we just can't see it because of a layer of indirection. We're referencing `App` using JSX. So this line in JSX:

```
<App />
```

Is actually this underneath the JSX abstraction:

```
React.createElement(App, null);
```

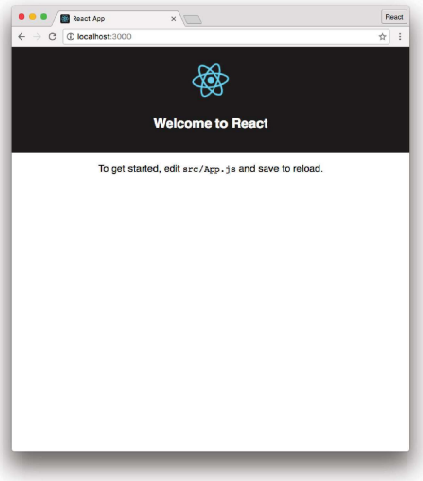

Booting the app

From the root of `heart-webpack`, run the start command:

```
$ npm start
```

This boots the **Webpack development server**. We dig into the details of this server momentarily.

Visiting `http://localhost:3000/`, we see the interface for the sample app that Create React App has provided:



The sample app

The `App` component is clearly present on the page. We see both the logo and text that the component specifies. How did it get there?

Let's view the source code behind this page. In both Chrome and Firefox, you can type `view-source:http://localhost:3000/` into the address bar to do so:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="shortcut icon" href="/favicon.ico">
    <!--
      ... comment omitted ...
    -->
    <title>React App</title>
  </head>
  <body>
    <div id="root"></div>
    <!--
      ... comment omitted ...
    -->
  <script type="text/javascript" src="/static/js/bundle.js"></script></body>
</html>
```

This `index.html` looks the same as the one we looked at earlier, save for one key difference: **There is a `script` tag appended to the bottom of the `body`**. This script tag references a `bundle.js`. As we'll see, the `App` component from `App.js` and the `ReactDOM.render()` call from `index.js` both live inside of that file.

The Webpack development server inserted this line into our `index.html`. To understand what `bundle.js` is, let's dig into how Webpack works.

This script defaults the server's port to `3000`. However, if it detects that `3000` is occupied, it will choose another. The script will tell you where the server is running, so check the console if it appears that it is not on `http://localhost:3000/`.

If you're running OS X, this script will automatically open a browser window pointing to `http://localhost:3000/`.

Webpack basics

In our first app (the voting app), we used the library `http-server` to serve our static assets, like `index.html`, our JavaScript files, and our images.

In the second app (the timers app), we used a small Node server to serve our static assets. We defined a server in `server.js` which both provided a set of API endpoints and served all assets under `public/`. Our API server and our static asset server were one in the same.

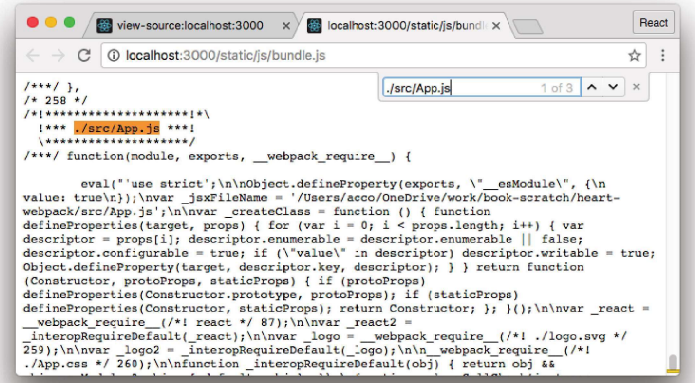
With Create React App, our static assets are served by the **Webpack development server** that is booted when we run `npm start`. At the moment, we're not working with an API.

As we saw, the original `index.html` did not contain any references to the React app. Webpack inserted a reference to `bundle.js` in `index.html` before serving it to our browser. If you look around on disk, `bundle.js` doesn't exist anywhere. **The Webpack development server produces this file on the fly and keeps it in memory**. When our browser makes a request to `localhost:3000/`, Webpack is serving its modified version of `index.html` and `bundle.js` from memory.

From the page `view-source:http://localhost:3000/`, you can click on `/static/js/bundle.js` to open that file in your browser. It will probably take a few seconds to open. It's a gigantic file.

`bundle.js` contains *all* the JavaScript code that our app needs to run. Not only does it contain the entire source for `App.js` — it contains the entire source for the React library!

You can search for the string `./src/App.js` in this file. Webpack demarcates each separate file it has included with a special comment. What you'll find is quite messy:



If you do a little hunting, you can see recognizable bits and pieces of `App.js` amidst the chaos. This is indeed our component. But it looks nothing like it.

Webpack has performed some transformation on all the included JavaScript. Notably, it used Babel to transpile our ES6 code to an ES5-compatible format.

If you look at the comment header for `App.js`, it has a number. In the screenshot above, that number was 258:

```
/* 258 */
/*!**********************!*\
  !*** ./src/App.js ***!
  \**********************/
```

Your module IDs might be different than the ones here in the text.

The module itself is encapsulated inside of a function that looks like this:

```
function(module, exports, __webpack_require__) {
  // The chaotic `App.js` code here
}
```

Each module of our web app is encapsulated inside of a function with this signature. Webpack has given each of our app's modules this function container as well as a module ID (in the case of `App.js`, 258).

But "module" here is not limited to JavaScript modules.

Remember how we imported the logo in `App.js`, like this:

webpack/heart-webpack-complete/src/App.js

```
import logo from './logo.svg';
```

And then in the component's markup it was used to set the `src` on an `img` tag:

webpack/heart-webpack-complete/src/App.js

```
          <img src={logo} className="App-logo" alt="logo" />
```

Here's what the variable declaration of `logo` looks like inside the chaos of the `App.js` Webpack module:

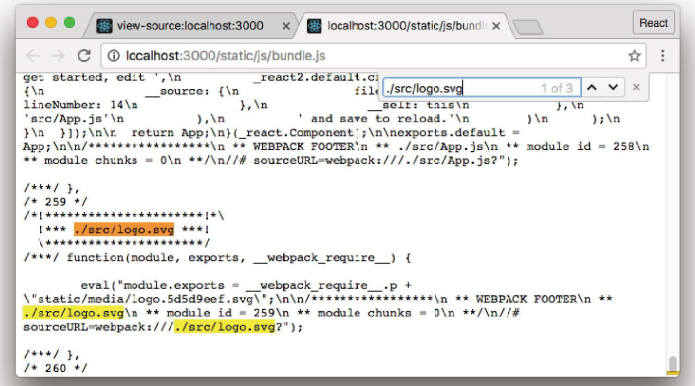

```
var _logo = __webpack_require__(/*! ./logo.svg */ 259);
```

This looks quite strange, mostly due to the in-line comment that Webpack provides for debugging purposes. Removing that comment:

```
var _logo = __webpack_require__(259);
```

Instead of an `import` statement, we have plain old ES5 code. What is this doing though?

To find the answer, search for `./src/logo.svg` in this file. (It should appear directly below `App.js`). The SVG is represented inside `bundle.js`, too!



Looking at the header for this module:

```
/* 259 */
/*!**********************!*\
  !*** ./src/logo.svg ***!
  \**********************/
```

Note that its module ID is 259, the same integer passed to `__webpack_require__()` above.

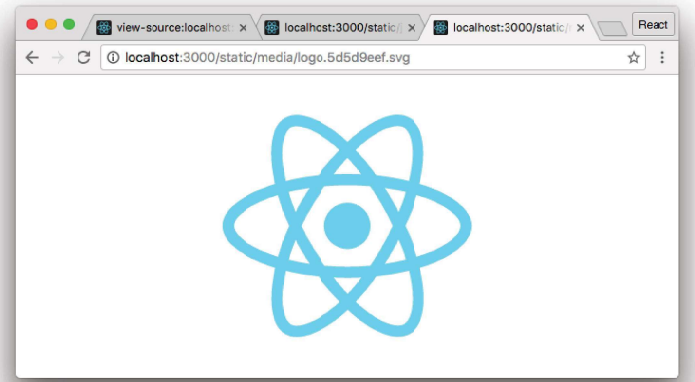
Webpack treats *everything* as a module, including image assets like `logo.svg`. We can get an idea of what's going on by picking out a path in the mess of the `logo.svg` module. Your path might be different, but it will look like this:

```
static/media/logo.5d5d9eef.svg
```

If you open a new browser tab and plug in this address:

```
http://localhost:3000/static/media/logo.5d5d9eef.svg
```

You should get the React logo:



So Webpack created a Webpack module for `logo.svg` by defining a function. While the implementation details of this function are opaque, we know it refers to the path to the SVG on the Webpack development server. Because of this modular paradigm, it was able to intelligently compile a statement like this:

```
import logo from './logo.svg';
```

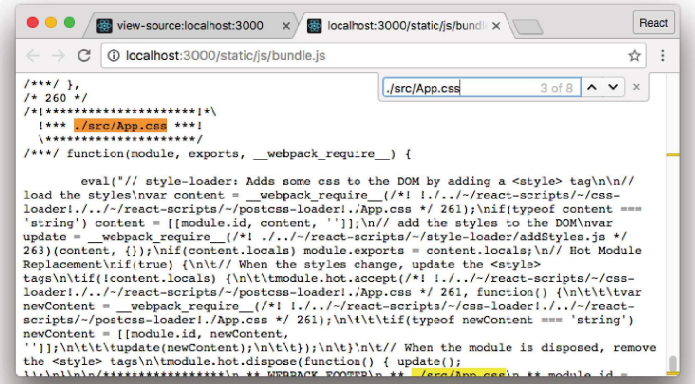
Into this ES5 statement:

```
var _logo = __webpack_require__(259);
```

`__webpack_require__()` is Webpack's special module loader. This call refers to the Webpack module corresponding to `logo.svg`, number 259. That module returns the string path to the logo's location on the Webpack development server, `static/media/logo.5d5d9eef.svg`:


```
var _logo = __webpack_require__(259);
console.log(_logo);
// -> "static/media/logo.5d5d9eef.svg"
```

What about our CSS assets? Yep, *everything* is a module in Webpack. Search for the string `./src/App.css`:



Webpack's `index.html` didn't include any references to CSS. That's because Webpack is including our CSS here via `bundle.js`. When our app loads, this cryptic Webpack module function dumps the contents of `App.css` into `style` tags on the page.

So we know *what* is happening: Webpack has rolled up every conceivable "module" for our app into `bundle.js`. You might be asking: Why?

The first motivation is universal to JavaScript bundlers. Webpack has converted all our ES6 modules into its own bespoke ES5-compatible module syntax.

Furthermore, Webpack, like other bundlers, consolidated all our JavaScript modules into a single file. While it *could* keep JavaScript modules in separate files, having a single file maximizes performance. The initiation and conclusion of each file transfer over HTTP adds overhead. Bundling up hundreds or thousands of smaller files into one bigger one produces a notable speed-up.

Webpack takes this module paradigm further than other bundlers, however. As we saw, it applies the same modular treatment to image assets, CSS, and npm packages (like React and ReactDOM).

This modular paradigm unleashes a lot of power. We touch on aspects of that power throughout the rest of this chapter.

With our nascent understanding of how Webpack works, let's turn our attention back to the sample app. We'll make some modifications and see first-hand how the Webpack development process works.

Making modifications to the sample app

We've been checking out the `bundle.js` produced by the Webpack development server in our browser. Recall that to boot this server, we ran the following command:

```
$ npm start
```

As we saw, this command was defined in `package.json`:

```
"start": "react-scripts start",
```

What exactly is going on here?

The `react-scripts` package defines a start script. Think of this start script as a special interface to Webpack that contains some features and conventions provided by Create React App. At a high-level, the start script:

- Sets up the Webpack configuration
- Provides some nice formatting and coloring for Webpack's console output
- Launches a web browser if you're on OS X

Let's take a look at what the development cycle of our Webpack-powered React app looks like.

Hot reloading

If the server is not already running, go ahead and run the start command to boot it:

```
$ npm start
```

Again, our app launches at `http://localhost:3000/`. The Webpack development server is listening on this port and serves the development bundle when our server makes a request.

One compelling development feature Webpack gives us is **hot reloading**. Hot reloading enables certain files in a web app to be hot-swapped on the fly whenever changes are detected without requiring a full page reload.

At the moment, Create React App only sets up hot reloading for CSS. This is because the React-specific hot reloader is not considered stable enough for the default setup.

Hot reloading for CSS is wonderful. With your browser window open, make an edit to `App.css` and witness as the app updates without a refresh.

For example, you can change the speed at which the logo spins. Here, we changed it from `20s` to `1s`:

```
.App-logo {
  animation: App-logo-spin infinite 1s linear;
  height: 80px;
}
```

Or you can change the color of the header's text. Here, we changed it from `white` to `purple`:

```
.App-header {
  background-color: #222;
  height: 150px;
  padding: 20px;
  color: purple;
}
```

How hot reloading works

Webpack includes client-side code to perform hot reloading inside `bundle.js`. The Webpack client maintains an open socket with the server. Whenever the bundle is modified, the client is notified via this websocket. The client then makes a request to the server, asking for a *patch* to the bundle. Instead of fetching the whole bundle, the server will just send the client the code that client needs to execute to "hot swap" the asset.

Webpack's modular paradigm makes hot reloading of assets possible. Recall that Webpack inserts CSS into the DOM inside `style` tags. To swap out a modified CSS asset, the client removes the previous `style` tags and inserts the new one. The browser renders the modification for the user, all without a page reload.

Auto-reloading

Even though hot reloading is not supported for our JavaScript files, Webpack will still auto-reload the page whenever it detects changes.

With our browser window still open, let's make a minor edit to `src/App.js`. We'll change the text in the `p` tag:

```
<p className="App-intro">
  I just made a change to <code>src/App.js</code>!
</p>
```

Save the file. You'll note the page refreshes shortly after you save and your change is reflected.

Because Webpack is at heart a platform for JavaScript development and deployment, there is an ever-growing ecosystem of plug-ins and tools for Webpack-powered apps.

For development, hot- and auto-reloading are two of the most compelling plug-ins that come configured with Create React App. In the later section on `eject` ("Ejecting"), we'll point to the Create React App configuration file that sets up Webpack for development so that you can see the rest.

For deployment, Create React App has configured Webpack with a variety of plug-ins that produce a production-level optimized build. We'll take a look at the production build process next.

Creating a production build

So far, we've been using the Webpack development server. In our investigation, we saw that this server produces a modified `index.html` which loads a `bundle.js`. Webpack produces and serves this file from memory — nothing is written to disk.

For production, we want Webpack to write a bundle to disk. We'll end up with a production-optimized build of our HTML, CSS, and JavaScript. We could then serve these assets using whatever HTTP server we wanted. To share our app with the world, we'd just need to upload this build to an asset host, like Amazon's S3.

Let's take a look at what a production build looks like.

Quit the server if it's running with `CTRL+C`. From the command line, run the `build` command that we saw in `package.json` earlier:

```
$ npm run build
```

When this finishes, you'll note a new folder is created in the project's root: `build`. `cd` into that directory and see what's inside:

```
$ cd build
$ ls
favicon.ico
index.html
static/
```


If you look at this `index.html`, you'll note that Webpack has performed some additional processing that it did not perform in development. Most notably: there are no newlines. The entire file is on a single line. Newlines are not necessary in HTML and are just extra bytes. We don't need them in production.

Here's what that exact file looks like in a human readable format:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta content="width=device-width,initial-scale=1" name="viewport">
    <link href="/favicon.ico?fd73a6eb" rel="shortcut icon">
    <title>React App</title>
    <link href="/static/css/main.9a0fe4f1.css" rel="stylesheet">
  </head>
  <body>
    <div id="root"></div>
    <script src="/static/js/main.590bf8bb.js" type="text/javascript">
    </script>
  </body>
</html>
```

Instead of referencing a `bundle.js`, this `index.html` references a file in `static/` which we'll look at momentarily. What's more, this production `index.html` now has a `link` tag to a CSS bundle. As we saw, in development Webpack inserts CSS via `bundle.js`. This feature enables hot reloading. In production, hot reloading capability is irrelevant. Therefore, Webpack deploys CSS normally.

Webpack versions assets. We can see above that our JavaScript bundle has a different name and is versioned (`main.<version>.js`).

Asset versioning is useful when dealing with browser caches in production. If a file is changed, the version of that file will be changed as well. Client browsers will be forced to fetch the latest version.

Note that your versions (or digests) for the files above may be different.

The `static/` folder is organized as follows:

```
$ ls static
css/
js/
media/
```

Checking out the folders individually:

```
$ ls static/css
main.9a0fe4f1.css
main.9a0fe4f1.css.map

$ ls static/js
main.f7b2704e.js
main.f7b2704e.js.map

$ ls static/media
logo.5d5d9eef.svg
```

Feel free to open up both the `.css` file and the `.js` file in your text editor. We refrain from printing them here in the book due to their size.

Be careful about opening these files — you might crash your editor due to their size!

If you open the CSS file, you'll see it's just two lines: The first line is *all* of our app's CSS, stripped of all superfluous whitespace. We could have hundreds of different CSS files in our app and they would end up on this single line. The second line is a special comment declaring the location of the map file.

The JavaScript file is even more packed. In development, `bundle.js` has some structure. You can pick apart where the individual modules live. The production build does not have this structure. What's more, our code has been both minified and uglified. If you're unfamiliar with minification or uglification, see the aside "Minification, uglification, and source maps."

Last, the media folder will contain all of our app's other static files, like images and videos. This app only has one image, the React logo SVG file.

Again, this bundle is entirely self-contained and ready to go. If we wanted to, we could install the same `http-server` package that we used in the first application and use it to serve this folder, like this:

```
http-server ./build -p 3000
```


Without the Webpack development server, you can imagine the development cycle would be a bit painful:

1. Modify the app
2. Run `npm run build` to generate the Webpack bundle
3. Boot/restart the HTTP server

This is why there is no way to "build" anything other than a bundle intended for production. The Webpack server services our needs for development.

Minification, uglification, and source maps

For production environments, we can significantly reduce the size of JavaScript files by converting them from a human-readable format to a more compact one that behaves exactly the same. The basic strategy is stripping all excess characters, like spaces. This process is called **minification**.

Uglification (or **obfuscation**) is the process of deliberately modifying JavaScript files so that they are harder for humans to read. Again, the actual behavior of the app is unchanged. Ideally, this process slows down the ability for outside developers to understand your codebase.

Both the `.css` and `.js` files are accompanied by a companion file ending in `.map`. The `.map` file is a **source map** that provides debugging assistance for production builds. Because they've been minified and uglified, the CSS and JavaScript for a production app are difficult to work with. If you encounter a JavaScript bug on production, for example, your browser will direct you to a cryptic line of this obscure code.

Through a source map, you can map this puzzling area of the codebase back to its original, unbuilt form. For more info on source maps and how to use them, see the blog post "Introduction to JavaScript Source Maps[a]."

[a] http://www.html5rocks.com/en/tutorials/developertools/sourcemaps/

Ejecting

When first introducing Create React App at the beginning of this chapter, we noted that the project provided a mechanism for "ejecting" your app.

This is comforting. You might find yourself in a position in the future where you would like further control over your React-Webpack setup. An eject will copy all the scripts and configuration encapsulated in `react-scripts` into your project's directory. It opens up the "black box," handing full control of your app back over to you.

Performing an eject is also a nice way to strip some of the "magic" from Create React App. We'll perform an eject in this section and take a quick look around.

There is no backing out from an eject. Be careful when using this command. Should you decide to eject in the future, make sure your app is checked in to source control.

If you've been adding to the app inside `heart-webpack`, you might consider duplicating that directory before proceeding. For example, you can do this:

```
cp -r heart-webpack heart-webpack-ejected
```

The `node_modules` folder does not behave well when it's moved wholesale like this, so you'll need to remove `node_modules` and re-install:

```
cd heart-webpack-ejected
rm -rf node_modules
npm i
```

You can then perform the steps in this section inside of `heart-webpack-ejected` and preserve `heart-webpack`.

Buckle up

From the root of `heart-webpack`, run the eject command:

```
$ npm run eject
```

Confirm you'd like to eject by typing `y` and hitting enter.

After all the files are copied from `react-scripts` into your directory, `npm install` will run. This is because, as we'll see, all the dependencies for `react-scripts` have been dumped into our `package.json`.

When the npm install is finished, take a look at our project's directory:

```
$ ls
README.md
build/
config/
node_modules/
package.json
public/
scripts/
src/
```


We have two new folders: `config/` and `scripts/`. If you looked inside `src/` you'd note that, as expected, it is unchanged.

Take a look at `package.json`. There are *loads* of dependencies. Some of these dependencies are necessary, like Babel and React. Others — like `eslint` and `whatwg-fetch` — are more "nice-to-haves." This reflects the ethos of the Create React App project: an opinionated starter kit for the React developer.

Check out `scripts/` next:

```
$ ls scripts
build.js
start.js
test.js
```

When we ran `npm start` and `npm run build` earlier, we were executing the scripts `start.js` and `build.js`, respectively. We won't look at these files here in the book, but feel free to peruse them. While complicated, they are well-annotated with comments. Simply reading through the comments can give you a good idea of what each of these scripts are doing (and what they are giving you "for free").

Finally, check out `config/` next:

```
$ ls config
env.js
jest/
paths.js
polyfills.js
webpack.config.dev.js
webpack.config.prod.js
```

`react-scripts` provided sensible defaults for the tools that it provides. In `package.json`, it specifies configuration for Babel. Here, it specifies configuration for Webpack and Jest (the testing library we use in the next chapter).

Of particular noteworthiness are the configuration files for Webpack. Again, we won't dive into those here. But these files are well-commented. Reading through the comments can give you a good idea of what the Webpack development and production pipelines look like and what plug-ins are used. In the future, if you're ever curious about how `react-scripts` has configured Webpack in development or production, you can refer to the comments inside these files.

Hopefully seeing the "guts" of `react-scripts` reduces a bit of its mysticism. Testing out eject as we have here gives you an idea of what the process looks like to abandon `react-scripts` should you need to in the future.

So far in this chapter, we've covered the fundamentals of Webpack and Create React App's interface to it. Specifically, we've seen:

- How the interface for Create React App works
- The general layout for a Webpack-powered React app
- How Webpack works (and some of the power it provides)
- How Create React App and Webpack help us generate production-optimized builds
- What an ejected Create React App project looks like

There's one essential element our demo Webpack-React app is missing, however.

In our second project (the timers app) we had a React app that interfaced with an API. The node server both served our static assets (HTML/CSS/JS) and provided a set of API endpoints that we used to persist data about our running timers.

As we've seen in this chapter, when using Webpack via Create React App we boot a Webpack development server. This server is responsible for serving our static assets.

What if we wanted our React app to interface with an API? We'd still want the Webpack development server to serve our static assets. Therefore, we can imagine we'd boot our API and Webpack servers separately. Our challenge then is getting the two to cooperate.

Using Create React App with an API server

In this section, we'll investigate a strategy for running a Webpack development server alongside an API server. Before digging into this strategy, let's take a look at the app we'll be working with.

The completed app

`food-lookup-complete` is in the root of the book's code download. Getting there from `heart-webpack`:

```
$ cd ../..
$ cd food-lookup-complete
```

Take a look at the folder's structure:


```
$ ls
README.md
client/
db/
node_modules/
package.json
server.js
start-client.js
start-server.js
```

In the root of the project is where the *server* lives. There's a `package.json` here along with a `server.js` file. Inside of `client/` is where the *React app* lives. The `client/` folder was generated with Create React App.

Look inside of `client/` now:

If you're on macOS or Linux run:

```
$ ls -a client
```

Windows users can run:

```
$ ls client
```

And you'll see:

```
.babelrc
.gitignore
node_modules/
package.json
public/
src/
tests/
```

In OSX and Unix, the `-a` flag for the `ls` command displays *all* files, including "hidden" files that are preceded by a `.` like `.babelrc`. Windows displays hidden files by default.

So, we have *two* `package.json` files. One sits in the root and specifies the packages that the server needs. And the other lives in `client/` and specifies the packages that the React app needs. While co-existing in this folder, we have two entirely independent apps.

`.babelrc`

A noteworthy file inside `cilent/` is `.babelrc`. The contents of that file:

```
// client/.babelrc
{
  "plugins": ["transform-class-properties"]
}
```

As you may recall, this plugin gives us the property initializer syntax which we used at the end of the first chapter. In that project, we specified we wanted to use this plugin by setting the `data-plugins` attribute on the script tag for `app.js`, like this:

```
<script
  type="text/babel"
  data-plugins="transform-class-properties"
  src="./js/app.js"
></script>
```

Now, for this project, Babel is being included and managed by `react-scripts`. To specify plugins we'd like Babel to use for our Create React App projects, we must first include the plugin in our `package.json`. It's already included:

food-lookup-complete/client/package.json

```
    "babel-plugin-transform-class-properties": "6.22.0",
```

We then just have to specify we'd like Babel to use the plugin in our `.babelrc`.

Running the app

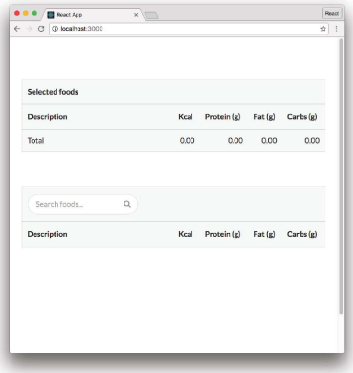
In order to boot our app, we need to install the packages for both the server and client. We'll run `npm i` inside both directories:

```
$ npm i
$ cd client
$ npm i
$ cd ..
```

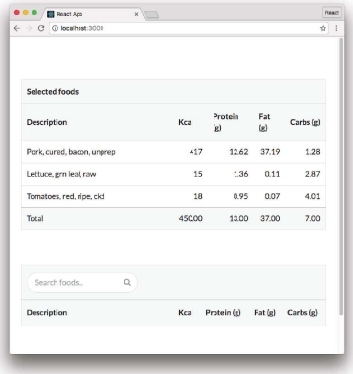
With the packages for both the server and client installed, we can run the app. Be sure to do this from the top-level directory of the project (where the server lives):

```
$ npm start
```


As things are booting, you'll see some console output from both the server and the client. Once the app is booted, you can visit `localhost:3000` to see the app:



The app provides a search field for looking up nutritional info in a food database. Typing a value into the search field performs a live search. You can click on food items to add them to the top table of totals:



The app's components

The app consists of three components:

- **App** (blue): The parent container for the application.
- **SelectedFoods** (yellow): A table that lists selected foods. Clicking on a food item removes it.
- **FoodSearch** (purple): Table that provides a live search field. Clicking on a food item in the table adds it to the total (`SelectedFoods`).

In this chapter, we won't dig into the details of any of these components. Instead, we're just going to focus on how we got this existing Webpack-React app to cooperate with a Node server.

How the app is organized

Now that we've seen the completed app, let's see how we got this to work.

Kill the app if it's running then change to the `food-lookup` directory (the non-complete version) inside `webpack`. Getting there from `food-lookup-complete`:


```
cd webpack/food-lookup
```

Again, we have to install the npm packages for both server and client:

```
$ npm i
$ cd client
$ npm i
$ cd ..
```

The server

Let's boot just the server and see how that works. In the completed version, we used `npm start` to start both the server and the client. If you check `package.json` in this directory, you'll see that this command is not yet defined. We can boot just the server with this:

```
$ npm run server
```

This server provides a single API endpoint, `/api/food`. It expects a single parameter, `q`, the food we are searching for.

You can give it a whirl yourself. You can use your browser to perform a search or use `curl`:

```
$ curl localhost:3001/api/food?q=hash+browns
```


```
[
  {
    "description": "Fast foods, potatoes, hash browns, rnd pieces or patty",
    "kcal": 272,
    "protein_g": 2.58,
    "carbohydrate_g": 28.88,
    "sugar_g": 0.56
  },
  {
    "description": "Chick-fil-a, hash browns",
    "kcal": 301,
    "protein_g": 3,
    "carbohydrate_g": 30.51,
    "sugar_g": 0.54
  },
  {
    "description": "Denny's, hash browns",
    "kcal": 197,
```

```
    "protein_g": 2.49,
    "carbohydrate_g": 26.59,
    "sugar_g": 1.38
  },
  {
    "description": "Restaurant, family style, hash browns",
    "kcal": 197,
    "protein_g": 2.49,
    "carbohydrate_g": 26.59,
    "sugar_g": 1.38
  }
]
```

Now that we understand how this endpoint works, let's take a look at the one area it is called in the client. Kill the server with CTRL+C.

`Client`

The `FoodSearch` component makes the call to `/api/foods`. It performs a request every time the user changes the search field. It uses a library, `Client`, to make the request.

The `Client` module is defined in `client/src/Client.js`. It exports an object with one method, `search()`. Looking just at the `search()` function:

webpack/food-lookup/client/src/Client.js

```
function search(query, cb) {
  return fetch(`http://localhost:3001/api/food?q=${query}`, {
    accept: 'application/json',
  }).then(checkStatus)
    .then(parseJSON)
    .then(cb);
}
```

The `search()` function is the one touch point between the client and the server. `search()` makes a call to `localhost:3001`, the default location of the server.

So, we have two different servers we need to have running in order for our app to work. We need the API server running (at `localhost:3001`) and the Webpack development server running (at `localhost:3000`). If we have both servers running, they should presumably be able to communicate.

We could use two terminal windows, but there's a better solution.

If you need a review on the Fetch API, we use it in Chapter 3: Components and Servers.

Concurrently

Concurrently[67] is a utility for running multiple processes. We'll see how it works by implementing it.

Concurrently is already included in the server's `package.json`:

webpack/food-lookup/package.json

```
  },
  "devDependencies": {
    "concurrently": "3.1.0"
```

We want concurrently to execute two commands, one to boot the API server and one to boot the Webpack development server. You boot multiple commands by passing them to concurrently in quotes like this:

```
# Example of using `concurrently`
$ concurrently "command1" "command2"
```

If you were writing your app to just work on Mac or Unix machines, you could do something like this:

```
$ concurrently "npm run server" "cd client && npm start"
```

Note the second command for booting the client changes into the `client` directory and then runs `npm start`.

However, the `&&` operator is not cross-platform and won't work on Windows. As such, we've included a `start-client.js` script with the project. This script will boot the client from the top-level directory.

Given this start script, you can boot the client app from the top-level directory like this:

```
$ babel-node start-client.js
```

We'll add a `client` command to our `package.json`. That way, the method for booting the server and the client will look the same:

[67] https://github.com/kimmobrunfeldt/concurrently

```
# Boot the server
$ npm run server
# Boot the client
$ npm run client
```

Therefore, using `concurrently` will look like this:

```
$ concurrently "npm run server" "npm run client"
```

Let's add the `start` and `client` commands to our `package.json` now:

food-lookup-complete/package.json

```
  "scripts": {
    "start": "concurrently \"npm run server\" \"npm run client\"",
    "server": "babel-node start-server.js",
    "client": "babel-node start-client.js"
  },
```

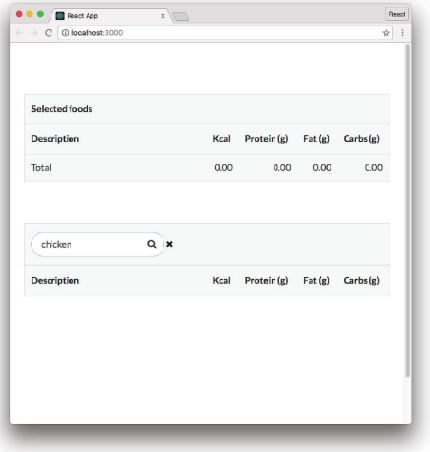
For `start`, we execute both commands, **escaping the quotes** because we're in a JSON file.

Save and close `package.json`. Now we can boot both servers by running `npm start`. Go ahead and do this now:

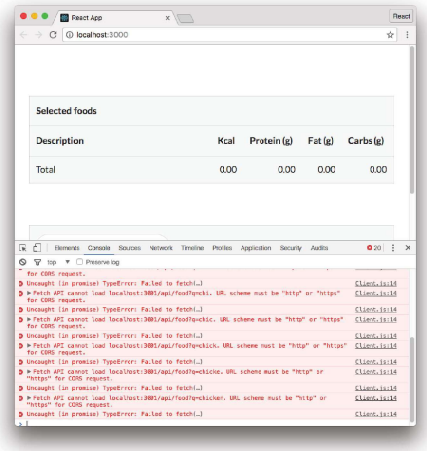
```
$ npm start
```

You'll see output for both the server and the client logged to the console. Concurrently has executed both run commands simultaneously.

When it appears everything has booted, visit `localhost:3000`. And then start typing some stuff in. Strangely, nothing appears to happen:



Popping open the developer console, we see that it is littered with errors:



Picking out one of them:

```
Fetch API cannot load http://localhost:3001/api/food?q=c. No 'Access-Control-All\
ow-Origin' header is present on the requested resource. Origin 'http://localhost\
:3000' is therefore not allowed access. If an opaque response serves your needs,\
 set the request's mode to 'no-cors' to fetch the resource with CORS disabled.
```

Our browser prevented our React app (hosted at `localhost:3000`) from loading a resource from a different origin (`localhost:3001`). We attempted to perform **Cross-Origin Resource Sharing** (or CORS). The browser prevents these types of requests from scripts for security reasons.

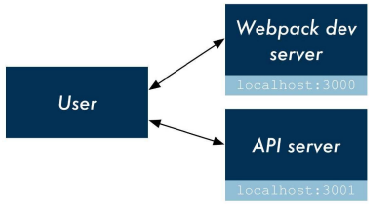
Note: If this issue didn't occur for you, you may want to verify your browser security settings are sound. Not restricting CORS leaves you open to significant security risk.

This is the primary difficulty with our two-server solution. But the two-server setup is common in development. Common enough that Create React App has a solution readily available for us to use.

Using the Webpack development proxy

Create React App enables you to setup the Webpack development server **to proxy requests to your API**. Instead of making a request to the API server at `localhost:3001`, our React app can make requests to `localhost:3000`. We can then have Webpack proxy those requests to our API server.

So, our original approach was to have the user's browser interact directly with both servers, like this:



However, we want the browser to just interact with the Webpack development server at `localhost:3000`. Webpack will forward along requests intended for the API, like this:

This proxy feature allows our React app to interact exclusively with the Webpack development server, eliminating any issues with CORS.

To do this, let's first modify `client/src/Client.js`. Remove the base URL, `localhost:3001`:

food-lookup-complete/client/src/Client.js

```
function search(query, cb) {
  return fetch(`/api/food?q=${query}`, {
    accept: 'application/json',
  }).then(checkStatus)
```

Now, `search()` will make calls to `localhost:3000`.

Next, in our client's `package.json`, we can set a special property, `proxy`. Add this property to `client/package.json` now:

```
// Inside client/package.json
"proxy": "http://localhost:3001/",
```


Make sure to add that line to the **client's** `package.json`, not the server's.

This property is special to Create React App, and instructs Create React App to setup our Webpack development server to proxy API requests to `localhost:3001`. The Webpack development server will infer what traffic to proxy. It will proxy a request to our API if the URL is not recognized or if the request is not loading static assets (like HTML, CSS, or JavaScript).

Try it out

Use Concurrently to boot both processes:

```
$ npm start
```

Visiting `localhost:3000`, we see everything working. Because our browser is interfacing with the API through `localhost:3000`, we have no issues with CORS.

Webpack at large

As a platform for JavaScript applications, Webpack packs numerous features. We witnessed some of these features in this chapter. Webpack's abilities can be considered more broadly underneath two categories:

Optimization

Webpack's optimization toolset for production environments is vast.

One immediate optimization Webpack provides is reducing the number of files the client browser has to fetch. For large JavaScript apps composed of many different files, serving a handful of bundle files (like `bundle.js`) is faster than serving tons of small files.

Code splitting is another optimization which builds off of the concept of a bundle. You can configure Webpack so that it only serves the JavaScript and CSS assets that are relevant to the page that a user is viewing. While your multi-page app might have hundreds of React components, you can have Webpack only serve the necessary components and CSS that the client needs to render whatever page that they are on.

Tooling

As with optimization, the ecosystem around Webpack's tooling is vast.

For development, we saw Webpack's handy hot- and auto-reloading features. In addition, Create React App configures other niceties in the development pipeline, like auto-linting your JavaScript code.

For production, we saw how we can configure Webpack to execute plug-ins that optimize our production build.

When to use Webpack/Create React App

So, given Webpack's power, you might ask: Should I just use Webpack/Create React App for all my future React projects?

It depends.

Loading React and Babel in `script` tags as we did in the first couple of chapters is still a completely sane approach. For some projects, the simplicity of this setup might be preferable. Plus, you can always start simple and then move a project to the more complex Webpack setup in the future.

What's more, if you're looking to roll React out inside an existing application this simple approach is your best bet. You don't have to adopt an entirely new build or deployment pipeline for your app. Instead, you can roll React components out one-by-one, all by simply ensuring that the React library is included in your app.

But, for many developers and many types of projects, Webpack is a compelling option with features that are too good to miss. If you're planning on writing a sizeable React application with many different components, Webpack's support for ES6 modules will help keep your codebase sensible. Support for npm packages is great. And thanks to Create React App, you get tons of tooling for development and production for free.

There is one additional deal breaker in favor of Webpack: testing. In the next chapter, we learn how to write tests for our React apps. As we'll see, Webpack provides a platform for easily executing our test suite in the console, outside of a browser.

Unit Testing

A robust test suite is a vital constituent of quality software. With a good test suite at his or her disposal, a developer can more confidently refactor or add features to an application. Test suites are an upfront investment that pay dividends over the lifetime of a system.

Testing user interfaces is notoriously difficult. Thankfully, testing React components is not. With the right tools and the right methodology, the interface for your web app can be just as fortified with tests as every other part of the system.

We'll begin by writing a small test suite without using any testing libraries. After getting a feel for a test suite's essence, we'll introduce the Jest testing framework to alleviate a lot of boilerplate and easily allow our tests to be much more expressive.

While using Jest, we'll see how we can organize our test suite in a behavior-driven style. Once we're comfortable with the basics, we'll take a look at how to approach testing React components in particular. We'll introduce Enzyme, a library for working with React components in a testing environment.

Finally, in the last section of this chapter, we work with a more complex React component that sits inside of a larger app. We use the concept of a mock to isolate the API-driven component we are testing.

Writing tests without a framework

If you're already familiar with JavaScript testing, you can skip ahead to the next section.

However, you might still find this section to be a useful reflection on what testing frameworks are doing behind the scenes.

The projects for this chapter are located inside of the folder `testing` that was provided with this book's code download.

We'll start in the `basics` folder:

```
$ cd testing/basics
```

The structure of this project:


```
$ ls
Modash.js
Modash.test.js
complete/
package.json
```

Inside of `complete/` you'll find files corresponding to each iteration of `Modash.test.js` as well as the completed version of `Modash.js`.

We'll be using `babel-node` to run our test suite from the command-line. `babel-node` is included in this folder's `package.json`. Go ahead and install the packages in `package.json` now:

```
$ npm install
```

In order to write tests, we need to have a library to test. Let's write a little utility library that we can test.

Preparing `Modash`

We'll write a small library in `Modash.js`. Modash will have some methods that might prove useful when working with JavaScript strings. We'll write the following three methods. Each returns a string:

`truncate(string, length)`

Truncates `string` if it's longer than the supplied `length`. If the string is truncated, it will end with `...`:

```
const s = 'All code and no tests makes Jack a precarious boy.';
Modash.truncate(s, 21);
  // => 'All code and no tests...'
Modash.truncate(s, 100);
  // => 'All code and no tests makes Jack a precarious boy.'
```

`capitalize(string)`

Capitalizes the first letter of `string` and lower cases the rest:

```
const s = 'stability was practically ASSURED.';
Modash.capitalize(s);
  // => 'Stability was practically assured.'
```

`camelCase(string)`

Takes a string of words delimited by spaces, dashes, or underscores and returns a camel-cased representation:

```
let s = 'started at';
Modash.camelCase(s);
  // => 'startedAt'
s = 'started_at';
Modash.camelCase(s);
  // => 'startedAt'
```

The name "Modash" is a play on the popular JavaScript utility library Lodash[68].

We'll write Modash as an ES6 module. For more details on how this works with Babel, see the aside ES6: Import/export with Babel. If you need a refresher on ES6 modules, refer to the previous chapter "Using Webpack with create-react-app.".

Open up `Modash.js` now. We'll write our library's three functions then export our interface at the bottom of this file.

First, we'll write the function for `truncate()`. There are many ways to do this. Here's one approach:

testing/basics/complete/Modash.js

```
function truncate(string, length) {
  if (string.length > length) {
    return string.slice(0, length) + '...';
  } else {
    return string;
  }
}
```

Next, here's the implementation for `capitalize()`:

testing/basics/complete/Modash.js

```
function capitalize(string) {
  return (
    string.charAt(0).toUpperCase() + string.slice(1).toLowerCase()
  );
}
```

Finally, we'll write `camelCase()`. This one's slightly trickier. Again, there are multiple ways to implement this but here's the strategy that follows:

[68] https://lodash.com/

1. Use `split` to get an array of the words in the string. Spaces, dashes, and underscores will be considered delimiters.
2. Create a new array. The first entry of this array will be the lower-cased version of the first word. The rest of the entries will be the capitalized version of each subsequent word.
3. Join that array with `join`.

That looks like this:

testing/basics/complete/Modash.js

```
function camelCase(string) {
  const words = string.split(/[\s|\-|_]+/);
  return [
    words[0].toLowerCase(),
    ...words.slice(1).map((w) => capitalize(w)),
  ].join('');
}
```

String's `split()` splits a string into an array of strings. It accepts as an argument the character(s) you would like to split on. The argument can be either a string or a regular expression. You can read more about `split()` here[69].

Array's `join()` combines all the members of an array into a string. You can read more about `join()` here[70].

With those three functions defined in `Modash.js`, we're ready to export our module.

At the bottom of `Modash.js`, we first create the object that encapsulates our methods:

testing/basics/complete/Modash.js

```
const Modash = {
  truncate,
  capitalize,
  camelCase,
};
```

And then we export it:

[69] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/split
[70] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/join

testing/basics/complete/Modash.js

```
export default Modash;
```

We'll write our testing code for this section inside of the file `Modash.test.js`. Open up that file in your text editor now.

ES6: Import/export with Babel

Our `package.json` already includes Babel. In addition, we're including a Babel plug-in, `babel-plugin-transform-class`

This package will let us use the ES6 import/export syntax. Importantly, we specify it as a Babel plugin inside the project's `.babelrc`:

```
// basics/.babelrc
{
  "plugins": ["transform-class-properties"]
}
```

With this plugin in place, we can now export a module from one file and import it into another.

However, note that this solution **won't work in the browser**. It works locally in the Node runtime, which is fine for the purposes of writing tests for our Modash library. But to support this in the browser requires additional tooling. As we mentioned in the last chapter, ES6 module support in the browser is one of our primary motivations for using Webpack.

Writing the first spec

Our test suite will import the library we're writing tests for, `Modash`. We'll call methods on that library and make **assertions** on how the methods should behave.

At the top of `Modash.test.js`, let's first import our library:

testing/basics/complete/Modash.test-1.js

```
import Modash from './Modash';
```

Our first assertion will be for the method `truncate`. We're going to **assert** that when given a string over the supplied length, `truncate` returns a truncated string.

First, we setup the test:

testing/basics/complete/Modash.test-1.js

```
const string = 'there was one catch, and that was CATCH-22';
const actual = Modash.truncate(string, 19);
const expected = 'there was one catch...';
```

We're declaring our sample test string, `string`. We then set two variables: `actual` and `expected`. In test suites, **actual** is what we call the behavior that was observed. In this case, it's what `Modash.truncate` actually returned. **expected** is the value we are expecting.

Next, we make our test's assertion. We'll print a message indicating whether `truncate` passed or failed:

testing/basics/complete/Modash.test-1.js

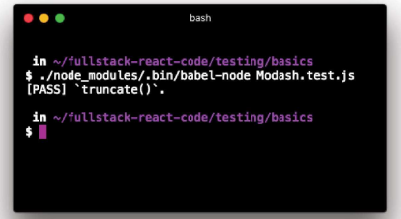
```
if (actual !== expected) {
  console.log(
    `[FAIL] Expected \`truncate()\` to return '${expected}', got '${actual}'`
  );
} else {
  console.log('[PASS] `truncate()`.');
}
```

Try it out

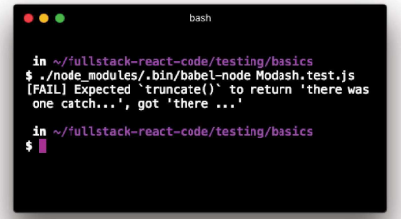
We can run our test suite at this stage in the command line. Save the file `Modash.test.js` and run the following from the `testing/basics` folder:

```
./node_modules/.bin/babel-node Modash.test.js
```

Executing this, we see a `[PASS]` message printed to the console. If you'd like, you can modify the `truncate` function in `Modash.js` to observe this test failing:



Test passing



Example of what it looks like when test fails

The `assertEqual()` function

Let's write some tests for the other two methods in `Modash`.

For all our tests, we're going to be following a similar pattern. We're going to have some assertion that checks if `actual` equals `expected`. We'll print a message to the console that indicates whether the function under test passed or failed.

To avoid this code duplication, we'll write a helper function, `assertEqual()`. `assertEqual()` will check equality between both its arguments. The function will then write a console message, indicating if the spec passed or failed.

At the top of `Modash.test.js`, below the import statement for `Modash`, declare `assertEqual`:

testing/basics/complete/Modash.test-2.js

```
import Modash from './Modash';

function assertEqual(description, actual, expected) {
  if (actual === expected) {
    console.log(`[PASS] ${description}`);
  } else {
    console.log(`[FAIL] ${description}`);
    console.log(`\tactual: '${actual}'`);
    console.log(`\texpected: '${expected}'`);
  }
}
```

A tab is represented as the `\t` character in JavaScript.

With `assertEqual` defined, let's re-write our first test spec. We're going to re-use the variables `actual`, `expected`, and `string` throughout the test suite, so we'll use the `let` declaration so that we can redefine them:

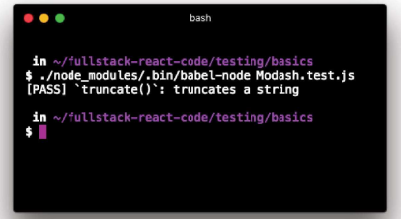
testing/basics/complete/Modash.test-2.js

```
let actual;
let expected;
let string;

string = 'there was one catch, and that was CATCH-22';
actual = Modash.truncate(string, 19);
expected = 'there was one catch...';

assertEqual('`truncate()`: truncates a string', actual, expected);
```

If you were to run `Modash.test.js` now, you'd note that things are working just as before. The console output is just slightly different:



Test passing

With our assert function written, let's write some more tests.

Let's write one more assertion for `truncate`. The function should return a string as-is if it's less than the supplied length. We'll use the same `string`. Write this assertion below the current one:

testing/basics/complete/Modash.test-2.js

```
actual = Modash.truncate(string, string.length);
expected = string;

assertEqual('`truncate()`: no-ops if <= length', actual, expected);
```

Next, let's write an assertion for `capitalize`. We can continue to use the same `string`:

testing/basics/complete/Modash.test-2.js

```
actual = Modash.capitalize(string);
expected = 'There was one catch, and that was catch-22';

assertEqual('`capitalize()`: capitalizes the string', actual, expected);
```

Given the example string we're using, this assertion tests both aspects of `capitalize`: That it capitalizes the first letter in the string and that it converts the rest to lowercase.

Last, we'll write our assertions for `camelCase`. We'll test this function with two different strings. One will be delimited by spaces and the other by underscores.

The assertion for spaces:

testing/basics/complete/Modash.test-2.js

```
string = 'customer responded at';
actual = Modash.camelCase(string);
expected = 'customerRespondedAt';

assertEqual('`camelCase()`: string with spaces', actual, expected);
```

And for underscores:

testing/basics/complete/Modash.test-2.js

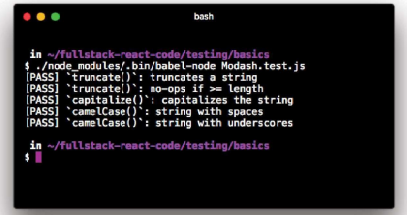
```
string = 'customer_responded_at';
actual = Modash.camelCase(string);
expected = 'customerRespondedAt';

assertEqual('`camelCase()`: string with underscores', actual, expected);
```

Try it out

Save `Modash.test.js`. From the console, run the test suite:

```
./node_modules/.bin/babel-node Modash.test.js
```



Tests passing

Feel free to tweak either the `expected` values for each assertion or break the library and watch the tests fail.

Our miniature assertion framework is clear but limited. It's hard to imagine how it would be both maintainable and scalable for a more complex app or module. And while `assertEqual()` works fine