React Intro

Instructor: Fernando Trigoso

Blog: trigoso.xyz

Twitter: @fertrig

What is React

- React is a technology that renders views
- A view is anything a user can see and interact with on a screen
- In web, React renders HTML elements
- In mobile, React renders native mobile elements

React Components (in the web)

- Components are the unit of work in React
- React breaks down views into components
- Each component renders HTML or other components, which in turn render their own HTML
- Each component is a file (not required)
- You can always tell how your component will render by looking at one source file

Why React

- Each component is a little state machine
- If you know the state of your component, you know the rendered output
- You don’t have to trace the program flow or hunt down multiple files
- When working on complex applications, or in teams, this is critically important
- Allows us to add more features and fix bugs with lower risk of breaking other things

Why React and Flux?

- React solves the problem of managing data in large applications
- Simple and Declarative
  - Express how your app should look with any given state
  - React will manage underlying UI changes
  - When data changes, React knows to update only the changed parts
- React Components are highly reusable
  - Very well encapsulated
  - Make code reuse, testing and separation of concerns easy
- Flux proposes a unidirectional data flow
  - Different from MVC pattern
  - Flux makes data flow its primary concern and makes it as simple as possible
- All of the above enable highly maintainable applications

Where did React come from

- React started in the web
- React was created in Facebook
- Facebook’s dev team built React to solve one problem: building large applications with data that changes over time
- React has evolved into an ecosystem

React

- Just the UI
  - Can be used with other frameworks
  - It can even be used in mobile apps and desktop apps
- Virtual DOM
  - You declare what the state of the DOM
  - React handles the best and fastest way to render it
- Data Flow
  - Unidirectional data flow
  - Does not use two-way binding

React Components

You build React applications around components

```
var Greeting = React.createClass({
    render() {
        return <h1>Hello World</h1>;
    }
});
var TimeNow = React.createClass({
    render() {
        return <span>The time is {Date.now()}</span>;
    }
});
var MyApp = React.createClass({
    render() {
        return <div><Greeting /><TimeNow /></div>;
    }
});
```

JSX Syntax

Components are a good way to manage views of applications. React doesn't use templates or display logic like other frameworks do. In React, you use code to generate HTML. That way you can use all the expressive power of a language like Javascript.

To make describing layout in code easy, there is JSX. JSX lets you create JavaScript objects using HTML syntax.

```
// plain javascript
React.createElement('h1', 'Hello World');


// using JSX
<h1>Hello World</h1>;
```

Starting point - ReactDOM.render

```
ReactDOM.render(
    <MyApp />,
    document.getElementById(‘main-container’)
);
```

Setup

Clone repo and fetch branches:

```
$ git clone https://github.com/fertrig/react-intro-class.git react-intro-class
$ cd react-intro-class
$ git fetch
```

Exercise

```
$ git checkout ex/00-react-dom-render
```

React Components

- Think of them as functions that take in props and state
- Render HTML in declarative way

```
var Greeting = React.createClass({
    render: function() {
        return (
            <div>
                <h1>Hello World</h1>
                <h3>Some title</h3>
            </div>
        );
    }
});
```

HTML Tags vs. React Components

- React can either render HTML tags or React components.
- HTML tags use lowercase tag names
- React components use variables that start with uppercase letter

```
var Greeting = React.createClass({...});

var Main = React.createClass({
   render: function() {
      return <div><Greeting></div>;
   }
});
```

JSX Gotchas

- class is reserved in javascript, use className instead
- for as well, use htmlFor

Exercise

```
$ git checkout ex/01-first-component
```

Props

Passing a prop to a component:

```
<Greeting planet=”Earth”/>

<Greeting planet={myHomePlanet} />

<Greeting planet={planets[2]} />
```

Using a prop from inside a component:

```
var Greeting = React.createClass({
    render: function() {
        return <h1>Hello from planet {this.props.planet}.</h1>;
    }
});
```

Exercise

```
$ git checkout ex/02-props
```

Javascript Expressions as props

Wrap the expression in a pair of curly braces ({}) instead of quotes ("").

```
<Greeting planet={isEarth ? ‘Earth’ : ‘Unknown’}>;
```

Event Handling

- Pass your event handler as a camelCased prop, like normal HTML
- React makes sure all events behave the same across browsers

```
var MyButton = React.createClass({
    render() {
        return <button onClick={this.handleClick}>;
    },
    handleClick() {
        ...
    }
});
```

Components are State Machines

- Think of your UI as being in various states
  - Inside React components, states are represented by `this.state` and `this.props`
- You just have render those states
- Update a component state and React renders the new UI based on the new state
- React handles updating the DOM

this.state

```
var Door = React.createClass({
    getInitialState() {
        return { isLocked: true };
    },
    render() {
        return (
            <div className={this.state.isLocked ? “locked” : “unlocked”}>
                <button onClick={this.lock}>Lock</button>
                <button onClick={this.unlock}>Unlock</button>
            </div>
        );
    },
    lock() {
        this.setState({ isLocked: true });
    }
    unlock() {
        this.setState({ isLocked: false });
    }
});
```

Exercise

```
$ git checkout ex/03-click
```

Which components should have `this.state`?

- Most components should take data from props
  - Makes them simple
  - Easy to reason about
- Some will need state to respond to user input (clicks, text entry, etc.)
- Most components should be stateless (just use props)
- Usually you will have high level components that will handle the state (Flux)

Computed data in state

Avoid storing computed data in state or props

```
var planets = [‘Mercury’, ‘Venus’, ‘Earth’];

// avoid
<Greeting planets={planets} planetCount={planets.length} >;

// let component compute count it in its render function
<Greeting planets={planets} >;
```

Owner, Parent and Children

- Owner-ownee relationship
- Parent-child relationship

```
var Car = React.createClass({
     render() {
          return (
               <div>
                    <Doors />
                    <Tires />
               </div>
          );
     }
});

// Car is the owner of div, Doors and Tires
// div is the parent (not the owner) of Doors and Tires
// Doors and Tires are children of div
```

Exercise

```
git checkout ex/04-state-props
```

Development Experience

Managing Files

- Keep things organized in multiple files
- Browser should not make multiple file requests
- Need a tool that will bundle all your files into one browser-optimized javascript file
- Webpack

ES6

- You should write ES6
- Community and documentation are rapidly adopting ES6
- Most browsers and most versions of node still run ES5
- Need a tool that will transpile your ES6 code to ES5
- Babel

Task Runner

- You want a task runner for your workflows
  - Run tests
  - Build assets: javascript, css, images
- Gulp

Webpack

- Be patient with it
- Lots of plugins
- Documentation is not great
- Will save a lot of time on the long run
- Also look at create-react-app

Setup Node.js

Install LTS version of node:

```
$ sudo npm i n -g

$ sudo n lts
$ node -v

(should say v4.x.x)
```

Setting up Webpack, Babel and Gulp

- $ git checkout sln/05-webpack
- React components files
- New modules in package.json
  - $ npm install
- Install gulp
  - $ sudo npm install -g gulp
- Two gulp tasks:
  - $ gulp build
  - $ gulp dev-server
- dev-server
  - Can see ES6 in chrome dev tools (source maps)
  - Can set breakpoints
- Make a quick change to your code

New single-page-app (spa)

- webpack can build multiple bundles for you
  - See webpack.config.js, entry property
- webpack can bundle many different files types for you
  - js, css, scss, less, png, jpg,
  - It uses loaders in different npm modules: style-loader, sass-loader, css-loader, file-loader
  - See webpack.config.js module.loaders property
- List of loaders
  - https://webpack.github.io/docs/list-of-loaders.html

Controlled Component

```
render() {
  return <input type="text" value="Hello!" />;
}
```

```
getInitialState() {
  return {value: 'Hello!'};
},
render() {
  return (
    <input
      type="text"
      value={this.state.value}
      onChange={this.handleChange}
    />
  );
},
handleChange(event) {
  this.setState({value: event.target.value});
}
```

Exercise

```
$ git checkout ex/05.1-input
```

Using images and css

```
render() {
     let myImageUrl = require(‘./my-image.png’);
     // myImageUrl will point to the image in the output or dist directory
}

---

render() {
     require(‘my-styles.css’);
    // when the component renders, the css will be injected to the head tag
}
```

Rendering lists

Array.map and key prop

```
// this.props.results is [
//   {id: ‘a’, text: ‘aaa’},
//   {id: ‘a1’, text:’a111’},
//   {id: ‘b’, text: ‘bbb’}]

  render() {
    return (
      <ol>
          {this.renderResults()};
      </ol>
    );
  },

  renderResults() {
     return this.props.results.map(function(result) {
       return <li key={result.id}>{result.text}</li>;
     });
  }
```

Component Specs and Lifecycle

```
React.createClass({
     // specs
     getInitialState() {},
     render() {},
     propTypes: {},
     getDefaultProps(): {},
     mixins: [],
     statics: {},

     // mounting lifecycle
     componentWillMount() {},
     componentDidMount() {},
     componentWillUnmount() {},

     // updating lifecycle
     componentWillReceiveProps(nextProps) {},
    shouldComponentUpdate(nextProps, nextState) {},
    componentWillUpdate(nextProps, nextState) {},
     componentDidUpdate(prevProps, prevState) {},
});
```

Accessing the underlying DOM elements

“React is very fast because it never talks to the DOM directly. React maintains a fast in-memory representation of the DOM. render() methods actually return a description of the DOM, and React can compare this description with the in-memory representation to compute the fastest way to update the browser.

...

Most of the time you should stay within React's "faked browser" world since it's more performant and easier to reason about. However, sometimes you simply need to access the underlying API, perhaps to work with a third-party library like a jQuery plugin. React provides escape hatches for you to use the underlying DOM API directly.”

-- https://facebook.github.io/react/docs/working-with-the-browser.html

Refs to DOM Elements

You can have callback refs and string refs, callback refs are preferred

```
// recommended
render() {
  return <input type="text" value="Hello!" ref={this.focus}/>;
},
focus(domElement) {
  domElement.focus();
}
```

```
render() {
  return <input type="text" value="Hello!" ref={(domElement) => domElement.focus()}/>;
}
```

```
render() {
  return <input type="text" value="Hello!" ref=”textInput”/>;
},
componentDidMount() {
  this.refs.textInput.focus();
}
```

Exercise - start the chat application

```
$ git checkout ex/06-chat
$ npm install
(restart gulp dev-server)
```

```
// standard react component
let Greeting = React.createClass({
  render() {
    return <h1>Hello {this.props.name}, {this.props.age}</h1>
  }
});

// ES6 class react component
class Greeting extends React.Component {
     constructor() {this._handleClick.bind(this);}
     render() {...}
}
```

```
// stateless function component
function Greeting(props) {
  return <h1>Hello {props.name}, {props.age}</h1>
}
```

Only for components that are pure functions of their props. Should not have internal state or refs. Do not have lifecycle methods.

```
// stateless function component with
// parameter destructuring
function Greeting({ name, age }) {
  return <h1>Hello {name}, {age}</h1>
}
```

Since it can be optimized further it is the recommended pattern. Use it whenever possible.

Flux

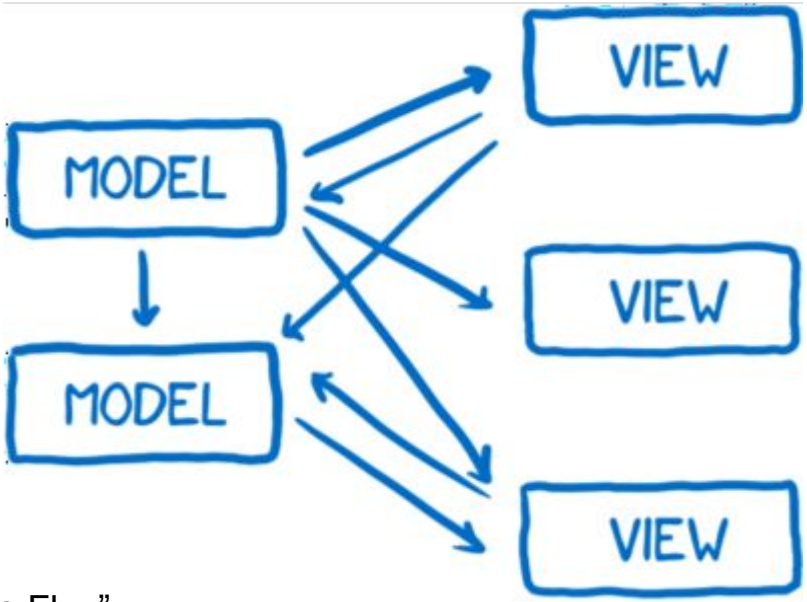
An application architecture

Flux

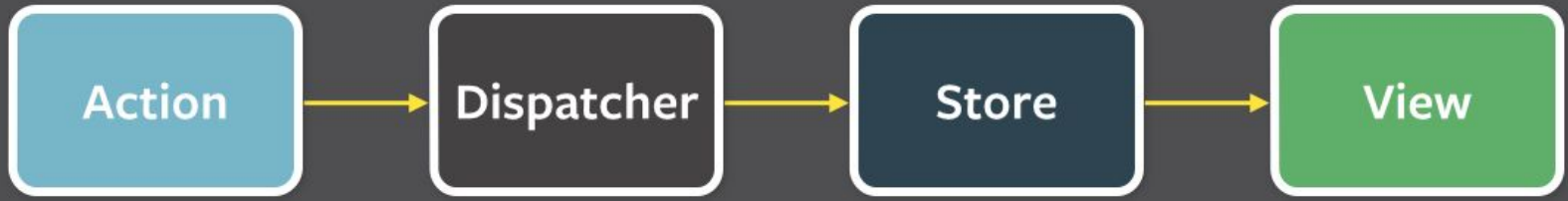
- A pattern, not a framework
- You can use frameworks that implement this pattern
  - Redux
    - Good one but you have to be comfortable with immutability and you are more locked in
  - alt
  - Fluxible
  - You can also just keep it vanilla and implement the pattern yourself
- Unidirectional data flow
- Has several parts
  - Dispatcher
  - Action creators
  - Stores
  - Views
- It is not MVC

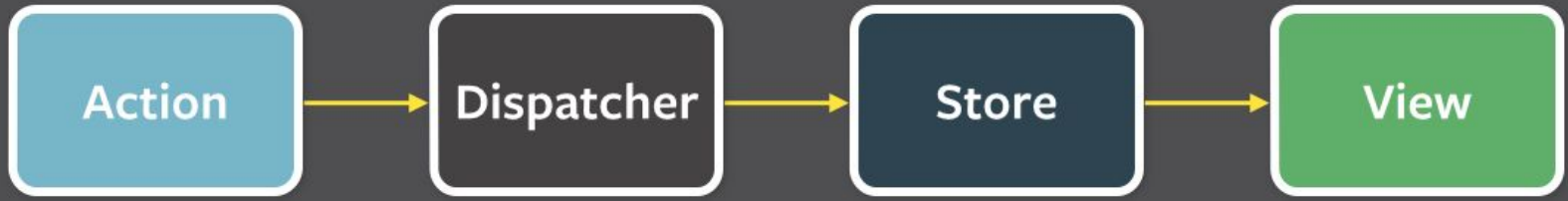
Why Flux? Why unidirectional?

In MVC views update models and models update other models. The data flow can be hard to maintain.



Pic from “A Cartoon Guide to Flux”:
https://code-cartoons.com/a-cartoon-guide-to-flux-6157355ab207#.a9wkrbwty


Action
Dispatcher
Store
View



Actions

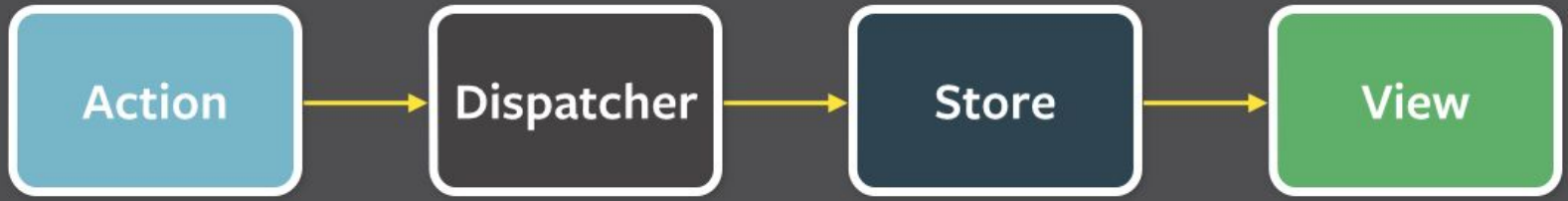
Actions are objects that may carry some data or payload.

Created through *action creators*, which are helper methods that create actions. Actions are usually created in response to a user event or server response.

Action creators will dispatch an action using the dispatcher. An action should have a type and it may have some payload. Payload consists of event data.

```
// sample action
let action = {
     type: ‘submit-new-message’,
     payload:
     {
          content: ‘Hello’
     }
};
```

```
// sample action
let action = {
     slug: ‘submit-new-message’,
     data:
     {
          content: ‘Hello’
     }
};
```



Dispatcher

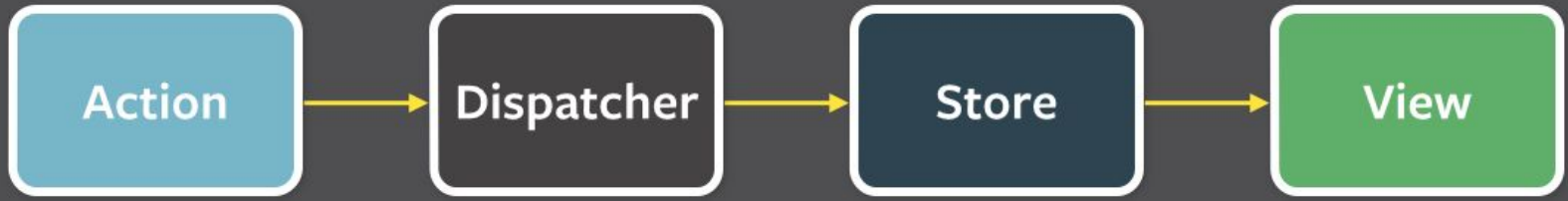
There is only one dispatcher in your application. Action creators dispatch actions using the dispatcher. Stores register themselves with the dispatcher.

Every store will get every action in the system. Each store will handle actions it cares about.

As the application grows, you will have dependencies between stores. Stores can manage this dependency through the dispatcher.

```
// action creator dispatches
dispatcher.dispatch(someAction);

// store registers with dispatcher and
// will handle actions it cares about
dispatcher.register(action => {
  if (action.type === 'submit-new-message') {
    addNewMessage(action.payload.content);
  }
});
```



Store

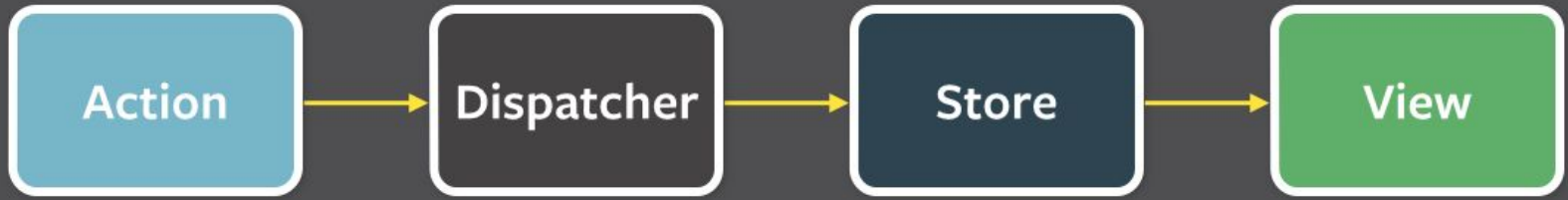
Stores contain the application state and logic. Similar to the model in MVC but not the same. You usually don't create a store per model. A store should represent a sub-domain of your application. It can track multiple objects. It can track collections of items and singleton items.

In our chat app, we could have a store that tracks channels and messages within those channels. We could have another store that tracks users and their online status.

Stores register themselves with dispatcher. When it receives an action it cares about it changes its state and it emits a change event. Views are listening to the change event. Views will retrieve the state from the store.

```
dispatcher.register(action => {
  if (action.type === 'submit-new-message') {
    addNewMessage(action.payload.content);
    // store emits change, views are listening
    emitChange();
  }
});

// views will get state from store
get messages() {...}
get newMessage() {...}
```



View

Views are React components. Some of these views will listen to store changes. The views that listen for changes are called view-controllers. They are not MVC controllers though.

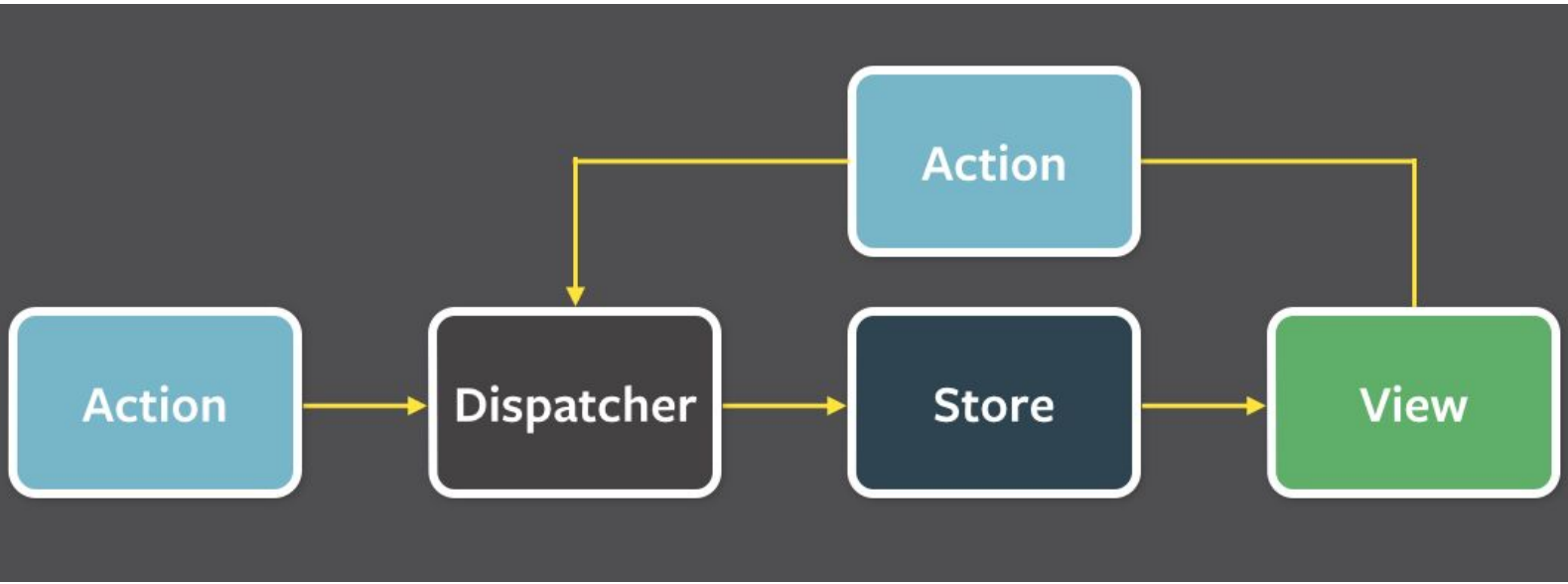
View-controllers own other views. View-controllers will get its state from the store, they will then pass data as props to the views they own.

View-controllers will get its initial state from the store and when a change event happens, they will set its own state using data from the store.

```
getInitialState() {
  return { messages: store.getMessages() };
}

componentDidMount() {
  store.addChangeListener(this.handleChange);
}

handleChange() {
  this.setState({
    messages: store.getMessages()
  });
}
```


Action
Action
Dispatcher
Store
View

Exercise

```
$ git checkout ex/07-flux
$ npm install
(restart gulp dev-server)
```

Exercise

```
$ git checkout sln/08-realtime
```

Asynchronous API calls with React

Asynchronous API calls

- When you make an asynchronous API call (with ajax) there are two very important moments
  - The moment you start the call
  - The moment you receive a response
- Usually, each of these moments requires a change in the application state
- In Flux, you dispatch actions to change application state
- These actions are synchronous
- For any API request you want to dispatch three actions

An API call dispatches three actions

- An action that informs the stores that the request started
  - Store may
    - set the request state to ‘fetching’ or
    - set a flag isFetching
- An action that informs the stores that the request completed successfully
  - Store may
    - set the request state to ‘success’ or
    - set a success flag and reset isFetching
- An action that informs the stores that the request failed
  - Store may
    - set the request state to ‘failed’ or
    - set an error flag

Action Types

- You can have one action type with a status field:
  - { type: ‘api-request’, status: ‘fetching’ }
  - { type: ‘api-request’, status: 'error', err: responseError }
  - { type: ‘api-request’, status: 'success', data: responseData }
- Or you can have separate action types for each request status:
  - { type: ‘api-request-starting’ }
  - { type: ‘api-request-success’, data: responseData }
  - { type: ‘api-request-error’, err: responseError }

Exercise

```
$ git checkout ex/09-ajax
$ npm install
```

Exercise

```
$ git checkout sln/10-routing
$ npm install
```