Chapter 14: React AJAX call

Section 14.1: HTTP GET request

Sometimes a component needs to render some data from a remote endpoint (e.g. a REST API). A standard practice is to make such calls in `componentDidMount` method.

Here is an example, using superagent as AJAX helper:

```
import React from 'react'
import request from 'superagent'

class App extends React.Component {
  constructor () {
    super()
    this.state = {}
  }
  componentDidMount () {
    request
      .get('/search')
      .query({ query: 'Manny' })
      .query({ range: '1..5' })
      .query({ order: 'desc' })
      .set('API-Key', 'foobar')
      .set('Accept', 'application/json')
      .end((err, resp) => {
        if (!err) {
          this.setState({someData: resp.text})
        }
      })
  },
  render() {
    return (
      <div>{this.state.someData || 'waiting for response...'}</div>
    )
  }
}

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

A request can be initiated by invoking the appropriate method on the `request` object, then calling `.end()` to send the request. Setting header fields is simple, invoke `.set()` with a field name and value.

The `.query()` method accepts objects, which when used with the GET method will form a query-string. The following will produce the path `/search?query=Manny&range=1..5&order=desc`.

POST requests

```
request.post('/user')
  .set('Content-Type', 'application/json')
  .send('{"name":"tj","pet":"tobi"}')
  .end(callback)
```

See Superagent docs for more details.

Section 14.2: HTTP GET request and looping through data

The following example shows how a set of data obtained from a remote source can be rendered into a component.

We make an AJAX request using `fetch`, which is build into most browsers. Use a `fetch` polyfill in production to support older browsers. You can also use any other library for making requests (e.g. axios, SuperAgent, or even plain Javascript).

We set the data we receive as component state, so we can access it inside the render method. There, we loop through the data using `map`. Don't forget to always add a unique key attribute (or prop) to the looped element, which is important for React's rendering performance.

```
import React from 'react';

class Users extends React.Component {
  constructor() {
    super();
    this.state = { users: [] };
  }

  componentDidMount() {
    fetch('/api/users')
      .then(response => response.json())
      .then(json => this.setState({ users: json.data }));
  }

  render() {
    return (
      <div>
        <h1>Users</h1>
        {
          this.state.users.length == 0
            ? 'Loading users...'
            : this.state.users.map(user => (
              <figure key={user.id}>
                <img src={user.avatar} />
                <figcaption>
                  {user.name}
                </figcaption>
              </figure>
            ))
        }
      </div>
    );
  }
}

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

Working example on JSBin.

Section 14.3: Ajax in React without a third party library - a.k.a with VanillaJS

The following would work in IE9+

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

```
class App extends React.Component {
  constructor () {
    super()
    this.state = {someData: null}
  }
  componentDidMount () {
    var request = new XMLHttpRequest();
    request.open('GET', '/my/url', true);

    request.onload = () => {
      if (request.status >= 200 && request.status < 400) {
        // Success!
        this.setState({someData: request.responseText})
      } else {
        // We reached our target server, but it returned an error
        // Possibly handle the error by changing your state.
      }
    };

    request.onerror = () => {
      // There was a connection error of some sort.
      // Possibly handle the error by changing your state.
    };

    request.send();
  },
  render() {
    return (
      <div>{this.state.someData || 'waiting for response...'}</div>
    )
  }
}

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