React Demo 2: Integrating a React App with a Backend API.

Configure Proxies and/or CORS

Generally speaking, conventional browsers implement a same-origin security policy when it comes to AJAX fetches. That is, AJAX fetches to domains different from the page from which they are made are blocked^ by default.

^ The truth is a bit more complicated. Not every Cross-Origin request is blocked. Browsers may allow requests through that are clearly not harmful (e.g., GET requests with no cookies).

The same origin policy means that we must take special steps to allow our React front end to communicate with the Rails API server. (In order to meet the requirements for Same Origin, the URI scheme, hostname, and port must all be identical. Therefore, it is not sufficient to run both React and Rails on your local machine with different ports.)

Proxy

Experts disagree; but, in my opinion, the safest solution is to set up a proxy on the server. When you do this, the API requests made by the client go back to the server (i.e., the same origin), and the proxy then forwards the requests to the Rails API. You can then configure Rails to only accept API requests from the proxy.

To set up a proxy in your development environment, add the following line to `package.json`.

```
    "proxy": "https://your-rails-server.com",
```

If you are using CodeAnywhere, you will also need to add this line to `.env.development.local`

```
    DANGEROUSLY_DISABLE_HOST_CHECK=true
```

As the name implies, disabling the host check is a big security hole. Don't do this in a production environment. I also would avoid doing this on your development machine.

CORS

Cross-origin resource sharing (CORS) is a standard that allows a web browser and server to interact in a way

that permits cross-origin AJAX calls from a browser to a server. In order to use our Rails API without a proxy, we need to enable CORS on it. To do this, you need to include the following in the Gemfile of your rails app:

```
gem 'rack-cors', :require => 'rack/cors'
```

Don't forget to run the bundle install command after updating the `Gemfile` :

```
bundle install
```

Now, add the following code to `config/initializers/cors.rb` , and update the `origins` setting as shown:

```
Rails.application.config.middleware.insert_before 0, Rack::Cors do
  allow do
    origins '*'

    resource '*',
      headers: :any,
      methods: [:get, :post, :put, :patch, :delete, :options, :head]
  end
end
```

Last week, we were lazy and just modified our existing controller to render JSON when requested. In general, this is problematic because APIs use different security measures than a full web app. At minimim, we should have created seprate controllers for the API (See https://stackoverflow.com/questions/38250405/combining-api-and-web-views-in-rails-5.) Another approach would be to create a seprate Rails app specifically configured to be an API (just add the `--api` flag when creating running `rails new` to create the app).

The quick fix for our laziness is to add this line to `application_controller.rb`

```
    protect_from_forgery unless: -> { request.format.json? }
```

Go ahead and run your rails server once you've made the above changes.

```
rails s
```

You are now ready to add the API to your React app.

React

1. We'll begin by adding a package named Axios to our React project. Axios is an HTTP client library that will

greatly simplify the fetching / processing of our blog API. Install it from your top level directory of your react projet as follows:

```
npm install --save axios
```

and then insert the following import statement at the top of your `src/index.js file`:

```
import axios from 'axios';
```

2. Next, we're going to define a constant for the base URL of our API endpoint at the top of src/index.js:

```
const API_BASE = 'http://localhost:3000/';
```

3. Now, we're ready to implement our CRUD methods. We'll start by rewriting our `loadAuthors` to use the API instead of the hard-coded data:

```
  loadAuthors() {
    axios
      .get(`${API_BASE}/authors.json`)
      .then(res => {
              this.setState({ authors: res.data });
              console.log(`Data loaded! = ${this.state.authors}`)
            })
      .catch(err => console.log(err));
  }
```

Save the above changes and take a look at your web page. Assuming you have data in your backend, it should be rendering now on the web page.

4. We're going to go ahead and implement CRUD helpers in terms of axios on our Authors component and later wire them up to our UI. This is what they look like:

```
  addAuthor(newAuthor) {
    axios
      .post(`${API_BASE}/authors.json`, newAuthor)
      .then(res => {
        res.data.key = res.data.id;
        this.setState({ authors: [...this.state.authors, res.data] });
      })
      .catch(err => console.log(err));
  }

  updateAuthor(author) {
    axios
      .put(`${API_BASE}/authors/${author.id}.json`, author)
      .then(res => {
          this.loadAuthors();
      })
      .catch(err => console.log(err));
  }

  removeAuthor(id) {
    let filteredArray = this.state.authors.filter(item => item.id !== id)
    this.setState({authors: filteredArray});
    axios
      .delete(`${API_BASE}/authors/${id}.json`)
      .then(res => {
              console.log(`Record Deleted`);
              //this.clearForm();
            })
      .catch(err => console.log(err));
  }
```

As before, we need to bind each of these functions in our Authors constructor:

```
this.removeAuthor = this.removeAuthor.bind(this);
this.addAuthor = this.addAuthor.bind(this);
this.updateAuthor = this.updateAuthor.bind(this);
```

5. Before we implement our form, we need to setup some additional helpers that will help us configure that form based on our state info. In particular, if we are creating a new author, all we need is a submit button, but if somebody taps on an edit button in the author list, we need to store the selected author in our state at the Authors component level, so that it can pass it down to the AuthorForm to fill in the form with the selected values. In addition, we need to modify the form so it has a cancel button to backout of the edit if we decide not to make a change after all. So first, we need to update our Author state a bit. In the

constructor of Authors we now initialize state as follows:

```
this.state = {
  authors: [],
  formMode: "new",
  author: {lname:"", fname:"", email:"", id: "9999999"}
};
```

In addition to the array of authors, we'll have a `formMode` state value that is "new" if the form is configured for creating a new author, and "edit" if it is configured for editing an existing authors.

We'll also add a state value `author` that will hold the values of the currently selected author. This state will be passed down to the AuthorForm. By default they will just be blank. Notice, however, that this blank author does have an id. React will watch this id for changes so it knows when to update the form.

6. Since the form configuration will be driven from the child form, we also need to provide a helper method to cause the form to transition from mode to mode. We will do that with this `updateForm` function defined on Authors:

```
  updateForm(mode, authorVals) {
      this.setState({
        author: Object.assign({}, authorVals),
        formMode: mode,
      });
  }
```

Note that all it does is update the state per the passed values. Once again, we need to bind this function in the Authors constructor.

7. We'll also add a helper that clears the form - e.g. puts it back into "new" mode with blank fields (un-comment the call in delete below) :

```
  clearForm()
  {
    console.log("clear form");
    this.updateForm("new",{fname:"",lname:"",email:"", id: "99999999"});
  }
```

8. Finally, we need a helper method that handles form submission - that is, it will issue an HTTP PUT to the API in the case of an "edit" mode submit or an HTTP POST to the API in the case of a "new" mode submit.

```
 formSubmitted(author) {
    if(this.state.formMode === "new") {
      this.addAuthor(author);
    } else {
      this.updateAuthor(author);
    }
    this.clearForm();
  }
```

9. With these helpers in place, we can now pass down the appropriate state and helper function references to the subordinate children by updating the render method of our Authors component:

```
  render() {
    return (
      <div className="authors">
        <AuthorForm
          onSubmit={(author) => this.formSubmitted(author)}
          onCancel={(mode,author) => this.updateForm(mode,author)}
          formMode={this.state.formMode}
          author={this.state.author}
          key={this.state.author.id}
        />
        <AuthorList
          authors={this.state.authors}
          onDelete={(id) => this.removeAuthor(id)}
          onEdit={(mode,author) => this.updateForm(mode,author)}
        />
      </div>
    );
  }
```

(By placing `key` as a property to the AuthorForm, React will re-create the component whenever it changes. It is important for React to re-create the component because the AuthorForm uses state that will change when the author changes.)

10. All we have left at this point is to implement the AuthorForm component. Since it is a form, it will basically maintain its own local copy of the state to keep track of the form input as it occurs. We start out by redefining AuthorForm as a ES6 class with a constructor that sets the state it will manage:

```
class AuthorForm extends React.Component {

  constructor(props) {
    super(props);
```

```
    this.state = {
      fname: props.author.fname,
      lname: props.author.lname,
      email:  props.author.email,
      id: props.author.id
    };
  }

  renderButtons() {
    if (this.props.formMode === "new") {
      return(
        <button type="submit" className="btn btn-primary">Create</button>
      );
    } else {
      return(
        <div className="form-group">
          <button type="submit" className="btn btn-primary">Save</button>
          <button type="submit" className="btn btn-danger" onClick={this.handleCa
ncel} >Cancel</button>
        </div>
      );
    }
  }

  render()  {
    return (
      <div className="author-form">
        <h1> Authors </h1>
        <form onSubmit={this.handleSubmit}>
          <div className="form-group">
            <label>First Name</label>
            <input type="text" className="form-control" autoComplete='given-name'
 name="fname" id="fname" placeholder="First Name" value={this.state.fname} onChan
ge={this.handleInputChange}/>
          </div>
          <div className="form-group">
            <label htmlFor="lname">Last Name</label>
            <input type="text" className="form-control" autoComplete='family-name
' name="lname" id="lname" placeholder="Last Name" value={this.state.lname} onChan
ge={this.handleInputChange}/>
          </div>
          <div className="form-group">
            <label htmlFor="email">Email address</label>
            <input type="email" className="form-control" autoComplete='email' nam
e="email"  id="email" placeholder="name@example.com" value={this.state.email} onC
```

```
hange={this.handleInputChange}/>
          </div>
          {this.renderButtons()}
        </form>
      </div>
    );
  }
}
```

Notice that the button configuration changes depending on the form's mode, so we pulled that logic out into a separate method.

11. Next, we need some input handlers to deal with input as well as the submit/cancel buttons. These are already referenced in the form we defined above.

```
  handleInputChange(event) {
    const target = event.target;
    const value = target.value;
    const name = target.name;
    this.setState({
      [name]: value
    });
  }

  handleSubmit(event)
  {
      this.props.onSubmit({
        fname: this.state.fname,
        lname: this.state.lname,
        email: this.state.email,
        id: this.state.id,
      });
      event.preventDefault();
  }

  handleCancel(event)
  {
    this.props.onCancel("new", {fname:"", lname:"", email:""});
    event.preventDefault();
  }
```

We will also bind these functions in the AuthorForm constructor:

```
this.handleInputChange = this.handleInputChange.bind(this);
this.handleSubmit = this.handleSubmit.bind(this);
this.handleCancel = this.handleCancel.bind(this);
```

12. Notice how these handlers work. As you type characters into the form, the state is updated by `handleInputChange`. However, the values of the form are tied directly to the component's state variables, so the component is updated whenever the state changes, and we see the form fill as we type. Confirm this by pulling up the app in the browser and inspecting with the React developer tools.

13. At this point, you should be able to try the app out and have full CRUD capabitilies.