Express & AJAX

Submit Data Without Reloading

Our Requests So Far

- In our apps, when we want to POST a form or get some new data, we've been making full page requests to Express
- The server then comes back with a whole new set of HTML that reflects what happened in our requests
- But some of the things we submit are either very small, or require immediate feedback
- Rather than reloading the whole page and implementing all of the logic in our templates, it would be nice if we could do everything in the same page

AJAX Requests

- You may recall AJAX requests from the Jukebox project
- To communicate with Soundcloud, we sent an AJAX request to their server, and got back some JSON data
- We were able to update our page if everything went well (in the `success` callback), or display errors if things went wrong (in the `error` callback)
- **This was Javascript that ran on the client computer, NOT on our Express servers**

Making the Request

- For the purposes of this lesson, we'll assume we're using jQuery and have the `$.ajax` function
- Instead of providing a full URL to the `$.ajax` function, we can provide a relative path to have it request to our own server
  - e.g. `"/auth/login"` instead of `"https://soundcloud.com/..."`
- We can also provide arguments to send using the `data` options argument
  - In a `GET` request, this will end up in `req.query`
  - In a `POST` or `PUT` request, this will end up in `req.body`
- We can either intercept a `<form>` submit and send the values during that, or we can just make our own request whenever we want

Making the Request - Form Submit

```
// assets/js/login.js
var $form = $("#login-form");
var $username = $form.find("[name=username]");
var $password = $form.find("[name=password]");

$form.on("submit", function(ev) {
    // Prevent form from submitting
    ev.preventDefault();

    // Submit the request via AJAX
    $.ajax("/auth/login", {
        method: "POST",
        data: {
            username: $username.val(),
            password: $password.val(),
        },
        success: function() {
            // Redirect to homepage if all goes well
            window.location = "/home";
        },
        error: function() {
            // Alert that the login was bad if it goes poorly
            alert("Invalid login info, please try again!");
        },
    });
});
```

Making the Request - Custom Event

```
// assets/js/like.js
var $likeButtons = $(".post-like")

$likeButtons.on("click", function(ev) {
    var $btn = $(ev.target);
    var postId = $btn.data("postid");

    $.ajax("/posts/" + postId + "/like", {
        method: "POST",
        success: function() {
            // If all went well, add a class that shows they liked it
            $btn.addClass("isLiked");
        },
        error: function() {
            // If it goes wrong, let them know, unlike button
            alert("Unable to like post");
            $btn.removeClass("isLiked");
        },
    });
});


<!-- On some user's post -->
<button class="post-like" data-postid="125719351">Like</button>
```

API Servers

- The server that returns information from an AJAX request is often referred to as an "API", an Application Programming Interface
- Soundcloud provided us with their API, which we sent our requests to
- We can also turn our own Express servers into an API, by adding the ability to reply with JSON instead of HTML
- **This is code that will run on our Express server, NOT on the client's computer**

API Servers - JSON Responses

- There are 2 ways to handle making an API server
- The first is to add JSON versions of responses to your HTML endpoints
  - This is nice because you end up with endpoints that work in either the traditional HTML format, or the JS AJAX format
- The second is to make purely API-only endpoints that only speak in JSON
  - This is nice because it keeps your code separate and organized, and allows you to change how HTML routes work without affecting AJAX routes
- We'll look at both

Sidenote: BodyParser.json

- We've been using `BodyParser.urlencoded()` to handle HTML form POSTs
  - This is because the POST body looks like `arg1=string,arg2=123` etc.
- However AJAX requests usually use a JSON POST body instead
  - These look like `{ "arg1": "string", "arg2:" 123 }`
- In order to handle these, we must make sure we're also using `BodyParser.json()`
- Nothing else needs to be done, these POSTs will also end up in `req.body`

Method 1: JSON & HTML Responses

- Most requests are sent with a `Content-Type` header for what they expect back
  - Requests from your browser want `text/html` back
  - Requests from Javascript want `application/json` back
- We can have different responses in the same handler for both types by using `res.format()`
- We pass that function an object of `"type": function()` pairs for different responses

Method 1: JSON & HTML Responses (Code)

```
// routes/auth.js
router.post("/login", function(req, res) {
    // Handle login and set variable `error` if there is one, `user` if succeeded
    res.format({
        "text/html": function() {
            if (error) {
                res.status(400);
                res.render("login", { error: error.message });
            } else {
                res.redirect("/home");
            }
        },
        "application/json": function() {
            if (error) {
                res.status(400);
                res.json({ error: error });
            } else {
                res.json({ user: user });
            }
        },
    });
});
```

Method #2: Separate Endpoints

- Instead of jamming everything into one endpoint, we can create separate endpoints for HTML and JSON
- We often separate these endpoints into separate routers as well
- This can be good because otherwise, we'd also need all of our middlewares to handle this `res.format` standard if they ever do a `res.redirect` or `res.respond`
- This also forces us to extend our models and provide shared utility functions, which is good practice

Method #2: Separate Endpoints (Code)

```
// app.js
const authRouter = require("./routes/auth");
const apiRouter = require("./routes/api");

app.use("/auth", authRouter);
app.use("/api", apiRouter);

app.listen(/* ... */);
```

Method #2: Separate Endpoints (Code cont.)

```
// routes/auth.js
const User = require("../models/user");

router.post("/login", function(req, res) {
    User.login(req.username, req.password)
        .then(function() {
            res.redirect("/home");
        })
        .catch(function(error) {
            res.status(400);
            res.render("login", {
                error: error.message,
            });
        });
});
```

Method #2: Separate Endpoints (Code cont.)

```
// routes/api.js
const User = require("../models/user");

router.post("/login", function(req, res) {
    User.login(req.username, req.password)
        .then(function(user) {
            res.json({ user: user });
        })
        .catch(function(error) {
            res.status(400);
            res.json({ error: error });
        });
});
```

Sidenote: AJAX / JS Reliance

- While Javascript submits and APIs are really cool, it's a best practice to handle both HTML submits and AJAX submits
- Some users browser without Javascript enabled, or if Javascript fails, the form may submit with a regular style POST request
- By implementing both possibilities, we make our applications as accessible as possible
- However, there's nothing wrong with having *some* functionality not work without Javascript, as long as your core flow works, or you have a message warning users otherwise

Challenge: DropDox API

- Let's take our DropDox project from before and enhance it with an API for signing up, logging in, and deleting files
- Download a new version of the project at the URL below
- Copy in your .env file from previous versions of this project
- Refer to the README for how to complete this challenge

https://github.com/wbobeirne/nycda-dropdox-challenge/tree/api