Seneca College

Web Programming

WEB322 – Week 5 – Class 1

Processing Forms with Express.js

In this week's lecture, we are going to learn about processing forms on a web page using Express.js. Now that you have a basi understanding of creating server side routes to handle requests from the client, we can build on that and look at a basic form sent to the server, processed, and a response is returned.

HTML Forms Review

Let's review the typical HTML elements used in an HTML form and look at some of the newer input types that work in HTML5 There are really just a few core html elements that you need to know to build forms on the web: form, label, input, textarea, a button. <input> however, has many different types that can be used for the 'type' attribute. A summary is provided below in a table for reference.

Element	Description
<form>	The form element is the main container to hold your form and all of its inputs and a submit button. The form e has several attributes that control it's behavior. The most common 3 are 'enctype', 'method', and 'name'. The en the encoding type. Setting this to 'multipart/form-data' is important if you are working with forms that have file uploads that go with the form data. The 'method' specifies which HTTP verb to use when making the submissio request (GET or POST). Finally, the 'name' attribute is typically used by your view or client side code to validate t form. For more information see the link.
<label>	The label element is used to provide a label for input boxes. You can use the label's 'for' attribute to make the la clickable to focus its associated input. This adds a nice touch of usability to forms and can make it easier to foc input areas associated with a label.
<input>	The input element has many different values that can be used for it's 'type' attribute. The most common are: bu checkbox, date, file, password, radio, and text. Each type determines how the input will appear in the browser t user. The full documentation about each type of input element is specified in the link for input's.
<textarea>	textarea is much like an <input type='text'> input, except it allows multiple lines of text. Basically, it is a text box has space to add a lot of text instead of 1 line of text in a regular text input element. The textarea is useful for capturing user input that would typically be long and detailed or several sentences long. `This is what a textarea looks like. Type something in!`

<button>	The button element can be used in place of an input with type submit. A button element can be used the same an input to reset the form or submit the form. The biggest advantage to using a button element instead of an i element to submit a form, is that the button element can have HTML content inside it. You can include images other content that will render inside the button, whereas input elements just allow text in their "value" attribute

Creating a form in HTML

Let's make a small form right here on this page that "submits" the data and writes to the console after attempting to submit. C will be responsible for creating a new user for a site that asks for a username to use, email address, name, password, and a ra button that asks if they would like to be subscribed to a monthly newsletter.

Choose a username: *A Username you'd like to use*

Your Email: *Email address*

Your Name: *Your name*

Your Password:

Confirm Password:

Would you like to subscribe to our monthly newsletter?

- Yes
- No

Submit (using a regular input element to submit)

Submit (using a button element to submit)

The code for this form looks like this:

```
<form onsubmit="console.log('form Submitted'); return false;">
  <table style="border: 1px dashed #cdcdcd;padding:6px;">
    <tbody>
      <tr>
        <td>
          <label for="username">Choose a username:</label>
        </td>
        <td>
          <input id="username" type="text" placeholder="A Username you'd like to use" />
        </td>
      </tr>
      <tr>
        <td><label for="email">Your Email:</label></td>
        <td><input id="email" type="text" placeholder="Email address" />
          <tr>
            <td><label for="name">Your Name:</label></td>
            <td><input id="name" type="text" placeholder="Your name" /></td>
          </tr>
          <tr>
            <td><label for="password">Your Password:</label></td>
```

```
            <td><input id="password" type="password" /></td>
          </tr>
          <tr>
            <td><label for="passwordconfirm">Confirm Password:</label></td>
            <td><input id="passwordconfirm" type="password" /></td>
          </tr>
          <tr>
            <td colspan="2">
              <p><em>Would you like to subscribe to our monthly newsletter?</em><br />
                <input id="yes" type="radio" name="newsletter" value="yes" /><label for="yes">
                <input id="no" type="radio" name="newsletter" value="no" /><label for="no">No<
              </p>
            </td>
          </tr>
          <tr>
            <td colspan="2"><input type="submit" value="Submit" /> (using a regular input elem
          </tr>
          <tr>
            <td colspan="2"><button type="submit">Submit</button> (using a button element to s
          </tr>
      </tbody>
    </table>
  </form>
```

Now that we've had a quick refresher on forms and how to build one in HTML let's look at how we can process the form on the side with Express.js. The first thing you should know is that there are 2 types of form submissions that take place on a website regular text based form with normal inputs, and a form that accepts file uploads using the <input type="file"> element. A form supports file uploading requires the enctype attribute to be set to enctype="multipart/form-data". More information on optior 'enctype' attribute are available here

Processing Forms in Express.js

Sending data from a form to the Express.js server is not all that complicated. It's really just a few steps:

1. Create your form in HTML
2. Set the 'action' attribute of the form to a url where you want the form to be submitted
3. Set the method attribute of the form to 'GET' or 'POST' depending on how your server expects the request to come in. t you want to use POST.
4. Choose a correct enctype for the form based on the forms contents you'll be sending to the server
5. Add some middleware to the express server that can parse out the contents of the form into the body of the request o you can gain access to the data in the request object in Express (req.body)
6. Create a route in Express to receive the POST data from the form submission request and work with the data.
7. (Optional) Respond back to the client with the status of the submission if desired.

Let's do a complete example following those steps, from start to finish, for a simple form that let's you register a name, usern email, password, and a photo id. The form will upload the photo file and all the text contents of the form to the site, the site w the photo to a folder, return the data back with a success message and the photo url, or if it fails, an error message.

STEP 1: (CLIENT) CREATE THE FORM IN HTML

The first thing that we need is an html page with a form on it that we can submit to the server. Let's call it "registerUser.html" a place it in a "views" folder within the following project structure:

- /public
  - /photos
- /views
  - registerUser.html
- server.js

The photos that are uploaded will be saved in the /public/photos folder. registerUser.html is our form page and server.js is ou file.

The HTML page can look something like this:

```
<!doctype html>
<html>
  <head>
    <style>
      input {
        margin: 4px;
        width: 250px;
      }
    </style>
  </head>
  <body>
    <h1>Week 5 example</h1>
    <p>Register a new user:</p>
    <div style="text-align:right;width:400px;border:1px dashed #6495de;padding:16px;">
      <form>
        <label for="name">Name</label>
        <input id="name" type="text" name="name"/><br />
        <label for="username">Username</label>
        <input id="username" type="text" name="username"/><br />
        <label for="email">Email</label>
        <input id="email" type="email" name="email"/><br />
        <label for="password">Password</label>
        <input id="password" type="password" name="password"/><br />
        <label for="photo">Photo ID</label>
        <input id="photo" type="file" name="photo"/><br />
        <input type="submit" value="Submit File" />
      </form>
    </div>
  </body>
</html>
```

Now that we have the basic form laid out we are ready to set it up for submitting...

STEP 2: (CLIENT) SET THE 'ACTION' ATTRIBUTE ON THE FORM

The action attribute controls where the form will submit to. The value for the action is a url that you want the submission to g this example we will use the route /register-user.

Update the form element to look like this:

```
<form action="/register-user">
```

STEP 3: (CLIENT) SET THE PROPER VALUE FOR THE METHOD ATTRIBUTE ON THE FORM

The method attribute on the form control which HTTP verb it will use when submitting to the action route. For this example, a almost always, you'll want to POST. Set the method attribute to POST.

```
<form action="/register-user" method="POST">
```

STEP 4: (CLIENT) SET THE PROPER VALUE FOR THE ENCTYPE ATTRIBUTE ON THE FORM

Now we just have to make sure that when the form is submitted the server can know that we are including multipart data (The When you are just submitting text on a form the default is to submit as regular text. The enctype does not need to be set if yo just sending regular text.

Set the enctype to multipart/form-data to support the photo upload. If you do not do this the server side can not receive and the photo.

```
<form action="/register-user" method="POST" enctype="multipart/form-data">
```

STEP 5: (SERVER) SETUP SOME MIDDLEWARE TO PARSE THE FORM CONTENTS

We will use the library 'multer' to parse the multipart form data (The photo image). Multer requires a few options to be setup t able to name and save the files to the file system when they are uploaded. Then we need to add a multer middleware functio chain of functions that are called when a route matches. In express when a route handler matches a route, all the functions pr after the route string are called successively. IE: app.get("/", foo(), bar(), baz()); foo() will be called, then bar(), then baz(). So we multer function in front of the main route handler function to process the file upload and add it to the req object. The data wi req.body and the file on req.file

```
// setup our requires
const express = require("express");
const app = express();
const multer = require("multer");
const path = require("path");

const HTTP_PORT = process.env.PORT || 8080;

// call this function after the http server starts listening for requests
function onHttpStart() {
  console.log("Express http server listening on: " + HTTP_PORT);
}

// multer requires a few options to be setup to store files with file extensions
```

```
// multer requires a few options to be setup to store files with file extensions
// by default it won't store extensions for security reasons
const storage = multer.diskStorage({
  destination: "./public/photos/",
  filename: function (req, file, cb) {
    // we write the filename as the current date down to the millisecond
    // in a large web service this would possibly cause a problem if two people
    // uploaded an image at the exact same time. A better way would be to use GUID's for filen
    // this is a simple example.
    cb(null, Date.now() + path.extname(file.originalname));
  }
});

// tell multer to use the diskStorage function for naming files instead of the default.
const upload = multer({ storage: storage });
```

STEP 6: (SERVER) CREATE A ROUTE IN EXPRESS TO HANDLE THE FORM DATA ON THE 'REQ' OBJECT

Now we need to setup a static folder to serve the photos from when the browser requests them, setup a get and post route, a the app to listen for requests

```
// setup the static folder that static resources can load from
// we need this so that the photo can be loaded from the server
// by the browser after sending it
app.use(express.static("./public/"));

// setup a route on the 'root' of the url that has our form
// IE: http://localhost/
app.get("/", (req, res) => {
  // send the html view with our form to the client
  res.sendFile(path.join(__dirname, "/views/registerUser.html"));
});

// now add a route that we can POST the form data to
// IE: http://localhost/register-user
// add the middleware function (upload.single("photo")) for multer to process the file upload
// the string you pass the single() function is the value of the
// 'name' attribute on the form for the file input element
app.post("/register-user", upload.single("photo"), (req, res) => {
  res.send("register");
});

app.listen(HTTP_PORT, onHttpStart);
```

STEP 7: (SERVER) RESPOND TO THE CLIENT WITH DATA

Now that the server is setup we can tailor the response that we send back after posting the form data to the route. We will ser the json structure of the form data and the form file as well as display the image that was uploaded and saved.

Modify the register-user route handler code to now look like this:

```
app.post("/register-user", upload.single("photo"), (req, res) => {
  const formData = req.body;
  const formFile = req.file;

  const dataReceived = "Your submission was received:<br/><br/>" +
    "Your form data was:<br/>" + JSON.stringify(formData) + "<br/><br/>" +
    "Your File data was:<br/>" + JSON.stringify(formFile) +
    "<br/><p>This is the image you sent:<br/><img src='/photos/" + formFile.filename + "'/>";
  res.send(dataReceived);
});
```

FINAL NOTES

The multer library is only needed when you are dealing with file uploads and multipart/form-data. If you are simply using text data you can use the body-parser library to handle regular text submissions and access the data on req.body. You'll need to ac body-parser middleware with an app.use() call to allow it to grab the data on the request and stick it onto the req.body proper available to the route handler function. To set this up properly, we must:

1. Download the "body-parser" module via npm:

   ```
   npm install body-parser --save
   ```

2. Add the "body-parser" dependency using the require() function:

   ```
   const bodyParser = require('body-parser');
   ```

3. Near the top of your server code, make a call to app.use() to set the middleware for "urlencoded" form data (normal HT data):

   ```
   app.use(bodyParser.urlencoded({ extended: true }));
   ```

Anytime you have a client side form that send back data and/or file contents, you should always validate both client side **AND** side: For example, it is important to validate the form on the client side so you can create a better user experience and inform if something is missing or a wrong filetype is provided right away. If a field is missing or an incorrect data type is provided, you the user know before you submit the form and wipe out the user's progress.

On the server side it is even **more** important to validate everything the client has sent. A malicious user can use a command li like cUrl to make api requests to your server and send anything they want, even if your client side code validates the data befc sent. Another method is to use the developer tools in the browser to remove the client side validation and then submit anythi your server. Therefore, it is extremely important to validate data on the server side before working with it – especially in a pro environment.

environment.
