Regex, Express, and Full-Stack Development (Week 11) - Learning Objectives

Assessment Structure

- 2 hours, 45 minutes
- Mixture of multiple choice (5-10), and VSCode problems (20-30 specs).
    - For the coding section, be comfortable with creating an express application from scratch. The projects from Thursday and Friday are good examples (on a smaller scale for an assessment environment).
    - This includes connecting database material introduced last week, such as making your database, initializing sequelize, creating migrations and models, adding seeders for your models, etc.
    - Be able to work within the express framework, creating the app, adding in middleware for csrf protection, urlencoded body parsing, etc., making route handlers to respond to various requests and pull in database information, etc.
    - Be able to display that information to the user using Pug templates, including csrf-protected forms for post routes, iterating over database data to display content of records, etc.
- Standard assessment procedures
    - You will be in an individual breakout room
    - Use a single monitor and share your screen
    - Only have open those resources needed to complete the assessment:
        - Zoom
        - VSCode
        - Browser with AAO and Progress Tracker (to ask questions)
        - Approved Resources for this assessment:
            - Express Docs: http://expressjs.com/
            - Pug Docs: http://pugjs.org/
            - csurf Docs: https://github.com/expressjs/csurf#readme
            - Sequelize Docs: https://sequelize.org/
            - Sequelize "Cheatsheet"

Regular Expressions and Node HTTP (W11D1) - Learning Objectives

Regular Expressions

1. Define the effect of the * operator and use it in a regular expression
2. Define the effect of the ? operator and use it in a regular expression
3. Define the effect of the + operator and use it in a regular expression
4. Define the effect of the . operator and use it in a regular expression
5. Define the effect of the ^ operator and use it in a regular expression
6. Define the effect of the $ operator and use it in a regular expression
7. Define the effect of the [] bracket expression and use it in a regular expression
8. Define the effect of the - inside brackets and use it in a regular expression
9. Define the effect of the ^ inside brackets and use it in a regular expression

Regex cheatsheet:

Regex Symbol	Meaning
*	zero or more
+	one or more
?	zero or one
{m, n}	from m to n number of characters
^	start of input
$	end of input
.	any single character
\	escape a special character
[]	match anything inside (or)
[a-z] or [0-9]	range of characters
[^a-za-z]	not those characters
()	group these characters
\s	whitespace
\d	digit
\w	wordy (letter, digit, or _)
\S	not whitespace
\D	not digit
\W	not wordy

Using Regex in JavaScript - aka How Do I Apply This Stuff?

- To create a regular expression we use / at the beginning and end of the pattern

```
const regex = /pattern/;
```

- Two flags that we can use at the end of our regex to make them even more powerful are `i` and `g`
  - `i` indicates that we want our regex to be case-insensitive. We can accomplish the same thing by expanding our pattern to include either case (`/[Hh][Ii]/` instead of `hi`, for example), but this is a convenient shorthand
  - `g` indicates that we want the regex to apply globally, meaning catch all matches, not just the first one. This is especially useful when we use regex in a `.replace()` function, indicating we want to change all matches.

```
```

```
const regex = /pattern/gi;
```

- We can use a regex's `.test()` method to see if a match is found within a string argument.

```
const pattern = /pattern/;

console.log(pattern.test('this pattern is plaid')); // true
console.log(pattern.test('THIS PATTERN IS PLAID')); // false (case-
sensitive)
```

- We can use a string's `.replace()` method and pass in regex as the first argument in order to replace many instances of our matching pattern, no matter how complicated.

```
const s = 'An Advancing Aardvark';
const replaced = s.replace(/a/gi, 'X');
console.log(replaced); // Xn XdvXncing XXrdvXrk
```

- The `.replace()` method can also take a callback function. The return value of the callback is what will be replaced. This allows us to change the value that is being inserted dynamically.

```
let index = 0;
const s = 'An Advancing Aardvark';
const replaced = s.replace(/a/gi, (match) => {
  index += 1;
  return index;
});
console.log(replaced); // 1n 2dv3ncing 45rdv6rk
```

- The `.search()` method of a string can take regex as an argument instead of a string, as well. It returns the starting index of the match

```
let test = ['this', 'that', 'the other', 'and this?'];
let indices = test.map(function (e) {
  // search is returning the index that starts the match, or -1 if not
found
  return e.search(/(this)|(that)/); // [0, 0, -1, 4]
});
console.log(indices);
```

HTTP Full-Stack

1. Identify the five parts of a URL

```
  foo://example.com:8042/over/there?name=ferret#nose
  \_/   \______________/\_________/ \_________/ \__/
   |           |            |            |        |
scheme     authority       path        query   fragment
```

- `Scheme`: Formerly known as "protocol", tells the browser what kind of connection you are making. Examples would include `http`, `https`, `file`, `ws`, etc.
- `Authority`: Normally the domain name, but may include the port, such as `localhost:3000`
- `Path`: The first part of an HTTP request, they indicate the location within the application. `https://google.com/images` has a path of `/images`. If no path is specified in the URL, it is assumed to be `/`
- `Query`: Extra information sent to the browser for processing the request. They are URL encoded key-value pairs, with an `=` between key and value, and pairs concatenated by `&`. Special characters are replaed with ASCII Code value, such as `$` being replaced with `%24`.
- `Fragment`: This part is not sent to the server; it's used by the browser to navigate to a specific section of the page on load. Often seen when clicking a navigation link that takes to you a spot further down the page. Changing this value will not reload the page

2. Identify at least three protocols handled by the browser

- See above. `http`, `https`, and `file` have all been used in this course so far.

3. Use an `IncomingMessage` (typically called the `req`) object to

- access the headers sent by a client (like a Web browser) as part of the HTTP request

```
const headers = req.headers;
// The headers are represented as a POJO with header names as keys
// { 'user-agent': 'curl/7.22.0',
//   host: '127.0.0.1:8000',
//   accept: '*/*' }
```

- access the HTTP method of the request

```
const method = req.method;
// The headers are represented as a string such as 'GET' or 'POST'
```

- access the path of the request
    - If we want to work with the full path we can use `req.url`

```
const path = req.url; // gives full path of request, such as '/images'
```

- If we want to manipulate the path, we can use the `path` library, as well. This can be useful if we want to check the extension name that is being requested.

```
const path = require('path');

const server = http.createServer(async (req, res) => {
  const ext = path.extname(req.url);

  if (ext === '.jpg') {
    // ...
  }
});
```

- access and read the stream of content for requests that have a body
  - The body of a request comes in to the server in pieces, or "chunks". We'll see this a lot when we make a `POST` request with a form. We can wait for the body to arrive and build it back up:

```
if (req.method === 'POST') {
  let bodyContent = '';
  for await (let chunk of req) {
    bodyContent += chunk;
  }
```

  - At this point the `bodyContent` is in the form `key1=encoded+value+1&key2=encoded+value+2`, etc., where key/value pairs are joined with `=`, pairs are separated with `&` and the values supplied use `+` for spaces and url encoding for special characters. We can split and map over this string in order to convert it into a more user-friendly POJO (this doesn't need to be memorized, but the process should make sense)

```
// bodyContent = 'my-input=In+this+box%3&another-
input=Yes%2C+this+box%2C+here.'

const keyValuePairs = bodyContent
  .split('&') // ['my-input=In+this+box%3', 'another-
input=Yes%2C+this+box%2C+here.']
  .map((keyValuePair) => keyValuePair.split('=')) // [['my-input',
'In+this+box%3'], ['another-input', 'Yes%2C+this+box%2C+here.']]
  .map(([key, value]) => [key, value.replace(/\+/g, ' ')]) // [['my-
input', 'In this box%3'], ['another-input', 'Yes%2C this box%2C
here.']]
  .map(([key, value]) => [key, decodeURIComponent(value)]) // [['my-
input', 'In this box?'], ['another-input', 'Yes, this box, here.']]
  .reduce((acc, [key, value]) => {
    acc[key] = value; // { 'my-input': 'In this box?', 'another-
input': 'Yes, this box, here.' }
    return acc; // We have to return acc to make sure we use the
```

```
updated value for the next iteration (adding in new key/value pairs
for each iteration)
  }, {}); // This {} is the starting value of acc (the accumulator).
It's allowing us to make a key/value pair for each inner array that we
are destructuring.

// keyValuePairs = { 'my-input': 'In this box?', 'another-input':
'Yes, this box, here.' }
```

4. Use a `ServerResponse` (typically called the `res`) object to

- write the status code, message, and headers for an HTTP response

```
res.statusCode = 400; // We can set the statusCode directly through
assignment
res.statusMessage = "That password doesn't match"; // We can specify a
custom status message if it differs from the default related to the code
res.setHeader('Content-Type', 'text/plain'); // .setHeader(<<headerName>>,
<<headerContent>>)
```

- write the content of the body of the response

```
// If we have a single string to pass as the content we can give it as the
argument directly to .end()
res.end('NOT FOUND');

// instead of using .end('message'), we can build up our response with
.write('messagePart')
for (let i = 1; i <= 10; i++) {
  res.write(`<p>This is p-tag #${i}</p>`);
}
res.end();
```

- properly end the response to indicate to the client (like a Web browser) that all content has been written
    - As above, we need to include `.end()` to indicate that the content is written and can be sent back to the client. If no arguments are given, it will send what has been built up with `.write()` (if anything) as the content. If an argument is given, it will add it on before sending back that content. (A common approach would be to build up a `content` variable in the javascript above, then pass the built up result to end, with `res.end(content)`)

Express and Pug Templates (W11D2) - Learning Objectives

Express

1. Send plain text responses for any HTTP request

- In order to set up express in our application, we need to package with `npm install express`

- With express installed, we can create a file that we will use to kick off our application. You'll often see this as `app.js` or `index.js` at the top level of your server, but can be customized for the individual project.
- In our `app.js`, we can require the express package and invoke it to create our server application:

```
const express = require('express');

// Create the Express app.
const app = express();
```

- The `app` object that we created here is able to respond to incoming requests by defining routes with these helpful functions:
    - `get()` responds to HTTP GET requests
    - `post()` responds to HTTP POST requests
    - `put()` responds to HTTP PUT requests
    - `delete()` responds to HTTP DELETE requests
- Each of these functions accepts two arguments: the route as a string or regex, as well as a callback function to define what we would like to do with the request. The format generally takes the form:
    - `app.get({path}, (req, res) => {functionality})`
- To start our app listening for traffic on a specific port, we can us the `app.listen({portNumber}, {successCallback})` function of our `app` object. We often use the callback just as a confirmation that the app started up successfully.

```
const port = 8081;

app.listen(port, () => console.log(`Listening on port ${port}...`));
```

- Now that the app is set up and we know how to create a route, responding with plain text is as easy as using the `res.send({text response})` method inside the route.

```
app.get('/', (req, res) => {
  res.send('Hello from Express!');
});
```

2. Use pattern matching to match HTTP request paths to route handlers

- When defining routes, express offers us some flexibility in a couple different ways.
    - We are able to capture parameters from our routes and set up restrictions on what characters will match these parameters.

```
// Capturing a parameter
app.get('/product/:id', (req, res) => {
  res.send(`Product ID: ${req.params.id}`);
});
```

```
// Putting a restriction on the parameter
// This route will only be used if what follows the /product/ is a
series of numbers (the + allows for multiple characters)
// We still capture it as a string, so it needs to be parsed into a
number if it is going to be used as such in our function
app.get('/product/:id(\\d+)', (req, res) => {
  const productId = parseInt(req.params.id, 10);
  res.send(`Product ID: ${productId}`);
});
```

- We can set up a string pattern, which is a similar concept to regex. Express sees the special characters in our string and is able to interpret patterns that match those characters.
  - The `?` indicates that the previous character is optional (either 0 or 1 instances will occur)
  - The `+` indicates that the previous character may be repeated (at least 1, but possibly more instances will occur)
  - The `*` indicates a wildcard. Any character and any number of characters will match.

```
// Within the string, we can use special characters to define
patterns, similar to regex.
// The pattern below can start will any base string (because of the
`*`), then must have `prod`, has an optional `u`, must have a `c`, has
at least one but possibly more `t` characters, then ends with an
optional `s`.
app.get('/*produ?ct+s?', (req, res) => {
  res.send('Products');
});
```

- We can use actual regex to define a pattern.

```
// Regex allows us to be much more specific and complex with the
patterns that we are matching.
// Don't worry if this looks crazy! You would very rarely need to
construct anything so complex, this is just showing the possibilities.
app.get(/^\/((our-)?produ?ct{1,2}s?|productos)\/?$/i, (req, res) => {
  res.send('Products');
});
```

- We can use any combination of these in an array to provide multiple options to match.

```
// Our array of options can have a mixture of the previous pattern
definitions, as well as multiple options of each.
app.get([/^\/(our-)?produ?ct{1,2}s?\/?$/i, '/productos'], (req, res)
=> {
  res.send('Products');
});
```

3. Use the Pug template engine to generate HTML from Pug templates to send to the browser

- To use pug templates in our application we can install the package with `npm install pug@^2.0.0` to install pug 2.0.
- With pug in our application, we can tell express to use it as the view engine:

```
app.set('view engine', 'pug');
```

- Instead of rendering plain text with `send`, we can render a template by using `res.render({templateName})`.
- By default, express will look in a `views` directory for each template. We can set up this folder with a `{templateName}.pug` file inside that will be rendered.

```
// Define a route.
app.all('*', (req, res) => {
  console.log(`Request method: ${req.method}`);
  console.log(`Request path: ${req.path}`);

  res.render('layout');
});
```

- In that `layout.pug` file used in the above demo route, we can start setting up our template.
    - The names of tags we would like to create are written out as plain text.
    - To nest tags, we use indentation. Spaces vs. tabs do not matter as long as we are consistent in our files.
    - The content of the tag can be written directly after the tag name

```
html
  head
    title My Awesome Title
  body
    h1 My Super Cool Heading
```

4. Pass data to Pug templates to generate dynamic content

- The `render()` method on our `app` can also take in a second argument with an object defininining variables to make available within our Pug templates.

```
app.all('*', (req, res) => {
  console.log(`Request method: ${req.method}`);
  console.log(`Request path: ${req.path}`);

  res.render('layout', { title: 'Pug Template Syntax Sandbox', heading:
```

```
'Welcome to the Sandbox!' });
});
```

- We would now be able to utilize the variables in the template file.
    - To use the value directly as content we can use the `=` operator for the element
    - If we want to interpolate the value within a larger string we can use Pug interpolation, which takes the form `My interpolated value is #{variableName}`

```
html
  head
    title= title
  body
    h1 My heading says #{heading}
```

5. Use the `Router` class to modularize the definition of routes

- In addition to defining routes directly on the `app` instance that we created, we can create multiple routers from the `express.Router()` function and then utilize them from different base routes.
- This functionality supports the modularity of our app. We'll often want to break up our routes into categories, such as routes for our users vs routes for our tweets, instead of having all of our routes defined in one location.
- When we have an instance of the `Router` we can define routes with the same `get()`, `post()`, `put()`, and `delete()` methods.

```
// In our main app.js file
const express = require('express');
const userRoutes = require('./routes/users.js'); // see following code
block
const tweetRoutes = require('./routes/tweets.js'); // see following code
block

const app = express();

app.use('/users', userRoutes);
app.use('/tweets', tweetRoutes);

const port = 4000;

app.listen(port, () => {
  console.log(`Listening on port ${port}...`);
});
```

```
// In a users route file
const express = require('express');

const router = express.Router();
```

```
router.get('/' (req, res) => {
  res.send('Hello from the base user route. We got here from /users/');
})

router.get('/:id' (req, res) => {
  res.send(`This is the page for user with id ${req.params.id}. We got
here from /users/:id`);
})

// other routes, such as editing a profile, deleting a user, etc.

module.exports = router;
```

```
// In a tweets route file
const express = require('express');

const router = express.Router();

router.get('/' (req, res) => {
  res.send('Hello from the base tweets route. We got here from /tweets/');
})

router.get('/:id' (req, res) => {
  res.send(`This is the page for tweet with id ${req.params.id}. We got
here from /tweets/:id`);
})

// other routes, such as posting a tweet, editing, deleting, etc.

module.exports = router;
```

Pug Templates

1. Declare HTML tags and their associated ids, classes, attributes, and content

- To declare a tag, we can use the name of the tag directly in the pug file.
- Nesting tags can be accomplished by indenting, with either spaces or tabs (just be consistent!)
- Ids and classes can be added by attaching them to the tag name, just like they are selected in a CSS file.
- Attributes can be given to a tag (like an href for an a tag, a src for an image, etc.) by supplying their values in parentheses:

```
html
  head
    title My Page
  body
    div#main
      div.blue
```

```
      div.yellow
        a(href="http://google.com") Click here
```

2. Use conditional statements to determine whether or not to render a block

- Inside of our Pug files we can include some logic, allowing our template to be even more dynamic.
- An `if` statement can be used with the condition directly following. Anything indented will only be included if the condition is true. We can similarly include an `else` as part of the condition:

```
if isEOD
  h2 Welcome back!
else
  h2 Keep coding!

if (time > 17)
  p See ya tomorrow!
```

3. Use interpolation to mix static text and dynamic values in content and attributes

- Whenever we want to interpolate a value inside our pug template's content, we can use the `#{}` syntax.
- It's important to distinguish this interpolation from JavaScript string interpolation. If we were to provide strings within this template that we needed to interpolate within, such as an href attribute, we are working with JavaScript at that point and would need to use our standard `${}` within backticks to interpolate those values:

```
res.render('layout', {
  title: 'Pug demo page',
  header: 'interpolation',
  route: 'tweets',
});
```

```
html
  head
    title= title
    style
      include style.css
  body
    h1 Pug does #{header}
    h2 Pug allows you to do many things
    ul
      li: a(href='http://google.com') This is a link to google.com
      li: a(href=`http://mycoolsite.com/${route}`) This is a link to my
cool site's #{route} route
```

4. Use iteration to generate multiple blocks of HTML based on data provided to the template

- We can iterate through variables within our pug templates using the `each ... in ...` syntax.

```
// app.js
app.get('/pug', (req, res) => {
  res.render('eod', { colors: ['blue', 'red', 'green'] });
});
```

```
ul
  each color in colors
    li= color
```

HTML Forms (W11D3) - Learning Objectives

HTML Forms

1. Describe the interaction between the client and server when an HTML form is loaded into the browser, the user submits it, and the server processes it

- Below is an example of a form in regular HTML for our reference (not Pug).

```
<form action="/users" method="post">
  <label
    >Username:
    <input type="text" name="user[username]" />
  </label>
  <label
    >Email:
    <input type="email" name="user[email]" />
  </label>
  <label
    >Age:
    <input type="number" name="user[age]" />
  </label>
  <label
    >Password:
    <input type="password" name="user[password]" />
  </label>
  <input type="submit" value="Sign Up" />
</form>
```

- The `action` attribute of the form element defines the url that the request is made to, ie what route our request is going to hit on our server.
    - We can use multiple formats for this url:
        - absolute URL: A complete url path such as https://www.wellsfargo.com/transfers
        - relative URL: Just providing a route will send the request to the same domain that we are one. Using `/users` when we are running on localhost in development will send the request

        to localhost:3000/users
      - no URL - form will be sent to the same page(url) the form is present on
- The `method` attribute defines the HTTP verb that will be used with the request ('GET' or 'POST')
  - If the `POST` method is used, form data is appended to the body of the HTTP request.
  - Only the `GET` and `POST` methods may be specified on the form. In order to do `PUT`, `DELETE`, or other methods, we must use `AJAX` requests using the `fetch` API, for example.
- The server receives a string that will be parsed in order to get the data as a list of key/value pairs.
  - The `name` attribute of each input is used as the key, with the value of the input used value in the key/value pair.
  - We can create nested objects within the parsed data by providing a name format `outerKey[innerKey]`. When this is parsed by express (see LO 3 and 4 below), we end up with a a nested `req.body`. From the example above we would see:

```
console.log(req.body);
/*
{
  user: {
    username: _value_,
    email: _value_,
    age: _value_,
    password: _value_
  }
}
*/
```

  - This could be extra helpful if we have multiple categories of information in the form we are submitting and want to have a way to easily distinguish between them.

2. Create an HTML form using the Pug template engine

- We can translate the above form into Pug fairly directly.
- Any attributes that we would include on an HTML tag would be included in parentheses in the template, with nested tags being indented.
- We've added some functionality in this example by including the hidden input for our csrf token (See LO 8) as well as providing values for our inputs. The values will help us retain content when a user sends an incorrectly filled out form instead of sending back a blank form and losing their input. Notice no `value` is used for password because we always want the user to type in that information.

```
form(method="post" action="/users")
  input(type="hidden" name="_csrf" value=csrfToken)
  label(for="user[username]") Username:
    input(type="username" id="username" name="username" value=username)
  label(for="user[email]") Email:
    input(type="email" id="email" name="email" value=email)
  label(for="user[age]") Age:
    input(type="age" id="age" name="user[age]" value=age)
  label(for="user[password]") Password:
```

```
    input(type="password" id="password" name="user[password]")
  input(type="submit" value="Sign Up")
```

3. Use express to handle a form's POST request

- In the 'POST' route, we can check that our user's input passes all of our data validation checks, then, if successful, create a new record for that user within our database. (In the project for today we just added an object into an array to simulate this process.)
- After we create our new record, we can redirect the user to another page. In this example, we are redirecting to our root page, but we could have redirected to a user page, an index, a welcome page, etc.

```
app.get('/create', csrfProtection, (req, res, next) => {
  res.render('create', {
    title: 'Create a user',
    errors: [],
    csrfToken: req.csrfToken(),
  });
});

app.post(`/users`, csrfProtection, checkFields, async (req, res) => {
  const { username, email, age, password } = req.body.user;

  // Our errors attribute was created by our checkFields middleware
  // If we had errors, we're rendering the 'create' form again, passing
along the errors as well as the user data so that we can prepopulate those
fields with the values that were originally submitted
  if (req.errors.length >= 1) {
    res.render(`create`, {
      title: 'Create a user',
      errors: req.errors,
      username,
      email,
      age,
      csrfToken: req.csrfToken(),
    });
    return;
  }

  await User.create({ username, email, age, password });

  res.redirect(`/`);
});
```

4. Use the built-in `express.urlencoded()` middleware function to parse incoming request body form data

- The express middleware `urlencoded` comes with the express library, we just need to tell our app to use it on all requests.
- Invoking `app.use({middlewareFunction})` will set up all requests to pass through this function.

- The `urlencoded` function will decode the body of our form post requests, allowing our routes to access the `req.body` property with each field as a property on this object.
- The `extended: true` is a newer parsing library compared to leaving it off or false, which allows for objects and arrays to be encoded. We'll always want to use this (you'll most likely see warnings in your terminal if you leave it off).

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

5. Explain what data validation is and why it's necessary for the server to validate incoming data

- Data validation is the process of ensuring that the incoming data is correct.
- This could mean anything related to the format or content of the data such as:
    - the type of data (is this actually a number?)
    - the value makes sense (is the age a positive number?)
    - length of content (is the password at least 8 characters long?)
    - etc.
- Even though you could add add validations on the client side, client-side validations are not as secure and can be circumvented. We could send a postman request to our server's routes, for example, and not utilize our client-side application at all.
- Because client-side validations can be circumvented, it's necessary to implement server-side data validations as well.
- Handling bad request data in our route handlers allows us to return 400 level messages with appropriate messaging to allow the user to correct their bad request.
- If we didn't have data validations on our server and allowed the bad data to go all the way to the database, the system would return a the generic "500: Internal Server Error", which is not helpful for the user.

6. Validate user-provided data from within an Express route handler function

- Before we attempt to create a record for this new user, we can verify that the information submitted by the user exists, is in a valid format, etc.
- If anything is outside of our expectation, we can handle the request differently (rendering the form again, displaying errors, etc.) as opposed to if the information is valid (where we would typically want to redirect the user to a new page).

```
app.post('/create', csrfProtection, async (req, res) => {
  const { username, email, age, password, confirmedPassword } = req.body;
  const errors = [];

  if (!username) {
    errors.push('Please provide a username.');
  }
```

```
  if (!email) {
    errors.push('Please provide an email.');
  }

  if (!age) {
    errors.push('Please provide an age.');
  }

  const ageAsNum = Number.parseInt(age, 10);

  if (age && (ageAsNum < 0 || ageAsNum > 120)) {
    errors.push('Please provide an age between 0 and 120');
  }

  if (!password) {
    errors.push('Please provide a password.');
  }

  if (password && password !== confirmedPassword) {
    errors.push('The provided values for the password and password
confirmation fields did not match.');
  }

  if (errors.length > 0) {
    res.render('create', {
      title: 'Create a user',
      username,
      email,
      age,
      csrfToken: req.csrfToken(),
      errors,
    });
    return;
  }

  const newUser = await User.create({ username, email, age, password });
  res.redirect(`/user/${newUser.id}`);
});
```

7. Write a custom middleware function that validates user-provided data

- Separating out the validation of our inputs into its own middleware allows us to reuse this functionality for multiple routes.
- When writing middleware, we take in the request, response, and a reference to the `next` function, which express is going to utilize to set up our chain of functions to invoke.
- Our middleware can interact with and modify our request and response objects however we like. For validating user input, it's convenient for us to be able to add in custom error messages to our request object if any of the data does not match our expectations. That way our routes can check to see if any of these errors occurred and respond appropriately.
- After implementing the functionality of our middleware, we invoke the `next` function in order to continue the middleware chain. I like to think of this as similar to invoking the `resolve` function of a Promise

when the functionality we are implementing is complete.

```
const validationMiddleware = (req, res, next) => {
  const { username, email, age, password, confirmedPassword } = req.body;
  const ageAsNum = Number.parseInt(age, 10);
  const errors = [];

  if (!username) {
    errors.push('Please provide a username.');
  }

  if (!email) {
    errors.push('Please provide an email.');
  }

  if (!age) {
    errors.push('Please provide an age.');
  }

  if (age && (ageAsNum < 0 || ageAsNum > 120)) {
    errors.push('Please provide an age between 0 and 120');
  }

  if (!password) {
    errors.push('Please provide a password.');
  }

  if (password && password !== confirmedPassword) {
    errors.push('The provided values for the password and password
confirmation fields did not match.');
  }

  req.errors = errors;
  next();
};
```

- With this function written, we can pass it as middleware to any route that requires it.
- With the validations being run, we can then interact with the `req.errors` array that our middleware created.

```
app.post('/create', csrfProtection, validationMiddleware, async (req, res)
=> {
  const { firstName, lastName, email, password, confirmedPassword } =
req.body;
  const errors = req.errors;

  if (errors.length > 0) {
    res.render('create', {
      title: 'Create a user',
      username,
```

```
      email,
      age,
      csrfToken: req.csrfToken(),
      errors,
    });
    return;
  }

  await User.create({ username, email, age, password });
  res.redirect('/');
});
```

8. Use the csurf middleware to embed a token value in forms to protect against Cross-Site Request Forgery exploits

- To set up CSRF protection, we want to utilize two tools: the `csurf` library and the `cookie-parser` library (since we'll use cookies for our csrf protection implementation)
- Once pulled in to our main file, we can create a csrfProtection middleware as well as tell our app to use the cookie parser:

```
app.use(cookieParser());
const csrfProtection = csrf({ cookie: true });
```

- Creating the middleware function is not enough to implement csrf protection. Anywhere that requires us to create or submit forms we need to utilize the middleware within the route. We do this by passing it as an argument to our `app.get` or `app.post` function calls.
- When creating our forms, we want to be able to provide a `_csrf` input field, hidden from the user.
- The value that we supply to this field is obtained from invoking our `req.csrfToken()` function that was created for us from the middleware.
- The middleware is also supplied to the 'POST' routes that respond to these submissions in order to check that the token was actually created by our application.

```
const express = require('express');
const cookieParser = require('cookie-parser');
const csrf = require('csurf');

const { User } = require('./models');

const app = express();

const port = process.env.PORT || 3000;

const csrfProtection = csrf({ cookie: true });

app.use(cookieParser());
app.use(express.urlencoded({ extended: true }));
app.set('view engine', 'pug');
```

```
app.get('/create', csrfProtection, (req, res, next) => {
  res.render('create', {
    title: 'Create a user',
    messages: [],
    csrfToken: req.csrfToken(),
  });
});

app.post('/create', csrfProtection, validationMiddleware, async (req, res)
=> {
  const { username, email, password } = req.body;
  const errors = req.errors;

  if (errors.length > 0) {
    res.render('create', {
      title: 'Create a user',
      username,
      email,
      age,
      csrfToken: req.csrfToken(),
      errors,
    });
    return;
  }

  await User.create({ username, email, age, password });
  res.redirect('/');
});
```

- The Pug files that create thes forms can simply create a hidden input field:

```
// Other content before the form
form(action='/create' method='post')
  input(type='hidden' name="_csrf" value=csrfToken)
  div(class="form-group")
    label(for='username') Username:
    input(id='username' class="form-control" name='username'
value=username type='text')
  // Other form fields
  div
    input(type='submit' value="Create User" class='btn btn-primary')
```

Full-Stack (Data-Driven Web Sites) (W11D4) - Learning Objectives

Data-Driven Web Sites

1. Use environment variables to specify configuration of or provide sensitive information for your code

- We can specify environment variables in the command line or script by setting them before the command we are running:

    - `PORT=8080 node app.js` in the terminal
    - `"start": "PORT=8080 node app.js"` in our scripts
- In our app, we can use `process.env.VARIABLE_NAME` to access any environment variables that have been set.
- We can also create a `.env` file to store all of our variables in one location. In order to access these variables, we use the `dotenv` package (see LO #2)

```
PORT=8080
DB_NAME=cool_app
DB_USER=cool_app_user
DB_PASSWORD=cool_app_user_password
```

2. Use the `dotenv` npm package to load environment variables defined in an `.env` file

- Adding the `dotenv` package to our project gives us three ways to load environment variables.
    1. In our app, we can require the `dotenv` package and invoke its `configure` method. Any point after this configuration will have access to environment variables defined in `.env`:

```
require('dotenv').config();
```

    2. When starting our application, either in the terminal directly or in a script, we can use `node` or `nodemon`'s `-r` flag to indicate we want to require a package at the very start. By requiring `dotenv/config`, we can omit the line above from our application code.
    - `"start": "nodemon -r dotenv/config app.js"`
    3. Similar to how `sequelize-cli` allows us to run sequelize commands in the command line, we can install `dotenv-cli` to our project with `npm install dotenv-cli` and then use `dotenv` before our commands that need to utilize these variables:
    - `npx dotenv sequelize-cli db:migrate`

3. Recall that Express cannot process unhandled Promise rejections from within route handler (or middleware) functions.

- If an error is thrown in a synchronous route handler or middleware, it will be caught by express error handlers and an error message will be sent back to the client.
- For asynchronous functions, unhandled rejected promises (errors not caught by a `catch` block or `catch` method in a promise chain) will not be handled by the error handlers. The server will not send a response back and the client's browser will hang. We need to catch the error and pass it along to the error handlers in order for the standard error-handling functionality to occur.

4. Use a Promise catch block or a `try`/`catch` statement with `async`/`await` to properly handle errors thrown from within an asynchronous route handler (or middleware) function

- To catch asynchronous errors, we've seen that we can use a `try`/`catch` block when using the `async`/`await` syntax, or we can chain a `.catch` on to a Promise directly.
- With express, we can indicate that an error has occurred by invoking the `next` function with an argument. The argument is the error that has occurred from our asynchronous code.

- Using our `try`/`catch` syntax:

```
app.get('*', async (req, res, next) => {
  // we specify an async function and capture the 'next' parameter
  try {
    const result = await someAsynchronousFunction();
    res.send(result);
  } catch (err) {
    // If an error occurs in the above try block, it is captured as err
    next(err); // The err variable that we captured is passed to next so
that error handlers can interact with it
  }
});
```

- Using our Promise chains:

```
app.get('*', async (req, res, next) => {
  someAsyncFunction()
    .then(() => {
      // Assume some command is here that throws an error
    })
    .catch((err) => {
      // We catch it here to make express happy
      next(err);
    });
});
```

5. Write a wrapper function to simplify catching errors thrown within asynchronous route handler (or middleware) functions

- The process of wrapping our asynchronous handlers in `try`/`catch` blocks is so common that we will often create a helper wrapper function in order to DRY up our code. It also makes the route handlers that we create look more similar to the synchronous ones that we are used to:

```
const asyncHandler = (handler) => {
  return (req, res, next) => {
    return handler(req, res, next).catch((err) => next(err));
  };
};

// We are utilizing ES6 implicit returns of single-line => functions
to shrink this to one line.
// This can be a little confusing, so you'll still often see the above
implementatin.
// THIS IS THE SAME AS THE ABOVE FUNCTION! Just a different structure.
const asyncHandler = (handler) => (req, res, next) => handler(req,
res, next).catch(next);
```

```
// By wrapping our normal route handler in the custom asyncHandler
that we created, we don't have to worry about writing try/catch blocks
or chaining .catch onto promises for all of our different routes
app.get(
  '*',
  asyncHandler(async (req, res) => {
    // our asyncHandler is returning the function that invokes our
handler defined here, with the addition of the catch method and
invocation of next(err) if an error occurs
    const result = await someAsynchronousFunction();
    res.send(result);
  })
);
```

6. Use the `morgan` npm package to log requests to the terminal window to assist with auditing and debugging

- Adding the `morgan` package lets us log the requests that hit our server. The `dev` format specifies the HTTP method/verb, the path, status code, response time, and content length of the response.
- `npm install morgan`

```
const morgan = require('morgan');
app.use(morgan('dev'));
```

7. Add support for the Bootstrap front-end component library to a Pug layout template

- Adding in the Bootstrap stylesheet and script to our template allows us to use classes that Bootstrap has defined for us with default styles. These are lines that you would copy over from Bootstrap template, not things that you would memorize https://getbootstrap.com/docs/4.4/getting-started/introduction/#starter-template
- The important takeaway is knowing that these lines are what are allowing us to use the classes Bootstrap has defined for us.

```
  doctype html
  html
    head
      meta(charset='utf-8')
      meta(name='viewport' content='width=device-width, initial-scale=1,
shrink-to-fit=no')

      // The following line is importing the bootstrap css file
      link(rel='stylesheet'
href='https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min
.css' integrity='sha384-
Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh'
crossorigin='anonymous')

      title Reading List - #{title}
```

```
    body
      nav(class='navbar navbar-expand-lg navbar-dark bg-primary')
        a(class='navbar-brand' href='/') Reading List
      .container
        h2(class='py-4') #{title}
        block content

      // The following lines are importing the bootstrap js files
      script(src='https://code.jquery.com/jquery-3.4.1.slim.min.js'
integrity='sha384-
J6qa4849blE2+poT4WnyKhv5vZF5SrPo0iEjwBvKU7imGFAV0wwj1yYfoRSJoZ+n'
crossorigin='anonymous')

script(src='https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.
min.js' integrity='sha384-
Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo'
crossorigin='anonymous')

script(src='https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/bootstra
p.min.js' integrity='sha384-
wfSDF2E50Y2D1uUdj0O3uMBJnjuUD4Ih7YwaYd1iqfktj0Uod8GCExl3Og8ifwB6'
crossorigin='anonymous')
```

8. Install and configure Sequelize within an Express application.

- We need to install packages associated with sequelize and postgres:

```
npm install sequelize@^5.0.0 pg@^8.0.0
npm install sequelize-cli@^5.0.0 --save-dev
```

- Creating a `.sequelizerc` file at the root of our app allows us to specify the paths to where our config file should be read from and where the directories for models, migrations, and seeds should be created:

```
const path = require('path');

module.exports = {
  config: path.resolve('config', 'database.js'),
  'models-path': path.resolve('db', 'models'),
  'seeders-path': path.resolve('db', 'seeders'),
  'migrations-path': path.resolve('db', 'migrations'),
};
```

- Run our sequelize initialization. This reads from `.sequelizerc` to see if we are overwriting any of the default config values, then generates the directories and config file.
    - `npx sequelize init`
- Create our database and user in `psql`

```
CREATE USER reading_list_app WITH PASSWORD 'strongpassword' CREATEDB;
CREATE DATABASE reading_list WITH OWNER reading_list_app;
```

- Add environment variables to `.env`

```
DB_USERNAME=reading_list_app
DB_PASSWORD=strongpassword
DB_DATABASE=reading_list
DB_HOST=localhost
```

- If we are using a config module, add these new variables in:

```
module.exports = {
  environment: process.env.NODE_ENV || 'development',
  port: process.env.PORT || 8080,
  db: {
    username: process.env.DB_USERNAME,
    password: process.env.DB_PASSWORD,
    database: process.env.DB_DATABASE,
    host: process.env.DB_HOST,
  },
};
```

- In the config file generated by sequelize (we specified `./config/database.js` in our example), set up the configuration to point to these values for any environment that we need:

```
const { username, password, database, host } = require('./index').db;

module.exports = {
  development: {
    username,
    password,
    database,
    host,
    dialect: 'postgres',
    seederStorage: 'sequelize',
  },
};
```

- If we didn't have a config module, we could use the environment variables directly:

```
module.exports = {
  development: {
    username: process.env.DB_USERNAME,
```

```
    password: process.env.DB_PASSWORD,
    database: process.env.DB_DATABASE,
    host: process.env.DB_HOST,
    dialect: 'postgres',
    seederStorage: 'sequelize',
  },
};
```

9. Use Sequelize to test the connection to a database before starting the HTTP server on application startup

- To test our connection, we get a reference to our database through the models directory that was created for us. We can call `.authenticate` on the instance of `sequelize` that is exported from this module.

```
#!/usr/bin/env node
// This file is being run to kick off our server. The app is being
imported separately. If these concerns were in the same file, the
process of importing the db and invoking authenticate would remain the
same.

const { port } = require('../config');

const app = require('../app');
const db = require('../db/models');

// Check the database connection before starting the app.
db.sequelize
  .authenticate()
  .then(() => {
    console.log('Database connection success! Sequelize is ready to
use...');

    // Start listening for connections.
    app.listen(port, () => console.log(`Listening on port
${port}...`));
  })
  .catch((err) => {
    console.log('Database connection failure.');
    console.error(err);
  });
```

10. Define a collection of routes (and views) that perform CRUD operations against a single resource using Sequelize

- Before creating routes and views, we should generate models, migrations, and seeds in order to have data to work with. This is the same process as working with sequelize previously.

    1. Generate our model and migration

    - `npx sequelize model:generate --name Book --attributes "title:string, author:string, releaseDate:dateonly, pageCount:integer, publisher:string"`
    2. Update the model and migration files to include constraints such as character liimts, `allowNull`, and `unique: true`, how to drop the table in the migration, etc.
    3. Apply the migrations to our database. Remember to include `dotenv` to include the environment variables. This is needed because `db:migrate` has to connect to our database, and the credential information is stored in our environment variables.
    - `npx dotenv sequelize db:migrate`
    4. Generate a seed file
    - `npx sequelize seed:generate --name test-data`
    5. Update the seed file to include records that we want to add to the database, as well as what to delete if we want to undo the seed.
    6. Apply the seed file. We need to prepend `dotenv` for the same reasons as our migrations.
    - `npx dotenv sequelize db:seed:all`

- Now that we have our database created and seeded, we can proceed with making our routes.

- In the file that we are creating our routes, require the models directory.

```
const db = require('./db/models');
```

- If we have any routes that need to post data (creating, updating, destroying records), we need to provide csrf protection. Add the `csurf` and `cookie-parser` packages to our project. Within our app, indicate that we are using cookieParser and parsing our requests:

```
const cookieParser = require('cookie-parser');

app.use(cookieParser());
app.use(express.urlencoded({ extended: false }));
// We can also use the bodyParser library instead of express's
urlencoded function
// app.use(bodyParser.urlencoded({ extended: false }))
```

- Our routes need to have access to the csrfProtection middleware that comes from the `csurf` package we added.

```
const csrf = require('csurf');

const csrfProtection = csrf({ cookie: true });
```

- In our routes, set up asynchronous route handlers and await the query to our database. We can then pass the results to our views.

    - Getting an index of records

```
router.get(
  '/',
  asyncHandler(async (req, res) => {
    const books = await db.Book.findAll({ order: [['title',
'ASC']] });
    res.render('book-list', { title: 'Books', books });
  })
);
```

- Getting a single records

```
router.get(
  '/book/:id(\\d+)',
  asyncHandler(async (req, res) => {
    const bookId = parseInt(req.params.id, 10); // converting the
string into an integer
    const book = await db.Book.findByPk(bookId); // get a
reference to our Book instance
    res.render('book-display', { title: 'Book Display', book });
  })
);
```

- Posting to create a record

```
router.post(
  '/book/add',
  csrfProtection,
  bookValidators,
  asyncHandler(async (req, res) => {
    // destructure the fields of the book from the request body
    const { title, author, releaseDate, pageCount, publisher } =
req.body;

    // build the instance of the book. We're doing this before
validation redirection because our
    const book = db.Book.build({
      title,
      author,
      releaseDate,
      pageCount,
      publisher,
    });

    const validatorErrors = validationResult(req);

    if (validatorErrors.isEmpty()) {
      await book.save();
      res.redirect('/');
```

```
    } else {
      const errors = validatorErrors.array().map((error) =>
error.msg);
      res.render('book-add', {
        title: 'Add Book',
        book,
        errors,
        csrfToken: req.csrfToken(),
      });
    }
  })
);
```

- Editing a record

```
router.post(
  '/book/edit/:id(\\d+)',
  csrfProtection,
  bookValidators,
  asyncHandler(async (req, res) => {
    const bookId = parseInt(req.params.id, 10); // converting the
string into an integer
    const bookToUpdate = await db.Book.findByPk(bookId); // get a
reference to our Book instance

    const { title, author, releaseDate, pageCount, publisher } =
req.body;

    // we extracted and then repackaged the relevent terms from
the request's body
    const book = {
      title,
      author,
      releaseDate,
      pageCount,
      publisher,
    };

    // The validation results can be accessed by invoking the
function with our request
    const validatorErrors = validationResult(req);

    // If the fields all passed validation, validatorErrors will
be an empty array
    // This means that we can update the book and send the user
back to the main page
    if (validatorErrors.isEmpty()) {
      await bookToUpdate.update(book);
      res.redirect('/');
      // If some fields did not pass the validator, we want to
keep them on the form page
      // and display the message associated with each error
```

```
    } else {
      // Convert the error objects into just their message
strings
      const errors = validatorErrors.array().map((error) =>
error.msg);
      res.render('book-edit', {
        title: 'Edit Book',
        book: { ...book, bookId },
        errors,
        csrfToken: req.csrfToken(),
      });
    }
  })
);
```

- Deleting a record

```
router.post(
  '/book/delete/:id(\\d+)',
  csrfProtection,
  asyncHandler(async (req, res) => {
    // 1. Convert the id in the route from a string to an integer
    // 2. Get a reference to the book that matches this id
    // 3. Destroy the book record
    const bookId = parseInt(req.params.id, 10);
    const book = await db.Book.findByPk(bookId);
    await book.destroy();

    // Instead of finding and then deleting an instance, we could
combine these into one method on the class
    // db.Book.destroy({ where: { id: bookId } })

    // Redirect to the main page
    res.redirect('/');
  })
);
```

11. Handle Sequelize validation errors when users are attempting to create or update data and display error messages to the user so that they can resolve any data quality issues

- We've seen two ways to implement data validation on the server. First is through sequelize model validations, and the second is from the `express-validator` package.
- For sequelize validations, we add a `validate` key on the model for each field we would like to validate:

```
module.exports = (sequelize, DataTypes) => {
  const Book = sequelize.define(
    'Book',
    {
```

```
        title: {
          type: DataTypes.STRING,
          allowNull: false,
          validate: {
            notNull: {
              msg: 'Please provide a value for Title',
            },
            notEmpty: {
              msg: 'Please provide a value for Title',
            },
            len: {
              args: [0, 255],
              msg: 'Title must not be more than 255 characters long',
            },
          },
        },
        author: {
          type: DataTypes.STRING(100),
          allowNull: false,
          validate: {
            notNull: {
              msg: 'Please provide a value for Author',
            },
            notEmpty: {
              msg: 'Please provide a value for Author',
            },
            len: {
              args: [0, 100],
              msg: 'Author must not be more than 100 characters long',
            },
          },
        },
        releaseDate: {
          type: DataTypes.DATEONLY,
          allowNull: false,
          validate: {
            notNull: {
              msg: 'Please provide a value for Release Date',
            },
            isDate: {
              msg: 'Please provide a valid date for Release Date',
            },
          },
        },
        pageCount: {
          type: DataTypes.INTEGER,
          allowNull: false,
          validate: {
            notNull: {
              msg: 'Please provide a value for Page Count',
            },
            isInt: {
              msg: 'Please provide a valid integer for Page Count',
            },
```

```
        },
      },
      publisher: {
        type: DataTypes.STRING(100),
        allowNull: false,
        validate: {
          notNull: {
            msg: 'Please provide a value for Publisher',
          },
          notEmpty: {
            msg: 'Please provide a value for Publisher',
          },
          len: {
            args: [0, 100],
            msg: 'Publisher must not be more than 100 characters
long',
          },
        },
      },
    },
    {}
  );
  Book.associate = function (models) {
    // associations can be defined here
  };
  return Book;
};
```

- With these validations in place, when we attempt to save a new instance (Book in this case), we will result in an error being thrown. This error will have a `name` property equal to `SequelizeValidationError`. We can implement a `try`/`catch` block to see if this error occurred, and if it did, render our form again with the error messages:

```
router.post(
  '/book/add',
  csrfProtection,
  asyncHandler(async (req, res, next) => {
    const { title, author, releaseDate, pageCount, publisher } =
req.body;

    const book = db.Book.build({
      title,
      author,
      releaseDate,
      pageCount,
      publisher,
    });

    try {
      await book.save();
      res.redirect('/');
```

```
    } catch (err) {
      if (err.name === 'SequelizeValidationError') {
        const errors = err.errors.map((error) => error.message);
        res.render('book-add', {
          title: 'Add Book',
          book,
          errors,
          csrfToken: req.csrfToken(),
        });
      } else {
        next(err);
      }
    }
  })
);
```

- We can also implement validations using the `express-validator` package, which we can use as middleware in our routes.
- We extract a reference to the `check` and `validationResult` functions from the `express-validator` library:

```
const { check, validationResult } = require('express-validator');
```

- We can set up a group of validations using an array. Each individual validation invokes `check` with the field that we are validating, then chains on the specific validations and messages we want to associate with that field:

```
const bookValidators = [
  check('title')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Title')
    .isLength({ max: 255 })
    .withMessage('Title must not be more than 255 characters long'),
  check('author')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Author')
    .isLength({ max: 100 })
    .withMessage('Author must not be more than 100 characters long'),
  check('releaseDate')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Release Date')
    .isISO8601()
    .withMessage('Please provide a valid date for Release Date'),
  check('pageCount')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Page Count')
    .isInt({ min: 0 })
    .withMessage('Please provide a valid integer for Page Count'),
```

```
  check('publisher')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Publisher')
    .isLength({ max: 100 })
    .withMessage('Publisher must not be more than 100 characters
long'),
];
```

- With these validations set up, we can pass the array of middleware to any route that needs to validate the input. Inside the route we can invoke `validationResult` with the request object to get access to an array of any errors that occurred with the validations. From there, we can determine if we need to render the route as normal if the the array is empty, or if we need to display errors to the user because of the failed validations.

```
// This is the same route as above, repeated for convenience
// Notice the bookValidators array passed in as middleware
router.post(
  '/book/add',
  csrfProtection,
  bookValidators,
  asyncHandler(async (req, res) => {
    const { title, author, releaseDate, pageCount, publisher } =
req.body;

    const book = db.Book.build({
      title,
      author,
      releaseDate,
      pageCount,
      publisher,
    });

    // The results of the validations are being captured by
validatorErrors. If all of the validations passed, it will be an empty
array.
    const validatorErrors = validationResult(req);

    if (validatorErrors.isEmpty()) {
      // No errors occurred, so save the new book
      await book.save();
      res.redirect('/');
    } else {
      // Validations failed. Extract the `msg` from each error and
pass the array to the form template so that it can iterate over them
and display them to the user
      const errors = validatorErrors.array().map((error) =>
error.msg);
      res.render('book-add', {
        title: 'Add Book',
        book,
        errors,
```

```
          csrfToken: req.csrfToken(),
        });
      }
    })
  );
```

12. Describe how an Express.js error handler function differs from middleware and route handler functions

- Error handlers have a very similar syntax to middleware and route handlers that we create. In addition to the `req`, `res`, and `next` arguments that we captured, we also capture the error (typically `err`) as the first argument to the handler.
- Error handlers are invoked whenever an error occurs in a synchronous route handler or if `next` is invoked *with and argument*.
- When either of these occur, express will continue parsing the file to see if we have any handlers defined below the route to take care of the error. This is why it's important to have our middleware up top so that it is used before routes are encountered, then our routes, then our error handlers at the bottom.
- Just like middleware, we can chain multiple error handlers on to each other. In order to pass control off to the next error handler, we simply invoke `next(err)`, making sure to pass the error in as the argument.
- If we don't have any custom error handlers, express has a default one that will respond to the client with a status of `500` and, if in development, display the error's stack trace. It's important to note that if an asynchronous function is rejected and not caught, the error handlers will not be triggered, causing the server to hang and no response sent back to the client.

13. Define a global Express.js error-handling function to catch and process unhandled errors

- Our handler uses `app.use()` with a callback that takes in `err`, `req`, `res`, and `next`.
- If the err object has a status key set on it, we set our res's status to that value, otherwise we use a generic 500 to indicate an internal server error occurred.
- We render a custom error template with the error information. In the example below, we set up different displays based on whether we are in a production environment or not. If we are, we simply state that an error occurred. The idea is that the end-user doesn't need to know the inner workings of our server, just that we messed up. If we aren't in production, we also pass the specific error message and stack trace to the template in order to render it to the page. This allows us to debug our code in development.

```
// (middleware and routes defined above)

// Generic error handler.
app.use((err, req, res, next) => {
  res.status(err.status || 500);
  const isProduction = process.env.NODE_ENV === 'production';
  res.render('error', {
    title: 'Server Error',
    message: isProduction ? 'An error occurred!' : err.message,
    stack: isProduction ? null : err.stack,
  });
});
```

14. Define a middleware function to handle requests for unknown routes by returning a 404 NOT FOUND error

- If a request comes in for a route that does not match anything that we have created, we can capture the request and throw a specific error for it.
- After all of our routes are defined, we use `app.use` just like our middleware up above. This ensures that the request passes through this handler if it makes it this far without encountering an error.
- In this function we create an error, set the `status` key of the error to 404 (for later use in the error handler), then invoke our error-handling chain by calling `next(err)` with the error we just made:

```
// Catch unhandled requests and forward to error handler.
app.use((req, res, next) => {
  const err = new Error("The requested page couldn't be found.");
  err.status = 404;
  next(err);
});
```

- Before our generic error handler, we can define a specific 404 error handler in order to generate a custom page. If an error passes through this handler, we check to see if its status is 404. If the error was created because of the missing route, then this will be the case. When that occurs, we set our response's status to 404 and render our custom template.

```
// Error handler for 404 errors.
app.use((err, req, res, next) => {
  if (err.status === 404) {
    res.status(404);
    res.render('page-not-found', {
      title: 'Page Not Found',
    });
  } else {
    next(err);
  }
});
```

- If the error didn't have a status of 404 (if the error happened inside one of our other routes or middleware), then we simply invoked `next(err)` to pass it along to the next error handler in the chain.