- ⏱ 6 hours

Amusement Park Tracker

Today you'll begin building your own data-driven website using Express!

The application will allow users to navigate through a list of parks. Users will also be able to create, read, update, and delete parks (CRUD features). This project is divided into two portions: building out pages related to parks and building out pages related to park attractions. As a bonus, you will autonomously build the features related to park attractions.

When you have completed the first portion of the project, your application should have the following features:

1. Navigation elements that allows users to easily navigate between the different park pages.
2. A "Home" index page.
3. A "Parks" page with a table listing all parks.
4. A page listing a park and its details.
5. A form for park creation.
6. A form for updating park details.
7. A page for park deletion.

Phase 0: Download the starter project

Begin by cloning the project skeleton:

```
git clone https://github.com/appacademy-starters/express-amusement-park-tracker.git
```

Now install your dependencies and run your Phase 1 tests:

```
npm install
npm run test-01
```

Phase 1: Initial application set-up

Begin by installing the following dependencies:

```
npm install express@^4.0.0 pug@^2.0.0 morgan
```

Then install Nodemon as a development dependency:

```
npm install nodemon@^2.0.0 --save-dev
```

Setting up your routes

Now that you are all set up, add a `routes` module by creating a `routes.js` file in the root of your project. Begin by requiring `express` and creating a router with `express.Router()`.
Use your router to define the default route that renders the `index` view to the `/` path (i.e. the application's "Home" page). Ensure that you are rendering a `title` for the page. Hint: think of where `{ title: 'Home' }` lives as an argument.

Initializing your views

It's time to create the initial `index` and `layout` views for your application. Begin by creating a `views` folder in the root of your project. Inside your `views` folder, create a `layout.pug` template and an `index.pug` template.
Begin building your HTML DOM tree by using the `html:5` emmet abbreviation to create an `html` element with two child elements: `head` and `body`.
Make sure your `head` renders your `title` - remember that you have defined a `title` variable, `{ title: 'Home' }`, in your default route. Your `title` element should render "Amusement Park Tracker - " in addition to your page's `title`.
Input the following scripts into the bottom of your body to incorporate `jquery`, `popper`, and `bootstrap`:

```
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+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfoo
Ao'
      crossorigin='anonymous')
script(src='https://stackpath.bootstrapcdn.com/bootstrap/4.4.1
/js/bootstrap.min.js'
      integrity='sha384-
wfSDF2E50Y2D1uUdj0O3uMBJnjuUD4Ih7YwaYd1iqfktj0Uod8GCExl3Og8ifw
B6'
      crossorigin='anonymous')
```

Replace the `<!DOCTYPE html>` on the first line with `doctype html` and add the stylesheet below into your `head` to add bootstrap stylings:

```
link(rel='stylesheet'

href='https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/b
ootstrap.min.css'
    integrity='sha384-
Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9If
jh'
    crossorigin='anonymous')
```

Let's begin with building the content of your `body`! Create a `nav` element at the top of `body`. You'll want two child elements in your `nav` bar:

- An "Amusement Park Tracker" hyperlink (`a` tag element) to the home page with a `href` path of `/`.
- An unordered list (`ul`).

The unordered list should contain a list item (`li`) with a "Parks" hyperlink to the `/parks` path.

Take a look at the bootstrapped navigation bar below. Note the use of bootstrap classes and components and how quickly you can improve the design of your application. Free to implement the navigation bar and explore the Bootstrap navbar component documentation for more!

```
nav(class='navbar navbar-expand-lg navbar-dark bg-primary')
      a(class='navbar-brand' href='/') Amusement Park Tracker
      button(class='navbar-toggler' type='button' data-
toggle='collapse' data-target='#navbarNav' aria-
controls='navbarNav' aria-expanded='false' aria-label='Toggle
navigation')
        span(class='navbar-toggler-icon')
      #navbarNav(class='collapse navbar-collapse')
        ul(class='navbar-nav')
          li(class='nav-item'): a(class='nav-link'
href='/parks') Parks
```

Create a `.container` bootstrapped element as another child of your `body`. Remember that this container will hold the `block content` imported from your other view templates. Above rendering the `block content`, you want to make sure that the content page's `title` is rendering a `h2` element.

At this point, your `layout.pug` file look something like this:

```
doctype html
html
```

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

href='https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/b
ootstrap.min.css'
        integrity='sha384-
Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9If
jh'
        crossorigin='anonymous')
    title Amusement Park Tracker - #{title}
  body
    nav(class='navbar navbar-expand-lg navbar-dark bg-primary')
      a(class='navbar-brand' href='/') Amusement Park Tracker
      button(class='navbar-toggler' type='button' data-
toggle='collapse' data-target='#navbarNav' aria-
controls='navbarNav' aria-expanded='false' aria-label='Toggle
navigation')
        span(class='navbar-toggler-icon')
      #navbarNav(class='collapse navbar-collapse')
        ul(class='navbar-nav')
          li(class='nav-item'): a(class='nav-link'
href='/parks') Parks
    .container
      h2(class='py-4') #{title}
      block content
    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+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfoo
Ao'
          crossorigin='anonymous')

script(src='https://stackpath.bootstrapcdn.com/bootstrap/4.4.1
/js/bootstrap.min.js'
          integrity='sha384-
wfSDF2E50Y2D1uUdj0O3uMBJnjuUD4Ih7YwaYd1iqfktj0Uod8GCExl3Og8ifw
B6'
          crossorigin='anonymous')
```
Lastly, let's work on your `index.pug`. Begin by extending the `layout` template and rendering `block content`. In your `block content`, create a `p` element with the following

text: "Hello from the Amusement Park Tracker app!". You are now ready to create your `app` module!

Creating the app module

Begin by creating an `app.js` file in the root of your project. Require your `express` package, your `morgan` package, and your `./routes` module. Initialize your `app` by invoking the `express` package you have required. Call the `app.set()` method to set your `view engine` to utilize `pug`.
Remember that you want your application to use `morgan` as a middleware so that your requests are logged to the console. Call the `app.use()` method to have your application utilize `morgan` in the *dev* format with `morgan('dev')`.
Call the `app.use()` method again to have your application utilize the `routes` required from your `./routes` module.
Lastly, don't forget to export the `app` and `router` modules you have created. Feel free to test your modules by running `npm run test-01`.

Creating your application entry point

Create a folder named `bin` in the root of your project and make a file named `www`. Begin by writing `#!/usr/bin/env node` on the first line of your file. Require your `app` module - think of what is the file path between `./bin/www` to `./app.js`. How would you require your `app.js` file?
After you have successfully imported your `app` module, define a port (i.e. "8080") and have your `app` begin listening for HTTP connections. In `app.listen`, console log a message stating that your application is now listening on your defined port.
Next, update your `package.json` by defining the `start` script. Update the empty `"start": ""` script with the script below:

```
"start": "nodemon ./bin/www"
```

You're now ready to manually test your application! Run the command `npm start` and then browse to `http://localhost:8080/`. You should see the "Home" page displaying the content of the `index.pug` view you have created.

Phase 2: Adding custom error handlers

It's time to add in your error handlers. In your `app` module, you will have your application use custom middleware functions that you will write to:

- Catch unhandled requests and forward errors to error handlers.
- Log errors.
- Render a "Page Not Found" view for 404 errors.
- Render a "Server Error" view for generic errors.

Catch unhandled requests

Let's begin by writing the middleware function to catch unhandled requests and create error objects to forward into our custom error handlers. Since you are writing a middleware function, remember to use `req`, `res`, and `next` as your function's parameters. In `app.js`, create a new middleware function. In the function, begin by instantiating a new `Error` object with the message "The requests page couldn't be found." and then assign your `Error` object's `status` to be `404`. Use the middleware function's `next` parameter to forward your new `Error` object to the next middleware function, which will be your error handler that logs errors.

Log errors

In the middleware function you just created, you passed your `Error` object as an argument into the `next` middleware function. In this error handler function, you will take in `err`, `req`, `res`, and `next` as parameters, with `err` being your `Error` object from the previous middleware function.
If your application is in `production` or `test` mode, you would want to log the error to the database so that you can document and review your application's error history. For now, you'll just focus on logging your errors in the console with `console.error` when your application is `development`. How can you use `process.env.NODE_ENV` to define a conditional that determines what this error handler does? No matter if your application is in `production`, `test`, or `development`, you'll want to pass `err` onto the `next` middleware function.

Page not found

This error handler will take care of rendering a "Page Not Found" view for all errors with a status of `404`. At this point, take a moment to create a `page-not-found.pug` template. Remember that you're already rendering the `title` in an `h2` element in your `layout`. Render a simple and user friendly message in the content of the view, as the view is your client-facing error message.

If the error's status is `404`, you'll want to set the HTTP response `status` to `404` and render the `page-not-found.pug` template. Make sure to include the `title` of your page. If the error's status is not `404`, simply pass on the error to the `next` middleware function.

Server error

This error handler will render a generic "Server Error" view if `err` does not have a status of `404`. Begin by setting the HTTP response `status` to be `err.status`, or `500` if `err` does not have a valid `status`. Now it's time to render an `error.pug` template with a `title`, `message`, and error `stack`.
Use the `message` and `stack` from your `err` object if your application is not running in `production`. Otherwise if your application is running in `production`, set `message` and `stack` to be `null`. You don't want your users to see the unfriendly error message and complicated error stack. Also, think of how you can DRY up your code by declaring a boolean variable that checks whether the application is in `production` mode. Create the `error.pug` template to render the `message` in a `p` element and the `stack` in a `pre` element if the stack trace exists. Now, add the following code to the bottom of your `routes.js` to throw a test error when you browse to the URL `http://localhost:8080/error-test`. Then test your current project by running `npm run test-02`.

```
if (process.env.NODE_ENV !== "production") {
  router.get("/error-test", () => {
    throw new Error("This is a test error.");
  });
}
```

Phase 3: Configuring environment variables

Begin by installing `dotenv` and `dotenv-cli` as development dependencies. Remember that the `dotenv` package is used to load environment variables from an `.env` file and the `dotenv-cli` package acts as an intermediary between tools or utilities (like the Sequelize CLI) to load your environment variables from an `.env` file and run the command that you pass into it.

```
npm install dotenv dotenv-cli --save-dev
```

Next, add an `.env` file and `.env.example` file to the root of your project. In both files, define the `PORT` environment variable and set it to your application's HTTP port (i.e. "PORT=8080"). Remember that your `.env` file shouldn't be committed to source control, because it might contain sensitive information (i.e. database credentials). Add `.env` as an entry to your project's `.gitignore` file.

Create the config module

You'll want your config module to provide access to all of the environment variables of your `process.env`. Begin by creating a folder named `config` to the root of your project. Then add a file named `index.js` to the config folder and begin configuring the project for `development` mode.
In your `index.js` file, set your `environment` to read from the node environment, `NODE_ENV`, and port of your `process.env`. If your `process.env` does not have a valid environment or port, then set the environment to `development` mode and the port to `8080`.

```
module.exports = {
  environment: process.env.NODE_ENV || 'development',
  port: process.env.PORT || 8080,
};
```

Now update the `./bin/www` file to require the `port` from your config module and have your application `listen` to the port from your config module.
You'll want to update all other references to your environment variables to use the `config` module. In what modules do you reference `process.env` in your project? Use `cmd + shift + f` in VS Code to search for where you reference `process.env`. You can also use `cmd + f` within files to search for `process.env.NODE_ENV` and `cmd + d` to select every `process.env.NODE_ENV` within the file to update them all to your config module's `environment`.
The last step is to update your `start` script in your `package.json`:

```
"start": "nodemon -r dotenv/config ./bin/www"
```

Now use the `npm run test-03` command to test your project up to this point.

Phase 4: Installing and configuring Sequelize

Now it's time to install `sequelize` and `pg` as dependencies:

```
npm install sequelize@^5.0.0 pg@^8.0.0
```

Then install the Sequelize CLI as a development dependency:

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

Take a moment to run your tests with `npm run test-04`. Throughout this phase, utilize the detailed the specs to guide your configuration of the Sequelize CLI.

Configuring the Sequelize CLI

You'll want to configure the Sequelize CLI by creating a `.sequelizerc` file in the root of your project. The `.sequelizerc` file configures the Sequelize CLI so that it knows where your database configuration is located and where to generate the models, seeders, and migrations folders.
Time to configure the Sequelize CLI! Begin by requiring `path` at the top of your `.sequelizerc` file.

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

Now we'll break down the module you will build and export in this file. You'll want to use the `path.resolve()` method in each value of your module's properties.

- For the `config` property, connect `config` to `database.js`.
- For the `models-path` property, connect `db` to `models`.
- For the `seeders-path` property, connect `db` to `seeders`.
- For the `migrations-path` property, connect `db` to `migrations`.

Take a moment to run your tests with `npm run test-04` to test your project up to this point.

Initializing Sequelize

It's time to initialize Sequelize by running the following command:

```
npx sequelize init
```

When the command completes, your project should contain the following:

- `config/database.js` file
- `db/migrations`, `db/models`, and `db/seeders` folders
- `db/models/index.js` file

Creating a new database and database user

Before you continue configuring Sequelize, take a moment to create a database user and database by opening `psql` and running the following SQL statements:

```
create database amusement_park_tracker;
create user amusement_park_tracker_app with password '«a strong
password for the user»';
grant all privileges on database amusement_park_tracker to
amusement_park_tracker_app;
```

Make note of the password that you use as you'll need it for the next step in the configuration process!

Adding the database environment variables

Now you're ready to add the `DB_USERNAME`, `DB_PASSWORD`, `DB_DATABASE`, and `DB_HOST` environment variables to the `.env` and `.env.example` files.
The next step is to go back and update your config module's `index.js` to include a `db` property object that defines `username`, `password`, `database`, and `host` properties by accessing the database environment variables you've just added to your `process.env` object through creating variables in your `.env` file.
Remember that the config module is responsible for providing access to your application's environment variables. Any part of the application that needs access to the `DB_USERNAME`, `DB_PASSWORD`, `DB_DATABASE`, and `DB_HOST` environment variables can use the `username`, `password`, `database`, and `host` properties on the config module's `db` object.

Configuring the Sequelize database connection

Now you're ready to configure the database connection for Sequelize! Update the Sequelize database config (`./config/database.js` file) to use the `config` module. Begin by using the `require()` function to import the `index` module.
Now you can destructure the `index` file's `db` object to allow the database configuration to have access to the `db` object's `username`, `password`, `database`, and `host` properties.
Use the `username`, `password`, `database`, and `host` properties to initialize the corresponding properties of the `development` object in `./config/database.js`.
Lastly, you need to set a `dialect` property so that Sequelize is configured for use with `postgres`.
Remember that the `development` property name indicates that these configuration settings are for the `development` environment. The `username`, `password`, `database`, `host`, and `dialect` properties are the Sequelize options used to configure the database connection.

Testing the connection to the database

Let's begin by updating the `./bin/www` file to use Sequelize to test the connection to the database.
Require your `db` from your models (`../db/models`). Notice how the file path is `../db/models` instead of `./db/models` because of how you are requiring into a file within the `bin` directory. If you were requiring into a file within the root directory of the project, you would use `./db/models`.
Next, access your `db.sequelize` property and invoke the `sequelize.authenticate()` function. Create a promise chain by chaining

a `.then` method to console log your `Listening on port #{port}...` message upon successful database connection.
Lastly, chain on a `.catch` method to console log a "Database connection failure." message and `console.error` the errors caught.
Lastly, run your tests with `npm run test-04` to ensure that you've configured the Sequelize CLI successfully.

Phase 5: Creating the Park model

It's time to use the Sequelize CLI to generate the Park model. You want to generate a model named `Park` with the following attributes:

- `parkName` (string)
- `city` (string)
- `provinceState` (string)
- `country` (string)
- `opened` (dateonly)
- `size` (string)
- `description` (text)

Command to create an `Example` model with an `exampleColumn` of datatype `string`:

```
npx sequelize model:generate --name Example --attributes
"exampleColumn:string"
```

The next step is to update the column nullability and string based column lengths in the migration file: 1. Update all columns to not be nullable. 1. Update your `parkName` column to have a maximum length of `255`. 2. Update your `city`, `provinceState`, `country`, and `size` columns to all have a maximum length of `100`.
In your `Park` model file, update each `Park` property to be an object with `type` and `allowNull` properties. You'll want to: 1. Update all columns to not be nullable. 1. Update your `parkName` column to have a maximum length of `255`. 2. Update your `city`, `provinceState`, `country`, and `size` columns to all have a maximum length of `100`.

The last step is to apply the pending migration and run your tests:

```
npx dotenv sequelize db:migrate
npm run test-05
```

Phase 6: Seeding the database

Now you'll want to create the seeder with the command below:

```
npx sequelize seed:generate --name test-data
```

Open your `[timestamp]-test-data.js` file and populate the seeder with test data for a park. Choose any values to create one `Park` object in the `up` property function.
Remember that the object needs to have the following attributes:

- `parkName`
- `city`
- `provinceState`
- `country`
- `opened`
- `size`
- `description`
- `createdAt`
- `updatedAt`

Note that you'll want to instantiate `Date` objects for the `opened`, `createdAt`, and `updatedAt` attributes.
Now, set the `down` property in your seeds module to delete all objects known as `Parks`.

Lastly, seed the database and run your tests with the following commands:

```
npx dotenv sequelize db:seed:all
npm run test-06
```

Phase 7: Adding the Park List page

It's now time to query the database for a list of parks. Begin by creating a `GET` route handler for the `/parks` route in your `routes.js` file. You'll want pass your route's function within an `asyncHandler`.
Define an `asyncHandler` function that accepts a reference to a route `handler` function and returns a function that defines three parameters, `req`, `res`, and `next`.
You will use the function within the `asyncHandler` to invoke the `handler` function with `req`, `res`, and `next` as arguments. You'll also want to invoke `.catch` with `next` so that errors are caught.
As a reminder, defining an `asyncHandler` function helps keep your code DRY by helping you avoid writing boilerplate `try`/`catch` code for each route handler.
When invoking the `asyncHandler` function, remember to use the `async` keyword to declare the `/parks` route handler as an asynchronous function so that you can `await` your database query. For example:

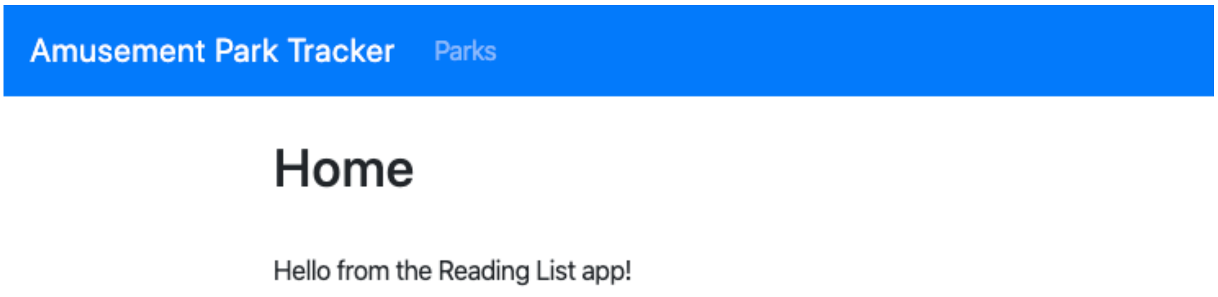
```
router.get('/parks', asyncHandler(async (req, res) => {
  // TODO: Await database query
  // TODO: Render template
});
```

Within the `/parks` route, declare a `parks` variable to `await` your database query of all of the parks with `db.Park.findAll`. Make sure you have required your `db` model from `./db/models`. Order the list of parks in alphabetical order by the `parkName` attribute.

Rendering the parks view

Now that you have a database fetch for all your parks, create and render a `park-list.pug` template. Make sure you pass in your array of `parks` to the view and set the page's `title` to "Parks".
Now we'll focus on updating the `park-list` view to render the list of parks. Begin by extending from your `layout.pug` at the top of the template and creating `block content`.
In the `block content`, create an "Add Park" hyperlink with a path of `/park/add`.
Next, create a `table` element with a header (`thead`) and body (`tbody`) as children elements. In the header, render a row (`tr`) for the "Park Name", "Location", and "Opened". Use `th` elements with a `scope` of `column` to indicate the elements as headers for the columns. In the table's body, render a new row for each park. You can use `td` elements to render the `parkName`, `city`, `provinceState`, `country`, and `opened` status for each park.
Lastly, create a hyperlink for each park that links to the each park's "Details" page with the `href` of `/park/${park.id}`. Render this hyperlink as the last `td` of each park row.
At this point, feel free to add some Bootstrap classes to style your elements. For example, you can use the bootstrap class `btn btn-success` and have a `role` attribute of `button` for the hyperlinks you've created to style your links as snazzy buttons. Take a look at the examples below for quick bootstrap stylings you can apply to your project:

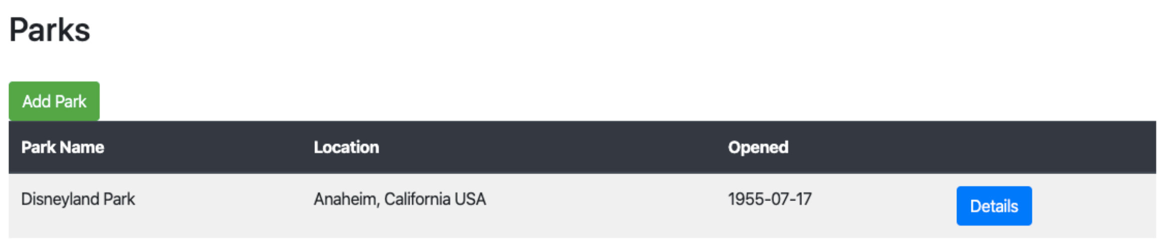
Home Page:



Home

Hello from the Reading List app!

Parks Page:



Parks

Add Park

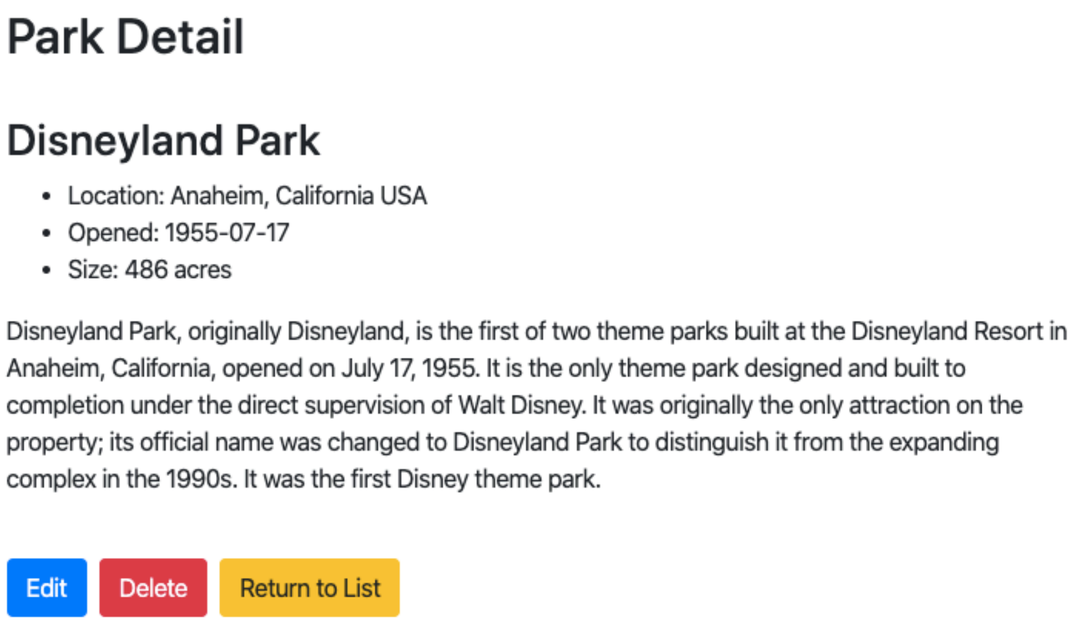
Park Name	Location	Opened	
Disneyland Park	Anaheim, California USA	1955-07-17	Details

Phase 8: Adding the Park Detail page

It's now time to update the `routes` module to route to a park's "Detail" page.
Create a new route for the path of `/park/:id(\\d+)`. Remember to pass the route handler through your `asyncHandler` function. In the handler, you'll want to parse the `req.params.id` property from a string to an integer.
Remember that `req` is an object encapsulating information from the HTTP request. There is a `params` property the holds information about the request path. Let's dissect the `/park/:id(\\d+)` route. The use of `:id` in the route means that your `req.params` will have a property of `id`. The regular expression `(\\d+)` is used in the route to ensure that the value of `req.params.id` is a digit.
Next, use the integer you've parsed from `req.params.id` to query the database for the `park`. You'll want to render a `park-detail.pug` template and pass the `park` to the view. Remember to also set the page `title` to "Park Detail".

Rendering the Park Detail view

It's time to create the `park-detail` view! Begin by extending your `layout` view and rendering `block content`. In the `block content`, create four child elements: one `h3` title for the park's `parkName` and three `div` elements.
Take this example page that renders the details of the `Disneyland Park` object:


Park Detail

Disneyland Park

- Location: Anaheim, California USA
- Opened: 1955-07-17
- Size: 486 acres

Disneyland Park, originally Disneyland, is the first of two theme parks built at the Disneyland Resort in Anaheim, California, opened on July 17, 1955. It is the only theme park designed and built to completion under the direct supervision of Walt Disney. It was originally the only attraction on the property; its official name was changed to Disneyland Park to distinguish it from the expanding complex in the 1990s. It was the first Disney theme park.

In the first `div`, you'll to render the park's details using a unordered list of three list items sharing information about the park's location, opening date, and size. Make sure to include the labels from the image above. Your tests will be checking for the existence of these labels and interpolation of park details.
In the second `div`, render a paragraph element with the park's `description`.
In the third `div`, you should display three hyperlinks:

- An "Edit" hyperlink to the park's edit page with a path of `/park/edit/${park.id}`.
- A "Delete" hyperlink to the park's deletion page with a path of `/park/delete/${park.id}`.
- A "Return to List" hyperlink to the park list page with a path of `/parks`.

The "Park Detail" page could be updated in the bonus phase to display a list of attractions available at that park.

Phase 9: Adding the Add Park page

Begin by updating the `routes` module with two new routes: a `GET` route and a `POST` route for `/park/add`:
In the `GET` route, begin by using the `db.Park.build()` method to create a new `park`. Render a `park-add.pug` template and remember to pass the the `park` and a `title` of `Add Park` into your view. Note that you do not need to wrap this `GET` route with your `asyncHandler` because you are not awaiting any database fetch or persistence before rendering the view.
Wrap your `POST` route within your `asyncHandler` so that park creation errors are caught and forwarded to the custom error handlers. Because of how the `POST` route allows users

to persist data into your database, you'll need access to the `req.body` object as well as protection against CSRF attacks.

Using middleware to protect your application

Let's take care of adding the `cookie-parser` and `express.urlencoded()` middleware to your application.
Install the `cookie-parser` and `csurf` npm packages:

```
npm install cookie-parser@^1.0.0
npm install csurf@^1.0.0
```

In your `app.js`, use the `require()` function to import the `cookie-parser` middleware. Invoke `cookie-parser` in an `app.use()` method to add the middleware to your application after where you have used `morgan`. Make sure you invoke the cookie parser you have required, otherwise your application will be caught in an unresolved promise. See what happens to your application when you use the middleware with `app.use(cookieParser)` instead of `app.use(cookieParser())`. You'll also want to use the built-in `express.urlencoded()` middleware to validate incoming request form data. Make sure you set the `extended` property to `false`.
In `routes.js` require the `csurf` module and create a middleware function by invoking the module with cookies enabled to protect routes against CSRF attacks.

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

Now you're ready to use CSRF protection middleware to keep your application safe from CSRF attacks and `express.urlencoded()` middleware to access validated information from the `req.body` object!

Setting up your protected routes

Add your `csrfProtection` as middleware to both your `GET` and `POST` routes for `/park/add`. Return to your `GET` route and pass in a `csrfToken` property with a value of `req.csrfToken()` to your view.
Now return to your `POST` route. Destructure the `req.body` object to access the form's `parkName`, `city`, `provinceState`, `country`, `opened`, `size`, and `description` properties. You'll use these in the `db.Park.build()` function to initialize your `park` object.
Now you'll want to `await` the persistence your park instance to the database with `park.save()` before redirecting the user to the "Home" page upon success. But wait, what if your form has bad input data?

Validating request form data

This is where validations come into play. It's time to validate your request form data and assign error messages that would be rendered to your user upon error. Feel free to use a popular data validation library like express-validator or Sequelize model validations.

Apply the following validation rules to your park model properties:

`parkName` should:

- Not be null or empty.
- Render an error message of "Please provide a value for Park Name" if null or empty.
- Not be longer than 255 characters in length or render an error message of "Park Name must not be more than 255 characters long".

`city` should:

- Not be null or empty.
- Render an error message of "Please provide a value for City" if null or empty.
- Not be longer than 100 characters in length or render an error message of "City must not be more than 100 characters long".

`provinceState` should:

- Not be null or empty.
- Render an error message of "Please provide a value for Province/State" if null or empty.
- Not be longer than 100 characters in length or render an error message of "Province/State must not be more than 100 characters long".

`country` should:

- Not be null or empty.
- Render an error message of "Please provide a value for Country" if null or empty.
- Not be longer than 100 characters in length or render an error message of "Country must not be more than 100 characters long".

`opened` should:

- Be a valid date or render an error message of "Please provide a valid date for Opened".
- Not be null or render an error message of "Please provide a value for Opened".

`size` should:

- Not be null or empty.
- Render an error message of "Please provide a value for Size" if null or empty.
- Not be longer than 100 characters in length or render an error message of "Size must not be more than 100 characters long" .

`description` should:

- Not be null or empty.
- Render an error message of "Please provide a value for Description" if null or empty.

Adding validation errors to your routes

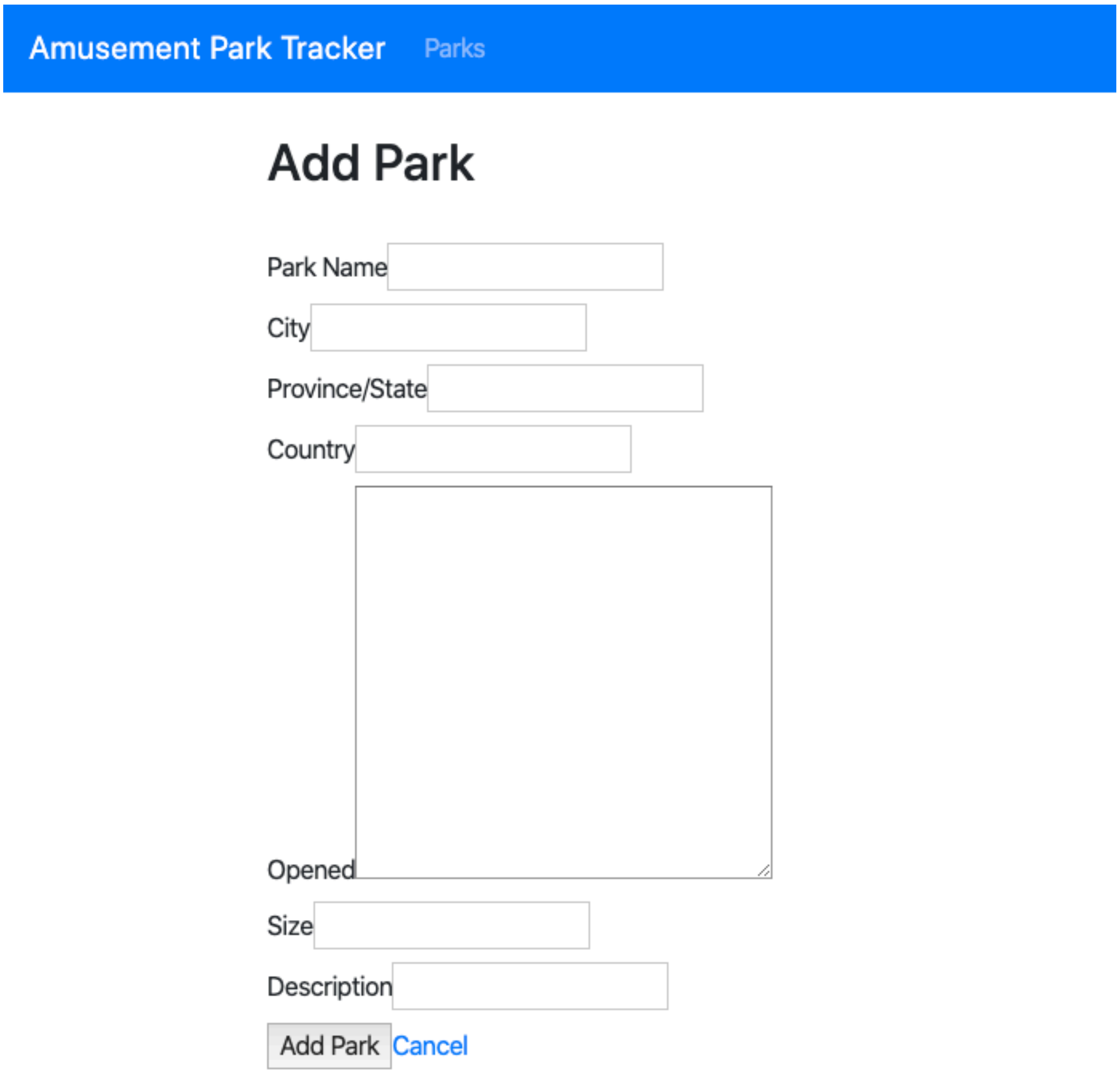
Back in your `POST` route, add a conditional that checks for the presence of validation errors before the park is saved to the database.
If you have used the `express-validator` library, make sure to pass in the array of park validator functions you have created as a middleware in the `POST` route. You can generate a collection of `validatorErrors` by invoking `validationResult()` with your `req` object. Make sure to take advantage of using `validatorErrors.array()` to transform the collection of errors into an actual array of error objects you can map.
If you used Sequelize model validations, use a `try`/`catch` block instead of an `if`/`else` conditional to determine the result of a successful or unsuccesful `park.save()`. If your park instance fails to save, catch the `error` and check whether the `error` has a `name` of `SequelizeValidationError`. If your error was not a Sequelize validation error, pass the error onto the `next` error handler. Remember that the Sequelize `error` object caught has a property called `errors` that holds an array of error objects.

Generating park creation errors

If you have validation errors present, declare an `errors` variable to map each of your error objects into an array of error messages. Remember that you can console log your error objects to determine how to access each error's message. You'll use these error messages in your front-end view to display validation messages to the end user.
Render your `park-add` template with the same `title` of `Add Park`, the `park` object, the `errors` array, and a `csrfToken` property with a value of `req.csrfToken()`.
Your `asyncHandler` function will take care of passing any errors onto the `next` error handler.

Rendering the park creation view

Now you'll use your templates to create an HTML form. If you look ahead into the next phase, you'll notice that creating the `Edit Park` and `Add Park` pages are very similar. With this in mind, you can plan ahead to create pug templates and mixins that you will `include` to your create and edit form views.



Begin by creating the `park-add.pug` template. Extend your `layout.pug` and render `block content`. In your `block content`, create a `form` element with an `action` of `/park/add`, your park creation route, and a `method` of `post`. Now you'll want to create your park form fields. Instead of manually adding each field to your `park-add.pug` template, you can plan ahead to reuse a `park-form-fields.pug` template that utilizes mixins.

Using mixins for DRYer code

Let's begin by creating `utils.pug` file to house your [mixins](). This way you only need to manually type out the pug code for a label and input field once. You can then reuse the mixin to generate the rest of your form fields.
Think of what a form field is comprised of. Begin by creating a `textField` mixin that takes in the parameters of `labelText`, `fieldName`, and `fieldValue`. Within the mixin, generate a `div` element that wraps a `label` element and a text `input` field. You can use this `div` parent element to apply [form bootstrap stylings]().
Create a label `for` your `fieldName` with your `labelText`. You'll want to create an input field for each label. Make your `input` a sibling element of `label` and assign your input `id` and `name` attributes to also be your `fieldName`. The label's `for` attribute references the input's `id` attribute to connect each label to an input field. Additionally, notice how the input `name` attributes connect to the keys in the `req.body` object generated by a form request. Lastly, set the default `value` for each input field to be the `fieldValue`.
Create your `park-form-fields.pug` template and include your `utils.pug` to access the mixins you have created. Add a CSRF protection `input` field with a `type` attribute of `hidden`. Assign the field's `name` attribute to be `_csrf` and its `value` to be the `csrfToken` you passed into the view. You can use the [emmet shortcut]() of typing `input:hidden` and hitting the `enter` key to quickly create a hidden input for your CSRF token.
Now use your `+textField` mixin to create input fields for your `parkName`, `city`, `provinceState`, `country`, `opened`, `size`, and `description` properties. Remember to use the properties of the `park` object you have passed into the view as each input's `fieldValue`.

Including templates for DRYer code

Now that you have created your mixins and `park-form-fields` template, it's time to use them in your `park-add` template! Under the `form` element you have created, `include` your `park-form-fields.pug` template to add all the form fields you have created.
Now create an "Add Park" `button` with a `type` attribute of `submit`. Upon click of this button, your form will package the field input values into the `req.body` object that is received by your park creation route. Under the submit button you have created, add a "Cancel" hyperlink that will link users back to the home page.

Rendering errors

But what if your user tries to submit the form with empty fields? This is where error rendering on the user end is important.

Just like how you used mixins in the `park-form-fields` template, you can render an errors mixin across different parts of the website to keep your code DRY.
Begin by creating a new mixin in your `utils.pug` that takes in your `errors` array. Remember that you only want to render the code from this mixin if you have errors present. Add a conditional to check whether you you have errors. Create a `div` element with two children elements: a paragraph with a message of `The following error(s) occurred:` and an unordered list. Render each of the errors in your `errors` array as a list item in your unordered list. Feel free to add the class of `alert alert-danger` and role of `alert` to the `div` element to apply bootstrap stylings for your error messages.
Now return to your `park-add` template to include your `utils.pug` and use your errors mixin to render errors above your form.

Refactoring the park description field

Run your tests with `npm run test-09` to check your progress up to this point. You should receive an error reporting that you are missing an input `placeholder` of `'ex: 2000-01-31'` and that your form should contain a `textarea` element for each park `description` field.
It's time to refactor your mixins to include a `placeholder` and create a `textarea` element instead of `input` element depending on the field's expected length.
In your `textField` mixin definition, add an `fieldPlaceholder` parameter and a `isMultiline` parameter. You'll begin by adding a `placeholder` attribute with your `fieldPlaceholder` parameters to your `input` element. Next, you want to use your `isMultiline` boolean to either render a `textarea` or your created `input`.
Let's focus on creating the `textarea` element. You want to connect the `textarea` to a specific label, so you'll use the `fieldName` as the `id` attribute. You also want to connect the `textarea` to the `req.body`, so assign the `name` attribute to be the `fieldName`. Assign the `rows` attribute to have a value of `5`. Instead of using an attribute to render a `value`, you'll use the `fieldValue` as child content within the `textarea`.
Now you'll need to refactor your `park-form-fields` template. Just like in JavaScript functions, you can have missing arguments when calling mixins. Add a fieldPlaceholder argument of `'ex: 2000-01-31'` for your park's `opened` field. Then, add a `null` fieldPlaceholder argument and a `true` isMultiline argument to your park `description` field.
Run your tests again with `npm run test-09` to check your progress. Feel free to add bootstrap classes to style your project or move forward to the next phase.

Phase 10: Adding the Edit Park page

Just like with the "Add Park" page, update your `routes` module with two new routes: a `GET` route and a `POST` route for `/park/edit/:id(\\d+)`.
In the `GET` route, begin by wrapping your route with your `asyncHandler` because you will be awaiting the database fetch of a park instance. Just like for the park detail page, parse the `req.params.id` from a string to an integer. Use the `db.Park.findByPk()` method to `await` the fetch of your `park` and render a `park-edit.pug` template. Remember to pass the the `park`, a `title` of `Edit Park`, and the `csrfToken` from your `req` object to your view.
Now, also wrap your `POST` route within your `asyncHandler`. This route will also `await` the database fetch of a park instance using the `parkId` parsed from your `req.params.id` to find your `parkToUpdate` instance.
You'll want to destructure the `req.body` to access the form's `parkName`, `city`, `provinceState`, `country`, `opened`, `size`, and `description` properties. You'll use these to generate a new `park` object with the values from your `req.body`. Declare a `park` variable and set it to an object with all the values you have just destructured from your `req.body` (i.e. `parkName`, `city`, etc).

Like in your park creation route, you want to check for the existence of errors to determine the action of your route.

If you do not have errors, you'll want to `await` your `parkToUpdate.update()` before redirecting the user to the park's detail page upon success. You'll invoke `parkToUpdate.update()` with the `park` object you just created from the destructured information of your request body form.
If you do have errors, you'll want to generate an `errors` array to pass into your view. Render your `park-edit` template with a `title` of `Edit Park`, the `park` object with information from the form, the `errors` array, and the `csrfToken` from your `req` object. When passing the `park` object into your view, you'll want to overwrite the `park.id` with `parkToUpdate.id` to actually update the `parkToUpdate` you fetched from your database. To do this, you can use spread syntax to destructure `park` and then assign the `id` attribute, like so:

```
park: {
  ...park,
  id: parkToUpdate.id
}
```

The spread syntax above actually creates the `park` object below:

```
park: {
  parkName,
  city,
  provinceState,
  country,
  opened,
  size,
  description,
  id: parkToUpdate.id
}
```

Alternatively, you could use the `parkId` parsed from your `req.params.id` to assign the `id`, like so:

```
park: {
  ...park,
  id: parkId
}
```

Lastly, don't forget to use your `csrfProtection` middleware to protect your park's update routes and set up your validation middleware (if you used `express-validator`) just like in your park's create routes!

Rendering the park edit view

Now create a `park-edit` view based on the image above. Utilize the templates and mixins you've created to keep your code DRY. For your form's `action`, you'll need to incorporate JavaScript interpolation to render the `park.id` in the form's action route. Think of how much rewriting of code you would have done if you did not create the `park-form-fields.pug` template or the mixins in `utils.pug`.

Make sure you also include error rendering so that your users can be updated when their form data does not successfully persist to the database.

Phase 11: Adding the Delete Park page

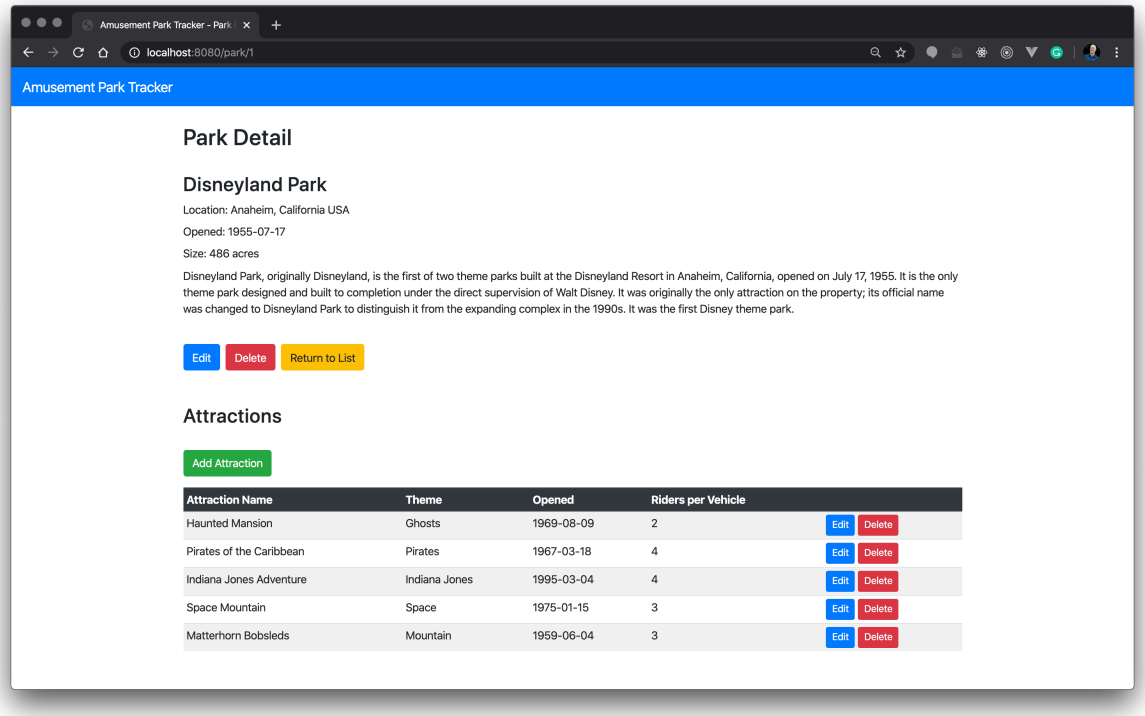
Begin by updating the `routes` module with a new `GET` route and a new `POST` route to `/park/delete/:id(\\d+)`. Remember that since the user will have access to delete entities from the database, both routes should protect against CSRF attacks by using the `csrfProtection` middleware.
In the `GET` route, parse the park `id` parameter from a string into an integer and use the parsed integer to `await` a database query for the `park`. You'll want to render the `park-delete` view and pass the `park` and `csrfToken` to the view. Remember to set the page title to `Delete Park` and to wrap your asynchronous route with your `asyncHandler`.
In the `POST` route, you'll also parse the `id` parameter to query the database for the park. Since you are querying the database, remember to wrap your function with the `asyncHandler` and `await` your database fetch for your `park`. Since you are redirecting the user upon successful deletion, you will not be rendering any template.
After fetching the `park` to delete, use the `park.destroy()` method to delete the park in the database. You'll want to `await` your park deletion before redirecting the user to the "Park List" page.

Rendering the park deletion view

Create a `park-delete.pug` template to render the delete park form. Remember to `extend` from your main `layout` and render `block content`. In the `block content`, the template should render a `h3` with the park's name and two `div` elements. Within the first `div`, a paragraph element should display the message, `Proceed with deleting this park?`.
In the second `div`, render the park deletion form. Render a `form` with an `action` that connects to the `POST` method for deleting a park you defined in `routes.js`. Just like in your edit form, you'll need to incorporate JavaScript interpolation to render the park's `id` in the form's `action` route.
Begin by adding a hidden `input` field for the `csrfToken`. Remember that you can use the emmet shortcut of typing `input:hidden` to quickly create a hidden input for your CSRF token. Then add a "Delete Park" submit `button` for users to submit the form and confirm the deletion. You can also quickly create submit buttons with the `button:submit` emmet shortcut. Lastly, you'll want a "Cancel" hyperlink for your users to be able to return to the park's detail page.

Bonus Phase 1: Adding the Attraction model

In a week you'll begin creating your own Express project! Often times, you begin planning an application by organizing what features to include. As practice for being a full-fledged autonomous engineer, you'll add features for the Attraction model to your project based on the design documentation below. As you build out the `Attraction` model, think what type of association will relate the `Park` and `Attraction` models.



Park Detail

Disneyland Park

Location: Anaheim, California USA

Opened: 1955-07-17

Size: 486 acres

Disneyland Park, originally Disneyland, is the first of two theme parks built at the Disneyland Resort in Anaheim, California, opened on July 17, 1955. It is the only theme park designed and built to completion under the direct supervision of Walt Disney. It was originally the only attraction on the property; its official name was changed to Disneyland Park to distinguish it from the expanding complex in the 1990s. It was the first Disney theme park.

Attractions

Attraction Name	Theme	Opened	Riders per Vehicle	
Haunted Mansion	Ghosts	1969-08-09	2	Edit Delete
Pirates of the Caribbean	Pirates	1967-03-18	4	Edit Delete
Indiana Jones Adventure	Indiana Jones	1995-03-04	4	Edit Delete
Space Mountain	Space	1975-01-15	3	Edit Delete
Matterhorn Bobsleds	Mountain	1959-06-04	3	Edit Delete

MVP List

1. Attraction model with the following attributes:

   - `attractionName`
   - `theme`
   - `opened`
   - `ridersPerVehicle`
   - `visitedOn`
   - `rating`
   - `comments`
   - `parkId`

2. Attractions list on the park details page

   - The park details page should display a table of associated attractions.
   - Each attraction row should:
     - Render all of its attributes except `comments`.
     - Link to its `Attraction Details` page.

3. Attraction details page

   - Render all of an attraction's attributes (including `comments`) as details.
   - Include “Edit” and “Delete” links and a “Return to Park” link (all styled as buttons)

4. Refactored park form

- The views to create and edit parks should utilize the same pug view.
- Think of how you can refactor and reuse the templates and mixins you have already created to keep your code DRY.

5. Attraction form

- The views to create and edit attractions should utilize the same pug view.
- The attraction form should include error validation and error rendering.

6. Attraction deletion page

- Render a form that allows users to confirm their deletion.
- Render a `Cancel` hyperlink to return to the attraction details page upon cancellation.

Attractions Database Schema

Column Name	**Data Type**	**Details**
id	integer	not null, primary key
parkId	integer	not null, foreign key
attractionName	string	not null
theme	string	not null
opened	dateonly	not null
ridersPerVehicle	integer	
visitedOn	dateonly	
rating	integer	
comments	text	
createdAt	date	not null
updatedAt	date	not null

Attractions Sample Data

```
{
  attractionName: 'Space Mountain',
  parkId: 1,
  theme: 'Space',
  opened: new Date('1975-01-15'),
  ridersPerVehicle: 3,
  createdAt: new Date(),
  updatedAt: new Date(),
},
{
  attractionName: 'Matterhorn Bobsleds',
  parkId: 1,
  theme: 'Mountain',
  opened: new Date('1959-06-04'),
  ridersPerVehicle: 3,
  createdAt: new Date(),
  updatedAt: new Date(),
},
{
  attractionName: 'Grizzly River Run',
  parkId: 2,
  theme: 'River Rafting',
  opened: new Date('2001-02-08'),
  ridersPerVehicle: 4,
  createdAt: new Date(),
  updatedAt: new Date(),
}
```

Bonus Phase 2: Planning and adding the AttractionVisit model

Begin by designing the database schema. Consider how the `Park` model, the `Attraction` model, and the `AttractionVisit` model are related. Think of how different foreign keys and primary keys relate certain `AttractionVisit` instances to certain `Attraction` instances and certain `Attraction` instances to certain `Park` instances.
After you have designed your `AttractionVisit` database schema, implement the model to your application and add the following features below. Think of how you would break down the plan to build each requested feature.

1. AttractionVisit list on the attraction details page
2. AttractionVisit count on the park details page

Did you find this lesson helpful?