WEEK 12
Full-Stack Applications (Part 2)

    - OAuth 2.0 in detail
- **Token-Based Authentication**
    - JSON Web Token (JWT)
    - Setting up token-based authentication

WEEK-12 DAY-1
Project Planning

Your First Group Project

This is a big thing! You're getting ready to start work on your first big project to become the first entry in your portfolio for interviewing and showcasing. After you get your team together, you need to choose a project and, then, start doing some analysis.

The groundwork

Somebody set up the group project repositories, one for the front-end and one for the backend. In those repositories, create a **documentation** directory with two subdirectories: **feature-list** and **feature-packets**.

Now, it's important to note:

THERE IS NOT A DEDICATED BACKEND TEAM NOR A DEDICATED FRONTEND TEAM.

Your workflow will be: as a pair, choose a feature that you want to complete, and then do that from front to back, UI to database. That is the good way to do things.

Choosing a Project

This project is meant to challenge you and your team to build something from the ground up. You've had practice using the tools of PostgreSQL, Sequelize, and npm to create skeleton projects. Then, you've installed packages like Express.js and Pug.js to get your framework in place. Now, you can bring all of that to bear in this first, big application.

You will pick one of the following options to clone.

- Bandcamp
- Goodreads
- Medium
- Product Hunt
- Remember the Milk
- Stack Overflow
- TeaWithStrangers
- Quora
- Yelp

These projects were chosen because they allow you to focus on basic CRUD actions rather than complicated front-end features like infinite scroll and single-page apps. Remember that this project is focused on express and we have not yet taught you a front-end framework so you'll be limited to vanilla javascript.

The feature list

Once you have picked a project, spend time with your team figuring out the features that you want your application to have. Ask yourselves

- What features are needed to make this an application that people would use?
- What features would be nice to have if the minimum viable product gets finished?
- Will these features demonstrate everything we've learned during the first half of the course?

Try to create the feature list from the perspective of the person who will be using your application. You can see examples of that in our example repository.

Save this feature list in the **feature-list** subdirectory of the **documentation** directory of the team's repository.

Feature scoping

Post your feature scoping list to the Slack channel and tag the instructors. They will work with you to help you determine what is a good amount of work for your MVP and, then, for your stretch goals.

Feature packets

Once your team knows what is the MVP, work on the feature packets. Create a new Markdown document for each feature for your MVP. In it, include the following documentation:

- Models needed
- Endpoints needed
- Templates needed
- Wire frames or sketches

This will help guide the team's development efforts.

Code, code, code!

Once those are done, your team is set up for success! Determine whether you want to work on features together, one at a time, or have pairs work on features together, or if everyone in the group gets their own feature.

Be smart about your Git workflow. Create branches, commit to them, then create Pull Requests on GitHub to let others see your changes.

Coordinate many times per day so everyone knows what's going on.

Keep up the communication.

Don't let someone fail. This is a team effort.

Evaluating your completion

Full-stack projects will be evaluated against the following "Minimal Viable Product" features.

1. New account creation, login, and guest/demo login
2. A production README file for your GitHub repository containing
   - Brief explanation of what the app is and does
   - Link to live site
   - Discussion of technologies used
   - Discussion of two features that show off the team's technical abilities
   - Discussion of both challenges faced and the way the team solved them
   - Code snippets to highlight the best code
3. Hosting on Heroku (you'll get instructions next week)
4. For four features:
   - Adequate styling
   - Smooth, bug-free navigation
   - Adequate and appropriate seed data to demonstrate the feature

In order for this to be considered complete, you must have the first three criteria completed. In addition to that, you will also need to have four of your features demonstrating the checklist under item 4. After the following section is a list of required features if you decide to clone one of the sites from the list above.

Show and tell

Instead of an assessment on the following Monday, each team will demonstrate what they've been able to accomplish! This type of demonstration normally occurs in real software development shops after a sprint ends. That's what you'll be doing: showing what you've been able to accomplish in such a short amount of time!

Required Features for Approved Clones

If you decide to do a direct clone of the one of the sites above, here are the four minimum features that you must implement for each of the choices.

Bandcamp

- Artist page
- Song player
- Search
- Upload/download songs
- **Bonus**: Purchase songs
- **Bonus**: Follows

Goodreads

- Books
- Bookshelves
- Reviews
- Read Status (will read, have read, etc.)
- **Bonus**: Search across multiple models
- **Bonus**: Tags

Medium

- Stories
- Commenting on stories
- Follows and feed
- Likes
- **Bonus**: Topics/categories
- **Bonus**: Bookmarks

Product Hunt

- Products
- Profile Page
- Product Discussion
- Search (Users or Products)
- **Bonus**: Collections
- **Bonus**: Upvotes and Tags

Quora

- Questions
- Answers/comments on answers
- Search Questions
- Topics/Tags
- **Bonus**: Upvotes, order questions by popularity
- **Bonus**: Replies to comments

Remember the Milk

- Tasks
- Lists
- List summary (time, num tasks, num completed)
- Search
- **Bonus**: Autocomplete SmartAdd of task properties
- **Bonus**: Subtasks

Stack Overflow

- Ask Questions
- Answer Questions
- Search for Questions
- Upvote / Downvote Answer
- **Bonus**: Question Categories
- **Bonus**: Comment on Questions / Answers
- **Bonus**: Polymorphic Up/Down Votes: Questions, Answers, Comments
- **Bonus**: Code Snippets in Answers

TeaWithStrangers

- Choose City
- Host Event
- Join Event in your city
- Dashboard of joined events/hosted events
- **Bonus**: Google Map API showing events based on location
- **Bonus**: Suggestions based on event details and user profiles

Yelp

- Business Page
- Search / filters
- Reviews / ratings
- Map
- **Bonus**: Mark reviews funny, cool, useful etc.
- **Bonus**: Profile
- **Bonus**: Friends

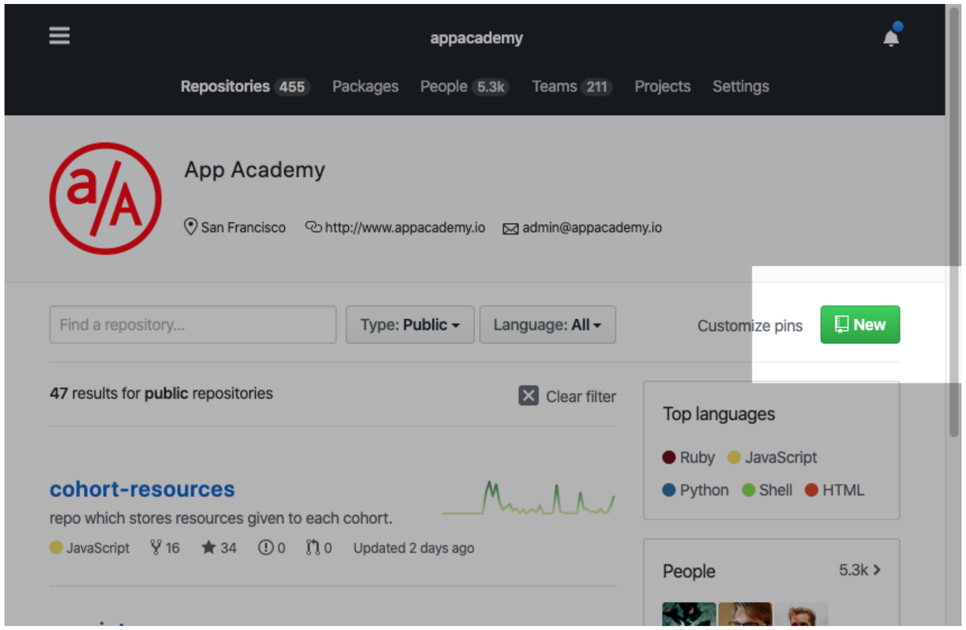
Setting Up Your Group's Project Repository

Only **one person** on the team should do this, the "project lead". Once the project is done, everyone should *fork* the repository to get their own local copies for future reference.

Your group will coordinate and collaborate around a GitHub repository. To do that, you'll need to set it up so that everyone can contribute. These are the steps to get you on your way.

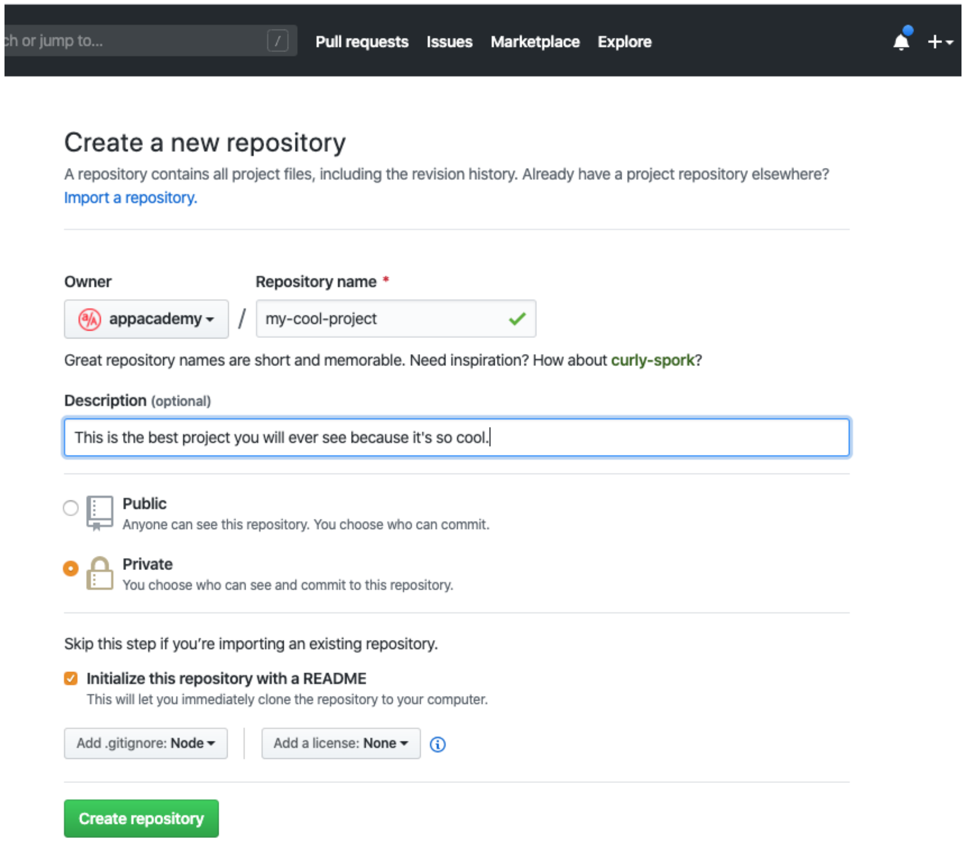
Create a new repository

Go to GitHub. On your profile page, you should see a "New" button that allows you to create a new repository. Click it.



Fill out the name of the repository, its description, and choose to initialize the repository with a README and a **.gitignore** file for Node.js projects.

MAKE IT PUBLIC EVEN THOUGH THE PICTURE BELOW SHOWS PRIVATE! If it's not
public, then you can't show it to the world.



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner: appacademy / Repository name *: my-cool-project

Great repository names are short and memorable. Need inspiration? How about curly-spork?

Description (optional): This is the best project you will ever see because it's so cool.

Public — Anyone can see this repository. You choose who can commit.

[✓ Private] — You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

[✓] Initialize this repository with a README
This will let you immediately clone the repository to your computer.

Add .gitignore: Node | Add a license: None

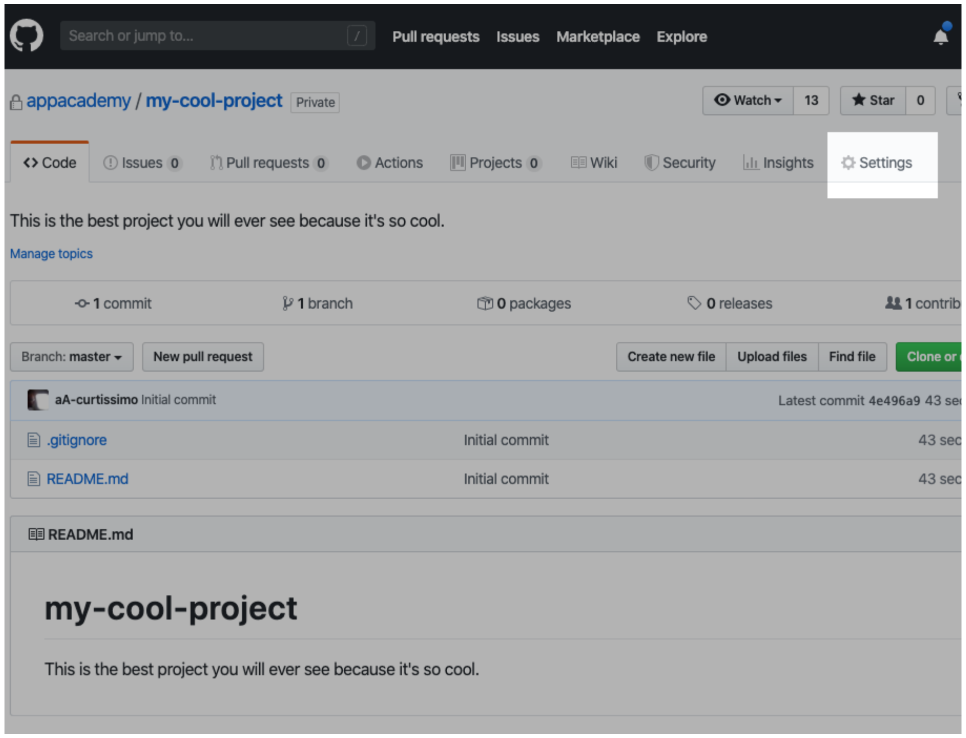
Create repository

Click the "Create repository" button.

Give access to your teammates

Now that you have a repository, you need to give some privileges to your teammates so they can use the repository, too.

On your project page, click the "Settings" tab.



This is the best project you will ever see because it's so cool.

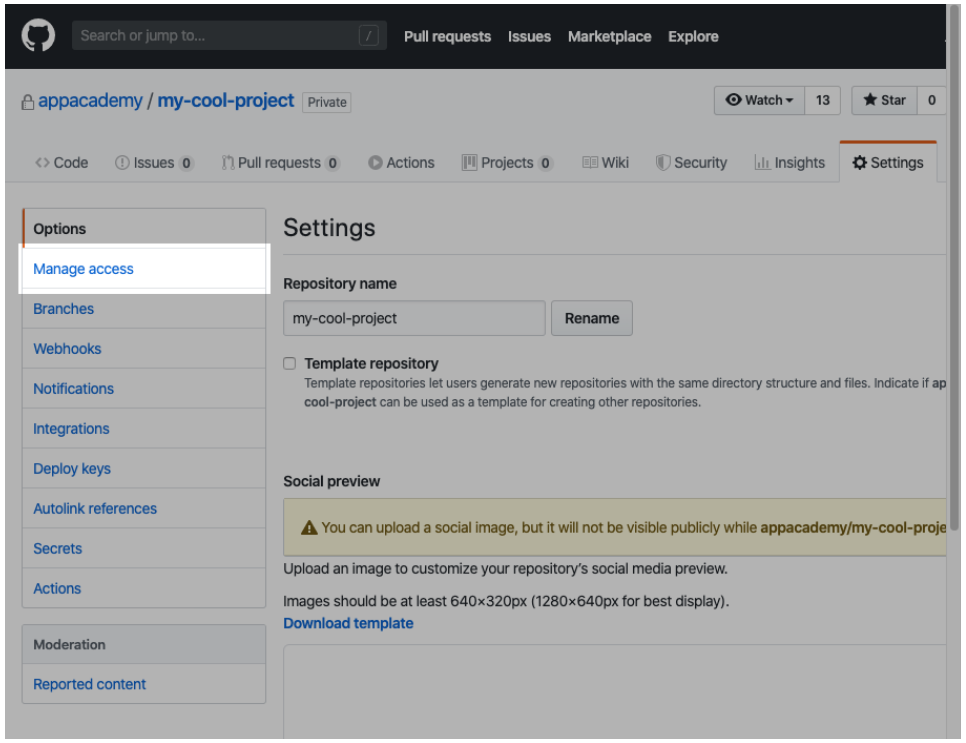
aA-curtissimo Initial commit Latest commit 4e496a9 43 se

.gitignore	Initial commit	43 sec
README.md	Initial commit	43 sec

my-cool-project

This is the best project you will ever see because it's so cool.

Once on the Settings page, click the "Manage access" link.



Settings

Repository name: my-cool-project

Template repository
Template repositories let users generate new repositories with the same directory structure and files. Indicate if ap cool-project can be used as a template for creating other repositories.

Social preview

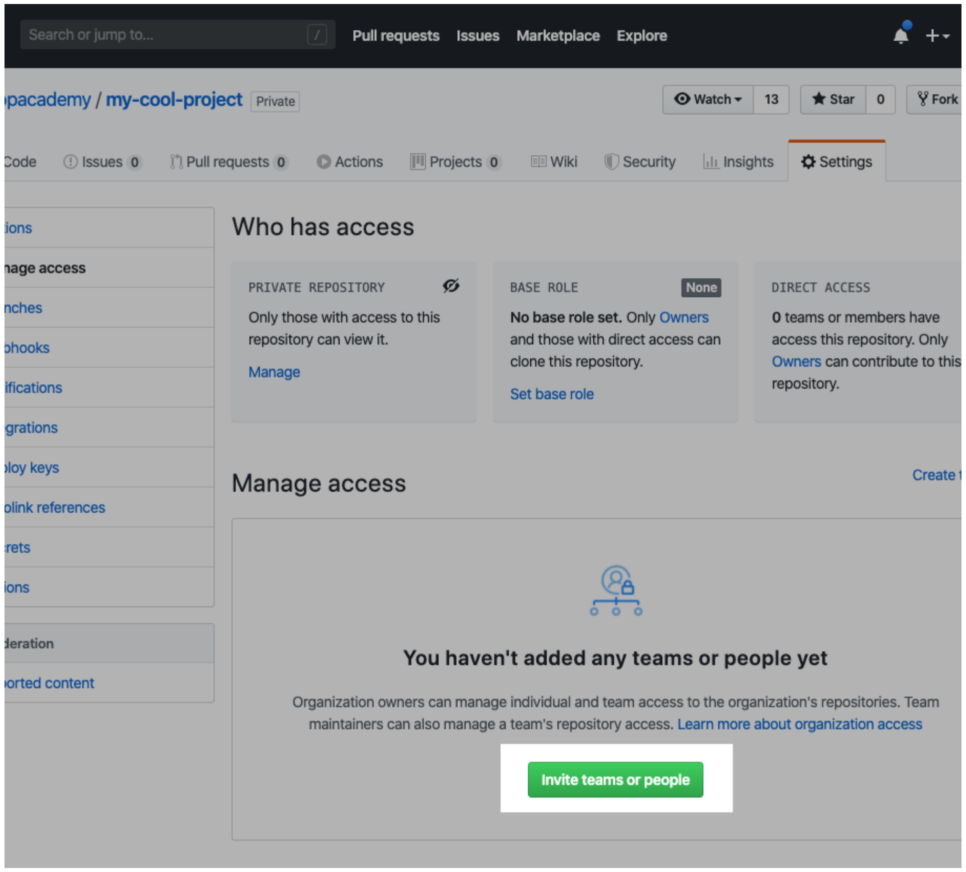
You can upload a social image, but it will not be visible publicly while appacademy/my-cool-proje

Upload an image to customize your repository's social media preview.

Images should be at least 640×320px (1280×640px for best display).

You may have to confirm your password.

Once you get there, click the "Invite teams or people".



Who has access

PRIVATE REPOSITORY
Only those with access to this repository can view it.
Manage

BASE ROLE None
No base role set. Only Owners and those with direct access can clone this repository.
Set base role

DIRECT ACCESS
0 teams or members have access this repository. Only Owners can contribute to this repository.

Manage access

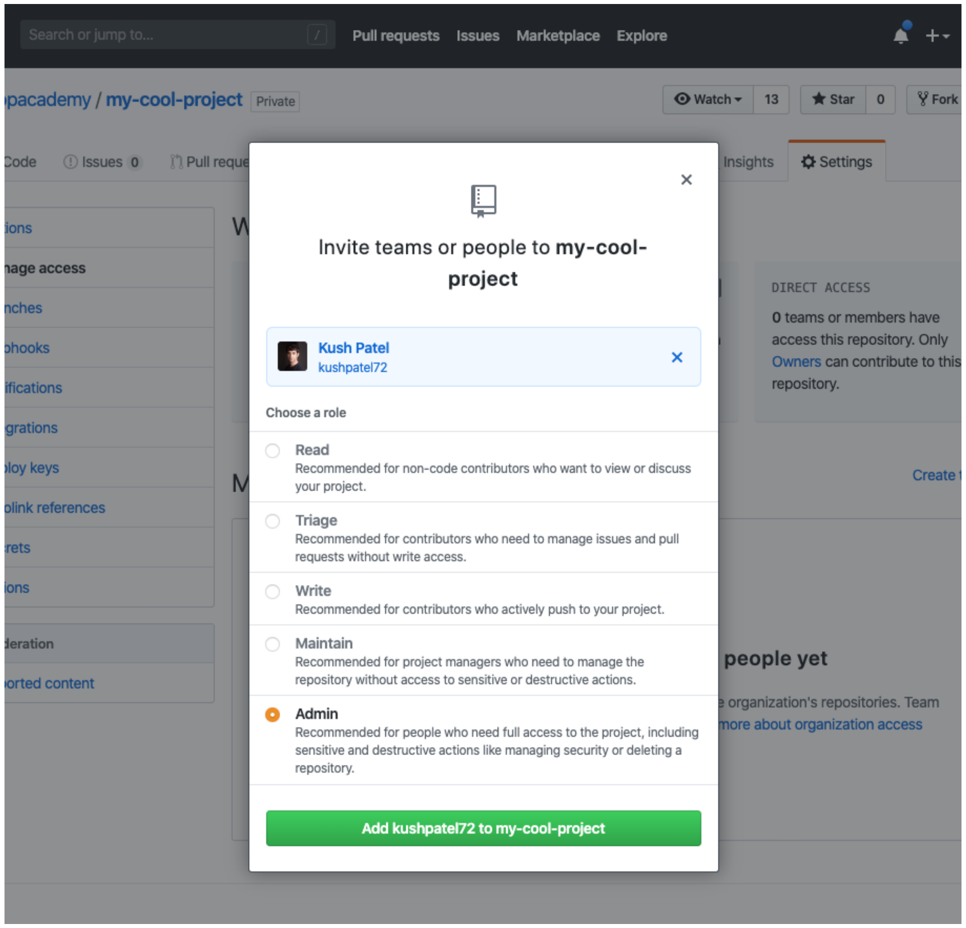
Create t

You haven't added any teams or people yet

Organization owners can manage individual and team access to the organization's repositories. Team maintainers can also manage a team's repository access. Learn more about organization access

Invite teams or people

For each person on your team, invite them to the repository and give them "Maintain" access.



Invite teams or people to **my-cool-project**

Kush Patel
kushpatel72

Choose a role

- [ ] Read
  Recommended for non-code contributors who want to view or discuss your project.
- [ ] Triage
  Recommended for contributors who need to manage issues and pull requests without write access.
- [ ] Write
  Recommended for contributors who actively push to your project.
- [ ] Maintain
  Recommended for project managers who need to manage the repository without access to sensitive or destructive actions.
- [✓] Admin
  Recommended for people who need full access to the project, including sensitive and destructive actions like managing security or deleting a repository.

Add kushpatel72 to my-cool-project

Now, have everyone clone the repository and you can start working together.

Start by creating a **documentation** directory. Inside that directory, create a directory named **feature-list**. In there, create a README.md document in which you can create the list of features and their estimates so that an instructor can review them.

WEEK-12 DAY-2
Authentication and Authorization

Authentication Learning Objectives

Almost every online software application has an **authentication** component that compels visitors to identify themselves. After reading and practicing, you should be able to

- Define the term *authentication*
- Describe the difference between asymmetric and symmetric cryptographic algorithms
- Identify "strong" vs. "broken" hash functions
- Implement session-based authentication in an Express application
- Implement a strong hash function to securely store passwords
- Describe and use the different security options for cookies

---

Using Bcrypt to Hash Passwords

Using BCrypt to hash passwords came from the idea of thinking about how to protect user information in the worst case scenario. For example, what if a hacker tries to inject your application with a SQL query to receive all user information, including user passwords? This is why a password *hash* is stored in the database instead of a user's actual password.

By the end of this article, you will be able to:

- Understand what cryptography is and why it's important;
- Understand what hashing and encryption are and how they differ from each other; and
- Use Bcrypt to hash stored user passwords.

What is cryptography?

Cryptography is a way to use algorithms and secret keys to keep information secure. It is comprised of techniques for secure communication for adversaries. Adversaries are third-parties who could possibly fake a user's identity to gain their secret values.

In modern times, cryptography is synonymous with encryption. Take note that encoding and encrypting are two different processes. An example of an encoding technique is compression, where you would encode the string "aaabcccabbccc" into "a3bc3ab2c3". This is a classic encoding scheme for highly repetitive data. If you know the encoding scheme, you are able to manually decipher the encoded data into it's original form.

What is encryption?

Encrypting is different from encoding. Encryption is the process of translating something that's readable into something that looks like nonsense (i.e. not readable) and being able to translate it from a non-readable state back into something that's readable.

An example of an encryption technique is the Caesar Cipher algorithm. Imagine that you are using a `caesarCipher` function below to transform the string "abcd". The function uses its key of `2` to shift all its characters two characters to "the right" to have a final encrypted string of "cdef".

```
caesarCipher("abcd", 2) // returns "cdef"
```

A string is encrypted when we still don't know for certain what the original input was even though if we understand the algorithm used (i.e. understanding Caesar Cipher, but not being able to read or determine its input from its output).

How does encryption work?

There are two kinds of encryption, **symmetric** and **asymmetric** encryption. The Caesar cipher is *symmetric* because it uses one value to determine how to encrypt data, the number of values to shift the letters in the message. If you know that one value, like `2` from the previous example, then you can reconstitute the original message by using that knowledge, shifting the letters left two steps.

Asymmetric encryption uses two pieces of information called the public and private keys. The public key is shared with anyone wanting to encrypt a message for the recipient. The private key is used to decrypt the message. Encrypt with public key, decrypt with private key. This is why it's known as asymmetric, because one key does the encrypting and the other key does the decrypting.

Establishing an HTTPS connection is an example of asymmetric encryption. To break down the steps of encryption between a computer and a Web server that is going to establish a secure HTTP connection (i.e., HTTPS), they would do the following:

1. The server passes on its public key to encrypt data along with its SSL certificate.
2. The browser client uses the server's public key to encrypt a value and generate a new private key.
3. The client sends the encrypted value and the client's new private key to the server.
4. The browser's private key is used to decrypt messages that have been encrypted with the server's public key.
5. The server sends encrypted data to the client using the client's public key.
6. The browser decrypts the data from the server and renders the decrypted information.

When is it appropriate to use encryption?

It's appropriate to use encryption to secure over the wire communication between the client and server (e.g. HTTPS or TLS/SSL). Data at rest (i.e. stored in a database) can also be encrypted. For example, credit card numbers should be encrypted (if they're stored at all). Sometimes data just needs to be protected at rest and you don't need the ability to decrypt it. For example, passwords need to be protected at rest, but you don't need the ability to translate a password back into human readable form. For this reason, hashing is a far more popular way of protecting user passwords.

What is hashing?

Hashing is the process of converting a message of any length into a short, fixed-length string. Hashed values cannot be translated back to their original input values. Hashing is deterministic, meaning that every time you hash the same input, you will receive the same output. This is why we use hashing to secure user credentials.

How does hashing work?

There are many different types of hashing functions. It's important to use cryptographic hashing functions, as they minimize hash collisions (i.e. creating the same hashed value from different inputs). We'll be focusing on using the BCrypt library which uses the Blowfish cipher, an encryption algorithm.

What is a "salt"?

Imagine if multiple users have the same password. This means that they would have the same password digest stored in the database. If one of the users has an exposed password, a hacker could find all the users with the same password digest and hack into all accounts with the exposed password. This is where salting comes into play. The basic idea behind salting is to begin by generating a small, random string or set of bits known as a `salt`. You would then append the `salt` to your the user's password before hashing.

If you create a new `salt` for each user, each time you hash a password for a different user, you are guaranteed to generate a unique password digest to store in the database. Instead of just storing a possibly generic password digest, the password and the unique `salt` generated for the user is stored.

Now when you see a password digest, the digest is no longer the digest of a common password. It's a secure digest of a common password concatenated with a salt to randomize the digest output.

When is it appropriate to use hashing?

Hashing is a popular way of storing passwords. A hashed password is often referred to as a "password digest". When a user creates an account, their password digest is saved to the database. This keeps the user credentials safe by ensuring that the user's actual password is never stored in the database. You almost never need to convert user passwords back into something that's human readable. You simply hash a provided password and compare the hash to the stored hash. If the hashes match, then the user has provided the correct password.

Although the password digest is in the database, the user's credentials are still safe even if user information is extracted via SQL injection because of how hashing is deterministic. If a hacker uses a user's password digest in attempt to logging in, the password digest would get hashed into a new hashed value that would not equate to the user's stored password digest.

Using Bcrypt to hash user passwords

BCrypt is a password hashing function that's widely used to hash user passwords. In this section, you'll learn how to implement BCrypt into your application.

Begin by installing the `bcryptjs` npm package:

```
npm install bcryptjs
```

Now require it in your application:

```
const bcrypt = require('bcryptjs');
```

You can either use BCrypt synchronously or asynchronously. It is recommended to generate hashes asynchronously. As of version `2.4.0`, `bcryptjs`'s asynchronous methods return a promise if a callback is omitted.

To generate a hashed value, you'll await the `hash` method. The `hash` method takes in a password to hash and the number of `saltRounds` to generate a salt. Note that whenever you invoke the `hash` function with a number of `saltRounds`, the function generates the `salt` for you within the hash function.

Auto-generates salt based on number of `saltRounds`:

```
const saltRounds = 10;
const hash = await bcrypt.hash(password, saltRounds);
```

Manually generates salt before generating password digest:

```
const saltRounds = 10;
const salt = await bcrypt.genSalt(saltRounds);
const hash = await bcrypt.hash(password, salt);
```

In order to log in a user with their credentials, you would use BCrypt's `compare()` method to check whether a user-provided password matches a stored database hash. `await` the call to the `compare()` to return a `isPassword`. Note that `isPassword` is a Boolean value.

```
const isPassword = await bcrypt.compare(password, hash);
```

As a recap, your asynchronous implementation of BCrypt hashing should look something like this:

```
const bcrypt = require('bcryptjs');

async function getHash(password, saltRounds) {
  const hash = await bcrypt.hash(password, saltRounds);
  console.log(hash);
  return hash;
}

async function isPassword(password, hash) {
  const isPassword = await bcrypt.compare(password, hash);
  console.log(isPassword);
  return isPassword;
};

(async () => {
  const hashedPassword = await getHash('P@ssw0rd', 10);
  const passwordIsMatch = await isPassword('P@ssw0rd', hashedPassword);
})();
```

Hashes can also be generated synchronously with BCrypt's `hashSync()` method. Like in asynchronous execution, there are also two ways of hash generation. The first way generates a `salt` and `hash` on separate function calls. The `hash` then uses the `salt` to generate a password digest.

```
const saltRounds = 10;
const salt = bcrypt.genSaltSync(saltRounds);
const hash = bcrypt.hashSync("B4c0/\/", salt);
```

The second way auto-generates a salt to be used in the hash function. In this case, BCrypt's `hashSync()` method takes in a password to be hashed and a number of salt rounds.

```
const saltRounds = 8;
const hash = bcrypt.hashSync('bacon', saltRounds);
```

You would then use BCrypt's `compareSync()` method to compare a password with a stored database hash to determine whether or not a user has entered valid credentials.

```
bcrypt.compareSync("B4c0/\/", hash); // true
bcrypt.compareSync("not_bacon", hash); // false
```

Now that you have learned how to use the `bcryptjs` library, you can safely store hashed passwords in your database without the worry that a hacker can reverse engineer them!

Configuring Sessions in Express

In this reading, you'll learn about HTTP sessions. You'll also learn about how to configure a session store as well as how to secure the session cookie.

Let's begin by revisiting how HTTP is stateless. Since HTTP is stateless, query string parameters and HTTP cookies provide ways to persist values across requests. However, storing data in query string parameters can become cumbersome and possibly look strange to end users. This is where session storage comes into play.

By the end of this article, you should be able to:

- Use the `express-session` middleware to configure an Express application to support sessions;
- Use the `req.session` object to store and retrieve a value;
- Configure the `express-session` middleware to use a production-ready session storage provider; and,
- Use the available security options (i.e. secure, httpOnly, domain, path, and expires) to secure a session's HTTP cookie.

Overview of sessions

Let's begin by revisiting how HTTP is a stateless protocol. This means that each HTTP request is independent from other requests that were executed before or after. Once the server has processed an incoming request and returned a response, it forgets about the client.

What are sessions?

Even though HTTP cookies allow the persistence of values across requests, cookies aren't an efficient way to store anything other than small amounts of data. Transmitting HTTP cookies between the server and client (and back again) can undesirably increase the amount of traffic on the server.

Most browsers don't reliably support more than 50 cookies per domain and the total size of all of the data in the cookies has to be less than 4093 characters of text. That may sound like a lot, but when you are storing data for a person's session in there, you will run out of room very quickly.

Sessions build upon the idea of an HTTP cookie. Instead of storing data in the cookie itself, a unique identifier known as the **session ID** is stored. This **session ID** is linked to an object stored on the server.

Why are sessions useful?

Sessions give you a way to identify a series of requests as being connected to the same client. Once you know that a request is connected to a known client session, you can associate the state (data) of that session without having to send that data to the client and rely upon them to send that data back to the server unaltered.

What are the drawbacks?

Although sessions are useful, there are still drawbacks to using sessions. Using sessions increases the overhead required to serve clients. Server affinity, the ability of a router to send a request to the same server over and over for a specific client, can be an issue depending on the session store that you're using.

Configuring Express to use sessions

Clone the starter project files at: starter files

Install the project's dependencies and run it.

```
npm install
npm start
```

The application is a simple website that contains three pages: "Home", "About", and "Contact".

Currently, when you browse from page to page, the server doesn't recognize or associate the page requests as being part of the same session.

Open up your web developer tools to view cookies under the "Application" tab. Click on "Storage > Cookies > `http://localhost:8080`" to view the cookies for your `localhost` domain. Currently, you shouldn't see any cookies listed.

Let's configure the application to use sessions!

Installing and configuring `express-session`

Install the `express-session` npm package.

```
npm install express-session
```

Since the `express-session` npm package handles your session cookies, you no longer need to install the `cookie-parser` package separately. If `cookie-parser` is configured separately, `express-session` and `cookie-parser` would both need to use the same `secret` value. You'll learn more about this `secret` value below.

Add the `express-session` middleware to the `app` module.

```
// ./app.js

const express = require('express');
const session = require('express-session');

const app = express();

app.set('view engine', 'pug');
app.use(session({
  secret: 'a5d63fc5-17a5-459c-b3ba-6d81792158fc',
  resave: false,
  saveUninitialized: false,
}));

// Code removed for brevity.
```

> **Note:** Ideally the session `secret` option value is set from an environment variable. Using a literal value here is being done to keep this example as simple as possible.

Configuration options

Let's take a closer look at the session middleware you configured in `app.js`:

```
app.use(session({
  secret: 'a5d63fc5-17a5-459c-b3ba-6d81792158fc',
  resave: false,
  saveUninitialized: false,
}));
```

Notice the following keys that configure your session creation:

secret: This is the secret used to sign the **session ID** cookie. The `secret` value above was generated using the uuid npm package. The `uuid` package allows you to generate universally unique identifiers (UUIDs) from random cryptographically-strong values, a timestamp, or a user-supplied string.

resave: This option forces the session to be saved into the session *store*, even if the session was never modified during the request. You would typically want to set this option as `false` to prevent overwriting sessions during race conditions, which are undesired parallel requests. However if your session store sets an expiration date on stored sessions, then you likely need to set "resave" as `true`.

saveUninitialized: This forces an *uninitialized* session to be saved to the store. An uninitialized session is when a session is new but not modified. It's useful to set this option as `false` when creating login sessions, reducing use of server storage, or complying with permission laws to set cookies. Setting the option as `false` also prevents race conditions when multiple requests are made without a session.

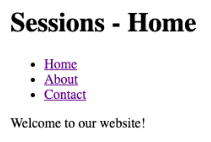
Not setting the `resave` and `saveUninitialized` options results in the following warning in the console:

```
express-session deprecated undefined resave option; provide resave option app.js:
express-session deprecated undefined saveUninitialized option; provide saveUninit
```

Another configuration option that you can set, but which is not listed in the example above, is the **name** of the cookie. By default, the **express-session** middleware uses the name `connect.sid`. Imagine if you have an application running on `localhost:8080` and a different application on `localhost:3000`. Since cookies are scoped to the general `localhost` domain, this means that cookies set for `localhost:8080` would appear in your `localhost:3000` cookies. This is why it's important to set a specific `name` property and separate each application's session cookies from each other.

Testing

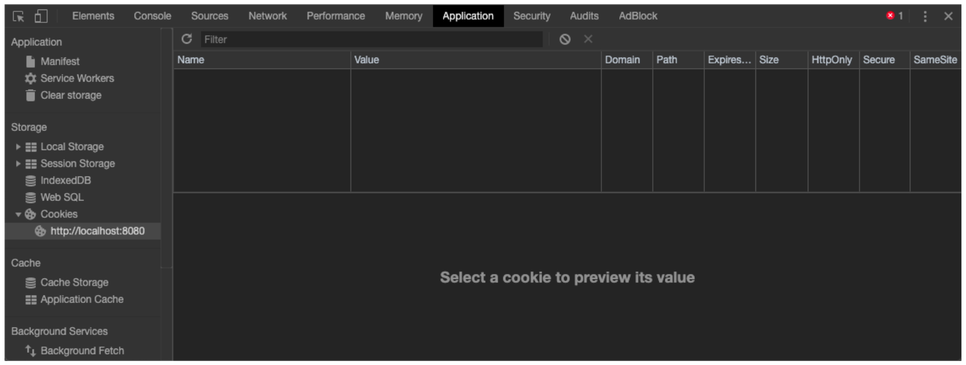
In order to test whether your session cookie is properly created in the next step, begin by opening up your developer tools to view the cookies of the `localhost` domain. Remember to access the cookies by clicking Application > Storage > Cookies > `http://localhost:8080`.



Sessions - Home

- Home
- About
- Contact

Welcome to our website!



As you can see, the session cookie isn't being created. This is because setting `saveUninitialized` to `false` prevents the session from being created until the session has been used. Let's use the `session` in your application! Keep your developer tools open to be able to view your cookies as they appear.

Setting and accessing session values

Let's use `session` to track each page that the user visits!

Create a middleware function that adds each request URL to an array of page visits stored in session.

```
// ./app.js

// Code removed for brevity.

app.use(session({
  secret: 'a5d63fc5-17a5-459c-b3ba-6d81792158fc',
  resave: false,
  saveUninitialized: false,
}));

app.use((req, res, next) => {
  // Attempt to get the `history` array from session.
  // If it's not initialized, then create an array
  // and assigned it back to session.
  let { history } = req.session;
  if (!history) {
    history = [];
    req.session.history = history;
  }

  // Construct the full URL for the current request.
  // Note: Using `req.get('host')` to get the hostname also
  // gives you the port number.
  const url = `${req.protocol}://${req.get('host')}${req.originalUrl}`;

  // Add the URL to the beginning of the array.
  history.unshift(url);

  // Note: We don't need to update the `session.history` property
  // with the updated array because arrays are passed by reference.
  // Because arrays are passed by reference, when we get a
  // reference to the array in the above code
  // `let { history } = req.session;` and modify the array by
  // calling `history.unshift(url);` we're modifying the original
  // array that's stored in session!

  next();
});

// Code removed for brevity.
```

After refreshing your page, you should now see a cookie in your developer tools.

![after-session-cookie-creation][devtools-with-session-cookie]

Notice how the code accessed the `session` attribute of the `req` object. In order to store or access session data, you use the `req.session` property.

In the example above, you attempted to get the `history` array from session. If the array was not already initialized, you created an array and assigned it to the "history" property of the `req.session` object.

Now you'll want to update each of the route handlers to pass the `req.session.history` array to the views.

```
app.get('/', (req, res) => {
  res.render('index', {
    title: 'Home',
    history: req.session.history,
  });
});

app.get('/about', (req, res) => {
  res.render('about', {
    title: 'About',
    history: req.session.history,
  });
});

app.get('/contact', (req, res) => {
  res.render('contact', {
    title: 'Contact',
    history: req.session.history,
  });
});
```

Then update the `layout.pug` view to render the history array.

```
doctype html
html(lang="en")
  head
    title Sessions - #{title}
  body
    h1 Sessions - #{title}
    ul
      li: a(href='/') Home
      li: a(href='/about') About
      li: a(href='/contact') Contact
    block content
    footer
      h2 History
      ul
        for item in history
          li: a(href=item)= item
```

Now when you browse from page to page, your history is tracked by the server using `session`. The middleware function below (that you have already added to your `app.js`) takes care of generating the `history` array of visited links. In each of your GET routes, your `req.session.history` array is passed into the view as a `history` attribute. Your views then use the `history` attribute to render a list of visited links in the footer of each page.

Note that browsing history is specific to each user session. If you open a private, Incognito tab, you'll see that the private client gets its own array of visited pages.

Session store configuration

The default in-memory store that you have been using (`MemoryStore`) is meant only for local development. There are many available session store options. For example, `connect-pg-simple` and `connect-session-sequelize` are two PostgreSQL compatible session store options.

Let's install and configure the `connect-pg-simple` session store!

Install the npm package.

```
npm install connect-pg-simple
```

Next, create a database and a database user. You can use the `psql` terminal commands below.

```
create database session_example;
create user session_example_app with encrypted password '«a strong password for t
grant all privileges on database session_example to session_example_app;
```

Then create the `session` table in the `session_example` database. You can run the following command from the root of your project to create the `session` table.

```
psql -U session_example_app session_example < node_modules/connect-pg-simple/tabl
```

Or you can run the following SQL statements in `psql` against the `session_example` database:

```
CREATE TABLE "session" (
  "sid" varchar NOT NULL COLLATE "default",
  "sess" json NOT NULL,
  "expire" timestamp(6) NOT NULL
)
WITH (OIDS=FALSE);

ALTER TABLE "session" ADD CONSTRAINT "session_pkey" PRIMARY KEY ("sid") NOT DEFER

CREATE INDEX "IDX_session_expire" ON "session" ("expire");
```

Update the npm `start` script by setting `PGUSER`, `PGDATABASE`, and `PGPASSWORD` environment variables. These variables are used by the `pg` npm package, which is used by the `connect-pg-simple` package to communicate to the PostgreSQL database. Make sure to set the `PGPASSWORD` to the password you've given to the `session_example_app` user.

```
"scripts": {
  "start": "PGUSER=session_example_app PGDATABASE=session_example PGPASSWORD=pass
}
```

An alternative way to update the npm `start` script is by defining the `DATABASE_URL` environment variable. The `DATABASE_URL` is used by `connect-pg-simple` to determine how to connect to the PostgreSQL database containing the `session` table.

```
"scripts": {
  "start": "DATABASE_URL=postgresql://session_example_app:password@localhost:5432
}
```

Now update the `app` module as follows.

```
// ./app.js

const express = require('express');
const session = require('express-session');
const store = require('connect-pg-simple');

const app = express();

app.set('view engine', 'pug');
app.use(session({
  store: new (store(session))(),
  secret: 'a5d63fc5-17a5-459c-b3ba-6d81792158fc',
  resave: false,
  saveUninitialized: false,
}));

// Code removed for brevity.
```

Test your app again, and it should behave as it did before, but now the session data is stored in the database!

In `psql`, connect to the `session_example` database and run the following query:

```
select * from session;
```

You should see a `sid` session ID, a session cookie, and a `history` array of visited urls stored in your PostgreSQL database, like the example below.

```
sid: 2kseXpBRl9h1ZHoBndzEyu_cem9l0GN4
sess: {"cookie":{"originalMaxAge":null,"expires":null,"httpOnly":true,"path":"/"},"history":["h
expire: 2020-04-11 12:19:13
```

Securing the session HTTP cookie

As you see in your database query, you have a `cookie` object with the following keys of `originalMaxAge`, `expires`, `httpOnly`, and `path`.

A session cookie has a default configuration of:

```
{
  httpOnly: true,
  maxAge: null,
  path: '/',
  secure: false
}
```

A secure session cookie should be configured like so:

```
{
  httpOnly: true,
  maxAge: «time in milliseconds»,
  path: '/',
  secure: true
}
```

It's important to configure cookies as secure HTTP cookies because it's possible for sessions to be hijacked. Insecure cookies allow hackers to make requests to the web application through another user's session.

Setting the cookie's `httpOnly` property to `true` prevents JavaScript on the page from accessing the cookie. This helps prevent cross-site scripting (XSS) attacks by making the `httpOnly` cookies inaccessible through the `document.cookie` API.

Setting the cookie's `secure` property requires that HTTPS is used. HTTPS uses a digital certificate for authentication known as an SSL certificate to make a web application more secure. Using `secure` cookies and HTTPS prevents anyone from being able to *sniff* or hijack the cookie as requests and responses are passed between the client and the server.

In production, the `secure` property should be set to `true`. If your Express application is behind a proxy you would need to configure it to trust the proxy with `app.set('trust proxy', 1)`.

Appropriately setting a `maxAge` on the cookie keeps the cookie from living longer than it needs to. Have you ever logged into a bank website and noticed how quickly the website will log you out if you're inactive? This is to prevent someone from stealing or accessing your information when sitting down at your computer while you're away. Expiring sessions as quickly as possible helps to protect your users from accidentally allowing someone to access the web application using their identity.

`path` - Defaults to `/`

Additional session configuration cookie options

You can also configure session cookies with the following options that are not set by default:

`domain`

Since no domain is set by default, this causes browsers to interpret the current domain as the cookie's domain. This is connected to why setting cookies in a `localhost` domain (i.e. `localhost:8080`) would set cookies in a different `localhost` domain (i.e. `localhost:3000`).

`expires`

Instead of setting this property directly, you can use `maxAge` instead. Note that `maxAge` takes in an integer number representing milliseconds to calculate when the cookie should expire.

What you have learned

You now know what HTTP sessions are as well as how to configure and access the session store in your Express application. You also know how to configure cookie options to secure the session cookie.

Now that you have learned about sessions, you can implement sessions in your own applications!

[devtools-with-session-cookie]: images/configuring-sessions-devtools-2.pngimages/configuring-sessions-devtools-2.pngpthFrom=2 depthTo=6 orderedList=false} -->

Implementing Session-Based Authentication

In this reading, you'll learn how to implement session-based authentication in an Express application.

By the end of this article, you should be able to:

- Understand what authentication and authorization are and how they differ from each other;
- Understand how to support user self-registration and login within a web application; and
- Add session-based authentication to an Express website including user flows to support self-registration and login.

Overview of authentication and authorization

So far, all of the users of your applications have been anonymous. While the server can use a feature like sessions to determine if a request is from a client that's been served before, the server still doesn't know the identity of that user.

If an application can't identify its users, it doesn't have a way to associate data with specific users. The ability to know whose data belongs to whom is an extremely important feature.

Think about popular web applications, such as Facebook. Being able to identify the current user is necessary so Facebook can determine who your friends are, what posts to display to you, and if you post, who to associate the post with.

What is authentication?

Authentication is the process of identifying a user. To authenticate a user, a `key` and a `secret` is required.

- The `key` is typically a username or an email address.
- The `secret` is typically a password.

The user's `key` and `secret` fields are stored in the application's database as a user account.

A user's account can also contain other information about the user. For example, an account can contain personal information such as a birth date, a mobile phone number, or a physical address.

Application preferences are also stored in a user account. These preferences can include how many items to include in lists, the designation of a default home page, or color scheme (light vs dark mode).

Authentication process

When a user authenticates, they provide their username and password via an HTML form. The form then posts to the server. The login route handler attempts to retrieve the user from the database using their username. If a user is found, then the user account's *hashed* password is checked against the provided password. Passwords are stored in the database as encrypted *hashes*.

The user's provided password needs to be hashed first before it can be compared to the password hash from the database. If the password hashes match, then the user is logged in. With session-based authentication, the user's ID is stored in the session. Subsequent requests to the application can then check if the session contains a user ID. If a user ID is available, then the user is logged in!

The application can then retrieve the user's account information and make it available to other middleware, the route handler, and the view.

What is authorization?

Authorization is the process of determining if the currently logged in user has access to an application's data or features. Sometimes, data is associated directly with a user. For example, a table in the database for storing posts or comments would include a `userId` column. The `userId` column would have the primary key `id` value from the `Users` table.

Relating data to specific users allows the application to retrieve just the user's own records. Sometimes data should only be accessed by users that are assigned a specific user function or role.

For example, a user might belong to an "Admin" role. Any user in the "Admin" role has permissions to add, update, or even delete other application users. "Admin" role users may also have additional access to certain application features. Don't let that power go to your head!

We'll explore how role-based access control works in a future article.

Supporting user self-registration

Before a user can login to an application or website, the application needs to know who that user is... they need to register!

Getting the starter project

Clone the starter project:

```
git clone https://github.com/appacademy-starters/express-reading-list-with-auth-s
```

Install the project's dependencies:

```
npm install
```

Add an `.env` file containing the environment variables from the `.env.example` file:

```
PORT=8080
DB_USERNAME=reading_list_app
DB_PASSWORD=«the reading_list_app user password»
DB_DATABASE=reading_list
DB_HOST=localhost
```

Create the database and database user (if needed). Open psql by running the command `psql` (to use the currently logged in user) or `psql -U «super user username»` to specify the username of the super user to use. Then execute the following SQL statements:

```
create database reading_list;
create user reading_list_app with encrypted password '«a strong password for the
grant all privileges on database reading_list to reading_list_app;
```

Make sure that the `.env` file contains the correct database user password!

Start and test the application:

```
npm start
```

Creating the User model

The `User` model should include the following properties:

- `emailAddress` - A string representing the user's email address;
- `firstName` - A string representing the user's first name;
- `lastName` - A string representing the user's last name; and
- `hashedPassword` - A string representing the user's hashed password.

From the terminal, run the following command to use the Sequelize CLI to generate the `User` model:

```
npx sequelize model:generate --name User --attributes "emailAddress:string, first
```

If the command succeeds, you'll see the following output in the console:

```
New model was created at [path to the project folder]/db/models/user.js .
New migration was created at [path to the project folder]/db/migrations/202004102
```

This confirms that two files were generated: a file for the `User` model and a file for a database migration to add the `Users` table to the database.

Open the `./db/models/user.js` file and update the `User` model's attribute data types and nullability:

```
'use strict';
module.exports = (sequelize, DataTypes) => {
  const User = sequelize.define('User', {
    emailAddress: {
      type: DataTypes.STRING(255),
      allowNull: false,
      unique: true
    },
    firstName: {
      type: DataTypes.STRING(50),
      allowNull: false
    },
    lastName: {
      type: DataTypes.STRING(50),
      allowNull: false
    },
    hashedPassword: {
      type: DataTypes.STRING.BINARY,
      allowNull: false
    }
  }, {});
  User.associate = function(models) {
    // associations can be defined here
  };
  return User;
};
```

Take a moment to notice that the `emailAddress` attribute is configured to be `unique`. This means that there is a *unique constraint* on that column in the database. The *unique constraint* ensures that each user in the database will have a unique `emailAddress`. Without the assurance of knowing that each user has a unique `emailAddress`, you wouldn't be able to reliably identify a user by their email address.

The Sequelize `STRING` data type has an available property of `BINARY`. Setting an attribute to have a datatype of `STRING.BINARY` means that the string is stored as raw-byte data. According to the PostgreSQL documentation, a binary string is a sequence of octets (or bytes).

Make sure to also set the `hashedPassword` attribute with a datatype of `STRING.BINARY` in the migration file:

```
'use strict';
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable('Users', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      emailAddress: {
        type: Sequelize.STRING(255),
        allowNull: false,
        unique: true
      },
      firstName: {
        type: Sequelize.STRING(50),
        allowNull: false
      },
      lastName: {
        type: Sequelize.STRING(50),
        allowNull: false
      },
      hashedPassword: {
        type: Sequelize.STRING.BINARY,
        allowNull: false
      },
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE
      }
    });
  },
  down: (queryInterface, Sequelize) => {
    return queryInterface.dropTable('Users');
  }
};
```

Then apply the migration:

```
npx dotenv sequelize db:migrate
```

In the console, you should see something similar to the following output:

```
Loaded configuration file "config/database.js".
Using environment "development".
== 20200410231702-create-user: migrating =======
== 20200410231702-create-user: migrated (0.028s)
```

To confirm the creation of the `Users` table, you can run the following command from within psql:

```
\d "Users"
```

> Be sure that you're connected to the `reading_list` database in psql. If you are, the cursor should read `reading_list=#` . If you're not connected to the correct database, you can run the command `\c reading_list` to connect to the `reading_list` database.

After running the `\d "Users"` command, you should see the following output within psql:

```
                                              Table "public.Users"
    Column      |           Type           | Collation | Nullable |
----------------+--------------------------+-----------+----------+-------------
 id             | integer                  |           | not null | nextval('"Use
 emailAddress   | character varying(255)   |           | not null |
 firstName      | character varying(50)    |           | not null |
 lastName       | character varying(50)    |           | not null |
 hashedPassword | bytea                    |           | not null |
 createdAt      | timestamp with time zone |           | not null |
 updatedAt      | timestamp with time zone |           | not null |
Indexes:
    "Users_pkey" PRIMARY KEY, btree (id)
    "Users_emailAddress_key" UNIQUE CONSTRAINT, btree ("emailAddress")
```

Adding the user registration form

Now you'll want to create a form that collects the required account information from the user.

To start, now that the application will have two resources, "books" and "users", let's do a little refactoring! Create a `routes` folder in the root of the project and then rename the `routes.js` file as `book.js` and move the file into the `routes` folder.

Update the `routes` module reference in the `app.js` file:

```
const bookRoutes = require('./routes/book');

// Code removed for brevity.

app.use(bookRoutes);
```

Then move the `csrfProtection` and `asyncHandler` variable declarations to a new `./routes/utils` module:

```
// ./routes/utils.js

const csrf = require('csurf');

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

const asyncHandler = (handler) => (req, res, next) => handler(req, res, next).cat

module.exports = {
  csrfProtection,
  asyncHandler,
};
```

And update the `./routes/book.js` file to import those items from the `utils` module:

```
const { csrfProtection, asyncHandler } = require('./utils');
```

Add a module named `user` to the `routes` folder containing the following code to define the routes for the "Register" page:

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

const db = require('../db/models');
const { csrfProtection, asyncHandler } = require('./utils');

const router = express.Router();

router.get('/user/register', csrfProtection, (req, res) => {
  const user = db.User.build();
  res.render('user-register', {
    title: 'Register',
    user,
    csrfToken: req.csrfToken(),
  });
});

const userValidators = [
  // TODO Define the user validators.
];

router.post('/user/register', csrfProtection, userValidators,
  asyncHandler(async (req, res) => {
    const {
      emailAddress,
      firstName,
      lastName,
      password,
    } = req.body;

    const user = db.User.build({
      emailAddress,
      firstName,
      lastName,
    });

    const validatorErrors = validationResult(req);

    if (validatorErrors.isEmpty()) {
      await user.save();
      res.redirect('/');
    } else {
      const errors = validatorErrors.array().map((error) => error.msg);
      res.render('user-register', {
        title: 'Register',
        user,
        errors,
        csrfToken: req.csrfToken(),
      });
    }
  }));

module.exports = router;
```

Rename the `textField` mixin to `field` in the `utils.pug` template file:

```
mixin field(labelText, fieldName, fieldValue, fieldType = 'text')
  div(class='form-group')
    label(for=fieldName)= labelText
    input(type=fieldType id=fieldName name=fieldName value=fieldValue class='form
```

Notice that a `fieldType` parameter was added that defaults to a value of `text`. Adding this parameter will give you a way to add a password field to the register form.

Then add the `user-register.pug` template file to the `views` folder containing the following code:

```
//- ./views/user-register.pug

extends layout.pug

include utils.pug

block content
  +validationErrorSummary(errors)
  form(action='/user/register' method='post')
    input(type='hidden' name='_csrf' value=csrfToken)
    +field('First Name', 'firstName', user.firstName)
    +field('Last Name', 'lastName', user.lastName)
    +field('Email Address', 'emailAddress', user.emailAddress)
    +field('Password', 'password', user.password, 'password')
    +field('Confirm Password', 'confirmPassword', '', 'password')
    div(class='py-4')
      button(type='submit' class='btn btn-primary') Register
      a(href='/' class='btn btn-warning ml-2') Cancel
```

Notice that there are two "password" fields: "Password" and "Confirm Password". When using the `input` element with a `type` attribute of `password`, the field will replace typed characters with bullets to hide what has been typed into the field. This is a great protection from prying eyes, but you need a way for users to be able to confirm the password that they're providing. Making them type their password twice is doing exactly that.

> **Note:** After renaming the `textField` mixin to `field`, be sure to update the `book-form-fields.pug` template to call the mixin using the new name!

Validating the user registration form

Implement the following validation rules:

- `firstName`
  - Not null or empty
  - Not longer than 50 characters
- `lastName`

    - Not null or empty
    - Not longer than 50 characters
- `emailAddress`
    - Not null or empty
    - Not longer than 255 characters
    - Is a valid email address
- `password`
    - Not null or empty
    - Not longer than 50 characters
- `confirmPassword`
    - Not null or empty
    - Not longer than 50 characters

Here's what the initial pass at setting up the validators looks like using the `express-validator` library:

```
const userValidators = [
  check('firstName')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for First Name')
    .isLength({ max: 50 })
    .withMessage('First Name must not be more than 50 characters long'),
  check('lastName')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Last Name')
    .isLength({ max: 50 })
    .withMessage('Last Name must not be more than 50 characters long'),
  check('emailAddress')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Email Address')
    .isLength({ max: 255 })
    .withMessage('Email Address must not be more than 255 characters long')
    .isEmail()
    .withMessage('Email Address is not a valid email'),
  check('password')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Password')
    .isLength({ max: 50 })
    .withMessage('Password must not be more than 50 characters long'),
  check('confirmPassword')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Confirm Password')
    .isLength({ max: 50 })
    .withMessage('Confirm Password must not be more than 50 characters long'),
];
```

You can use a regular expression to enforce password complexity:

```
check('password')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Password')
    .isLength({ max: 50 })
    .withMessage('Password must not be more than 50 characters long')
    .matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#$%^&*])/, 'g')
    .withMessage('Password must contain at least 1 lowercase letter, uppercase le
```

`^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#$%^&*])`

This article from The Polyglot Developer breaks down the RegEx that tests the password strength in the validation above. To recap:

- The hat operator `^` is used to start matching at the beginning of the password.
- The expression `(?=.*[a-z])` is used to check that the password contains at least one lowercase character.
- The expression `(?=.*[A-Z])` is used to check that the password contains at least one uppercase character.
- The expression `(?=.*[0-9])` is used to check that the password contains at least one numeric character.
- The expression `(?=.*[!@#$%^&*])` is used to check that the password contains at least one special character.

The validation below checks that the two passwords should match:

```
check('confirmPassword')
  .exists({ checkFalsy: true })
  .withMessage('Please provide a value for Confirm Password')
  .isLength({ max: 50 })
  .withMessage('Confirm Password must not be more than 50 characters long')
  .custom((value, { req }) => {
    if (value !== req.body.password) {
      throw new Error('Confirm Password does not match Password');
    }
    return true;
  })
```

You can use another custom validator to check if the provided email address is already in use by another account:

```
check('emailAddress')
  .exists({ checkFalsy: true })
  .withMessage('Please provide a value for Email Address')
  .isLength({ max: 255 })
  .withMessage('Email Address must not be more than 255 characters long')
  .isEmail()
  .withMessage('Email Address is not a valid email'),
  .custom((value) => {
    return db.User.findOne({ where: { emailAddress: value } })
      .then((user) => {
        if (user) {
          return Promise.reject('The provided Email Address is already in use by
        }
      });
  }),
```

Hashing user passwords

To keep your user's personal information as secure as possible, you need to avoid storing user passwords in clear text. You can do this by encrypting the password with BCrypt, a password hashing function.

Install the `bcryptjs` npm package:

```
npm install bcryptjs
```

Update the `POST` `/user/register` route handler to use `bcrypt` to hash the user's password:

```
const bcrypt = require('bcryptjs');

// Code removed for brevity.

router.post('/user/register', csrfProtection, userValidators,
  asyncHandler(async (req, res) => {
    const {
      emailAddress,
      firstName,
      lastName,
      password,
    } = req.body;

    const user = db.User.build({
      emailAddress,
      firstName,
      lastName,
    });

    const validatorErrors = validationResult(req);

    if (validatorErrors.isEmpty()) {
      const hashedPassword = await bcrypt.hash(password, 10);
      user.hashedPassword = hashedPassword;
      await user.save();
      res.redirect('/');
    } else {
      const errors = validatorErrors.array().map((error) => error.msg);
      res.render('user-register', {
        title: 'Register',
        user,
        errors,
        csrfToken: req.csrfToken(),
      });
    }
  }));
```

Notice that the asynchronous method `bcrypt.hash()` is called to hash the `password` variable. The hashed value returned from the method call is used to set the `user.hashedPassword` property.

Adding the user routes to the `app` module

In the `app` module, add the user routes:

```
const bookRoutes = require('./routes/book');
const userRoutes = require('./routes/user');

// Code removed for brevity.

app.use(bookRoutes);
app.use(userRoutes);
```

Testing user registration

Run the application (`npm start`) and test the `/user/register` route by filling out and submitting the form to create a new user.

Using psql, view the user in the database by running the following SELECT SQL statement:

```
select * from "Users";
```

You should see a user with a hashed password like in this example:

```
 id |    emailAddress    | firstName | lastName  |
----+--------------------+-----------+-----------+-----------------------------
  1 | james@smashdev.com | James     | Churchill | \x24326124303824446c32684c546e
(1 row)
```

Supporting user login

Now that you have a way to support user registration, you need a way to allow existing users to log in using their email address and password.

Adding the user login form

Start by adding the routes and validators for the "Login" page to the `./routes/user` module just below the existing routes for the "Register" page:

```
router.get('/user/login', csrfProtection, (req, res) => {
  res.render('user-login', {
    title: 'Login',
    csrfToken: req.csrfToken(),
  });
});

const loginValidators = [
  check('emailAddress')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Email Address'),
  check('password')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Password'),
];

router.post('/user/login', csrfProtection, loginValidators,
  asyncHandler(async (req, res) => {
    const {
      emailAddress,
      password,
    } = req.body;

    let errors = [];
    const validatorErrors = validationResult(req);

    if (validatorErrors.isEmpty()) {
      // TODO Attempt to login the user.
    } else {
      errors = validatorErrors.array().map((error) => error.msg);
    }

    res.render('user-login', {
      title: 'Login',
      emailAddress,
      errors,
      csrfToken: req.csrfToken(),
    });
  }));
```

> Notice the slightly different approach of declaring the `errors` array outside of the `else` block. Using this approach will allow you to manually add an error message to the `errors` array if the user login process fails (you'll implement this process in just a bit).

Then add the `user-login.pug` template file to the `views` folder containing the following code:

```
//- ./views/user-login.pug

extends layout.pug

include utils.pug

block content
  +validationErrorSummary(errors)
  form(action='/user/login' method='post')
    input(type='hidden' name='_csrf' value=csrfToken)
    +field('Email Address', 'emailAddress', emailAddress)
    +field('Password', 'password', null, 'password')
    div(class='py-4')
      button(type='submit' class='btn btn-primary') Login
      a(href='/' class='btn btn-warning ml-2') Cancel
```

Implementing the user login process

To implement the user login process, the following steps need to be followed:

1. Attempt to retrieve the user from the database using the supplied email address.

```
// Attempt to get the user by their email address.
const user = await db.User.findOne({ where: { emailAddress } });
```

2. If a user was found in the database, then use the `bcrypt.compare()` method to compare the supplied password to the user's hashed password.

```
// Attempt to get the user by their email address.
const user = await db.User.findOne({ where: { emailAddress } });

if (user !== null) {
  // If the user exists then compare their password
  // to the provided password.
  const passwordMatch = await bcrypt.compare(password, user.hashedPassword.toStri
}
```

3. If the hashed passwords match (i.e. the `bcrypt.compare()` method returns `true`), then login the user and redirect them to the default route (i.e. `/`).

```
// Attempt to get the user by their email address.
const user = await db.User.findOne({ where: { emailAddress } });

if (user !== null) {
  // If the user exists then compare their password
  // to the provided password.
  const passwordMatch = await bcrypt.compare(password, user.hashedPassword.toStri

  if (passwordMatch) {
    // If the password hashes match, then login the user
    // and redirect them to the default route.
    // TODO Login the user.
    return res.redirect('/');
  }
}
```

> **Note:** We'll implement a method to "login" the user in just a bit. For now, add a `TODO` comment as a placeholder for the actual method call.

4. If a user wasn't found in the database or the hashed passwords don't match, then add a validation error message and render the `user-login` view to let the user know that the login process failed.

```
// Attempt to get the user by their email address.
const user = await db.User.findOne({ where: { emailAddress } });

if (user !== null) {
  // If the user exists then compare their password
  // to the provided password.
  const passwordMatch = await bcrypt.compare(password, user.hashedPassword.toStri

  if (passwordMatch) {
    // If the password hashes match, then login the user
    // and redirect them to the default route.
    // TODO Login the user.
    return res.redirect('/');
  }
}

// Otherwise display an error message to the user.
errors.push('Login failed for the provided email address and password');
```

Notice that you're not letting the user know if the supplied email address or password is to blame for the failed login attempt. This is intentional! Not providing this information, while potentially frustrating to end users, makes it more difficult for hackers to guess at email address and password combinations.

Here's the completed `POST` `/user/login` route:

```
router.post('/user/login', csrfProtection, loginValidators,
  asyncHandler(async (req, res) => {
    const {
      emailAddress,
      password,
    } = req.body;

    let errors = [];
    const validatorErrors = validationResult(req);

    if (validatorErrors.isEmpty()) {
      // Attempt to get the user by their email address.
      const user = await db.User.findOne({ where: { emailAddress } });

      if (user !== null) {
        // If the user exists then compare their password
        // to the provided password.
        const passwordMatch = await bcrypt.compare(password, user.hashedPassword.

        if (passwordMatch) {
          // If the password hashes match, then login the user
          // and redirect them to the default route.
          // TODO Login the user.
          return res.redirect('/');
        }
      }

      // Otherwise display an error message to the user.
      errors.push('Login failed for the provided email address and password');
    } else {
      errors = validatorErrors.array().map((error) => error.msg);
    }

    res.render('user-login', {
      title: 'Login',
      emailAddress,
      errors,
      csrfToken: req.csrfToken(),
    });
  }));
```

Improving the user navigation

One small change to the `user-register` and `user-login` views before you test the user login form. As a convenience for the user, add links below the user registration and login forms that allow the user to easily navigate between the "Register" and "Login" pages:

```
//- ./views/user-register.pug

extends layout.pug

include utils.pug

block content
  +validationErrorSummary(errors)
  form(action='/user/register' method='post')
    input(type='hidden' name='_csrf' value=csrfToken)
    +field('First Name', 'firstName', user.firstName)
    +field('Last Name', 'lastName', user.lastName)
    +field('Email Address', 'emailAddress', user.emailAddress)
    +field('Password', 'password', user.password, 'password')
    +field('Confirm Password', 'confirmPassword', '', 'password')
    div(class='py-4')
      button(type='submit' class='btn btn-primary') Register
      a(href='/' class='btn btn-warning ml-2') Cancel
  div
    p: a(href='/user/login') Already have an account?
```

```
//- ./views/user-login.pug

extends layout.pug

include utils.pug

block content
  +validationErrorSummary(errors)
  form(action='/user/login' method='post')
    input(type='hidden' name='_csrf' value=csrfToken)
    +field('Email Address', 'emailAddress', emailAddress)
    +field('Password', 'password', null, 'password')
    div(class='py-4')
      button(type='submit' class='btn btn-primary') Login
      a(href='/' class='btn btn-warning ml-2') Cancel
  div
    p: a(href='/user/register') Don't have an account?
```

Testing user login

Run the application (if it's not already running) and browse to the `/user/login` route. Test the user login form by completing the following actions:

- Submit the form with no values.
  - You should see two validation messages asking you to provide values.
- Submit the form with an email address that isn't associated with a user record in the database and a password (doesn't matter if the password is correct or not).
  - You should see a validation message letting you know that the login attempt failed.
- Submit the form with an email address that's associated with a user record in the database **but with an incorrect password**.
  - You should see a validation message letting you know that the login attempt failed.

- Submit the form with an email address that's associated with a user record in the database **and with a correct password**.
    - This time you should be redirected to the "Home" page.

The login process succeeded, but a crucial piece is still missing: persisting the user's login state.

Persisting user login state

Now it's time to handle persisting the user's login state after they've successfully logged into the website!

Remember that HTTP is a stateless protocol. Each HTTP request is independent from other requests that were executed before or after. Once the server has processed an incoming request and returned a response, it forgets about the client.

To persist to user's login state, we can implement and use sessions!

Configuring Express to use sessions

Install the `express-session` npm package.

```
npm install express-session
```

Add a new environment variable named `SESSION_SECRET` to the `.env` file:

```
SESSION_SECRET=f1f079b1-68fe-4324-8010-0a5cff63a288
```

Don't forget to update the `.env.example` file too:

```
SESSION_SECRET=«strong session secret»
```

After updating the `.env` file, update the `config` module to export a property named `sessionSecret` initialized to the `process.env.SESSION_SECRET` property value:

```
// ./config/index.js

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

Now it's time to add the `express-session` middleware to the `app` module Start with importing the `express-session` module:

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

Then use the `require()` function to get the session secret environment variable from the `config` module:

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

Now you can add the `session` middleware to the application just after the call to the `app.use()` method that adds the `cookieParser` middleware:

```
app.use(cookieParser(sessionSecret));
app.use(session({
  name: 'reading-list.sid',
  secret: sessionSecret,
  resave: false,
  saveUninitialized: false,
}));
```

Some things to note about the above code:

- Notice that the `sessionSecret` config value is passed to the `cookieParser` middleware. If your application is using both `cookie-parser` and `express-session` they need to use the same `secret` value.
- The `name` option is set to `reading-list.sid` so that session cookies for the Reading List application won't affect any other applications that are using the general `localhost` domain.

Here's what the top portion of the `app` module should look like now:

```
// ./app.js

const express = require('express');
const morgan = require('morgan');
const cookieParser = require('cookie-parser');
const session = require('express-session');

const { sessionSecret } = require('./config');
const bookRoutes = require('./routes/book');
const userRoutes = require('./routes/user');

const app = express();

app.set('view engine', 'pug');
app.use(morgan('dev'));
app.use(cookieParser(sessionSecret));
app.use(session({
  name: 'reading-list.sid',
  secret: sessionSecret,
  resave: false,
  saveUninitialized: false,
}));
app.use(express.urlencoded({ extended: false }));
app.use(bookRoutes);
app.use(userRoutes);

// Code removed for brevity.
```

> **Note:** Remember that the default in-memory session store (`MemoryStore`) is only meant for local development. While the in-memory session store works for the purposes of this article, you'd want to replace it with a more robust option (i.e. `connect-pg-simple` and `connect-session-sequelize`) before deploying your application to a production environment.

Using sessions to persist a user's login state

Now that you've configured sessions in Express application, you can persist the user's login state using a session.

Add a new module named `auth` to the root of your project and add the following code to the module:

```
// ./auth.js

const loginUser = (req, res, user) => {
  req.session.auth = {
    userId: user.id,
  };
};

module.exports = {
  loginUser,
};
```

Putting all of the authentication related code in its own module helps to keep things organized in the project. It also helps to keep your modules focused on solving a single problem or group of related problems. All of this will improve the readability and maintainability of your project.

Now update the `./routes/user` module to import the `loginUser()` function from the `auth` module:

```
const { loginUser } = require('../auth');
```

Then within the `POST` `/user/login` route handler add a call to the `loginUser()` function just before redirecting the user to the default route if the password matched:

```
// Code removed for brevity.

if (passwordMatch) {
  // If the password hashes match, then login the user
  // and redirect them to the default route.
  loginUser(req, res, user);
  return res.redirect('/');
}
```

You can also login the user after a new user has registered. In the `POST` `/user/register` route handler add a call to the `loginUser()` function after saving the user to the database but before redirecting then to the default route:

```
// Code removed for brevity.

if (validatorErrors.isEmpty()) {
  const hashedPassword = await bcrypt.hash(password, 10);
  user.hashedPassword = hashedPassword;
  await user.save();
  loginUser(req, res, user);
  res.redirect('/');
} else {
  // Code removed for brevity.
}
```

For your reference, here are the updated `POST` `/user/register` and `/user/login` route handlers:

```
router.post('/user/register', csrfProtection, userValidators,
  asyncHandler(async (req, res) => {
    const {
      emailAddress,
      firstName,
      lastName,
      password,
    } = req.body;

    const user = db.User.build({
      emailAddress,
      firstName,
      lastName,
    });

    const validatorErrors = validationResult(req);

    if (validatorErrors.isEmpty()) {
      const hashedPassword = await bcrypt.hash(password, 10);
      user.hashedPassword = hashedPassword;
      await user.save();
      loginUser(req, res, user);
      res.redirect('/');
    } else {
      const errors = validatorErrors.array().map((error) => error.msg);
      res.render('user-register', {
        title: 'Register',
        user,
        errors,
        csrfToken: req.csrfToken(),
      });
    }
  }));

// Code removed for brevity.

router.post('/user/login', csrfProtection, loginValidators,
  asyncHandler(async (req, res) => {
    const {
      emailAddress,
      password,
    } = req.body;

    let errors = [];
    const validatorErrors = validationResult(req);

    if (validatorErrors.isEmpty()) {
      // Attempt to get the user by their email address.
      const user = await db.User.findOne({ where: { emailAddress } });

      if (user !== null) {
        // If the user exists then compare their password
        // to the provided password.
        const passwordMatch = await bcrypt.compare(password, user.hashedPassword.

        if (passwordMatch) {
          // If the password hashes match, then login the user
          // and redirect them to the default route.
          loginUser(req, res, user);
          return res.redirect('/');
        }
      }

      // Otherwise display an error message to the user.
      errors.push('Login failed for the provided email address and password');
    } else {
      errors = validatorErrors.array().map((error) => error.msg);
    }

    res.render('user-login', {
      title: 'Login',
      emailAddress,
      errors,
      csrfToken: req.csrfToken(),
    });
  }));
```

Testing user login state persistence

Run the application (if it's not already running) and use the "Register" and "Login" pages to register a new user and login an existing user. Everything should work as it did before, but the user's login state is being persisted in session.

At this point in the project, there isn't any visual indication if the user is logged in or not (that's something that you'll fix in a bit). If you open the DevTools in Chrome and view the "Application" tab, you can view the cookies for `http://localhost:8080`. After registering a new user or logging in an existing user, you should see a cookie named `reading-list.sid`. That's the session cookie!

Restoring the authenticated user from session

Now that you're persisting a user's login state to session, you need to make that user's information easily accessible to your application when it's processing requests.

In the `auth` module, define a middleware function named `restoreUser()` to retrieve the user's information from the database if they're authenticated:

```
// ./auth.js

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

const loginUser = (req, res, user) => {
  req.session.auth = {
    userId: user.id,
  };
};

const restoreUser = async (req, res, next) => {
  // Log the session object to the console
  // to assist with debugging.
  console.log(req.session);

  if (req.session.auth) {
    const { userId } = req.session.auth;

    try {
      const user = await db.User.findByPk(userId);

      if (user) {
        res.locals.authenticated = true;
        res.locals.user = user;
        next();
      }
    } catch (err) {
      res.locals.authenticated = false;
      next(err);
    }
  } else {
    res.locals.authenticated = false;
    next();
  }
};

module.exports = {
  loginUser,
  restoreUser,
};
```

The `restoreUser()` middleware function starts by logging the `req.session` object to the console. Doing this will help with testing and debugging.

Then the function checks if the `req.session.auth` property is defined to determine if there's an authenticated user. If there is, then it uses destructuring to extract the `userId` from the `req.session.auth` property and calls the `db.User.findByPk()` method to retrieve the user from the database.

If the user is successfully retrieved from the database, then the `res.locals` object is used to define and set two properties:

- `authenticated` - Set to `true` to indicate that the current request has an authenticated user; and
- `user` - Set to the user that was just retrieved from the database.

The `res.locals` object is scoped to the current request and available to anything that follows the `restoreUser()` middleware function, including middleware and route handler functions and any views that are rendered as part of the current request/response cycle. It's a convenient way to pass values to other middleware, route handlers, or views.

If the `req.session.auth` property isn't defined or if the `db.User.findByPk()` method call throws an error then the `res.locals.authenticated` property is set to `false` to indicate that the current request doesn't have an authenticated user (i.e. it's an anonymous request).

After defining the `restoreUser()` function and exporting it from the `auth` module, you need to import it into the `app` module:

```
const { restoreUser } = require('./auth');
```

Then add the `restoreUser()` middleware function to the application just before the routes are added:

```
app.use(restoreUser);
app.use(bookRoutes);
app.use(userRoutes);
```

Retrieving the user's information from the database on every request, instead of storing the user's information in the session, ensures that the user's information doesn't get stale. While it's possible to refresh the session if the user were to change their information using the application, using that approach could break if a user can change their information using other means (e.g. a mobile app).

Displaying the user's login state

It's helpful to display to the end user whether or not they're currently logged in. A common approach is to display login and registration links or a welcome message in the header of the website.

If the user isn't logged in, they would see links to log in or register:

> Login | Register

If the user is logged in, they would be welcomed and have access to logging out:

> Welcome «current user name»! | Logout

To do that, update your `./views/layout.pug` template to the following:

```
doctype html
html
  head
    meta(charset='utf-8')
    meta(name='viewport' content='width=device-width, initial-scale=1, shrink-to-
    link(rel='stylesheet' href='https://stackpath.bootstrapcdn.com/bootstrap/4.4.
    title Reading List - #{title}
  body
    nav(class='navbar navbar-expand-lg navbar-dark bg-primary')
      a(class='navbar-brand' href='/') Reading List
      button(class='navbar-toggler' type='button' data-toggle='collapse' data-tar
        span(class='navbar-toggler-icon')
      div(class='collapse navbar-collapse' id='navbarText')
        ul(class='navbar-nav mr-auto')
          //- Empty menu keeps the content that follows the
          //- unordered list correctly positioned on the
          //- right side of the navbar.
        if locals.authenticated
          span(class='navbar-text px-4') Welcome #{user.firstName}!
          form(class='form-inline pr-4' action='/user/logout' method='post')
            button(class='btn btn-sm btn-warning' type='submit') Logout
        else
          span(class='navbar-text px-4')
            a(class='btn btn-sm btn-dark mr-2' href='/user/login') Login
            a(class='btn btn-sm btn-dark' href='/user/register') Register
    .container
      h2(class='py-4') #{title}
      block content
    script(src='https://code.jquery.com/jquery-3.4.1.slim.min.js' integrity='sha3
```

A good portion of the new code is related to styling the Bootstrap Navbar component. The section that's responsible for displaying the user's login state is this bit:

```
if locals.authenticated
  span(class='navbar-text px-4') Welcome #{user.firstName}!
  form(class='form-inline pr-4' action='/user/logout' method='post')
    button(class='btn btn-sm btn-warning' type='submit') Logout
else
  span(class='navbar-text px-4')
    a(class='btn btn-sm btn-dark mr-2' href='/user/login') Login
    a(class='btn btn-sm btn-dark' href='/user/register') Register
```

Notice that the view is using the `locals.authenticated` property to determine if the current user is logged in or not. `locals` within a view is the same object that's available via `res.locals`. Remember that the `restoreUser` function defined in the `auth` module is responsible for determining if there's an authenticated user stored in session and defining the initializing the `res.locals.authenticated` property to the appropriate value.

If the current user isn't logged in, then links (styled as buttons using Bootstrap CSS classes) are rendered to the "Login" and "Register" pages. If the current user is logged in, then a short, friendly "welcome" message is rendered along with a simple form that contains a single "Logout" submit button.

Now if you run and test your application, you'll see the current user's login state displayed in the header! If you log in and click the "Logout" button in the header, you'll receive a "Page Not Found" error. This is occurring because the `POST` `/user/logout` route doesn't exist. Time to fix that!

Implementing user logout

Before adding the new route to logout a user, define and export a `logoutUser()` function in the `auth` module:

```
const logoutUser = (req, res) => {
  delete req.session.auth;
};
```

The `logoutUser()` function uses the JavaScript `delete` operator to remove the `auth` property from the `req.session` object which destroys the user's persisted login state.

Now you're ready to add the `POST` `/user/logout` to the `./routes/user` module to process `POST` requests from the logout form. Start by importing the `logoutUser()` function from the `auth` module:

```
const { loginUser, logoutUser } = require('../auth');
```

Then add the new route:

```
router.post('/user/logout', (req, res) => {
  logoutUser(req, res);
  res.redirect('/user/login');
});
```

Notice that the route doesn't use the `csrfProtection` middleware. The `POST` `/user/logout` route isn't modifying any of the user's data in the database, so for simplicity's sake the route isn't requiring a valid CSRF token to be present on the request.

After calling the `logoutUser()` function to logout the user, the user is redirected to the `/user/login` route. In some situations, it'd be more appropriate to redirect the user to the default route, but in just a bit you're going to update the default route to only be visible if the current user is logged in. Given that, it's a better option to redirect the user to the "Login" page.

Testing the latest changes

Run the application (if it's not already running) and use the "Login" page to login an existing user. After logging into the website, you should now see the user's first name displayed in the header.

Remember that the `restoreUser()` function logs the `req.session` object to the console. After logging in, you should see something like this logged to the console:

```
Session {
  cookie: { path: '/', _expires: null, originalMaxAge: null, httpOnly: true },
  auth: { userId: 1 }
}
```

The `session.auth.userId` property value is the user ID of the currently logged in user.

Now if you click the "Logout" button, you'll be redirected to the "Login" page. In the console you'll see that the `session.auth` property is no longer defined on the `session` object:

```
Session {
  cookie: { path: '/', _expires: null, originalMaxAge: null, httpOnly: true }
}
```

Protecting a route

Now that the Reading List application supports user self-registration and login, it's time to restrict access to the routes that should only be accessible to authenticated users.

For some applications, you might need to restrict access to one or two routes. For the Reading List application, you'll restrict access to all of the routes in the `./routes/book` module so that the user needs to log in view their list of books (in just a bit you'll update the `Book` model so that each book record will be associated with a user) or to add, update, or delete a book.

Defining a middleware function to require an authenticated user

Add a new function named `requireAuth()` to the `auth` module. Update the `requireAuth()` function to redirect the user to the "Login" page if the `res.locals.authenticated` property is set to `false`, otherwise pass control to the next middleware function by calling the `next()` method:

```
// ./auth.js

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

const loginUser = (req, res, user) => {
  // Code removed for brevity.
};

const logoutUser = (req, res) => {
  delete req.session.auth;
};

const requireAuth = (req, res, next) => {
  if (!res.locals.authenticated) {
    return res.redirect('/user/login');
  }
  return next();
};

const restoreUser = async (req, res, next) => {
  // Code removed for brevity.
};

module.exports = {
  loginUser,
  logoutUser,
  requireAuth,
  restoreUser,
};
```

Import the `requireAuth` function into the `./routes/book` module:

```
const { requireAuth } = require('../auth');
```

Then add the `requireAuth` to every route:

```
// ./routes/book.js

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

const db = require('../db/models');
const { csrfProtection, asyncHandler } = require('./utils');
const { requireAuth } = require('../auth');

const router = express.Router();

router.get('/', requireAuth, asyncHandler(async (req, res) => {
  // Code removed for brevity.
}));

router.get('/book/add', requireAuth, csrfProtection, (req, res) => {
  // Code removed for brevity.
});

const bookValidators = [
  // Code removed for brevity.
];

router.post('/book/add', requireAuth, csrfProtection, bookValidators,
  asyncHandler(async (req, res) => {
    // Code removed for brevity.
  }));

router.get('/book/edit/:id(\\d+)', requireAuth, csrfProtection,
  asyncHandler(async (req, res) => {
    // Code removed for brevity.
  }));

router.post('/book/edit/:id(\\d+)', requireAuth, csrfProtection, bookValidators,
  asyncHandler(async (req, res) => {
    // Code removed for brevity.
  }));

router.get('/book/delete/:id(\\d+)', requireAuth, csrfProtection,
  asyncHandler(async (req, res) => {
    // Code removed for brevity.
  }));

router.post('/book/delete/:id(\\d+)', requireAuth, csrfProtection,
  asyncHandler(async (req, res) => {
    // Code removed for brevity.
  }));

module.exports = router;
```

Run the application (if it's not already running) and browse to the default route (`/`). You'll be redirected to the "Login" page. Go ahead and login using an existing account. After logging in, you'll be redirected to the default route (`/`) to view your list of books.

Associating data with a user

Now that you've restricted access to all of the book related routes so that a logged or authenticated user is required, you can update the `Book` model so that each book record will be associated with a user.

Defining an association

To start, run the following command in a terminal from the root of your project to remove all of the seed data from the database:

```
npx dotenv sequelize db:seed:undo:all
```

Then update the `Book` and `User` models as follows:

```
// ./db/models/book.js

'use strict';
module.exports = (sequelize, DataTypes) => {
  const Book = sequelize.define('Book', {
    // Code removed for brevity.
  }, {});
  Book.associate = function(models) {
    Book.belongsTo(models.User, {
      as: 'user',
      foreignKey: 'userId'
    });
  };
  return Book;
};
```

```
// ./db/models/user.js

'use strict';
module.exports = (sequelize, DataTypes) => {
  const User = sequelize.define('User', {
    // Code removed for brevity.
  }, {});
  User.associate = function(models) {
    User.hasMany(models.Book, {
      as: 'books',
      foreignKey: 'userId'
    });
  };
  return User;
};
```

Updating the models in this way creates a one-to-many association between the `User` and `Book` models (i.e. a user can have one or more books).

After defining the association in the models, you need to create a new migration to add the a foreign key column to the `Books` table in the database. Run the following command to add a skeleton migration file:

```
npx sequelize migration:generate --name update-book
```

Then update the contents of the `./db/migrations/[timestamp]-update-book.js` file to this:

```
'use strict';
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.addColumn('Book', 'userId', {
      allowNull: false,
      references: {
        model: 'Users',
        key: 'id',
      },
      type: Sequelize.INTEGER,
    });
  },
  down: (queryInterface, Sequelize) => {
  }
};
```

The above migration uses the `queryInterface.addColumn()` method to add a new column to the `Books` table. In this specific case, you're adding a column named `userId` that's a foreign key to the `Users` table `id` column.

Now apply the pending migration:

```
npx dotenv sequelize db:migrate
```

Updating the seed data

Open the `./db/seeders/[timestamp]-test-data.js` file and update its contents to this:

```
'use strict';

const bcrypt = require('bcryptjs');

module.exports = {
  up: async (queryInterface, Sequelize) => {
    const users = await queryInterface.bulkInsert('Users', [
      {
        emailAddress: 'john@smith.com',
        firstName: 'John',
        lastName: 'Smith',
        hashedPassword: bcrypt.hashSync('P@ssw0rd', 10),
        createdAt: new Date(),
        updatedAt: new Date()
      }
    ], { returning: true });

    return queryInterface.bulkInsert('Books', [
      {
        userId: users[0].id,
        title: 'The Martian',
        author: 'Andy Weir',
        releaseDate: new Date('2014-02-11'),
        pageCount: 384,
        publisher: 'Crown',
        createdAt: new Date(),
        updatedAt: new Date()
      },
      {
        userId: users[0].id,
        title: 'Ready Player One',
        author: 'Ernest Cline',
        releaseDate: new Date('2011-08-16'),
        pageCount: 384,
        publisher: 'Crown',
        createdAt: new Date(),
        updatedAt: new Date()
      },
      {
        userId: users[0].id,
        title: 'Harry Potter and the Sorcerer\'s Stone',
        author: 'J.K. Rowling',
```

```
        releaseDate: new Date('1998-10-01'),
        pageCount: 309,
        publisher: 'Scholastic Press',
        createdAt: new Date(),
        updatedAt: new Date()
      },
    ], {});
  },
  down: async (queryInterface, Sequelize) => {
    await queryInterface.bulkDelete('Books', null, {});
    return queryInterface.bulkDelete('Users', null, {});
  }
};
```

An additional call to the `queryInterface.bulkInsert()` method has been added to seed a test user, "John Smith", into the `Users` database table. Notice that an object literal has been supplied to the `queryInterface.bulkInsert()` method to specify the `returning` option. The `returning` optional configures the bulk insert to return the newly inserted data. This gives you a way to set the `userId` foreign key column when bulk inserting the test data into the `Books` table.

To seed the database, run the following command:

```
npx dotenv sequelize db:seed:all
```

Updating the book related routes

Now that books are associated with a user, you can update the default route to retrieve the list of books for the currently authenticated user:

```
router.get('/', requireAuth, asyncHandler(async (req, res) => {
  const books = await db.Book.findAll({ where: { userId: res.locals.user.id }, or
  res.render('book-list', { title: 'Books', books });
}));
```

The options object passed into the `db.Book.findAll()` method call has been updated with a `where` property. The `where` property is set to an object literal that defines the properties to filter the query results by. In this case, you're filtering by the `userId` column, using the `res.locals.user.id` property value.

When users initially create an account, they won't have any books in their reading list. You can update the `book-list` view to display a friendly message when that occurs:

```
// ./views/book-list.pug

extends layout.pug

block content
  div(class='py-3')
    a(class='btn btn-success' href='/book/add' role='button') Add Book
  if books && books.length > 0
    table(class='table table-striped table-hover')
      thead(class='thead-dark')
        tr
          th(scope='col') Title
          th(scope='col') Author
          th(scope='col') Release Date
          th(scope='col') Page Count
          th(scope='col') Publisher
          th(scope='col')
      tbody
        each book in books
          tr
            td= book.title
            td= book.author
            td= book.releaseDate
            td= book.pageCount
            td= book.publisher
            td
              a(class='btn btn-primary' href=`/book/edit/${book.id}` role='button
              a(class='btn btn-danger ml-2' href=`/book/delete/${book.id}` role='
  else
    p: em You don't have any books in your reading list!
```

When a user adds a new book, you need to update the book's `userId` property with the authenticated user's `id`:

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

    const book = db.Book.build({
      userId: res.locals.user.id,
      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(),
      });
    }
  }));
```

And lastly, you need to ensure that the current can only edit or delete their own books. To be clear, as long as the user uses the application's user interface to edit and delete books, this would never be an issue. This situation would only arise if a user maliciously edited their current URL to attempt to edit or delete a book that belonged to another user.

Luckily, this issue is easy to prevent. In the `./routes/book` module, add the following function above the route definitions:

```
const checkPermissions = (book, currentUser) => {
  if (book.userId !== currentUser.id) {
    const err = new Error('Illegal operation.');
    err.status = 403; // Forbidden
    throw err;
  }
};
```

The `checkPermissions()` function accepts a book and current user and throws an error if the book's associated user doesn't match the current user.

Then update each of the edit and delete routes to call the `checkPermissions()` function:

```
router.get('/book/edit/:id(\\d+)', requireAuth, csrfProtection,
  asyncHandler(async (req, res) => {
    const bookId = parseInt(req.params.id, 10);
    const book = await db.Book.findByPk(bookId);

    checkPermissions(book, res.locals.user);

    res.render('book-edit', {
      title: 'Edit Book',
      book,
      csrfToken: req.csrfToken(),
    });
  }));

router.post('/book/edit/:id(\\d+)', requireAuth, csrfProtection, bookValidators,
  asyncHandler(async (req, res) => {
    const bookId = parseInt(req.params.id, 10);
    const bookToUpdate = await db.Book.findByPk(bookId);

    checkPermissions(bookToUpdate, res.locals.user);

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

    const book = {
      title,
      author,
      releaseDate,
      pageCount,
      publisher,
    };

    const validatorErrors = validationResult(req);

    if (validatorErrors.isEmpty()) {
      await bookToUpdate.update(book);
      res.redirect('/');
    } else {
      const errors = validatorErrors.array().map((error) => error.msg);
      res.render('book-edit', {
        title: 'Edit Book',
        book: { ...book, bookId },
        errors,
        csrfToken: req.csrfToken(),
      });
    }
  }));

router.get('/book/delete/:id(\\d+)', requireAuth, csrfProtection,
  asyncHandler(async (req, res) => {
    const bookId = parseInt(req.params.id, 10);
    const book = await db.Book.findByPk(bookId);

    checkPermissions(book, res.locals.user);

    res.render('book-delete', {
      title: 'Delete Book',
      book,
      csrfToken: req.csrfToken(),
    });
  }));

router.post('/book/delete/:id(\\d+)', requireAuth, csrfProtection,
  asyncHandler(async (req, res) => {
    const bookId = parseInt(req.params.id, 10);
    const book = await db.Book.findByPk(bookId);

    checkPermissions(book, res.locals.user);

    await book.destroy();
    res.redirect('/');
  }));
```

Testing one more time

Now you're ready to do your final testing!

Login using the test user that you defined in your seed data. You should see the test user's list of books. Now logout the test user and login as another user. You should now see that user's list of books. If the user doesn't have any books in their reading list, add a new book. You should now only be seeing this user's books.

Congrats on completing the Reading List application... now with authentication!

The importance of using HTTPS

As a reminder, form fields are submitted in clear text to the server! This means that passwords can be *sniffed* or hijacked if the communication between the client and the server isn't encrypted. This brings us to why HTTPS (hypertext transfer protocol secure) is important. HTTP uses a digital certificate for authentication known as an SSL certificate to make a web application more secure. TSL and SSL protocol are often lumped together, as TLS is another protocol that encrypts communication between the client and server.

The use of HTTPS instead of HTTP means that TSL/SSL encryption is being used to keep form posts away from prying eyes!

In order to implement HTTPS, a domain requires an SSL certificate. SSL certificates can be purchased from companies that verify your identity so that you can be issued you a certificate for your domain. Once you have the certificate (which is simply a digital file), you then install the certificate on the server.

Typically, this server isn't your Node or Express application, but another web server that serves as a proxy server for your application. A proxy server receives all of the HTTP requests from the internet to your application's domain and forwards those requests to your application. Your application will then send responses to the proxy server, which then forwards them on to the client.

Next steps

There's so much more to learn about authentication! For example, it is important to know about implementing user roles as well as other user authentication flows. Examples of other user authentication processes include email confirmation, allowing users to reset their password, and two-factor authentication.

You'll dive into more components of user authentication as you learn more about web security and authentication!

Project: Amusement Park Tracker with Authentication

This project picks up where the first Amusement Park Tracker project left off! In the provided starter project, you can view, create, update, and delete both parks and attractions. In this project you'll extend the provided application with the following features:

- User self-registration and login; and
- Ability for users to record visits to park attractions.

Phase 0: Download the starter project

Clone the starter project:

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

Then complete the following set up steps:

- Create the database and limited access database user;
- Add an `.env` file containing the variables from the `.env.example` file;
- Install the project's dependencies (`npm install`); and
- Use the Sequelize CLI to apply the provided database migrations and seeder.

Now you can start (`npm start`) and test the application!

Phase 1: Create the User model

The `User` model should include the following properties:

- `emailAddress` - A non-nullable string (length: 255) representing the user's email address;
- `firstName` - A non-nullable string (length: 50) representing the user's first name;
- `lastName` - A non-nullable string (length: 50) representing the user's last name; and
- `hashedPassword` - A non-nullable binary string representing the user's hashed password.

Use the Sequelize CLI to generate the `User` model and migration. Then edit both files to use the expected attribute and column configuration. Remember to make the `hashedPassword` attribute have a datatype of `STRING.BINARY`.

Apply the migration when you're ready.

Phase 2: Configure Express to use sessions

Take a moment to install:

```
npm install express-session
```

Now let's configure your session store. Add the `express-session` middleware to the `app` module:

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

Make sure you have required `session` from the `express-session` package and have your application use the session. Make sure you configure the session with both `resave` and `saveUninitialized` set to `false`.

Take a moment set a `SESSION_SECRET` environment variable in your `.env` file. Add a key of `sessionSecret` connected to the `process.env.SESSION_SECRET` in your `./config/index.js` module as well. As a reminder, you can generate a UUID to have a more secure `sessionSecret` variable value.

In the `app` module, make sure to also import the `sessionSecret` in your `./config` require statement. As a reminder, be sure to use the same `secret` value for the `express-session` and `cookie-parser` middleware.

Your session should be configured like so:

```
app.use(session({
  secret: sessionSecret,
  resave: false,
  saveUninitialized: false,
}));
```

Phase 3: Support user self-registration

Now it's time to add the user registration form.

Begin by creating a `./routes/user` module. Import `express` and instantiate a `router` with `express.Router()`. Import your `db` from your `../db/models` and create `GET` and `POST` routes for the "Register" page (`/user/register`). Make sure to use CSRF protection as well the `bcryptjs` npm package to hash user passwords. Lastly, don't forget to export the router module you have just created.

Take note that you already have a `routes/utils.js` module that holds utility functions like the `csrfProtection` and `asyncHandler` methods you are familiar with. Import both of these methods into your `./routes/user` module with a require statement to the `./utils` module like so:

```
const { csrfProtection, asyncHandler } = require('./utils');
```

Now you can use your `csrfProtection` and `asyncHandler` in your `./routes/user` module!

In your GET `/user/register` route, use `db.User.build()` to initialize a new user to pass into the `user-register` view. Make sure to also pass in a `title` for your "Register" page as well as a `csrfToken` (with a value of `req.csrfToken()`). Add `csrfProtection` to your route, and let's create your `user-register` template.

Render a template that extends the main layout and has a form within `block content`. Take note of the mixins in the `utils.pug` file that are available for you to use. The form should contain the following input fields:

- First Name
- Last Name
- Email Address
- Password
- Confirm Password

Don't forget to have a hidden input field for your `_csrf` field as well as a submit button. Once you have your view set up, let's create your POST route for user registration!

In your POST `/user/register` route, make sure to use `userValidators` in addition to your `csrfProtection` middleware. This means you'll need to import `check` and `validationResult` from `express-validator`.

At this moment, implement the following validation rules:

- `firstName`
    - Not null or empty
    - Not longer than 50 characters
- `lastName`
    - Not null or empty
    - Not longer than 50 characters
- `emailAddress`
    - Not null or empty
    - Not longer than 255 characters
    - Is a valid email address
    - Should not be in use by an existing account
- `password`
    - Not null or empty
    - Not longer than 50 characters
    - Should contain at least 1 lowercase letter, uppercase letter, number, and special character (i.e. "!@#$%^&*") Hint: review the *Implementing Session-Based Authentication* reading and see below for reminders on how to
      use regex for validation!
- `confirmPassword`
    - Not null or empty
    - Not longer than 50 characters
    - Should match the provided `password` value

Regex Reminders

- The hat operator `^` is used to start matching at the beginning of the password.
- The expression `(?=.*[a-z])` is used to check that the password contains at least one lowercase character.
- The expression `(?=.*[A-Z])` is used to check that the password contains at least one uppercase character.
- The expression `(?=.*[0-9])` is used to check that the password contains at least one numeric character.
- The expression `(?=.*[!@#$%^&*])` is used to check that the password contains at least one special character.

Now return to your POST route and wrap your asynchronous route function with your `asyncHandler` so that you can `await` certain processes in your route. Begin by destructuring the `emailAddress`, `firstName`, `lastName`, and `password` from your `req.body` object. Then use the `emailAddress`, `firstName`, and `lastName` variables (but not the `password` variable) to build a user with the `db.User.build()` method.

At this point, generate your `validatorErrors` within your route by using the `validationResult` method from `express-validator`. If the `validatorErrors` are empty, `await` the generation of your `hashedPassword` created with `bcrypt.hash()`. Make sure import `bcrypt` by installing and requiring the `bcryptjs` package. Remember that the first argument of `bcrypt.hash()` is a password string. You can use use an integer for the second argument to auto-generate a salt that will be incorporated in the password hash process. After hashing the user password, set the `user.hashedPassword` property and `await` the save of your user instance. Lastly, redirect the user to the home page (`/`) upon successful registration.

If the `validatorErrors` are NOT empty, use `array()` to transform the `validatorErrors` object into a mappable array. Map over each `error` object in the array and pluck out each error's `msg` property to generate an array of error messages. Lastly, re-render your `user-register` form and pass in your `title` of "Register", the `user` object, the `errors` array, and the `csrfToken`.

Now run your application and test the `/user/register` route! Remember that you can test your route by registering a user through the form and using Postbird to confirm whether or not your user has been persisted to the database.

Phase 4: Support user login

Now it's time to add the user login form! Begin by updating the `./routes/user` module with `GET` and `POST` routes for the "Login" page (`/user/login`). Make sure to use CSRF protection for both routes.

Render your `user-login` template in your GET `/user/login` route. Pass along a `title` for your "Login" page as well as a `csrfToken`. Now let's create the view template for your login page!

Create a `user-login.pug` template in your views directory. Think of how you can include and re-use mixins from your `utils.pug` file just like in your "Register" page. The "Login" form should contain an "Email Address" field, a "Password" field, a hidden `_csrf` field, and a submit button.

Now let's revisit the POST `/user/login` route. You'll want to validate your login form data, so take a moment to implement the following validation rules:

- `emailAddress`
  - Not null or empty
- `password`
  - Not null or empty

After your `loginValidators` have been created, wrap your asynchronous route function within your `asyncHandler` function and destructure the `emailAddress` and `password` from your `req.body` object. Generate your `validatorErrors` by passing in the `req` body into the `validationResult` function. Also take a moment to initialize an `errors` array. You'll manually *push* error messages to render into this array.

If your validation errors are empty, try to find the user by their email address. You can `await` the database fetch of a `user` by using the `db.User.findOne()` function `where` the user has a matching `emailAddress`. If the user exists, use the `bcrypt.compare()` function to check whether the `user.hashedPassword` (parsed into string format) property matches the provided password from `req.body`. If there is a password match, log in the user (for now just leave yourself a `TODO` comment to log in the user) and redirect the user to the home page (`/`).

If your validator errors are empty and the user was not found, or the password did not match the `hashedPassword`, display an error message to the user by pushing in a "Login failed for the provided email address and password" message into the `errors` array you initialized.

If your validation errors are not empty, convert your `validatorErrors` object into an mappable `errors` array to pluck error messages and generate an array of error messages. Lastly, you need to render a `user-login` view for this route. Make sure to pass in the "Login" title, the `emailAddress` from `req.body`, the `errors` array, and a `csrfToken`.

Testing user login

Run the application and browse to the `/user/login` route. You can test the user login form with the following actions:

- Submit the form with no values.
  - You should see two validation messages asking you to provide values.
- Submit the form with an email address that isn't associated with a user record in the database and a password (doesn't matter if the password is correct or not).
  - You should see a validation message letting you know that the login attempt failed.
- Submit the form with an email address that's associated with a user record in the database **but with an incorrect password**.
  - You should see a validation message letting you know that the login attempt failed.
- Submit the form with an email address that's associated with a user record in the database **and with a correct password**.
  - This time you should be redirected to the "Home" page.

Phase 5: Persist user login state

Now it's time to handle persisting the user's login state after they've successfully logged into the website!

Using sessions to persist a user's login state

Add a new module named `auth` to the root of your project and add function named `loginUser()` to handle persisting a user's login state to session.

Update the `./routes/user` module to import the `loginUser()` function from the `auth` module. Then within the `POST` `/user/login` route handler add a call to the `loginUser()` function just before redirecting the user to the default route if the password matched.

Also, after a new user has registered in the `POST` `/user/register` route handler, add a call to the `loginUser()` function after saving the user to the database but before redirecting then to the default route.

Testing user login state persistence

Run the application (if it's not already running) and use the "Register" and "Login" pages to register a new user and login an existing user. Everything should work as it did before, but the user's login state is being persisted in session.

At this point in the project, there isn't any visual indication if the user is logged in or not (that's something that you'll fix in a bit). If you open your developer tools and view the "Application" tab, you can view the cookies for `http://localhost:8080`. After registering a new user or logging in an existing user, you should see a cookie named `reading-list.sid`. That's the session cookie!

Phase 6: Restore the authenticated user from session

Now that you're persisting a user's login state to session, you need to make that user's information easily accessible to your application when it's processing requests.

In the `auth` module, define a middleware function named `restoreUser()` to retrieve the user's information from the database if they're authenticated.

The function should check if the `req.session.auth` property is defined to determine if there's an authenticated user. If there is, extract the `userId` from the `req.session.auth` property and retrieve the user from the database.

If the user is successfully retrieved from the database, then use the `res.locals` object to define and set two properties:

- `authenticated` - Set to `true` to indicate that the current request has an authenticated user; and
- `user` - Set to the user that was just retrieved from the database.

If the `req.session.auth` property isn't defined or if retrieving the user from the database throws an error then set the `res.locals.authenticated` property to `false` to indicate that the current request doesn't have an authenticated user (i.e. it's an anonymous request).

After defining the `restoreUser()` function, export it from the `auth` module and import it into the `app` module. Then add the `restoreUser()` middleware function to the application just before the routes are added.

Phase 7: Display the user's login state

It's helpful to display to the end user whether or not they're currently logged in. A common approach is to display login and registration links or a welcome message in the header of the website.

If the user isn't logged in, they would see links to log in or register:

> Login | Register

If the user is logged in, they would be welcomed and have access to logging out:

> Welcome «current user name»! | Logout

To do that, update your `./views/layout.pug` template to use the `locals.authenticated` property to determine if the current user is logged in or not.

If the current user is logged in, then render a short, friendly "welcome" message is along with a simple form that contains a single "Logout" submit button:

```
span(class='navbar-text px-4') Welcome #{user.firstName}!
form(class='form-inline pr-4' action='/user/logout' method='post')
  button(class='btn btn-sm btn-warning' type='submit') Logout
```

If the current user isn't logged in, then render links (styled as buttons using Bootstrap CSS classes) to the "Login" and "Register" pages.

```
span(class='navbar-text px-4')
  a(class='btn btn-sm btn-dark mr-2' href='/user/login') Login
  a(class='btn btn-sm btn-dark' href='/user/register') Register
```

Now if you run and test your application, you'll see the current user's login state displayed in the header! If you log in and click the "Logout" button in the header, you'll receive a "Page Not Found" error. This is occurring because the `POST` `/user/logout` route doesn't exist. Time to fix that!

Phase 8: Implement user logout

Define and export a `logoutUser()` function in the `auth` module that removes the `auth` property from the `req.session` object.

Then add a `POST` `/user/logout` to the `./routes/user` module to process `POST` requests from the logout form. Import the `logoutUser()` function from the `auth` module and call it within the `POST` `/user/logout` route handler then redirect the user to the default route.

The `POST` `/user/logout` route isn't modifying any of the user's data in the database so there's no need to protect it from CSRF attacks.

Testing the latest changes

Run the application and use the "Login" page to login an existing user. You should now see the user's first name displayed in the header.

After logging in, you should see something like this logged to the console:

```
Session {
  cookie: { path: '/', _expires: null, originalMaxAge: null, httpOnly: true },
  auth: { userId: 1 }
}
```

Now click the "Logout" button, and you should be redirected to the "Login" page. In the console you should see that the `session.auth` property is no longer defined on the `session` object:

```
Session {
  cookie: { path: '/', _expires: null, originalMaxAge: null, httpOnly: true }
}
```

Phase 9: Support user attraction visits

Now you're ready to add support for user attraction visits.

Create the AttractionVisit model

The `AttractionVisit` model should include the following properties:

- `visitedOn` - A non-nullable date only attribute representing the date that the user visited the attraction;
- `rating` - A non-nullable integer attribute representing the user's rating of the attraction; and
- `comments` - A nullable text attribute representing the user's comments about the attraction.

Use the Sequelize CLI to generate the `AttractionVisit` model and migration. Then edit both files to use the expected attribute and column configuration.

Before applying the migration, associate the model with both the `Attraction` and `User` models:

```js
// ./db/models/attractionvisit.js

'use strict';
module.exports = (sequelize, DataTypes) => {
  const AttractionVisit = sequelize.define('AttractionVisit', {

    // Code removed for brevity.

  }, {});
  AttractionVisit.associate = function(models) {
    AttractionVisit.belongsTo(models.Attraction, {
      as: 'attraction',
      foreignKey: 'attractionId'
    });
    AttractionVisit.belongsTo(models.User, {
      as: 'user',
      foreignKey: 'userId'
    });
  };
  return AttractionVisit;
};
```

```js
// ./db/models/attraction.js

'use strict';
module.exports = (sequelize, DataTypes) => {
  const Attraction = sequelize.define('Attraction', {

    // Code removed for brevity.

  }, {});
  Attraction.associate = function(models) {
    Attraction.belongsTo(models.Park, {
      as: 'park',
      foreignKey: 'parkId'
    });
    Attraction.hasMany(models.AttractionVisit, {
      as: 'visits',
      foreignKey: 'attractionId'
    });
  };
  return Attraction;
};
```

```js
// ./db/models/user.js

'use strict';
module.exports = (sequelize, DataTypes) => {
  const User = sequelize.define('User', {

    // Code removed for brevity.

  }, {});
  User.associate = function(models) {
    User.hasMany(models.AttractionVisit, {
      as: 'visits',
      foreignKey: 'userId'
    });
  };
  return User;
};
```

Then update the `./db/migrations/[timestamp]-create-attraction-visit.js` migration file with the `userId` and `attractionId` foreign key columns:

```
'use strict';
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable('AttractionVisits', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      userId: {
        allowNull: false,
        references: {
          model: 'Users',
          key: 'id',
        },
        type: Sequelize.INTEGER,
      },
      attractionId: {
        allowNull: false,
        references: {
          model: 'Attractions',
          key: 'id',
        },
        type: Sequelize.INTEGER,
      },

      // Code removed for brevity.

    });
  },
  down: (queryInterface, Sequelize) => {
    return queryInterface.dropTable('AttractionVisits');
  }
};
```

These model associations create a one-to-many relationship between the `Attraction` and `AttractionVisit` models and a one-to-many relationship between the `User` and `AttractionVisit` model. Alternatively, you can think of the relationship as a many-to-many between the `Attraction` and `User` models (i.e. an attraction can be visited by many users and a user can visit many attractions).

Apply the migration when you're ready.

Update the Attraction Detail page

Update the Attraction Detail page to display a list of attraction visits. Display an "Add Visit" button (a hyperlink styled as a button using Bootstrap's CSS classes) above the list of attraction visits that when clicked, navigates the user to the "Add Visit" page.

Add the Add Visit page

Add a `./routes/visit` module, then add the `GET` and `POST` routes for the "Add Visit" page:

```
const visitValidators = [
  // TODO Define validators.
];

router.get('/attraction/:attractionId(\\d+)/visit/add', csrfProtection,
  asyncHandler(async (req, res) => {
    // TODO Implement route handler.
  }));

router.post('/attraction/:attractionId(\\d+)/visit/add', csrfProtection, visitVal
  asyncHandler(async (req, res) => {
    // TODO Implement route handler.
  }));
```

In your POST `/user/register` route, make sure to use `visitValidators` in addition to your `csrfProtection` middleware. This means you'll need to import `check` and `validationResult` from `express-validator`.

Implement the following validation rules:

- `visitedOn`
  - Not null or empty
  - Is a valid date
- `rating`
  - Not null or empty
  - Is an integer between 1 and 5

Render a template that extends the main layout and has a form within `block content`. Take note of the mixins and `validationErrorSummary` template that are available for you to use. The form should contain the following input fields:

- Visited On
- Rating
- Comments

Require a logged in user

To add a new visit, the user needs to be logged into the website. Without a logged in user, you wouldn't know who to add the visit for!

Add a new function named `requireAuth()` to the `auth` module. Update the `requireAuth()` function to redirect the user to the "Login" page if the `res.locals.authenticated` property is set to `false`, otherwise pass control to the next middleware function by calling the `next()` method.

Then import the `requireAuth()` function into the `./routes/visit` module and add it to the `GET` and `POST` routes for the "Add Visit" page:

```
router.get('/attraction/:attractionId(\\d+)/visit/add', requireAuth, csrfProtecti
  asyncHandler(async (req, res) => {
    // Code removed for brevity.
  }));

router.post('/attraction/:attractionId(\\d+)/visit/add', requireAuth, csrfProtect
  asyncHandler(async (req, res) => {
    // Code removed for brevity.
  }));
```

Now, if the current user isn't logged in, they'll be redirected to the "Login" page if they attempt view the "Add Visit" page!

Bonus Phase 1: Adding the Edit Visit and Delete Visit pages

- Add the edit and delete attraction visit routes and views.
- Only display the "Edit" and "Delete" buttons on a visit if the visit's user is the current user.
- Check that the current user is the owner of the visit before allowing them to edit or delete it.

Bonus Phase 2: Locking down parks and attractions

- Add an attribute to the `User` model that allows you to indicate which users are "Admin" users.
- Update the park and attraction CRUD routes to only allow authenticated "Admin" users.

WEEK-12 DAY-3
Data APIs With Express

Application Programming Interfaces Learning Objectives

Modern Web applications can serve both HTML *and* data. You see that when you go to Facebook, Twitter, and TurboTax Online. There are some widely-accepted practices for how to architect your applications so that they can do this. After homework, lecture, and practicing, you should be able to

- Recall that REST is an acronym for Representational State Transfer
- Describe how RESTful endpoints differ from traditional remote-procedure call (RPC) services
- Identify and describe the RESTful meanings of the combinations of HTTP verbs and endpoint types for both HTML-based applications and APIs
  - HTTP verbs: GET, POST, PUT, PATCH, and DELETE
  - Endpoint types: collections of resources and singular resources
- Recall that RESTful is *not* a standard (like ECMAScript or URLs), but a common way to organize data interactions
- Explain how RESTful APIs are meant to be stateless
- Given a data model, design RESTful endpoints that interact with the data model to define application functionality
- Use the `express.json()` middleware to parse HTTP request bodies with type `application/json`
- Determine the maximum data an API response needs and map the content from a Sequelize object to a more limited data object
- Define a global Express error handler to return appropriate responses and status codes given a specific Accept header in the HTTP request
- Define Cross-Origin Resource Sharing (CORS) and how it is implemented in some Web clients
- Explain that CORS is an opt-in feature of Web clients
- Configure your API to use CORS to prevent unauthorized access from browser-based Web requests

---

Getting Started With RESTful Endpoints

ReST stands for REpresentational State Transfer. The acronym doesn't fit perfectly, but we're developers and we can come up with whatever cool acronyms work for us! This may sound like a complex concept, but don't let it scare you too much.

In this reading, you will learn the definition of ReST and how to apply its design principles. You will learn about how it replaces another kind of interaction called "remote procedure calls". Finally, you'll get to a very practical way to implement ReST using the RESTful endpoint design convention.

Rules of ReST

ReST (Representational State Transfer) is an architecture style for designing networked applications. To be clear, ReST is not an official standard. Instead, it's a set of rules/constraints. Though ReST is commonly associated with APIs, not every API actually follows RESTful conventions.

ReST defines six architectural constraints, and in this reading, we'll focus on three of them:

1. **Decoupled client-server**: The client and the server should be decoupled so that they can evolve separately without any dependence on one another.
2. **Stateless**: This means that there is no necessary session between the client and the server. Data received from the server can be used by the client independently. This allows you to have short discrete operations. Luckily, this is a natural fit for HTTP operations in which requests are intended to be independent and short-lived.
3. **Uniform interface**: RESTful APIs are meant to be self-describing and uniform in their definition. Each operation is intended to be separated by a separate endpoint or URL. In practical real world terms, most RESTful APIs implement the classic CRUD (Create, Read, Update, Delete) operations against a resource that just happens to be in your data model. This uniformity allows developers to easily learn the usage pattern of each API.

What does a RESTful API look like?

Because RESTful APIs are meant to be representational, you can start with the data model that the API is meant to represent. For example, if you're building a Twitter clone application, you will most likely want to define API endpoints that manage the operations of your users "tweets", such as "create a tweet" and "like a tweet."

URLs

In RESTful APIs, you generally have two kinds of URLs, ones that points at *collections of resources* and ones that point at *single resources*. Using the Twitter application as an example, a path like **/my/tweets** would point to a collection of tweets made by you. A path like **/my/tweets/17** would point to a tweet made by you with the id of 17. Usually, those ids are the primary keys of rows in your database for the record that contains the information for that specific tweet.

The naming scheme, again, is quite simple. A path that ends in a plural noun represents a collection of resources that your API provides for developers to interact with. The following examples are just naming schemes that *you would decide* as the person creating the paths that your application will handle.

- **/invoices** would represent a collection of invoices that you're allowed to see
- **/people** would represent the people in the application that you're allowed to see
- **/houses** would represent a collection of houses

A path that combines a plural noun and a specific identifier represents a single resource in your application. Often, the identifier is a primary key from the database. However, if you have a unique column that also identifies a specific record, you could use that instead.

- **/invoices/PK-200201** would represent the single invoice that has the the invoice number "PK-200201"
- **/people/10103** would represent the single person with id 10103
- **/houses/bdfa5ef9-0c86-4810-bc13-10415250af09** would represent the house with the specific globally unique identifier "bdfa5ef9-0c86-4810-bc13-10415250af09"

AJAX URLs and HTTP verbs

For collections of resources URLs, the following table describes what each HTTP verb means for interacting with that URL. The responses will almost invariably be formatted as JSON.

HTTP Verb	Meaning	With respect to /my/tweets

HTTP Verb	Meaning	With respect to /my/tweets
GET	Get "all" of the specified resources	Get all of your tweets
POST	Create a new resource	Create a new tweet
PUT	n/a	n/a
PATCH	n/a	n/a
DELETE	Delete all of the resources	Delete all of your tweets

For single resource URLs, the following table describes what each HTTP verb means for interacting with that URL. Again, the responses will invariably be formatted as JSON.

HTTP Verb	Meaning	With respect to /my/tweets/17
GET	Get the details of the resource	Get that specific tweet with id 17
POST	n/a	n/a
PUT	Replace the resource	Replace all of the tweet details with the provided data
PATCH	Update the resource	Update specific properties of the tweet
DELETE	Delete the specified resource	Delete that specific tweet

When dealing with these RESTful endpoints, the actions don't necessarily mean that records *will* get created or destroyed in your database, or only one record will be affected.

For example, you may decide to perform "soft deletes" on some of your data. This means that, instead of removing a record from a table, you mark the record "deleted". (You can enable this with the `paranoid` configuration setting in your Sequelize models.) This means that the HTTP DELETE request would cause a SQL UPDATE
rather than DELETE. What is important is that the *concept* of a delete has been performed.

In another example, consider the path that reads **/weather/current**. That doesn't point to any static single record in the weather database. Instead, it would return the *most recent* record of weather in the database. The id of "current" would be treated special and initiate a lookup of the most recent record rather than a specific record like **/weather/10392**.

HTML URLs and HTTP verbs

RESTful APIs don't need to be just data APIs returning JSON. The URLs that return HTML can follow a RESTful concept, as well. What it means, though, is that you will be limited to just the verbs GET and POST. That's all HTML-based views can generate. The following tables show the paths and HTTP verbs used to interact with HTML-based versions of a RESTful application.

Path	HTTP Verb	Meaning
/my/tweets	GET	Get an HTML-based list of your tweets
/my/tweets/new	GET	Show a form to create a new tweet
/my/tweets	POST	Create a new tweet
/my/tweets/17	GET	See the details of your tweet with the id of 17

Path	HTTP Verb	Meaning
/my/tweets/17/edit	GET	Show the edit form for your tweet with the id of 17
/my/tweets/17	POST	Update the tweet with the submitted details
/my/tweets/17/delete	POST	Delete your tweet with the id of 17

All of the GET requests get HTML responses for the browser to show. All of the POST requests usually end in a redirect to another page that makes sense:

- After creating a resource, redirect to its detail page
- After editing a resource, redirect to its detail page
- After deleting a resource, redirect to the list page

Designing your API

The simplest way to figure out what paths you need in your application is to figure out what resources you need in your application. For many developers, this means looking at the models in your application (and the database tables that power the models).

If you're creating a pet shelter management application, you would need to know about animals, animal types, what kennel they're in, who works there, and more. So, you would have some models like

- `Animal`
- `AnimalType`
- `HealthRecord`
- `Kennel`
- and more...

For each of those, you could create paths in your RESTful API to allow the Web application to interact with those resources:

- **/animals** would be the collection of animals
- **/animal-types** would be the collection of animal types
- **/animals/137/health-records** would be the health records for the specific animal whose id is 137
- **/kennels** would be the collection of kennels available at the shelter in which you could put the animals
- and more...

It all depends on what you need to interact with to make your application work like you want it to.

GitHub API example

GitHub has a much vaunted RESTful API that many people use as a model for how to do *good API design*. This section checks out some of its features.

First, check out this GitHub REST API endpoint for a GET request to the `app-academy` user and the endpoint's response below. Notice how the information is formatted as JSON. JSON is the preferred format over other formats like XML. Feel free to take a look at the GitHub API documentation for sending a GET request for a single user.

If your browser is rendering the JSON below without the quotes around the property names, you're likely using the extension JSONView to *prettify* your JSON in JavaScript. The JSONView extension *prettifies* JSON by parsing the JSON text data into a JavaScript object. Note that you can also parse JSON into JavaScript by using the `JSON.parse()` method. Another extension, JSON Viewer, also *prettifies* JSON but it correctly preserves the quotes around the property names.

If you opened https://api.github.com/users/app-academy endpoint in your browser, you should see the JSON formatted data below. Feel free to navigate to the links in the JSON.

```
{
  "login": "app-academy",
  "id": 3155975,
  "node_id": "MDQ6VXNlcjMxNTU5NzU=",
  "avatar_url": "https://avatars0.githubusercontent.com/u/3155975?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/app-academy",
  "html_url": "https://github.com/app-academy",
  "followers_url": "https://api.github.com/users/app-academy/followers",
  "following_url": "https://api.github.com/users/app-academy/following{/other_use
  "gists_url": "https://api.github.com/users/app-academy/gists{/gist_id}",
  "starred_url": "https://api.github.com/users/app-academy/starred{/owner}{/repo}
  "subscriptions_url": "https://api.github.com/users/app-academy/subscriptions",
  "organizations_url": "https://api.github.com/users/app-academy/orgs",
  "repos_url": "https://api.github.com/users/app-academy/repos",
  "events_url": "https://api.github.com/users/app-academy/events{/privacy}",
  "received_events_url": "https://api.github.com/users/app-academy/received_event
  "type": "User",
  "site_admin": false,
  "name": null,
  "company": null,
  "blog": "",
  "location": null,
  "email": null,
  "hireable": null,
  "bio": null,
  "public_repos": 2,
  "public_gists": 1,
  "followers": 3,
  "following": 0,
  "created_at": "2012-12-31T00:08:43Z",
  "updated_at": "2016-02-27T05:27:57Z"
}
```

Navigate to the JSON's followers_url and repos_url. Notice how the url path of those endpoints connect to the endpoint that fetches all the data connected to the `app-academy` user.

Now open the followers_url endpoint in your browser. You are now are sending a `GET` request to GitHub's server to receive an array of followers associated to the `app-academy` user. Keep in mind that there are many Public APIs available as you plan your first full-stack project next week.

RESTful vs remote procedure calls (RPC)

Defining HTTP APIs can come in many shapes and styles. The style you adopt may flow out of the problem that you are attempting to solve or it could just be your personal preference. In programming, there is usually more than one valid approach, and making HTTP based APIs is no exception. And while RESTful APIs are certainly one of the most popular styles for creating an API, it is not the only way.

A remote procedure call

Remote procedure calls are like methods on objects rather than database operations. Clients make requests centered around specific operations. For example, with a RESTful API, you would see a GET request to this path to retrieve a specific tweet: `http://localhost/tweets/12`. In an RPC-based API, you could see a path similar to this `http://localhost/getTweetById?id=12`.

Your route would then have an action to handle that specific "call".

RPC style endpoints are notorious for specifying the method name in the URL.

This can be convenient, because by looking up the URL you can immediately understand what it does and what you get in return, and it can offer a way to get around some of the more complicated nested paths mentioned earlier.

The drawback is that you need comprehensive documentation to know all of the different methods available by an RPC-designed API. Contrast that to RESTful APIs where you need know only the resources; once you know the resources, you immediately know how to interact with them using the HTTP verbs.

Which is better? ReST or RPC?

"Better" is very subjective. RESTful APIs are definitely more popular nowadays. Although Express is an un-opinionated framework, a typical Express setup is more conducive to following RESTful conventions. For example, routers and routes are organized in a way to match RESTful URLs, and routes are defined around the core HTTP verbs.

Generally speaking, in this course, you'll want to use RESTful APIs, as it offers a predictable and systematic way of organizing your API.

What you learned

In this article, you covered quite a bit. You learned that

1. ReST is an acronym for Representational State Transfer.
2. RESTful services differ from traditional remote-procedure call styled services.
3. ReST is not an official standard.
4. RESTful APIs utilize a client/server architecture.
5. Data from your APIs are represented by resources and are accessed using Uniform Resource Locators (URLs).
6. Endpoints are a combination of a resource and an operation.
7. CRUD operations are represented by HTTP methods (POST, GET, PUT, DELETE, and sometimes PATCH) and a URL.
8. RESTful APIs are stateless.

Express APIs

It’s time to dive deeper into RESTful APIs. As you might remember, RESTful APIs utilize a client-server architecture. An example of client-server architecture that you are very familiar with is a web application loaded through a browser that connects the user to a server. RESTful APIs act as an intermediary to allow users to interact with database storage through CRUD operations.

In this reading, you'll be creating a lightweight API to manage tasks. You can imagine using this API in creating a To-Do list application. This reading will primarily be a walk-through and code demo of how to build out your own basic CRUD API with Express, Sequelize, and Postbird. You'll use Postbird to easily view your database objects. You'll also use Postman to send GET, POST, PUT, and DELETE requests to test your routes.

When you finish this article, you should be able to:

- Define a collection of endpoints (i.e. routes) that perform CRUD operations against a single resource using Sequelize and return responses to the client with appropriate HTTP status codes;
- Use the built-in `express.json()` middleware function to parse request body content as JSON;
- Use Postbird to view data in your database;
- Use Postman to interact with your API;
- Transform data for responses so that the content returned to clients only contains what's necessary and doesn't expose any sensitive information; and
- Understand that a global error handling function catches and processes all unhandled errors to return appropriate responses (status codes and content).

Project setup

Begin by cloning the project skeleton:

```
git clone https://github.com/appacademy-starters/express-sequelize-starter.git
```

Open your project's `package.json` and view the `dependencies` and `devDependencies`. The `package.json` has included the following packages as dependencies: `dotenv`, `express`, `express-validator`, `morgan`, `per-env`, `pg`, `sequelize`, `dotenv-cli`, `nodemon`, and `sequelize-cli`.

Since your `package.json` includes the packages above, running `npm install` will take care of installing all of the dependencies you need to create an Express application that utilizes Sequelize, dotenv, and the Express validator library. Your project tomorrow will also have a `package.json` with all the dependencies required of your project.

Initializing Sequelize

Normally, the Sequelize CLI needs to be configured in order to know where your database configuration is located and where to generate the `models`, `seeders`, and `migrations` folders. The project skeleton has already taken care of configuration by including complete `.sequelizerc`, `./config/index.js`, and `./config/database.js` files.

Creating the database

Take a moment to create a database user and database:

- The login name that you must use is "task_list_app" (make sure to set a login password).
- Your user must have the CREATEDB privilege so that you can run `npx dotenv sequelize-cli db:create`.
- The database name that you must use is "task_list_development".

Connecting Sequelize to your database

Although the Sequelize CLI configuration files are already complete, Sequelize CLI still needs access to your database credentials. Remember to create a `.env` file based off of the `.env.example` file included in your project skeleton, and to then replace the values with the values you used when creating the database. For example, if you used "password1234" for your password, then set the `DB_PASSWORD` to be equal to `password1234` in your own `.env` file.

Set up the Task model

Generate and migrate your `Task` model with the following command:

```
npx sequelize model:generate --name Task --attributes "name:string"
```

You should see that a `New model was created at...` and a `New migration was created at...` to confirm the successful creation of your `Task` model and migration files.

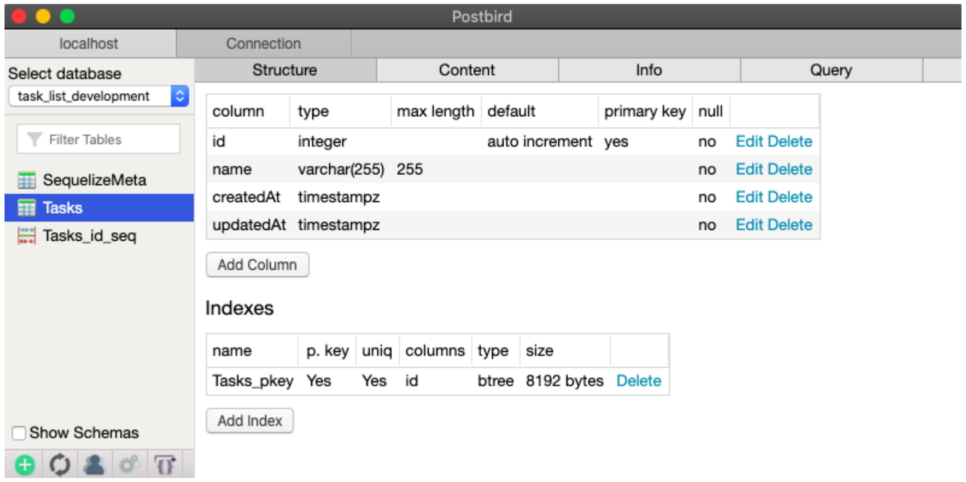
In the model and migration files, make sure to configure the task `name` to not be nullable. Each task should have a `name` column of type `string` that has a maximum length of `255` and is not nullable.

It's important to include `dotenv` in your migration command so that the Sequelize CLI uses correct database credentials when running the migration. Run your migrations with the command below:

```
npx dotenv sequelize db:migrate
```

After migrating, you can use the Postbird client to view the `Tasks` table in your database.

Postbird is a cross-platform PostgreSQL GUI client. A `GUI` is a graphical user interface, meaning that its users interact with an application through icons instead of text-based navigation. Postbird is an open-source GUI client while Psql is an interactive terminal - both allow you to interact with Postgres.



column	type	max length	default	primary key	null	
id	integer		auto increment	yes	no	Edit Delete
name	varchar(255)	255			no	Edit Delete
createdAt	timestampz				no	Edit Delete
updatedAt	timestampz				no	Edit Delete

Indexes

name	p. key	uniq	columns	type	size	
Tasks_pkey	Yes	Yes	id	btree	8192 bytes	Delete

Now generate a seed file by running the following command from the root of your project:

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

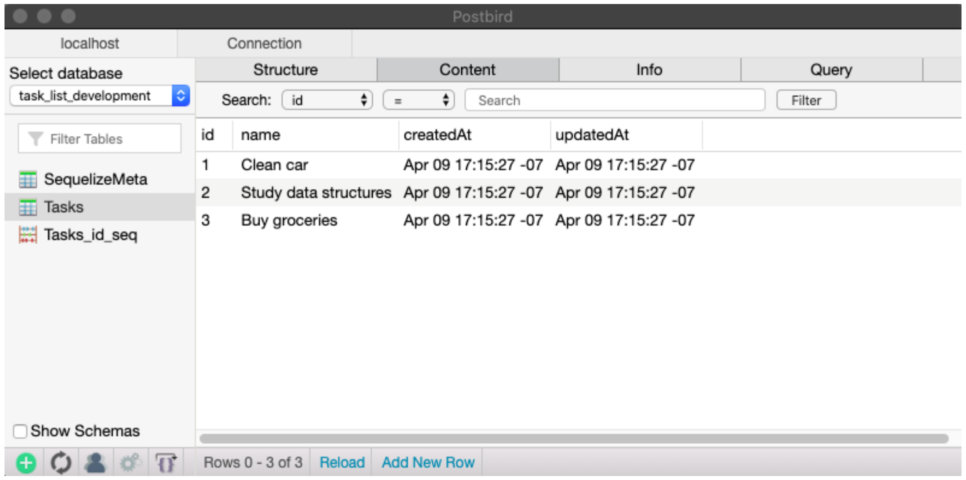
Go ahead and replace the contents of the `./db/seeders/[timestamp]-test-data.js` with the following code to seed four tasks into your `task_list_development` database:

```
//- ./db/seeders/[timestamp]-test-data.js

"use strict";

module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.bulkInsert(
      "Tasks",
      [
        {
          name: "Clean car",
          createdAt: new Date(),
          updatedAt: new Date(),
        },
        {
          name: "Study data structures",
          createdAt: new Date(),
          updatedAt: new Date(),
        },
        {
          name: "Buy groceries",
          createdAt: new Date(),
          updatedAt: new Date(),
        },
      ],
      {}
    );
  },

  down: (queryInterface, Sequelize) => {
    return queryInterface.bulkDelete("Tasks", null, {});
  },
};
```

Now run your seed files with the command below:

```
npx dotenv sequelize db:seed:all
```

After seeding, you should see the tasks you have created in your Postbird client:



id	name	createdAt	updatedAt
1	Clean car	Apr 09 17:15:27 -07	Apr 09 17:15:27 -07
2	Study data structures	Apr 09 17:15:27 -07	Apr 09 17:15:27 -07
3	Buy groceries	Apr 09 17:15:27 -07	Apr 09 17:15:27 -07

Initial router setup

Now remember that you are creating this API to manage tasks. You'll need a way to read a task, create a task, update a task, and delete a task in your database. You'll also need to render error messages to the API in case any of your requests are unsuccessful.

Your skeleton has a basic Express application with a generic error handling function set up in `app.js`, but you still need to create routes before you can test your API with Postman. Create a `routes` directory in the root if your project. Within the `routes` folder, create an `index.js` file for your root-level router and a `tasks.js` file for your task router. In both files, begin by requiring `express` and creating a `router` with `express.Router()`.

To test that your API is up and working properly, let's set up a route handler for a GET request to the `/` path in the `./routes/index.js` file. Have your result send JSON response with a message of `"Hello to the Express APIs demo app!"`.
Lastly, make sure you export the router you just created.

```js
//- ./routes/index.js

const express = require("express");
const router = express.Router();

router.get("/", (req, res) => {
  res.json({ message: "Hello to the Express APIs demo app!" });
});

module.exports = router;
```

Let's connect your `tasks.js` route to your `Task` database model. Begin by requiring the database models from `../db/models` and using destructuring to get a reference to the `Task` model. Then set up a route handler for a GET request to the `/` path. Make sure you export the router you just created.

```js
//- ./routes/tasks.js

const express = require("express");
const router = express.Router();
const db = require("../db/models");

const { Task } = db;

router.get("/", (req, res) => {
  // TODO: Fetch all tasks from the database (Read)
  // TODO: Render all tasks in JSON format
});

module.exports = router;
```

Since you are creating an API working with JSON data, you'll want to update your `app.js` file to have your application `use` the `express.json()` middleware. The `express.json()` middleware is needed to parse request body content formatted in JSON so that it is available via the `req.body` property.

```
app.use(express.json());
```

At this moment, you should notice that you have a route defined in the `app` module for the `/` path. Remove the GET route to `/` in your `app.js`.

Lastly, in order to connect to the routes you have just created, import your `indexRouter` and your `tasksRouter`. In this demo, you will be using separate routers. Feel free to take a moment to review last Tuesday's homework reading, "Using Separate Routers".

Mount your `indexRouter` with a path of `/` and mount your `tasksRouter` with a path of `/tasks`. Your `app.js` should look something like this:

```
//- app.js

const express = require("express");
const morgan = require("morgan");

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

const indexRouter = require("./routes/index");
const tasksRouter = require("./routes/tasks");

const app = express();

app.use(morgan("dev"));
app.use(express.json());

app.use("/", indexRouter);
app.use("/tasks", tasksRouter);

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

// Custom error handlers.

// Generic error handler.
app.use((err, req, res, next) => {
  res.status(err.status || 500);
  const isProduction = environment === "production";
  res.json({
    title: err.title || "Server Error",
    message: err.message,
    stack: isProduction ? null : err.stack,
  });
});

module.exports = app;
```

Now you can start your server and open `http://localhost:8080/` in your browser to test your API's index route! You should see your JSON response from the index route rendered in your application.

Global error handling

As a reminder, the function below is not an error handler. Remember that error handlers always define four parameters: `err`, `req`, `res`, and `next`. This middleware function is designed to match any failed requests so that it can create and pass a 404 error into the `next()` method, which prompts Express to handle the request as an error.

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

After calling the `next()` method with an argument, Express will invoke the first defined error handler. Remember that a global error handling function is defined to process unhandled errors.

The below error handler renders global error messages in JSON and returns a `500 Server Error` if the `err` does not have a valid status or title.

Remember that you don't want your error stack to be printed when running the application in production. This is why the `isProduction` variable is set to a boolean value determined by whether or not your application is in production. When your application is in production, the error will be rendered in JSON without the `err.stack`.

```
app.use((err, req, res, next) => {
  res.status(err.status || 500);
  const isProduction = environment === "production";
  res.json({
    title: err.title || "Server Error",
    message: err.message,
    stack: isProduction ? null : err.stack,
  });
});
```

Reading all tasks

Now you'll begin writing your routes to implement CRUD operations to your API.

As a reminder, CRUD stands for create, read, update, and delete. You'll interact with your `Task` model to implement CRUD features by using the following built-in methods:

- `Task.findAll()` to fetch all of your database tasks. (Read)
- `Task.findByPk()` to fetch a database task based on the `id` from your request parameters. (Read)
- `Task.create()` to generate a task in your database. (Create)
- `Task.update()` to update a task in your database. (Update)
- `Task.destroy()` to delete a task from your database. (Delete)

First up, the writing the route to fetch all tasks!

Let's bring back your `asyncHandler` to *catch* errors in a DRY way. Add the `asyncHandler` below to your `tasks.js` file:

```
const asyncHandler = (handler) => (req, res, next) =>
  handler(req, res, next).catch(next);
```

Now you can refactor the GET route to fetch all your tasks in the database. Because you are using the `async` keyword to pass the middleware function below into your `asyncHandler`, you can `await` database queries. Refactor your GET `/` route to `await` the fetch of all your tasks.

```
router.get(
  "/",
  asyncHandler(async (req, res) => {
    const tasks = await Task.findAll();
    // TODO: Render all tasks in JSON format
  })
);
```

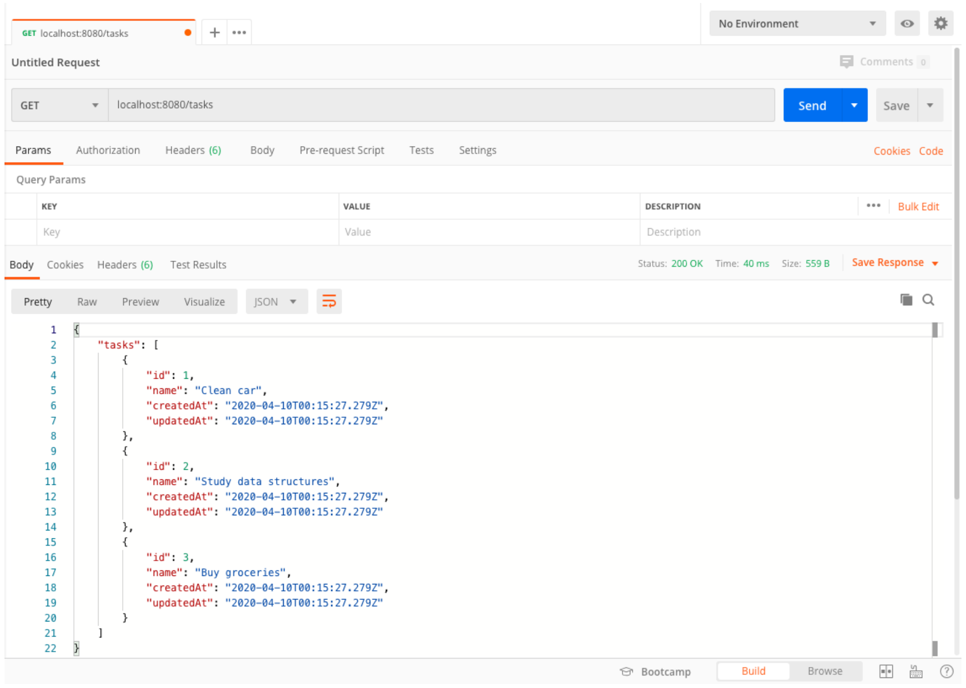
Now you'll want to render all tasks in JSON format by using `res.json({tasks})`.

Imagine you are rendering a list of users instead of a list of tasks. You wouldn't want to expose every user's private email or address. In the case of a user list, you could pluck out each user's `username` and decide to only send that property to the client. Remember that it's important to only send properties that the client needs and to be careful not to expose too much information.

At this point, your `tasks.js` file should look like this:

```
//- ./routes/tasks.js

const express = require("express");
const router = express.Router();
const db = require("../db/models");

const { Task } = db;

const asyncHandler = (handler) => (req, res, next) =>
  handler(req, res, next).catch(next);

router.get(
  "/",
  asyncHandler(async (req, res) => {
    const tasks = await Task.findAll();
    res.json({ tasks });
  })
);

module.exports = router;
```

Now start your server with `npm start` and go into Postman to make a GET request to `localhost:8080/tasks` and test the route you have just created! As a reminder, your request should go to `/tasks` even though you defined a router for the `/` path because of how you are using the `tasksRouter` with the `/tasks` path in your `app.js`. Every router you create in your `tasks.js` file just adds its path onto the `/tasks` route you designated for `tasksRouter`. You should see the all your tasks from the database render in the response `Body`.


GET localhost:8080/tasks

Untitled Request

GET localhost:8080/tasks

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (6) Test Results — Status: 200 OK Time: 40 ms Size: 559 B

Pretty Raw Preview Visualize JSON

```
{
    "tasks": [
        {
            "id": 1,
            "name": "Clean car",
            "createdAt": "2020-04-10T00:15:27.279Z",
            "updatedAt": "2020-04-10T00:15:27.279Z"
        },
        {
            "id": 2,
            "name": "Study data structures",
            "createdAt": "2020-04-10T00:15:27.279Z",
            "updatedAt": "2020-04-10T00:15:27.279Z"
        },
        {
            "id": 3,
            "name": "Buy groceries",
            "createdAt": "2020-04-10T00:15:27.279Z",
            "updatedAt": "2020-04-10T00:15:27.279Z"
        }
    ]
}
```

Note that if you use your browser to navigate to `localhost:8080/tasks`, you'll see the same response rendered as in Postman. This is because you are sending a GET request in Postman and your browser automatically sends a GET request when visiting `localhost:8080/tasks`.

Create a task

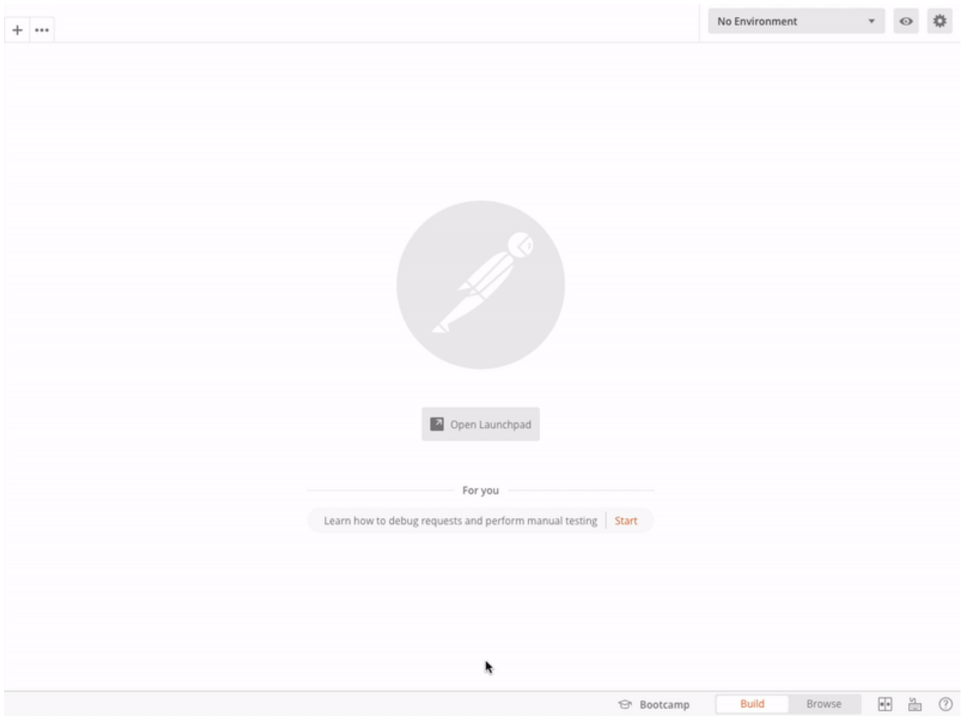
Now you'll work on setting up a POST `/tasks` route to create a task.

```
router.post(
  "/",
  asyncHandler(async (req, res) => {
    // TODO: Create and render a task in JSON
  })
);
```

Begin by extracting the task `name` from the `req.body`. Then, use `Task.create()` to to instantiate a `task` and save it to the database. Lastly, render the `task` in a JSON response with a created status of `201`. Your POST route should look something like this:

```
router.post(
  "/",
  asyncHandler(async (req, res) => {
    const { name } = req.body;
    const task = await Task.create({ name });
    res.status(201).json({ task });
  })
);
```

You can test this route by making a POST request to `/tasks` in Postman to create a task. Make sure to edit your `Headers` by adding a `Content-Type` header with a value of `application/json`. Send your POST request data through the `Body` by using the `raw` format and selecting `JSON` in the dropdown to create a task.



After you have created a task, verify your success in Postbird.

Validate data

Now it's time to validate your data. You can use the combination of the `express-validator` middleware with a custom middleware function to do so. The `express-validator` middleware will validate your data and the custom middleware function will create a `400 Bad request.` error response upon invalid data.

Import `check` and `validationResult` from `express-validator` to add validations to your application by adding the following line to in your `tasks.js` file:

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

Now create your array of validation middleware functions with `check` from `express-validator` to validate incoming task data:

```
const validateTask = [
  //  Task name cannot be empty:
  check("name")
    .exists({ checkFalsy: true })
    .withMessage("Task name can't be empty."),
  //  Task name cannot be longer than 255 characters:
  check("name")
    .isLength({ max: 255 })
    .withMessage("Task name can't be longer than 255 characters."),
];
```

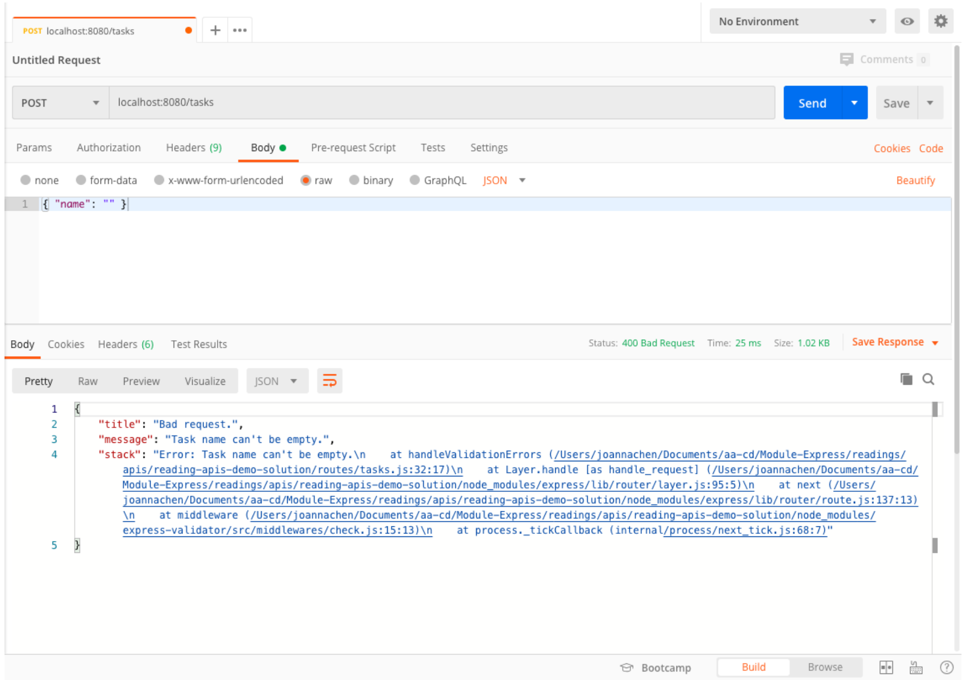
Now define a custom middleware function that checks to see if the request has any validation errors:

```
const handleValidationErrors = (req, res, next) => {
  const validationErrors = validationResult(req);

  // If the validation errors are empty,
  if (!validationErrors.isEmpty()) {
    // Generate an array of error messages
    const errors = validationErrors.array().map((error) => error.msg);

    // Generate a new `400 Bad request.` Error object
    // and invoke the next function passing in `err`
    // to pass control to the global error handler.
    const err = Error("Bad request.");
    err.status = 400;
    err.title = "Bad request.";
    err.errors = errors;
    return next(err);
  }

  // Invoke the next middleware function
  next();
};
```

Your validated POST route should look something like this:

```
router.post(
  "/",
  validateTask,
  handleValidationErrors,
  asyncHandler(async (req, res) => {
    const { name } = req.body;
    const task = await Task.create({ name });
    res.status(201).json({ task });
  })
);
```

You should now receive errors for creating a task with invalid data. Take note about how you are receiving an error `status` of `400`.



POST localhost:8080/tasks

Untitled Request

POST localhost:8080/tasks

```
{ "name": "" }
```

Status: 400 Bad Request Time: 25 ms Size: 1.02 KB

```
{
    "title": "Bad request.",
    "message": "Task name can't be empty.",
    "stack": "Error: Task name can't be empty.\n    at handleValidationErrors (/Users/joannachen/Documents/aa-cd/Module-Express/readings/apis/reading-apis-demo-solution/routes/tasks.js:32:17)\n    at Layer.handle [as handle_request] (/Users/joannachen/Documents/aa-cd/Module-Express/readings/apis/reading-apis-demo-solution/node_modules/express/lib/router/layer.js:95:5)\n    at next (/Users/joannachen/Documents/aa-cd/Module-Express/readings/apis/reading-apis-demo-solution/node_modules/express/lib/router/route.js:137:13)\n    at middleware (/Users/joannachen/Documents/aa-cd/Module-Express/readings/apis/reading-apis-demo-solution/node_modules/express-validator/src/middlewares/check.js:15:13)\n    at process._tickCallback (internal/process/next_tick.js:68:7)"
}
```

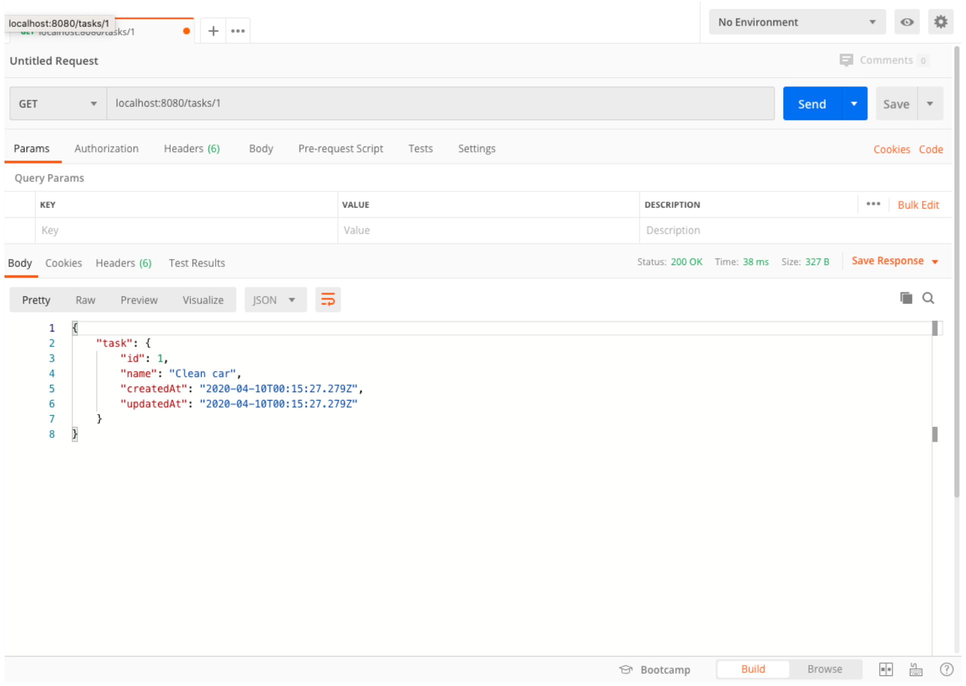
Read a task

Now it's time to set up a route for reading a specific task. You'll use the `id` from the `req.params` object. Begin by parsing the `id` and using `Task.findByPk()` to fetch the database task that matches the `id`. If the task was found, you'll render a JSON response.

As a reminder, `req.params` is an object that holds values parsed from your URL path. For example, reading the URL path to fetch your task with an `id` of `1` is `/tasks/1`. Since you have defined the GET route for a `/:id` path, your request parameters (`req.params`) will have a key of `id` with a value of `1`.

```
router.get(
  "/:id(\\d+)",
  asyncHandler(async (req, res, next) => {
    const taskId = parseInt(req.params.id, 10);
    const task = await Task.findByPk(taskId);
    res.json({ task });
  })
);
```

Test your new route with Postman!



GET localhost:8080/tasks/1

Status: 200 OK Time: 38 ms Size: 327 B

```
{
    "task": {
        "id": 1,
        "name": "Clean car",
        "createdAt": "2020-04-10T00:15:27.279Z",
        "updatedAt": "2020-04-10T00:15:27.279Z"
    }
}
```

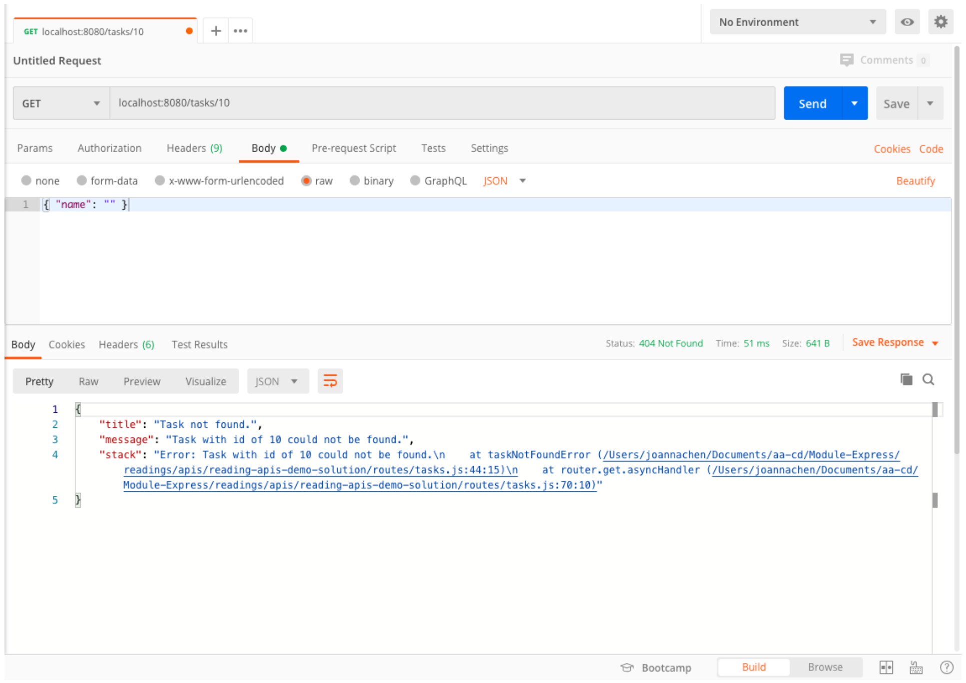
However, what if you are querying for a task `id` that is not in your system? You'll need to set up error handling to return a `404 Task not found.` error if the `id` is invalid. You'll want to instantiate a new `Error` object with an error message about the invalid `id`. The `Error` object should have a `title` of "Task not found." and a `status` of "404".

```
const taskNotFoundError = (id) => {
  const err = Error(`Task with id of ${id} could not be found.`);
  err.title = "Task not found.";
  err.status = 404;
  return err;
};
```

You'll also need to refactor your GET route for `/:id`. Make sure to add `next` as a parameter. You'll call the `next()` method by passing in the return value from the `taskNotFoundError(taskId)` function call.

```
router.get(
  "/:id(\\d+)",
  asyncHandler(async (req, res, next) => {
    const taskId = parseInt(req.params.id, 10);
    const task = await Task.findByPk(taskId);

    if (task) {
      res.json({ task });
    } else {
      next(taskNotFoundError(taskId));
    }
  })
);
```

Now test your `taskNotFoundError` function by making a request with an invalid `id`:



```
{ "name": "" }
```

```
{
    "title": "Task not found.",
    "message": "Task with id of 10 could not be found.",
    "stack": "Error: Task with id of 10 could not be found.\n    at taskNotFoundError (/Users/joannachen/Documents/aa-cd/Module-Express/
        readings/apis/reading-apis-demo-solution/routes/tasks.js:44:15)\n    at router.get.asyncHandler (/Users/joannachen/Documents/aa-cd/
        Module-Express/readings/apis/reading-apis-demo-solution/routes/tasks.js:70:10)"
}
```

Update a task

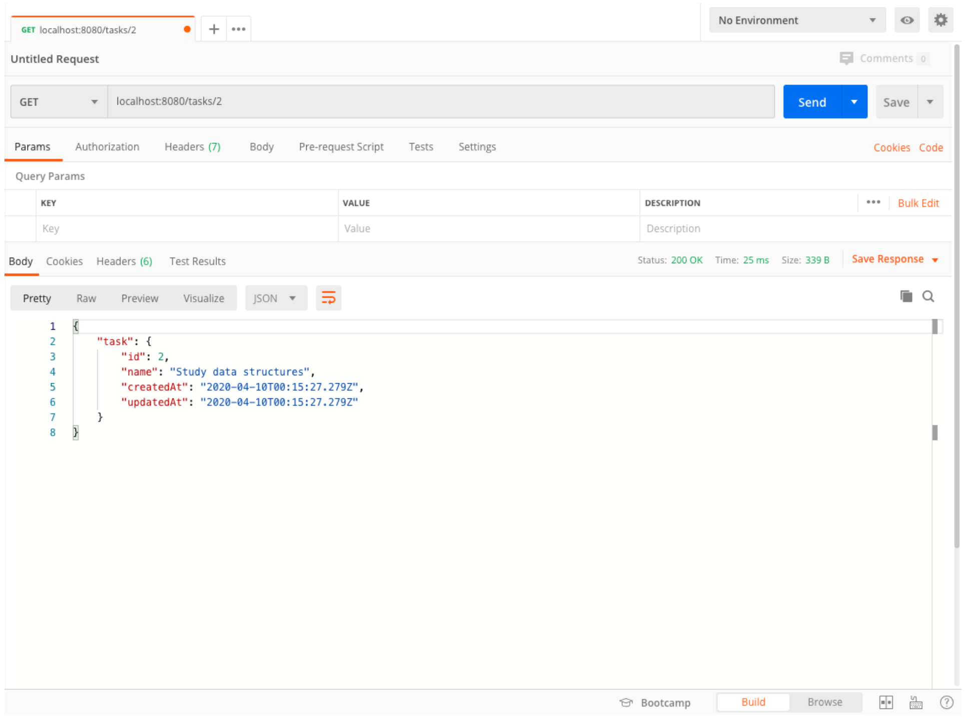
You can update a task with either a PUT or PATCH request. The key difference between PUT/PATCH is that a PUT request replaces an object in its entirety while a PATCH request only updates specific attributes.

It's time to set up a route for updating a task with a PUT request. Parse the task `id` within your `req.params` and use the parsed `taskId` to fetch the `task` to update. If you have a valid task, await the update and then render the updated `task` in JSON format.

Don't forget to use the same data validations that the create route used so that you receive the same type of error messages. Lastly, make sure to pass your `taskId` into your `taskNotFoundError` function call to generate a `404` error if the task does not exist. Then pass the return value of the `taskNotFoundError` function call into the `next` method to invoke the global error handler.

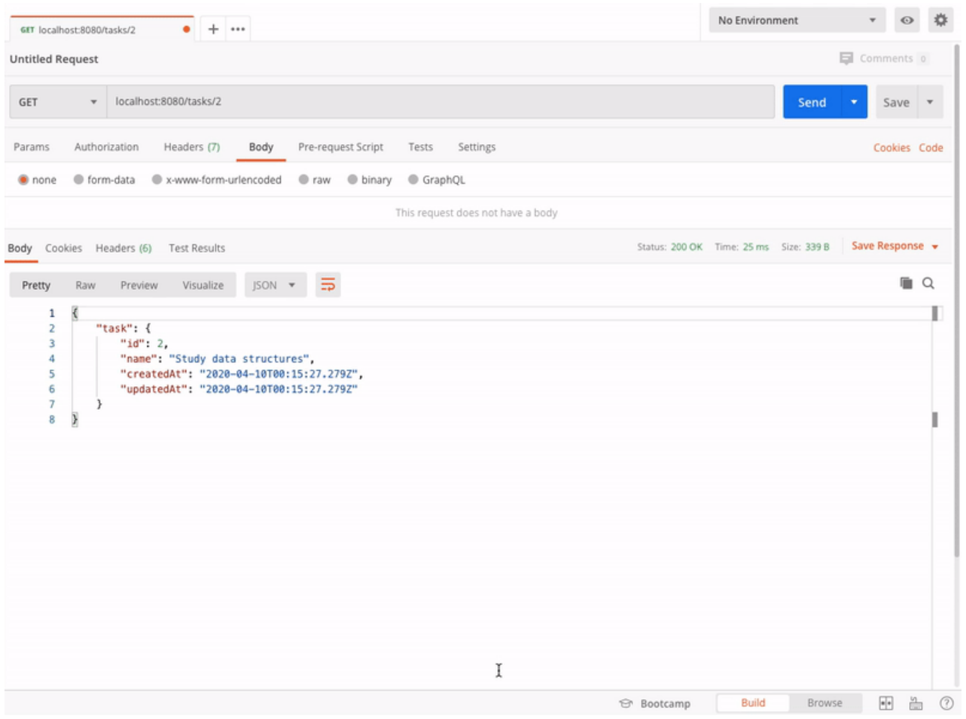
```
router.put(
  "/:id(\\d+)",
  validateTask,
  handleValidationErrors,
  asyncHandler(async (req, res, next) => {
    const taskId = parseInt(req.params.id, 10);
    const task = await Task.findByPk(taskId);

    if (task) {
      await task.update({ name: req.body.name });
      res.json({ task });
    } else {
      next(taskNotFoundError(taskId));
    }
  })
);
```

Begin by sending a GET request to `localhost:8080/tasks/2` to view the data of your second task:



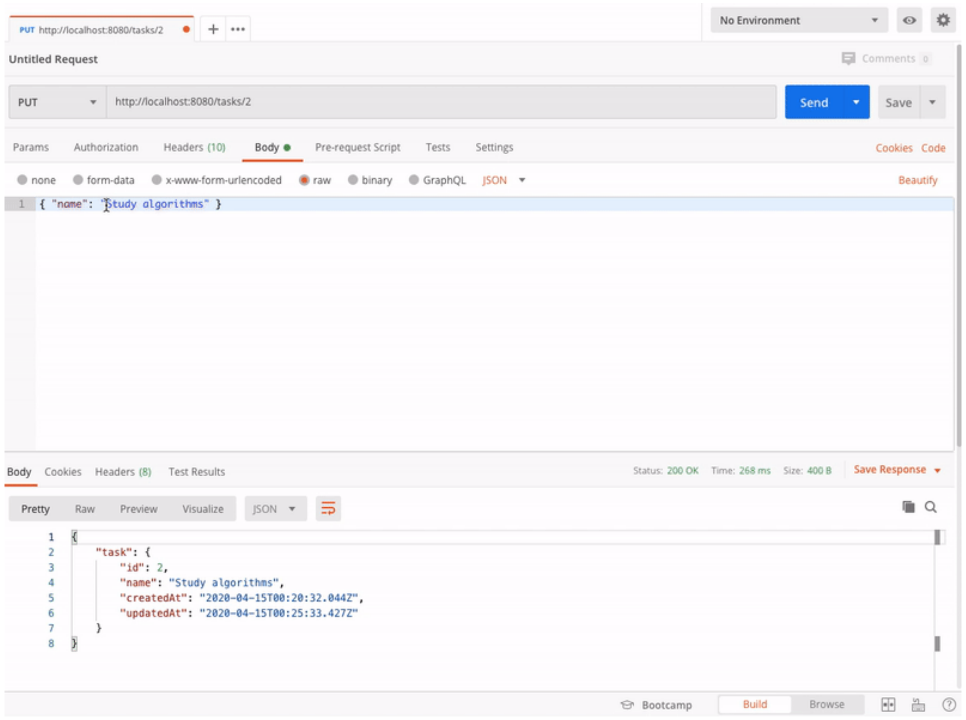
```
{
    "task": {
        "id": 2,
        "name": "Study data structures",
        "createdAt": "2020-04-10T00:15:27.279Z",
        "updatedAt": "2020-04-10T00:15:27.279Z"
    }
}
```

Like when testing your create route, make sure to add a `Content-Type` header with a value of `application/json`. Send your PUT request data through the `Body` by using the `raw` format and selecting `JSON` in the dropdown to update the task name:



```
{
    "task": {
        "id": 2,
        "name": "Study data structures",
        "createdAt": "2020-04-10T00:15:27.279Z",
        "updatedAt": "2020-04-10T00:15:27.279Z"
    }
}
```

Lastly, erase the `name` value and send a PUT request to test your error handling for invalid update data:



PUT http://localhost:8080/tasks/2

Untitled Request

PUT http://localhost:8080/tasks/2

```
{ "name": "Study algorithms" }
```

Status: 200 OK Time: 268 ms Size: 400 B

```
{
    "task": {
        "id": 2,
        "name": "Study algorithms",
        "createdAt": "2020-04-15T00:20:32.044Z",
        "updatedAt": "2020-04-15T00:25:33.427Z"
    }
}
```

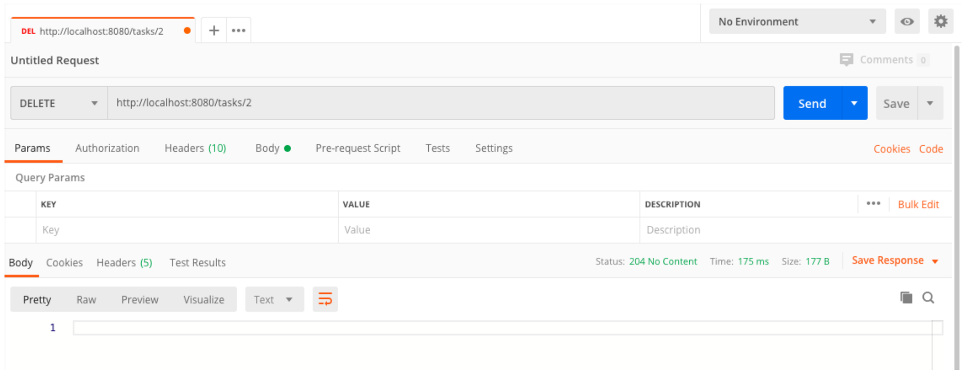
Delete a task

It's time to create the last route! Set up the route for deleting a task and verify that it was properly deleted using Postbird.

You'll begin by parsing the `taskId` and using the id to find your `task` to delete. If a valid `task` is found, destroy the task and render a `204 NO CONTENT` response with `res.status(204).end()` to confirm the deletion. If there is no valid task, pass the `taskId` into your `taskNotFoundError` function call to generate a `404` error. Then pass the return value of the `taskNotFoundError` function call into the `next` method to invoke the global error handler.

```
router.delete(
  "/:id(\\d+)",
  asyncHandler(async (req, res, next) => {
    const taskId = parseInt(req.params.id, 10);
    const task = await Task.findByPk(taskId);

    if (task) {
      await task.destroy();
      res.status(204).end();
    } else {
      next(taskNotFoundError(taskId));
    }
  })
);
```

Now test your route and send a DELETE request to `localhost:8080/tasks/2` to delete your task with an id of `2`:



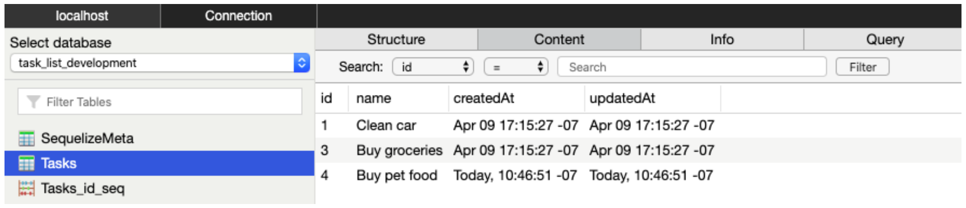
DEL http://localhost:8080/tasks/2

Untitled Request

DELETE http://localhost:8080/tasks/2

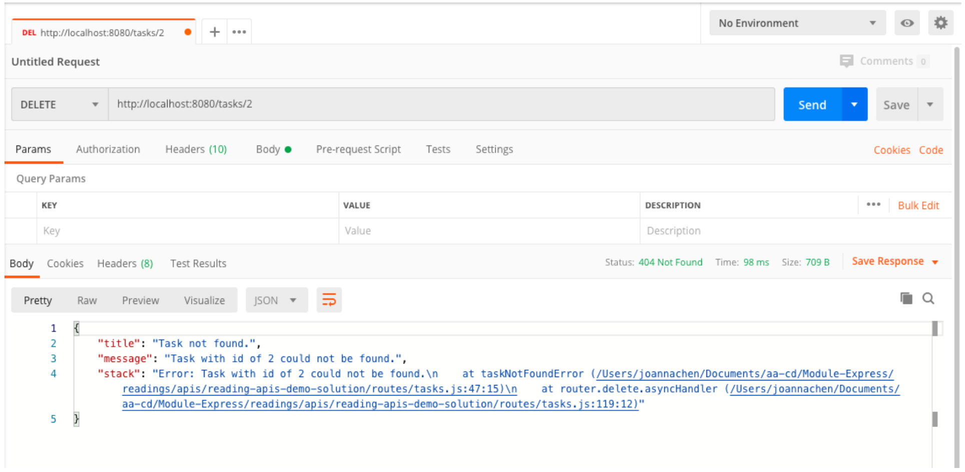
Status: 204 No Content Time: 175 ms Size: 177 B

Notice how there is no task with an id of `2` in the `Tasks` table `Content` in Postbird:



id	name	createdAt	updatedAt
1	Clean car	Apr 09 17:15:27 -07	Apr 09 17:15:27 -07
3	Buy groceries	Apr 09 17:15:27 -07	Apr 09 17:15:27 -07
4	Buy pet food	Today, 10:46:51 -07	Today, 10:46:51 -07

Now you'll want to test your `taskNotFoundError` handler to ensure that it takes care of trying to delete tasks that are not found:



Status: 404 Not Found Time: 98 ms Size: 709 B

```
{
    "title": "Task not found.",
    "message": "Task with id of 2 could not be found.",
    "stack": "Error: Task with id of 2 could not be found.\n    at taskNotFoundError (/Users/joannachen/Documents/aa-cd/Module-Express/
      readings/apis/reading-apis-demo-solution/routes/tasks.js:47:15)\n    at router.delete.asyncHandler (/Users/joannachen/Documents/
      aa-cd/Module-Express/readings/apis/reading-apis-demo-solution/routes/tasks.js:119:12)"
}
```

What you've learned

You now know how to define a collection of route endpoints to perform CRUD operations. You know how to return responses to the client with appropriate HTTP status codes. You have also used the built-in `express.json()` middleware function to parse request body content in JSON format.

In the project, you used Postbird to view your database while using Postman to test your API. Within your routes, you transformed data for responses and processed errors to return appropriate responses (status codes and content).

Now that you've learned how to create your own REST API, you can create a client-side project that utilizes your own API! As a reminder, you can use the Fetch API in your client-side project to easily fetch data to render in your views.

Cross-Origin Resource Sharing (CORS)

Now that you've built out an API, it's time to build out a client to interact with this API. You have a couple of choices here: you can either serve this client application out of the same server or you can build out a separate client application that's served from a different server.

Serving the client out of your API server reduces complexity and makes the development experience simpler. However, you might want to serve the client and the API out of different servers in order to reduce the overall load on one single server and to improve maintainability of your application as it grows over time.

In this lesson, you'll build out a client that's completely separate from the API server. Along the way, you'll learn about the Cross-Origin Resource Sharing (CORS) issues that need to be considered when accessing API resources from a different origin. By the end of this lesson, you will:

1. Understand that by default a web application running in the browser can only access API resources from the same origin (i.e. domain, protocol, port).
2. Understand that an API can be configured to use Cross-Origin Resource Sharing (CORS) to grant web applications from a different origin access to some or all of its resources.
3. Use the cors npm package to enable cross-origin resource sharing.

Why CORS

In the previous reading, you built out a RESTful API to perform CRUD operations on tasks. Let's now build out a client to interact with that API.

Demo project setup

To do this, begin by cloning the project skeleton for this frontend:

```
git clone https://github.com/appacademy-starters/express-apis-frontend.git
```

Much of this starter repo should be pretty familiar. There's an Express server at `index.js` that's currently handling GET requests to `'/'` by rendering the `index.pug` template.

The `index.pug` template extends `layout.pug`. In the head tag of `layout.pug`, there are two links: one that fetches the bootstrap stylesheets and one that imports a CSS file called `style.css`.

Here's what's going on: in `index.js`, the Express application is using a built-in middleware function called `express.static`. This middleware function allows the application to serve static assets, such as JavaScript, CSS, or image files, out of a specific directory.

In this repo, the server has been configured to serve the files that are in `public` as static assets: `app.use(express.static(path.join(__dirname, "public")));`.

To see what this means, go ahead and run `npm install` and then start up the server by running `npm start`. Then, go to `localhost:4000/style.css` to see the server serving up the `style.css` file. So in essence, the `link(rel="stylesheet" type="text/css" href="style.css")` tag that you see in `layout.pug` is effectively the same as `link(rel="stylesheet" type="text/css" href="http://localhost:4000/style.css")`.

This was a pretty high-level overview of the `express.static` middleware function, and you can read more about how this works in the Express documentation on serving static files.

Attempting to fetch all tasks

Now, if you look inside of `index.pug`, there's a script tag that's importing the `public/js/index.js` JavaScript file. Confirm that this importing is working by navigating to `localhost:4000/`. There's currently a `console.log` statement in `public/js/index.js`, so at this point, you should see a "Hello from index.js!" statement when you open the browser console.

Let's replace that `console.log` with some code to fetch all the tasks from the API that you built in the previous reading. Be sure that the API is up and running on `localhost:8080`. Then, let's use the Fetch API to make a GET request for all of the tasks.

A few weeks ago, you went over a quick overview of the Fetch API. As a reminder, the Fetch API is used to make HTTP requests. It uses Promises to handle the asynchronous nature of HTTP requests and responses.

You learned that the `fetch`'s first argument is the URL that you want to make a request to while the second argument is an optional options object.

Take a moment to review this example fetch from MDN. The comments denote the possible values for each key in the additional `options` object.

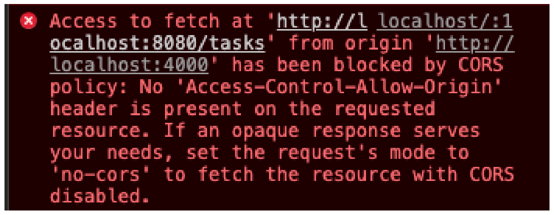
Out of the all of the `options`, you'll primarily be working with `method`, `headers`, and `body` today:

```
const response = await fetch(url, {
  method: "POST", // GET, POST, PUT, DELETE, etc.
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify(data), // body data type must match "Content-Type" header
});
```

Now that you've reviewed the Fetch API, let's use it in `public/js/index.js` to fetch all tasks from the API server.

```
// public/js/index.js

document.addEventListener("DOMContentLoaded", async () => {
  const res = await fetch("http://localhost:8080/tasks");
  const { tasks } = await res.json();
  console.log(tasks);
});
```

In the code snippet above, an event listener is set up for the `DOMContentLoaded` event, so when all of the HTML elements are loaded, it fires a fetch request to the API server. Looking in the console, we can see that this failed:



```
Access to fetch at 'http://l localhost/:1
ocalhost:8080/tasks' from origin 'http://
localhost:4000' has been blocked by CORS
policy: No 'Access-Control-Allow-Origin'
header is present on the requested
resource. If an opaque response serves
your needs, set the request's mode to
'no-cors' to fetch the resource with CORS
disabled.
```

CORS

The error message mentions that the request was blocked by "CORS policy". Cross-Origin Resource Sharing means being able to access resources that are located at an origin different than the origin of the application that's making the request. Simply put, in this case, the origin of this fetch request was made from `localhost:4000`, and the tasks resources are located at `localhost:8080`.

To clarify, being from the same origin means having the same protocol (http vs https), domain, and port number. In our current example, both the client and the API server are using the same protocol (http) and domain (localhost), but are served on different ports, which is why it's considered a cross-origin request.

By default, browsers prevent cross-origin requests from happening for security reasons. Although this might make life a little more difficult for developers, restricting this type of requests is a good thing that reduces the number of attack vectors against your applications.

This cross-origin restriction is enforced by browsers, so if you were to make a server-side request to the API, there's nothing preventing that by default.

Setting up CORS

Although Cross-Origin Resource Sharing is prevented by default, you can configure your API server to allow these types of cross-origin requests from trusted origins. Let's walk through an example of how to do this with your API.

To start, install the cors npm package by running `npm install cors`. Be sure that you're in the correct directory (the API server) when you're running this.

Setting up CORS to fetch all tasks

Now, go to `routes/tasks.js`. In there, import the `cors` middleware at the top of the file. Then, go down to `router.get("/")` and add the `cors()` middleware function. Here's what your `routes/tasks.js` file should look like:

```
const express = require("express");
const db = require("../db/models");
const { check, validationResult } = require("express-validator");
const cors = require("cors");

// CODE IN BETWEEN NOT SHOWN

router.get(
  "/",
  cors(),
  asyncHandler(async (req, res) => {
    const tasks = await Task.findAll();
    res.json({ tasks });
  })
);

// REST OF CODE NOT SHOWN
```

Now, if you go back to `localhost:4000/` and refresh, the console should no longer show the CORS error. Instead, it should be showing the list of tasks from your API. Yay!

Let's do more than just console logging the tasks. In your `views/index.pug` (in the frontend project), add a div with a class name of "tasks-container":

```
extends layout.pug

block content
  h1 Hello World!
  div.tasks-container
  script(src="js/index.js")
```

Then, in `public/js/index.js`, manipulate the DOM to add the tasks to the "tasks-container" div. Feel free to use some Bootstrap class names to style the task items:

```
document.addEventListener("DOMContentLoaded", async () => {
  const res = await fetch("http://localhost:8080/tasks");
  const { tasks } = await res.json();

  const tasksContainer = document.querySelector(".tasks-container");
  const tasksHtml = tasks.map(
    ({ name }) => `
      <div class="card">
        <div class="card-body">
          <p class="card-text">${name}</p>
        </div>
      </div>
    `
  );
  tasksContainer.innerHTML = tasksHtml.join("");
});
```

Next, let's set up a view for creating a task.

Attempt to create a task

First, in `views/index.pug`, add a link to the "/create" page. Also, go ahead and update the `h1` to say "Task List" instead:

```
extends layout.pug

block content
  h1 Task List
  div.tasks-container
  a(class="btn btn-primary" href="/create") Create Task
  script(src="js/index.js")
```

Next, set up a `views/create.pug` template that would allow someone to create a task:

```
extends layout.pug

block content
  .errors-container
  form(class="create-form")
    .form-group
      label(for='name') Task name
      input#name.form-control(type='text', name="name", placeholder='Task name')
    button.btn.btn-primary(type='submit') Create Task

  script(src="js/create.js")
```

Notice how in the template above, it imports a new script at `js/create.js`. Create a new `public/js/create.js` file. In there, set up the JavaScript that would make a POST request to `localhost:8080/tasks` when the "Create Task" form is submitted:

```
const form = document.querySelector(".create-form");

form.addEventListener("submit", async (e) => {
  e.preventDefault();
  const formData = new FormData(form);
  const name = formData.get("name");
  const body = { name };
  await fetch("http://localhost:8080/tasks", {
    method: "POST",
    body: JSON.stringify(body),
    headers: {
      "Content-Type": "application/json",
    },
  });

  window.location.href = "/";
});
```

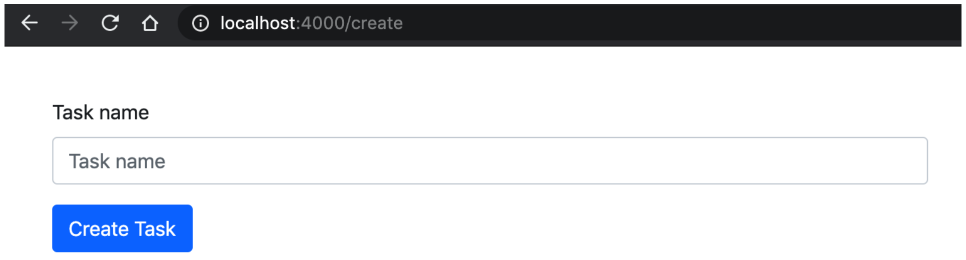
In the code above, when a task is successfully created, the use gets redirected back to the home page (`localhost:4000/`) where all of the tasks are displayed.

The user is redirected by setting `window.location.href` equal to `"/"`.

Then, in `index.js`, set up a route handler for `app.get('/create')`:

```
app.get("/create", (req, res) => {
  res.render("create");
});
```

Finally, when you click on the `Create Task` button on the home page, it should take you to a form that looks like this:



If we review back in the `public/js/index.js` file, when someone fills out the task name field and then submits this form, it should make a POST request to the API server to create a task.

> **Note:** The API requests that you've added so far do not implement proper error handling because it's more important to focus on the CORS-related code for now. In tomorrow's project, you'll get a more in-depth overview of how to properly handle fetch errors.

At this point, you might be thinking to yourself, "Wait a second, this is *obviously* going to fail because we haven't configured CORS for this specific route yet." And you're right! So let's go ahead and add the `cors()` middleware function to `routes/tasks.js` in the API:

```
// POST /tasks (Create)
router.post(
  "/",
  cors(),
  validateTask,
  handleValidationErrors,
  asyncHandler(async (req, res) => {
    const { name } = req.body;
    const task = await Task.create({ name });
    res.status(201).json({ task });
  })
);
```

Great, let's fill out a task name and then submit this form. And...it failed. It failed with the same CORS error that you previously saw. How frustrating!

So what happened there? Well it turns out that when it comes to CORS, there are actually two types of requests: simple requests and not-so-simple requests.

According to the MDN docs on CORS, in order to be considered a "simple" request, it must meet **all** of the following criteria:

1. It has to be one of these HTTP methods: `GET`, `HEAD`, or `POST`.
2. It can only have "CORS-safelisted request-headers". Essentially this means that if you manually set a header that's not included on the safelisted request headers list, then the request is no longer a "simple" request.
3. The `Content-Type` must be `application/x-www-form-urlencoded`, `multipart/form-data`, or `text/plain`.

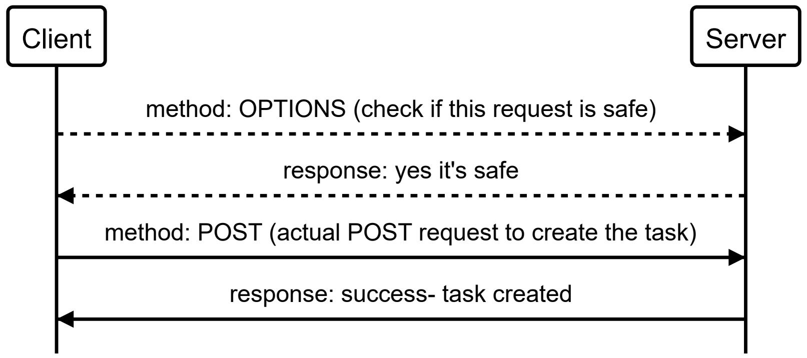
In the case of the task creation form, it met the first criteria since it was a POST request. It also met the second criteria since the only request header you set was `Content-Type`, which is on the safelisted request headers list. Unfortunately, because the `Content-Type` was `application/json`, it failed the third criteria.

Preflighted requests

What happens when the request is not considered to be a "simple" request?

When the request does not meet all three criteria, then the browser takes some extra precaution and actually makes a preflighted `OPTIONS` request to the server before making the actual request.

In other words, when it comes to a "not-so-simple" cross-origin request, there are actually two HTTP requests being made. First, a request is made with an `OPTIONS` method, and if the server responds successfully, then the actual request would go through:


Client
Server
method: OPTIONS (check if this request is safe)
response: yes it's safe
method: POST (actual POST request to create the task)
response: success- task created

The problem right now is that because CORS is only set up for the `POST /tasks` endpoint, when the first `OPTIONS` request is made, it fails the CORS check:

preflight failure

To resolve this, go back to `routes/tasks.js` and set up CORS for the `OPTIONS` route to "/tasks". As a reminder, the entire `tasks` router is nested under `"/tasks"`, so you don't need to define "/tasks" in the route handler path again:

```
// POST /tasks (Create)
router.options("/", cors());
router.post(
  "/",
  cors(),
  validateTask,
  handleValidationErrors,
  asyncHandler(async (req, res) => {
    const { name } = req.body;
    const task = await Task.create({ name });
    res.status(201).json({ task });
  })
);
```

Now, when you try to create the task, both routes are configured for CORS!

Setting up CORS for the entire API

You've now properly configured CORS for two of your API endpoints. Unfortunately, there are currently a couple of problems.

First, it would be pretty tedious to have to add the `cors()` middleware function individually to every single route. In addition, you would also have to think through whether or not the specific requests would also require the associated `options` route to be set up to handle CORS.

Second, right now those two endpoints are configured to allow requests from *any* origin. In terms of security, that's a bad idea. You only want to allow requests from origins that you know and trust.

Let's address both of these issues.

First, in your API's `app.js`, import the `cors` middleware function at the top and then configure for the entire application to use that middleware function:

```
const express = require("express");
const morgan = require("morgan");
const cors = require("cors");

const { environment } = require("./config");

const indexRouter = require("./routes/index");
const tasksRouter = require("./routes/tasks");

const app = express();

app.use(morgan("dev"));
app.use(express.json());
app.use(cors());

// REST OF CODE NOT SHOWN
```

Then, configure the `cors` middleware function to only allow requests from trusted origins. In this case, the client requests are coming from `http://localhost:4000`:

```
app.use(cors({ origin: "http://localhost:4000" }));
```

Check out the cors npm package documentation for other configuration options you can pass into the `cors` middleware function.

Finally, since CORS is configured for the entire app now for any requests that come from `http://localhost:4000`, remove the `cors()` middleware from the routes that you previously added them to in `routes/tasks.js`.

You can also remove `router.options("/", cors());` since the app-wide CORS configuration will also work for `OPTIONS` requests.

After you've removed all usage of `cors()` from `routes/tasks.js`, go ahead and remove the import of `cors` from the top of that file.

What you've learned

In this reading, you learned:

1. That by default, a web application running in the browser can only access API resources from the same origin (i.e. domain, protocol, port).
2. That an API can be configured to use Cross-Origin Resource Sharing (CORS) to grant web applications from a different origin access to some or all of its resources.
3. How to use the cors npm package to enable cross-origin resource sharing.

Now that you've built out an API, it's time to build out a client to interact with this API. You have a couple of choices here: you can either serve this client application out of the same server or you can build out a separate client application that's served from a different server.

Serving the client out of your API server reduces complexity and makes the development experience simpler. However, you might want to serve the client and the API out of different servers in order to reduce the overall load on one single server and to improve maintainability of your application as it grows over time.

In this lesson, you'll build out a client that's completely separate from the API server. Along the way, you'll learn about the Cross-Origin Resource Sharing (CORS) issues that need to be considered when accessing API resources from a different origin. By the end of this lesson, you will:

1. Understand that by default a web application running in the browser can only access API resources from the same origin (i.e. domain, protocol, port).
2. Understand that an API can be configured to use Cross-Origin Resource Sharing (CORS) to grant web applications from a different origin access to some or all of its resources.
3. Use the cors npm package to enable cross-origin resource sharing.

Twitter Lite Project

Today you'll begin building a lightweight Twitter API using Express, Sequelize, and Postbird! There will be two parts to this project: creating an API and creating a client-side application.

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

1. A default "/" GET route.
2. A "/tweets" GET route to fetch an index of tweets.
3. A "/tweets" POST route to create tweets.
4. A "/tweets/:id" PUT route to update tweets.
5. A "/tweets/:id" DELETE route to delete tweets.
6. A backend API that connects to a simple frontend server.
7. A Users model.

Phase 0: Initialize project

Begin by cloning the project skeleton:

```
git clone https://github.com/appacademy-starters/express-sequelize-starter.git
```

Install your packages with `npm install`.

Initializing Sequelize

Just like in the To-Do list demo project, the Sequelize CLI is already configured to know where your database configuration is located and where to generate the `models`, `seeders`, and `migrations` folders. The project skeleton has already taken care of configuration by including complete `.sequelizerc`, `./config/index.js`, and `./config/database.js` files.

Configuring your environment variables

Before you set up your database, remember that the Sequelize CLI still needs access to your database credentials. Create an `.env` file based off of the `.env.example` file included in your project skeleton.

Creating your database

Before initializing the `Tweet` model, you'll need to create a new database.

Take a moment to create a database user and database:

- The login name that you must use is "twitter_lite_app" (make sure to set a login password).
- Your user must be granted the CREATEDB privilege so that you can run `npx dotenv sequelize-cli db:create`.
- The database name that you must use is "twitter_lite".

Phase 1: Set up the Tweet model

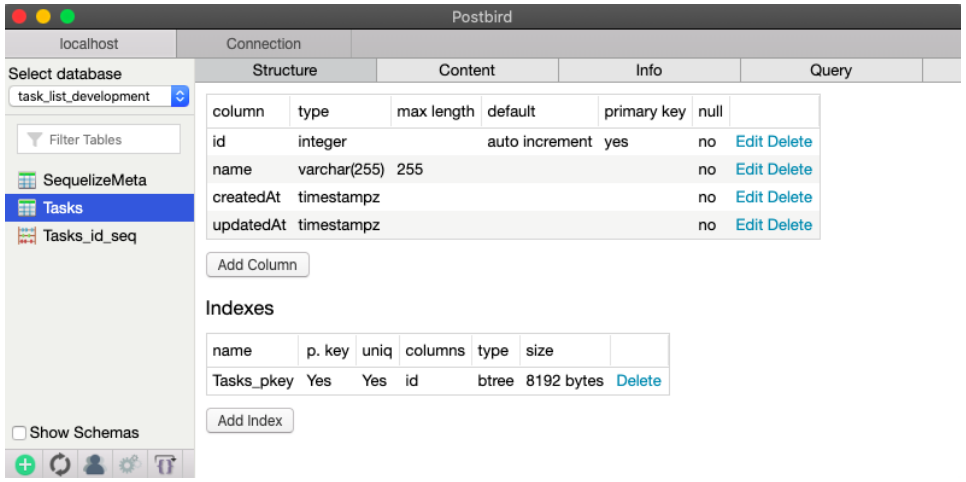
It's time to generate and migrate your Tweet model with the following commands:

```
npx sequelize model:generate --name Tweet --attributes "message:string"
```

In the model and migration files, make sure to configure the tweet `message` to not be nullable. Each tweet should have a `message` column of type `string` that has a maximum length of `280` and is not nullable. Then run your migrations:

```
npx dotenv sequelize db:migrate
```

After migrating, you should see the `Tweets` table in your Postbird client:



column	type	max length	default	primary key	null	
id	integer		auto increment	yes	no	Edit Delete
name	varchar(255)	255			no	Edit Delete
createdAt	timestampz				no	Edit Delete
updatedAt	timestampz				no	Edit Delete

Indexes

name	p. key	uniq	columns	type	size	
Tasks_pkey	Yes	Yes	id	btree	8192 bytes	Delete

Now generate a seed file by running the following command from the root of your project:

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

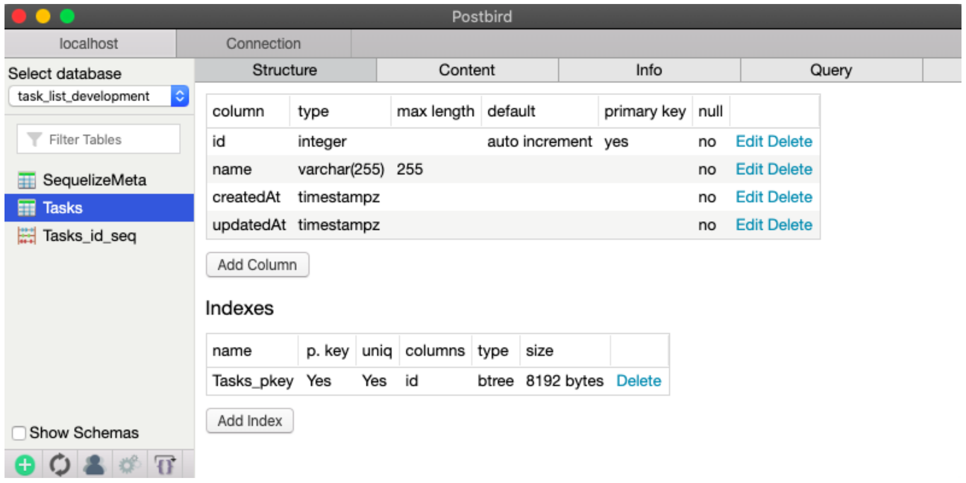
Go ahead and replace the contents of the `./db/seeders/[timestamp]-test-data.js` with the following code:

```
"use strict";

module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.bulkInsert(
      "Tweets",
      [
        {
          message: "The Martian was awesome!",
          createdAt: new Date(),
          updatedAt: new Date(),
        },
        {
          message: "Has anyone seen Ready Player One?",
          createdAt: new Date(),
          updatedAt: new Date(),
        },
        {
          message:
            "Harry Potter and the Sorcerer's Stone is the best out of all seven H
          createdAt: new Date(),
          updatedAt: new Date(),
        },
      ],
      {}
    );
  },

  down: (queryInterface, Sequelize) => {
    return queryInterface.bulkDelete("Tweets", null, {});
  },
};
```

Now run your seed files with the command below:

```
npx dotenv sequelize db:seed:all
```

After seeding, you should see the tweets you have created in your Postbird client:



column	type	max length	default	primary key	null	
id	integer		auto increment	yes	no	Edit Delete
name	varchar(255)	255			no	Edit Delete
createdAt	timestampz				no	Edit Delete
updatedAt	timestampz				no	Edit Delete

Add Column

Indexes

name	p. key	uniq	columns	type	size	
Tasks_pkey	Yes	Yes	id	btree	8192 bytes	Delete

Add Index

Phase 2: Set up test routes

Notice that your skeleton already has a basic Express application set up in `app.js`. It's time for you to set up the routers for your project.

Create a `routes` module by creating a `routes` directory in the root of your project. Within your `routes` folder, create an `index.js` file for your default router and a `tweets.js` file for your tweet routers. In both files, begin by requiring `express` and creating a `router` with `express.Router()`. Lastly, make sure you export the routers you just created.

A majority of RESTful APIs serve data in a JSON format. Let's do the same for this project! To do this, update your `app.js` file to have your application `use(express.json())`. The `express.json()` middleware is needed to parse request body content formatted in JSON so that it is available via the `req.body` property.

Lastly, in order to connect to the routes modules you have just created, import your `./routes/index` file as the `indexRouter` and import your `./routes/tweets` file as the `tweetsRouter`. Make sure your application is using the `/` route with the `indexRouter` as well as the `/tweets` route with the `tweetsRouter`.

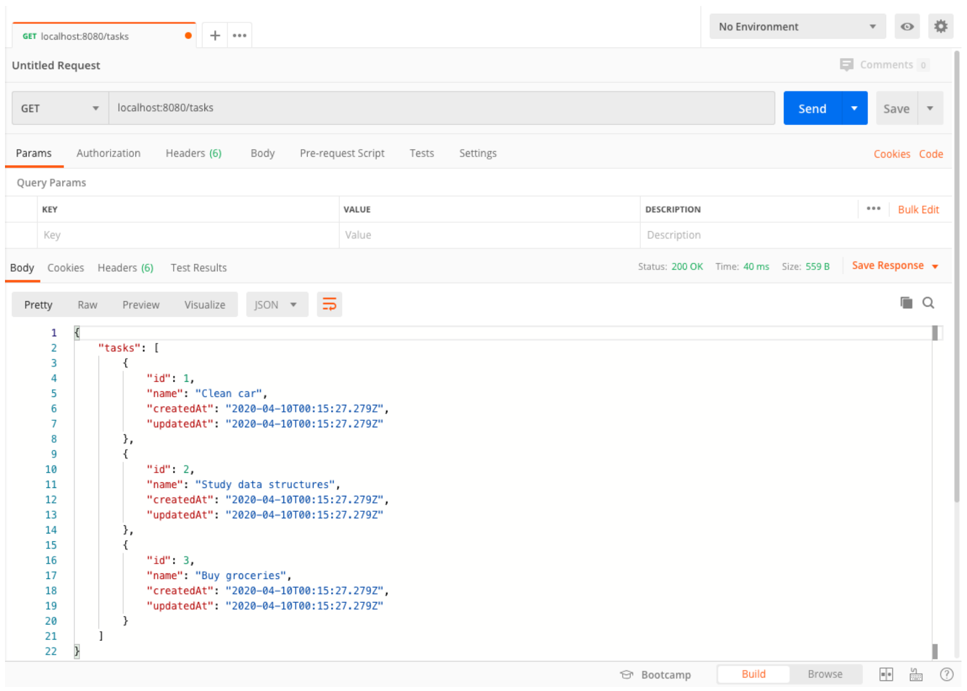
Creating test routes

Let's get started in the `./routes/index.js` file. Move your GET route for `/` from `app.js` into your `./routes/index.js` file. Update the route to use `res.json()` in order to render a JSON response of `message: "test index root"`.

Now you'll add a test route in your `./routes/tweets.js`. Create a route that handles a GET request to the `/` path. Have your result `send` a JSON response of `message: "test tweets index"`.

Testing requests on Postman

Now run `npm start` to start your server and open up Postman to test your GET requests.



```
{
    "tasks": [
        {
            "id": 1,
            "name": "Clean car",
            "createdAt": "2020-04-10T00:15:27.279Z",
            "updatedAt": "2020-04-10T00:15:27.279Z"
        },
        {
            "id": 2,
            "name": "Study data structures",
            "createdAt": "2020-04-10T00:15:27.279Z",
            "updatedAt": "2020-04-10T00:15:27.279Z"
        },
        {
            "id": 3,
            "name": "Buy groceries",
            "createdAt": "2020-04-10T00:15:27.279Z",
            "updatedAt": "2020-04-10T00:15:27.279Z"
        }
    ]
}
```

When sending a GET request to `localhost:8080/`, you should see JSON with your "test index root" message in the body response. When sending a GET request to `localhost:8080/tweets`, you should see JSON with your "test tweets index" message in the body response.

Note that if you use your browser to navigate to `localhost:8080/` and `localhost:8080/tweets`, you'll see the same response as in Postman.

Phase 3: Set up tweet routes

Now you'll add your Tweet routes in your `./routes/tweets.js`. Begin by requiring your `db` from your `../db/models` directory:

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

Now destructure your `Tweet` model from the `db` you have just imported:

```
const { Tweet } = db;
```

In this phase, you'll interact with your `Tweet` model by using built-in methods to set up the basic CRUD functionalities for tweets:

- `Tweet.findAll()` to fetch all of your database tweets. (Read)
- `Tweet.findByPk()` to fetch a database tweet based on the `id` from your request parameters. (Read)
- `Tweet.create()` to generate a tweet in your database. (Create)
- `Tweet.update()` to update a tweet in your database. (Update)
- `Tweet.destroy()` to delete a tweet from your database. (Delete)

GET /tweets

It's time to set up a GET `/` route to fetch all of your seeded tweets when sending a GET request to `localhost:8080/tweets`. Since you'll be awaiting a database fetch, let's bring back your `asyncHandler` function to help you catch errors in a DRY way!

As a reminder, your `asyncHandler` takes in a `handler` function to return a middleware function that invokes the handler function with `req`, `res`, and `next`. It then chains on a `catch` statement by passing in the `next` function.

Let's begin by updating the `tweets` GET route to `/`. Wrap the route function with your `asyncHandler` to be able to `await` the database fetch of all your tweets. Look at the beginning of the Phase 3 instructions to determine which Tweet model method to use.

Lastly, remember to render the tweets you have fetched from your database in JSON by using `res.json({ tweets })`.

GET /tweets/:id

Now you'll set up the GET `/:id(\\d+)` route to read a specific tweet when sending a GET request to `localhost:8080/tweets/:id`. Parse the `tweetId` from your `req.params` object and use your `tweetId` to fetch a specific tweet from the database.

Note that the `app` module contains the following middleware function to catch unhandled requests and pass a `404` error to the global error handler:

```
app.use((req, res, next) => {
  const err = new Error("The requested resource couldn't be found.");
  err.status = 404;
  next(err);
});
```

This ensures that if a client makes a request to a route that doesn't exist they'll receive an appropriate error message.

Any error that occurs in your routes will be handled by the below global error handler so that error messages can be formatted and returned to the client in a consistent way. The error's title, message, and stacktrace is rendered in JSON. If your application is in production, the error will be rendered without the error stacktrace.

```
app.use((err, req, res, next) => {
  res.status(err.status || 500);
  const isProduction = environment === "production";
  res.json({
    title: err.title || "Server Error",
    message: err.message,
    stack: isProduction ? null : err.stack,
  });
});
```

Now let's go back to your GET route for a single tweet. Now that you have written a database fetch for a tweet, you want to render errors for tweets that were not found. Make sure your asynchronous route function is taking in `next` as a parameter. If you have fetched a valid tweet, render the tweet in JSON. If you have not fetched a valid tweet, you can use a function to generate an error before invoking the `next` method.

Let's define your error generator function for tweets not found. In your tweet route module, define a `tweetNotFoundError` function that takes in a tweet ID. Generate a new `Error` object with a message stating that a tweet of the given ID could not be found. Assign the error to have a `title` property to be "Tweet not found." and a `status` of `404`. At the end of the function, return the error.

Now that the `tweetNotFoundError` function is written to generate and return an `Error` object, you can pass the return value of the `tweetNotFoundError` function call into the `next` method to invoke the global error handler.

Take a moment to test your route and error handling in Postman.

POST /tweets

Set up a POST `/` route to create a new tweet by sending a POST request to `localhost:8080/tweets`. Now that you will take in JSON data to handle a request, remember that your application is using the `express.json()` middleware in `app.js` to parse the body content's JSON and access the `req.body`.

Now that you are taking in data, you'll need to validate that data. Import `check` and `validationResult` from `express-validator`. Use the `express-validator` library to check that the `message` value is present, and that it's not over 280 characters long.

You'll also need to handle your validation errors. In previous projects, you've been handling validations in each of the route handlers. Today, let's DRY up our code and set up one middleware function that can check for errors in the `req` object, and if there are errors, then we can generate an `Error` from that middleware function and handle it there.

Define a `handleValidationErrors` function and have it take in `req`, `res`, and `next` as parameters. Begin by generating a `validationErrors` object by invoking the `validationResult` function with the request:

```
const handleValidationErrors = (req, res, next) => {
  const validationErrors = validationResult(req);
  // TODO: Generate error object and invoke next middleware function
};
```

If you do you have validation errors, use `array()` to transform your `validationErrors` object into a mappable array. Pluck out each error's `msg` attribute to generate an `errors` array of error messages:

```
const errors = validationErrors.array().map((error) => error.msg);
```

After generating the `errors` array, you'll want to create a new `Error` object with a 400 `status` and title of "Bad request":

```
const err = Error("Bad request.");
err.errors = errors;
err.status = 400;
err.title = "Bad request.";
next(err);
```

You'll also want the `Error` object to set an `errors` array as a `errors` property. Invoke the `next` middleware function with the `Error` object you have created. If you do not have any validation errors, invoke the `next` middleware function without an argument.

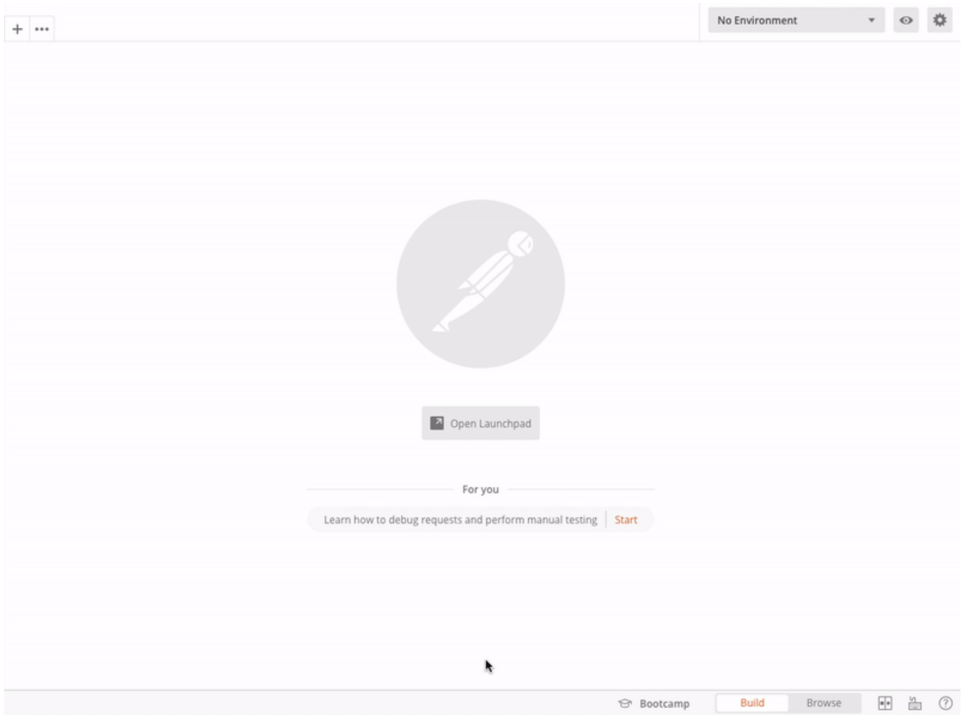
Your `handleValidationErrors` function should look something like this:

```
const handleValidationErrors = (req, res, next) => {
  const validationErrors = validationResult(req);

  if (!validationErrors.isEmpty()) {
    const errors = validationErrors.array().map((error) => error.msg);

    const err = Error("Bad request.");
    err.errors = errors;
    err.status = 400;
    err.title = "Bad request.";
    return next(err);
  }
  next();
};
```

Now that you are adding the `.errors` property to this error you will need to modify the generic error handler in `app.js` to add this `.errors` array to the JSON you return, otherwise when you call our API and get validation errors you won't be able to see them!

```
app.use((err, req, res, next) => {
  res.status(err.status || 500);
  const isProduction = environment === "production";
  res.json({
    title: err.title || "Server Error",
    message: err.message,
    errors: err.errors, // Includes our array of validation errors in our JSON re
    stack: isProduction ? null : err.stack,
  });
});
```

Now make sure the POST route is using your tweet validations and `handleValidationErrors` function as middleware. Test your new route in Postman. Don't forget to set the "Content-Type" header to "application/json" and send raw JSON data through the request body. Feel free to review how to send a POST request through the image below.



Take a moment to also test your error handling. You should see an error response in JSON upon submitting bad data (i.e. an empty `message` field). Notice that your error response has the following properties: a `status` of "400", a `title` of "Bad request.", and an array of `errors`. Make sure that you see a `400 Bad Request` error if there are failing data validations.

PUT /tweets/:id

Set up a PUT `/:id(\\d+)` route to update a tweet. Begin by parsing the tweet id within your `req.params` object and using the parsed `tweetId` to fetch the tweet to update. If you have a valid tweet, await the update and then render the updated tweet in JSON format.

If you have not fetched a valid tweet, use the same `404` error handling that the `GET /tweets/:id` route used. Invoke your `tweetNotFoundError` function to generate a `404` error. Then pass the return value of the `tweetNotFoundError` function call into the `next` method.

Make sure to also use the same `400` error handling that the `POST /tweets` route used. Validate your data and handle your validation errors, like in the POST route, to generate an error response upon receiving bad form data.

Test your PUT route:

1. Use Postman to send a GET request to `localhost:8080/tweets/1` to view the data of your first database tweet.
2. Configure your "Content-Type" header as "application/json".
3. Send a PUT request to `localhost:8080/tweets/1` with updated fields in the `raw` request body.
4. View your updated tweet in Postbird.
5. Send a PUT request to `localhost:8080/tweets/1` with invalid data to check the error handling.

DELETE /tweets/:id

Set up the DELETE `/:id(\\d+)` route for deleting a tweet by sending a DELETE request to `localhost:8080/tweets/:id`. Begin by parsing the tweet ID and using the id to find your tweet to delete. If a valid tweet is found, await to destroy the tweet and render a `204` response to confirm the deletion by using `res.status(204).end()`.

If a valid tweet wasn't found, use the same `404` error handling that the GET and PUT routes used. Pass the tweet ID into your `tweetNotFoundError` function call to generate a `404` error. Then pass your newly generated `404` error into the `next` method to invoke the global error handler.

Lastly, send a request with Postman to test your DELETE route and delete a tweet. Verify that the tweet was properly deleted by viewing your seeded objects with Postbird.

Phase 4: Render tweets in the frontend application

Up to this point, you've set up a CRUD API for tweets. All users of your API have access to the following features:

- Viewing all existing tweets
- Viewing a tweet specific tweet
- Creating a tweet
- Updating a tweet
- Deleting a tweet

Now that you have a fully functioning API, you can create another Express application to serve the pages to render the client-side code for the Twitter Lite user interface. To review, the term "client-side" means that user triggered events (i.e. form submissions), the API request/response cycle, and rendering data is handled by JavaScript code running in the browser.

This new Express application will use the Pug template engine to render HTML pages on the server-side or backend. Regardless, we'll refer to this Express application as the "frontend" of the Twitter Lite project since the backend rendered pages are primarily used to deliver the client-side JavaScript to the browser.

Here are all the things that you still need to do to make your project a lite version of Twitter:

- Add an Express "frontend" application to deliver the client-side code to interact with the API;

- Set up users and auth in the API; and
- Set up users and auth in the frontend application.

To add the frontend application, clone the repo below into a new folder that's a sibling to the API project folder.

```
git clone https://github.com/appacademy-starters/express-apis-frontend.git
```

Move into your frontend `express-apis-frontend` directory and run `npm install`. Now you'll want to start up the frontend server and the backend server in different terminal tabs or windows. Start your backend server from the first terminal window by running the `npm start` command from the root of the API project. Then open a second terminal to start your frontend server by running `npm start` from within the `express-apis-frontend` folder.

Take a moment to navigate to your backend at `http://localhost:8080/`. You should see a JSON response of a "test index root" message. Now navigate to your frontend at `http://localhost:4000/`. Open up your developer tools and you should see an `h1` element with the content of "Hello World!".

Open your `public/js/index.js` file and add an event listener script. This file is where you will fetch all tweets from `http://localhost:8080/tweets`. Start off by awaiting the fetch response (`res`) of all your API tweets. Parse your response into JSON and destructure the `tweets` property from your parsed response object. You'll eventually use these `tweets` to render a list of each tweet message, but just console log all the tweets for now. Make sure to catch and `console.error` any errors. Your script function should look something like this:

```
document.addEventListener("DOMContentLoaded", async () => {
  try {
    const res = await fetch("http://localhost:8080/tweets");
    const { tweets } = await res.json();
    console.log(tweets);
  } catch (e) {
    console.error(e);
  }
});
```

As a reminder, your `index.js` script will run as soon as the DOM content in your browser loads for your frontend. Your API fetch call will fire after the DOM content has loaded and the `tweets` from your response will be logged in your developer tools console.

If you navigate to `localhost:4000/` to view your frontend application, you should see a `Failed to fetch` error message. Let's investigate! Instead of console logging your `tweets`, try console logging the response (`res`) to figure out more context around your error. If you look closer, your response has `type` of "cors" and that you have a success status code of `200 OK`. This indicates that your application has a CORS error!

Resolve this CORS issue back on the API side by installing the `cors` npm package. In `app.js`, require and configure the `cors` middleware function to allow requests from the origin `localhost:4000`:

```
app.use(cors({ origin: "http://localhost:4000" }));
```

Go back to the frontend application and try to console log the tweets again. This time, all the existing tweets should console log! Now let's actually display the tweets in HTML elements by updating the "DOMContentLoaded" event listener in `public/js/index.js`.

After your fetch call in the `try` block, redirect your user to the `/log-in` page and `return` out of the event listener function if your `res` has a status of `401`. If the response does not have a status of `401`, parse your response as JSON and destructure the `tweets` property from your JSON response.

Take a moment to create a `div` element in your `index.pug` template before the load of your `index.js` script. Set the `div` with an ID of "tweets-container". In your event listener, use the Vanilla JavaScript `querySelector()` method to find the `tweetsContainer`. Declare a `tweetsHtml` variable for an array of stringified HTML blocks that will render tweet messages. Map over your array of `tweets` to generate the array of stringified HTML blocks.

You can use Bootstrap classes to style the tweets. For each message, render a `div.card` parent element with a `div.card-body` child element. Within the `div.card-body` element should live a `p.card-text` element that renders each tweet's `message` with interpolation. You'll want to join your `tweetsHtml` array by an empty string (`""`) to create a full stringified HTML block for all your tweet message blocks. Set the `innerHTML` property of the `tweetsContainer` to be the joined `tweetsHtml`.

Lastly, make sure that you are using `console.error` to log any caught errors. Your script should look something like this:

```
document.addEventListener("DOMContentLoaded", async () => {
  try {
    const res = await fetch("http://localhost:8080/tweets");
    const { tweets } = await res.json();

    const tweetsContainer = document.querySelector("#tweets-container");
    const tweetsHtml = tweets.map(
      ({ message }) => `
      <div class="card">
        <div class="card-body">
          <p class="card-text">${message}</p>
        </div>
      </div>
    `
    );
    tweetsContainer.innerHTML = tweetsHtml.join("");
  } catch (e) {
    console.error(e);
  }
});
```

Take a moment to confirm that your frontend is fetching and rendering the tweets on `localhost:4000/`.

Phase 5: Add the users model

Now it's time to set up the users model to begin implementing user authentication! Use the following Sequelize command to generate a User model with `username`, `email`, and `hashedPassword` attributes.

```
npx sequelize model:generate --name User --attributes 'username:string,email:stri
```

Make sure to update the user model and migration files with the following constraints:

- `username` is unique and not nullable

- `email` is unique and not nullable
- `hashedPassword` is the type `STRING.BINARY` and not nullable.

Now run the migration with `npx dotenv sequelize db:migrate` and check in Postbird to confirm the creation of your User table.

Twitter Lite Project... with Authentication!

At this point, you've set up an API for tweets and connected a frontend application to interact with your API. You've also added a Users model to your project to prepare for implementing user authentication.

In this project, you'll be writing Vanilla JS to render frontend elements. You have been using string template literals to render data from the database as HTML. Later on in the class, you'll use React to render components and build frontend clients. In the next portion of the project, you will set up the Users router and protect your tweet resources from unauthenticated users.

When you have completed the second part of the project, your application should have the following features:

1. Users routes
2. Protected tweet resources
3. User registration
4. User log-in
5. Authenticated tweet creation
6. A form to create tweets

Phase 1: Users router

Begin by creating a users routes module at `routes/users.js`. Require `express` to initialize a router with `express.Router()` and make sure to export the router you just initialized.

In your `app.js` file, require the `usersRouter` you have just created and have your application connect the `/users` path to the `usersRouter`.

The users router will need to use a couple of helper functions that currently exist in the tweets router: `asyncHandler` and `handleValidationErrors`. Go ahead and create a `utils.js` file within the root of your project and then move those two functions into that file.

Now add a route in the user router to handle POST requests to `localhost:8080/users`. In this route, use `express-validator` to validate the params. Make sure to import `check` from `express-validator` as well as the `asyncHandler` and `handleValidationErrors` helper functions from the `../utils.js` file:

```
const { check } = require("express-validator");
const { asyncHandler, handleValidationErrors } = require("../utils");
```

In your POST route, begin by wrapping the route handler with the `asyncHandler` function because there will be some asynchronous logic inside the route.

Add the following validations to validate your user authentication data so that you can render error messages upon submission of bad registration data. Go ahead and also use the `handleValidationErrors` helper middleware function.

```
const validateUsername =
  check("username")
    .exists({ checkFalsy: true })
    .withMessage("Please provide a username");

const validateEmailAndPassword = [
  check("email")
    .exists({ checkFalsy: true })
    .isEmail()
    .withMessage("Please provide a valid email."),
  check("password")
    .exists({ checkFalsy: true })
    .withMessage("Please provide a password."),
];

router.post(
  "/",
  validateUsername,
  validateEmailAndPassword,
  handleValidationErrors,
  asyncHandler(async (req, res) => {
    // TODO: User creation logic
  })
);
```

Within the asynchronous route handler function, destructure the `username`, `email`, and `password` from the `body` of your request. You'll use these properties to create a new user in your database. For now, leave a `TODO` note to create the user.

Now to actually create the user, install and require the `bcryptjs` library to use the `bcrypt.hash()` method. Invoke the `bcrypt.hash()` method with the user's `password` and a salt round of `10`. Await the generation of a `hashedPassword` before creating the user.

At this point, make sure you have required the `db` from `../db/models`. Destructure your `User` model from your `db` module and await the creation of your user (`User.create()`). In order to protect the user's credentials, create the user with the `hashedPassword` instead of the `password` from the form request body.

At this point, your user creation route should look something like this:

```
router.post(
  "/",
  validateUsername,
  validateEmailAndPassword,
  handleValidationErrors,
  asyncHandler(async (req, res) => {
    const { username, email, password } = req.body;
    const hashedPassword = await bcrypt.hash(password, 10);
    const user = await User.create({ username, email, hashedPassword });
    // TODO: Generate JSON Web Token (access token)
    // TODO: Render user in JSON
  })
);
```

Once the user has been created, you want to return an access token. You'll be using a JWT as the access token. To generate and decode the JWT, use the jsonwebtoken npm package. Begin by running `npm install jsonwebtoken`.

There are a few configuration steps to set up JWT generation. You need to set a *secret key* and a number representing how many seconds before the token expires. Add a `JWT_SECRET` and `JWT_EXPIRES_IN` variable to your `.env` file.

To generate a secret, open up your node repl in your terminal by running `node` and then use the built-in crypto module to generate your `JWT_SECRET` key:

```
require("crypto").randomBytes(32).toString("hex");
```

For the `JWT_EXPIRES_IN` set it to `604800`, which is the number of seconds for one week. Then update your `config/index.js` file to use these environment variables:

```
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,
  },
  jwtConfig: {
    secret: process.env.JWT_SECRET,
    expiresIn: process.env.JWT_EXPIRES_IN,
  },
};
```

Finally, let's put this JWT generation logic inside of an `auth.js` file. Create an `auth.js` file in the root of your project and require the `jsonwebtoken` package as well as your `jwtConfig` variables from your `./config` module.

Now define a function called `getUserToken`. This function will take a user object as a parameter and then create a payload from the user. Then, it uses the `jsonwebtoken.sign()` function to generate a token.

```
const jwt = require("jsonwebtoken");
const { jwtConfig } = require("./config");

const { secret, expiresIn } = jwtConfig;

const getUserToken = (user) => {
  // Don't store the user's hashed password
  // in the token data.
  const userDataForToken = {
    id: user.id,
    email: user.email,
  };

  // Create the token.
  const token = jwt.sign(
    { data: userDataForToken },
    secret,
    { expiresIn: parseInt(expiresIn, 10) } // 604,800 seconds = 1 week
  );

  return token;
};

module.exports = { getUserToken };
```

Notice how in the code above, we had to convert `expiresIn` from a string to an integer because environment variables declared in the `.env` file are strings, and according to the jsonwebtoken docs, anything that gets set to the `expiresIn` as an **integer** will be calculated as **seconds**, and **strings** will be calculated in **milliseconds**.

Now, go back to the users routes module and import the `getUserToken` function from the `../auth` module. Then update the user creation route to call the `getUserToken` function and return a token to the user when they sign up for an account. Your user creation route should look something like this:

```
router.post(
  "/",
  validateUsername,
  validateEmailAndPassword,
  handleValidationErrors,
  asyncHandler(async (req, res) => {
    const { username, email, password } = req.body;
    const hashedPassword = await bcrypt.hash(password, 10);
    const user = await User.create({ username, email, hashedPassword });

    const token = getUserToken(user);
    res.status(201).json({
      user: { id: user.id },
      token,
    });
  })
);
```

Once you've confirmed that this is working correctly on Postman, let's put this JWT token to use by protecting the tweets resources and only allowing signed in users to access the tweets routes!

Phase 2: Protecting the tweets resources

It's time to protect the tweet resources in your API! On each request to tweet routes, do the following:

1. Parse out the JWT token from the request header.
2. Decode the JWT.
3. Find the user based on the JWT payload.

If all of the steps above succeed, then the user is considered to be signed in and can access the tweet resources.

To carry out the plan above, go to back to your `auth.js` file. Here, you'll want to use two middleware functions. The first will parse out the JWT token from the request header. The second function decodes the JWT to find the user and stores the user in the `req` object.

For the first middleware function, let's use the `express-bearer-token` middleware to do this parsing. Run `npm install express-bearer-token` to add the middleware to your application. Begin by requiring the middleware as `bearerToken` as well as requiring your `User` model from the database models.

Define a middleware function named `restoreUser` that takes in the parameters: `req` , `res` , and `next` . Begin by destructuring the JWT `token` from the `req` object. If the token is invalid, return an invocation of the `next` function.

If the token is valid, invoke the `jwt.verify()` method with four arguments:

- JWT `token` from the request body;
- JWT `secret` from your configuration file;
- `null` options; and
- An asynchronous function that takes in an error and a `jwtPayload` (more steps below).

Currently, your `restoreUser` function should look something like this. Notice if we don't get a token in the request at all we set the "WWW-Authenticate" header to "Bearer" and return a "401 Unauthorized" status code.

```
const restoreUser = (req, res, next) => {
  const { token } = req;

  if (!token) {
    return res.set("WWW-Authenticate", "Bearer").status(401).end();
  }

  return jwt.verify(token, secret, null, async (err, jwtPayload) => {
    // TODO: Define asynchronous function for jwtPayload logic
  });
};
```

In the asynchronous function, check whether there is an error object. If so, set the error's status to be `401` and return an invocation of the `next` function passing in the error object:

```
if (err) {
  err.status = 401;
  return next(err);
}
```

If there is no error, extract the user's `id` from the `data` property of your `jwtPayload`:

```
const { id } = jwtPayload.data;
```

Then use a `try` / `catch` block to await the fetch of a user (by using `User.findByPk()`) or catch an error to return an invocation of the `next` function:

```
try {
  req.user = await User.findByPk(id);
} catch (e) {
  return next(e);
}
```

Now check to see whether the `req` object has an associated `user` property. If there isn't a valid user, return a response that sets the "WWW-Authenticate" header with a value of "Bearer" and sends a "401 Unauthorized" status code before invoking the `next` function:

```
if (!req.user) {
  return res.set("WWW-Authenticate", "Bearer").status(401).end();
}
```

Your complete `restoreUser` function should look something like this:

```
const restoreUser = (req, res, next) => {
  const { token } = req;

  if (!token) {
    return res.set("WWW-Authenticate", "Bearer").status(401).end();
  }

  return jwt.verify(token, secret, null, async (err, jwtPayload) => {
    if (err) {
      err.status = 401;
      return next(err);
    }

    const { id } = jwtPayload.data;

    try {
      req.user = await User.findByPk(id);
    } catch (e) {
      return next(e);
    }

    if (!req.user) {
      return res.set("WWW-Authenticate", "Bearer").status(401).end();
    }

    return next();
  });
};
```

Now, take a moment to organize your `bearerToken` generator function and the `restoreUser` function into an array named `requireAuth`. Update your module exports to export the two functions as one `requireAuth` array:

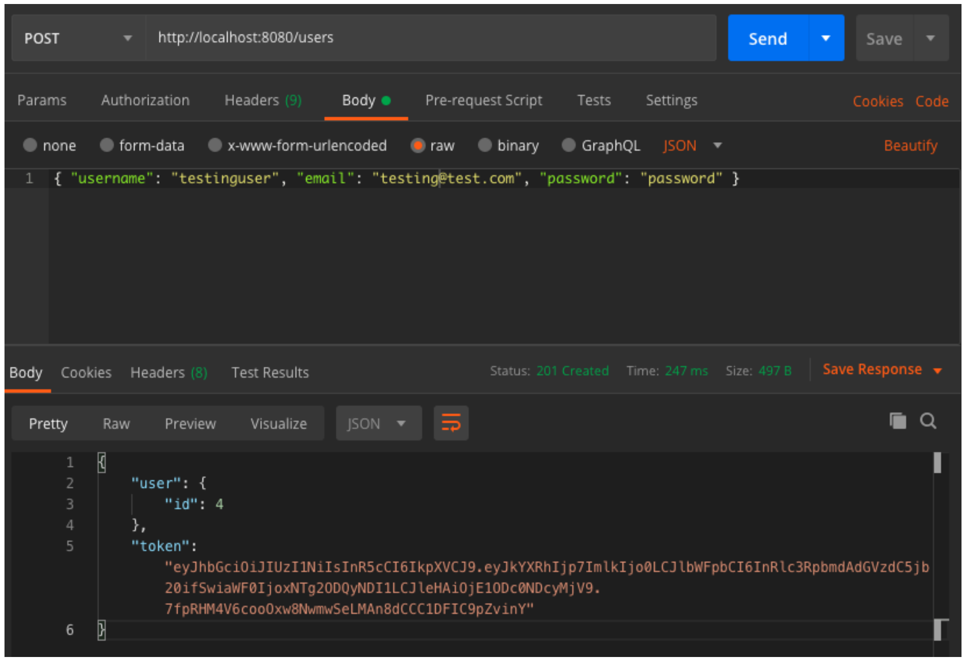
```
const requireAuth = [bearerToken(), restoreUser];
module.exports = { getUserToken, requireAuth };
```

Now it's time to add the `requireAuth` middleware to the tweets router! Add the following code to your `routes/tweets.js` file to use the user authentication functions you just created:

```
const { requireAuth } = require("../auth");

// REST OF FILE NOT SHOWN

router.use(requireAuth);
```

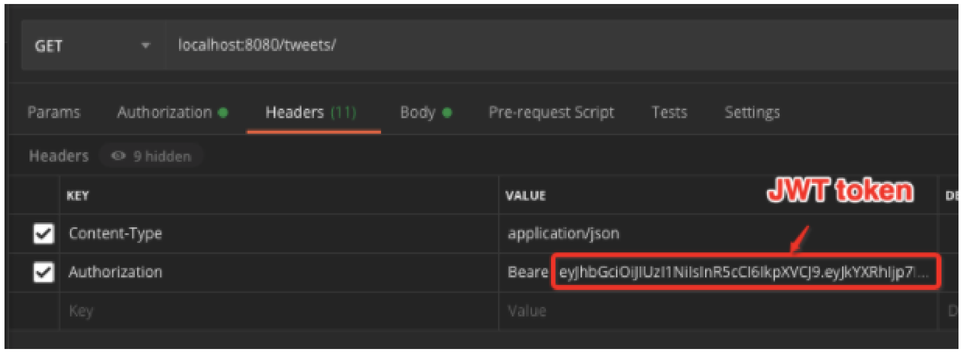
Verify that your tweet protection works by trying to make a GET request to `localhost:8080/tweets` from Postman. You should see a `401` response.

To fix this, create another user by posting to `localhost:8080/users`. This time, when the token comes back in the response, copy that response and paste it in to the *Authorization header*:



```
{ "username": "testinguser", "email": "testing@test.com", "password": "password" }
```

```
{
    "user": {
        "id": 4
    },
    "token":
        "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjp7ImlkIjo0LCJlbWFpbCI6InRlc3RpbmdAdGVzdC5jb
        20ifSwiaWF0IjoxNTg2ODQyNDI1LCJleHAiOjE1ODc0NDcyMjV9.
        7fpRHM4V6cooOxw8NwmwSeLMAn8dCCC1DFIC9pZvinY"
}
```



KEY	VALUE
✓ Content-Type	application/json
✓ Authorization	Beare eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjp7I...
Key	Value

Then try to get all of the tweets again. This time, your request should pass through the `requireAuth` middleware functions successfully. Now that your tweets are protected, let's add a view for users to sign up from the client application!

Phase 3: Sign up for user account from the client

Begin by adding a GET route in the `express-apis-frontend/index.js` file to render a `sign-up` template:

```
app.get("/sign-up", (req, res) => {
  res.render("sign-up");
});
```

Use the snippet below to create a registration form for the `sign-up.pug` template.

```
extends layout.pug

block content
  div(class="sign-up-container")
    div(class="errors-container")
    h2 Sign Up
    form(class="sign-up-form")
      .form-group
        label(for='username') Username
        input#username.form-control(type='text' name='username' placeholder="User
      .form-group
        label(for='email') Email address
        input#email.form-control(type='email', name="email", placeholder='Enter e
      .form-group
        label(for='password') Password
        input#password.form-control(type='password', name="password", placeholder
      button.btn.btn-primary(type='submit') Sign Up
    a(href='/log-in') Already have an account? Log in here.
  script(src="js/sign-up.js")
```

In the code above, there's also an anchor element to navigate to the `/log-in` route that we'll implement next. Since the user will use the email and password input fields for the later log in form, let's move those to an *includes* file to DRY up our code.

Move the following `.form-group` elements from your `sign-up.pug` file to a new `views/includes/auth-form-fields.pug` file:

```
.form-group
  label(for='email') Email address
  input#email.form-control(type='email', name="email", placeholder='Enter email')
.form-group
  label(for='password') Password
  input#password.form-control(type='password', name="password", placeholder='Pass
```

Now take a moment to refactor your `sign-up.pug` template to use your new *includes* file.

At the bottom of the `sign-up.pug` template, notice that a `js/sign-up.js` script has already been imported. Set up this `sign-up.js` script inside your `express-apis-frontend/public/js` directory. In the script, you want to add a *submit* event listener to the sign up form (hint: search for the element with a class of `sign-up-form`) and an asynchronous callback function to handle the sign up request. Remember that `submit` events automatically prompt a GET request to re-render the page. Make sure to prevent the form from re-rendering by using `e.preventDefault()` at the beginning of your event listener:

```
const signUpForm = document.querySelector(".sign-up-form");

signUpForm.addEventListener("submit", async (e) => {
  e.preventDefault();
  // Sign up logic here
});
```

Generate a new `FormData` object with your `signUpForm`:

```
const formData = new FormData(signUpForm);
```

Now use the formData.get() method to retrieve the `username`, `email`, and `password` from the form. You can use the form values to declare a `body` variable:

```
const email = // TODO: Get email
const password = // TODO: Get password
const username = // TODO: Get username

const body = { email, password, username };
```

When a submit event happens, parse out the data from the form and then use a `fetch` call to make a POST request to `localhost:8080/users` to create the user in a `try` block. Set the `fetch` call's `method` to be "POST" and the `body` to be a JSON string of the form fields (hint: use `JSON.stringify({ email, password, username })`). Lastly, set a `Content-Type` header of "application/json".

Your fetch call should look something like this:

```
try {
  const res = await fetch("http://localhost:8080/users", {
    method: "POST",
    body: JSON.stringify(body),
    headers: {
      "Content-Type": "application/json",
    },
  });
}
```

Fetch calls don't throw errors on error status code responses. Note that the Fetch API only rejects network failure errors. This means you need to manually handle response errors by checking the response's `ok` property. Add the code snippet below in your `try` block to manually throw an error if the fetch request was rejected:

```
if (!res.ok) {
  throw res;
}
```

When the response body returns, store the JWT token and the user id in `localStorage` and redirect the user to the home page:

```
const {
  token,
  user: { id },
} = await res.json();

localStorage.setItem("TWITTER_LITE_ACCESS_TOKEN", token);
localStorage.setItem("TWITTER_LITE_CURRENT_USER_ID", id);
```

If an error happens, await the parsing of the error object as JSON and query for your element with the class of `errors-container`. Your `try` / `catch` block should currently look something like this:

```
try {
  // CODE OMITTED FOR BREVITY

  if (!res.ok) {
    throw res;
  }
  const {
    token,
    user: { id },
  } = await res.json();

  // CODE OMITTED FOR BREVITY
} catch (err) {
  if (err.status >= 400 && err.status < 600) {
    const errorJSON = await err.json();
    const errorsContainer = document.querySelector(".errors-container");

    // TODO: Generate and render errors
  } else {
    // TODO: Alert user about bad internet connection
  }
}
```

Like how you rendered your list of tweets when you first connected the frontend to your application, you'll set the `innerHTML` of your `errorsContainer` to a stringified HTML block. Sometimes there might not be an errors array returned, and in those cases, go ahead and declare the `errorsHtml` array to display a generic "Something went wrong" message.

```
// errorsHtml block to alert user of error:
let errorsHtml = [
  `
    <div class="alert alert-danger">
        Something went wrong. Please try again.
    </div>
  `,
];
```

Now take a look at the global error handling function in your `app.js` and notice how you are rendering an `errors` property in your JSON response. This `errors` property is the array of error messages from your validations. Iterate through the `errors` array and display them:

```
if (errors && Array.isArray(errors)) {
  errorsHtml = errors.map(
    (message) => `
    <div class="alert alert-danger">
        ${message}
    </div>
  `
  );
  // TODO: Join errorsHtml and set the errorsContainer.innerHTML
  // TODO: Error rendering
}
```

Since the Fetch API only throws errors on network errors (like if your internet cut out), so then we would also need to account for those types of errors. To account for these cases, go ahead and use the JavaScript `alert()` function to let the user know to check their internet connection:

```
alert(
  "Something went wrong. Please check your internet connection and try again!"
```

```
  "Something went wrong. Please check your internet connection and try again!"
);
```

If the create request succeeds, redirect the user back to the home page right after you set your `localStorage` items. Use the `window.location.href` property so that the user can see all existing tweets:

```
window.location.href = "/";
```

At this point, your `sign-up.js` file should look something like this:

```
const signUpForm = document.querySelector(".sign-up-form");

signUpForm.addEventListener("submit", async (e) => {
  e.preventDefault();
  const formData = new FormData(signUpForm);

  const username = formData.get("username");
  const email = formData.get("email");
  const password = formData.get("password");

  const body = { email, password, username };
  try {
    const res = await fetch("http://localhost:8080/users", {
      method: "POST",
      body: JSON.stringify(body),
      headers: {
        "Content-Type": "application/json",
      },
    });

    if (!res.ok) {
      throw res;
    }

    const {
      token,
      user: { id },
    } = await res.json();
    // storage access_token in localStorage:
    localStorage.setItem("TWITTER_LITE_ACCESS_TOKEN", token);
    localStorage.setItem("TWITTER_LITE_CURRENT_USER_ID", id);
    // redirect to home page to see all tweets:
    window.location.href = "/";
  } catch (err) {
    if (err.status >= 400 && err.status < 600) {
      const errorJSON = await err.json();
      const errorsContainer = document.querySelector(".errors-container");
      let errorsHtml = [
        `
        <div class="alert alert-danger">
            Something went wrong. Please try again.
        </div>
```

```
        `,
      ];
      const { errors } = errorJSON;
      if (errors && Array.isArray(errors)) {
        errorsHtml = errors.map(
          (message) => `
          <div class="alert alert-danger">
              ${message}
          </div>
        `
        );
      }
      errorsContainer.innerHTML = errorsHtml.join("");
    } else {
      alert(
        "Something went wrong. Please check your internet connection and try agai
      );
    }
  }
});
```

When a user signs up and is redirected to the home page, the request for all tweets should be failing because that request doesn't currently have the correct authentication headers. Let's fix this.

In the `public/js/index.js` file, add the `Authorization` header to your fetch call.

```
const res = await fetch("http://localhost:8080/tweets", {
  headers: {
    Authorization: `Bearer ${localStorage.getItem(
      "TWITTER_LITE_ACCESS_TOKEN"
    )}`,
  },
});
```

Then, add the following code into your "DOMContentLoaded" event listener to redirect users to the log-in route if a user is not logged in. The next step is to implement the log-in route!

```
if (res.status === 401) {
  window.location.href = "/log-in";
  return;
}
```

Now that you've implemented a sign-up flow, let's next implement a log-in flow!

Phase 4: Setting up the log-in flow

Go back to the API and add a route to the users router that allows users to fetch a new token, effectively logging them in. Do this by creating POST route to the `/token` path and using the `validateEmailAndPassword` middleware. Wrap your asynchronous route handler function with your `asyncHandler` helper method and destructure the `email` and `password` from your request body. Begin by using the `email` to find your user instance:

```
router.post(
  "/token",
  validateEmailAndPassword,
  asyncHandler(async (req, res, next) => {
    const { email, password } = req.body;
    const user = await User.findOne({
      where: {
        email,
      },
    });

    // TODO: Password validation and error handling
    // TODO: Token generation
  })
);
```

Now we need a way to validate whether or not the provided password is correct. Let's implement a password validation function in the User model as an instance method! Add the instance method below to your `user.js` model file:

```
User.prototype.validatePassword = function (password) {
  // Note that since this function is a model instance method,
  // `this` is the user instance here:
  return bcrypt.compareSync(password, this.hashedPassword.toString());
};
```

Note how the `hashedPassword` has to be converted back from its *binary* format to a *string* format before it can be compared with the provided `password`.

Back in the `/token` route handler, verify whether the a valid `user` has been found and use the `validatePassword` instance method to verify whether or not the user provided the correct credentials. If a valid user was found with a valid password, return a `token` and the user's `id` in the JSON response. If a valid user was not found or the password was incorrect, generate an error and return an invocation of the `next` function passing in the error:

```
if (!user || !user.validatePassword(password)) {
  const err = new Error("Login failed");
  err.status = 401;
  err.title = "Login failed";
  err.errors = ["The provided credentials were invalid."];
  return next(err);
}
```

Make sure to generate a token by invoking your `getUserToken()` function with the `user` fetched from your database. Lastly, render a JSON response with the `token` and the user's `id`:

```
const token = getUserToken(user);
res.json({ token, user: { id: user.id } });
```

Next, go back to the frontend and set up a `localhost:4000/log-in` route. The logic is fairly similar to the sign-up route. First, set up a GET route in your `express-apis-frontend/index.js` file to render a `log-in` template:

```
app.get("/log-in", (req, res) => {
  res.render("log-in");
});
```

Then create a `views/log-in.pug` template with the snippet below:

```
extends layout.pug

block content
  div(class="log-in-container")
    div(class="errors-container")
    h2 Log In
    form(class="log-in-form")
      include includes/auth-form-fields
      button.btn.btn-primary(type='submit') Log In
    a(href='/sign-up') Don't have an account? Sign up here.
  script(src="js/log-in.js")
```

Now create a `public/js/log-in.js` file and add a *submit* event listener to do fairly similar logic to the event listener in your `public/js/sign-up.js` file:

```
const logInForm = document.querySelector(".log-in-form");

logInForm.addEventListener("submit", async (e) => {
  e.preventDefault();
  const formData = new FormData(logInForm);
  const email = formData.get("email");
  const password = formData.get("password");
  const body = { email, password };
  try {
    const res = await fetch("http://localhost:8080/users/token", {
      method: "POST",
      body: JSON.stringify(body),
      headers: {
        "Content-Type": "application/json",
      },
    });
    if (!res.ok) {
      throw res;
    }
    const {
      token,
      user: { id },
    } = await res.json();
    // storage access_token in localStorage:
    localStorage.setItem("TWITTER_LITE_ACCESS_TOKEN", token);
    localStorage.setItem("TWITTER_LITE_CURRENT_USER_ID", id);
    // redirect to home page to see all tweets:
    window.location.href = "/";
  } catch (err) {
    if (err.status >= 400 && err.status < 600) {
      const errorJSON = await err.json();
      const errorsContainer = document.querySelector(".errors-container");
      let errorsHtml = [
        `
        <div class="alert alert-danger">
            Something went wrong. Please try again.
        </div>
      `,
      ];
      const { errors } = errorJSON;
      if (errors && Array.isArray(errors)) {
        errorsHtml = errors.map(
          (message) => `
          <div class="alert alert-danger">
              ${message}
          </div>
        `
        );
      }
      errorsContainer.innerHTML = errorsHtml.join("");
    } else {
      alert(
        "Something went wrong. Please check your internet connection and try agai
      );
    }
  }
});
```

At this point, you might be thinking to yourself that this all seems super repetitive. In server-side Node.js code, you've been using CommonJS modules to DRY up your code. Unfortunately, you haven't learned about client-side modules yet, but when you get to React, you'll learn more about using a frontend module system in order to clean up your code!

Now create a new user, and then go to the log in form to verify that the log in process works when you log in with the created user credentials.

Phase 5: Creating tweets from the frontend application

Let's first reorganize the API so that tweets belong to a specific user. First, rollback all of your migrations by running the command below:

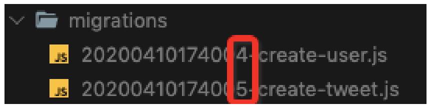
```
npx dotenv sequelize db:migrate:undo:all
```

Then since we need to associate tweets with a user, add a foreign key of `userId` so that tweets can belong to users. Here's what your updated tweets migration file should look like:

```
"use strict";
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable("Tweets", {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER,
      },
      message: {
        type: Sequelize.STRING(280),
        allowNull: false,
      },
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE,
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE,
      },
      userId: {
        allowNull: false,
        type: Sequelize.INTEGER,
        references: {
          model: "Users",
          key: "id",
        },
      },
    });
  },
  down: (queryInterface, Sequelize) => {
    return queryInterface.dropTable("Tweets");
  },
};
```

However, there's a problem right now. The `create-tweet` migration file was created first, and because of the timestamp in the first part of the file name, that specific migration will run first. The problem is that when we try to reference the Users table in the `references` definition part of the migration, the Users table has not actually been created yet. We're going to take a *hacky* approach to resolve this issue and make the `create-user` migration run first.

To be clear, this is absolutely not how you would want to handle migrations in production. This whole "undo all migrates and then switch up the order of migrations" is purely for learning purposes and to help you get a better of how migrations work. So, go ahead and rename the `create-tweet` migration file so that the timestamp comes later than the `create-user` migration file:



Now, that you've updated the migrations to properly configure the tables in your database, update the models to also reflect this relationship. This is what your tweet model should look like:

```
"use strict";
module.exports = (sequelize, DataTypes) => {
  const Tweet = sequelize.define(
    "Tweet",
    {
      message: {
        type: DataTypes.STRING(280),
        allowNull: false,
      },
    },
    {}
  );
  Tweet.associate = function (models) {
    Tweet.belongsTo(models.User, {
      as: "user",
      foreignKey: "userId",
    });
  };
  return Tweet;
};
```

This is what your user model should look like after you add the associations:

```
"use strict";
const bcrypt = require("bcryptjs");

module.exports = (sequelize, DataTypes) => {
  const User = sequelize.define(
    "User",
    {
      email: { type: DataTypes.STRING, allowNull: false, unique: true },
      username: { type: DataTypes.STRING, allowNull: false, unique: true },
      hashedPassword: {
        type: DataTypes.STRING.BINARY,
        allowNull: false,
      },
    },
    {}
  );
  User.associate = function (models) {
    User.hasMany(models.Tweet, {
      as: "tweets",
      foreignKey: "userId",
    });
  };

  User.prototype.validatePassword = function (password) {
    // because this is a model instance method, `this` is the user instance here:
    return bcrypt.compareSync(password, this.hashedPassword.toString());
  };

  return User;
};
```

Finally, let's also update the seeders to reflect this new relationship between tweets and users. We'll also use the faker library to generate fake content, so run `npm install faker`, and then update the seeders to:

```
"use strict";

const faker = require("faker");
const bcrypt = require("bcryptjs");

module.exports = {
  up: async (queryInterface, Sequelize) => {
    const users = await queryInterface.bulkInsert(
      "Users",
      [
        {
          username: faker.internet.userName(),
          email: faker.internet.email(),
          hashedPassword: bcrypt.hashSync(faker.internet.password()),
          createdAt: new Date(),
          updatedAt: new Date(),
        },
        {
          username: faker.internet.userName(),
          email: faker.internet.email(),
          hashedPassword: bcrypt.hashSync(faker.internet.password()),
          createdAt: new Date(),
          updatedAt: new Date(),
        },
      ],
      { returning: true }
    );

    return queryInterface.bulkInsert(
      "Tweets",
      [
        {
          message: faker.company.catchPhrase(),
          createdAt: new Date(),
          updatedAt: new Date(),
          userId: users[0].id,
        },
        {
          message: faker.company.catchPhrase(),
          createdAt: new Date(),
          updatedAt: new Date(),
          userId: users[0].id,
        },
        {
          message: faker.company.catchPhrase(),
          createdAt: new Date(),
          updatedAt: new Date(),
          userId: users[1].id,
        },
      ],
      {}
    );
  },

  down: async (queryInterface, Sequelize) => {
    await queryInterface.bulkDelete("Tweets", null, {});
    return queryInterface.bulkDelete("Users", null, {});
  },
};
```

Now that you've updated the migrations, models, and seeders to reflect the tweets *belongsTo* users and user *hasMany* tweets relationships, let's run all of the migrations and seed scripts:

```
npx dotenv sequelize db:migrate
npx dotenv sequelize db:seed:all
```

Check Postbird to verify all the data has been properly created. Go to `routes/tweets.js` and update the post route so that when a tweet is created, the `id` of the `req.user` is stored in the `userId` column of the tweet:

```
const tweet = await Tweet.create({ message, userId: req.user.id });
```

Finally, update the GET `/` tweets route so that it also includes the author of the tweet. You previously learned about eager loading associations. When designing APIs, it's critical that you don't return any more info than is necessary.

So for example, in this case, when you eager load the user, if you didn't filter anything out, then it would include extra info like user email address or the hashedPassword, when really all the client needs to know from this route is the username of the author of the specific tweet and perhaps the user id of the author.

Fortunately, we can use sequelize `attributes` to filter out the fields that get returned from the sequelize query:

```
const tweets = await Tweet.findAll({
  include: [{ model: User, as: "user", attributes: ["username"] }],
  order: [["createdAt", "DESC"]],
  attributes: ["message"],
});
```

Let's now go back to the client to set up a form for an authenticated user to create tweets!

Phase 6: Seeing tweets with authors and creating tweets from the client

First, let's update the `public/js/index.js` script to render the username on top of the message (hint: destructure the user from each tweet and then destructure the username `{ message, user: { username } }`).

Update the `tweetsHtml` to render a `username` as content within a `div.card-header` element. Make the `div.card-header` element a child of the `div.card` element. Feel free to make use fo the HTML snippet below:

```
<div class="card-header">
  ${username}
</div>
```

Next, let's set up a `create` view for the user to create a tweet. First add the route in your `express-apis-frontend/index.js` file:

```
app.get("/create", (req, res) => {
  res.render("create");
});
```

Then set up the associated `create.pug` template with the snippet below:

```
extends layout.pug

block content
  .errors-container
  form(class="create-form")
    .form-group
      label(for='message') Message
      input#message.form-control(type='message', name="message", placeholder='Wha
    button.btn.btn-primary(type='submit') Tweet

  script(src="js/create.js")
```

Before we implement the `public/js/create.js` script file, let's update the `layout.pug` template to now also render a nav bar at the top of the application to allow users to easily navigate around:

```
doctype html
html
  head
    title Twitter Lite
    link(rel='stylesheet' href='https://stackpath.bootstrapcdn.com/bootstrap/4.4.
    link(rel='stylesheet' type='text/css' href='/style.css')
  body
    nav.navbar.navbar-expand-lg.navbar-light.bg-light
      a.navbar-brand(href='/') Twitter Lite
      .navbar-nav
        a.nav-item.nav-link(href='/create') Create Tweet
        a.nav-item.nav-link(href='/profile') My Tweets
    block content
```

Notice how there's a link to a `/profile` route, which we will implement in the next phase. Now create the `public/js/create.js` file. In this file, begin by adding an event listener to the element with a class name of `create-form`. Remember to use `e.preventDefault()` within the event listener to prevent the form from sending a GET request upon submission. Then parse the form data:

```
const form = document.querySelector(".create-form");

form.addEventListener("submit", async (e) => {
  e.preventDefault();
  const formData = new FormData(form);
  const message = formData.get("message");
  const body = { message };

  // TODO: Fetch tweets
  // TODO: Redirect users to login page upon a 401 error
  // TODO: Handle errors
});
```

Within a `try` block, use the Fetch API to make a POST request to `http://localhost:8080/tweets` with the necessary `body` (formatted as a JSON string) and headers (`Content-Type` and `Authorization`). Upon a successful fetch request, remember to redirect the user to the home page by using the `window.location.href` property.

Now handle the following response and errors caught. If you receive a response with a status of `401`, redirect users to log-in page. Throw the response (`res`) for any other response statuses. If you catch an error with a status greater than or equal to `400` and less than `600`, parse your caught error as JSON and find your `errorsContainer` (element with a class of `errors-container`). Declare an array of error elements by using the `errorsHtml` snippet below:

```
let errorsHtml = [
  `
  <div class="alert alert-danger">
      Something went wrong. Please try again.
  </div>
`,
];
```

Now destructure your `errors` array from your error parsed as JSON and verify if `errors` is a valid, truthy array. If `errors` is a valid array, map through each `message` in the array to generate an array of stringified error elements:

```
const { errors } = errorJSON;
if (errors && Array.isArray(errors)) {
  errorsHtml = errors.map(
    (message) => `
    <div class="alert alert-danger">
        ${message}
    </div>
  `
  );
}
```

Now join the elements within your `errorsHtml` array and set the joined array as the `innerHTML` of the `errorsContainer`. Lastly, use the `alert()` method to notify the end user with a generic error message.

Phase 7: Set up the profile page

In the profile page, a user should be able to see their own tweets. Up to this point we haven't dealt with nested resources yet, but let's go ahead and build out a route where tweets are nested under a user.

In the users router, add an endpoint for `localhost:8080/users/:id/tweets`. You'll need to also import the Tweet model into this router. This route should await the database fetch of all tweets that belong to a specific user based on the `id` in the request `params`.

Make sure to import and your `requireAuth` middleware from the `../auth.js` file to keep your tweets *protected*. Also be sure to wrap the route handling function with your `asyncHandler` to be able to `await` the database fetch. Lastly, render the tweets in a JSON response.

Go to the client now and set up the profile page. Add the following route in the `express-apis-frontend/index.js` file to serve a `profile.pug` template:

```
app.get("/profile", (req, res) => {
  res.render("profile");
});
```

Then set up the `profile.pug` template:

```
extends layout.pug
block content
  h2 Your Tweets
  .tweets-container
  script(src="js/profile.js")
```

Finally, set up a script in the `public/js/profile.js` file to fetch all of the tweets that belong to the logged in user. Begin by adding a `DOMContentLoaded` event listener. Now remember that after a user logs in or signs up, the server gives the client the user's id, which is stored in localStorage. Use that id to `try` to make a fetch request to the correct route (hint: interpolate the user's id into the fetch path). Make sure to include an `Authorization` header that gets the `TWITTER_LITE_ACCESS_TOKEN` from localStorage.

Upon a response status of `401`, re-direct the user to the `/log-in` page by using the `window.location.href` property and `return` out of the event listener function.

Upon a successful fetch, parse the response as JSON and destructure the response `tweets`. Search for the `tweetsContainer` (element with a class name of `tweets-container`) and declare a `tweetsHtml` variable. Now map over the `tweets` from your fetch response to generate an array of stringified tweet elements and set the `tweetsHtml` variable:

```
const tweetsHtml = tweets.map(
  ({ message, id }) => `
  <div class="card" id="tweet-${id}">
    <div class="card-body">
      <p class="card-text">${message}</p>
    </div>
  </div>
`
);
```

Make sure to join your stringified tweet elements to set the `innerHTML` of the `tweetsContainer`:

```
tweetsContainer.innerHTML = tweetsHtml.join("");
```

Lastly, use `console.error()` to log any caught errors.

Nice work making it to this part of the project! In the next bonus phases, let's add a little more functionality, improve the UX of the app, and then also clean up the code!

Bonus Phase: Add ability to delete your own tweet

Let's add the ability to delete your own tweets, while not allowing anyone else to delete them!

Update your `profile.js` script so that after you fetch your own tweets, each message can show a delete button. Then, after the HTML has been added to the DOM, select all of the delete buttons to add a click handler to each button. The click handler's callback function should make a DELETE request to `http://localhost:8080/tweets/:id` in order to delete a tweet.

To add click handlers for each delete button, here's what you might want to write at the bottom of your "DOMContentLoaded" event listener in the `profile.js` file:

```
const deleteButtons = document.querySelectorAll(".delete-button");
if (deleteButtons) {
  deleteButtons.forEach((button) => {
    button.addEventListener("click", handleDelete(button.id));
  });
}
```

In the code snippet above, there's a `handleDelete` callback function that's not shown. Go ahead and implement the logic for that function!

Next, let's add some logic so that only the author of a tweet can actually delete the tweet. Go to the endpoint for deleting tweets. How can you check whether or not the current user who's making this request is actually authorized to delete this tweet?

Once you've tested that a tweet can only be deleted by its author in Postman, move on to the next bonus phase!

Bonus Phase: Combine create and index views

Let's make it easier for users to create a tweet in the app. Right now, the whole process to create a tweet is a little cumbersome. Users have to navigate to a separate page for creating the tweet, and after it's created, they're then redirected back to another page to see the newly created.

Go ahead and move the create tweet form from `create.pug` to above the tweets container in `index.pug`. This will give the users the ability to create a tweet at the top of their timeline, which is an experience that's more in line with what the real Twitter does.

In this view, when users create a tweet, go ahead and just re-fetch all of the tweets again. Be sure to keep your code DRY here by extracting the logic to fetch all tweets into its own function.

Also, since you are no longer redirecting users to a different page after a tweet is created make sure you are clearing out the create tweet form inputs upon a successful tweet creation.

As a reminder, since you no longer need a create tweet page, go ahead and delete those template and JavaScript files, and then remove the navigation route from `layout.pug`.

When you are done with this phase, you should have an index page that allows users to create a tweet and then see that newly created tweet without ever requiring a page reload. This type of experience is more in line with what users expect from modern web apps. You'll dive deeper into how to build these types of modern frontends as you progress into the React curriculum soon!

Bonus Phase: ES Modules

Finally, let's clean up some of the JavaScript code in the client app! Right now a lot of the error handling logic is being duplicated between the different files. DRY up your code with JavaScript modules.

You'll need to declare `type="module"` in any script tags that load JavaScript files that use modules. Also, unlike the CommonJS module system (`require` and `module.exports`) that you've been using in your server-side JavaScript, you'll need to use the ES module system (`import` and `export`) in client-side JavaScript.

In the JavaScript modules documentation, there's a section that talks about how ES modules are not supported by all browsers. In the React curriculum, you'll learn more about how to effectively handle client-side modules using build tools like webpack.

WEEK-12 DAY-4
Apis

API Security Learning Objectives

Once you have an API up and running, you need to secure it to ward off leaking sensitive data or allowing people to use it in an unauthorized way. To do so, you will introduce the concept of authentication and authorization, again, but with tools that help support the stateless nature of RESTful APIs. After your homework, lecture, and practice, you should be able to

- Explain the fundamental concepts of OAuth as a way to authenticate users
- Describe the workflow of OAuth resource owner password credentials grant (RFC 6749 Section 4.3)
- Describe the components of a JSON Web Token (JWT) and how it is constructed
- Configure an Express application to use token-based authentication and authorization using OAuth resource owner password credentials grant

---

OAuth 2.0

As an application developer, your application sometimes needs to access your user's information from a different web application.

For example, let's say you're working on a website called MyMint.com, a simple site that lets people with multiple bank accounts access all of their accounts in one centralized place. In this example, Johnny wants to use your web app because he has an account with Chase and Bank of America, and he's tired of having to go back and forth between chase.com and bankofamerica.com in order to check his financial transactions.

So, Johnny signs up for MyMint.com, and you now need to be able to get Johnny's transactions from chase.com.

In the past, one way for you to access that information might have been for Johnny to directly give your application his password, and then your web app would go to chase.com to authenticate and then fetch the transactions. This was clearly not ideal from a security point of view.

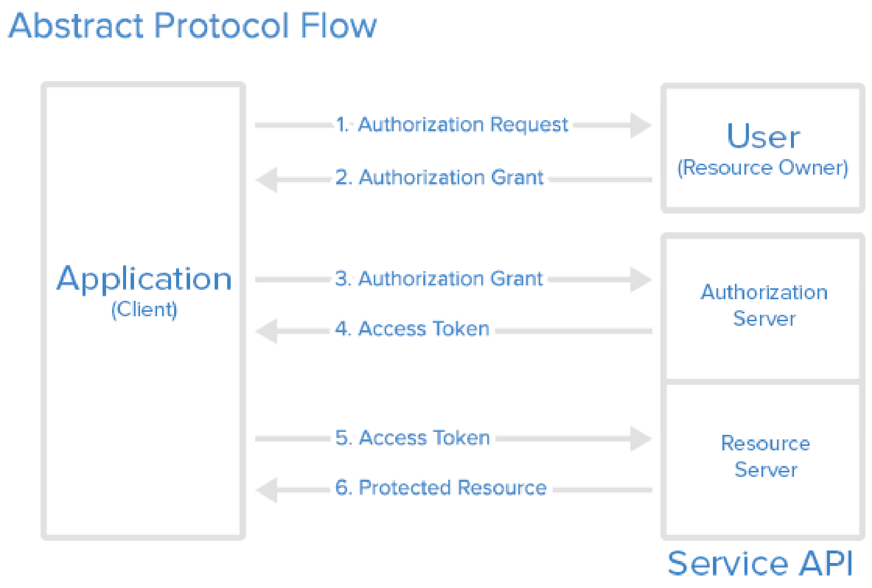
OAuth 2.0 is a standard that allows for you to access that information without Johnny giving you his passwords to his bank account logins. It is defined by RFC 6749 "The OAuth 2.0 Authorization Framework". OAuth 2.0 defines a very specific protocol for how your app (MyMint.com), Johnny, and chase.com should coordinate together so that you can securely get Johnny's chase.com transactions.

This article will help you learn how to

- Get your application ready for OAuth 2.0
- Teach you about how OAuth 2.0 works
- Use OAuth 2.0 to authenticate (and authorize) a person for your application from a third-party authority

High level overview of OAuth 2.0

At a high level, there are six main steps to the OAuth 2.0 standard. Here is a high-level diagram from a Digital Ocean overview of OAuth 2.0:



Here are the key players mentioned in the diagram above:

1. **Application**- In our example, the application, or client, is the app that you've built-- MyMint.com.
2. **User**- The user in our example is Johnny. The user is also called the **Resource Owner** because he is the owner of the **protected resource**, which in our example is his financial transactions in chase.com. Other examples of resources that Johnny could have at chase.com would be his mortgage account information and his IRA information.
3. **Service API** We can refer to chase.com in our example as the **Service API**. Notice that it's broken down further into an **Authorization Server** and a **Resource Server**. The **Authorization Server** is solely responsible for authorizing the **Application** (MyMint.com). The **Resource Server** is where the **protected resources** (Johnny's financial transactions) are actually stored and served.

Now, let's do a high-level walk through of the six steps above using our example. Each of these will be covered in more detail later in the article. However, it's good to hold them in context as they're discussed.

1. **Application requests authorization from user**- Your app, MyMint.com, asks Johnny for authorization to access Johnny's financial transactions from chase.com.
2. **User issues Authorization Grant to Application**- Johnny authorizes the MyMint.com, and an Authorization Grant is issued to MyMint.com.
3. **Application uses the Authorization Grant**- MyMint.com uses the Authorization Grant to request an access token from chase.com's Authorization Server.
4. **Authorization Server issues access token to Application**- chase.com's Authorization Server checks the authorization grant and issues an access token to MyMint.com. From this point on, MyMint.com will use that access token to communicate with chase.com's Resource Server.
5. **Access token is used**- MyMint.com uses the access token to request Johnny's financial transactions from chase.com's Resource Server.
6. **Protected Resources are served**- If chase.com's Resource Server is presented with a valid access token, then it sends back the resources (Johnny's financial transactions).

Getting your app ready for OAuth 2.0

In the high-level overview above, it seems like Johnny gave MyMint.com permission to access chase.com's resource, and chase.com just agreed to it. At this point, you might be wondering why chase.com would be willing to send MyMint.com protected resources. After all, how can chase.com trust that MyMint.com is actually a credible web application?

In reality, you would need to first register MyMint.com with chase.com's API before you are able to start requesting resources from it.

Specifically, you would need to provide details about your website and supply a callback URL (more on this later). Once you are registered, chase.com will give you a set of credentials to use during the authorization process. The credentials include:

1. **client ID**- public unique identifier for your web application
2. **client secret**- a password that only your application and the API will know about. MyMint.com will use this when requesting resources from chase.com's servers.

OAuth 2.0 in detail

Now that you have learned about OAuth 2.0 at a high level, as well as the prerequisites for getting your site ready for OAuth 2.0, let's discuss each step in more detail.

Everything over HTTPS

Note that all communication that occurs in the following sections occurs over HTTPS. This provides encrypted communications between the application and the service APIs. That way, no one can sniff the information as it goes by across the Internet, get the secret information being sent, and use it to maliciously impersonate Johnny, MyMint.com, or chase.com.

Application requests authorization from user

Let's imagine that Johnny is logged in to MyMint.com and now trying to add the ability to start viewing his Chase financial transactions. For simplicity's sake, let's suppose that your app, MyMint.com, has a link for Johnny to click when Johnny wants to add his Chase account to MyMint.com. This link would look something like this:

```
authorization.chase.com/?client_id=${MyMint_com_client_id}&redirect_uri=${callback_url}
```

Let's break down the components of the URL above:

1. authorization.chase.com- This URL is an example of what chase's authorization endpoint might look like.
2. `client_id=${MyMint_com_client_id}` - This is MyMint.com's unique ID that was provided when you registered MyMint.com with chase.com's API services. chase.com uses this to identify your app.
3. `redirect_uri=${callback_url}` - During the registration process, you also provided a `callback_url`, which is the URL that you want chase.com to redirect Johnny to after Johnny authorizes chase.com to send MyMint.com resources. **For the purposes of our example, let's assume that you decided to use `https://mymint<span>.com</span>/callback-time` as your app's callback URL**.

User issues authorization grant

Once Johnny clicks on this link, Johnny would be directed to chase.com's authorization page, where he logs in directly to chase.com by providing his Chase username and password. Once Johnny has logged in to his chase.com account, chase.com will ask Johnny whether or not he actually wants to authorize MyMint.com to access his Chase financial transactions data.

If Johnny authorizes MyMint.com, then Chase will issue an Authorization Grant to MyMint.com.

There are actually multiple forms of an **authorization grant**. For this example, the type of authorization grant that we will focus on is the **authorization code**. chase.com will send an authorization code to MyMint.com so that MyMint.com can authenticate itself. chase.com sends this code when it redirects the user back to the callback URL that you provided. That redirect URL for Johnny would look something like this:

```
https://mymint.com/callback-time?code=dlsie239084903j4092j340j
```

That random string of characters, "dlsie239084903j4092j340j", that's the authorization code that chase.com sends to your application. These types of codes are usually only valid for a short period of time. That greatly reduces the chances of a bad actor from being able to guess a valid authentication code.

Note that because he is interacting directly with chase.com, at no point in time would Johnny ever provide his chase.com credentials to your app, MyMint.com.

Application uses the authorization grant

Now that MyMint.com has received an authorization code, it's ready to authenticate itself with chase.com so that it can start receiving Johnny's financial transaction data.

To do this, MyMint.com will send a POST request to chase.com's auth server using the recently provided authorization code. That post request might look something like this:

```
const fetch = require('node-fetch');
const qs = require('querystring');

const options = {
  method: 'POST',
  headers: {'content-type': 'application/x-www-form-urlencoded'},
  body: qs.stringify({
    grant_type: 'authorization_code',
    client_id: `${mymint.com_client_id}`,
    client_secret: `${mymint.com_client_secret}`,
    code: `${authorization_code}`,
    redirect_uri: 'https://mintmint.com/callback-time' // same callback_url you p
  })
};
const url = 'https://authorization.chase.com/oauth/token';
const response = await fetch(url, options);
const data = await response.json();
console.log(data);
```

Note that in the POST request above, you provided the `client_id`, `client_secret` and `authorization_code`.

Authorization server issues an access token

If everything is valid, then chase.com's auth server will send back a response that contains the access token. That response might look like this:

```
{
  "access_token": "eyJz93ak4laUWw",
  "refresh_token": "GEbRxBNedjnXbL",
  "id_token": "eyJ0XAi4faeEoQ",
  "token_type": "Bearer"
}
```

MyMint.com is now authorized! Now, whenever MyMint.com wants to request Johnny's transaction data, it would send along the `access_token` with each request.

Access token is used

From this point forward, the request for Johnny's transaction data would be made to chase.com's resource servers. Here's an example of what that request might look like:

```
var request = require("node-fetch");

const url = 'https://resources.chase.com/api/johnny/transactions';
const options = {
  headers: {
    'content-type': 'application/json',
    'authorization': `Bearer ${access_token}`
  }
};

const response = await fetch(url, options);
const data = await response.json();
console.log(data);
```

Protected resources are served

If the access_token is valid, then chase.com will send back the requested resources, which in this case is Johnny's financial transaction data.

What you learned

During this article, you learned about OAuth 2.0 and the interaction between your application and a server that can authenticate and authorize people's credentials. You saw that there is a six-step process to this interplay over HTTPS:

1. Your application requests authorization from the person
2. The person informs the authorization server that they want to issue an authorization grant for your application
3. The application uses the authorization grant with its secret information to request an access token
4. The authorization server returns an access token
5. Your application uses the access token to gain access to the user's data from the service API
6. The service API returns the resource that the token allows access to

---

Token-Based Authentication

Up until now, you've been building out a todo-list API that only dealt with tasks. A todo-list app without users is not very useful, so let add some users!

In particular, let's set up users and a way of authenticating users so that there is an accurate way of keeping track of who tasks belong to.

In this lesson, you will:

1. Understand what a JSON Web Token (JWT) is and how it can be used to securely send user information between servers or across requests.
2. Add token-based (session-less) authentication to an Express API.
3. Use the jsonwebtoken npm package to sign and verify JWTs.

JSON Web Token (JWT)

In a previous lesson, you learned about session-based authentication. With session-based authentication, when a user logs in, the server stores information about the session and sends back a session id to the client as a cookie. When the user makes a subsequent request, the cookie is presented and checked against the session data that's stored on the server side. To emphasize, with session-based authentication, data is stored on the server to keep track of sessions.

This presents an issue when it comes to building a RESTful API. Specifically, one of the constraints of REST is statelessness. As a reminder, the term stateless means that the data received from the server can be used by the client independently. Under the statelessness constraint, every request from the client should contain all necessary information for the server to process that request, and the server should not be storing any data about the client state.

What we need here is a way to identify that the user is logged in without requiring the server to store anything. JSON Web Tokens (JWTs) are perfect for these situations.

Conceptual overview of a JWT: an example

So what is a JWT? Let's first gain some conceptual understanding with an example.

Let's say you're throwing a party. It's a gonna be a big party, and everyone wants to come, but unfortunately only people you send an invite to are allowed to come.

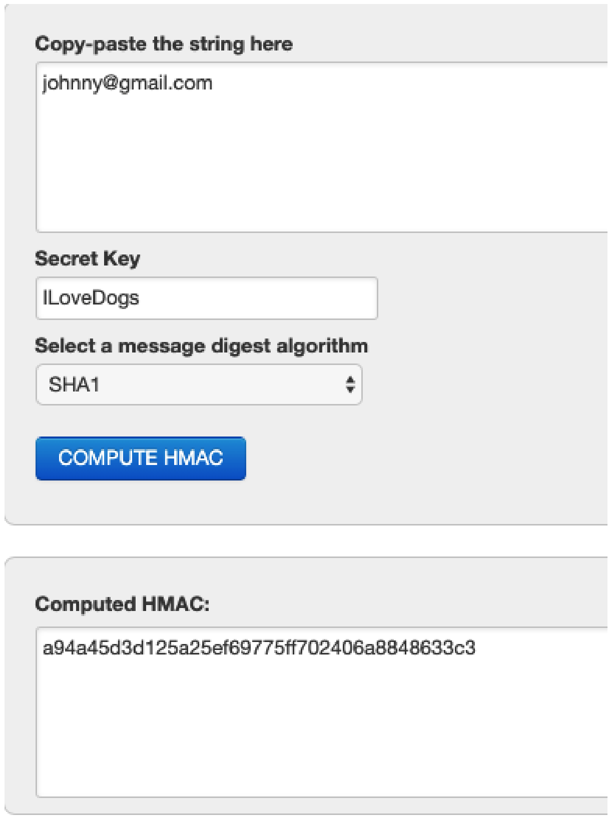
You prepare to start sending out invites, and you need to figure out a way to make sure only your invited guests are allowed into the party. One way would be to just keep a guest list, but you don't want to have to maintain a guest list.

So, you devise a genius way to send out tamper-proof invites. This invitation method requires:

1. **the guest's email address**: This is effectively a unique identifier for your guests, so for example let's say you want to invite your friend Johnny Rocket, whose email address is `johnny@gmail.com`.
2. **secret key/password that only you know**: This is the only thing that you have to store on your end. You decide that your secret password will be "ILoveDogs".
3. **a hashing function**: A hashing function is something that will take your guest's email and your secret password as inputs and then output a specific string digest. Keep in mind, the same input email and secret password will always produce the same output string digest. A key feature of a good hashing function is that it is not invertible. In other words, there should be no way to figure out what the inputs were based on the output string digest other than just trying to brute force your way into it.

> Some popular hashing functions are: SHA1, MD5, SHA2, Scrypt, HS256, and Blowfish. In the first part of this article, you'll see screenshot demos that use the SHA1 hashing function. When you set up the jsonwebtoken library in this reading's project, you'll be using the HS256 hashing function (default algorithm of the jsonwebtoken library). Whenever you use BCrypt for encryption in your projects, you will use Blowfish, as it is the default hashing function used with BCrypt.

Here's what you do (feel free to follow along). First, you take the guest's email address (`johnny@gmail.com`) and your secret password that nobody else knows (`ILoveDogs`), and you hash these two inputs with a SHA1 hashing function, which produces a string digest of `a94a45d3d125a25ef69775ff702406a8848633c3`:



johnny@gmail.com

ILoveDogs

SHA1

a94a45d3d125a25ef69775ff702406a8848633c3

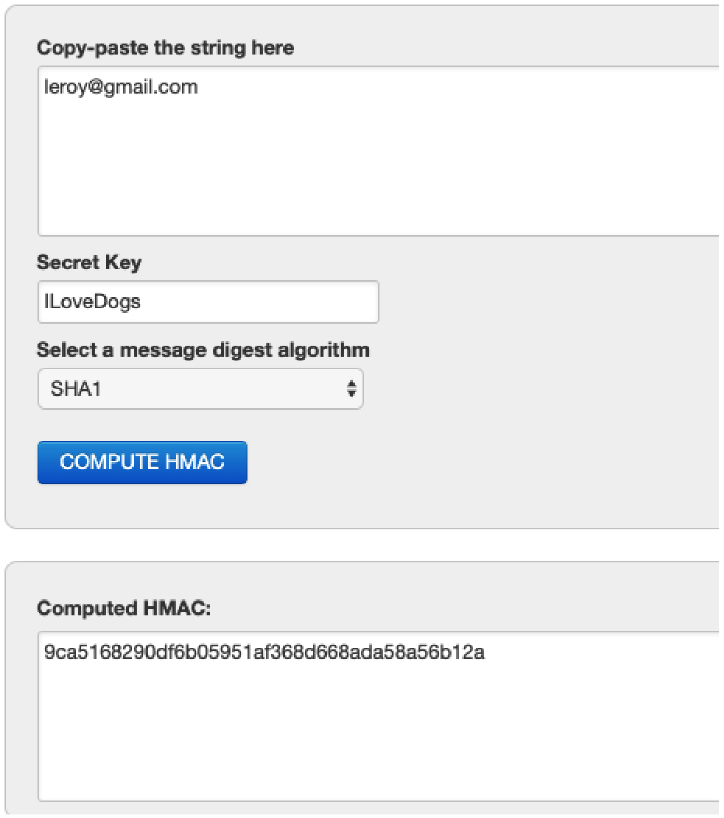
You then send an email to Johnny:

```
Dear Johnny,

My party is this Friday. Present this code at the door:

"johnny@gmail.com/a94a45d3d125a25ef69775ff702406a8848633c3"
```

Now, when Johnny shows up at your party on Friday, he presents you with `johnny@gmail.com/a94a45d3d125a25ef69775ff702406a8848633c3`. You take the first part of the code that he presents, which is his email, and hash it again with your secret password. You then compare the output against the second part of the code, the string digest, that Johnny presents you with. If it's the same, then you know the invite is valid and hasn't been tampered with because the string digest couldn't have been generated without your secret key.

You have a frenemy named Leroy who didn't get invited. He's friends with Johnny Rocket, so he asks Johnny for the string digest, figuring that's the secret code that everyone needs to present to get in. Leroy shows up at your door and presents his own version of the code, which is just his email plus the digest that Johnny got: `leroy@gmail.com/a94a45d3d125a25ef69775ff702406a8848633c3`.

So you run `leroy@gmail.com` + `ILoveDogs` through your hashing function:


Copy-paste the string here

leroy@gmail.com

Secret Key

ILoveDogs

Select a message digest algorithm

SHA1

Computed HMAC:

9ca5168290df6b05951af368d668ada58a56b12a

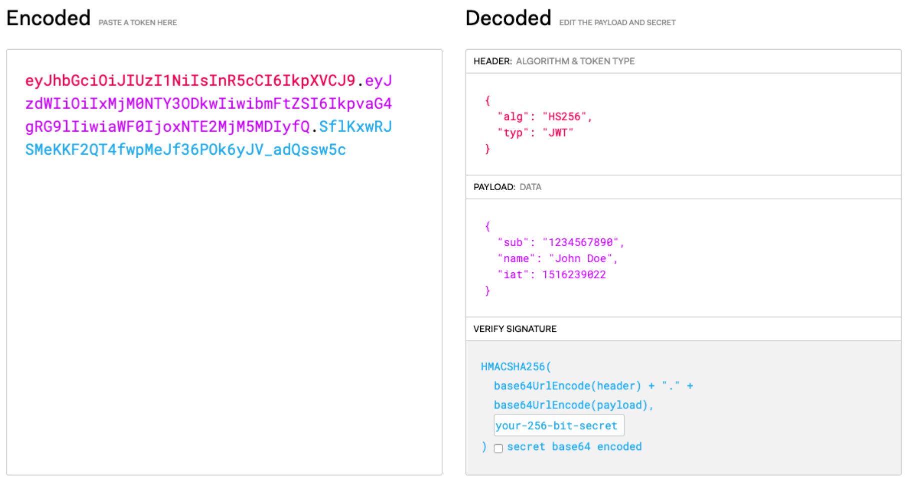
Immediately, you know that Leroy has an invalid invite because the string digest output of the hashing function (`9ca5168290df6b05951af368d668ada58a56b12a`) was different than the one he presented.

This concept of validating a "code" against the hashed output of a value and a secret password is essentially how a JSON Web Token (JWT) works.

Anatomy of a JWT

JSON Web Token is actually an internet standard that defines how to create JSON-based access tokens.

A JWT is composed of three parts: the header, the payload, and the signature.


Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
zdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJ
SMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

If you look at the left side of the image above, you can see the typical format of a JWT. The three parts of the JWT are separated by a period. Although the JWT might seem super cryptic, the first two parts are simply base64 encoded. The right side shows the base64 decoded version of the first two parts of the JWT. To clarify, base64 encoding is not an encryption mechanism. Therefore, a JWT is not encrypted by default and anyone who gains access to the token can see the contents of the payload.

The header describes the hashing algorithm that the JWT uses. The payload is the data being stored in the token. The signature is a hash of the header + the payload + a secret key.

To tie it back into our previous example, if you were to follow the JWT standard for your party invite to Johnny Rocket, it might have looked something like this:

```
base64Encode({ "typ": "JWT", "alg": "SHA1" }) // header
  .base64Encode({ "email": "johnny@gmail.com" }) // payload
  .SHA1HASH(header + payload + "ILoveDogs") // signature
```

Ultimately, you can use a JWT to identify that someone is logged in because you can store identifying information in the payload, and you can also verify the validity of the information. In contrast to sessions, which required storing data on the server-side, a JWT has all of the information you need.

Setting up token-based authentication

Now that you know what a JWT is, let's implement a token-based authentication flow in the todo-list app that we've been working through in the last few readings.

Create the Users table

First, create a migration to create the Users table by running:

```
npx sequelize model:generate --name User --attributes email:string,hashedPassword:
```

This should create a new `db/migrations/[timestamp]-create-user.js` migration file and a new `db/models/user.js` file.

Update the `db/models/user.js` file so that the `email` is unique and not nullable and the `hashedPassword` is type `STRING.BINARY` and not nullable. When you're done updating the model file, it should look something like this.

```
"use strict";
module.exports = (sequelize, DataTypes) => {
  const User = sequelize.define(
    "User",
    {
      email: {
        type: DataTypes.STRING,
        allowNull: false,
        unique: true,
      },
      hashedPassword: {
        type: DataTypes.STRING.BINARY,
        allowNull: false,
      },
    },
    {}
  );
  User.associate = function (models) {
    // associations can be defined here
  };
  return User;
};
```

Then, update the `db/migrations/[timestamp]-create-user.js` file so that the fields and its properties matches the User model:

```
"use strict";
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable("Users", {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER,
      },
      email: {
        type: Sequelize.STRING,
        allowNull: false,
        unique: true,
      },
      hashedPassword: {
        type: Sequelize.STRING.BINARY,
        allowNull: false,
      },
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE,
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE,
      },
    });
  },
  down: (queryInterface, Sequelize) => {
    return queryInterface.dropTable("Users");
  },
};
```

Run `npx dotenv sequelize db:migrate`. Verify that you now have a `Users` table with the correct columns in Postbird.

Users router

Next, create a new router for users at `routes/users.js`. In the users router, you'll need the `asyncHandler` and `handleValidationErrors` functions that are currently defined in `routes/tasks.js`. Go ahead and start a new `utils.js` file at the directory root, and move the `asyncHandler` function and `handleValidationErrors` function from `routes/tasks.js` into `utils.js`. This is what your `utils.js` file should look like:

```
const { validationResult } = require("express-validator");
const asyncHandler = (handler) => (req, res, next) =>
  handler(req, res, next).catch(next);

const handleValidationErrors = (req, res, next) => {
  const validationErrors = validationResult(req);

  // If the validation errors are empty,
  if (!validationErrors.isEmpty()) {
    // Generate an array of error messages
    const errors = validationErrors.array().map((error) => error.msg);

    // Generate a new `400 Bad request.` Error object
    // and invoke the next function passing in `err`
    // to pass control to the global error handler.
    const err = Error("Bad request.");
    err.status = 400;
    err.title = "Bad request.";
    err.errors = errors;
    return next(err);
  }

  // Invoke the next middleware function
  next();
};



module.exports = { asyncHandler, handleValidationErrors };
```

In `routes/task.js`, be sure to remove the original `asyncHandler` and `handleValidationErros` function definitions, and instead import them now from `utils.js`:

```
//- ./routes/tasks.js

const express = require("express");
const db = require("../db/models");
const { check } = require("express-validator");
const { asyncHandler, handleValidationErrors } = require("../utils");

// REST OF FILE NOT SHOWN
```

Now that you're done with the refactor, let's build out the users router!

To start, create a `routes/users.js` file to hold your users router. Go ahead and declare an express router and export it:

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

module.exports = router;
```

Then, add this users router in your `app.js` file:

```
// only new code shown:

const usersRouter = require("./routes/users");

app.use("/users", usersRouter);
```

As a refresher, now all requests to a path that start with '/users' will be routed to the `usersRouter`.

Back in the users router, let's set up a route for creating users. There are a couple of things you'll need to set up. First, set up a router to handle POST requests to http://localhost:8080/users. Since this route handler will interact asynchronously with the database, you'll need to wrap the route handler function in the `asyncHandler` function:

```
const { asyncHandler } = require("../utils");

router.post("/", asyncHandler((req, res) => {
// TODO: implement creation of user
});

// REST OF FILE NOT SHOWN
```

Then, you'll also need to add some route-level validations to ensure that the client is passing in a valid email and password. Just like you did in the the tasks router, go ahead and use the `express-validator` library's `check` function to define a series of middleware functions that will check the `email` and `password` params.

Go ahead and define these validation middleware functions inside of a `validateEmailAndPassword` array. In a later section, you'll be implementing a "log in" route that will require the same `email` and `password` validations, so this array of validations can be reused for that route!

Here are a few things you should check for:

1. check that `email` is a truthy value (i.e. not `undefined`, `null`, or an empty string)
2. check that `email` is a valid email.
3. check that `password` is a truthy value.

Feel free to add more validations, like ensuring that the password has a digit in it, but make sure that you at least have the following validations:

```
const validateEmailAndPassword = [
  check("email")
    .exists({ checkFalsy: true })
    .isEmail()
    .withMessage("Please provide a valid email."),
  check("password")
    .exists({ checkFalsy: true })
    .withMessage("Please provide a password."),
];
```

Back in the tasks router, you used the `handleValidationErrors` function to handle any validation errors that `express-validator` found. Import that `handleValidationErrors` function from the `utils.js` file now so that you can also handle validation errors in the users router. You have a couple of options here on how to set this up. You can either pass in both `validateEmailAndPassword` and `handleValidationErrors` as middleware functions in `router.post("/")` (like you did in the tasks router), or you could simply add `handleValidationErrors` as the last entry in `validateEmailAndPassword`:

```
const validateEmailAndPassword = [
  check("email")
    .exists({ checkFalsy: true })
    .isEmail()
    .withMessage("Please provide a valid email."),
  check("password")
    .exists({ checkFalsy: true })
    .withMessage("Please provide a password."),
  handleValidationErrors,
];
```

Regardless of which option you choose, go ahead and add those validations to `router.post("/")`:

```
router.post(
  "/",
  validateEmailAndPassword,
  asyncHandler(async (req, res) => {
    // TODO: handle user creation
  })
);
```

Now that validations are taken care of, let's actually implement the user creation!

The first part of creating a user will be fairly similar to what you were doing back in the session-based authentication lesson. You want to hash the password so that you can store it in the `hashedPassword` column, and you'll be using the bcryptjs library again to do the hashing. Go ahead and run `npm install bcryptjs`.

Then, use the library to hash your password before using the `User` model to create your new user:

```
const bcrypt = require("bcryptjs");
const db = require("../db/models");

const { User } = db;

// REST OF FILE IN BETWEEN NOT SHOWN

router.post(
  "/",
  validateEmailAndPassword,
  asyncHandler(async (req, res) => {
    const { email, password } = req.body;
    const hashedPassword = await bcrypt.hash(password, 10);

    const user = await User.create({ email, hashedPassword });

    // TODO: implement rest of route handler
  })
);
```

At this point, you've set up a route that creates a user. The next section will cover how to generate an access token for the user so that the user can be "logged in" for subsequent requests.

Generate an access token for the client

Let's finish up the rest of the route handler for `router.post("/")` in the users router!

Back in the previous lesson when you were implementing session-based authentication, you "logged in" a user by storing the new user's id in session and then storing the session id as a cookie on the client side.

With token-based authentication, you'll now instead generate an access token that holds identifying information in its payload, such as user id and email. Then, you'll return this token to the client, and the client will include this token in all subsequent requests.

The access token will be a JWT, and you'll use the jsonwebtoken package to handle JWT generation and decoding. Go ahead and run `npm install jsonwebtoken`.

Here's the game plan for generating this token:

1. First, let's add all of the components that are necessary for generating a JWT.
2. Then, let's set up a new `auth.js` file that will have a set of utility functions that handles authentication-related logic.

Let's start with adding all the components that are necessary for generating a JWT!

As a refresher, a JWT has three parts: the header, the payload, and the signature.

The header describes the algorithm that will be used for hashing. By default the jsonwebtoken library uses the `HS256` algorithm by default.

The payload holds identifying information about the user. You'll be storing the user id and email in the payload. There's one more piece that's necessary in the payload. Specifically, you probably don't want a JWT to be valid forever, as this would increase the risk of your application's security being compromised.

Instead, you want to expire these tokens after a certain amount of time. To do this, you can pass an `expiresIn` option to the jsonwebtoken token generation method. That method will take care of then including a field in the payload that notates how long the token is valid for. Then, when the JWT is later being decoded, that expiration timestamp can be checked to determine whether or not the token is still valid.

Finally, the third part of the JWT is the signature. The signature is the output of hashing the header, the payload, and the secret key, so you'll need to set up a secret key.

To recap, there are two components you need to set up: a secret key and the duration that a JWT should be valid for.

Let's store both of those components in environment variables.

To do this, first generate a secret key by opening up a node repl in your terminal (run `node`). In the node repl, use the built-in `crypto` module to generate a random string by running the following:

```
require("crypto").randomBytes(32).toString("hex");
```

Then add the following fields to your `.env`:

```
JWT_SECRET=<<YOUR GENERATED RANDOM STRING>>
JWT_EXPIRES_IN=604800
```

"604800" was set as the `JWT_EXPIRES_IN` value because there are 604800 seconds in a week.

Next, set up your 'config/index.js' so that those newly added environment variables are accessible in your app:

```
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,
  },
  jwtConfig: {
    secret: process.env.JWT_SECRET,
    expiresIn: process.env.JWT_EXPIRES_IN,
  },
};
```

Finally, let's create an `auth.js` file and declare a `getUserToken` function that will generate an access token for the user. In this function, take in the user object as a parameter. Then, use the `jsonwebtoken` library to generate a JWT:

```
const jwt = require("jsonwebtoken");
const { jwtConfig } = require("./config");

const { secret, expiresIn } = jwtConfig;

const getUserToken = (user) => {
  // Don't store the user's hashed password
  // in the token data.
  const userDataForToken = {
    id: user.id,
    email: user.email,
  };

  // Create the token.
  const token = jwt.sign(
    { data: userDataForToken },
    secret,
    { expiresIn: parseInt(expiresIn, 10) } // 604,800 seconds = 1 week
  );

  return token;
};

module.exports = { getUserToken };
```

To recap, in the code snippet above, the `jwt.sign` method's first argument is the payload. The second argument is the secret key used to sign the JWT, and the third argument is an `options` object that you can use to customize the JWT. Notice how we also had to convert `expiresIn` from a string to an integer. This is because environment variables declared in the `.env` file are strings, and according to the jsonwebtoken docs, anything that gets set to the `expiresIn` as an **integer** will be calculated as **seconds**, and **strings** will be calculated in **milliseconds**.

Let's go back to the users router to finish generating an access token for the client. In the `router.post("/")` route handler, after the user has been created, use the `getUserToken` function to generate the JWT. Then, set it in the body of the response:

```
const { getUserToken } = require("../auth");

// REST OF FILE NOT SHOWN

router.post(
  "/",
  validateEmailAndPassword,
  asyncHandler(async (req, res) => {
    const { email, password } = req.body;
    const hashedPassword = await bcrypt.hash(password, 10);
    const user = await User.create({ email, hashedPassword });

    const token = getUserToken(user);
    res.status(201).json({
      user: { id: user.id },
      token,
    });
  })
);
```

Finally, go to Postman and make a POST request to `http://localhost:8080/users` with the correct params in order to create a user. Verify that you are able to get the `token` in the response body.

Using the access token

Let's put the access token to use by protecting the tasks resource. In order to protect the tasks routes, let's set it up so that the request must contain a valid, unexpired JWT that identifies a signed up user. This means that the client will now be required to include a valid access token in the header of the request if they want to access the tasks routes.

In the server, you'll need to then set up a couple of functions that will parse the access token from the header of the request, verify that it is a valid JWT, and then find the user based on the identifying data, such as the user id and email, in the payload.

Let's go to the `auth.js` file to set this up! First, in order to parse the access token from the request header, install the express-bearer-token npm package by running `npm install express-bearer-token`. This package provides a middleware function that will automatically look in the header under the `Authorization` key for a token, parse it out, and then set the token as a field in the `req` object. Specifically, it expects for the header to be formatted like: `Authorization: Bearer <token>`, which is a standard format for setting access tokens in the request header.

Next, now that the token has been parsed out, you need to set up a middleware function that will verify the validity of the JWT. Once it's been verified, you can take the decoded payload and then use the identifying information to search for the user in the database. Once a user has been found, set it as a field in the `req` object so that subsequent middleware functions have access to it. If any steps fail along the way, then throw an error and return a 401 Unauthorized status. This is what that function should look like:

```
const restoreUser = (req, res, next) => {
  // token being parsed from request header by the bearerToken middleware
  // function in app.js:
  const { token } = req;

  if (!token) {
    const err = new Error("Unauthorized");
    err.status = 401;
    return next(err);
  }

  return jwt.verify(token, secret, null, async (err, jwtPayload) => {
    if (err) {
      err.status = 401;
      return next(err);
    }

    const { id } = jwtPayload.data;

    try {
      req.user = await User.findByPk(id);
    } catch (e) {
      e.status = 401;
      return next(e);
    }

    if (!req.user) {
      // Send a "401 Unauthorized" response status code
      // along with an "WWW-Authenticate" header value of "Bearer".
      return res.set("WWW-Authenticate", "Bearer").status(401).end();
    }

    return next();
  });
};
```

Make sure to use import the `User` model at the top of the file as well.

Now that you've got both the express-bearer-token middleware function and the `restoreUser` function, go ahead and export both of those functions together in an array called `requireAuth`. This array of middleware functions can then be passed in to any routes or routers that you want to protect. This is what that should look like:

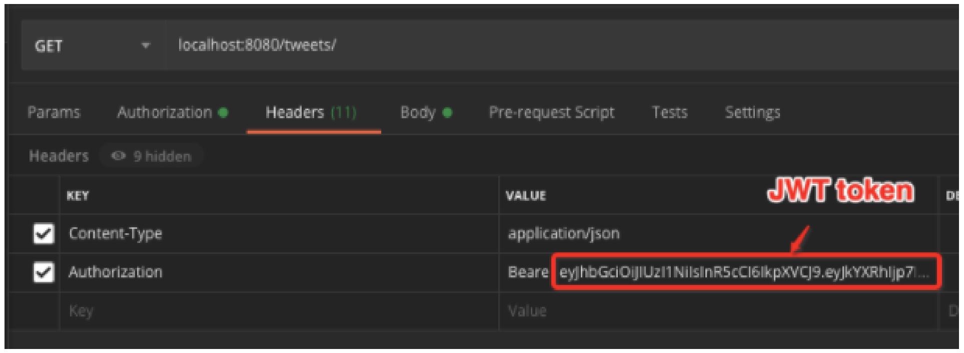
```
const bearerToken = require("express-bearer-token");

// OTHER CODE IN THE FILE NOT SHOWN

const requireAuth = [bearerToken(), restoreUser];

module.exports = { getUserToken, requireAuth };
```

Finally, let's put this array of middleware functions to use by going to the tasks router and adding it as a middleware function (before any of the route definitions) for all routes in this router:

```
const { requireAuth } = require("../auth");

// REST OF FILE NOT SHOWN

router.use(requireAuth);

// ROUTE DEFINITIONS NOT SHOWN
```

At this point, go ahead and go to Postman to try to make a request for all tasks (`GET http://localhost:8080/tasks`). This should fail with a 401 response.

To fix this, use the `POST http://localhost:8080/users` endpoint to create another user. This time, take the `token` that's in the response body. Use that `token` value in the `Authorization` header:



GET localhost:8080/tweets/

	KEY	VALUE
✓	Content-Type	application/json
✓	Authorization	Beare eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjp7I...
	Key	Value

Then, make another request for all tasks, and this time, since the client is authenticated, getting all of the tasks works!

Where to store the access token on the client-side

In the project that you'll be working on today, you'll be building out a frontend app that allows users to sign up and log in, and the response to these two types of requests would include the access token.

Once the client receives this access token, it needs to store it somewhere so that it can be included with all subsequent requests. In the previous lesson on session-based auth, the session id was stored in a cookie.

One popular place to store access tokens nowadays is by using the Web Storage API. The Web Storage API, implemented by browsers, exposes two place to store data on the client side: `localStorage` and `sessionStorage`.

> `localStorage` stores data without an expiration date. `sessionStorage` stores data until the browser or the tab is closed.

In choosing between cookies and something like `localStorage`, the upside of `localStorage` is that it's very simple to use. With `cookies` there are very specific configurations that must be set on both the server and the client in order to allow for it to work in a cross-domain setting.

On the other hand, data stored in `localStorage` is retrievable by client-side JavaScript, so if your application does not take the necessary measures against Cross-Site Scripting (XSS), then the access tokens stored in `localStorage` could be stolen. You'll learn more about XSS in a later lesson.

For tomorrow's project, you'll be using `localStorage` to store the access token, and again, it is a fairly popular way to store access tokens. At the same time, it's definitely useful to know what options are available and the pros and cons of each option.

What you've learned

In this lesson, you learned:

1. What a JSON Web Token (JWT) is and how it can be used to securely send user information between servers or across requests.
2. How to add token-based (session-less) authentication to an Express API.
3. How to use the jsonwebtoken npm package to sign and verify JWTs.