- ⏱️ 21 minutes
- ✅ Completed

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.

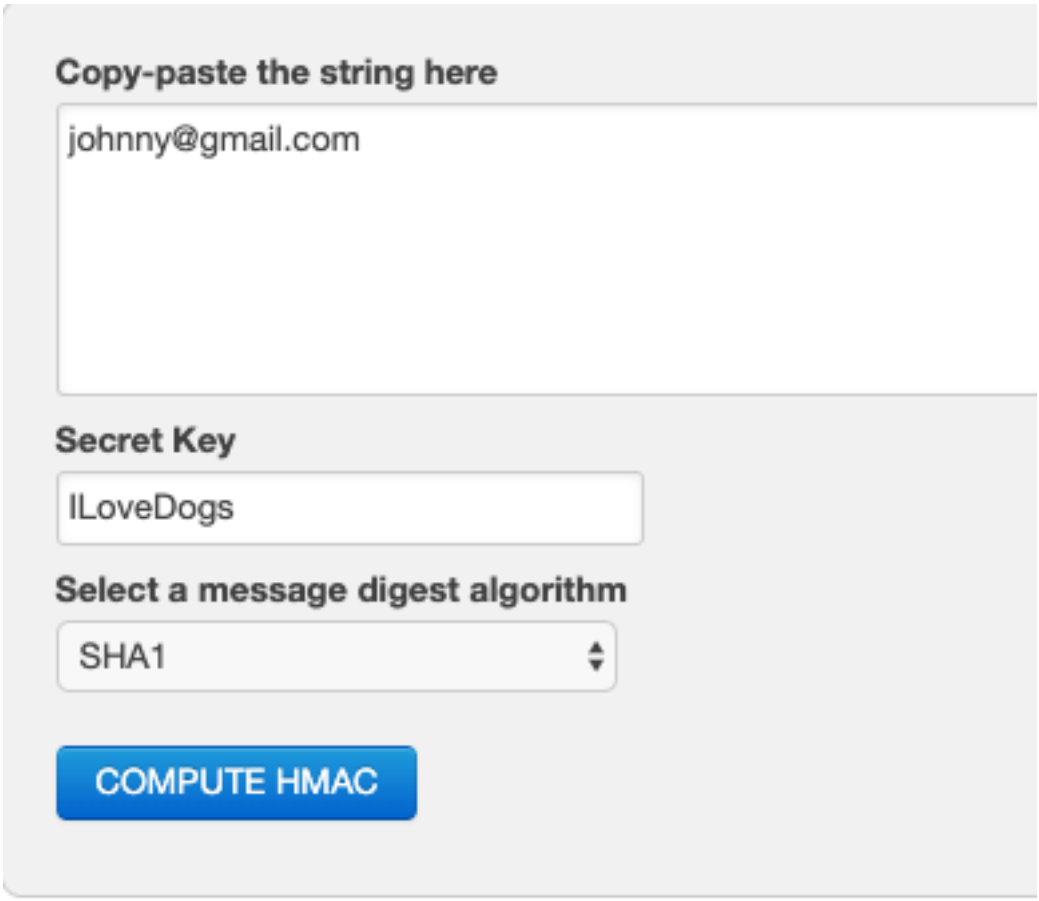
You prepare to start sending out invites, and you need to figure out a way to make sure only your invited guests are allowed into the party. One way would be to just keep a guest list, but you don't want to have to maintain a guest list.

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`:



You then send an email to Johnny:

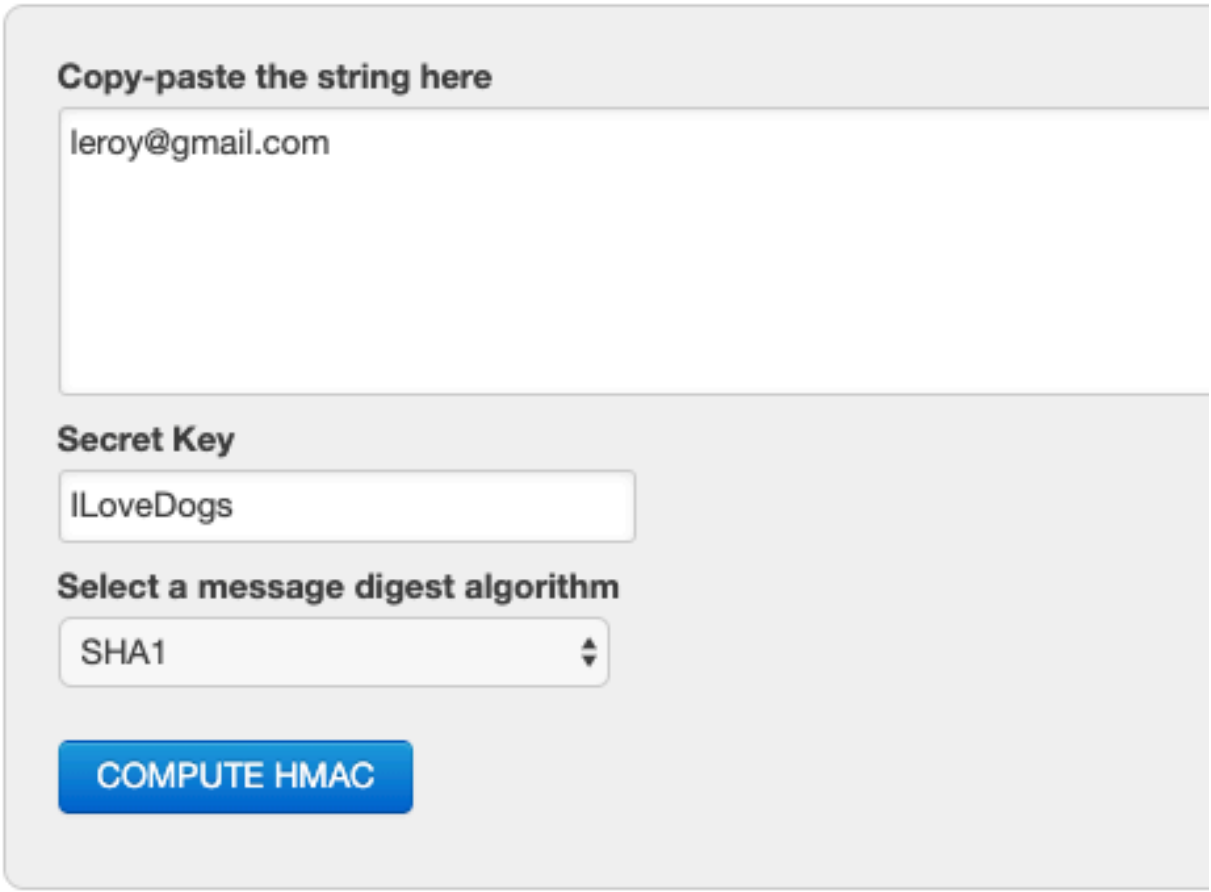
```
Dear Johnny,

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:



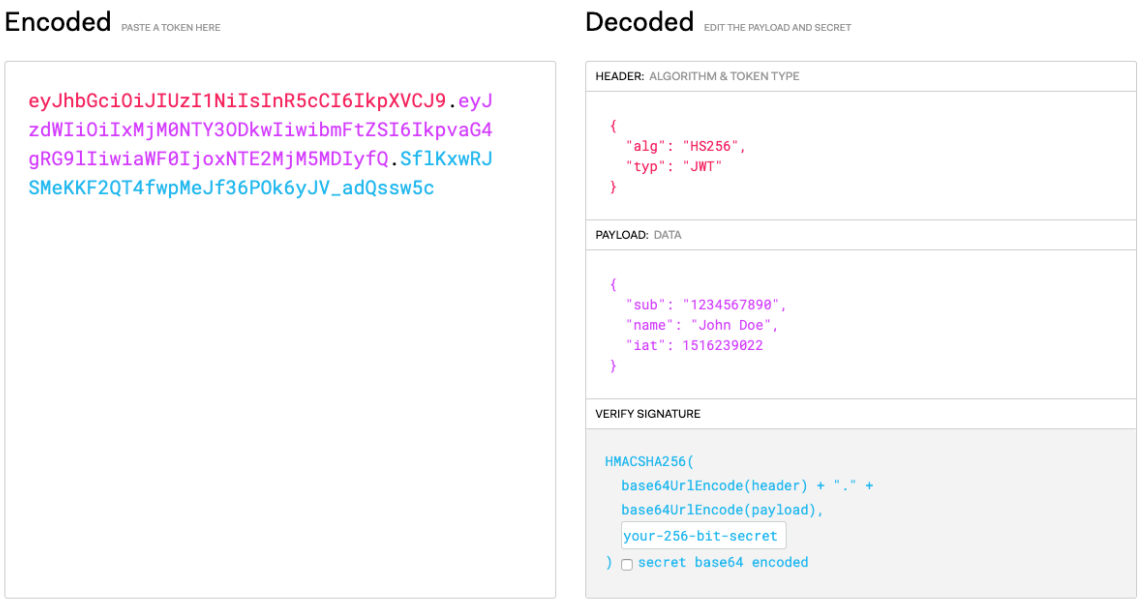
Immediately, you know that Leroy has an invalid invite because the string digest output of the hashing function (`9ca5168290df6b05951af368d668ada58a56b12a`) was different than the one he presented.

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.



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:string`
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.
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:

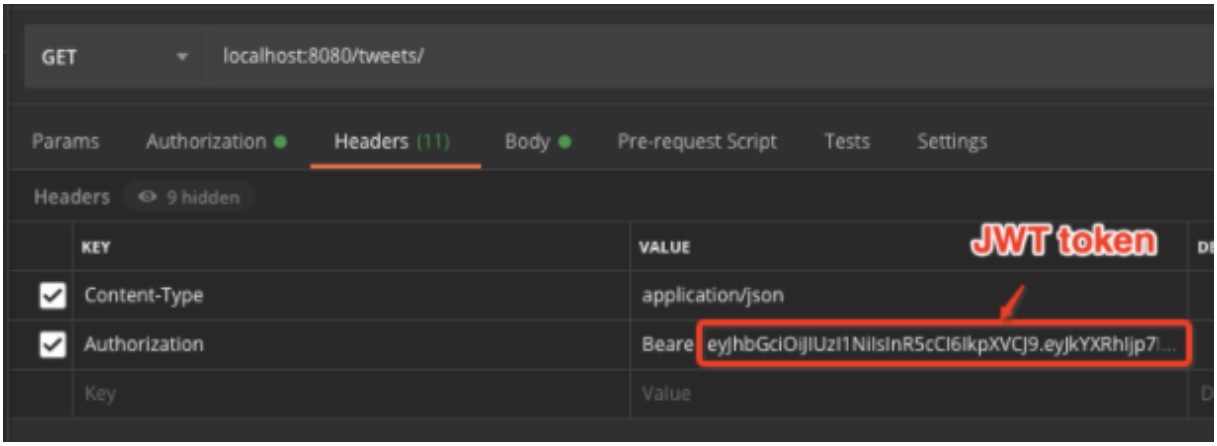
```
const { requireAuth } = require("../auth");

// 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:



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.