API Authentication and Authorization

- API security is an important topic, and there are many different ways in which an API could and should be secured.

- ***Authentication*** and ***authorization*** play a central role in API security.

- Broadly speaking, authentication is the process of verifying someone's identity. For example, entering your username and password on a website is a way of authenticating yourself to that website, i.e. verifying that you are yourself.

- Authorization, on the other hand, is the process of verifying that someone has the permission to perform some action or to access some resource. For example, one user may have permission to create content in a web application, while another user may only have permission to view content.

- In an API, authorization often goes hand-in-hand with authentication. In other words, a user might have to authenticate themself before being authorized to access a certain resource. However, even anonymous (i.e. unauthenticated) users may be authorized to perform certain actions or access certain resources.

- Here, we'll explore how to incorporate authentication and authorization into our API. We'll start with authentication.

Disclaimer

Before we move on to talk about authentication, please read this disclaimer:

> *All of the authentication schemes we'll discuss here involve clients sending users' passwords to the API server within a request.* ***In any kind of meaningful application, such requests should always be encrypted (i.e. sent using HTTPS) to prevent attackers from accessing user passwords in cleartext.*** *Using HTTPS is beyond the scope of this course, so we won't discuss it in any depth, but please be aware of its importance in your future API work.*

Authentication schemes

- There are various kinds of authentication schemes that can be used in conjunction with an API. The most commonly used authentication schemes involve a user making a claim about their identity (e.g. by specifying their username or email address) and providing a secret password, known only to the user, as a way to verify that identity.

- In the simplest such authentication scheme, known as ***HTTP Basic authentication***, involves the client sending the user's username and password in the headers of every API request that requires authentication.

- This scheme is easy to implement on the server side: each API endpoint that requires authentication must simply verify whether each request contains valid authentication credentials.

- However, because HTTP Basic authentication requires the username and password to be sent with each authenticated request, the client must either ask the user for their username and password each time a request is made or else store them on the client side. The former approach is inconvenient and annoying for the user, while the latter approach is a potential security risk.

- ***Token-based authentication*** avoids these shortcomings of HTTP Basic authentication. Under a token-based authentication scheme, the client "logs in" to the server a single time with the user's credentials. Upon receiving valid credentials, the server sends the client a token that uniquely identifies the user. The client can attach this token to every API request that requires authentication.

- Token-based authentication has several advantages over HTTP Basic authentication. Most importantly, token-based authentication does not require storing the user's credentials on the client side. The client instead stores the token. This is a much lower security risk because tokens can be much more easily revoked than a user's password, and because tokens are typically assigned an expiry time, after which they are no longer valid, thus mitigating the amount of damage that can be done with a compromised token.

- There are different forms of token-based authentication. ***Session-based authentication*** is one of these.

- In session-based authentication, the server stores a record of each token it issues (e.g. in a database). This record includes information about the user associated with the token. Each time an authenticated request is made, the server checks its records to see if the token provided by the client is valid and unexpired. If so, the request is successfully authenticated.

- One of the main drawbacks of session-based authentication is that it places a burden on the API server to store records of all valid sessions.

- Several stateless token-based authentication approaches have been developed to eliminate the need for the server to store session records. In these approaches, all of the “session” information that might otherwise need to be stored in a database (e.g. the user’s identity) is encoded within the token itself. The token is then encrypted or cryptographically signed to ensure its authenticity.

- One of the most popular of these stateless approaches is based on ***JSON Web Tokens*** (or ***JWTs***, the singular of which is sometimes pronounced as “jot”).

JSON Web Tokens

- A JSON Web Token (JWT) is a JSON-based object that is encoded and cryptographically-signed to form a token. JWTs are designed to be both compact and URL-safe.

- A JWT is a string that typically consists of three parts: a header, a payload, and a signature. These are separated by dots, e.g.: `aaaaa.bbbbb.ccccc`.

- The JWT header is a JSON-encoded object that is Base64Url encoded to produce a compact string. The header object specifies information about the token itself and usually contains two properties indicating the type of the token (JWT) and the algorithm used to sign the token (e.g. HMAC-SHA256), e.g.:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

- After Base64Url encoding, this header object becomes the following string:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

- The JWT payload is also a JSON-encoded object that is Bas64Url encoded. The payload object contains a set of claims, which are typically statements about the user, along with additional metadata.

- There are several standardized properties that can be included in the payload object, such as sub (subject, e.g. the ID of the user), iat (issued at, the time at which the JWT was issued), exp (expiration time, the time at which the JWT expires). However, the JWT payload object can contain any custom properties that are agreed upon by everyone who will consume the token. An example JWT payload object might look like this:

```
{
  "sub": "5a1f3e8864be89eef867596a",
  "name": "Luke Skywalker",
  "admin": false
}
```

- The resulting Base64Url-encoded string for this object is:

```
eyJzdWIiOiI1YTFmM2U4ODY0YmU4OWVlZjg2NzU5NmEiLCJuYW1lIjoiTHV
rZSBTa3l3YWxrZXIiLCJhZG1pbiI6ZmFsc2V9
```

- Finally, the JWT signature is computed by concatenating the Base64Url-encoded header and payload strings with a dot (`.`) and then using a secret key to sign them with a cryptographic signing algorithm like HMAC-SHA256, i.e.:

```
HMACSHA256(
  base64UrlEncode(header) + "." + base64UrlEncode(payload),
  secretKey
)
```

- Because this is a cryptographic signature of the contents of the token, it can be used to verify that the contents of the token weren't modified. In addition, if the secret key used in the signature is the private key of a public-private key pair, the signature can also be used to verify the signer's identity.

- After computing the signature from the header and payload above using the HMAC-SHA256 algorithm and the secret key `hunter2`, the final JWT is:

  ```
  eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI1YTFmM2U4OD
  Y0YmU4OWVlZjg2NzU5NmEiLCJuYW1lIjoiTHVrZSBTa3l3YWxrZXIiLCJhZ
  G1pbiI6ZmFsc2V9.9xSTB-9pUyzdIGPXNiWY0Sh6xcOx-XBUriLtGPOZMmE
  ```

- ***Importantly, the header and payload portions of a JWT are not encrypted, only simply Base64Url encoded, so you should never place sensitive information into a JWT.***

- To demonstrate this, copy either the header portion (before the first dot) or the payload portion (between the dots) of the JWT above and paste it into the following decoder, and you should be able to retrieve the original JSON: http://www.simplycalc.com/base64-decode.php.

- It is simply not the intent of JWTs to be used to transmit sensitive information, and they are becoming increasingly widely-used in token-based authentication. This is the authentication scheme we will focus on in this course.

Registering users and storing their passwords

- Before we start discussing how to incorporate JWTs into our application, we'll need to be able to register users and store their passwords. We must take special care when doing this, in order to protect the user's password.

- As a starting point, we'll use the code for the Book-a-Place API, which we wrote previously (note that the URL here refers to a specific commit):

  https://github.com/OSU-CS493-Sp18/auth/tree/2187ef10ac731a2fcd292d0cd665db13028ba197

- Our primary focus for now will be the `insertNewUser()` function that we already wrote. This function takes a `user` object, passed directly from the body of a request to our POST `/users` endpoint, and inserts this user as a document in the `users` collection of our MongoDB database.

- In addition to the fields we previously expected to be contained in the `user` object (`userID`, `name`, and `email`), we'll now require that it contains a `password` field. This field will contain the user's *plaintext* password.
    - Again, remember the disclaimer above. The client will send the user's password in plaintext within the body of the request to our POST `/users` endpoint. It is thus *imperative* that HTTPS is used in a production setting so this request body is encrypted in transit across the network.

- We will store the user's password in the database along with the rest of their information. ***However, it is important that we don't store the user's password in plaintext.*** In particular, if user passwords are stored in plaintext, an attacker that gained access to our database could easily know the passwords of all of our users.

- It is common practice to ***hash*** the user's password, i.e. to pass it through a cryptographic hashing function, and to store the hashed value in the database. Then, when the user attempts to log in, the password they provide is hashed and compared against the stored hashed password. If the values, match, the user's identity is verified.

- In addition to simply hashing the user's password, it is also typically ***salted***. This means that a unique, random string is generated and concatenated to the user's password before it is hashed. This process is specifically intended to protect against rainbow table attacks.

- We will both salt and hash passwords before storing them in our database. To do this, we will use a Node.js package called `bcrypt.js`, which implements both salting and hashing based on the cryptographic cipher Blowfish. We'll start by installing this as a dependency with npm:

```
npm install --save bcryptjs
```

- Then, we can import the `bcryptjs` module:

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

- Let's now turn our attention to the `insertNewUser()` function. Remember, here, the user's password will be specified in a `password` field of the `user` object. We'll use `bcrypt.js` to salt and hash it before storing it in the database.

- We'll specifically use the `bcrypt.js` package's `hash()` function, which both salts and hashes the password. This is an asynchronous function, which takes as arguments the password and the length of the salt to generate and returns a promise that resolves to a string that contains both the hashed password and the generated salt. We'll return this promise, after chaining it with the remainder of the functionality of the `insertNewUser()` function:

```
return bcrypt.hash(user.password, 8)
  .then((passwordHash) => {...})
```

- Inside the `.then()` clause here, we'll move the code we had written previously to generate the user document and insert that document into the database. We'll add the hashed password to the user document before inserting it:

```
const userDocument = {
  ...
  password: passwordHash
};
const usersCollection = mongoDB.collection('users');
return usersCollection.insertOne(userDocument);
```

- We had previously attached a `.then()` clause to our call to `insertOne()` here to return the ID of the just-inserted user document. We'll now simply chain this `.then()` clause to the one we just wrote:

```
.then((result) => {
  return Promise.resolve(result.insertedId);
});
```

- Now, users will be inserted into the database with their hashed password. One last detail we must worry about: our GET `/users/{userID}` endpoint simply returns the entire user document from the database. Now that document will contain the user's hashed password.

- It is a security risk to expose even a hashed password, so we'll want to make sure not to include this in our responses to the GET `/users/{userID}` endpoint. We can do this by specifying a ***projection*** to explicitly exclude the `password` field from the results of our MongoDB query in `getUserByID()`.

- As we'll see later, we'll sometimes want to include the user's password from this function, so we'll add a function parameter to explicitly tell the function to include the password when needed:

```
function getUserByID(userID, includePassword) {
  ...
}
```

- Then, inside the function, we'll generate a projection document that explicitly excludes the user's password if `includePassword` is truthy:

```
const projection = includePassword ? {} : { password: 0 };
```

- Then, our MongoDB query will look like this:

```
usersCollection
  .find(query)
  .project(projection)
  .toArray()
```

Validating passwords to log in users

- Now that we're successfully storing hashed user passwords, we can implement an API endpoint to allow users to log in. We'll implement this as a POST endpoint, to which clients will send a user's ID and password in the request body:

```
app.post('/users/login', function (req, res) {...});
```

- Within this endpoint, we need to make sure the request body contains the user's ID and password. If it doesn't we'll respond with a 400 error:

```
if (req.body && req.body.userID && req.body.password) {
  ...
} else {
  res.status(400).json({
    error: "Request body needs user ID and password."
  });
}
```

- If the request body is properly formatted, we'll want first to fetch the user entry (with password) corresponding to the user ID specified in the request body. We can use our `getUserByID()` function for this:

  ```
  getUserByID(req.body.userID, true)
    .then((user) => {...})
    .catch((err) => {...});
  ```

- Within the `.then()` clause here, we need to account for the possibility that the specified user ID is invalid, in which case `user` would be null. It is best practice to return the same error response regardless of whether the specified user ID or password is invalid, thereby making brute-force attacks more difficult.

- As we'll see in a second, we'll also add a second `.then()` clause to our promise chain here, so if the user ID is invalid and `user` is null, we'll simply return a rejected promise wrapped around an error code indicating that we'll want to respond with a 401 (unauthorized) error:

  ```
  if (user) {
    ...
  } else {
    return Promise.reject(401);
  }
  ```

- If the specified user ID is valid, and `user` is not null, we'll want to verify the provided password against the one we have stored in the database for the user. We'll use the `bcrypt.js` package's `compare()` function to do this. The `compare()` function takes as arguments a plaintext password string and a password hash string computed by `bcrypt.hash()`. It salts and hashes the plaintext password, compares it to the provided hash, and returns a promise that resolves to true if the two values are equal or false otherwise. We'll simply return this promise here:

  ```
  return bcrypt.compare(req.body.password, user.password);
  ```

- We'll attach another `.then()` clause to our promise chain to handle the results of this comparison:

  ```
  .then((loginSuccessful) => {...})
  ```

- If the login attempt was successful (the specified username and password were valid), we'll want to return a success response to the client. Eventually we'll put a JWT in this response, but for now, we'll simply send an empty response. If the login attempt was unsuccessful, we'll again return a rejected promise indicating that we want to reply with a 401 error:

```
if (loginSuccessful) {
  res.status(200).json({});
} else {
  return Promise.reject(401);
}
```

- We can now finally implement the `.catch()` clause in our promise chain. Here, we'll want to differentiate between server errors (e.g. problems connecting with the database) and authentication errors. If at any point in our promise chain, a promise was rejected with an error code of 401, we'll return a response with a 401 status. Otherwise, we'll return a 500 error:

```
if (err === 401) {
  res.status(401).json({
    error: "Invalid user ID and/or password."
  });
} else {
  res.status(500).json({
    error: "Unable to verify credentials. Try again later."
  });
}
```

- We should now be able to see our login endpoint working. If we send a request to this endpoint containing valid credentials, we should see a 200-status response. Any request with invalid credentials should result in a 401 error. Our last step will be to return a JWT along with the success response.

Generating JWTs

- We want now to start generating JWTs and using them for authentication. To do this, we'll use the popular `jsonwebtoken` package, which includes functionality to both generate and verify JTWs:

```
npm install --save jsonwebtoken
```

- We'll do JWT-related things in their own module, so we can keep all of our auth-related code in a single place, and so we can easily reuse our auth functions across our codebase. Let's create this file, calling it `lib/auth.js`.

- We'll start this file by including the `jsonwebtoken` package:

```
const jwt = require('jsonwebtoken');
```

- We'll also create a secret key, which we'll use to sign and verify JWTs. For simplicity, we'll use a constant, hard-coded string for the secret key, though this is obviously bad practice. There are many ways you could obtain a secret key in a real API, including reading a key from an environment variable or file or randomly generating one, e.g. using the Node.js crypto module. Anyway, here's our key:

```
const secretKey = 'SuperSecret';
```

- With these initial steps out of the way, let's write a function to use the `jsonwebtoken` package to generate a JWT. We'll export this function from our module. Within the payload of our JWT, we'll simply store the ID of the user we're authenticating. We'll take this as a function parameter:

```
function generateAuthToken(userID) {
  ...
}

exports.generateAuthToken = generateAuthToken;
```

- The `jsonwebtoken` package has a function called `sign()`, which we can use to generate a JWT from a given payload, signed with a specific secret key. This is an asynchronous function, and we'll wrap it with a promise, which we'll return:

```
return new Promise((resolve, reject) => {...});
```

- Next, we'll generate our JWT payload, which will be an object containing the provided user ID under the `sub` (subject) field:

```
const payload = { sub: userID };
```

- Now, we can call `jwt.sign()`. In addition to our secret key and payload, we'll also pass the `expiresIn` option, which will add an `exp` field to the JWT's payload based on a time span we can specify as a string. In the callback we pass to `jwt.sign()`, we'll simply reject or resolve the promise:

```
jwt.sign(
  payload,
  secret,
  { expiresIn: '24h' },
  function (err, token) {
    if (err) {
      reject(err);
    } else {
      resolve(token);
    }
  }
);
```

  - Note that here we're allowing `jwt.sign()` to use the default signing algorithm, HMAC-SHA256.

- Now, we can move back to our POST `/users/login` endpoint. Here, in the last `.then()` clause in our promise chain, we'll call `generateAuthToken()` (which we need to make sure to import) instead of returning an empty response to the client:

```
if (loginSuccessful) {
  return generateAuthToken(req.body.userID);
} ...
```

- And we can attach an additional .then() clause to take the generated JWT and return it to the client:

```
.then((token) => {
  res.status(200).json({ token: token });
})
```

- Now, if we send a valid authentication request to our POST `/users/login` endpoint, we'll get a valid JWT back. If you copy the returned JWT and paste it into the debugger at https://jwt.io, you should see your original payload.

Using JWTs for authorization

- Now that we can allow users to login and send them a JWT as proof of authentication, how can we authorize our API's endpoints by using JWTs?

- Let's go back to our `lib/auth.js` file and begin a new Express middleware function called `requireAuthentication()`, which we'll also export:

```
function requireAuthentication(req, res, next) {
  ...
}

exports.requireAuthentication = requireAuthentication;
```

- We'll eventually plug this middleware function into our Express application to make sure the request contains a valid JWT. If the request *does* contain a valid JWT, we'll set `req.user` to the ID of the logged-in user, so we can use this value in later middleware functions. Otherwise, we'll return a 401 response to the client, to indicate that the user is not authorized.

- It is best practice for clients to send a JWT token within the headers of the request, typically within an `Authorization` header using the `Bearer` schema:

```
Authorization: Bearer <token>
```

- In an Express middleware function, we can get a request header from the request object using `req.get()`. We'll use this to grab the `Authorization` header value, make sure this value is non-null, and then split it at the space to separate the `Bearer` scheme string from the actual token:

```
const authHeader = req.get('Authorization') || '';
const authHeaderParts = authHeader.split(' ');
```

- Next, we'll verify that the header does in fact specify the `Bearer` scheme, and then we'll grab the token, setting the token to null if the header doesn't specify the `Bearer` scheme:

  ```
  const token = authHeaderParts[0] === 'Bearer' ?
    authHeaderParts[1] : null;
  ```

- Once we have the token value (even if it's null), we can use the `jwt.verify()` function to verify the token. This function takes our secret key and asynchronously uses it to compute a signature over the header and payload of the JWT, just like when it was originally signed. If the computed signature matches the one on the token, the token is validated.

- The `jwt.verify()` function also checks things like the expiration time of the token. If the token is expired, then it is not valid.

- If the token passes all of `jwt.verify()`'s checks, the function calls a provided callback and passes it the decoded payload of the token. If the token cannot be verified, an error is passed to the callback.

- Our call to `jwt.verify()` will look like this:

  ```
  jwt.verify(token, 'SuperSecret', function (err, payload) {
    ...
  });
  ```

- Inside the callback function here, if the token was successfully validated (i.e. `err` is null), we'll set `req.user` to the user's ID, which will be available in the `sub` field of the decoded JWT payload, and we'll call the next middleware function. If there was an error, we'll return a 401 (unauthorized) response:

  ```
  if (!err) {
    req.user = payload.sub;
    next();
  } else {
    res.status(401).json({
      error: "Invalid authentication token provided."
    });
  }
  ```

- This completes our authorization function, but how do we use it? If we wanted to require authentication for our entire API, we could plug `requireAuthentication()` into our app to run on every request:

  ```
  app.use(requireAuthentication);
  ```

- However, this is probably not the best approach. At a minimum, we want users to be able to use our `/users/login` endpoint without already being logged in!

- Luckily, we can be more precise about where we require authentication using a feature of the Express framework, where we can specify a *sequence* of middleware functions to be called, in order for any given route specification.

- For example, if we had two functions, `fn1` and `fn2`, that we wanted to be executed in order on HTTP GET requests to the path `/`, we could specify the route like this:

  ```
  app.get('/', fn1, fn2);
  ```

- Here, when a GET request is made to the path `/`, `fn1` will first be executed, then `fn2`.

- We can take advantage of this feature and include our `requireAuthentication()` function in the route specification for any API endpoint for which we want to require authentication, adding it *before* the normal route handler function, to ensure that it is called first.

- For example, if we wanted to require authentication for our API's GET `/users/{userID}` endpoint, we could import `requireAuthentication()` and incorporate it like this:

  ```
  app.get(
    '/users/:userID',
    requireAuthentication,
    function (req, res, next) {...}
  );
  ```

- Not only does this make it simple to require authentication on precisely the endpoint we desire, the syntax makes it very clear which endpoints require authentication.

- Now, if we try to access our GET `/users/{userID}` endpoint, we will receive an error unless we provide a valid authentication token.

- We can go further though. What if we want to further restrict the GET `/users/{userID}` endpoint so that only the user specified by `userID` is authorized to access their own data?

- Recall that we implemented our `requireAuthorization()` function to set `req.user` to the ID of the logged-in user. Within any authenticated endpoint, we can use this value to limit authorization.

- For example, in our GET `/users/{userID}` endpoint, we could include the following check to restrict authorization to the specified user:

  ```
  if (req.user !== req.params.userID) {
    res.status(403).json({
      error: "Unauthorized to access the specified resource"
    });
  } else {
    ...
  }
  ```

- The normal functionality for the endpoint can go within the else clause, and it will only be executed if the logged-in user is authorized.

- Importantly, note that we are returning a 403 (forbidden) response above. In some situations, it may be desirable to instead return a 404 (not found) response in order to hide the existence of the specified resource from unauthorized users.

- Our endpoint is now authenticated and has its authorization restricted. Now, to successfully access the endpoint, we will need to provide a valid authentication token for the user whose data we are requesting.