Authentication Learning Objectives

* Define the term authentication
* Authentication is concerned with verifying someone is who they say they are. We generally provide a username/email and password to the server.
* This is different from **auhorization**, which is determining if a user is allowed to perform a certain action.

1. Describe the difference between asymmetric and symmetric cryptographic algorithms

* Symmetric algorithms use one key for both encoding and decoding.
* Asymmetric algorithms use one key for encoding and another key for decoding. The key used for decoding is kept private so that only the issuer is able to decode.

1. Identify "**strong**" vs. "**broken**" hash functions

* "**Broken**" hash functions have been cracked. The original input to these functions can be determined by the hashed values produced.
  + Examples: md4, md5, sha1
* "**Strong**" hash functions have not been cracked. Given a hashed value, we cannot (at this point) determine what the original input was without some brute force, trial and error calculations.
  + Examples: sha256, sha512
* Slow algorithms hash inputs sequentially many times. They use a hash function to hash an input value, then take the result and hash it again, repeating this process thousands of times just for one input. These hashing algorithms are computationally expensive and difficult to crack even with large computing power, so they are recommended for use in hashing passwords that we want to store in our database.
  + Examples: PBKDF2, bcrypt, Argon2

1. Implement session-based authentication in an Express application

* We can use the express-session package to create session middleware.
* The session will create a cookie that is passed back and forth between the server and client.
* We can set values on our session to indicate that we have been authenticated. On subsequent requests, when we read the cookie we see that we were logged in previously and can see who the user is.
* With each request that comes in, we check if the authorization key has been created on the session previously. If it has been, we find the user's information and add it in to the local response variables in order to be able to use this information in subsequent middleware or routes.

*// app.js*

*// We use the express-session library in order to set up session middleware*

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

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

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

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

*// We pass in the same secret to our cookieParser as we do to our session middleware*

app.use(cookieParser(sessionSecret));

*// Our session middleware sets up a name in order to easily identify the cookie that it creates*

app.use(session({

name: 'amusement-park-tracker.sid',

secret: sessionSecret,

resave: false,

saveUninitialized: false,

}));

*// We use our restoreUser middleware (defined in our auth file below) in order to add the whole user instance to our response's locals key, as well as a flag to indicate we have been authenticated. We can use these values in subsequent routes or middleware functions in order to restrict access, provide customized information, etc.*

app.use(restoreUser);

*// auth.js*

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

*// We signify that a user is logged in by creating an auth key on the request's session.*

*// In this implementation, we are adding the userId in order to signify who is logged in.*

const loginUser = (req, res, user) => {

req.session.auth = {

userId: user.id,

};

};

*// Removing the auth key on our session signifies that we are no longer logged in*

const logoutUser = (req, res) => {

delete req.session.auth;

};

*// We create a middleware function that can be added to routes that we want to restrict to logged in users*

const requireAuth = (req, res, next) => {

if (!res.locals.authenticated) {

return res.redirect('/user/login');

}

return next();

};

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) {

*// If we had an auth key, that means we are logged in in. We pull the userId out of auth, then find the user record associated with it.*

const { userId } = req.session.auth;

try {

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

if (user) {

*// If we successfully found the user, we indicate that we are authenticated and add the user information to the response's locals key for use in other middleware/routes*

res.locals.authenticated = true;

res.locals.user = user;

next();

}

} catch (err) {

*// If we ran into an error finding our user, we indicate we are not authenticated and invoke our error handlers*

res.locals.authenticated = false;

next(err);

}

} else {

*// If we didn't have an auth key at all, we indicate we are not authenticated and continue to the next middleware (or routes)*

res.locals.authenticated = false;

next();

}

};

module.exports = {

loginUser,

logoutUser,

requireAuth,

restoreUser,

};

* In a production environment, we would generally store our session information in a database instead of in local memory (the default MemoryStore). In order for us to do so, we can use another package, such as connect-pg-simple. We have to make sure we are providing environment variables to connect to our database as well as providing a store key on our session options to indicate that we are using something other than the default MemoryStore

*// ./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,

}));

1. Implement a strong hash function to securely store passwords

* In our express applications, using the bcryptjs package is an easy way to hash our passwords and compare passwords for logging a user in.

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;

};

1. Describe and use the different security options for cookies

* When setting up our session, we can provide a cookie key, pointing to an object that specifies options for how our session cookie is set up.
  + httpOnly: prevents JavaScript on the page from accessing the cookie
  + maxAge: sets an expiration for our cookie, which would require the user to re-authenticate
  + path: sets where our cookie is valid. We can make cookies for more specific paths on our app, but the root (/) is most common, available throughout the app
  + secure: requires https to be used
  + domain: if not present, the current domain is used by the cookie, but we can also specify a different domain by including this key
  + expires: where maxAge will calculate an expiration time in the future, expires specifies a specific time to expire at

app.use(

session({

store: new (store(session))(),

secret: 'a5d63fc5-17a5-459c-b3ba-6d81792158fc',

resave: false,

saveUninitialized: false,

cookie: {

httpOnly: true,

maxAge: 60000,

path: '/',

secure: true

}

})

CORS

* Cross-Origin Resource Sharing
* An origin is the combination of protocol, domain, and port
* If we try to access resources from another origin, modern browsers will prevent the request unless we do some configuration. This is for added security so that we only accept resources from trusted sources.

cors NPM Package

* We can add the cors package to our application and use it as middleware for our routes.

const cors = require("cors");

router.get(

"/",

cors(),

asyncHandler(async (req, res) => {

const tasks = await Task.findAll();

res.json({ tasks });

})

);

"Simple" vs "Preflighted" Requests

* Some requests sent by a browser are determined to be "simple" and are immediately sent on to the server. Other requests are "preflighted", which means the browser sends a request with the OPTIONS method to check to see if the server has been configured to accept the request.
* In order for a request to be considered simple, it has to meet certain criteria:
  + The HTTP method has to be either GET, HEAD, or POST
  + It can only use safe-listed request headers:
    - accept
    - accept-language
    - content-language
    - content-type
    - width
    - viewport-width
    - (a few more less common)
  + The content-type can only be application/x-www-form-urlencoded, multipart/form-data, or text/plain

Using CORS for the entire API

* We can tell our whole application to use the cors middleware instead of individual routes. To be safe, we should specify where we are allowing requests to come from

const cors = require("cors");

app.use(cors({ origin: "http://localhost:4000" }));

**## REST**

- Representational State Transfer

**## REST Architectural Constraints**

1. Client-server architecture

  - Request-response cycle between a client (your browser) and a server

2. Stateless (Mostly with us)

  - We can maintain connections with protocols like web sockets, and retain data in cookies that are passed back and forth, but in general our communication is stateless. Each individual request does not know about previous requests.

3. Cache (Not really with us anymore)

  - We used to cache responses on intermediary servers to be able to serve subsequent duplicate requests faster. For security, this doesn't really exist anymore. We can cache responses locally on our own machine, but this is a slightly different concept.

4. Uniform interface

  - Uniform Resource Locator (URL) is used to indicate where we are sending our request and what information we are asking for.

5. Layered system

  - Layers of devices between the client and the server. Our request is passed along until it gets to its final destination.

6. Code on demand

  - JavaScript! We can provide additional functionality in our responses. Before JavaScript, we could only display the content that was sent as-is or use some pre-installed software on the client's machine in order to manipulate the response. Now we can send along code (in a script tag) that tells the client how to work with the data.

**## REST and HTTP**

- REST is not the same as or a replacement for HTTP

- It is simply the set of architectural constraints (defined above) that can be implemented by othe protocols (including HTTP)

**## RESTful**

- A collection of conventional endpoints to organize data interaction.

- There is NO "standard" with rules governing it, just a very popular and convenient organization method for setting up routes with recognizable meaning.

**### Collection vs Single-Resource URLs**

- Collection URLs end in plural nouns and point to an entire group/collection of data

  - `/posts` would indicate we are referencing all posts

- Single-Resource URLs use a plural noun, followed by a unique identifier, typically the primary key assigned to that resource

  - `/posts/19` would reference the post with an ID of 19

  - `/users/the\_best\_username` important to note that the unique identifier does not strictly have to be a numbered ID, it's just very common for it to be implemented that way

- Nested Collection URLs indicate a many-to-one connection; a collection that is tied to one specific resource, such as all of the comments for a single post. It takes the format of a plural noun, followed by a unique identifier, followed by the plural noun indicating the nested collection.

  - `/posts/19/comments` would reference all comments associated with the post that has an ID of 19.

**### Convention over Configuration**

- We should be able to determine what a request is intending to accomplish just by looking at the HTTP verb that is used along with the URL that it is being sent to.

- Ideal HTTP Verb Interactions (can be performed with ajax, fetch, etc.)

| HTTP Verb | Collection URL Meaning (`/posts`)    | Single-Resource URL Meaning (`/posts/19`) |

|-----------|--------------------------------------|-------------------------------------------|

| GET       | Get "all" of the specified resources | Get the details of the resource           |

| POST      | Create a new resource                | n/a                                       |

| PUT       | n/a                                  | Replace the resource                      |

| PATCH     | n/a                                  | Update the resource                       |

| DELETE    | Delete all of the resources          | Delete the specified resource             |

- HTML-only Interactions (without JavaScript, a browser can only use GET and POST verbs)

| HTTP Verb | URL                  | Meaning                                    |

|-----------|----------------------|--------------------------------------------|

| GET       | **\*\*/posts\*\***           | Get a list of the blog posts               |

| POST      | **\*\*/posts\*\***           | Create a new blog post                     |

| GET       | **\*\*/posts/new\*\***       | Get the form to create a post              |

| GET       | **\*\*/posts/18\*\***        | Get the single blog post with id 18        |

| POST      | **\*\*/posts/18\*\***        | Update the blog post with id 18            |

| GET       | **\*\*/posts/18/edit\*\***   | Get the edit form for blog post with id 18 |

| POST      | **\*\*/posts/18/delete\*\*** | Delete the blog post with id 18            |

- HTML-only Interactions for Nested Resources

| HTTP Verb | URL                        | Meaning                                          |

|-----------|----------------------------|--------------------------------------------------|

| GET       | **\*\*/posts/18/comments\*\***     | Get all comments for blog post 18                |

| POST      | **\*\*/posts/18/comments\*\***     | Create a new comment for post 18                 |

| GET       | **\*\*/posts/18/comments/new\*\*** | Get the form to create a new comment for post 18 |

| GET       | **\*\*/comments/3\*\***            | Get the single comment with id 3                 |

| POST      | **\*\*/comments/3\*\***            | Update the comment with id 3                     |

| GET       | **\*\*/comments/3/edit\*\***       | Get the edit form for comment with id 3          |

| POST      | **\*\*/comments/3/delete\*\***     | Delete the comment with id 3                     |

API (Application Programming Interfaces) Learning Objectives

1. Recall that REST is an acronym for Representational State Transfer
2. Describe how RESTful endpoints differ from traditional remote-procedure call (RPC) services

* Remote-procedure call (RPC) services mirror methods on objects. They usually specify the method name in the URL, such as /getTweetById, with parameters being passed in through a query string, http://localhost/getTweetById?id=12.
* With RPC, we end up with many routes with custom names, unlike the conventions observed in RESTful routes. This means we have to be very familiar with the API in order to use it effectively, as well as necessitating more documentation.
* RPC can be convenient in that the URL tells you exactly what is happening, rather than inferring it like with RESTful conventions. It can also be more convenient in getting related information instead of nesting routes, such as http://localhost/getCommentsForTweetWithId?id=12. The name is much more complicated, but tells you exactly what it does.
* Many times the choice is preference, not strictly which is better/worse. RESTful is more common and the convention over configuration ideas are great for building consistent, predictable applications. It's what we'll be using throughout the course.

1. 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

| **HTTP Verb** | **Collection URL Meaning (/posts)** | **Single-Resource URL Meaning (/tweets/19)** |
| --- | --- | --- |
| GET | Get "all" of the specified resources | Get the details of the resource |
| POST | Create a new resource | n/a |
| PUT | n/a | Replace the resource |
| PATCH | n/a | Update the resource |
| DELETE | Delete all of the resources | Delete the specified resource |

1. Recall that RESTful is not a standard (like ECMAScript or URLs), but a common way to organize data interactions
2. Explain how RESTful APIs are meant to be stateless

* This lends itself well to a typical HTTP implementation. Each interaction between the server and client is independent of each other. We do not need to maintain a session, we can make short, discrete requests.

1. Given a data model, design RESTful endpoints that interact with the data model to define application functionality

* Get all of the posts: GET /posts
* Get a specific post: GET /posts/1
* Create a new post: POST /posts
* Delete a specific post: DELETE /posts/1
* Update a specific post: PATCH /posts/1 (or PUT to replace)
* Get all comments on a post: GET /posts/1/comments
* Get a specific comment: GET /comments/1
* Create a comment on a post: POST /posts/1/comments
* Delete a specific comment: DELETE /comments/1
* Update a specific comment: PATCH /comments/1 (or PUT to replace)

1. Use the express.json() middleware to parse HTTP request bodies with type application/json

* In order for us to use json that has been sent in the body of a request, we have to parse the data with middlware. Without parsing the body, it will appear empty. Express has a built in middleware function to parse json, so we can add it in to the top level of our application in order to parse all requests that come in with json-formatted bodies.

app.use(express.json());

1. Determine the maximum data an API response needs and map the content from a Sequelize object to a more limited data object

* We often don't need to send back to the client *all* of the information that's been provided to us from our database query. We may have extra information that is just unnecessary or could even be a security risk to send in our response (we probably don't want to send the user's hashedPassword on a response, for example).
* We can specify what properties of our Sequelize object that we want to send back instead of sending the entire object.
* In this example, after signing a user up for our app and creating a token, we send back the token and only the id of the user in the response.

router.post(

'/',

check('username')

.exists({ checkFalsy: true })

.withMessage('Please provide a username'),

validateEmailAndPassword,

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,

});

})

);

1. Define a global Express error handler to return appropriate responses and status codes

* This is primarily review from last week.
* We can define a middleware to catch unhandled routes and throw a 404 error.
* We can define an error handler that catches Sequelize validation errors and maps the messages to an errors array.
* We include a generic error handler at the end which will catch any errors, whether they have been manipulated by our previous handlers or not, and responds with an appropriate status and information about the error. Here we can also tailor the response based on whether or not we are in a production environment (not passing the stack trace to end-users in production, for example)

*// 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.errors = ["The requested resource couldn't be found."];

err.status = 404;

next(err);

});

# API Security Roundtable

## OAuth

### Explain the fundamental concepts of OAuth as a way to authenticate users.

- What is OAuth?

- Authentication through a 3rd party.

- Who are the three key players in the OAuth flow?

1. Application (Developers)

2. User

3. Service API

- Is OAuth a standardized? If so, where can I find those standards?

- [RFC 6749 Section 4.3](https://tools.ietf.org/html/rfc6749)

### Describe the workflow of OAuth resource owner password credentials grant

- What are the six high-level steps of the OAuth flow? What happens in each step?

1. Application asks for if they can get info from a third party

2. User issues an Authorization Grant to Application

3. Application sends the Authorization Grant to Service API - Authorization Server

4. Issues a Auth token to the Application

5. Application send the Auth token to Service API - Resource Server

6. GET THE RESOURCE!!!

![flow](./images/abstract\_flow.png)

## JWT

### Describe the components of a JWT and how it is constructed

- What is a JWT?

- packet of stuff that we can verify the contents and the sender. JSON Web Token

- What are the three parts of a JWT token? What are their purposes?

1. Header

- Describes the hashing algorithm that the JWT uses

2. Payload

- Holds the data

3. Signature

- Hash (header + payload + secret)

- On Server we take the (Header + Payload) + secret on server === Signature

This is what one looks like, the '.' separates each part.

```eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.d8FM2YvcKtygfaqgkrxx7aOt4sVWC3V0-xu5e0TVSvg```

### Configure an Express application to use token-based authentication and authorization using OAuth resource owner password credentials grant

- How do we create a JWT in our projects?

- What package do we use?

- `jsonwebtoken`

- Where should we store our secret for our signature?

- In the .env file so it's not shared

- What method do we use to create a JTW token?

jwt.sign(payload, secretOrPrivateKey, [options, callback]);

- Bearer Tokens?? :scream:

- How do we parse a bearer token using middlewear?

- What library do we use?

- `express-bearer-token`

- Okay, I'm using this as a middlewear... what does it do?

- It parses the HTTP header `Authorization` and attaches the value to the req as token. We can then access the token at req.token

- What method do we use to verify a JWT token?

jwt.verify(token, secretOrPrivateKey, [options, callback]);

// We like using the callback, it'll run after it verifies the token

![verify passed](./images/payload.png)

Here you can see that since the secret was right, we have access to the JWT payload

![verify failed](./images/wrong.png)

Here you can see that we now have an err and our payload is undefined. We can use this functionality to authenticate a user, verify that it's not some random token, and create error handlers if the verify fails.

- How can we send a bearer token in Postman?

1. Create a Header called “Authorization” and you can give that a key of

```javascript

`Bearer ${token}`;

//The word Bearer, a space, the token without quotes

2. Go to the Auth tab (located next to Headers), select Bearer from the “type” dropdown, then input your token into the token input box, again no quotes.

Week 12 Learning Objectives

Algorithms

Define the term authentication

Identification of an actor given known credentials

Define the term authorization

Restriction of access to certain actions for an identified actor

Describe the difference between asymmetric and symmetric cryptographic algorithms

symmetric uses one key for both encryption and decryption

asymmetric uses two keys, one for encryption, one for decryption

Describe the difference between cryptographic functions and hash functions

crytographic functions are reversible and the original content can be

decrypted and restored

hash functions are irreversible and the original content cannot be

decrypted and restored

hash functions also usually include a cryptographic salt to add to the

original content when put through the hash function to make the hash function’s outputs more randomized and protect against rainbow attacks

Identify “strong” vs. “broken” hash functions

Current strong hash functions:

Argon2

PBKDF2

Bcrypt

Broken hash functions are especially prone to Rainbow Attacks:

md5

SHA1

Rainbow Attack - an attack on a site that tries to sign into multiple

users accounts using a table of common passwords to hashes

Both md5 and SHA1 are weak algorithms with well-known rainbow tables and should never be used to store sensitive data at rest.

ReSTful Endpoints

Recall that ReST is an acronym for Representational State Transfer

Recall that RESTful is not a standard (like ECMAScript or URLs), but a common way to organize data interactions

THERE IS NO RESTFUL STANDARD!!!

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

Method Meaning

GET Get one or some resources

POST Create a resource

PUT Update a resource

DELETE Delete one or some resources

Singular resource: /tasks/:id

Collection of a resource: /tasks

POST /tasks is actually conventionally used for creating a single resource

KNOW HOW TO CREATE THESE BASED ON A GIVEN PURPOSE For example, know how to write out the RESTful endpoints if given a data model like the following what endpoints would you create to support getting a specific resource, deleting a resource, creating a resource, updating a resource and retrieving all resources

Tasks

id

description

created\_at

updated\_at

Explain how RESTful APIs are meant to be stateless

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.

Middlewares

CORS

Define Cross-Origin Resource Sharing (CORS) and how it is implemented in some Web clients

CORS - Cross-Site Resource Sharing

Allow requests to come from places of origin other than the server’s url origin

cors - package for implementing cross-site resource sharing

Bearer Token

express-bearer-token - package for extracting bearer tokens from a request in express

JSON Web Token

3 Parts to a JWT:

Header - holds information about how JWT was signed, e.g. the algorithm used

Payload - data of the JWT

Signature - verifies that the origin of the JWT is the one who signed it

Describe the components of a JSON Web Token (JWT) and how it is constructed

Header and payload are base64 encoded and can be easily decoded by anyone

without the secret

Auth tokens are used in the processes of:

identification

authentication

authorization

jsonwebtoken - package for signing, decoding, and verifying JWT’s

jwt.sign(payload, secret) - creates a JWT with the given payload and creates

a signature with the secret

jwt.decode(token) - decodes the token and returns the payload

(doesn’t need secret)

jwt.verify(token, secret) - verifies that the JWT was signed with the given

secret and, if so, returns the payload. Otherwise throws an error

express.json

Use the express.json() middleware to parse HTTP request bodies with type application/json

middleware for parsing the body of an HTTP request that has a Content-Type

of application/json

express.urlencoded

middleware for parsing the body of an HTTP request that has a Content-Type

of application/x-www-form-urlencoded

OAuth 2.0

Protocol for authenticating users via a trusted 3rd party