Week 12 Study Guide

Table of Contents

Authentication Learning Objectives

Define the term authentication

Authentication = Who

Authentication is verifying a user is who they say they are. This involves
the user having a *secret* or *credential* of some kind and supplying that to the server
and having the server verify that secret.

This differs from Authorization.

Authorization = What

Authorization is *what* Role or Permissions the user has.

Types of Secrets

- **password** - Usually entered directly by the user
- **token** - Like a password but generated by the server and stored on the client
  so that the client can continue to make requests on the user's behalf.
  Examples of this include OAUTH tokens and JWT Tokens.
- **session id** - Similar to a token, this is used by the browser to keep track
  of a user's session with the server, thus verifying that the user is the
  correct user. (This isn't enough on it's own to authenticate, you must use a
  token or password first). But it still should be protected just like a
  password or token.
- **two factor authentication code** - A code a user enters in addition to
  their password, can be sent via SMS, Email or generated by some software
  on the user's device.

Describe the difference between asymmetric and symmetric cryptographic algorithms

- **Symmetric Encryption** - A single *private* key is used to encrypt some data
  using a lossless algorithm. You can use the *private* key to decrypt and
  restore the original data

  Examples:

  - 1Password or LastPass encryption
  - Encrypting your hard drive with Filevault
    or Bitlocker.
  - Using GNUPG to encrypt a file.

- **Asymmetric Encryption** - A system with two keys, a *private* key
  and a *public* key. If Bob wants to send an encrypted message to Alice then
  Bob uses his *private* key and Alice's *public* key to encrypt the message.
  Now not even Bob can decrypt the message. The only way to decrypt the message
  is for Alice to use her *private* key anb Bob's *public* key.

  Examples:

  - Web Servers use SSL (Secure Socker Layer) to create an encrypted
    communication channel between the client's *private* key and the web server's

public key.
    - GNUPG can be used to send encrypted Emails to other users as long as you have that user's public key.
    - SSH (Secure Shell) creates an encrypted connection between a client and server to provide remote access (github also uses this for SSH git access)

Identify "strong" vs. "broken" hash functions

Hash functions are algorithms that generate a lossy *hash* of a value such that the original value cannot be recovered if someone has the *hash*

A Salt is a random value added to a password *before* hashing and stored alongside the password in the database. This makes rainbow tables of pre-computed hashes less useful for figuring out the original password.

Broken hash functions

Currently these hash functions are "broken", meaning they should not be used to store user passwords in a database at rest.
(Either because the algorithm has been solved, or the algorithm has been shown to have a flaw)

1. MD5
2. SHA-1

Strong hash functions

These are currently "strong" hash functions:

1. PBKDF2
2. bcrypt
3. Argon2

Neither of these are an exhaustive list, but these are common ones available for use with NodeJS libraries and covered in this course.

It's still important to protect the hashes in your database because if they are leaked, they can be attacked in several different ways.

- *Brute force* - This is just where you try to hash every possible combination of characters until you get a matching hash. This is slow.
- *Rainbow Attack* - You use a list of pre-computed hashes (A Rainbow Table) and compare the hash you want to crack against it. This is still slow but faster than the brute force attack

Implement session-based authentication in an Express application

Session based authentication is where you use a session-id, stored in a cookie to authenticate a user once they have first authenticated with a password.

This is the package we use for express session authentication:

```
npm install express-session
```

We configure it as express middleware like so.

```
app.use(session({
  secret: 'a5d63fc5-17a5-459c-b3ba-6d81792158fc', // Secret used to verify the session-id
  re`sa`ve: false, // Recommended by the express-session package
  saveUninitialized: false, // Recommended by the express-session package
}));
```

> The secret should probably be stored in an *environment variable* in your `.env` file instead of hard coded like this.

The middleware creates a `.session` property on the express `req` (Request) object. You can use this to store information you would like to persist for the user, such as the user's `id`, or other information that is relevant for your application.

By default `express-session` stores the session data in memory. Which means if you restart your express server you are going to lose your session data (all users will suddenly be logged out of the sytem).

To remedy this you can configure a different mechanism for the session store.

We can use the `connect-pg-simple` module to connect to postgreSQL and store the session in a table in our database.

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

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

It will need an environment variable named `DATABASE_URL` to know which postgreSQL database to connect to.

The postgreSQL url should look something like this:

```
DATABASE_URL=postgresql://<username>:<password>@<host>:<port>/<database name>
```

Implement a strong hash function to securely store passwords

We can use `bcrypt` to store a password hash in the database instead of storing the plain text password, which is insecure.

Installing bcrypt

```
npm install bcryptjs
```

Generating a hashedPassword

Then wherever you register a new user you can add a line like this to generate a hash to store in the `user` table in the database

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

Verifying a password against a hashed password

Whenever we need to login a user in, we just check the password they give us against the hashed password from the database like so:

```
const passwordMatch = await bcrypt.compare(password, user.hashedPassword.toString());
```

passwordMatch will be a boolean.

Describe and use the different security options for cookies

- **httpOnly** - Makes the cookie only be able to be read and written by the browser and *not* by JavaScript.
- **secure** - The browser will only send this cookie if the connection is HTTPS.
- **domain** - If this isn't set it will assume the cookie should be the same as the site's domain.
- **expires** - The date and time when the cookie expires
- **maxAge** - The number of milliseconds before the cookie expires (usually easier to use than expires)
- **path** - Defaults to '/' but it's a prefix for the path in the URL that the cookie is valid for (seldom used)

Application Programming Interfaces Objectives

Recall that REST is an acronym for Representational State Transfer

Describe how RESTful endpoints differ from traditional remote-procedure call (RPC) services

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`. You need comprehensive documentation to know all of the different methods available by an RPC-designed API. You immediately know how to perform the basic CRUD (Create, Read, Update, Delete) operations on RESTful resources by using the HTTP verbs.

In RESTful APIs the URIs are the nouns while the HTTP methods are the verbs.

In RPC based APIs, you are calling a remote *function* on the server from the client, so you will see a method name and parameters in the request or URI.

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` - Get a representation of resource
- `POST` - Create a new resource
- `PUT` - Update a resource with complete data
- `PATCH` - Update a resource with partial data
- `DELETE` - Delete a resource

Resources

- `/tweets` would be a *collection resource*
- `/tweets/17` would be a *single resource*

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

Given the table:

Tweets
id
message
createdAt
updatedAt

These might be some good RESTful endpoints for an API (These would return JSON)

Path	HTTP Verb	Meaning
/api/tweets	GET	Get an list of your tweets
/api/tweets	POST	Create a new tweet
/api/tweets/17	GET	Get the details of a tweet with the id of 17
/api/tweets/17	PUT/PATCH	Update a tweet with the id of 17
/api/tweets/17	DELETE	Delete a tweet with the id of 17

While these might be good RESTful endpoints for a frontend (These would return HTML)

Path	HTTP Verb	Meaning
/tweets	GET	Show a page of tweets
/tweets/new	GET	Show a form to add a new tweet
/tweets	POST	Create a new tweet
/tweets/17	GET	Show a page displaying a single tweet
/tweets/17/edit	GET	Show a form to edit a tweet with the id of 17
/tweets/17	PUT/PATCH	Update a tweet with the id of 17
/tweets/17	DELETE	Delete a tweet with the id of 17

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

The reason we call it *'RESTful'* is because it is more of an idea or a set of guidelines for building APIs which conform nicely with how the web already works.

Explain how RESTful APIs are meant to be 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.

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

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 automatically parse request body content formatted in JSON so that it is available via the req.body property.

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

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

Instead of this which will return everything about the user, including the hashed password…

```
app.get('/users/:id', asyncHandler(async (req, res) => {
  const user = await User.findByPk(req.params.id);
  res.json(user);
}));
```

We can build a new POJO to return instead of the user object like this:

```
app.get('/users/:id', asyncHandler(async (req, res) => {
  const user = await User.findByPk(req.params.id);
  // A new POJO containing only user.id and user.email
  const userData = {
    id: user.id,
    email: user.email
  };
  res.json(userData);
}));
```

If you have a resource that returns a *collection* you can use map in a handy way to build a new POJO for each model object including only the info you want.

```
app.get('/users', asyncHandler(async (req, res) => {
  const users = await User.all()
  // A new array of POJOs containing only the user.id ans user.email fields
  const usersData = users.map(user => {
    id: user.id,
    email: user.email
  });
  res.json(usersData)
}));
```

Define a global Express error handler to return appropriate responses and status codes given a specific Accept header in the HTTP request

> Note: we didn't do this in class, so you aren't required to know this, but here's an example you can use for projects.

You would need to do this if your application is a *single* express server with both a JSON API and Pug templates that return HTML.

```
// 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);
});

// Error handler that returns the correct type of data
// based on the `Accept` header
app.use((err, req, res, next) => {
  res.status(err.status || 500);
  const acceptHeader = req.get("Accept");

  const errorData = {
    title: err.title || "Server Error",
    message: err.message,
    stack: isProduction ? null : err.stack
  }

  if (acceptHeader === 'text/html') {
    res.render('error-page', errorData)
  } else if (acceptHeader === 'application/json') {
    res.json(errorData);
  } else {
    res.send("Server Error");
  }
});
```

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

By default JavaScript running in the browser can only access API resources from the same origin (i.e. domain, protocol, port).

CORS works by setting HTTP Headers on the API endpoint which tell the browser if a particular domain, protocol, port combination is allowed to access that resource on the API.

Configure your API to use CORS to prevent unauthorized access from browser-based Web requests

```
npm install cors
```

This would allow an application running on localhost port 4000 to access this API

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

> You probably want to store the CORS origin in an environment variable in your `.env` file instead of hard coding it.

API Security Objectives

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

Oauth 2.0 is a standardized mechanism for authenticating users using a 3rd party Oauth provider.

Describe the workflow of OAuth resource owner password credentials grant (RFC 6749 Section 4.3)

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

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

JWT can be used for *authentication* and *authorization*, but they can't be used to decrypt data.

- *header* - Contains basic info about the JWT such as the `typ` (type) and `alg` (algortihm)
- *payload* - Contains any extra info you would like to store inside the JWT
- *signature* - a digital signature created using a hashing algorithm used to verify that the payload hasn't been tampered with. It also verifies the origin of the JWT.

The JWT token consists of a single string with these three parts separated by the `.` character. Each part of the token is Base64 encoded.