Month 2, Week 4: The Unabridged Learning Guide

Introduction: The Gates of the Application

In our journey so far, we have built a functioning Express.js API. It has well-structured routes, clean controllers, and can create, read, update, and delete data. However, it has a catastrophic flaw: it is an open house. Anyone can walk in and do anything. Anyone can read sensitive data, anyone can modify information, and anyone can delete the entire dataset.

This week, we build the gates. We learn the non-negotiable, foundational skills of backend security: **Authentication** ("Who are you?") and **Authorization** ("What are you allowed to do?").

We will begin by addressing the single most critical security practice: **never storing plain-text passwords**. We will take a deep dive into the art of cryptographic hashing and salting with `bcrypt`, the industry-standard tool for protecting user credentials.

Next, we will architect a modern, professional authentication system using **JSON Web Tokens (JWT)**. This is the cornerstone of the stateless architecture that RESTful APIs are built upon. We will explore the structure of a JWT, understand the cryptographic signature that makes it secure, and learn how to generate tokens for users when they log in.

Finally, we will put it all into practice by building a custom **authentication middleware**. This function will act as the gatekeeper for our protected routes, verifying incoming tokens and ensuring that only authenticated users can access sensitive endpoints. By the end of this lesson, you will have transformed your open-house API into a secure, professional-grade service.

Table of Contents

5. Advanced Concepts & Security Best Practices
    - Refresh Tokens
    - Storing your JWT Secret
6. Take-Home Assessment: Role-Based Access Control

Module 1: The Sanctity of Passwords

Why Storing Plain-Text Passwords is Unforgivable If an attacker breaches your database and finds a table of plain-text passwords, you have not just failed technically; you have failed ethically. Users frequently reuse passwords across many services. A breach of your application could lead to the compromise of their email, banking, and social media accounts. The damage is catastrophic. As a professional architect, your primary duty is to protect your users' data, and that begins with their credentials.

Cryptographic Hashing: The One-Way Street The solution is to never store the password itself. We store a **hash** of the password. A cryptographic hash function is a one-way algorithm: * **Easy to compute:** `password -> hash` * **Impossible to reverse:** `hash -> password`

When a user registers, we hash their password and store the hash. When they log in, we hash the password they just submitted and compare it to the hash we have stored. If the hashes match, the password is correct. We never need to know or see the original password after registration.

Salting: Defeating Pre-Computation Attacks A simple hash is not enough. Attackers have pre-computed tables of hashes for common passwords, called "rainbow tables." If they find the hash for "password123" in your database, they can look it up in their table and find the original password.

To prevent this, we use a **salt**. A salt is a random string of data that is unique to each user. We combine the salt with the password *before* hashing it.

```
hash(password + salt)
```

Now, even if two users have the same password ("password123"), their stored hashes will be completely different because they have different salts. Rainbow tables become useless. The `bcrypt` library handles the generation and storage of this salt for you automatically. The salt is stored as part of the final hash string, so you don't need a separate database column for it.

Implementing `bcrypt` in Practice

- `bcrypt.hash(plainTextPassword, saltRounds)`: This is an `async` function. The `saltRounds` is a "cost factor." It determines how computationally complex the hash calculation is. A higher number is slower but exponentially more secure against brute-force attacks. A value of `10` to `12` is a good baseline today.
- `bcrypt.compare(plainTextPassword, hashFromDB)`: This is also an `async` function. It takes the plain-text password the user just submitted and the full hash string from your database. It will automatically extract the salt from the stored hash,

apply it to the plain-text password, perform the hash, and then do a timing-safe comparison to see if they match.

Module 2: Stateless Authentication with JWT

The Problem with Server-Side Sessions In traditional stateful applications, after a user logs in, the server creates a "session" in its memory or database and gives the client a cookie with a Session ID. On every request, the client sends the cookie, and the server looks up the session to see who the user is.

This works, but it violates the REST principle of **statelessness**. It creates problems in modern, scalable architectures: * **Scalability:** If you have multiple servers behind a load balancer, how does Server B know about a session that was created on Server A? You need complex solutions like sticky sessions or a shared session store (like Redis), which adds complexity. * **Coupling:** The client and server are tightly coupled. Other types of clients (like mobile apps or other backend services) may not work well with web-based cookies.

The JWT Solution: A Self-Contained Passport A JWT is like a passport. When you log in, the server (the issuing country) gives you a passport (the JWT) that contains verifiable information about you (the payload, e.g., your user ID).

When you want to access a protected resource (cross a border), you simply present your passport. The border agent (the server's authentication middleware) can look at the passport, verify its security features (the signature), and confirm that it was issued by a trusted authority and has not been tampered with. The agent does not need to call back to the main office (a session store) to check if you are a valid citizen. The passport is self-contained.

Anatomy of a JWT: Header, Payload, Signature

- **Header:** Contains metadata. The `alg` (algorithm) and `typ` (type) are standard.
- **Payload:** Contains the "claims" or data. These are statements about an entity.
  - **Registered Claims:** A set of predefined claims like `iss` (issuer), `exp` (expiration time), `sub` (subject). `exp` is critical for security.
  - **Public Claims:** Custom claims you define, like `userId`, `role`, etc. Keep this data minimal and non-sensitive.
- **Signature:** This is the security. It is created by hashing the header, the payload, and a secret key that is **known only to your server**. `HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)`

Module 3: Implementing JWT in Express.js

The `jsonwebtoken` Library The `jsonwebtoken` library is the de facto standard for working with JWTs in Node.js. Install it with `npm install jsonwebtoken`.

The Registration Flow: Hashing the Password The registration flow is straightforward. Its only job is to validate the input, hash the password, and store the new user record. It does not return a token. A user must explicitly log in after registering.

The Login Flow: Comparing the Hash and Signing the Token This is a multi-step process: 1. Find the user in the database by their email/username. If not found, send a generic "Invalid credentials" error. 2. Use `bcrypt.compare()` to check if the submitted password matches the stored hash. If not, send the same generic error. 3. If credentials are valid, create a payload object for the JWT. Only include the essential, non-sensitive information needed to identify the user in future requests (e.g., `{ userId: user.id }`). 4. Call `jwt.sign()` with the payload, your secret key, and options like `expiresIn: '1h'`. 5. Send the resulting token back to the client in a JSON response.

Module 4: Protecting Routes with Middleware

The "Gatekeeper" Middleware Strategy We will create a single, reusable middleware function. Its only job is to check for a valid JWT. We can then apply this middleware to any route or router that needs to be protected.

Writing the Authentication Middleware

1. **Extract the Token:** Look for the `Authorization` header in the `req` object. It should be in the format `Bearer <token>`.
2. **Validate:** Check if the header and token exist. If not, send a `401 Unauthorized` response.
3. **Verify:** Use `jwt.verify(token, secret)` inside a `try...catch` block. This function will throw an error if the token is invalid (bad signature) or expired. If it throws, the `catch` block will execute, and you should send a `400 Bad Request` or `401 Unauthorized` response.
4. **Attach Payload:** If `verify()` succeeds, it returns the decoded payload. The best practice is to attach this payload to the request object: `req.user = payload;`.
5. **Pass Control:** Call `next()` to pass the now-augmented request to the next middleware or the final route handler.

Applying the Middleware to Routes You can apply it to a single route: `router.delete('/:id', authMiddleware, deleteUserController);`

Or, you can apply it to an entire router. Any route defined *after* this `use` call will be protected: `router.use(authMiddleware); router.patch('/:id', ...); router.delete('/:id', ...);`

Advanced Concepts & Security Best Practices

- **Refresh Tokens:** JWTs are meant to be short-lived (e.g., 15 minutes). This limits the damage if one is stolen. But you don't want to force the user to log in every 15 minutes. The solution is a **Refresh Token**. When a user logs in, you issue both a

short-lived *access token* (the JWT) and a long-lived *refresh token* (a random, secure string stored in your database). When the access token expires, the client sends the refresh token to a special `/refresh_token` endpoint. The server validates the refresh token against the database and, if valid, issues a new access token.

- **Storing your JWT Secret:** Never hard-code secrets in your source code. They should be stored in **environment variables**. Create a `.env` file (and add it to `.gitignore`!), and use a library like `dotenv` (`npm install dotenv`) to load these variables into `process.env`.

Take-Home Assessment: Role-Based Access Control

Objective: To extend the in-class exercise by implementing a simple authorization system.

The Task: Building on the secured API from the in-class exercise: 1. **Add a `role` property** to your user objects in the in-memory array. Make one user an `'admin'` and the others `'user'`. 2. When you generate the JWT during login, **include the user's `role` in the payload**. 3. **Create a new middleware function** called `adminOnly`. This middleware should run *after* your main `authMiddleware`. 4. The `adminOnly` middleware should check if `req.user.role` is equal to `'admin'`. * If it is, it should call `next()`. * If it is not, it should send a `403 Forbidden` status with an error message. 5. **Apply the middleware:** * The `DELETE /users/:id` route should now require *both* `authMiddleware` and `adminOnly`. A regular user should not be able to delete anyone. * The `PATCH /users/:id` route should have more complex logic: an admin should be able to patch anyone's profile, but a regular user should only be able to patch their *own* profile. You will need to add this logic inside the controller, comparing `req.user.userId` to `req.params.id`.

Submission: Submit the entire updated project via a Pull Request to your personal assignments repository. ‘