Backend Security

bcrypt

Content

- Backend Security Basics
- Environment Variables
- Input Validation
- Hashing
- Bcrypt
- JWT
- Cookies
- Rate Limiting
- Cross-Origin Resource Sharing
- Access Control and Role-Based Authorization
- Logging & Monitoring

Backend Security Basics

Backend Security

- **What is Backend Security? :**

    - Backend security refers to the practices, tools, and strategies used to protect the server-side part of a web application including the database, APIs, user data, and application logic from unauthorized access, attacks, and data leaks.

    - It is about making sure only the right people can access the right data, and keeping hackers out of your system.

Backend Security

- Why is backend security important? :
  - To protect sensitive user data (like emails, passwords, financial info)
  - To prevent data loss or leaks
  - To stop attackers from:
    - Taking over user accounts
    - Deleting or modifying data
    - Crashing your server
    - To build trust with your users and comply with privacy laws

Backend Security

- Checklist for a secure backend:
  - Only lets the right people in
    (like having a key to the door)
  - Checks that people are who they say they are
    (like showing ID)
  - Keeps sensitive stuff private
    (like locking things in a safe)
  - Watches out for weird or dangerous behavior
    (like security cameras)

Environment Variables

Environment Variables

- What are environment variables? :
  - Environment variables are like secret notes your app reads at startup. They hold important information that you don’t want to hard-code directly into your files, like passwords, API keys, or database URLs.
  - Benefits of using environment variables:
    - Keep secrets out of your code
    - Easily switch settings between development & production
    - Avoid repeating the same config info everywhere

Environment Variables

- Without environment variables you would store your app secrets in accessible files like:

```
const dbPassword = "MyDBPassword123";
```

  - This is risky – if someone sees your code (on GitHub for example), they also see your secrets.

  - But with environment variables you could simply type:

```
const dbPassword = process.env.DB_PASSWORD;
```

    - Now your code is clean, and the secret stays outside the file.

Environment Variables

ex

- How to Use Environment Variables in Node.js

1. Create a .env file in your project folder:

```
DB_PASSWORD=MySuperSecret123
JWT_SECRET=mysecretkey123
```

1. Install dotenv: $ npm i dotenv
2. Use it in your code:

```
import dotenv from 'dotenv';

dotenv.config();

const dbPassword = process.env.DB_PASSWORD;

console.log("Database password is:", dbPassword);
```

Environment Variables

ex

- Important Rules :
  1. Never upload .env to GitHub! (add it to .gitignore)
  2. Use clear names like PORT, MONGO_URI, JWT_SECRET
  3. Keep separate .env files for development and production

```
PORT=3000
MONGO_URI=mongodb+srv://user:pass@cluster.mongodb.net/mydb
JWT_SECRET=<super_secret_jwt>
```

Input Validation

Input Validation

- What **is Input Validation? :**

  1. Input validation means checking that the data coming from the user is:

     - The right type (e.g. a number, not a string),
     - In the right format (e.g. a valid email),
     - And safe to use (not trying to break your app).

  2. Why is Input Validation Important? :

     - Without validation, users (or hackers) could:

       - Enter nonsense (like "abc" for age),
       - Inject dangerous code (like JavaScript or database commands),
       - Crash your app or steal data.
       - Example: A password field with "`<script>alert(1)</script>`" (XSS attack)

Input Validation

- **How to Validate Inputs in Express? :**

  1. **Use express-validator, a powerful and easy-to-use library: $ npm I express-validator**

```
app.post('/register',[
   body('email').isEmail().withMessage('Email is invalid'),
   body('password').isLength({ min: 6 }).withMessage('Password must be at least 6 characters'),

  (req, res) => {
   const errors = validationResult(req);
   if (!errors.isEmpty()) {
     return res.status(400).json({ errors: errors.array() });
   }
   res.send('User registered!');
  }
```

Input Validation

- **How to Validate Inputs in Express? :**
    - If you are using a router – controller architecture, use **validationResult** from express validator

```
router.post('/signup', [
body('email').isEmail().withMessage('Valid email is required'),
body('password').isLength({ min: 6 }).withMessage('Password must be at least 6 characters')], register);

// Controller
import { validationResult } from 'express-validator';

export const register = async (req, res) => {
const errors = validationResult(req);
if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });
.....
```

Input Validation

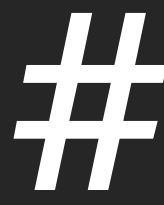
- Any user input – even simple names or strings – can be used for injection attacks if you don’t validate and sanitize it.
- Always check: Is it a string? Is it too long? Is it clean?
- Even simple fields need validation
- Don't trust anything from req.body, req.params, or req.query
- Use .escape() converts characters like <, >, ", ' to safe symbols

```
body('productName').escape();    // Prevents XSS attacks
```

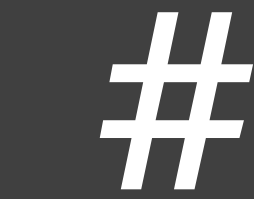
Input Validation

- **Task :**
  - **Clone Repo:** https://github.com/MamdouhSaleh/Tasks.git
  - Switch to the task1 branch and create a branch with <your_name_task1> from this branch
  - Fix codebase to ensure full input validation and sanitization from users
  - Test your APIs using postman or you preferred testing platform
  - Push your fixed code to the repository
  - Please note:
    - Always put the node_modules folder and .env file in a .gitignore file

Hashing

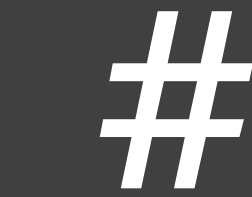


Hashing



- **What is Hashing? :**
  - Hashing is the process of turning data (like a password) into a fixed-length string of characters a fingerprint that cannot be reversed.
  - You turn a password into a secret code that you can check, but you can’t turn back into the original.
  - we should never store passwords in plain text. If someone hacks your database they can access all your users credentials
  - Hashing algorithms always generates the same hash for the same password, that’s why you should always add random strings in your hashed outcome, which is known as salting

Hashing



- **Hashing vs Encryption:**
  - In backend systems, both hashing and encryption are used to protect sensitive data, but they serve very different purposes and operate in fundamentally different ways.
  - Encryption is a two-way process: it transforms readable data (plaintext) into unreadable data (ciphertext) using a key, and with the same or a corresponding key, it can be reversed back into its original form.
  - **Hashing** is a **one-way** process: it converts input data into a fixed-length string (called a digest), and it **cannot be reversed** to retrieve the original input.

Bcrypt

Bcrypt

- What is bcrypt? :
  - bcrypt is a password hashing algorithm designed specifically for secure password storage.
  - Uses salting automatically and is slow by design (which makes brute-force attacks expensive)
  - While SHA-256 and MD5 are technically hash functions, they are fast and predictable, which makes them vulnerable to brute-force attacks.
  - Used in almost all of professional express apps and other frameworks

Bcrypt

1. How to Use **bcrypt in a Node.js App:**
    - Install bcrypt: **$ npm install bcrypt**
2. Import the bcrypt module and use its functions:

```
import bcrypt from 'bcrypt';

const plainPassword = 'mySecurePassword123';
const saltRounds = 10;

const hashedPassword = await bcrypt.hash(plainPassword, saltRounds);

console.log('Hashed:', hashedPassword);
```

Bcrypt

3. **Compare Your passwords:**

```
const isMatch = await bcrypt.compare('userInputPassword', storedHashedPassword);
if (isMatch) {
  console.log(' Password is correct');
} else {
  console.log(' Incorrect password');}
```

- `saltRounds` controls how slow the hash process is (recommended: 10–12 for most apps)
- The returned `hashedPassword` is what you store in the database
- `bcrypt.compare()` hashes the input using the salt inside `storedHashedPassword`, It then compares the result with the stored one
- You never decrypt a password — you just re-hash and compare

Bcrypt

- What is saltRounds? :
  - saltRounds controls how many times the hashing algorithm runs internally.
  - 10 means 2^10 = 1024 rounds
  - Higher = more secure but slower
  - Most real apps use between 10–12

- Good Practices:
  - Always hash passwords before storing
  - Use bcrypt.compare() to validate, never decrypt
  - Store only the final hash (never store plaintext or salt separately)
  - Never log or expose user passwords (even hashed)

Bcrypt

- **Task :**
  - **Clone Repo:** https://github.com/MamdouhSaleh/Tasks.git
  - Switch to the task2 branch and create a branch with <your_name_task2> from this branch
  - Design and implement a simple user authentication backend using Express.js and MongoDB. Your app should support user registration and login, while applying input validation and secure password storage.
  - Prevent duplicate email registration
  - Test your APIs using postman or you preferred testing platform
  - Push your codebase to the repository

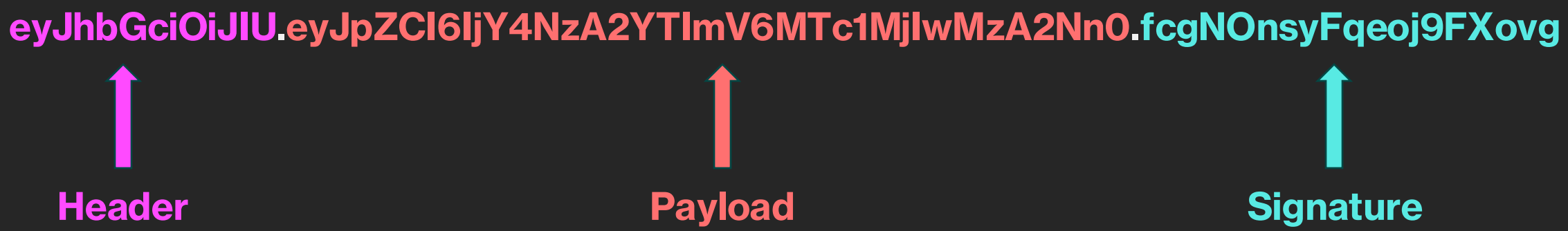
JSON Web Tokens

JWT

- What is JWT? :
  - A JWT (JSON Web Token) is a compact, URL-safe means of representing claims to be transferred between two parties.
  - It's essentially a standard for securely transmitting information as a JSON object, commonly used for authentication and authorization in web applications and APIs.
  - A JWT is composed of three parts, separated by dots: a header, a payload, and a signature.

JWT

- Header: Contains metadata about the token, such as the type (JWT) and the signing algorithm used.
- Payload: Carries the claims, which are statements about an entity (usually a user) and additional data.
- Signature: Used to verify the integrity of the token and ensure it hasn't been tampered with.



JWT

- How does it work? :
  - When a user logs in, the server generates a JWT containing information about the user and their roles.
  - This JWT is then sent to the client (e.g., a web browser).
  - Subsequent requests from the client to access protected resources include this JWT in the Authorization header.
  - The server verifies the signature of the JWT to ensure it's valid and hasn't been altered.
  - If the signature is valid, the server trusts the claims within the JWT and grants access to the requested resource.

JWT

- How to use JWT in express? :
  - First we install the JWT module: $ npm install jsonwebtoken
  - We then need to create a secret key to validate our tokens, we can use a built in module in node called 'crypto' to generate a random hash for us:

```
$ node

-> require('crypto').randomBytes(64).toString('hex')
```

  - This will generate our secret key that we can use in our .env file

```
JWT_SECRET=<your_secret_key>
```

JWT

- We can now use JWT functions in our authentication services and middlewares:

```
import jwt from "jsonwebtoken";

export const login = async (email, password) => {

const user = await User.findOne({ email });
if (!user) throw new Error("User not found");
const isMatch = await bcrypt.compare(password, user.password);

if (!isMatch) throw new Error("Incorrect Username or password");

const token = jwt.sign({ id: user._id, email: user.email }, process.env.JWT_SECRET, {
expiresIn: "1h",
});
return { token, user };};
```

JWT

- This following middleware can now be used for every request that requires a user to be logged in:

```
import jwt from "jsonwebtoken";

const authMiddleware = (req, res, next) => {
const header = req.headers.authorization;
if (!header) return res.status(401).json({ message: "Missing token" });

const token = header.split(" ")[1];

try {
const decoded = jwt.verify(token, process.env.JWT_SECRET);
req.user = decoded;
next();
} catch (err) {
res.status(403).json({ message: "Invalid or expired token" });}};
```

JWT

- To add this middleware to any API request, simply add it like this:

```
app.get("/api/profile", authMiddleware, (req, res) => {
res.json({ message: "Protected route", user: req.user });
});
```

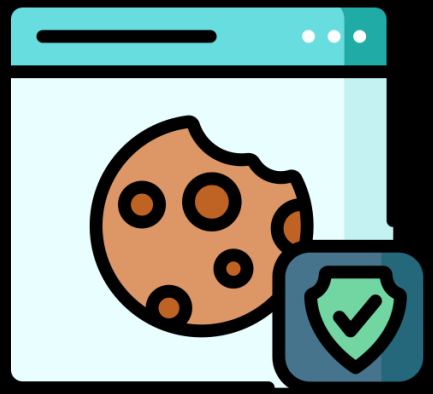
- Now if we want to access this API, the request should include an authorization header with a valid JWT request :

```
$ curl -X GET http://localhost:3000/api/profile \
  -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6I....
```

JWT

- Use refresh tokens to refresh the user's access token after expiry
- Store refresh tokens in DB or Redis with revocation (Blacklisting) support.
- Implement Refresh Token Rotation
- Hash or encrypt your tokens if you store them
- Use HTTPS in production
- Keep payload minimal and Store only essential user claims: role, username, etc.
- Access tokens should be short-lived and can only be regenerated with the appropriate refresh token or by logging in again
- After user's logout, invalidate their refresh tokens

Cookies



Cookies

- What are cookies in web development? :
  - Cookies are small pieces of data that a server sends to the client (browser) and stores there. The browser then automatically sends them back to the server with every subsequent request.
  - You log in to a website → server sets a cookie like:
    Set-Cookie: connect.sid=abc123
  - Now every time you visit another page, your browser sends:
    Cookie: connect.sid=abc123
  - The server uses that ID to identify you and restore your session.

Cookies

- How to use cookies in express.js? :
  - To manage authentication with cookies in Express, we use:
    - express-session – session management
    - connect-mongo – save sessions in MongoDB

```
$ npm i express-session connect-mongo
```

Cookies

- We then initialize the session in our index.js :

```
import session from 'express-session';
import MongoStore from 'connect-mongo';

app.use(session({
  secret: process.env.SESSION_SECRET,     // Used to sign the session ID cookie
  resave: false,                          // Don't save if nothing changed
  saveUninitialized: false,               // Only save if something is set in req.session
  store: MongoStore.create({              // Save sessions to MongoDB
    mongoUrl: process.env.MONGO_URI,
    collectionName: 'sessions'}),
  cookie: {
    httpOnly: true,                       // Can't access cookie from JavaScript (security)
    maxAge: 1000 * 60 * 60,               // 1 hour
    secure: false,                        // Only HTTPS in production
    sameSite: 'lax'                       // Prevents CSRF, but allows navigation
```

Cookies

- Now in our login controller, we can set the user's session:

```
export const login = async (req, res) => {
try {
const { email, password } = req.body;
const user = await loginService(email, password);

req.session.user = user;

res.status(200).json({ message: "Login successful", user });
} catch (err) {
res.status(401).json({ error: err.message });
}
};
```

Cookies

- Now in our authentication middleware, we can simply check if the user is logged in or not :

```
const authMiddleware = (req, res, next) => {
if (!req.session.user) {
return res.status(401).json({ message: "Unauthorized" });
}
next();
};

export default authMiddleware;
```

Cookies

- The module connect-mongo saves our active sessions info in our MongoDB and we can view them by :

```
$ mongosh                          // access MongoDB cli

-> use <your_db_name>              // access your collection

-> db.sessions.find()              // retrieves all active sessions
```

- Only the session ID (connect.sid) is stored in the cookie.
- The actual user data is safely stored on your server (MongoDB).

Rate Limiting

Rate Limiting

- What is Rate Limiting?
  - Rate limiting is a technique used to control the number of requests a client can make to your server in a specific period of time.
  - For example, you might only allow:
    - 100 requests per IP address every 15 minutes.
    - This is extremely useful in APIs to protect the server from:
    - Overuse (accidental or intentional)
    - Abuse (like spamming or brute-force attacks)
    - Sudden traffic spikes

Rate Limiting

- Why Use Rate Limiting?
  - Security:
    - Helps prevent brute-force attacks on login endpoints.
    - Slows down attackers trying to spam or overwhelm your API.
  - Fair Use:
    - Ensures one user/IP doesn’t consume all the resources, impacting others.
  - Server Load Management:
    - Helps reduce unnecessary load on your server and database.
  - Cost Control:
    - Reduces backend resource usage, which can help cut cloud/server costs.

Rate Limiting

- How to Implement Rate Limiting in Express? :
  - We can use a popular package express-rate-limit: **$ npm install express-rate-limit**
  - Then start implementing its middleware:

```
import rateLimit from 'express-rate-limit';

export const apiLimiter = rateLimit({
  windowMs: 15 * 60 * 1000,                    // 15 minutes
  max: 100,                                    // Limit each IP to 100 requests per windowMs
  message: {
    status: 429,
    message: "Too many requests. Please try again later."
  },
  standardHeaders: true,                       // Return rate limit info in the `RateLimit-*` headers
  legacyHeaders: false,                        // Disable the `X-RateLimit-*` headers
});
```

Rate Limiting

- How to Implement Rate Limiting in Express? :
  - Now, to start using it in your APIs, just start injecting it for all routes:

```
import express from 'express';
import { apiLimiter } from './rateLimiter.js';

const app = express();

app.use('/api/', apiLimiter);                    // Apply rate limiting to all API routes

app.listen(3000, () => console.log('Server running on port 3000'));
```

Cross-Origin Resource Sharing

CORS

Cross Origin Resource Sharing

CORS

CORS
Cross Origin Resource Sharing

- What is CORS?
  - CORS (Cross-Origin Resource Sharing) is a security mechanism implemented by browsers,.
  - It is a browser security feature that prevents JavaScript on one origin from accessing data on another origin.
  - Your frontend is running on http://localhost:3000 (React, Vue, etc.).
  - Your backend API is on http://localhost:5000 (Node/Express).
  - When your frontend tries to fetch() data from the backend, the browser blocks the request.

CORS

CORS
Cross Origin Resource Sharing

- To start using CORS inside your express app:
  - Install the middleware: **$ npm install cors**
  - **Then in your main file:**

```
import express from 'express';
import cors from 'cors';

const app = express();

app.use(cors({
  origin: 'http://localhost:3000',
  credentials: true                          // Add thif if you are using cookies
}));
```

Access Control & Role-Based Authorization

Access Control

- What is API Access Control? :
  - Access control in APIs is the process of managing and regulating who or what can access and interact with an API and its resources.
  - It's a crucial security mechanism that ensures only authorized users, applications, or systems can perform specific actions or access certain data through the API.
  - This involves verifying the identity of the requester (authentication) and then determining what they are allowed to do based on their identity and context (authorization).

Access Control

- RBAC Access Control :
  - First, start by adding a role property to your User's schema:

```
const userSchema = new mongoose.Schema({
  name: String,
  email: { type: String, unique: true },
  password: String,
  role: { type: String, enum: ['user', 'admin'], default: 'user' }
});
```

Access Control

- RBAC Access Control :
  - Then add your authorization middleware:

```
export const authorizeRole = (...roles) => {
  return (req, res, next) => {
    if (!roles.includes(req.user.role)) {
      return res.status(403).json({ message: 'Access denied: insufficient role' });
    }
    next();
  };};
```

Access Control

- RBAC Access Control :
  - Then start using the middleware in your routes :

```
router.get('/admin', verifyToken, authorizeRole('admin'), getAdminPanel);
```

  - Finally, sign your JWTs with your user role in your login controller :

```
const token = jwt.sign(
{ id: user._id, role: user.role, name: user.name },
process.env.JWT_SECRET,
{ expiresIn: 'JWT_EXPIRY_TIME' }
);
```

Any Questions?

bcrypt