# Authentication And Session Management

*Software Development Bootcamp*

JSON Web Tokens, Validation Middleware

# What Is Authentication And Session Management?

Components of web security that verify user identities and maintain user states during interactions with web applications.

- Key components include:
  - User Authentication
  - Session Handling
  - Token-based Authentication
  - Encryption
  - Middleware for Session Validation

# Why Are Authentication and Session Management Important?

- They ensure only authorized users access protected resources
- They maintain user state across multiple requests
- They protect against common security threats like session hijacking
- They are essential for compliance with data protection regulations

# Authentication

- Verifies the identity of users
- Typically involves submitting a username/ID and password
- Best practices:
  - User IDs should be case-insensitive
  - User IDs should be unique
  - For high security, assign and keep usernames secret

# Session Management

- Keeps track of who a user is across multiple page visits
- Works like a wristband at an amusement park, in our case the "wristband" will take the form of a **Token**:
  - Server gives each user a unique ID (the wristband)
  - User's browser stores this ID (wearing the wristband)
  - Server checks the ID on each visit (showing the wristband)
- The server remembers user info
- The browser only keeps the ID (**Token)**

# What Are JSON Web Tokens?

- JSON Web Tokens (JWTs) are a compact, URL-safe means of representing claims between two parties
- Structure of a JWT:
  - **Header**: Contains the type of token and the hashing algorithm used
  - **Payload**: Contains claims (statements about the user and additional data)
  - **Signature**: Ensures the token hasn't been altered

# Why Use JSON Web Tokens?

- **Scalability**: Easier to scale across multiple servers or services
- **Flexibility**: Can be used across different domains and services
- **Rich in information**: Can contain user roles, permissions, and other relevant data
- **Mobile-friendly**: Works well with native mobile applications
- **Extensibility**: Can be extended with custom claims for specific use cases

# Example Token

- **.sign()**: Combines the **payload** and **secret** to create the token. Takes three arguments.
  - **Payload:** In this case its an object with the current users id as a value
  - **Encrypt/Decrypt Message:** Passed as a string in the example, should be stored as a **.env** variable
  - **Time:** Details the time the time span for the token.

```
const token = jwt.sign({ id: user._id },
"secret message", {
   expiresIn: 60 * 60 * 24,
 });
```

# When To Issue Tokens

Tokens are typically issued at specific points in the user's interaction with your application:

- **User Registration:** Issue a token immediately after successful account creation. Allows the user to be automatically logged in after registration
- **User Login:** The most common time to issue a token. Verifies user credentials and provides access to protected resources

# Issuing A Token: Signup

- `const user = new User({...})`:
  - Creates a new User object populated with the data from the request body
- `const newUser = await user.save()`:
  - Saves the new user to the database
  - The saved user (including any auto-generated fields like `_id`) is stored in `newUser`
- `const token = jwt.sign(...)`:
  - Generates a new token for the user
  - Payload includes the users `_id`
  - `SECRET` is a secret key used to sign the token (stored in the .env file)

```
router.post('/signup', async(req, res) => {
    try {
        // destructuring the request body
        const { firstName, lastName, email, password
} = req.body
        const user = new User({
            firstName,
            lastName,
            email,
            password
        })
        // saving the user to the database
        const newUser = await user.save()
        // issuing token to user
        const token = jwt.sign(
            {_id: newUser._id},
            SECRET,
            {expiresIn: 60 * 60 * 24}
        )
        res.status(200).json({
            user: newUser,
            token,
            message: "success"
        })
    } catch (error) {
        res.status(500)
    }
})
```

# Encryption

- Plain text passwords within a database are insecure
- Encryption provides protection to both users and databases
- **Bcrpyt** is a package that helps us encrypt passwords

# What Is Bcrypt?

- Bcrypt is commonly used for password encryption
- Uses hashing and salting to hide the password/value
- Hashing:
  - Hashing produces a one-way randomized string based off the plain text string provided.
- Salting:
  - Process of including a randomized string to the previously hashed password prior to being sent to the database
  - Makes the hashed value unpredictable
  - With bcrypt we can determine how many iterations the hashed value should be salted

# Hashing with bcrypt

- **hashSync** a method from the bcrpyt library used to hash a password
- **hashSync** takes two parameters
  - **password**: the plaintext password you want to hash
  - **saltRounds**: A number indicating how many times you want to scramble the password. The higher the number the more secure.

```
const bcrypt = require('bcrypt');
const password = 'myPlainPassword';
const saltRounds = 10;
// Hash the password
const hashedPassword =
bcrypt.hashSync(password, saltRounds);
// Take a look at what our hashed password
looks like
console.log('Hashed Password:',
hashedPassword);
```

# Comparing with bcrypt

- bcrypt.compare checks if the given password matches the stored hashed password.

```
const bcrypt = require("bcrypt");
// The password you want to store
const originalPassword = "mySecret123";
// The stored hashed password (created
earlier)
const hashedPassword =
bcrypt.hashSync(originalPassword, 10);
// When someone tries to log in
const loginPassword = "mySecret123";
// Use bcrypt.compare to check if the
passwords match
bcrypt.compare(loginPassword, hashedPassword,
(err, result) => {
 if (result) {
   console.log("Passwords match!");
 } else {
   console.log("Passwords do not match.");
 }
});
```

# Issuing A Token: Login

- `const email = req.body.email` and `const password = req.body.password`
  - extract the email and password from the request body
- `const foundUser = await User.findOne({ email })`
  - searches the database for a user with the provided email.
- `const verifyPwd = await bcrypt.compare(password, foundUser.password)`
  - compares the provided password with the hashed password stored in the database.
- `const token = jwt.sign(...)`
  - If the email and password are correct, a JSON Web Token (JWT) is generated.

```
router.post('/login', async(req, res) => {
    try {
        // not using destructuring
        const email = req.body.email
        const password = req.body.password
        // find user in database using the email
        const foundUser = await User.findOne({ email
})
        if(!foundUser) throw new Error("User does not
exist")
        // compare password provided with password
connected to user
        const verifyPwd = await
bcrypt.compare(password, foundUser.password)
        if(!verifyPwd) throw new Error('incorrect
password')
        // issue token
        const token = jwt.sign(
            {_id: foundUser._id},
            SECRET,
            {expiresIn: 60 * 60 * 24}
        )
        res.status(200).json({
            message: "success",
            foundUser,
            token
        })
    } catch (error) {
        res.status(500)
    }
})
```

*Topic*

# Creating validateSession Middleware

# Middleware

- Middleware is essentially a function that accesses our request, response, and then moves on to other aspects of our code (using `next()`)
  - Middleware can be customized to your projects needs
  - Middleware functions have 3 parameters (request, response, next)

# Validation Middleware

- validateSession is a middleware process that helps verify what actions (CRUD) a user can make within our application
- For example when a user logs into their social media account, they should only be allowed to post, update, or delete their own content.
- We can tie data together using the unique _id

# Step 1 Imports and Setup

- `import jwt from 'jsonwebtoken'`
  - imports the jsonwebtoken library, which is used to verify the JWT token.
- `import User from '../models/user.model.js'`
  - This imports the User model, which will be used to look up the user in the database.

```
// Bring in JWT to access it's token methods/functionality
import jwt from 'jsonwebtoken'
// Bring in our User model to reference
import User from '../models/user.model.js'
```

# Step 2 Middleware function

- `const validateSession = async (req, res, next) => {...}`
  - defines an asynchronous middleware function that takes the standard Express middleware parameters: request, response, and next.
- The function is wrapped in a **try catch** block to handle any errors that might occur

```
const validateSession = async (req, res, next) => {
    // Middleware still has access to the request, response, and requires the next() function to move past it.
    try {
    //1. Take token provided by request object (headers.authorization)
    const token = req.headers.authorization;
    //2. Check the status of token. (expired?)
    const decodedToken = await jwt.verify(token, process.env.JWT_SECRET);
    //3. Provide response - if valid, generate a variable that holds user info.
    // use the .findById() to check for user that matches token user id
    const user = await User.findById(decodedToken.id);
    if (!user) throw Error("User not found.");
    // Creating a new key within our req (request) object to store our user information
    req.user = user;
    return next(); // moves us onto our routes/endpoint
    } catch (err) {
        res.json({message: err.message});
    }
}
```

# Step 3 Token Extraction and Verification

- `const token = req.headers.authorization`
  - extracts the JWT from the Authorization header of the request.
- `const decodedToken = await jwt.verify(token, process.env.JWT_SECRET)`
  - verifies the JWT using the secret key stored in the environment variables.
  - If the token is invalid or expired, this will throw an error

```
const validateSession = async (req, res, next) => {
    // Middleware still has access to the request, response, and requires the next() function to move past it.
    try {
    //1. Take token provided by request object (headers.authorization)
    const token = req.headers.authorization;
    //2. Check the status of token. (expired?)
    const decodedToken = await jwt.verify(token, process.env.JWT_SECRET);
    //3. Provide response - if valid, generate a variable that holds user info.
    // use the .findById() to check for user that matches token user id
    const user = await User.findById(decodedToken.id);
    if (!user) throw Error("User not found.");
    // Creating a new key within our req (request) object to store our user information
    req.user = user;
    return next(); // moves us onto our routes/endpoint
    } catch (err) {
        res.json({message: err.message});
    }
}
```

# Step 4 User Verification

- `const user = await User.findById(decodedToken.id)`
  - ooks up the user in the database using the ID from the decoded token.
- If no user is found with the ID from the token, it throws an error

```
const validateSession = async (req, res, next) => {
    // Middleware still has access to the request, response, and requires the next() function to move past it.
    try {
    //1. Take token provided by request object (headers.authorization)
    const token = req.headers.authorization;
    //2. Check the status of token. (expired?)
    const decodedToken = await jwt.verify(token, process.env.JWT_SECRET);
    //3. Provide response - if valid, generate a variable that holds user info.
    // use the .findById() to check for user that matches token user id
    const user = await User.findById(decodedToken.id);
    if (!user) throw Error("User not found.");
    // Creating a new key within our req (request) object to store our user information
    req.user = user;
    return next(); // moves us onto our routes/endpoint
    } catch (err) {
        res.json({message: err.message});
    }
}
```

# Step 5 Request Modification

- `req.user = user`
  - If a user is found, it attaches the user object to the request.
  - This makes the user information available to subsequent middleware or route handlers.
- `return next();`
  - If everything is successful, it calls the next middleware or route handler.

```
const validateSession = async (req, res, next) => {
    // Middleware still has access to the request, response, and requires the next() function to move past it.
    try {
    //1. Take token provided by request object (headers.authorization)
    const token = req.headers.authorization;
    //2. Check the status of token. (expired?)
    const decodedToken = await jwt.verify(token, process.env.JWT_SECRET);
    //3. Provide response - if valid, generate a variable that holds user info.
    // use the .findById() to check for user that matches token user id
    const user = await User.findById(decodedToken.id);
    if (!user) throw Error("User not found.");
    // Creating a new key within our req (request) object to store our user information
    req.user = user;
    return next(); // moves us onto our routes/endpoint
    } catch (err) {
        res.json({message: err.message});
    }
}
```

# Using Validate Session

- Here is a sample POST route that creates a new Message.
- We use the validateSession middleware to…
  - Only allow users with a token to create a message
  - Tie the message to the user who created it

```
router.post('/new-message', validateSession,
async(req, res) => {
    try {
        // get the content of the message from
the req.body
        const content = req.body.content
        // get the id of the user creating the
message from req.user (courtesy of
validateSession)
        const user = req.user._id
        const newMessage = new Message({
            content,
            owner: user
        })
        newMessage.save()
        res.status(200).json({
            newMessage,
            message: "success"
        })
    } catch (error) {
        res.status(500)
    }
})
```

*Exercise*

# Add Validate Session Middleware To Your Movies App