HyperionDev

Full-Stack Web Development

HyperionDev

Implementing User Registration and Login using JWT

Full Stack Web Development Session Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly. **(Fundamental British Values: Mutual Respect and Tolerance)**
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Academic Sessions. You can submit these questions here: **Questions**

Full Stack Web Development Session Housekeeping cont.

- For all **non-academic questions**, please submit a query: **www.hyperiondev.com/support**
- Report a **safeguarding** incident: **www.hyperiondev.com/safeguardreporting**
- We would love your **feedback** on lectures: **Feedback on Lectures**

Objectives

- Implement user registration and login functionality in a web application.
- Understand how JWT is used for session management.
- Secure APIs using JWT.
- Handle authentication and authorization in a full-stack application.

Initial Assessment

- Have you implemented user authentication before? (Yes/No)
- How familiar are you with JWT for managing user sessions? (1-5 scale)
- What challenges have you faced with user authentication? (Open-ended)

Introduction

JSON Web Tokens (JWT) are a compact and self-contained way to securely transmit information between parties as a JSON object. They are commonly used for managing user sessions in web applications.

Key points about JWT

- **Structure**:
  - A JWT is composed of three parts: the header, the payload, and the signature. The header typically specifies the token type (JWT) and the signing algorithm (e.g., HMAC SHA256). The payload contains claims, which are statements about the user (e.g., user ID, roles). The signature ensures that the token hasn't been altered.
- **Authentication**:
  - When a user logs in, the server generates a JWT and sends it back to the client. This token can then be included in the HTTP headers for subsequent requests, allowing the server to verify the user's identity without needing to store session information on the server.

Key points about JWT: cont

- **Stateless Sessions**:
  - JWTs allow for stateless session management, meaning the server does not need to keep track of user sessions. This reduces server load and improves scalability, as the server can authenticate users based on the token alone.
- **Expiration**:
  - JWTs can include an expiration time, ensuring that tokens are only valid for a specified period. This enhances security by limiting the window of opportunity for token misuse.

Overview of Token-Based Authentication with JWT

- **User Login**:
  - The user submits their credentials (e.g., username and password) to the server via a login form.
- **Token Generation**:
  - If the credentials are valid, the server generates a JWT. This token includes essential user information (claims) in the payload, such as the user ID and any relevant roles or permissions. The token is then signed using a secret key to prevent tampering.
- **Token Delivery**:
  - The server sends the generated JWT back to the client (usually the browser or mobile app). This token is typically included in the response body or as a cookie.
- **Token Storage**:
  - The client stores the JWT (commonly in local storage or session storage) for future use.

Overview of Token-Based Authentication with JWT: cont

- **Authenticated Requests**:
  - For subsequent requests to protected resources, the client includes the JWT in the HTTP headers (often in the `Authorization` header as `Bearer <token>`).
- **Token Verification**:
  - The server receives the request and extracts the JWT from the header. It then verifies the token's signature using the same secret key used during token generation. If the token is valid and not expired, the server processes the request and grants access to the requested resource.
- **Access Control**:
  - The server can also check the claims within the token to enforce specific access controls, such as ensuring the user has the necessary permissions to perform certain actions.
- **Token Expiration**:
  - JWTs can have an expiration time set (using the `exp` claim). Once expired, the client must re-authenticate (e.g., by logging in again) or use a refresh token (if implemented) to obtain a new JWT without needing to re-enter credentials.

Benefits of Token-Based Authentication with JWT

- **Stateless**:
  - The server does not need to store session information, making it scalable and efficient.
- **Cross-Domain**:
  - JWTs can be used across different domains or platforms, as they are self-contained.
- **Decentralized**:
  - Since the token contains all the necessary information, it can be used by multiple servers or microservices without additional overhead.
- **Mobile-Friendly**:
  - JWTs can easily be utilized in mobile applications, enhancing user experience by allowing for persistent sessions.

Traditional Session Management vs. JWT

- **Traditional Sessions**:
  - In traditional server-side session management, after a user logs in, the server creates a session and stores session data (like the user ID) in memory or a database. The client is given a session ID (often via cookies), which is sent with each request. The server checks the session ID, retrieves the session data, and validates the user.
- **JWT-Based Sessions**:
  - With JWT, the session state (user info, roles, etc.) is encoded directly into the token and sent to the client. The server doesn't need to store any session data, making the system **stateless**. The client sends the JWT with each request, and the server verifies it to authenticate the user.

How JWT Manages Sessions

- **User Authentication (Login)**:
  - When a user logs in with valid credentials, the server creates a JWT that contains the user's data (like the user ID) and signs it using a secret key. The server sends this token to the client.
- **Storing the Token**:
  - The client (usually a web or mobile app) stores the JWT, typically in **localStorage** or **sessionStorage**. It can also be stored in an HTTP-only cookie for added security (to prevent JavaScript from accessing it).
- **Subsequent Requests**:
  - On every request to a protected resource, the client sends the JWT in the **Authorization** header (usually as a Bearer token).
- **Token Validation**:
  - The server extracts the JWT from the request and verifies its signature using the secret key. If the token is valid, the server processes the request. Since the token contains all necessary user information, the server does not need to look up session data in a database.
- **Session Expiration**:
  - JWTs typically include an expiration (`exp`) claim, specifying how long the token is valid. Once expired, the user is logged out (i.e., the token is no longer accepted), and they need to log in again or use a **refresh token** (if implemented).

Q&A

- Please ask any questions if you want any clarification

Hyperiondev

Questions and Answers

HyperionDev

Thank You for attending!