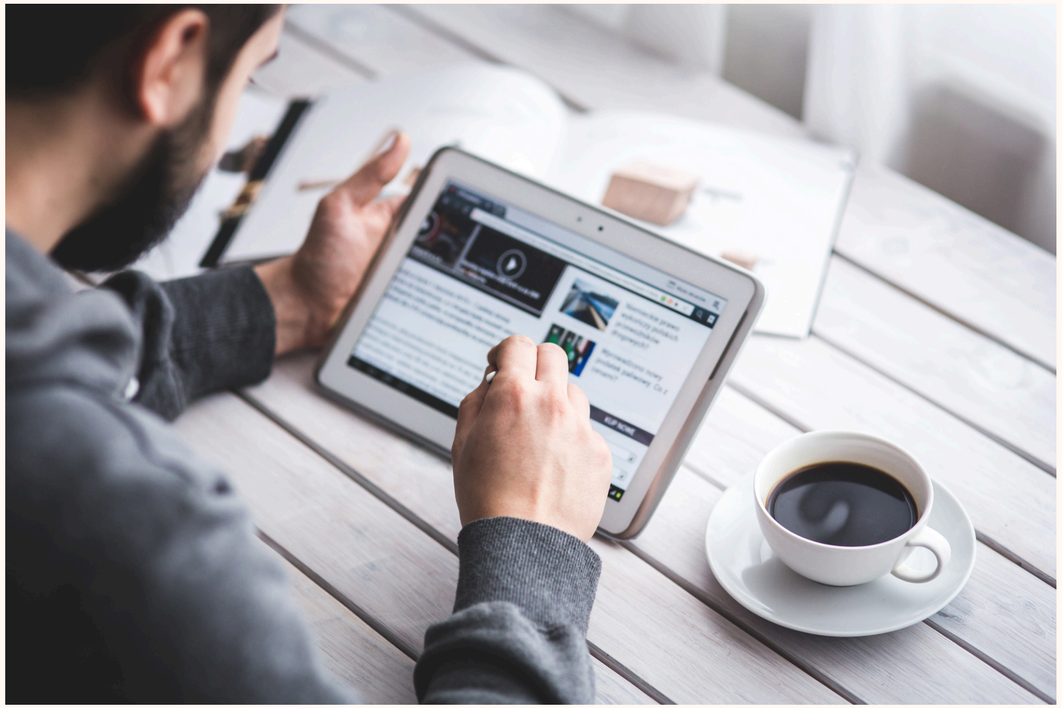
JWT AUTHENTICATION

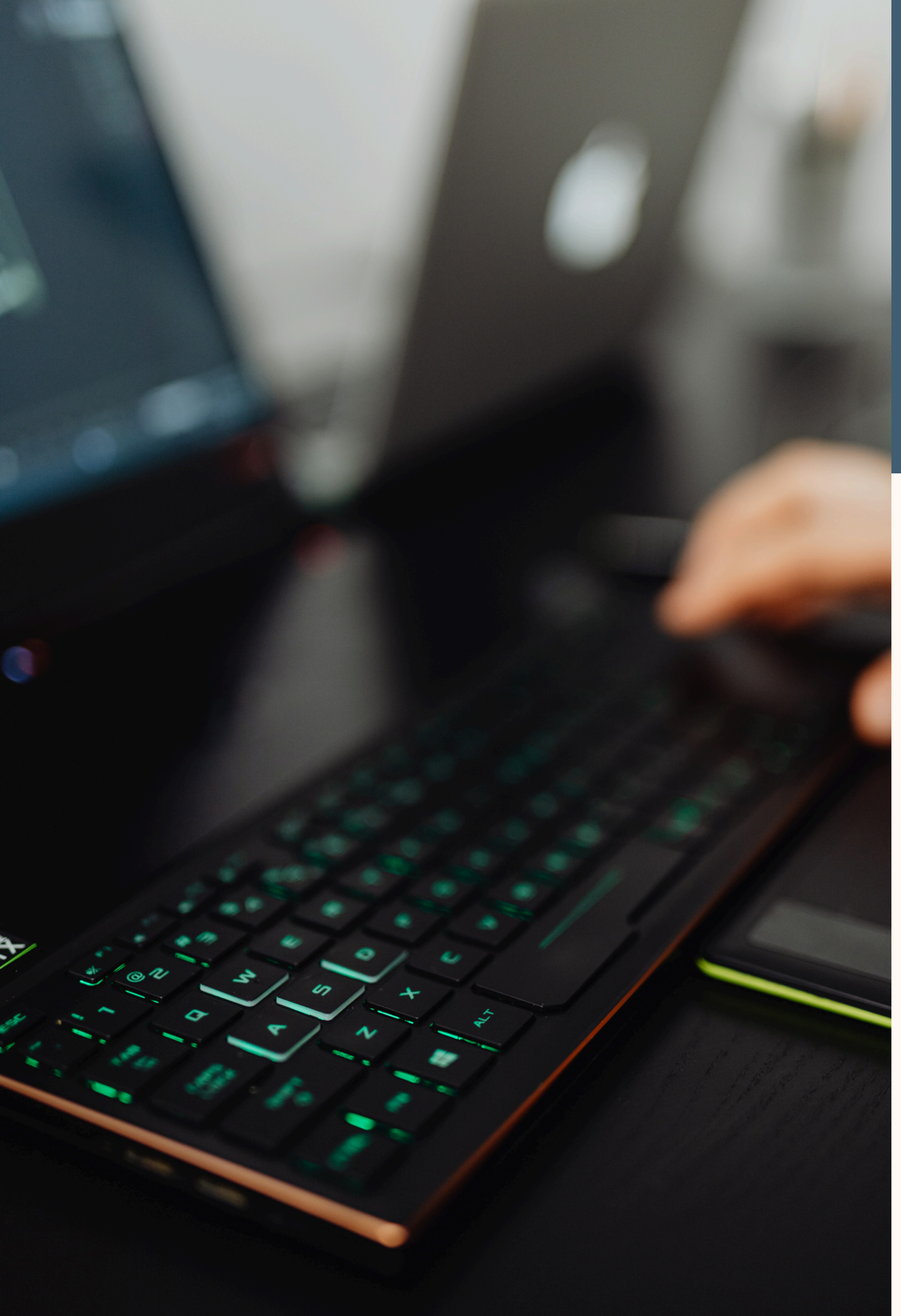
A SIMPLE DEMONSTRATION USING DJANGO

Presented by Pranav Singh

TOPICS TO BE COVERED



1. What is JWT Authentication ?
2. How does JWT Authentication Works ?
3. Creating the user model in Django
4. Access and Refresh Tokens
5. Ways to store JWT Tokens in Frontend



WHAT IS JWT AUTHENTICATION ?

- JWT (JSON Web Token) authentication is **a stateless, token-based method for verifying a user's identity after they have logged in**, commonly used for securing APIs and web applications.

- After a user authenticates, a server issues a JWT which is then sent with subsequent requests to access protected resources, allowing the server to confirm the user's identity without needing to look them up in a database each time.

HOW DOES JWT AUTHENTICATION WORKS ?

1. LOGIN

A user logs in with their credentials (e.g., username and password).

2. TOKEN GENERATION

The authentication server verifies the credentials and creates a JWT. This token is digitally signed and contains information (claims) about the user, such as their ID and permissions.

3. TOKEN TRANSMISSION

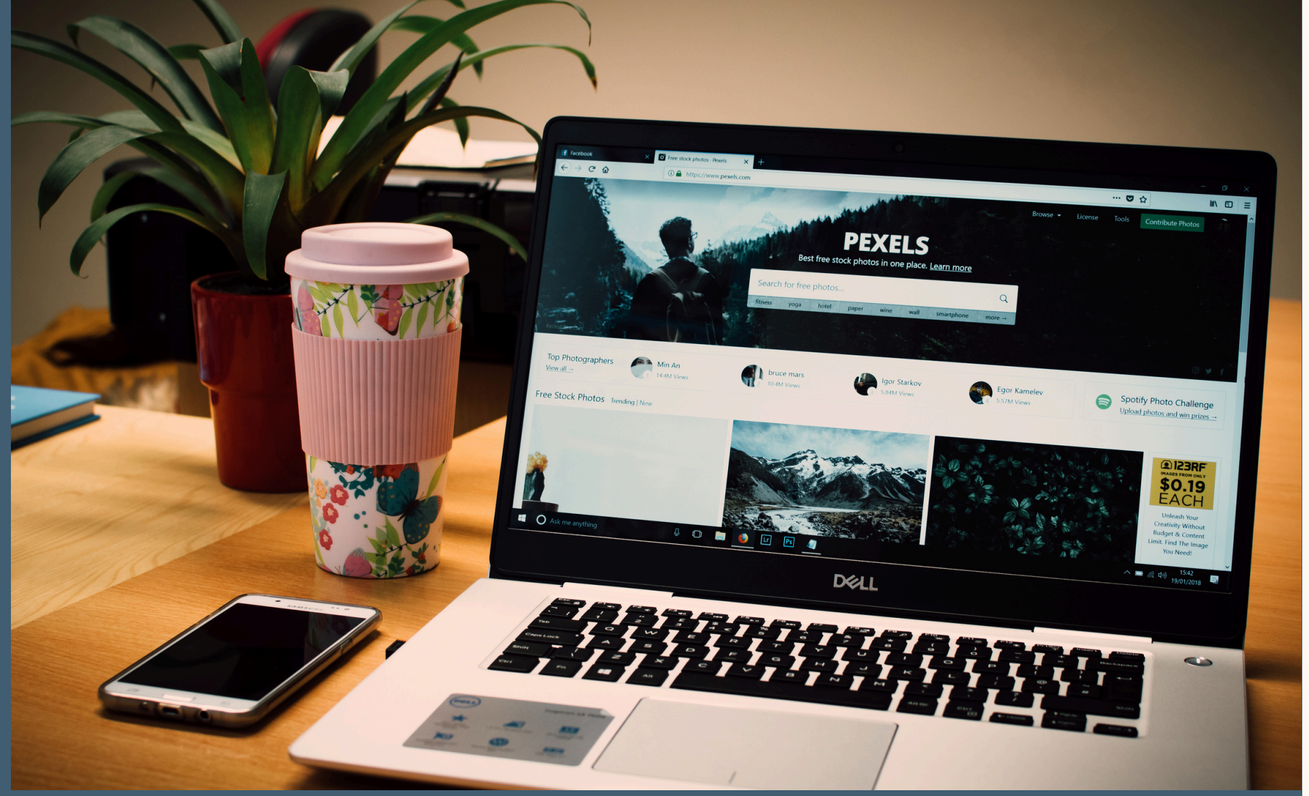
The server sends the JWT back to the client.

4. ACCESSING PROTECTED RESOURCES

The client includes the JWT in the Authorization header of future requests to a protected API or resource.

5. VERIFICATION

The server receives the request, verifies the signature on the JWT using a private key, and checks the claims to ensure the user is authorized to access the resource.



CREATING A USER MODEL IN DJANGO

Built In User Model

Django provides a robust built-in user model and authentication system. While you can use it directly, it is not highly recommended as it does not allow you to add new fields unlike in a custom user model.

Custom User Models

Custom User models on the other hand, are more reliable and allows you to customize the user model by adding new fields. Django allows the creation of a custom user model by inheriting from either AbstractUser or AbstractBaseUser.

```python
class AbstractUser(AbstractUser):
    dob = models.DateField(null=True,blank=True)
    phoneno = models.IntegerField(blank=True)

    def __str__(self):
        return self.username
```

CREATING A USER MODEL IN DJANGO

Custom User Models

1. AbstractUser

- **Purpose:** AbstractUser is used when you want to extend Django's default User model by adding or modifying fields, while retaining all the built-in functionality and fields (like username, email, first_name, last_name, is_staff, is_active, date_joined, etc.).

- **Advantages:** Recommended when you need minor customizations in the User model while retaining all the features of Django's default User model.

```python
class Users(AbstractBaseUser, PermissionsMixin):
    email = models.EmailField(unique=True)
    firstname = models.CharField(max_length=50)
    lastname = models.CharField(max_length=50)
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)
    date_joined = models.DateTimeField(default=timezone.now)

    USERNAME_FIELD = "email"
    REQUIRED_FIELDS = ["firstname","lastname"]

    objects = UserManager()

    def __str__(self):
        return self.email
```

CREATING A USER MODEL IN DJANGO

Custom User Models

2. AbstractBaseUser

- **Purpose:** AbstractBaseUser provides only the core authentication functionality (password hashing, is_active, get_full_name, etc.) and does not include any default fields. You are responsible for defining all user-related fields from scratch.

- **Advantages:** Offers complete control over the user model, allowing for highly customized user structures and authentication flows.

WHY NOT TO USE EMAIL OR USERNAME AS THE PRIMARY KEY ?

Using an email address or username as a primary key initially seems logical due to inherent uniqueness but introduces significant scalability issues. The primary challenge lies in data maintenance when a user updates their email. If the email is the primary key, all related tables referencing the user must undergo complex, potentially risky cascading updates to maintain data integrity. Failure to manage this correctly can result in data loss.

SOLUTION

In Django, if no primary key is explicitly defined, it automatically adds an id field as an auto-incrementing integer primary key. Since this surrogate key is internal and immutable and user is not concerned with this key, it serves as a reliable reference point for all other database relations.

WHY DO WE NEED TWO TOKENS INSTEAD OF A SINGLE TOKEN ?

Access tokens are typically short-lived, often maintaining validity for only 10 to 15 minutes. This limited duration is implemented to enhance security. Should an access token be compromised, the attacker's window of opportunity for misuse is significantly minimized, as the token will rapidly expire.

What is the role of the second token ?

The second token, that is, the refresh token serves a critical function upon the expiration of an access token. It is utilized to generate a new access token upon expiration, which will likewise be valid for a brief period.

Refresh tokens are designed to be long-lived, commonly remaining valid for periods ranging from days to several months. This extended validity ensures a seamless user experience, permitting the user to remain logged in for an extended duration without the necessity of re-entering their credentials each time the access token expires.

WHAT IF THE REFRESH TOKEN IS ALSO COMPROMISED ?

The refresh token might likewise be compromised and can be used by the attacker to generate unlimited access tokens, so how does it solve the problem ?

Even though true, but the refresh token mechanism still represents a superior security posture compared to relying solely on a long-lived access token. This enhanced security is attributed to two primary distinctions: statefulness and transmission scope.

WHAT IF THE REFRESH TOKEN IS ALSO COMPROMISED ?

1. Stateful :

- Refresh tokens are stateful, this means that the server maintains a record for the refresh token in the database, and it can immediately revoke a refresh token if it is compromised by deleting or invalidating its entry in the database.

- Access tokens, on the other hand, are typically stateless and their validity relies solely on a cryptographic signature and an expiration time, which means that the server cannot immediately revoke an access token before its expiration. It will simply trust the information in the token's signature and validate it.

WHAT IF THE REFRESH TOKEN IS ALSO COMPROMISED ?

2. Limited Transmission Scope

- Access tokens are transmitted frequently across various API endpoints to authorize routine data access, significantly increasing the surface area for potential interception.

- Refresh tokens, conversely, are generally sent to a single, specific API endpoint (the Authorization Server's token endpoint) solely for the purpose of obtaining a new access token. By focusing rigorous security measures on this single, high-value endpoint, the probability of an attacker successfully exploiting a captured refresh token can be substantially diminished.

ACCESS TOKEN

- **Purpose:** Access tokens are used to grant immediate access to protected resources. They contain the necessary information (claims) about the user and their permissions to allow a resource server to authorize requests.
- **Lifespan:** Access tokens are typically short-lived (e.g., minutes to a few hours). This short lifespan minimizes the risk associated with a compromised token, as its validity window is limited.
- **Usage:** They are sent with every request to a protected resource in the Authorization header, usually as a Bearer token.
- **Security:** If an access token is stolen, the attacker can impersonate the user for the duration of the token's validity. However, due to their short lifespan, the window of opportunity for misuse is small.

REFRESH TOKEN

- **Purpose:** Refresh tokens are used to obtain a new access token when the current access token expires, without requiring the user to re-authenticate with their credentials.
- **Lifespan:** Refresh tokens are typically long-lived (e.g., days, weeks, or even months).
- **Usage:** They are sent to a dedicated token refresh endpoint to request a new access token and potentially a new refresh token.
- **Security:** Refresh tokens are more sensitive than access tokens as they can grant indefinite access if compromised. Therefore, they should be stored securely (e.g., in an HTTP-only cookie) and often come with mechanisms for revocation.

WAYS TO STORE JWT TOKENS IN FRONTEND

LOCAL STORAGE

Local storage is a popular choice for storing JWTs as it lets you persist tokens across pages and is easy to access from JavaScript. However, it is vulnerable to XSS attacks if not handled properly.

SESSION STORAGE

Session storage is more secure than local storage and offers a simple way to store JWTs but it is not reliable as the token is lost when the tab or window is closed.

HTTP ONLY COOKIE

Cookie storage provides a good balance of functionality and security against XSS attacks., but it requires the frontend and backend to run on the same port and provides only 4KB of storage.

IN MEMORY STORAGE

In-memory storage keeps the JWT in the web application's memory using a JavaScript variable, rather than persisting it in browser storage. It offers better security but token is cleared when navigating between pages and tabs.

THANK YOU

Compiled By:
Pranav Singh