Session Authentication

🔹 Concept

Session authentication is **stateful** — the server keeps track of logged-in users using a **session stored on the server**.

When a user logs in:

1. The server verifies credentials.
2. Creates a **session record** in the database (or cache/memory).
3. Sends a **session ID** back to the browser as a **cookie**.
4. For every next request, the browser automatically sends this cookie.
5. The server looks up that session ID in its session store to identify the user.

🔹 Flow

```pgsql
Client -> POST /login (username, password)
Server -> validates -> creates session_id = 12345
Server -> sends Set-Cookie: sessionid=12345

Client -> GET /profile with Cookie: sessionid=12345
Server -> looks up 12345 -> finds user -> returns profile
```

🔹 Storage

- **Server-side:** session data (user ID, login info)
- **Client-side:** session ID cookie only (not actual data)

🔹 Advantages

- Easy to use and built-in in Django.
- Secure — session data never leaves the server.
- Cookies are automatically handled by browsers.

🔹 Disadvantages

- Not ideal for mobile apps or multiple backend instances (scaling issue).
- Requires server memory/storage for each user session.
- Doesn't work well with **stateless** microservices or APIs

Working with Sessions in Django:

1. During login process add these steps:
   a.
   ```
   # Manually create session
   request.session['user_id'] = user.id
   request.session['username'] = user.username
   ```
   b. You can write request.session.set_expirty(10) #number in seconds
2. Using session data:

   ### 🧩 Profile (uses stored session)

   a.
   ```python
   def profile(request):
       user_id = request.session.get('user_id')
       username = request.session.get('username')

       if not user_id:
           return JsonResponse({'error': 'Not authenticated'}, status=401)

       return JsonResponse({'user_id': user_id, 'username': username})
   ```
3. Logout with Sessions:

🧩 Logout (clears session manually)

```python
def session_logout(request):
    if 'user_id' in request.session:
        request.session.flush()  # clears all session data
        return JsonResponse({'message': 'Logged out successfully'})
    return JsonResponse({'error': 'Not logged in'}, status=400)
```

a.

Why Session Authentication is *not suitable* for Mobile Apps

Session authentication was originally designed for **browser-based** logins — not APIs or mobile apps.

1. The session system relies on **cookies** (`sessionid`) stored in the browser.
2. **Mobile apps don't have a browser context**, so they **don't automatically store or send cookies.**
    a. **You'd have to manually extract, store, and resend them — which defeats the simplicity.**
3. Mobile apps usually communicate with **stateless APIs** (no session memory on the server). That's why JWTs are a much better fit.

For...	Best Authentication
Web apps with browser	Session Authentication
Single Page Apps (React, Angular)	JWT
Mobile apps (Android, iOS)	JWT
Microservices / distributed APIs	JWT

🧩 1. What are Tokens?

A **token** is a small piece of **data issued by the server** after successful authentication — like a **digital key** proving the user's identity.

Instead of maintaining a session on the server (stateful), the **token is self-contained** — it carries the information required to verify a user on its own (stateless).

🔹 Where tokens are used:

- In **JWT authentication** (stateless logins)
- For **API calls** between frontend and backend
- In **mobile applications**
- For **password reset links**, **email verification**, etc.

🧮 2. Types of Tokens

Type	Description
Session Token	Generated and stored on the server (sessionid cookie)
JWT (JSON Web Token)	Self-contained, digitally signed JSON data
OAuth Tokens (Access/Refresh)	Used in external APIs like Google or GitHub login
CSRF Token	Protects against cross-site request forgery
Custom Tokens	Developer-defined tokens for one-time or temporary access

JWT Tokens

Concept

JWT authentication is **stateless** — the server does **not** store user sessions.
Instead, it issues a **token (JWT)** that contains all necessary user data (like ID, roles, expiry time), **digitally signed** by the server.

When a user logs in:

1. The server verifies credentials.
2. Creates a **JWT** (encoded JSON object) and signs it.
3. Send it to the client.
4. The client stores it (usually in localStorage or cookies).
5. For every next request, the client sends the token in the **Authorization header**.

🔹 Flow

```rust
Client -> POST /login (username, password)
Server -> validates -> creates JWT token
Server -> returns { "token": "eyJhbGciOi..." }

Client -> GET /profile with Header: Authorization: Bearer eyJhbGciOi...
Server -> verifies token signature -> identifies user -> returns profile
```

🔹 Structure

A JWT has three parts:

```css
HEADER.PAYLOAD.SIGNATURE
```

Example:

```json
HEADER: { "alg": "HS256", "typ": "JWT" }
PAYLOAD: { "user_id": 1, "exp": 1736200000 }
```

🔹 Advantages

- **Stateless:** No server-side storage needed.
- **Scalable:** Perfect for distributed systems.
- Can be used across **different services/domains**.
- Can carry extra info (roles, permissions, etc.).

🔹 Disadvantages

- Harder to revoke or logout (token stays valid till expiry).
- Larger payloads can affect the network slightly.
- Must be stored securely — localStorage can be attacked via XSS.

How Does the Algorithm Work? (HS256)

`HS256` = HMAC + SHA256

1. **HMAC** stands for **Hash-based Message Authentication Code**.

Process:

1. Combine `header` + `payload` and Base64 encodes them.
2. Hash with SHA256 using the **secret key**.
3. Attach the hash as the **signature**.

When decoding, JWT re-computes the hash using the same key — if it matches, the token is valid.

⚖️ Summary Table

Feature	Session Authentication	JWT Authentication
Server-side storage	✅ Yes (session store)	❌ No (stateless)
Token type	Session ID	JWT (JSON Web Token)
Data location	Server	Inside the token itself
Scaling	Harder ،stateful)	Easier (stateless)
Logout	Easy (delete session)	Hard (token stays valid)
Use case	Web apps (browser cookies)	APIs, mobile apps, SPAs
Security	Safer for web	Needs careful handling (XSS risk)
Built-in in Django	✅ Yes (`SessionAuthentication`)	✅ Yes (`JWTAuthentication` in DRF SimpleJWT)

Using JWT in Django:

1. Step in: pip install PyJWT
2. Step 2: **How to Generate a Token and Decode it:**

```
import jwt
import datetime

# 1️⃣ Secret key (use your Django project's SECRET_KEY)
from django.conf import settings
SECRET_KEY = settings.SECRET_KEY

# 2️⃣ Data to include in token
data = {
    'user_id': 1,
    'username': 'ajay',
    'exp': datetime.datetime.utcnow() + datetime.timedelta(minutes=30),  # expiry
    'iat': datetime.datetime.utcnow()  # issued at
}

# 3️⃣ Create token
my_token = jwt.encode(data, SECRET_KEY, algorithm='HS256')

print("Generated Token:", my_token)

# 4️⃣ Decode token (to verify)
decoded = jwt.decode(my_token, SECRET_KEY, algorithms=['HS256'])
print("Decoded:", decoded)
```

9. Sending Token in Next Request

From frontend or Postman:

```sql
GET /jwt/profile/
Authorization: Bearer <your_token_here>
```

In Django:

```python
auth_header = request.headers.get('Authorization')
token = auth_header.split(' ')[1]
decoded = jwt.decode(token, settings.SECRET_KEY, algorithms=['HS256'])
```

If you want to write a view which handle both browser requests and mobile/API requests then you can write view like this:

1.

```python
def get_token_from_request(request):
    token = None
    # Check Header
    auth_header = request.headers.get('Authorization')
    if auth_header and auth_header.startswith('Bearer '):
        token = auth_header.split(' ')[1]
    # Check Cookie
    elif 'access_token' in request.COOKIES:
        token = request.COOKIES.get('access_token')
    return token
```