Authentication & Authorization in Chatbots

Authentication & Authorization in Chatbots

- What is Authentication & Authorization?
- Sessions & Cookies
- Tokens & JWT
- OAuth & SSO
- Authorization Models
- FastAPI Implementation
- LangChain & LangGraph & MCP
- Security Best Practices

Authentication & Authorization in Chatbots

- **Securing chatbot applications**
  - Learn how to **protect sensitive** user data and prevent unauthorized access to your AI-powered chatbot systems.

- **FastAPI, LangChain, LangGraph**
  - Explore authentication patterns across modern Python frameworks used for building production-ready chatbots.

- **Best practices for AI systems**
  - Discover industry-standard security practices specifically designed for AI and machine learning applications.

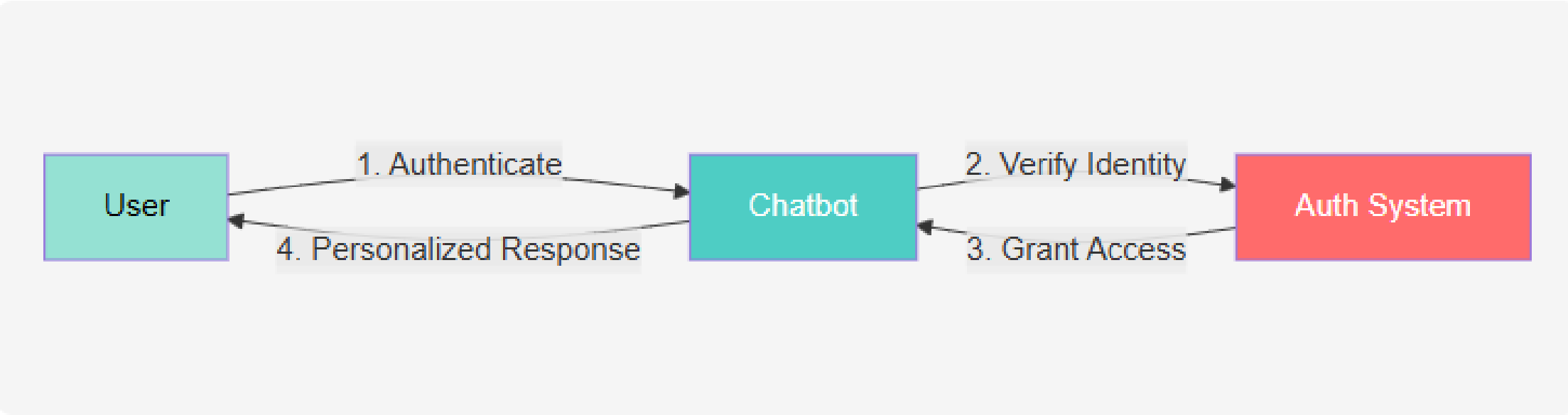
What is Authentication?

- **Verifying identity - 'Who are you?'**
  - Authentication is the process of confirming that someone is **who they claim to be**, like checking a driver's license or passport.

- **Like showing ID at hotel check-in**
  - Just as hotels verify your reservation before giving you a room key, systems verify your credentials before granting access.

- **Methods: passwords, tokens, biometrics**
  - Common authentication methods include traditional passwords, secure tokens (like JWT), and biometric data such as fingerprints or facial recognition.

Why This Matters for Chatbots



What is Authorization?

- **Granting permissions - 'What can you do?'**
  - Authorization determines what **actions an authenticated user is allowed to perform within the system, such as reading, writing, or deleting data**.

- **Like a keycard opening specific rooms**
  - After authentication, authorization acts like a hotel keycard that only opens certain doors based on your room assignment and guest status.

- **Controls access after authentication**
  - Even if you prove who you are, authorization ensures you can only access resources and perform actions you're permitted to use.

Why This Matters for Chatbots

- **Protect sensitive user data**
  - Chatbots often handle personal information, medical records, or financial data that must be kept secure from unauthorized access and breaches.

- **Enable personalized experiences**
  - Authentication allows chatbots to remember conversation history and user preferences, creating tailored responses for each individual user.

- **Comply with privacy regulations**
  - Proper authentication and authorization help meet legal requirements like GDPR, HIPAA, and other data protection laws.

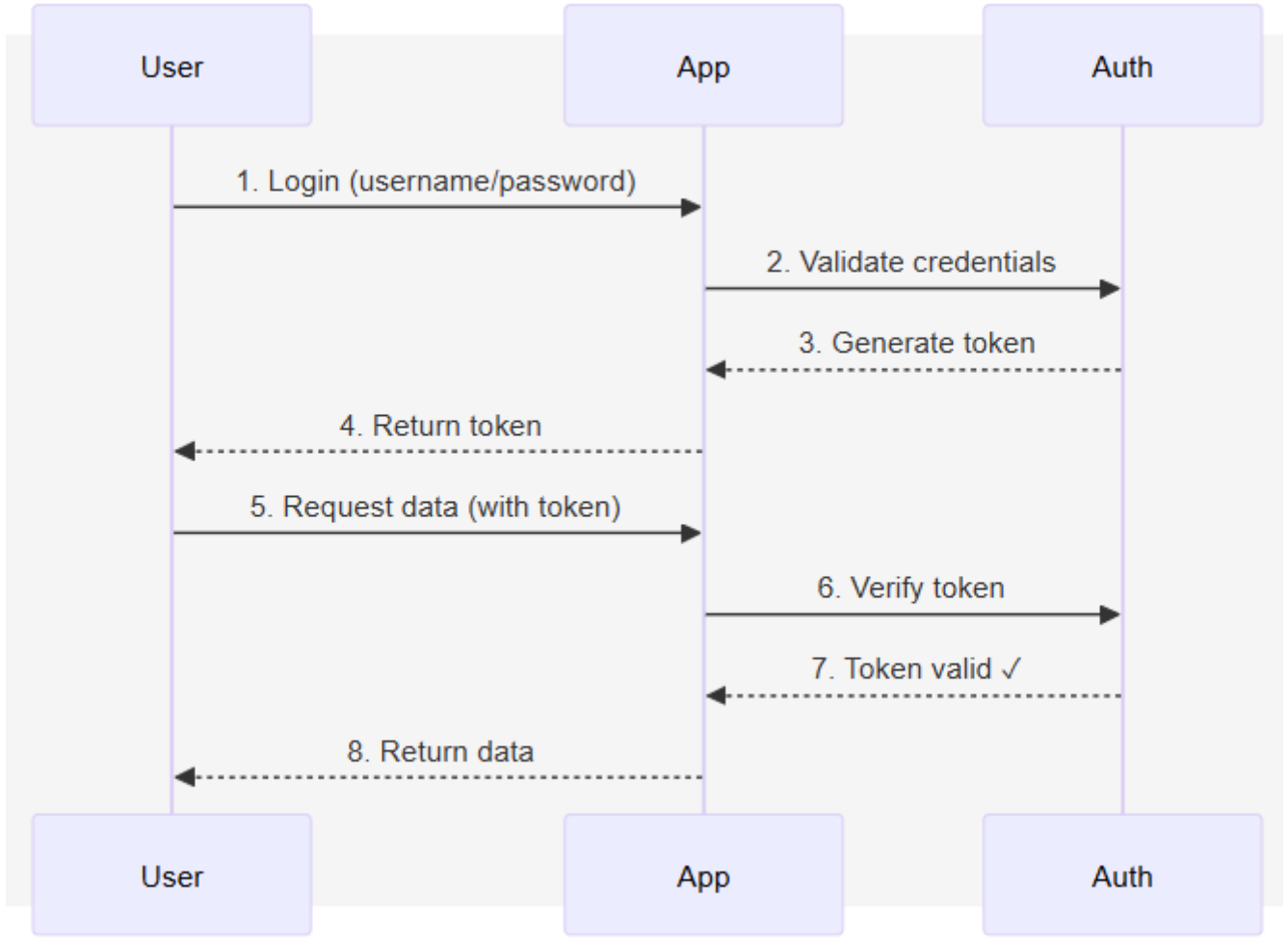
The Authentication Flow

- **User provides credentials**
  - The authentication process begins when a user submits their username and password or other identifying information to the system.

- **System validates and creates token**
  - The server verifies the credentials against stored records and generates a secure token (like JWT) that proves the user's identity.

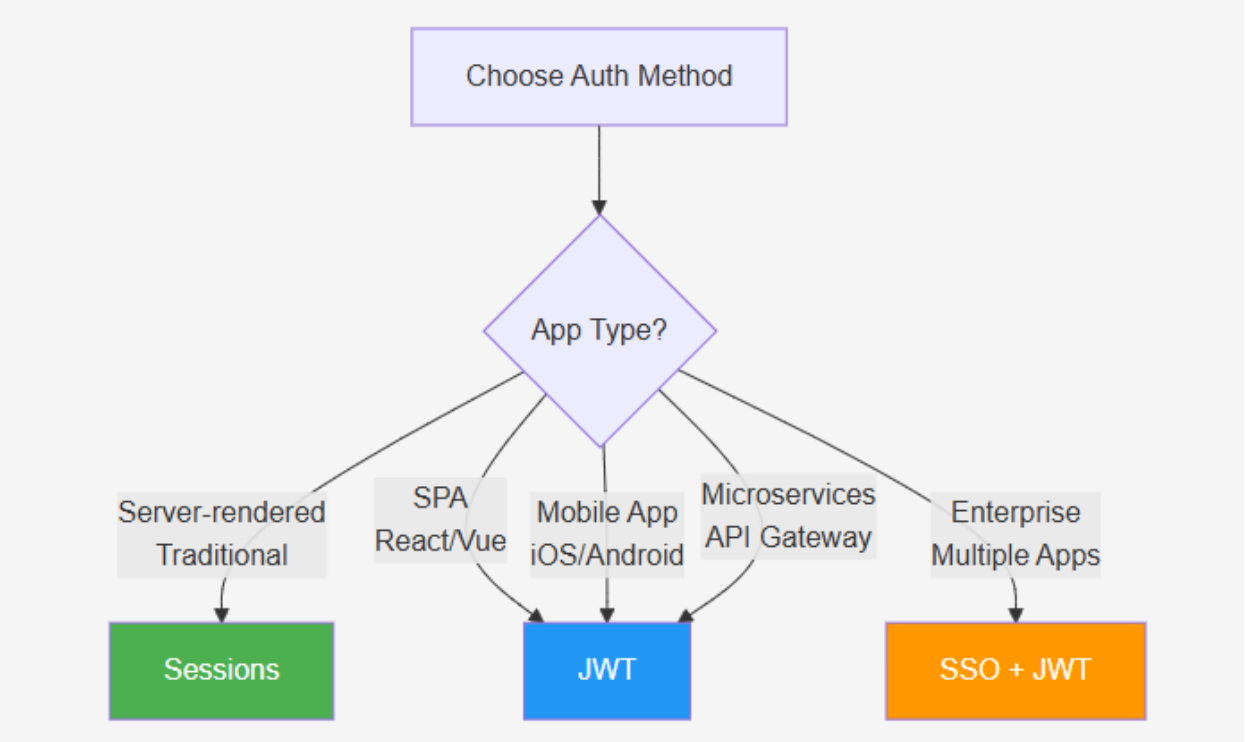
- **Token used for subsequent requests**
  - Instead of sending credentials repeatedly, the user includes this token with each request, allowing the server to quickly verify identity without database lookups.

The Authentication Flow



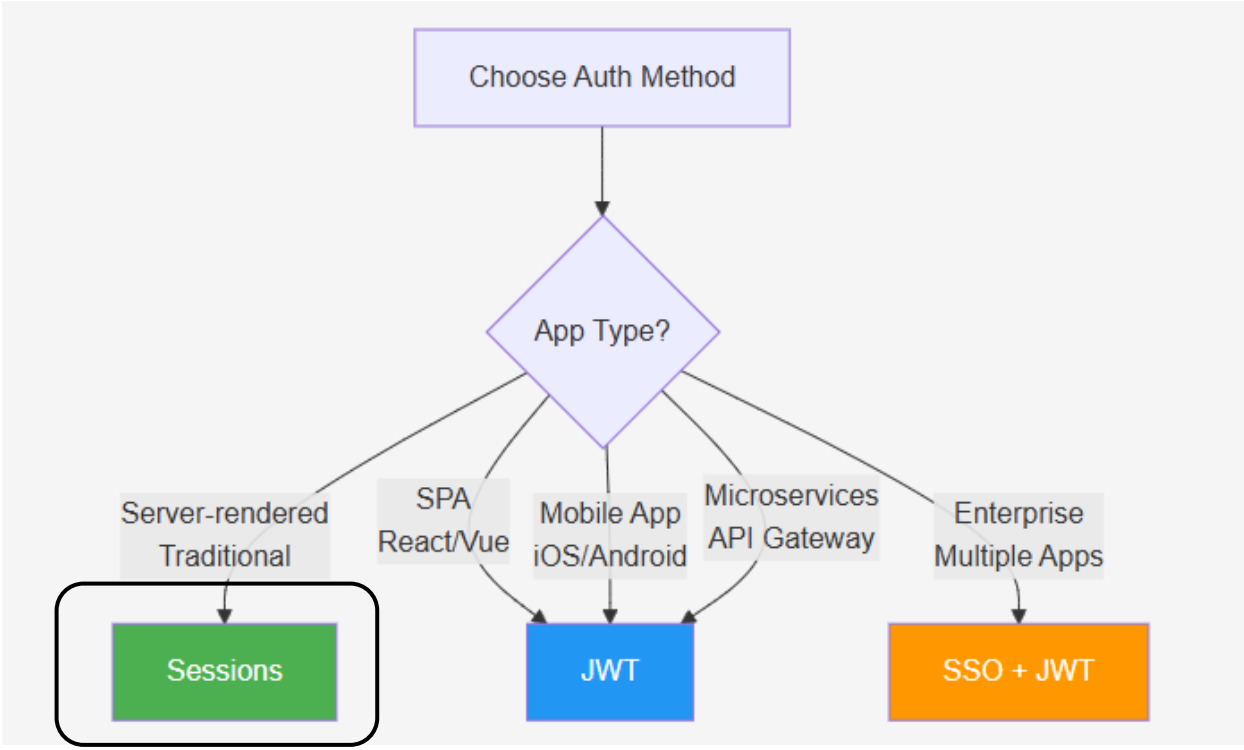
When to Use Each Method

- **JWT = Json Web Token**
- **Traditional web app → Sessions + Cookies**
  - Server-rendered applications with monolithic architectures benefit from sessions as they already maintain server-side state and use sticky load balancing.
- **SPA/Mobile/API → JWT Tokens**
  - Single-page applications, mobile apps, and REST APIs need stateless authentication that works across distributed servers and doesn't rely on cookies.
- **Microservices → JWT for stateless auth**
  - Microservices architectures require each service to independently verify authentication without sharing session state or querying a central database.



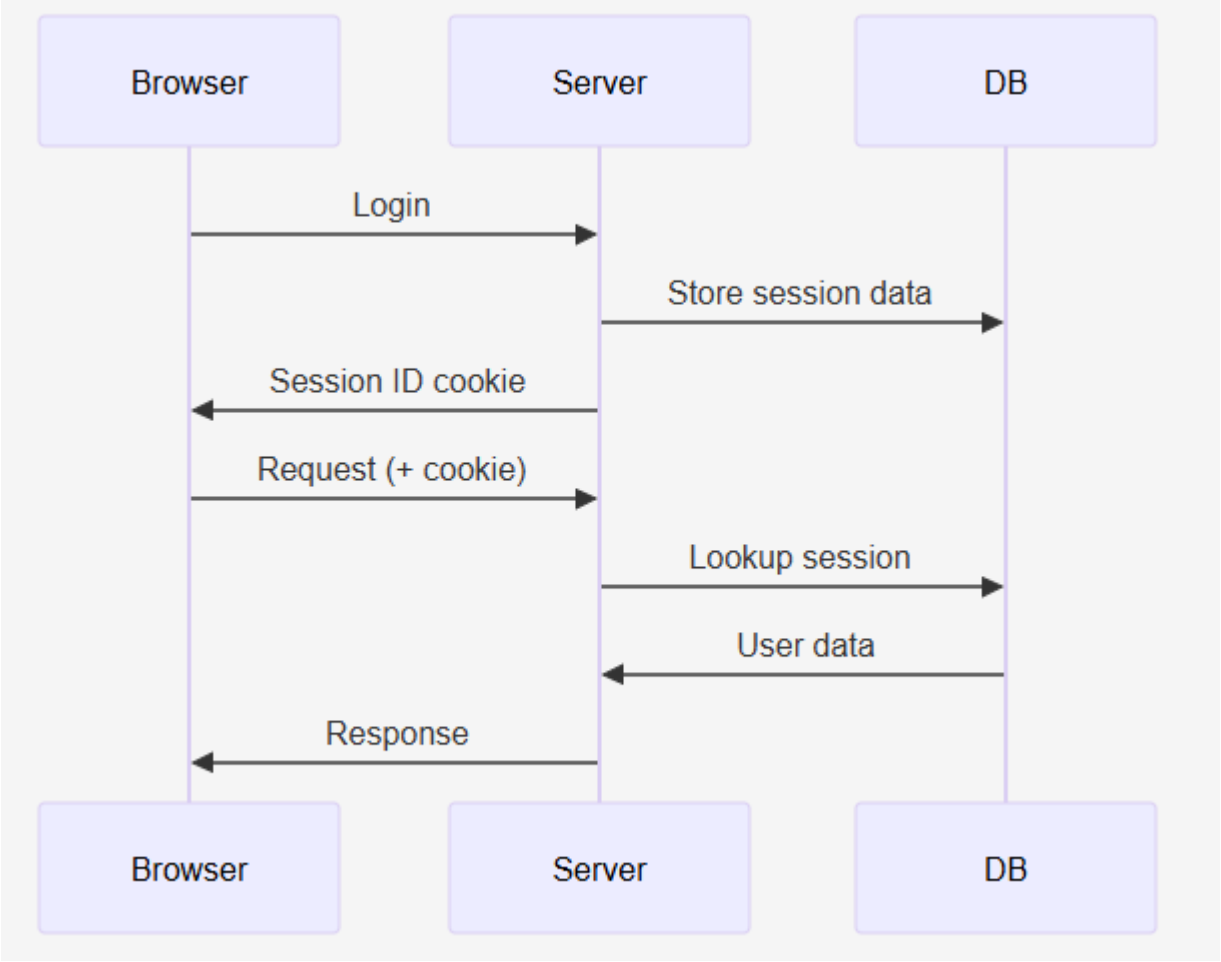
Sessions & Cookies

When to Use Each Method



Understanding Sessions

- **Server stores user data, browser gets ID**
  - The server maintains a session store containing all user information while sending only a lightweight session identifier to the client's browser.
- **Like hotel keycard - card has ID, hotel has your info**
  - Your hotel keycard only contains a room number (session ID), but the hotel's system has all your personal information, reservation details, and preferences.
- **Session ID stored in cookie**
  - Browsers automatically store the session ID in a cookie and include it with every request, allowing the server to identify and retrieve the user's session data.



Cookies Explained

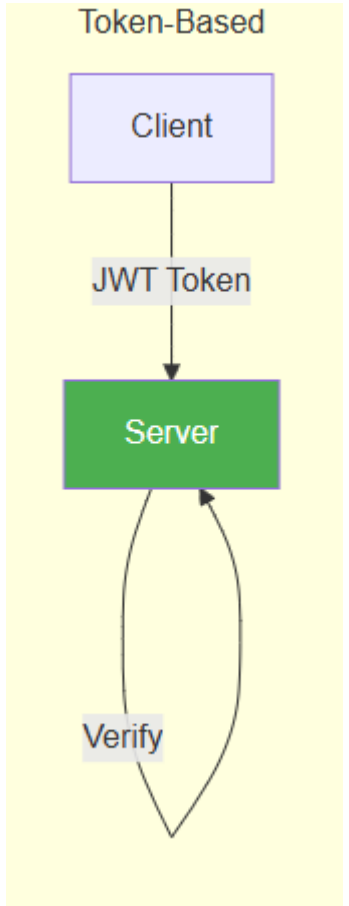
- **Small data stored in browser**
  - Cookies are tiny text files (typically under 4KB) that websites store on your computer to remember information between page visits.

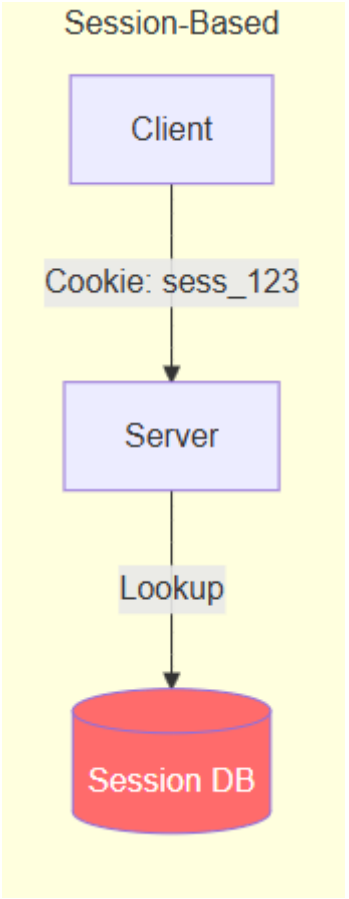
- **Like a claim ticket at coat check**
  - Just as a coat check gives you a numbered ticket to retrieve your coat, cookies give your browser an identifier to retrieve your session data from the server.

- **Automatically sent with requests**
  - Your browser automatically includes relevant cookies with each HTTP request to the website, allowing the server to recognize you without manual input.

Sessions vs Tokens Comparison

- **Sessions: Data on server, scalability issues**
  - Session-based authentication requires a centralized session store that all servers must access, creating a bottleneck and single point of failure in distributed systems.
- **Tokens (JWT): Data in token, scales infinitely**
  - JWT tokens enable horizontal scaling as any server can verify the token independently without shared state or database lookups, perfect for cloud-native architectures.
- **Choose based on your architecture needs**
  - Use sessions for traditional monolithic apps with sticky sessions, and JWTs for modern distributed systems, SPAs, mobile apps, and microservices.





Tokens

Sessions vs Tokens

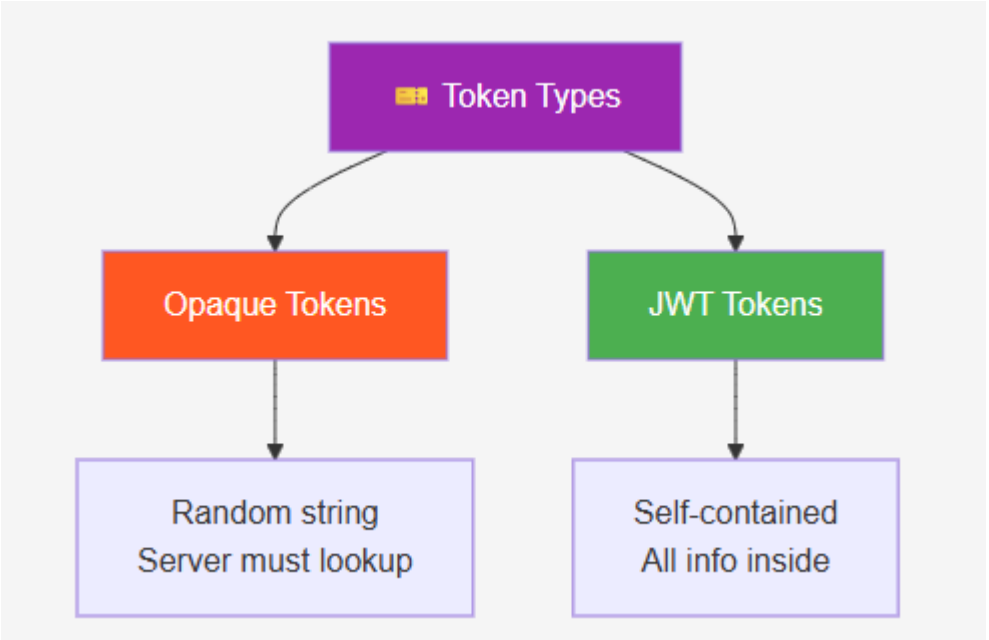
- **Sessions: Data stored on server**
  - Session-based authentication stores user data on the server and gives the browser only a session ID, requiring server lookups for each request.

- **Tokens: Self-contained, no server lookup**
  - Token-based authentication (like JWT) embeds all user information within the token itself, eliminating the need for server-side session storage.

- **Tokens scale better for APIs**
  - Tokens are stateless and can be verified independently by any server, making them ideal for distributed systems, microservices, and mobile applications.

What Are Tokens?

- **Self-contained proof of authentication**
  - Tokens embed all necessary authentication data (user ID, roles, expiration) within themselves, eliminating the need for server-side session storage.
- **Like a driver's license with your info**
  - Just as a driver's license contains your photo, name, and expiration date for verification without calling the DMV, tokens contain verifiable user information.
- **No server lookup needed to verify**
  - Servers can validate tokens by checking their cryptographic signature, enabling fast authentication without database queries for every request.



Opaque vs JWT Tokens

- **Opaque: Random string, requires database**
  - Opaque tokens are random identifiers that act as keys to look up user data in a database - the token itself contains no user information.
- **JWT: Contains user data, no database**
  - JWT tokens encode user data in base64 format within the token itself, allowing servers to extract user information without any database lookup.
- **JWT scales better for distributed systems**
  - Since JWTs are stateless and don't require database lookups, they excel in microservices architectures where multiple servers need to verify users independently.

```
# Opaque Token
token = "xK7mP9qR2vN8"  # Random string
user = db.lookup_token(token)  # Must query DB

# JWT Token
token = "eyJhbGc..."  # Contains user data
user = jwt.decode(token, verify=True)  # No DB
needed
```

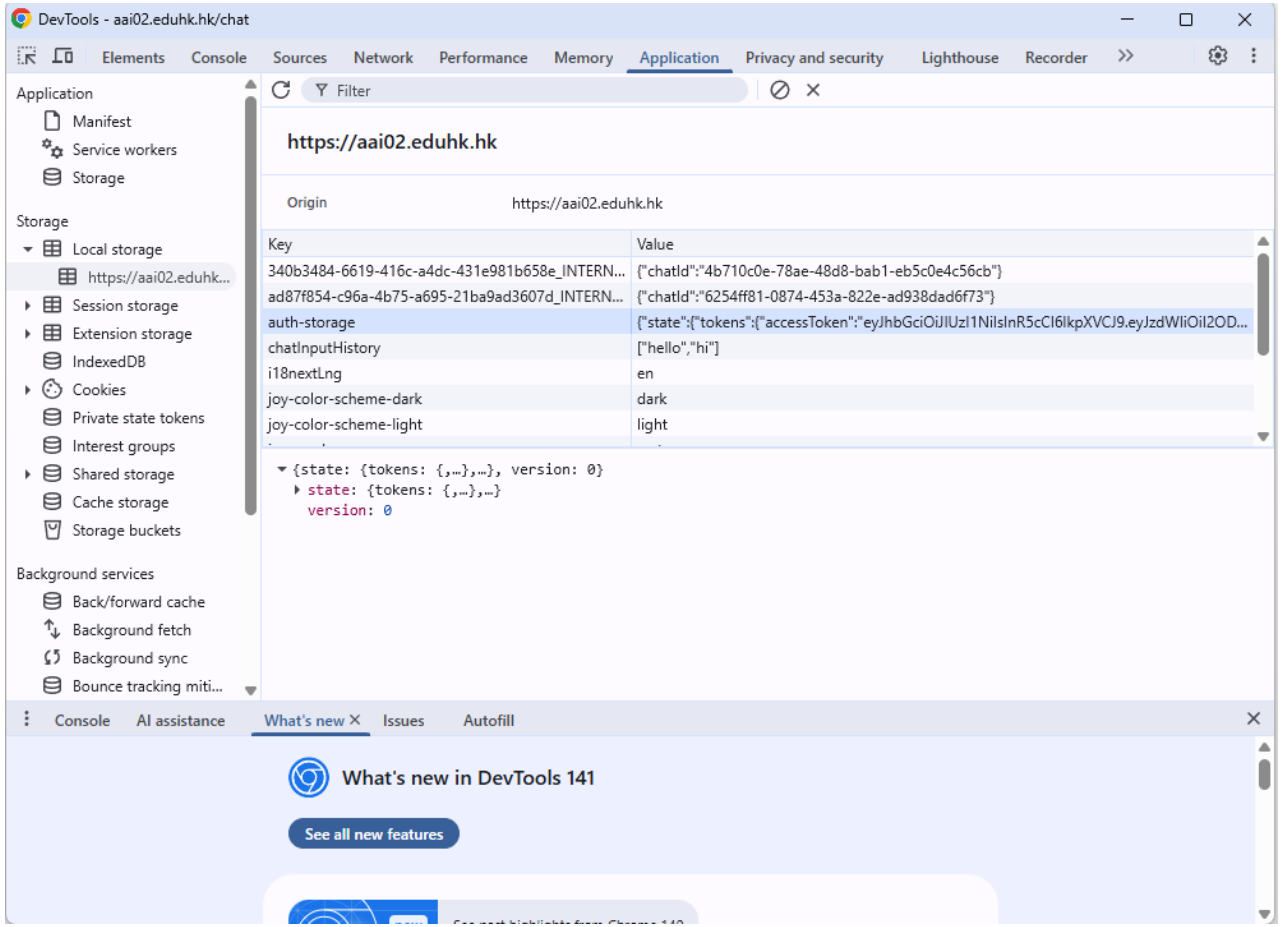
JWT (JSON Web Token)

- **Industry standard for tokens**
  - JWT has become the de facto standard for secure information exchange on the web, used by major platforms like Google, Facebook, and AWS.

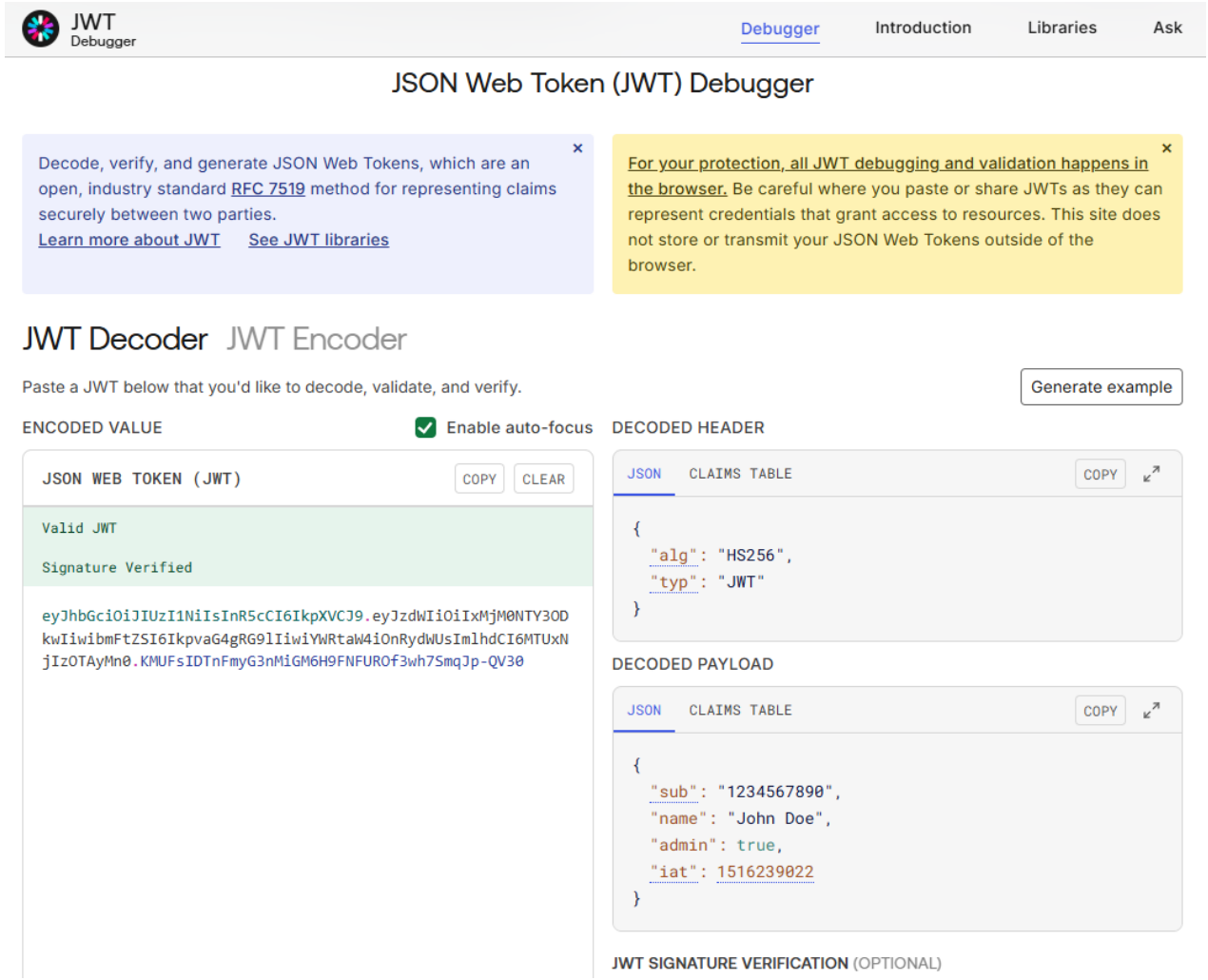
- **Three parts: Header, Payload, Signature**
  - A JWT consists of a header (algorithm info), payload (claims/data), and signature (verification), all base64-encoded and joined by dots.

- **Self-contained user information**
  - The token payload contains all necessary user data (ID, roles, permissions), allowing servers to make authorization decisions without database queries.

JWT (JSON Web Token)



https://www.jwt.io/



JWT (JSON Web Token)

- // JWT Structure
- eyJhbGc... (Header)
- .eyJzdWI... (Payload - user data)
- .SflKxw... (Signature - verification)

JWT Example

- **Contains user ID, name, expiration**
  - A JWT payload includes essential claims like 'sub' (subject/user ID), 'name', and 'exp' (expiration timestamp) to identify users and enforce time limits.

- **Cryptographically signed**
  - The signature is created using a secret key and ensures the token hasn't been tampered with - any modification invalidates the signature.

- **Verified without database lookup**
  - Servers can validate the token's signature and extract user data instantly without querying a database, significantly improving performance.

JWT Example

```python
import jwt

# Create token
token = jwt.encode({
  "sub": "user_123",
  "name": "Alice",
  "exp": datetime.now() + timedelta(minutes=30)
}, "secret-key", algorithm="HS256")

# Verify token
decoded = jwt.decode(token, "secret-key",
                     algorithms=["HS256"])
```

JWT Standard Claims

- **iss: Who created the token**
  - The issuer claim identifies the authentication server that created this token, allowing clients to verify the token came from a trusted source.
- **sub: User ID, exp: Expiration time**
  - The subject identifies the user, while expiration provides built-in security by ensuring tokens become invalid after a specific timestamp.
- **aud: Who should accept it**
  - The audience claim specifies which applications or APIs should accept this token, preventing tokens from being used on unintended services.

```
{
  "iss": "https://auth.example.com",
  "sub": "user_12345",
  "aud": "https://api.example.com",
  "exp": 1735689600,
  "iat": 1735686000,
  "name": "Alice",
  "roles": ["user", "premium"]
}
```

Is JWT used for authentication or authorization?

OAuth & SSO

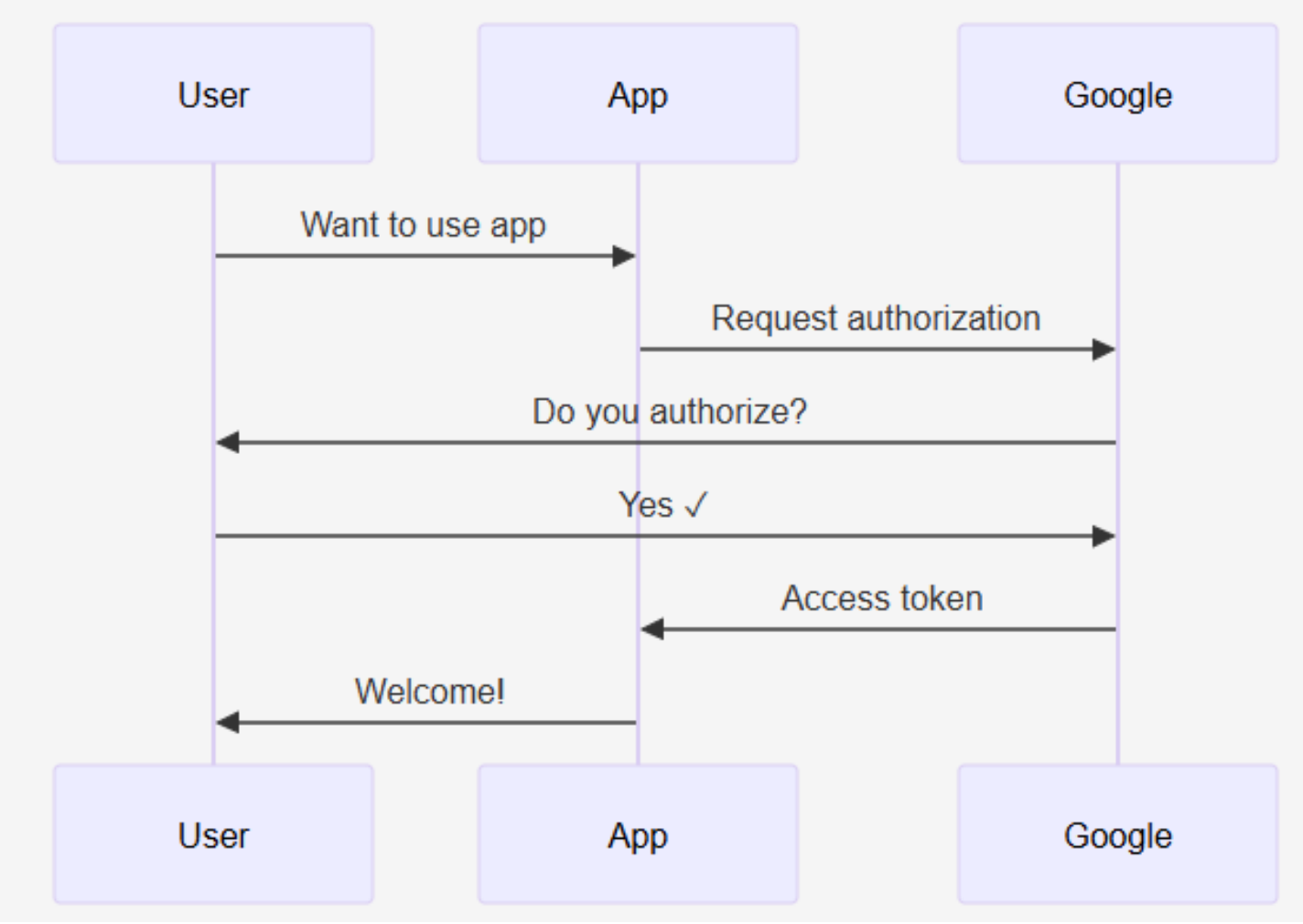
What is OAuth 2.0?

- **Authorization framework (not authentication)**
  - OAuth 2.0 is designed for granting access permissions rather than verifying identity - it lets apps access your data without sharing passwords.

- **Third-party access without passwords**
  - Allows external applications to access your resources (like Google Photos) without ever seeing or storing your actual password credentials.

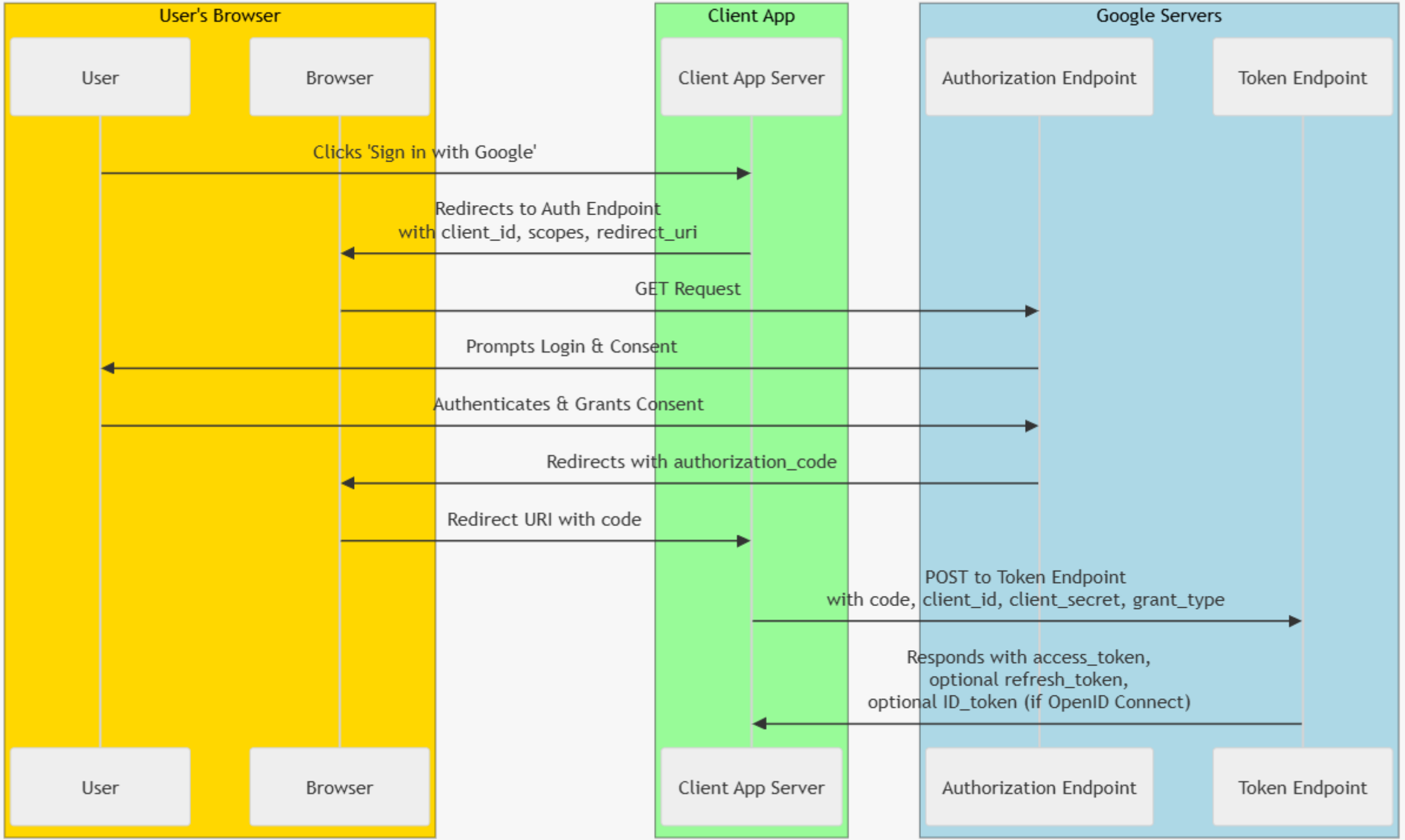
- **Example: 'Sign in with Google'**
  - When you click 'Sign in with Google', OAuth 2.0 lets the application access specific information from your Google account with your permission.

What is OAuth 2.0?



What is OAuth 2.0?

- Authorization Code Flow
  - App Initiates the Request:
    - The user clicks "Sign in with Google" in your app. Your app redirects the user's browser to Google's authorization endpoint with parameters like your client ID, requested scopes (e.g., access to email or calendar), and a redirect URI.
  - User Authenticates and Consents:
    - Google prompts the user to log in (if not already) and grant consent for the requested scopes. If approved, Google redirects the browser back to your app's redirect URI with an authorization code (a short-lived token) in the URL parameters.
  - App Exchanges Code for Tokens
    - Your app's server takes this authorization code and sends a POST request to Google's token endpoint, including the code, your client ID, client secret, and grant type. In response, Google sends back: An access token, a refresh token and an ID token


User's Browser
Client App
Google Servers
User
Browser
Client App Server
Authorization Endpoint
Token Endpoint
Clicks 'Sign in with Google'
Redirects to Auth Endpoint
with client_id, scopes, redirect_uri
GET Request
Prompts Login & Consent
Authenticates & Grants Consent
Redirects with authorization_code
Redirect URI with code
POST to Token Endpoint
with code, client_id, client_secret, grant_type
Responds with access_token,
optional refresh_token,
optional ID_token (if OpenID Connect)
User
Browser
Client App Server
Authorization Endpoint
Token Endpoint

What is OAuth 2.0?

- An access token
  - a string credential for accessing protected resources.
- A refresh token
  - to get new access tokens when the current one expires.

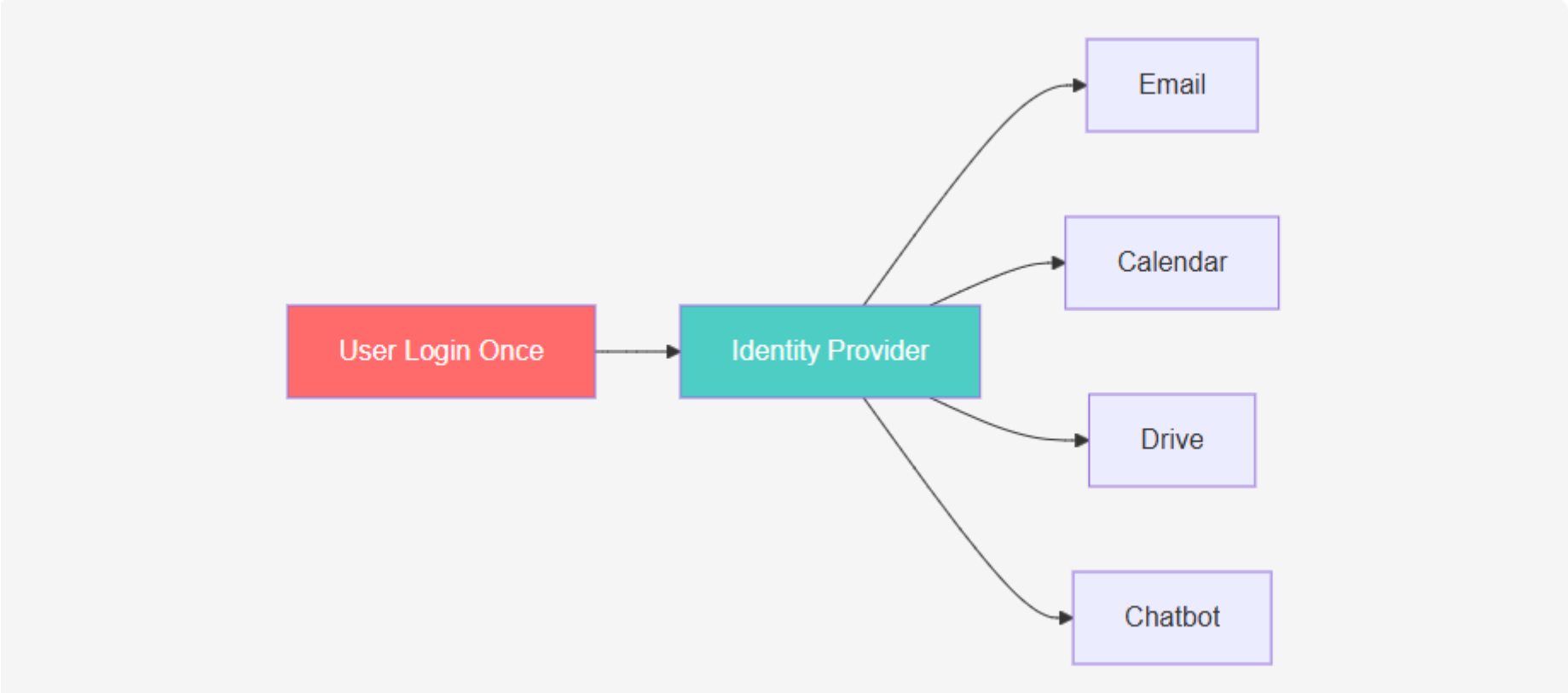
Single Sign-On (SSO)

- **Login once, access multiple apps**
  - SSO allows users to authenticate once with a central identity provider and gain access to all connected applications without re-entering credentials.

- **Example: Google account for Gmail, Drive, Calendar**
  - When you log into Gmail, you automatically get access to Google Drive, Calendar, and other Google services without logging in again.

- **Improves user experience and security**
  - Users manage fewer passwords (reducing password fatigue), while organizations gain centralized control over authentication and can enforce stronger security policies.

Single Sign-On (SSO)



Authorization Models

Authorization Models: RBAC vs ABAC

- **RBAC: Role-based (admin, user, guest)**
  - Role-Based Access Control assigns users to roles with predefined permissions, making it simple to manage access for common user types like administrators and regular users.
- **ABAC: Attribute-based (time, location, age)**
  - Attribute-Based Access Control evaluates multiple user attributes and context (location, time of day, device type) to make dynamic access decisions.
- **Permissions: Granular control per action**
  - Permission-based systems grant specific capabilities (like 'chatbot:write' or 'data:delete') independent of roles, allowing fine-grained access control.

```
# RBAC - Role-Based
if user.role == "admin":
    allow_access()

# ABAC - Attribute-Based
if user.age >= 18 and user.country == "US":
    allow_access()

# Permissions
if "chatbot:write" in user.permissions:
    allow_send_message()
```

Authorization in Practice

- **Start simple with roles (admin, user)**
  - Begin with basic role-based access control to quickly establish access tiers - most applications only need 2-3 roles to start.
- **Add permissions for fine control**
  - Layer on specific permissions as your application grows, allowing nuanced control like separating 'read' from 'write' access within the same role.
- **Use ABAC for complex rules**
  - Implement attribute-based rules when you need context-aware decisions, such as allowing data access only during business hours or from specific locations.

```python
@app.post("/admin/delete")
async def delete_data(user: User =
Depends(get_user)):
    # Check role
    if user.role != "admin":
        raise HTTPException(403, "Admin only")

    # Check permission
    if "data:delete" not in user.permissions:
        raise HTTPException(403, "No delete
permission")

    # Perform action
    delete_all_data()
```

FastAPI Implementation

FastAPI Authentication

- **Built-in security tools**
  - FastAPI provides ready-to-use security utilities like OAuth2PasswordBearer, APIKeyHeader, and HTTPBasic for implementing authentication with minimal code.

- **OAuth2 password flow support**
  - FastAPI has native support for the OAuth2 password flow, making it easy to implement username/password authentication with JWT tokens.

- **Easy JWT integration**
  - Works seamlessly with popular JWT libraries like python-jose, allowing you to create and verify tokens with just a few lines of code.

FastAPI Authentication

```python
from fastapi import FastAPI, Depends
from fastapi.security import OAuth2PasswordBearer

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

@app.get("/users/me")
async def get_user(token: str = Depends(oauth2_scheme)):
    return verify_token(token)
```

API Key Authentication

- **Simplest authentication method**
  - API keys provide a straightforward way to authenticate requests by including a secret key in the request header, ideal for server-to-server communication.

- **API key in request header**
  - The API key is typically sent as a custom HTTP header (like 'X-API-Key') with each request, keeping it separate from the URL for better security.

- **Good for service-to-service**
  - Perfect for backend services, microservices, and automated tools that need to authenticate without user interaction or complex OAuth flows.

API Key Authentication

```python
from fastapi.security import APIKeyHeader

api_key_header = APIKeyHeader(name="X-API-Key")

@app.get("/chatbot/ask")
async def ask(
    question: str,
    api_key: str = Security(api_key_header)
):
    if api_key not in VALID_KEYS:
        raise HTTPException(401, "Invalid API Key")
    return {"answer": "Hello!"}
```

OAuth2 Scopes (Permissions)

- **Fine-grained access control**
  - Scopes allow you to define specific permissions (like 'read', 'write', 'delete') that can be granted independently, giving precise control over what each user can do.

- **Example: read vs write permissions**
  - A user might have 'chatbot:read' scope to view messages but not 'chatbot:write' scope to send messages, enabling role-based restrictions within your API.

- **Token contains allowed scopes**
  - The JWT token includes a list of approved scopes in its payload, allowing the server to enforce permissions without additional database queries.

OAuth2 Scopes (Permissions)

```
oauth2_scheme = OAuth2PasswordBearer(
    tokenUrl="token",
    scopes={
        "chatbot:read": "Read messages",
        "chatbot:write": "Send messages",
        "admin": "Admin access"
    }
)
```

LangChain & LangGraph & MCP

LangChain Authentication

- **Secure API keys in environment variables**
  - Store sensitive keys like OPENAI_API_KEY in .env files or environment variables to prevent accidental exposure in version control systems.

- **Never hardcode credentials**
  - Hardcoding API keys directly in code is a major security risk - they can be exposed through GitHub, logs, or error messages, leading to unauthorized access and billing issues.

- **User-specific conversation history**
  - Implement authentication to maintain separate conversation memories for each user, enabling personalized responses and protecting user privacy.

LangChain Authentication

```
# WRONG - Never do this!
llm = ChatOpenAI(api_key="sk-proj-abc123...")

# CORRECT - Use environment variables
from dotenv import load_dotenv
load_dotenv()

llm = ChatOpenAI()  # Reads from OPENAI_API_KEY
```

Personalized Chatbot with LangChain

- **Each user gets own conversation memory**
  - By storing separate ConversationBufferMemory instances per user ID, the chatbot remembers previous exchanges for each individual user independently.
- **Authentication identifies the user**
  - FastAPI's Depends(get_current_user) middleware extracts and verifies the user from their JWT token, ensuring each request is properly authenticated.
- **Responses tailored to user context**
  - The AI can reference previous conversations, user preferences, and historical context to provide more relevant and personalized answers.

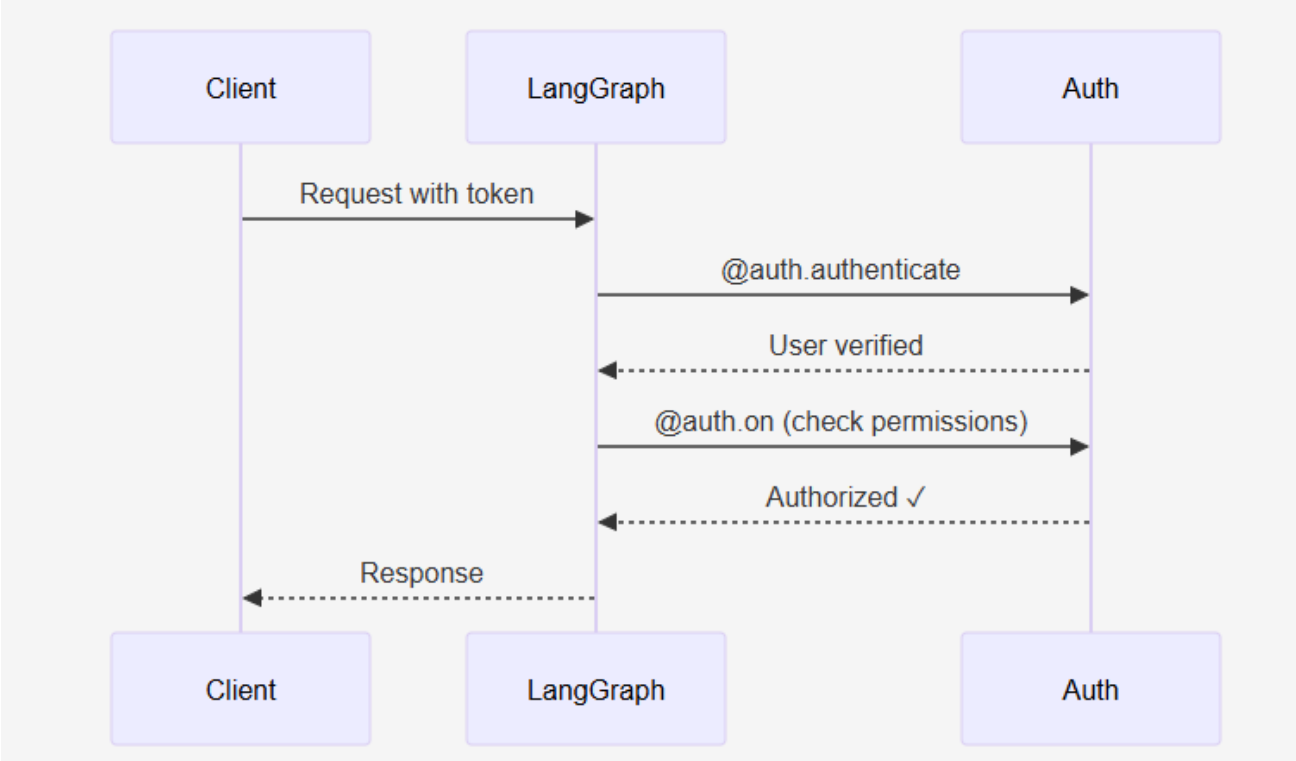
```
user_memories = {}

@app.post("/chat")
async def chat(message: str, user: User =
Depends(get_current_user)):
    if user.id not in user_memories:
        user_memories[user.id] =
ConversationBufferMemory()

    chain = ConversationChain(llm=llm,
memory=user_memories[user.id])
    return chain.predict(input=message)
```

LangGraph Platform Authentication

- **Stateful multi-actor applications**
  - LangGraph enables complex AI applications with multiple agents and persistent state, requiring robust authentication to manage user sessions and data access.
- **Built-in auth handlers**
  - LangGraph provides decorators like @auth.authenticate and @auth.on to implement authentication and authorization logic without custom middleware.
- **Resource-level authorization**
  - Control access to specific resources (threads, messages, runs) ensuring users can only access their own data or data they're explicitly authorized to view.



LangGraph Auth Handler

- **@auth.authenticate - verify user identity**
  - This decorator intercepts every request to validate the authorization token and extract user information before any endpoint logic executes.
- **@auth.on - control resource access**
  - The @auth.on decorator runs before accessing specific resources (like threads or messages) to enforce ownership rules and permission checks.
- **Filter by user ownership**
  - Automatically filter database queries to only return resources owned by the authenticated user, preventing unauthorized data access.

```python
from langgraph_sdk import Auth

auth = Auth()

@auth.authenticate
async def authenticate(authorization: str):
    token = parse_bearer_token(authorization)
    if token not in VALID_TOKENS:
        raise HTTPException(401, "Invalid
token")
    return {"identity": get_user_id(token)}
```

LangGraph Authorization

- **Users only see their own threads**
  - Authorization filters ensure that regular users can only view and interact with conversation threads they created, maintaining data privacy and isolation.
- **Admins can access all resources**
  - Users with admin permissions bypass ownership filters, allowing them to view all threads for moderation, support, or analytics purposes.
- **Automatic metadata filtering**
  - LangGraph automatically applies authorization filters to database queries based on user metadata, eliminating the need for manual permission checks in every endpoint.

```python
@auth.on.threads.read
async def authorize_read(ctx: AuthContext,
value: dict):
    if "admin" in ctx.permissions:
        return True  # Admins see all
    else:
        # Users see only their threads
        return {"owner": ctx.user.identity}
```

Model Context Protocol (MCP)

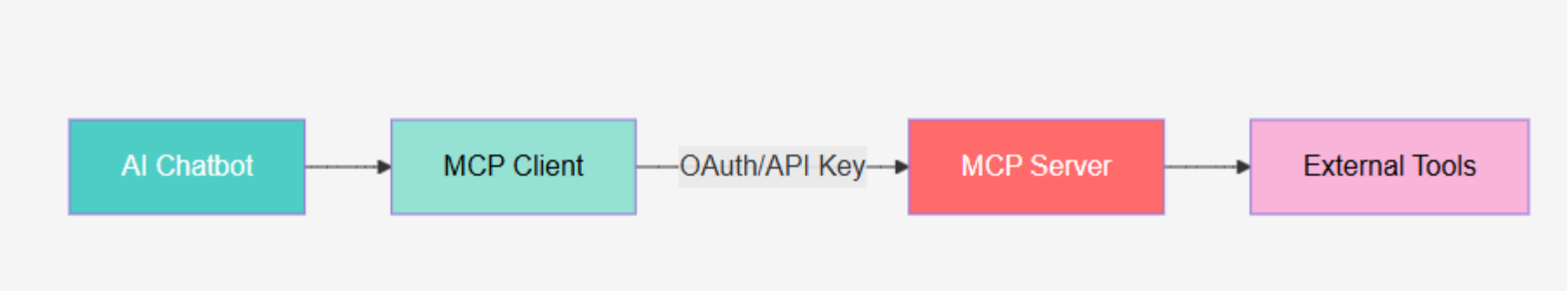
- **New standard for AI tool connections**
  - MCP is an emerging protocol that standardizes how AI applications connect to external tools, databases, and services with consistent authentication patterns.

- **Secure bridge between chatbot and services**
  - Acts as a secure intermediary layer that handles authentication, authorization, and communication between your AI chatbot and third-party services.

- **OAuth 2.1 and API key support**
  - Supports modern authentication standards including OAuth 2.1 (the updated OAuth spec) and traditional API keys for backward compatibility.

MCP Architecture

- **AI Host (chatbot) initiates requests**
  - Your chatbot application acts as the host, making requests to MCP clients when it needs to interact with external tools or data sources.
- **MCP Client handles authentication**
  - The MCP client manages all authentication logic, token refresh, and credential storage, abstracting complexity from the AI application.
- **MCP Server performs actions securely**
  - The MCP server validates credentials, enforces permissions, and executes requested actions on external systems while maintaining security boundaries.



Security Best Practices

- **Always use HTTPS in production**
  - HTTPS encrypts all data in transit, protecting sensitive information like passwords and tokens from interception by attackers using man-in-the-middle attacks.

- **Set short token expiration (15-30 min)**
  - Short-lived tokens minimize the damage if a token is stolen - attackers have a limited window to use it before it expires and becomes useless.

- **Never store passwords in plain text**
  - Always hash passwords using strong algorithms like bcrypt or argon2 before storing them, so even database breaches don't expose user passwords.

JWT Security Tips

- **Use strong algorithms (RS256, HS256)**
  - RS256 (asymmetric) and HS256 (symmetric) are industry-standard algorithms for JWT signatures - avoid weak or deprecated algorithms like 'none'.
- **Validate all claims (exp, iss, aud)**
  - Always verify the token hasn't expired (exp), came from a trusted issuer (iss), and is intended for your application (aud) to prevent token replay and substitution attacks.
- **Store tokens securely (httpOnly cookies)**
  - Use httpOnly cookies to store tokens in the browser - this prevents JavaScript from accessing them, protecting against XSS (cross-site scripting) attacks.

```
# Validate JWT properly
try:
    payload = jwt.decode(
        token,
        SECRET_KEY,
        algorithms=["HS256"]
    )
    if payload.get("exp") < time.time():
        raise TokenExpired()
except jwt.InvalidTokenError:
    raise HTTPException(401, "Invalid token")
```

Secure Chatbot

- **FastAPI backend with JWT auth**
  - Combine FastAPI's OAuth2PasswordBearer with JWT tokens to create a secure API that authenticates users and protects chatbot endpoints from unauthorized access.
- **LangChain for AI responses**
  - Use LangChain's ChatOpenAI and ConversationChain to generate intelligent responses while maintaining conversation context and memory for each user.
- **User-specific conversation history**
  - Store separate conversation memories per user ID, ensuring privacy, personalization, and the ability to resume conversations across sessions.

```python
@app.post("/chat")
async def chat(
    message: str,
    user: User = Depends(get_current_user)
):
    # Get user's conversation
    conv = get_user_conversation(user.id)

    # Generate response
    response = conv.predict(input=message)

    return {
        "user": user.username,
        "response": response
    }
```

Security DO's

- **Always use HTTPS in production**
  - HTTPS encrypts all data in transit, preventing attackers from intercepting sensitive information like passwords, tokens, and personal data.
- **Hash passwords with bcrypt/argon2**
  - Never store plain text passwords - use industry-standard hashing algorithms that are designed to be slow and resistant to brute-force attacks.
- **Set short token expiration (15-30 min)**
  - Limiting token lifetime reduces the window of opportunity for attackers if a token is compromised, forcing periodic re-authentication.

```
#  DO - Use environment variables
SECRET_KEY = os.getenv("SECRET_KEY")

#  DO - Hash passwords
hashed = pwd_context.hash(password)

#  DO - Short expiration
exp = datetime.now() + timedelta(minutes=30)
```

Security DON'Ts

- **Never hardcode secrets in code**
  - Hardcoded secrets in source code will be exposed in version control, build artifacts, and to anyone with code access - always use environment variables.
- **Don't store passwords in plain text**
  - Storing unhashed passwords means a single database breach exposes all user credentials, allowing attackers to access user accounts across multiple sites.
- **Never skip token verification**
  - Disabling JWT signature verification allows attackers to forge tokens and impersonate any user - always verify tokens with your secret key.

```
#  DON'T - Hardcode secrets
SECRET_KEY = "my-secret-123"  # NEVER!

#  DON'T - Plain text passwords
user.password = "password123"  # NEVER!

#  DON'T - Skip verification
payload = jwt.decode(token, verify=False)  #
NEVER!
```

Common Web Vulnerabilities

- **SQL Injection: Use parameterized queries**
  - SQL injection occurs when user input is directly concatenated into SQL queries - always use parameterized queries or ORMs to prevent attackers from executing malicious SQL.
- **XSS: Sanitize user input always**
  - Cross-Site Scripting allows attackers to inject malicious scripts into web pages - escape and sanitize all user input before rendering it in HTML.
- **CSRF: Use SameSite cookies**
  - Cross-Site Request Forgery tricks browsers into making unwanted requests - set SameSite=Strict on cookies to prevent cross-origin cookie submission.

❌

```
query = f"SELECT * FROM users WHERE id =
'{user_id}'"

# GOOD
query = "SELECT * FROM users WHERE id = %s"
cursor.execute(query, (user_id,))

# CSRF Prevention
response.set_cookie("token", value,
samesite="strict")
```

AI Chatbot Security

- **Prompt injection: Validate/sanitize input**
  - Attackers can manipulate AI responses by injecting malicious prompts - validate user messages, use system prompts, and implement content filtering before sending to LLMs.
- **Rate limiting: Prevent API abuse**
  - Without rate limits, attackers can abuse your expensive AI API endpoints or overwhelm your system – implement per-user request throttling and quotas.
- **Token theft: Use httpOnly cookies**
  - JavaScript cannot access httpOnly cookies, protecting authentication tokens from XSS attacks that steal credentials to hijack user sessions.

```
# Prompt injection prevention
dangerous = ["ignore previous", "system:",
"admin:"]
for pattern in dangerous:
    if pattern in user_input.lower():
        raise ValueError("Invalid input")

# Rate limiting
if request_count > MAX_REQUESTS_PER_MINUTE:
    raise HTTPException(429, "Too many
requests")
```

Summary: Authentication Methods

- **Sessions: Server-side storage, traditional**
  - Session-based authentication stores user data on the server and works well for traditional server-rendered applications with sticky sessions.
- **Tokens (JWT): Self-contained, modern APIs**
  - JWT tokens carry user information within themselves, making them perfect for stateless REST APIs, microservices, and mobile applications.
- **OAuth 2.0: Third-party authorization**
  - OAuth 2.0 enables secure third-party access delegation, allowing users to grant limited permissions to apps without sharing passwords.

Key Takeaways

- **Authentication verifies identity**
  - Authentication answers 'Who are you?' by validating credentials (password, biometric, token) to confirm a user is who they claim to be.
- **Authorization controls permissions**
  - Authorization answers 'What can you do?' by checking if an authenticated user has permission to access specific resources or perform certain actions.
- **Use tokens (JWT) for scalable APIs**
  - JWT tokens enable stateless authentication that scales horizontally without shared session storage, making them ideal for modern distributed systems and microservices.

Take home exercise

- **Implement FastAPI authentication**
  - Start building your secure API by implementing OAuth2 with password flow in FastAPI, using JWT tokens for stateless authentication across endpoints.

- **Secure your LangChain API keys**
  - Store OpenAI and other API keys in environment variables, use key rotation strategies, and implement rate limiting to protect against abuse.

- **Build user-specific chatbot features**
  - Create personalized experiences by storing per-user conversation history, preferences, and context using the authenticated user's ID.