



Authentication

Importance of Secure Authentication

- Authentication is the initial line of defense in any application.
- Discuss issues like unauthorized access, data breaches, legal implications.
- It's always about balancing security and user experience.

What is a token?

- A token is typically a long string generated by an authentication provider.
- If often contains additional information such as the id of the user who requested the token, an expiry date, or other useful data needed for authorization.
- JSON Web Tokens are commonly used for this.

Where do you store a token?

- Cookies!
 - Local/session storage is not the place for sensitive data.
- Cookies are easy to implement, stateful. They can be set on the server side.
- Cons: Vulnerable to CSRF attacks (**Cross-site request forgery**), potential for misuse.
- Secure flag, HttpOnly, same-site attributes, CSRF tokens.

Basic Email/Password Flow

- Most common form of authentication, using email and password.

Pros and Cons

- Pros: Simplicity, widespread user familiarity.
- Cons: Vulnerable to brute-force attacks, phishing.

Best Practices

- Strong password requirements, two-factor authentication, secure hashing.

Federated Login

- Using third-party identity providers like Google, Facebook, etc.
- Built on OAuth - explain the standard OAuth flow and show with an example.

Pros and Cons

- Pros: Easier for users, potentially more secure.
- Cons: Third-party dependency, privacy concerns.

Best Practices

- Ensure secure OAuth implementation, always validate tokens server-side.

Magic Links

- Login via a one-time clickable link sent via email.

Pros and Cons

- Pros: No need for users to remember passwords.
- Cons: Dependent on email security.

Best Practices

- Limit the validity period, make sure links are single use.

Passkeys

- Passkeys are a new authentication method that removes the need for using passwords as well as additional security layers such as 2FA.
- It relies on a mechanism using a combination of a private key and a public key. The private key is stored on an authentication device (i.e. a phone) and is used to sign a challenge.
- The public key is stored at the service you want to sign in to. This key is used to verify that a challenge was correctly signed.
- Passkey creation can happen either when a new user registers for an account, or when an authenticated user decides to create a passkey for their existing account in

the profile settings. Passkey creation is triggered on the client (e.g. "Create a passkey" button), initiating a three-step sequence.

- The client then calls the WebAuthn API with `credentials.create(challenge)`, handing over the challenge, which results in a key pair – a private and a public key – being created on the device (usually a phone or computer, called "Authenticator" in the WebAuthn context). The private key (i.e. the passkey) is used to sign the challenge and stored on the user's device inside secure hardware like a TPM or secure enclave. The matching public key, a credential ID, as well as the signed challenge are returned by the function. The user may get prompted for verification with a biometric or a PIN during this step.
- To finalize passkey registration, the client now sends the public key, the credential ID, and the signed challenge back to the server. The server associates the public key and the credential ID with the user for future authentication flows and checks the validity of the operation by verifying the signed challenge with the public key.
- Different to a password that has to be sent over to the authenticating server, a passkey does not leave the user's device to sign in the user. Instead, the passkey is used locally to create a unique cryptographic signature that proves to the server that it has been created with the passkey. The sequence for passkey authentication is, again, triggered by the user on the client ("Sign in with a passkey" button):
- The client requests a random challenge from the server.
- The client calls the WebAuthn API with `credentials.get(challenge)`, which results in prompting the user for verification with a biometric or a PIN and using a stored passkey to sign the challenge. The signed challenge as well as the username and the credential ID of the used passkey are returned by the function.
- To finalize passkey authentication, the client sends the signed challenge, the username, and the credential ID back to the server. The server now checks if the username and credential ID are known and, if so, picks up the public key to check the validity of the signed challenge. If the signed challenge can be successfully verified by the public key, the server can safely assume that the user was in possession of the passkey and create a session or hand out an authorization token.

API keys

- API key-based authentication is the most basic form of API security.
- An API key is a unique identifier used to authenticate a user, developer, or calling program to an API.
- API keys are best for projects where security demands are not very high, but some level of control is needed.
- API keys are passed along with the API request, usually in the request header.
- While API keys are more secure than no authentication, they are not foolproof and should be kept confidential.
- You would normally only use this for communicating between servers. You don't want to store an API key as part of the client code because then that would give access to anyone who can see the client code (which, if it's a web app, is everyone).

Client Credentials Flow (OAuth)

- Bearer tokens are a variety of simple API keys: they can be dynamic and therefore offer more security.
- You would typically obtain a bearer token through a separate authentication request.
- Used for server-to-server interaction.
- Allows clients to obtain an access token outside of the context of a user.
- The flow is very simple: you start with a client ID and secret that you get from the API service you want to access.
- To get an access token, send the client ID and secret to an auth API endpoint. This generates the token which you can then use to access the API.
- The token is called a “bearer” token, that you then include in the header.
- Since the token is generated dynamically, it’s safer to use since your requests will only include a (temporary) token, instead of a key that always allows access.
- Optionally, you can work with a refresh token on top of this flow. So, the access token expires at some point. You can then request a new token using the refresh token.

Summary and Key Takeaways

- Multiple ways to authenticate users, each with its own pros and cons.
- Crucial to balance security with usability.
- Always adhere to best practices for the chosen authentication method.
- Keep an eye on passkeys as I expect this is going to replace the other login methods over time.