===========================================================API AUTHENTICATION MERGE DEV===========================================================

**Basic Auth** -> A simple and fast method of HTTP authentication. To access the API endpoint, the user must send a username and password to the API provider in the authentication header of the request. The API provider checks the credentials and, in the case of success, grants access to the user.

**API Key Authentication** -> The API provider assigns a unique key to each client accessing the API. The client needs to include their API key as part of the request to authenticate themselves. The API key can be included anywhere in the request. Different permissions can be given to different API Keys. Security is lost if the user's API Key is intercepted as the interceptor can use it to access the same resources as the user.

**Token-based authentication** -> It uses an access token to verify a user's identity. Tokens are used to identify users, since they have user-specific info. Tokens can be generated dynamically within the API itself. They can also be customized to make them device specific. They minimize interception risk since they can be encrypted and are usually short-lived, and they can offer granular access control.

However, tokens are difficult to design.

===========================================================API AUTHENTICATION MERGE DEV===========================================================

============================================================STACK-OVERFLOW AUTHENTICATION AUTHORIZATION===========================================

1) USE TLS -> Use TLS for encrypting communications. TLS requires a certificate issued by a CA. Most hosting providers will provide a TLS certificate automatically.

2) Use Single Sign On (SSO) with Google, Meta, etc. using OAuth2 -> It is done by letting users exchange a token with the verification services of these companies using OAuth2. This avoids us having to manage passwords and accounts, not have to manage MFA and not store user data.

OAuth2 is a standard that describes how a 3rd-party application can access data from an application on behalf of a user. OAuth2 doesnâ€™t directly handle authentication and is a more general framework built primarily for authorization.

We should use OpenID Connect to implement authentication on top of OAuth.

3) Use API keys to give existing users programmatic access -> Issue API Keys, for each REST API to authenticate using OAuth2.

4) API Key Best Practices

\* When a user signs up for access to your API, generate an API key

\* All API Keys should be stored with associated users in the database.

\* Have your users provide their API keys as a header

\* To authenticate a user's API request, look up their API key in the database.

5) All API Keys should be stored safely

6) Use request-level-authorization, if appropriate, i.e., looking at an incoming request to decide if the user has access to your resources or not.

7) Configure different permissions for different API keys -> Consider having several API keys with different permission levels. To do this, store permissions in the database alongside the API keys as a list of strings. Keep this simple at first: "read" and "write" are a great start! Then, add a request middleware that fetches the user and the permissions for the key they've submitted and checks the token permissions against the API.

8) Leave the rest of the authorization to the business logic -> Instead, leave that level of authorization logic to your application code. Any authorization checks made on resources should happen in the app, not in the middleware. If you need to handle complex authorization logic in your app, use a tool like Oso, which will let you reduce your authorization policy to a few simple rules.

9) Use Good libraries -> Try to offload as much work as you can to trusted libraries.

============================================================STACK-OVERFLOW AUTHENTICATION AUTHORIZATION===========================================

============================================================NODEJS EXPRESS AUTHENTICATION CONCEPTS - APIDOG=======================================

Link: https://apidog.com/blog/node-js-express-authentication/

**Authentication** -> The process of verifying the identity of a user. (Who are you?)

**Authorization** -> The process of determining what an authenticated user is allowed to do. (What can you do?)

**Stateless Authentication** -> The server does not maintain any session data. Each request must contain all authentication data (through tokens).

**Stateful Authentication** -> The server maintains session data for authenticated users (in a DB or in RAM). The client holds a session ID to access the stored session data.

**Basic Authentication** -> Users provide their username and password for each request. Simple and quick to implement. Suitable for basic applications and internal tools. Implemented using base64 encoding and decoding.

**Token-based Authentication (JWT)** -> Users authenticate by receiving a JSON Web Token (JWT) after logging in, which they include in the header of subsequent requests. Used in modern web apps & mobile apps. Eg - "jsonwebtoken", "express-jwt", etc.

**Session-based Authentication** -> User credentials are stored in a session on the server after login. The session ID is stored in a cookie on the client side. Traditional web applications where server-side sessions are manageable. Eg - "express-session", "connect-mongo", etc.

**OAuth2** -> Delegated authorization framework that allows third-party services to exchange tokens on behalf of the user. Integration with Google, Meta, etc. Used where easy access to those services (Google, Meta, GitHub) is needed. Eg - "passport", "passport-google-oauth20", "passport-facebook", "passport-github"

**MFA** -> Adds an extra layer of security by requiring multiple forms of verification. Used in high-security applications where additional authentication layers are necessary. Eg - "speakeasy" (for OTP gen), "node-2fa", etc.

**API-Key Authentication** -> Users include an API key in the request header for each API call. Commonly used for service-to-service communication or for public APIs. No specific library.

**LDAP Authentication** -> Authenticates users against a directory service like Microsoft Active Directory. Used in enterprise applications where centralized authentication is required. Eg - "passport-ldapauth", "ldapjs", etc.

**SAML Authentication** -> Security Assertion Markup Language (SAML) is an XML-based protocol for exchanging authentication and authorization data between parties. Used in Enterprise single sign-on (SSO) solutions. Eg - "passport-saml".

============================================================NODEJS EXPRESS AUTHENTICATION CONCEPTS - APIDOG=======================================

| **Authentication Method** | **Use Cases** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| **Basic Authentication** | - Internal tools or prototypes- Quick setup scenarios | - Simple to implement- No additional libraries needed | - Not secure for sensitive data- Requires HTTPS for minimal security |
| **Token-Based Authentication (JWT)** | - SPAs (React, Angular, Vue.js)- Mobile apps- Microservices | - Stateless- Cross-domain support | - Token storage is challenging- Token revocation is complex |
| **Session-Based Authentication** | - Traditional server-rendered web apps- Persistent user sessions | - Centralized session management- Easy with frameworks | - Requires server-side storage- Less scalable |
| **OAuth2** | - Third-party integrations (Google, GitHub)- Delegated authorization | - Secure and widely adopted- No password sharing | - Complex to set up- Depends on third-party providers |
| **Social Login** | - Consumer-facing apps- Reduce login friction | - Easy for users- No need to manage passwords | - Dependency on social media- Potential privacy concerns |
| **Multi-Factor Authentication (MFA)** | - High-security apps (banking, healthcare)- Compliance-heavy industries | - Strong security boost- Works with other auth methods | - More complex to manage- Can reduce user convenience |
| **API Key Authentication** | - Service-to-service communication- Public APIs for third parties | - Simple and manageable- Easy to revoke | - Less secure (keys can leak)- No user-specific context |
| **LDAP Authentication** | - Enterprise apps with centralized user directories- Internal tools for employee access | - Centralized user control- Good corporate integration | - LDAP setup required- Can be complex to implement/debug |
| **SAML Authentication** | - Enterprise SSO- Federated identity across organizations | - Secure and standardized- Enables cross-app SSO | - Complex to configure- Requires supporting infrastructure |