API

An **API**, which stands for **Application Programming Interface**, is a set of rules, protocols, and tools that allow different software applications to communicate with each other. It acts as an intermediary, processing requests and ensuring that different systems can work together seamlessly.

Think of an API like a **waiter in a restaurant**. 👨‍🍳

- You (the **client application**) want to order food. You don't go directly into the kitchen (the **server system**) to prepare your meal.
- Instead, you look at the menu (the **API documentation**) to see what you can order.
- You give your order (an **API request**) to the waiter (the **API**).
- The waiter takes your request to the kitchen, which then prepares the meal.
- The waiter brings the food (the **API response**) back to you.

The API simplifies things by hiding the complex internal workings of the kitchen (the server) from you (the client). You only need to know how to make a valid request.

Types of APIs

APIs can be categorized based on their availability and intended audience.

1. **Private (Internal) APIs**:

   These APIs are designed for use **only within a company**. They are used to connect a company's own internal systems and applications, improving efficiency and communication between different teams or departments. They are not exposed to any third-party developers.

   - **Example**: An API used by a company's mobile banking app to communicate with its internal customer database.
2. **Partner APIs**:

   These are shared with **specific business partners** but are not available to the public. Access is restricted and requires specific licenses or permissions. They are used to facilitate business-to-business (B2B) integrations.

   - **Example**: Uber might provide a partner API to a restaurant chain, allowing the restaurant's app to directly request a ride for a customer.
3. **Public (Open) APIs**:

   These APIs are publicly available for **any third-party developer** to use. They often have extensive documentation and allow other businesses to build applications that leverage the API provider's data or services.

    - **Example**: The **Google Maps API**, which allows developers to embed Google Maps, add markers, and calculate routes within their own websites and applications.

API Architectural Styles

An architectural style is a set of constraints and principles used to design an API. It defines how the API is structured and how it communicates.

1. REST (Representational State Transfer)

REST is the most popular architectural style for designing web services. It's not a protocol but a set of guiding principles for creating scalable, simple, and reliable web APIs.

- **Key Concepts**:
    - **It is Stateless**: Every request from a client to the server must contain all the information needed to process it. The server does not store any information about the client's session between requests.
    - **It is Client-Server Architecture**: The client (front-end) and server (back-end) are separate and evolve independently.
    - **It has Uniform Interface**: Resources (like a user or a product) are identified by URIs (e.g., `/users/123`). Standard HTTP methods are used to perform actions on these resources:
        - `GET`: Retrieve a resource.
        - `POST`: Create a new resource.
        - `PUT`/`PATCH`: Update an existing resource.
        - `DELETE`: Remove a resource.
    - **It uses commonly used Data Format** like **JSON** (JavaScript Object Notation) but can also use XML, HTML, or plain text.

2. SOAP (Simple Object Access Protocol)

SOAP is a formal protocol with a stricter set of rules compared to REST. It was more popular before REST but is still widely used in enterprise environments, especially for systems that require high security and reliability.

- **Key Concepts**:
    - **It is a Protocol, Not a Style**: It has a rigid standard for how messages are structured.
    - **It is XML-Based**: All messages are formatted in XML.
    - **It** Can be used for both stateful and stateless operations.
    - **It** has built-in standards for security (WS-Security) and error handling, which makes it robust for enterprise applications.

3. GraphQL

GraphQL is a modern query language for APIs developed by Facebook. It gives clients the power to ask for exactly what they need and nothing more.

- **Key Concepts**:
    - It solves the problem of **over-fetching** (getting more data than needed) and **under-fetching** (having to make multiple API calls to get related data) that can occur with REST.
    - **It has Single Endpoint**: Unlike REST, which uses multiple endpoints for different resources, GraphQL typically exposes a single endpoint.
    - **It supports Client-Driven Queries**: The client specifies the structure of the data it requires in a query, and the server returns a JSON response that matches that exact structure.

4. RPC (Remote Procedure Call)

RPC is one of the simplest API styles where the client executes a function (or procedure) on a remote server as if it were a local call.

- **Key Concepts**:
    - **It is Action-Oriented**: The client calls a specific method on the server (e.g., `getUser(userId: 123)`).
    - **It is Simple and Performant**: It's very straightforward. Modern implementations like **gRPC** (Google's RPC) use protocol buffers for serializing data, which is highly efficient and performant.
    - **It** Can use various formats like JSON (JSON-RPC) or Protobuf (gRPC).

API Components

APIs are built from several fundamental components that work together to handle communication between a client and a server.

- **Endpoint**: This is the specific URL where an API can be accessed. Each endpoint corresponds to a specific function or resource. For example, `https://api.example.com/v1/users` is an endpoint to get a list of users.
- **Request**: A message sent by the client to the server to ask for information or perform an action. A request typically includes:
    - **HTTP Method**: The type of action to perform (e.g., `GET` to retrieve data, `POST` to create data, `PUT` to update data, `DELETE` to remove data).
    - **Headers**: Metadata containing information like authentication tokens and the format of the request body (e.g., `Content-Type: application/json`).
    - **Body**: The actual data sent from the client to the server, usually in JSON format. This is used in `POST` and `PUT` requests.
- **Response**: The message sent back from the server to the client after processing the request. A response includes:

    - **Status Code**: A three-digit code indicating the outcome. Common codes include:
        - `200 OK`: The request was successful.
        - `201 Created`: A new resource was successfully created.
        - `400 Bad Request`: The server couldn't understand the request.
        - `401 Unauthorized`: The client is not authenticated.
        - `404 Not Found`: The requested resource could not be found.
        - `500 Internal Server Error`: A generic error on the server.
    - **Body**: The data requested by the client, often in JSON format.

API Design Concepts

Good API design focuses on making the API easy to use, consistent, and predictable for developers. This is often referred to as creating a good **Developer Experience (DX)**.

- **Use Nouns for Endpoints**: Endpoints should represent resources (nouns), not actions (verbs). The HTTP method should define the action.
- **Use Plural Nouns**: Keep your endpoints consistent by using plural nouns for collections of resources.
- **Versioning**: Plan for future changes by including a version number in your API path. This prevents breaking changes for existing users when you update the API.
- **Consistent Naming and Structure**: Use a consistent naming convention (like `camelCase` or `snake_case`) for your JSON fields and parameters. The structure of your responses should also be predictable.
- **Provide Filtering, Sorting, and Pagination**: For APIs that return large lists of data, allow clients to:
    - **Filter**: Narrow down results (e.g., `/orders?status=shipped`).
    - **Sort**: Order the results (e.g., `/products?sort=price_desc`).
    - **Paginate**: Retrieve data in manageable chunks instead of all at once (e.g., `/articles?page=2&limit=25`).

API Lifecycle

The API lifecycle refers to the entire lifespan of an API, from its conception to its retirement. Managing this lifecycle ensures that an API is developed, deployed, and maintained in a structured and efficient way.

1. **Planning and Design**: The initial phase where the API's purpose is defined, requirements are gathered, and the API contract (endpoints, data models) is designed.
2. **Development**: The stage where the API is actually coded and built according to the design specifications.

3. **Testing**: A critical phase where the API is rigorously tested for functionality, performance, security, and reliability.
4. **Deployment**: The API is deployed to a server environment (like staging for final tests, and then production for public use).
5. **Publishing**: The API is made available to developers, and its documentation is published. This often involves listing it in an API marketplace or developer portal.
6. **Monitoring**: Continuously monitoring the API for uptime, latency, error rates, and security threats.
7. **Versioning**: As the API evolves, new versions are created (e.g., `/v2/`) to introduce new features or breaking changes without disrupting existing users of the older version.
8. **Retirement**: When an old version is no longer supported, it is gracefully retired after notifying users and providing them a clear path to migrate to a newer version.

API Testing

API testing focuses on verifying the **business logic, functionality, security, and performance** of the API, independent of any user interface.

- **Functional Testing**: Checks if the API endpoints work as expected. For example, does a `POST` request to `/users` correctly create a new user in the database?
- **Validation Testing**: Ensures the API correctly handles invalid input. For example, what happens if a client sends a text string for a field that expects a number? The API should return an appropriate error code (like `400 Bad Request`).
- **Performance Testing**: Measures how the API performs under stress. This includes:
    - **Load Testing**: Simulating a high number of concurrent users to see how the API handles the traffic.
    - **Stress Testing**: Pushing the API beyond its normal capacity to find its breaking point.
- **Security Testing**: Probes the API for vulnerabilities. This involves checking for improper authentication, data exposure, and susceptibility to attacks like SQL injection.

API Monitoring & Analytics

API Monitoring is the real-time observation of an API's health and performance. **API Analytics** is the analysis of historical usage data to gain business insights.

Key metrics to monitor include:

- **Uptime / Availability**: Is the API online and accessible? (e.g., 99.9% uptime).
- **Latency (Response Time)**: How long does it take for the API to process a request and send a response?

- **Error Rate**: What percentage of API calls are failing? A spike in the error rate can indicate a problem.
- **Traffic Volume**: The number of requests the API is handling over time. This helps in capacity planning.
- **Most Used Endpoints**: Identifying which features of your API are most popular to guide future development.

Real-World API Examples

- **Stripe API**: A financial API that allows businesses to easily integrate payment processing (credit cards, bank transfers) into their websites and applications.
- **Google Maps API**: A mapping API that lets developers embed maps, get directions, search for locations, and use satellite imagery.
- **Twilio API**: A communications API that enables applications to programmatically make and receive phone calls, send and receive text messages, and manage other communication functions.
- **GitHub API**: Allows developers to interact with GitHub repositories, users, and issues, enabling the creation of tools that integrate with the GitHub workflow.

Protocols Used in APIs

A protocol is a set of rules governing the exchange of data. While people often associate APIs with REST over HTTP, several protocols are used.

- **HTTP/HTTPS (Hypertext Transfer Protocol/Secure)**: The foundation of the web and the most common protocol for RESTful and GraphQL APIs. It uses a request-response model. HTTPS is the encrypted, secure version and is considered essential.
- **TCP (Transmission Control Protocol)**: A core protocol of the Internet Protocol (IP) suite. It ensures reliable, ordered, and error-checked delivery of data between applications. HTTP runs on top of TCP.
- **WebSocket**: A protocol that provides a persistent, two-way communication channel between a client and a server over a single TCP connection. It's ideal for real-time applications like live chat, notifications, or online gaming.
- **AMQP (Advanced Message Queuing Protocol)**: A protocol for message-oriented middleware. It is used for asynchronous, message-based communication between different services, often in complex microservices architectures.

API Gateway

An **API Gateway** is a management tool that acts as a single entry point for all client requests to your backend services. Instead of clients calling multiple different microservices directly, they

make a single call to the API Gateway, which then intelligently routes the request to the appropriate service.

Think of it as a **receptionist** in a large office building. You don't need to know the exact room number for every person. You just go to the front desk (the gateway), say who you want to see, and the receptionist (the gateway) directs you, after handling security checks.

Key Functions:

- **Request Routing**: Directs incoming traffic to the correct backend service.
- **Authentication & Authorization**: Centralizes security, ensuring only authorized clients can access the APIs.
- **Rate Limiting & Throttling**: Protects backend services from being overwhelmed by too many requests.
- **Logging & Monitoring**: Provides a single place to log and monitor all API traffic.
- **SSL/TLS Termination**: Manages HTTPS encryption and decryption, offloading this work from the backend services.

Microservices Communication

Microservices can communicate with each other in two primary ways:

1. **Synchronous Communication**:

   The client service sends a request and **waits for an immediate response**.

   - **Method**: Typically done using protocols like **HTTP/REST** or **gRPC**.
   - **Pros**: Simple to understand and implement.
   - **Cons**: Creates tight coupling. If the service being called is slow or unavailable, the calling service is blocked, which can lead to cascading failures.
2. **Asynchronous Communication**:

   The client service sends a message and **does not wait for an immediate reply**. Communication happens through an intermediary message broker.

   - **Method**: Uses a **Message Queue** (like RabbitMQ) or an **Event Broker** (like Apache Kafka).
   - **Pros**: Decouples services, increasing resilience and scalability. If a service is down, messages can queue up and be processed when it's back online.
   - **Cons**: More complex to implement and requires managing a message broker.

Serverless APIs

Serverless APIs are built using a "serverless" computing model where you run your code without provisioning or managing any servers. The cloud provider handles all the infrastructure management, scaling, and maintenance.

A common pattern is using **Amazon API Gateway + AWS Lambda**.

1. The **API Gateway** provides an HTTP endpoint.
2. When a client hits that endpoint, the API Gateway triggers an **AWS Lambda function**.
3. The Lambda function (which contains your business logic) executes, processes the request, and returns a response.

You only pay for the compute time you consume when your code is running, and it scales automatically from zero to thousands of requests.

Rate Limiting Algorithms

Rate limiting is used to control the number of requests a client can make to an API within a certain time frame. This protects the API from abuse and ensures fair usage.

1. **Token Bucket**:
    - **Analogy**: A bucket is filled with tokens at a constant rate. To make a request, you must take a token from the bucket.
    - **Behavior**: If the bucket has tokens, requests can be made. This allows for **bursts** of traffic, as a client can use up all available tokens at once. If the bucket is empty, requests are rejected until new tokens are added.
2. **Leaky Bucket**:
    - **Analogy**: A bucket has a hole and leaks at a constant rate. Incoming requests fill the bucket.
    - **Behavior**: Requests are processed at the steady rate of the "leak." This algorithm **smooths out** traffic into a constant flow. It does not allow for bursts; if requests arrive too quickly, the bucket overflows, and requests are discarded.

REST vs SOAP

This is a classic comparison between two ways of building APIs.

- **REST (Representational State Transfer)** is an **architectural style**, not a rigid protocol. It's lighter, more flexible, and uses standard HTTP methods (`GET`, `POST`, `PUT`, `DELETE`). It commonly uses JSON for data exchange and is the most popular choice for modern web and mobile applications.
- **SOAP (Simple Object Access Protocol)** is a **formal protocol** with a strict set of rules. It uses XML for all its messages and has built-in standards for complex operations like

security (WS-Security) and transactions. It's generally considered heavier and more complex than REST and is often found in enterprise or legacy systems.

Analogy: REST is like sending a postcard—it's simple, flexible, and uses the standard postal system. SOAP is like sending a notarized legal document in a sealed envelope—it has a strict format, built-in security, and more overhead.

REST vs GraphQL

This comparison is about how clients request data from an API.

- **REST** uses **multiple endpoints** to expose different resources (e.g., `/users`, `/users/{id}/posts`). This can lead to:
  - **Over-fetching**: Getting more data than you need (e.g., getting the full user object when you only need the name).
  - **Under-fetching**: Having to make multiple API calls to get all the data you need (e.g., one call to get the user, then another to get their posts).
- **GraphQL** uses a **single endpoint** and acts as a query language for your API. The client sends a query specifying *exactly* the data fields it needs, and the server returns a JSON object with just that data. This solves the over-fetching and under-fetching problem by giving control to the client.

Analogy: REST is like ordering from a fixed menu at a restaurant. GraphQL is like a build-your-own-bowl bar where you pick exactly which ingredients you want.

Status Codes and Their Meaning

HTTP status codes are standard server responses to a client's request. They are grouped into five classes:

- **2xx (Success)**: The request was successfully received, understood, and accepted.
  - **200 OK**: The request succeeded.
  - **201 Created**: The request succeeded, and a new resource was created.
  - **204 No Content**: The server successfully processed the request but is not returning any content.
- **4xx (Client Error)**: The request contains bad syntax or cannot be fulfilled.
  - **400 Bad Request**: The server cannot process the request due to a client error (e.g., malformed JSON).
  - **401 Unauthorized**: The client must authenticate itself to get the requested response.
  - **403 Forbidden**: The client does not have access rights to the content.

    - **404 Not Found**: The server cannot find the requested resource.
- **5xx (Server Error)**: The server failed to fulfill a valid request.
    - **500 Internal Server Error**: A generic error message, given when an unexpected condition was encountered on the server.
    - **503 Service Unavailable**: The server is not ready to handle the request (e.g., it's down for maintenance or overloaded).

Idempotent Methods

In the context of HTTP, an operation is **idempotent** if making the same request multiple times has the same effect on the server as making it just once.

- **Idempotent Methods**:
    - `GET`: Retrieving data doesn't change it.
    - `PUT`: Updating a resource to a specific state. Running it again with the same data won't change the state further.
    - `DELETE`: Deleting a resource. The first call deletes it; subsequent calls do nothing new (the resource is still gone).
- **Non-Idempotent Method**:
    - `POST`: Used to create a new resource. Making multiple `POST` requests will create multiple new resources.

Why it matters: Idempotency allows clients to safely retry requests after a network error without the risk of creating duplicate entries or performing an action multiple times.