Application Programming Interfaces

(APIs)

What is an API

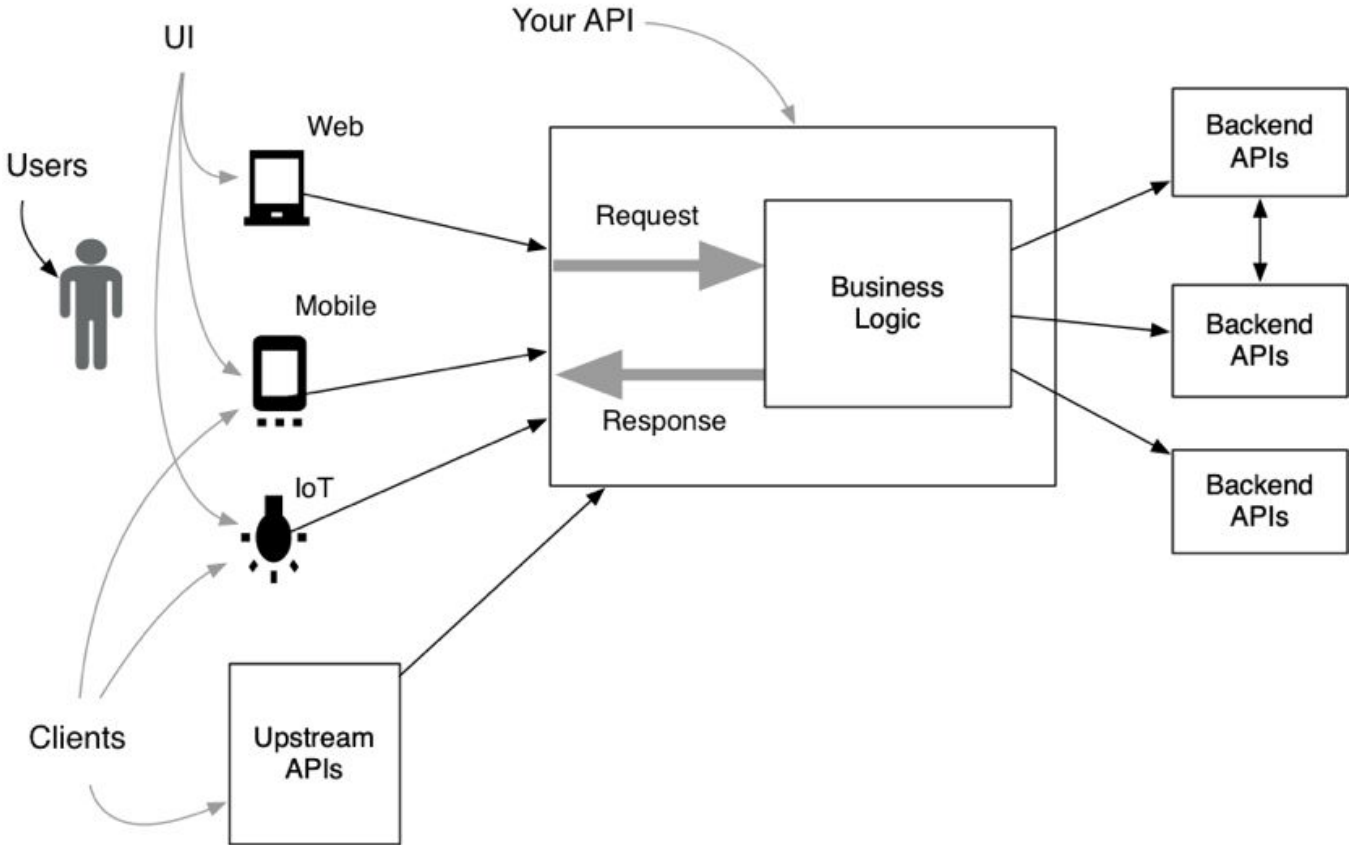
An API, or Application Programming Interface, is a set of rules and protocols that allows different software applications to communicate with each other.

It defines the methods and data formats that applications can use to request and exchange information, enabling them to work together and perform various tasks.

Types of APIs

Library or Framework APIs: These are used within the context of a programming language or framework to access pre-built functions and classes. Examples: Java API and the .NET Framework API.

Web APIs: These are accessible over the internet and allow applications to interact with web services. For example, social media platforms, payment gateways, and cloud services offer web APIs


UI
Your API
Users
Web
Mobile
IoT
Request
Response
Business Logic
Backend APIs
Backend APIs
Backend APIs
Clients
Upstream APIs

Why define an API?

Interoperability: APIs enable different software components or systems, often developed by different organizations, to work together. They serve as the bridge that allows applications to understand and interact with one another.

Abstraction: APIs provide a level of abstraction. They hide the underlying complexity of how a system or service operates and expose only the necessary functionalities and data. This simplifies integration for developers.

Why define an API?

Data Access: APIs are commonly used to access and manipulate data. This can include retrieving information from a database, making requests to web services, or controlling hardware devices.

Security: APIs can be designed with security in mind. They often require authentication and authorization to control who can access and modify the data or functionality they expose.

API properties

Stateless Protocols

In Stateless Protocol each request/response is self contained and unrelated to those that precedes or follow it.

Examples: UDP, HTTP

Pros of stateless protocols

Scalability: Stateless protocols are generally more scalable because each request is independent, and servers don't need to keep track of client state. This makes it easier to distribute requests across multiple servers.

Simplicity: Stateless protocols are often simpler to implement and understand because there is no need to manage and synchronize state information between the client and the server.

Fault Tolerance: Stateless protocols can be more fault-tolerant because the failure of one transaction does not impact subsequent transactions. Clients can retry requests without concern for the server's state.

Stateful Protocols

A stateful protocol is a communication protocol in which the sender and receiver maintain a shared state or context throughout the duration of a session.

The server keeps track of user authentication, session variables, or transaction information.

Examples: FTP, TCP, HTTP with session cookies

Stateful Protocol: pros and cons

Complexity: Stateful protocols can be more complex because both the client and server need to manage and synchronize the state information. This complexity can make implementation and maintenance more challenging.

Efficiency: In some cases, stateful protocols can be more efficient because the server has context about the client, reducing the need for redundant information in each transaction.

Delivery Semantics

"At most once," "at least once," and "exactly once" are terms used to describe the **delivery semantics** of messages in distributed systems. These semantics define how messages are sent and processed in the context of potential failures, network issues, and system errors.

At most once semantics

A message is delivered to the recipient at most once. This means that the sender attempts to send the message, and if the delivery is successful, it is considered complete. If the delivery fails, the sender does not retry, and the message may be lost.

- Acknowledgments are not typically used, or if used, they are not relied upon for retransmission.
- It prioritizes avoiding duplicate deliveries over ensuring message delivery.

Examples: scenarios where the cost of retransmission is high, and lost messages have minimal impact.

At Least Once Semantics

Sender ensures that the message is delivered to the recipient at least once. This means that the sender continues to retry delivery until it receives acknowledgment of successful delivery from the recipient.

- Acknowledgments are crucial for ensuring reliable delivery.
- The system tolerates the possibility of duplicate messages due to retries.

Examples: reliable file transfer, database replication, or distributed task execution.

Exactly Once Semantics

Sender ensures that the message is delivered to the recipient exactly once. This is a more stringent requirement compared to "at least once" semantics and aims to eliminate both message loss and duplicates.

- Often requires additional mechanisms such as unique message IDs, transactional processing, or two-phase commit protocols.
- More complex to implement and may have a higher overhead.

Examples: financial transactions, critical updates, or scenarios where maintaining data consistency is paramount.

Idempotence definition

Operation that can be **applied multiple times without changing the result** beyond the initial application.

Pros: Operation can be retried as often as necessary without causing unintended effects.

Example: HTTP GET, PUT and DELETE operations.

(HTTP POST is non-idempotent!)

Idempotence

Pros of idempotence

Reliability and Resilience: Idempotent operations enhance the reliability of systems by ensuring that repeating an operation doesn't cause unintended side effects. This is crucial in distributed environments where network issues, message duplication, or retries are common.

Simplifies Error Handling: If a client receives a response indicating that an operation was not successfully processed, it can retry the same request without having to worry about introducing inconsistencies.

Pros of idempotence

Supports Parallel and Concurrent Processing: Multiple instances of the same operation can be processed concurrently without introducing conflicts or inconsistencies.

Synchronization Across Nodes: In distributed systems with multiple nodes, idempotence facilitates synchronization. Nodes can exchange idempotent messages without worrying about the order of execution, leading to a more resilient and scalable system.

Pros of idempotence

Caching and Optimization: Since the result of an idempotent operation is the same regardless of how many times it is executed, caching mechanisms can be more effectively employed to store and serve the results of previous requests.