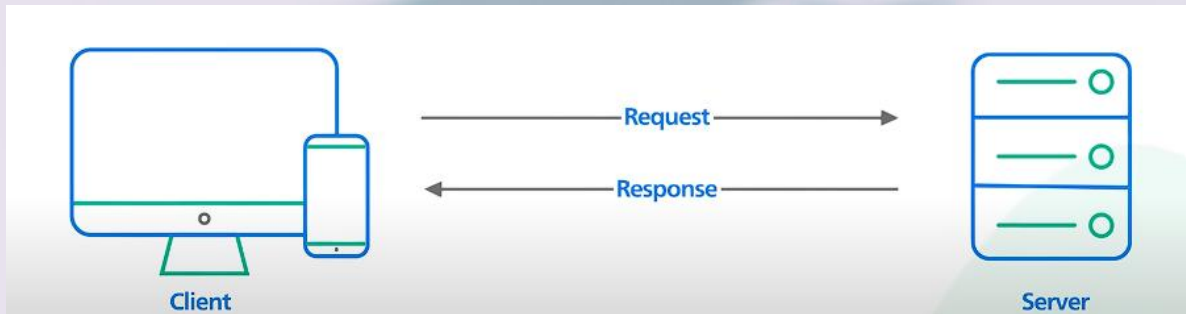


Java Microservices

Web, Rest, Soap API

- API stands for Application Programming Interface
- It is a way for 2 computers to interact with each other; the client side sends request and it receives the corresponding response from the server




➤ Web API

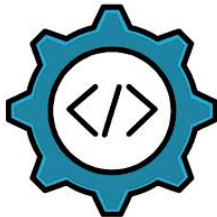
What is Web API

provide a platform-independent interface

doesn't follow a specific architecture or structure

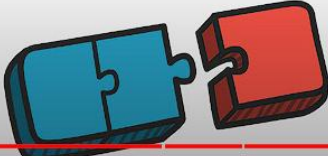


all types of APIs
REST API, SOAP, XML-RPC, etc



HTTP, HTTPS, and TCP/IP, to communicate between the clients and servers

"flexible interface"
expose various functionalities of the application, including CRUD operations, authentication, authorization, etc



SUBSCRIBE

➤ REST API

- REST stands for Representational State Transfer. It is just an api which follows REST architectural style (Restful API)
- It follows certain rules
 - Primary objective is to provide an architecture for building web services which can be consumed by various clients like web, mobile, desktop applications
 - It is based on HTTP standard protocol and methods like GET, POST, PUT, DELETE for request purposes
 - Each request has unique URL and response is returned in format of JSON or

XML

- Also, some Rest guidelines are it should be cacheable, layered system, uniform interface etc
- It follows **stateless** client server model - Server doesn't store any information about the client's state between requests. Each request contains all necessary information required by server to process. HTTP protocol facilitates this, server don't need to remember info from previous request.

What is REST API

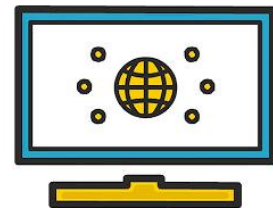
Representational State Transfer



HTTP

GET, POST, PUT, DELETE

<https://example.com/unique-url>



REST architectural style

stateless client-server model

server doesn't store any information about the client's state between requests



REST API scalable & easy to maintain

building web services (web, mobile, & desktop applications)

- Scalable and easy to maintain. Eg: Google Maps
- Useful for CRUD :

When to use REST API ?

building web services that require a stateless, scalable, & easy-to-maintain architecture

**apps with
CRUD operations**

(creating, reading, updating, & deleting data)

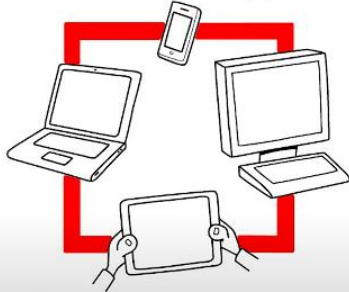


- **SOAP API**
- SOAP - Simple Object Access Protocol follows SOAP messaging protocol
- Uses XML for request and response

What is SOAP API

Simple Object Access Protocol

used for exchanging structured data between different applications



regardless of the platform or programming language used



SOAP messaging protocol



including text, numbers, dates, and binary data

HTTP, SMTP, and FTP

"set of rules"

SUBSCRIBE

-
- Set of rules defined in WSDL
- Advanced security mechanism support

the platform or programming language used

set of rules

Web Services Description Language (or WSDL)

digital signatures and encryption

```
<?xml version="1.0" encoding="utf-8" ?>
<definitions name="AktienKurs"
  targetNamespace="http://loc:
  xmlns:xsd="http://schemas.xmlsoap.org
  xmlns="http://schemas.xmlsoap.org/wsd
  <service name="AktienKurs">
    <port name="AktienSoapPort" binding
      <soap:address location="http://loc
    </port>
    <message name="Aktie.HoleWert">
      <part name="body" element="xsd:Tra
    </message>
  </service>
</definitions>
```

WSDL

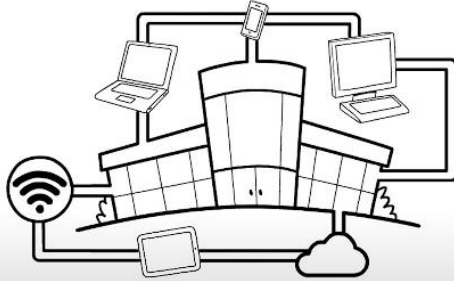
SUBSCRIBE

-
- When to use:

2014.5.18

When to use SOAP API

building apps that require a highly secure and reliable method of exchanging data between applications



require complex data structures & business logic



digital signatures & encryption

SUBSCRIBE

Difference between these APIs

Difference Between REST API vs Web API vs SOAP API Explained

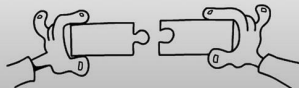
REST API

- REST architectural style
- HTTP methods (GET, POST, PUT, DELETE)
- returns data in JSON or XML format only
- stateless client-server model
- suitable for simple applications

CRUD operations & real-time communications

Web API

- doesn't follow any structure
- use any protocol or technology
 - use any data format
- can be stateful or stateless
- suitable for building complex applications



SOAP API

- uses XML as its data format
- follows specific set of rules and protocols for communication
- SOAP specific protocol, such as WSDL or UDDI
- suitable for building enterprise-level apps



2014.5.18

Monolithic vs Microservices Architecture

- Microservice architecture is an architecture where the application is loosely coupled into a collection of smaller independent services which can be independently developed, deployed and maintained
- Differences between Monolithic and Microservices architectures:

Aspect	Monolithic Architecture	Microservices Architecture
Definition	Single, unified codebase and application	Collection of small, independent services
Development	Single codebase, often leads to large and complex systems	Individual services can be developed and deployed independently
Deployment	Typically deployed as a single unit	Each service can be deployed independently
Scalability	Scales as a whole unit, which can be less efficient	Can scale individual services independently
Flexibility	Less flexible, changes affect the entire application	More flexible, changes in one service don't affect others
Technology Stack	Usually uses a single technology stack	Allows using different technologies for different services
Fault Isolation	Failure in one part can potentially bring down the whole application	Failure in one service does not affect others
Communication	Internal communication is typically simpler and faster	Services communicate over the network, adding latency and complexity
Data Management	Usually a single database	Each service can have its own database
Team Structure	Typically involves a single, unified team	Enables small, autonomous teams for each service
Testing	Can be easier to test as a single unit	Requires end-to-end testing of multiple services
Performance	Generally better due to less overhead	Can have more overhead due to inter-service communication
Complexity	Simpler to develop initially but can become complex over time ↓	More complex to develop and manage initially but can handle complexity better over time

Preference on **Monolithic** when:

- Small projects
- Easier to develop initially, and testing takes less time
- Can be faster as internal communication is simple

Preference on **Microservices** when:

- Large, Complex and growing project
- Team distributed or working on different parts of the application
- Systems needing improved fault isolation and resilience.
- Individual deployments, load balancing and fixes i.e. scalability

Microservices Start

- If you have a monolithic application, identify all possible standalone functionalities
- Once you identified them, you need to create standalone projects i.e. microservices using spring boot or something
- You need to interact between the microservices using Rest API or messaging
- Then you may need to use Eureka for service discovery, load balancer, API gateways etc
- Decentralised framework, Polygot architecture, Domain driven design, Deploy in containers
- Note: Microservices are different when compared to Service Oriented Architecture (SOA). SOA shares and reuses data as much as possible while MS focuses on sharing as little as possible

Microservices Design Patterns

- Service Discovery:
 - Mechanisms like Eureka or Consul to help services find each other.
 - It will keep track of all the microservices by registering
 - The service's port is defaulted to the well-known 8761 .
- API Gateway:
 - A single entry point for all client interactions, often handling routing, authentication, and load balancing.
 - All the requests from the clients first come to the API Gateway and the API Gateway routes the request to the correct microservice
 - Security features can be implemented at API gateway level instead of at all the microservices
 - API Gateway can also aggregate the results from the microservices back to the client
 - Eg: Spring cloud gateway, zuul
- Circuit Breaker:
 - It improves overall resilience of the system by isolating the failing services and stopping the cascading effect of failures.
 - Hystrix circuit Breaker will tolerate failures up to a threshold. Beyond that, it leaves the circuit open. Which means, it will forward all subsequent calls to the fallback method, to prevent future failures. This creates a time buffer for the related service to recover from its failing state.
 - Without a circuit breaker in place, the client would have continuously sent requests to the service which is down. Resources will get exhausted with low performance and a bad user experience due to this. To avoid this kind of problem, a circuit breaker pattern can be used.
- CQRS (Command Query Responsibility Segregation)
 - It proposes the separation of the read data model (Read) from the writing data

model (Create, Update and Delete). The application will be segregated into Command and Query parts

- The command part will be responsible for the Create, update, and delete operations. The query part will be responsible for the read operation through materialized views

➤ SAGA Design pattern:

- Needed to overcome failure of distributed transaction in microservices (a transaction that spans across multiple microservices like zomato order service, payment service and delivery service)
- A saga is a sequence of local transactions. It takes care of failure flow too. It has 2 jobs:
 - ◆ Update the current microservice and make required changes
 - ◆ Publish events to trigger the next transaction for the next microservices
- So SAGA pattern groups these local transactions and sequentially invokes the next one
- If one of the steps is failed, then SAGA pattern triggers the rollback transactions that are a set of compensating transactions that rollback the changes on previous microservices and restore data consistency
- 2 types:
 - ◆ Choreography SAGA
 - ◆ Orchestration SAGA (both in detail later)

Microservice Communication

- There are 2 types of inter service communication, Synchronous and Asynchronous
- In **Synchronous** communication between microservices, the client waits for the response within a time limit. The possible solution is using HTTP Protocol using via Rest Template, Rest API
 - In **Asynchronous** communication:
 - ◆ The client service doesn't wait for the response from another service. When the client microservice calls another microservice, the thread is not blocked till a response comes from the server. The message producer service generates a message and sends the message to a message broker on a defined topic. The message producer waits for only the acknowledgment from the message broker to know that message is received by the broker.
 - ◆ Message broker's job now is to just pass the msg to consuming service, if the recipient consuming service is down, the broker might be configured to retry as long as necessary for successful delivery. If the message is stored in memory and not persisted, it may be lost when the broker is restarted and if the msg is not yet sent to the consumer.
 - ◆ The consuming service subscribes to a topic in the messaging queue. All the messages belonging to that topic will be fed to the consuming

system(s). The message producer service and consuming services don't even know each other.

- ◆ The response is received in the same methodology through a message broker via defined message topics. Even if both microservices are down, message can still be sent via broker
- ◆ Different messaging tools are based on the AMQP (Advanced Message Queuing Protocol). Some examples are Apache Kafka, RabbitMQ
- ◆ A message broker is a vital part of the asynchronous architecture and hence must be fault tolerant. This can be achieved by setting up additional standby replicas that can do failover. Still, even with auxiliary replicas, failures of the messaging system might happen from time to time. If it's essential to ensure the message arrives at its destination, a broker might be configured to work in at-least-once mode. After the message reaches the consumer, it needs to send back acknowledgement to the broker. If no acknowledgement gets to the broker, it will retry the delivery after some time

Microservice handling more users?

- Load balancing
 - You can do it by having multiple instances and rather than overloading one server, implement load balancer to evenly distribute all the new users to multiple instances
- Horizontal Scaling
 - It's just adding more and more instances of your service
- Auto Scaling
 - Cloud servers automatically add more replicated instances and remove depending on user load
- Caching
 - It will help in accessing data quickly and send to the users
- Database Scaling
 - It's adding more and more databases
- Asynchronous Processing
 - For quick responses, no need to always wait. Eg: sending initial mail instantly after ordering

Difference between 2Phase and SAGA Microservice

2 Phase - Order service?

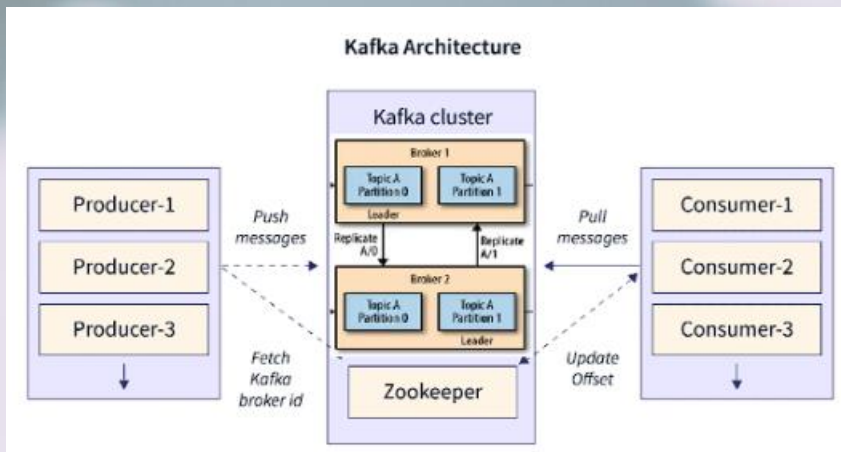
SAGA - Email service notification

2 Phase Commit	Saga Pattern
Strong Consistency	Eventual Consistency
Synchronous	Asynchronous
Blocking	Non Blocking
Complexity	Simplicity
Performance Overhead	Better Performance
Strong Data Consistency	Temporarily Inconsistency

Apache Kafka

- Apache Kafka is publish-subscribe based fault tolerant asynchronous messaging system. It is fast, scalable and distributed by design.
- In publisher subscriber messaging based communication, the publisher will send the message which is stored in the partition in topic in the message broker and then subscribers that subscribe to that topic will consume the message
- Traditional messaging models are queue based like RabbitMQ, messages are deleted after consuming. However in Kafka publish-subscribe, message is received by all consumers and they can even re read them
- RabbitMQ - push type, smart broker/dumb consumer
- Apache Kafka - pull type, dumb broker/smart consumer
- Also, we use Zookeeper to maintain Kafka clusters
- Pros of Kafka:
 - Loose coupling — Neither service knows about each other regarding data update matters.
 - Durability — Guarantees that the message will be delivered even if the consumer service is down. Whenever the consumer gets up again, all messages will be there.
 - Scalability — Since the messages get stored in a bucket called topic, there is no need to wait for responses. We create asynchronous communication between all services.
 - Flexibility — The sender of a message has no idea who is going to consume it. Meaning you can easily add new consumers (new functionality) with less work
- Cons of Kafka:
 - Semantics — The developer needs to have a deep understanding of the message flow as its strict requirements. Complex fallback approaches may take place.
 - Message Visibility — You must track all those messages to allow you to debug whenever a problem occurs. Correlation IDs may be an option.

Kafka Architecture



- Producers:
 - Producers are clients that publish (write) data to Kafka topics.
 - They send messages/records to specific topics and can choose which partition within the topic to send messages to, based on a partitioning key or round-robin fashion.
- Topics:
 - Topics are categories or feed names to which messages/records are sent.
 - A topic is split into partitions to allow data parallelism and scalability.
 - Each topic can have multiple partitions, which allows Kafka to scale horizontally.
- Partitions:
 - Partitions are the basic unit of parallelism and scalability in Kafka.
 - Kafka distributes the data of a topic across its partitions. When you produce messages to a topic, Kafka uses a partitioning strategy to determine which partition the message should be sent to. This can be based on the message key(hash function on the key) or a round-robin approach.
 - Each partition is an ordered, immutable sequence of messages/records.
 - Records in partitions are identified by their offset (an integer value).
 - Data within a partition is ordered, but there is no global ordering across partitions.
 - By dividing a topic into multiple partitions, Kafka can distribute the data across multiple brokers (servers), allowing for horizontal scaling. Each partition can reside on a different broker, enabling Kafka to handle large volumes of data
- Consumers:
 - Consumers are clients that read (consume) data from Kafka topics.
 - They read messages/records from specific topics and partitions and can do so independently.

- Consumers keep track of their position (offset) within partitions.
- Consumer Groups:
 - A consumer group is a group of consumers that work together to consume data from a topic.
 - Each consumer in a group is assigned one or more partitions exclusively.
 - This allows for horizontal scaling of consumers while ensuring that each record is processed once.
- Cluster:
 - A Kafka cluster consists of multiple brokers.
 - It provides a distributed and fault-tolerant system for data streaming.
 - Brokers coordinate with each other using ZooKeeper.
- Brokers/Kafka Server:
 - Brokers are Kafka servers that store data and serve client requests.
 - A Kafka cluster consists of multiple brokers.
 - Each broker is responsible for some partitions and replicates partitions for fault tolerance.
- ZooKeeper:
 - ZooKeeper is used to manage and coordinate Kafka brokers.
 - It maintains information about the cluster state, broker metadata, and partition leader election.
 - Kafka relies on ZooKeeper for leader election and configuration management.
- Replication:
 - Kafka replicates partitions across multiple brokers for fault tolerance.
 - Each partition has one leader and multiple followers.
 - The leader handles all reads and writes, while followers replicate the data.
- Leader and Followers:
 - The leader is the node responsible for all reads and writes for a given partition.
 - Followers replicate the leader's data and take over if the leader fails.
 - This ensures high availability and fault tolerance.

Kafka Terminologies

- Record/Message:
 - The basic unit of data in Kafka, also known as a message.
 - A record consists of a key, a value, and a timestamp.

- Offset:
 - A unique identifier assigned to each record within a partition.
 - It represents the position of the record in the partition.
- Log:
 - A log is the data structure used to store records in a partition.
 - It is an append-only sequence of records.
- Retention Policy:
 - The policy that determines how long Kafka retains records.
 - Can be based on time (e.g., 7 days) or log size (e.g., 100 GB).
- Compaction:
 - A cleanup policy that ensures only the latest value for each key within a topic is retained.
 - Useful for topics that store updates to keys.
- Replication Factor:
 - The number of copies of each partition that Kafka maintains.
 - Higher replication factors increase fault tolerance but also require more storage.
- Leader Election:
 - The process by which Kafka selects a new leader for a partition if the current leader fails.
 - Managed by ZooKeeper.

Kafka Example Workflow

- Producing Data:
 - A producer sends a record to a Kafka topic.
 - The record is appended to a partition within the topic.
- Storing Data:
 - The record is stored in the partition log.
 - The partition is replicated across multiple brokers for fault tolerance.
- Consuming Data:
 - A consumer in a consumer group reads the record from the topic.
 - The consumer processes the record and updates its offset to mark the record as consumed.
- Handling Failures:
 - If a broker fails, ZooKeeper triggers a leader election.

- A new leader is elected from the followers, ensuring high availability.

