Microservices Architecture

Agenda

- What are microservices and Why are they becoming popular
- Design Principles for Building Microservices
- Microservices Architecture in Practice
- Best Practices
- Communication Between Microservices
- Service Discovery And Registration
- API Gateway
- Deployment and Scaling

What are microservices?

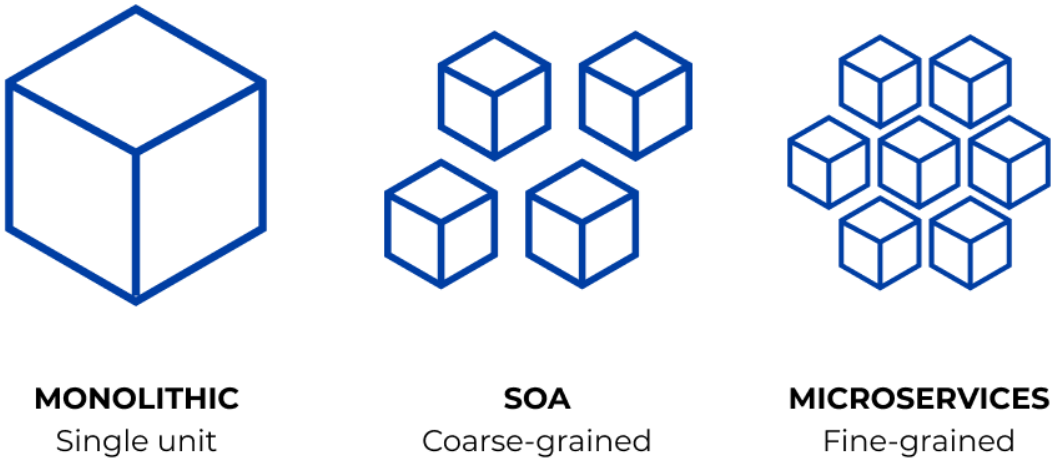
Microservices are a **software architecture style** in which a large application is broken down into smaller, independently deployable services.
An architecture style refers to a set of **guidelines**, **principles**, and **patterns** for designing and building software systems. It provides a framework for organizing and

structuring the different components and modules of a system, as well as for specifying how those components and modules interact with each other.
Different architecture styles have different characteristics and are suited to different types of systems and use cases. Some examples of architecture styles include:

1. **Monolithic Architecture**: A traditional, all-in-one approach where all components of the system are tightly integrated into a single codebase and deployable unit. It's simple to develop and deploy, but can be challenging to scale and evolve over time.
2. **Event-Driven Architecture**: A style where the system is built around the concept of events, with different components of the system reacting to and generating events. It allows for greater scalability and flexibility, but can be difficult to debug and reason about.
3. **Layered (N-Tier) Architecture**: A style where the system is divided into several layers, each with a specific responsibility. It allows for separation of concerns and easier maintenance, but can lead to tight coupling between layers. For example, Spring MVC architecture.
4. **Serverless Architecture**: A style where the system is built using serverless computing services, such as AWS Lambda or Azure Functions or OCI Functions. It allows for scalability and cost-effectiveness, but can be challenging to develop, test and debug.
5. **Component-Based (Service-Oriented) Architecture**: A style where the system is built as a set of reusable, loosely coupled components. Each service provides a business capability, and services can also communicate with each other across platforms and languages. Developers use SOA to reuse services in different systems or combine several independent services to perform complex tasks. It allows for flexibility and code reuse, but can be complex to design and implement.

The choice of architecture style depends on the specific requirements of the system and the desired trade-offs in terms of scalability, maintainability, and performance.

- Each service is responsible for a specific business function and can be **developed**, **deployed**, and **scaled independently** of the other services.
- Microservices are typically built using modern technologies such as **containers** and **cloud-native infrastructure**.



Why are microservices becoming popular?

Microservices offer a number of benefits over traditional monolithic architectures -

● They allow for faster development and deployment, as each service can be developed and deployed independently.

● They also make it easier to scale and manage an application, as each service can be scaled independently.

● Additionally, microservices make it easier to implement new technologies and handle changes in business requirements.

● Microservices also provide flexibility and allows for different teams to work on different services independently and also promotes reuse of services.

● Microservices also provide fault isolation, as a failure in one service will not bring down the entire system.

Design Principles for building Microservices

The key principles that should be considered when designing and building microservices.

1. **Single Responsibility Principle**
   Each microservice should have a single responsibility and should only be

responsible for **one specific business function**. This makes it easier to understand, test, and maintain the service.

2. **Loose Coupling**
   Microservices should be loosely coupled, meaning that they should have minimal dependencies on other services. This allows for greater flexibility and makes it easier to change or replace a service without affecting other services.
3. **High Cohesion**
   Microservices should be highly cohesive, meaning that **all the components within a service should be closely related to the service's single responsibility**. This makes it easier to understand and maintain the service.
4. **Autonomy**
   Microservices should be **autonomous and able to function independently** of other services. This allows for **faster development, deployment, and scaling of services**.
5. **Stateless**
   Microservices should be stateless, meaning that they should not maintain any state information. This allows for greater scalability and fault tolerance.
6. **Smart Endpoints and Dumb Pipes**
   Microservices should have "smart endpoints" and "dumb pipes" for communication. The endpoints should contain business logic and functionality, while the pipes should only be responsible for transporting data between services.
7. **Monitoring and Logging**
   Microservices should be designed with monitoring and logging in mind, to make it easier to identify and troubleshoot issues.

Microservices Architecture in Practice

Designing an e-commerce application using microservices architecture requires a thorough understanding of the domain and the various business functions involved in an e-commerce application. Here are the general steps you can take to design an e-commerce application using microservices architecture:

1. **Identify the core business functions**: E-commerce applications typically include functions such as product catalog management, shopping cart management,

ordering, payment, and shipping. Identify the key business functions for your e-commerce application and determine which functions would be best suited for implementation as microservices.

2. **Decompose the monolithic application**: Break down the monolithic application into smaller, independently deployable services. Each service should be responsible for a specific business function and should have a well-defined API.
3. **Design the communication between the microservices**: Determine how the microservices will communicate with each other. Microservices should communicate over a lightweight protocol, such as HTTP or gRPC. Consider using an API Gateway to handle tasks such as authentication, rate limiting, and request routing etc.
4. **Implement Service discovery and Registration**: Use a service discovery and registration mechanism to allow microservices to discover and communicate with other services in the system.
5. **Implement monitoring and logging**: Implement monitoring and logging to make it easier to identify and troubleshoot issues. Consider using tools such as Prometheus and ELK stack (Elastic search, Logstash, Kibana).
6. **Test and deploy the services**: Test each service individually, and deploy them to a production environment using containerization and orchestration tools like Docker and Kubernetes.
7. **Scalability**: Scale up or down the services independently based on the traffic and resource requirements.

Best Practices

- Communication between microservices should be done over a lightweight protocol, such as HTTP or gRPC.
- Services should be designed to be **fault-tolerant**, to handle communication failures. Fault tolerance is a way for microservices to handle the unavailability of a service by using different stability patterns. There are many stability patterns which have been used to achieve robustness and resilience including **Circuit Breaker Pattern**, **Timeout Pattern**, and **Retry Pattern** etc**.**

- Services should be designed to handle **high-throughput** and **low-latency** communication.
- Services should be designed to handle different types of requests, such as query and command requests. For example, CQRS i.e. Command Query Responsibility Segregations.
- Services should be designed to handle different types of data, such as text and binary data.

Communication Between Microservices In a Distributed System

Below are the different options and best practices for communicating between microservices in a distributed system

1. **Synchronous Communication**
   ● Synchronous communication is when a service sends a request to another service and waits for a response.
   ● This is the most common form of communication between microservices.
   ● Examples of synchronous communication include REST and gRPC.
2. **Asynchronous Communication**
   ● Asynchronous communication is when a service sends a request to another service without waiting for a response.
   ● This can be useful for handling tasks that are time-consuming or can be performed offline.
   ● Examples of asynchronous communication include message queues and event-driven architectures.
3. **API Gateway**
   ● An API Gateway is a service that acts as an intermediary between the client and the microservices.
   ● It can be used to handle tasks such as authentication, rate limiting, and request routing.
   ● This can help to reduce the load on the microservices and simplify communication between them. Examples of popular API gateways are Zuul Server and NGINX etc.

4. **Service Discovery and Registration**
   ● Service Discovery and Registration is a mechanism that allows microservices to discover and communicate with other services in the system.
   ● Examples of service discovery and registration include Netflix Eureka and Consul.

Communicating between the Microservices

Synchronous Communication

There are several ways to implement synchronous communication between microservices, here
are some common methods:

1. **REST (Representational State Transfer)**: REST is a widely used architecture style for building web services. RESTful services can be implemented using HTTP and the various HTTP methods, such as GET, POST, PUT, and DELETE. RESTful services can be consumed using any programming language that can make HTTP requests.
2. **gRPC**: gRPC is a high-performance, open-source framework for building remote procedure call (RPC) APIs. It uses Protocol Buffers as the underlying data serialization format and supports a variety of programming languages. gRPC enables client and server applications to communicate transparently and make it easier to build connected systems.
3. **SOAP (Simple Object Access Protocol)**: SOAP protocol is used for exchanging structured information in the implementation of web services in computer networks. It uses XML as its message format, and can be carried over a variety of lower-level protocols, including HTTP and SMTP.
4. **JSON-RPC**: JSON-RPC is a remote procedure call (RPC) protocol encoded in JSON. It is a light-weight protocol involving a small number of data types and commands. JSON-RPC allows for notifications (server to client) as well as for multiple calls to be sent to the server which may be answered out of order.

Asynchronous Communication

There are several ways to implement asynchronous communication between microservices,
here are some common methods:

1. **Message Queues**: Message queues are a way for microservices to send and receive messages asynchronously. The sender sends a message to the queue and the receiver retrieves it from the queue. Some popular message queue technologies include RabbitMQ, Apache Kafka and Amazon SQS.
2. **Event-Driven Architecture**: In this approach, microservices communicate by producing and consuming events, rather than by making direct requests to each other. Microservices can publish events to a central event bus, and other microservices can subscribe to those events. This allows for loosely coupled communication, and can be useful for handling tasks that are time-consuming or can be performed offline.
3. **Webhooks**: Webhooks are a way for one service to send a notification to another service when a specific event occurs. For example, service A may send a webhook to service B when a new order is placed. Service B can then process the order asynchronously without having to repeatedly poll service A for new orders.
4. **Pub/Sub pattern**: Pub/Sub pattern is a messaging pattern where a service sends a message to a topic, and other services subscribe to that topic to receive the message. This allows for decoupled communication between services, and can improve scalability and fault-tolerance.

API Gateways

It can handle tasks such as:

1. **Routing**: The API Gateway routes incoming requests to the appropriate microservice based on the URL or other request information. This allows the client to access different services and resources in the e-commerce application through a single endpoint.
2. **Authentication and Authorization**: The API Gateway can handle tasks such as authenticating the client and authorizing the request before forwarding it to the microservice. This can include tasks such as verifying user credentials, checking permissions, and enforcing security policies.

3. **Rate Limiting**: The API Gateway can limit the number of requests that a client can make in a given time period, to protect the microservices from excessive traffic.
4. **Caching**: The API Gateway can cache frequently requested data to reduce the load on the microservices and improve performance. This can include data such as product information, customer information, and order history.
5. **Transformation**: The API Gateway can transform requests and responses between the client and the microservices to ensure compatibility. This can include tasks such as converting data formats, encrypting or decrypting data, and compressing or decompressing data.
6. **Aggregation**: The API Gateway can aggregate the responses from multiple microservices to provide a single response to the client. This can include data such as product catalog, shopping cart, order and payment details.
7. **Monitoring and Logging**: The API Gateway can collect and log data about incoming requests and outgoing responses to help with debugging and troubleshooting. This can include data such as request and response headers, payloads, timestamps, and error messages.

Service Registry and Discovery

Service Registry and Discovery is a mechanism that allows microservices to discover and
communicate with other services in a distributed system.

1. **Service Registry**: The service registry is a central repository where all the microservices register themselves, providing information such as their name, IP address, and port number. This information can be used by other microservices to locate and communicate with the registered services.
2. **Service Discovery**: Service discovery is the process of locating a specific microservice in the service registry. This can be done by querying the service registry for a specific service by its name or by searching for all services of a certain type.
3. **Load balancing**: Once a microservice is located, the service discovery mechanism can also be used to perform load balancing by redirecting incoming requests to the service instance with the lowest load.

4. **Failure detection**: The service registry and discovery mechanism can also be used to detect when a service is no longer available and remove it from the registry. This can be done by periodically sending heartbeats to the service and removing the service from the registry if the heartbeat is not received.
5. **Service Mesh**: A service mesh is a configurable infrastructure layer for microservices application that makes communication between service instances flexible, reliable, and fast. It makes it easy to manage service-to-service communication inside a cluster. Service registry and discovery are important components in a microservices architecture, as they allow for dynamic and flexible communication between the microservices.

Deploying and scaling

The first step in deploying an ecommerce application with microservices architecture is to define the various components that will make up the system. This includes things like databases, API servers, web servers, and other services. After establishing what components are needed for the application, it is important to determine how they will interact with each other.

Once all of the components have been defined, they must be containerized. This means creating a lightweight environment for running the application's components, which is typically done through Docker or Kubernetes. After they are containerized and configured, they can then be deployed to a cloud provider such as AWS, Google Cloud Platform, or Microsoft Azure.

Once the components have been deployed, it is important to test the system to ensure that it functions as intended. This includes verifying that all of the components are communicating correctly, ensuring that there are no security vulnerabilities, and validating performance metrics.

Once all of the tests have been passed, the system can then be released into production. During this phase, it is important to make sure that the system is monitored closely in order to ensure that it remains stable and secure. This can be done by monitoring performance metrics, logging errors, and regularly applying security patches.