REST Endpoints with Annotations

Basic REST Annotations

- @RestController: Marks a class as a controller where every method returns a domain object instead of a view.
- @GetMapping: Maps HTTP GET requests onto specific handler methods.
- @PathVariable: Extracts values from the URI path.
- @RequestParam: Extracts query parameters from the URL.
- @PostMapping: Maps HTTP POST requests onto specific handler methods. It's often used with `@RequestBody` to handle data in the request body.
- @PutMapping: Maps HTTP PUT requests onto specific handler methods for updating resources.
- @DeleteMapping: Maps HTTP DELETE requests onto specific handler methods for deleting resources.

Microservices Architecture

What are Microservices?

> Microservices architecture is an architectural style that structures an application as a collection of smaller, interconnected services. Each service typically implements a set of distinct features or functionality, such as order management or customer management.

Some microservices expose an API that is consumed by other microservices or by applications clients.

Characteristics of Microservices

- Highly maintainable and testable
- Loosely coupled: Changes in one service don't affect other services.
- Independently deployable
- Has its own DB (if a DB is needed)
- Owned by a small team

Monolith Architecture

In a monolith architecture, all application components are packaged into a single container.

Advantages and Disadvantages of Monolithic Architecture

Advantages	Disadvantages
Simple to develop (at the beginning)	Maintenance becomes challenging if the application is large and complex.
Simple to test	The application size can slow down startup time.
Simple to deploy	Requires redeploying the entire application on each update.
Simple to scale horizontally via load balancers	A bug in any module can potentially bring down the entire process.
N/A	Monolithic applications struggle to adopt new technologies due to the complexity and cost of making changes that affect the entire application.

Monolith vs. Microservices

Feature	Monolith	Microservices
Business Goal	Single code base for all business goals	Deliver one specific business goal
Coupling	Tightly coupled	Loosely coupled: A change in one microservice doesn't affect other services.
Startup Time	Service startup takes a long time	Service startup is quick
Failure Impact	The whole system goes down	If one service goes down, other services can continue working.
Team Size	Large team, requires considerable managing effort	Small, focused team for a service
Technology Stack	All modules use the same technology stack	Different microservices can use completely different technology stacks.
Data Sources	Centralized data source	Individual data sources for each service, enabling different data models per service.

Communication

Microservice-based applications are distributed systems running on multiple services that need inter-service communication. The communication method is decided based on the nature of each service. Clients and services can communicate through synchronous and asynchronous communication.

- Synchronous Protocol: The client sends a request and waits for the response. HTTP/HTTPS is a synchronous protocol.
- Asynchronous Protocol: The client does not block a thread while waiting for a response. This is often realized with messaging brokers. The message producer waits for an acknowledgement that the message has been received by the broker, but not necessarily processed. AMQP (Advanced Message Queuing Protocol) is a popular protocol for this type of communication. Popular message brokers include RabbitMQ and Apache Kafka.

Service Registry

> A service registry is a database of available service instances that enables microservices to consume other microservices without knowing their exact locations (usually a URL).

Examples include Apache Zookeeper and Netflix Eureka.

Advantages of the Service Registry Pattern

- Helps to identify dynamically assigned network locations in modern, cloud-based microservices applications.
- Handles dynamic changes due to autoscaling, failures, and upgrades.
- Manages IP addresses and ports of each service.

What if a Service Fails?

If a service fails, other collaborating services might be unable to proceed until the failing service responds. While waiting, expensive resources (such as threads) might be consumed in the caller, potentially leading to resource exhaustion. There should be a way to prevent a network or service failure from cascading to other services.

Circuit Breaker

> A circuit breaker monitors failures of a service. Once the failures reach a certain threshold, the circuit breaker trips, and all calls to the circuit breaker return an error message or alternate service. After a timeout period, the circuit breaker checks again whether the service is available.

If the service is available, it stops returning error messages.

Advantages of Circuit Breaker

- Failing fast.
- Valuable for monitoring.
- Fallback functionality: Services handle the failure of the services that they invoke.
- Automatic recovery.

Circuit Breaker States

- **Closed**: When everything is normal, all calls pass through to the services.
- **Open**: When the number of failures exceeds a threshold, the breaker trips and returns an error for calls without executing the function.
- **Half-Open**: After a timeout period, the circuit switches to a half-open state to test if the underlying problem still exists. If a single call fails in this half-open state, the breaker is once again tripped. If it succeeds, the circuit breaker resets back to the normal, closed state.

Client Communication with Microservices

Clients can directly call microservices using the exposed endpoint. However, there are limitations:

- Mismatch between client needs and the API exposed by a microservice.
- The client has to make several requests to obtain data for one feature.
- Making several requests would be inefficient over the public internet and impossible on mobile networks.
- It makes it difficult to refactor the microservices.

Because of these problems, it rarely makes sense for clients to talk directly to microservices.

API Gateway

> An API Gateway is a server that is the single-entry point into the system. It is responsible for request routing, composition, and protocol translation. All requests from clients first go through the API Gateway, which then routes requests to the appropriate microservice.

The API Gateway will often handle a request by invoking multiple microservices and aggregating the results. It can translate between web protocols (such as HTTP and WebSockets) and web-unfriendly protocols that are used internally. The API Gateway can also provide each client with a custom API.

Single Point of Failure (SPOF)

> A single point of failure (SPOF) is a part of a system that, if it fails, will stop the entire system from working.

Avoid SPOF in Microservices

1. Adding multiple application servers.
2. Adding multiple database servers (master-slave).
3. Distributing servers within multiple regions (AWS).

Microservices Application Implementation

Goal

Build a RESTful web service (a simple microservice) using Spring Boot and a MySQL database.

Dependencies

- SPRING WEB
- SPRING DATA JPA
- MYSQL DRIVER

Spring Boot Configuration

The `application.properties` file, located in the `resources/` directory, contains application-wide properties that Spring reads to configure the application.

Configuring MySQL Database

Spring Boot tries to auto-configure a DataSource if `spring-data-jpa` dependency is in the class path by reading the database configuration from `application.properties` file.

Creating the Entity Class

- **@Entity**: Specifies that the class is an entity and is mapped to a database table.
- **@Table**: Specifies the name of the database table to be used for mapping.
- **@Id**: Specifies the primary key of an entity.
- **@GeneratedValue(strategy = GenerationType.IDENTITY)**: Specifies the generation strategy for the values of primary keys.
- **@Column**: Used to mention the details of a column in the table. It has elements such as name, length, nullable, and unique.

Creating the Repository Class

- **@Repository**: Indicates that an annotated class is a repository, which is an abstraction of data access and storage.
- **@Query**: In order to define SQL to execute for a Spring Data repository method, we can annotate the method with the `@Query` annotation. Its value attribute contains the JPQL or SQL to execute.

Dependency Injection

- **@Autowired**: Allows Spring to resolve and inject collaborating beans into our bean.