**S.O.L.I.D Principles:**

**Single Responsibility Principle (SRP):**

* The single responsibility principle states that **every Java class must perform a single functionality.**
* Implementation of multiple functionalities in a single class mashup the code and if any modification is required it may affect the whole class

|  |
| --- |
| * Better maintainability and separation of concerns. |

**Open-Closed Principle (OCP):**

* The open-closed principle states that according to new requirements **the module should be open for extension but closed for modification.**
* We should be able to add new functionality without modifying existing code.

|  |
| --- |
| * New features can be added without modifying existing code. |

**Liskov Substitution Principle (LSP):**

* It applies to inheritance in such a way that the **derived classes must be completely substitutable for their base classes**.
* Objects of a subclass should be able to replace objects of the superclass without breaking the application.
* In other words, if class A is a subtype of class B, then we should be able to replace B with A without interrupting the behavior of the program.
* Subclasses should be **replaceable** for their base class without breaking functionality.
* Prevents unexpected behaviors

**Interface Segregation Principle (ISP):**

* The principle states that the larger interfaces split into smaller ones. Because the implementation classes use only the methods that are required. We should not force the client to use the methods that they do not want to use.
* Clients should not be forced to implement **unnecessary methods**.

|  |
| --- |
| * Avoids bloated interfaces. |

**Dependency Inversion Principle (DIP):**

* The principle states that we must use abstraction (abstract classes and interfaces) instead of concrete implementations.
* High-level modules should not depend on the low-level module but both should depend on abstraction. Because the abstraction does not depend on detail, but the detail depends on abstraction.
* Depends on abstraction, but not concretions.
* Improves flexibility and testability.

**Domain Driven Design Pattern (DDD):**

* **DDD** helps structure microservices around **business domains** rather than technical layers
* It ensures **loose coupling** and **high cohesion** within each microservice.
* Represents the **business problem** that the application solves.
* Example: In a banking system, domains can be **Accounts, Transactions, Loans, etc.**
* Defines **clear boundaries** for each microservice.
* Each microservice has **its own domain model** and **database**.
* Example: The **Order Service** and **Payment Service** should be separate with their own logic and data.
* Ensures that **developers, business analysts, and stakeholders** use a common language.
* Example: Instead of saying *"fetch order details"*, all teams should agree on the term **"Retrieve Order"**.
* **Aggregate**: A group of related objects treated as a single unit.
* **Aggregate Root**: The main entry point to modify the aggregate.
* Example: In an **Order Service**, an Order aggregate can include Order Items and Shipping Details, with Order as the **aggregate root**.
* **Entity**: Has a unique identifier (ID) and lifecycle. Example: Customer, Order.
* **Value Object**: Does not have an ID, is immutable. Example: Address, Money.
* Used to **capture and propagate changes** across microservices asynchronously.
* Example: When an order is placed, a **"OrderPlacedEvent"** is published to notify the **Inventory Service**.
* **Scalability** – Each microservice can scale independently.
* **Maintainability** – Smaller, domain-focused microservices are easier to manage.
* **Decoupling** – Clear boundaries prevent services from interfering with each other.
* **Better Communication** – Ubiquitous language ensures alignment between business and development teams.
* **Data Integrity** – Aggregate rules enforce consistency within a bounded context.
* **Event-Driven Architecture, CQRS, and Event Sourcing** are widely used patterns in DDD-based microservices.

Order Service (Bounded Context)

├── Order (Aggregate Root)

│ ├── Order Items (Entity)

│ ├── Shipping Details (Value Object)

├── OrderCreatedEvent (Domain Event)

├── OrderRepository (Repository Pattern)

├── OrderService (Application Layer)

**Key Factors to Consider When Designing Microservices Architecture**

|  |  |
| --- | --- |
| **1. Service Boundaries** | Use DDD, define clear Bounded Contexts |

|  |  |
| --- | --- |
| **2. API Design** | Use REST, GraphQL, or gRPC |

|  |  |
| --- | --- |
| **3. Database** | Use Database-per-Service (SQL/NoSQL) |

|  |  |
| --- | --- |
| **4. Communication** | Prefer async messaging (Kafka, RabbitMQ) |

|  |  |
| --- | --- |
| **5. Resilience** | Use Circuit Breaker, Retry Mechanisms |

|  |  |
| --- | --- |
| **6. Security** | Implement OAuth2, JWT, API Gateway |

|  |  |
| --- | --- |
| **7. API Gateway** | Manage traffic and authentication centrally |

|  |  |
| --- | --- |
| **8. Scalability** | Use Kubernetes, Load Balancing |

|  |  |
| --- | --- |
| **9. Observability** | Use Spring Boot Actuator, ELK Stack, Zipkin |

|  |  |
| --- | --- |
| **10. CI/CD** | Automate deployments with Docker, Kubernetes, Jenkins |

|  |  |
| --- | --- |
| **11. Configuration** | Use Spring Cloud Config for centralized config |

|  |  |
| --- | --- |
| **12. Transactions** | Use CQRS, Event Sourcing for managing large datasets |
|  |  |

1. **Service Boundaries (Domain-Driven Design - DDD)**

* Define **clear boundaries** using **Bounded Contexts** from **Domain-Driven Design (DDD)**.
* Each microservice should handle **a specific business capability**.

**Example**  
For an **e-commerce system**, split services as:  
✔ **Order Service** – Manages orders  
✔ **Payment Service** – Handles transactions  
✔ **Inventory Service** – Tracks stock levels

**2. API Design (REST, GraphQL, gRPC)**

* Choose the right communication style
  + **RESTful APIs** for standard CRUD operations.
  + **GraphQL** for flexible querying.
  + **gRPC** for high-performance, low-latency communication.
* Example: REST API with Spring Boot

**3. Database Design (Database per Service)**

* Avoid **monolithic databases**.
* Use **Database-per-Service** pattern
  + **SQL (PostgreSQL, MySQL)** for structured data.
  + **NoSQL (MongoDB, Cassandra)** for unstructured/scalable data.
* **Example: Using PostgreSQL in Spring Boot**

**4. Inter-Service Communication**

**Use Asynchronous vs Synchronous**

* **Synchronous**: REST/gRPC (direct calls).
* **Asynchronous**: **Kafka, RabbitMQ** for event-driven communication.
* **Example**: Event-Driven Communication with Kafka

**5. Resilience & Fault Tolerance**

* Implement **Circuit Breaker (Hystrix, Resilience4J)** to handle failures gracefully.
* Use **Retry Mechanisms** for transient failures.
* **Example**: Circuit Breaker with Resilience4J

**6. Security Considerations**

* Use **OAuth2, JWT, or API Gateway** for authentication.
* Secure sensitive data with **encryption**.
* **Example**: Secure Microservices with OAuth2

**7. API Gateway (Centralized Access)**

* Use an **API Gateway** (e.g., **Spring Cloud Gateway, Kong, Zuul**) to manage:
  + Authentication
  + Rate limiting
  + Request routing
* **Example**: Spring Cloud Gateway Configuration

**8. Scalability & Load Balancing**

* Deploy services in **Kubernetes** for auto-scaling.
* Use **Netflix Eureka, Consul** for service discovery.
* Implement **Load Balancing (Ribbon, Nginx, ALB in AWS)**.
* **Example**: Kubernetes Horizontal AutoScaling

**9. Observability (Monitoring & Logging)**

* Use **Spring Boot Actuator** for health checks.
* Centralized logging with **ELK Stack (Elasticsearch, Logstash, Kibana)**.
* Distributed tracing with **Zipkin, Jaeger**.
* **Example:** Spring Boot Actuator for Health Monitoring

10. **Deployment & CI/CD Pipeline**

* Use Docker for containerization.
* Automate deployments with Jenkins, GitHub Actions.
* Deploy with Kubernetes & Helm Charts
* Example: Dockerfile for Spring Boot App

**11. Configuration Management**

* Use Spring Cloud Config or Consul for centralized config management.
* Example: Spring Cloud Config Server

**12. Handling Large Data & Transactions**

* Use CQRS (Command Query Responsibility Segregation) to separate read/write workloads.
* Implement Event Sourcing for tracking changes.
* **Example: Kafka with Event Sourcing**

**12 Factors in Micro Services**

**1️ Codebase - One Codebase, Multiple Deployments**

* Maintain **a single codebase** for an application.
* Use **version control (Git, SVN, etc.)**.
* Deploy the same codebase across different environments (DEV, QA, PROD) with **config variations**.

**2️ Dependencies - Explicitly Declare & Isolate Dependencies**

* Use **dependency management** tools like Maven or Gradle.
* Do not rely on system-wide installed packages.
* Define dependencies in pom.xml or build.gradle.

**3️ Configuration - Store Config in Environment Variables**

* **Do not store configs** inside the codebase.
* Use **environment variables** (SPRING\_PROFILES\_ACTIVE, DATABASE\_URL).
* Separate configurations for **DEV, QA, PROD** using application.properties or application.yml.

**4️ Backing Services - Treat Services as Attached Resources**

* Treat **databases, message queues (Kafka, RabbitMQ), APIs** as external resources.
* Use **config-driven URLs** instead of hardcoding connection details.
* Example:  
  ✅ Good → DATABASE\_URL=jdbc:mysql://db-host:3306/app  
  ❌ Bad → jdbc:mysql://localhost:3306/app in code.

**5️ Build, Release, Run - Strictly Separate Build, Release & Run Stages**

* **Build Stage**: Compile code, package artifacts.
* **Release Stage**: Merge build artifacts with configs.
* **Run Stage**: Deploy and execute the app.
* Example: **Use CI/CD pipelines (Jenkins, GitHub Actions, GitLab CI)** to automate these steps.

**6️ Processes - Execute the App as Stateless Processes**

* Avoid **storing session data** on local disk.
* Use **distributed caches (Redis, Memcached)** for session storage.
* Stateless services allow **horizontal scaling**.

**7️ Port Binding - Export Services via Port Binding**

* Applications should **self-contain web servers** (e.g., Tomcat, Jetty).
* Expose services via ports instead of relying on a centralized server.
* Example: **Spring Boot apps run on server.port=8080 and expose REST APIs**.

**8️ Concurrency - Scale Out via Process Model**

* Use **multiple instances** of services to handle high traffic.
* Utilize **container orchestration** (Kubernetes, Docker Swarm).
* Implement **auto-scaling** based on load.

**9️ Disposability - Fast Startup & Graceful Shutdown**

* Applications should **start quickly** and **shutdown gracefully**.
* Use **proper signal handling (SIGTERM)** to clean up resources.

**10 Dev/Prod Parity - Keep Development, Staging, and Production as Similar as Possible**

* Use **identical environments** for development, testing, and production.
* Utilize **Docker & Kubernetes** to create uniform environments.
* Example: Run **local Kubernetes cluster using Minikube or Kind**.

**11 Logs - Treat Logs as Event Streams**

* Do not write logs to files; instead, **stream logs to a centralized system** (ELK, Splunk, CloudWatch).
* Use **structured logging** with JSON format.
* Example: **Spring Boot Logging via Logback**

**1️2 Admin Processes - Run Admin Tasks as One-Off Processes**

* Run **database migrations, backups, data processing as separate tasks**.
* Use **Spring Boot’s Flyway or Liquibase** for database migrations.
* Example:  
  ✅ **Running DB Migration**

flyway migrate -url=jdbc:mysql://db-host -user=root -password=admin

#### **Service Registry**

* A **Service Registry** is a database or repository that keeps track of available microservices and their network locations (IP addresses & ports).
* It helps in dynamically locating services instead of using hardcoded URLs.
* Example: **Eureka Server (Netflix), Consul, Zookeeper**

@**EnableEurekaClient or @EnableDiscoveryClient**

* Microservices **register** with Eureka and allowing other microservices to discover and interact with each other dynamically.

@**EnableEurekaServer**

* @**EnableEurekaServer** annotation in Spring Cloud is used to create Eureka Server, which acts as a service registry for microservices. This enables **service discovery**, allowing microservices to dynamically register themselves and communicate with each other.

**Client-Side Service Discovery**

* The client (or API Gateway) queries the Service Registry and picks a service instance.
* The client handles load balancing.
* **Example**: Netflix Eureka, Consul  
  **Workflow**:
* Service registers itself with Eureka.
* The client queries Eureka to find available instances.
* The client directly calls the service.

@**EnableDiscoveryClient**

* Microservices **register** with Eureka

**Server-Side Service Discovery**

* The client requests a service via a **Load Balancer**, which queries the Service Registry.
* The **Load Balancer** selects the best service instance and forwards the request.
* **Example:** AWS ALB, Kubernetes Service Discovery  
  **Workflow:**
* The client sends a request to a **Load Balancer**.
* The Load Balancer queries the **Service Registry**.
* It routes the request to the appropriate instance.

A **microservice** can discover other services dynamically:

@**LoadBalanced**

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

@Autowired

Using **Eureka Discovery Client**:

private DiscoveryClient discoveryClient;

public String getServiceUrl() {

List<ServiceInstance> instances = discoveryClient.getInstances("PAYMENT-SERVICE");

return instances.get(0).getUri().toString();

}

**Benefits of Service Registry & Discovery**

✅ **Dynamic Service Registration** – No need for hardcoded URLs.  
✅ **Load Balancing Support** – Services are distributed evenly.  
✅ **Fault Tolerance** – Automatically removes unhealthy services.  
✅ **Scalability** – Easily scales up/down microservices.

**Popular Service Registry Tools**

| **Service Registry Tool** | **Used In** |
| --- | --- |
| Netflix Eureka | Spring Cloud |
| HashiCorp Consul | Multi-cloud setups |
| Apache Zookeeper | Distributed applications |
| Kubernetes Service Registry | Containerized applications |
|  |  |

**Spring Cloud annotations and configuration:**

* **@EnableEurekaServer** annotation allows us to register microservices to the spring cloud.
* **@EnableDiscoveryClient** annotation also allows us to query Discovery server to find miroservices.
* **@LoadBalanced** Spring provides smart RestTemplate for service discovery and load balancing by using **@LoadBalanced** annotation with RestTemplate instance.

**Netflix Ribbon:** it provides several algorithms for Client-Side Load Balancing.

**Zuul:**

* Zuul is a JVM based router and server-side load balancer by Netflix
* Provides a unified access to multiple different microservices
* Load balances across multiple service instances

**Eureka**: Eureka is the Netflix Service Discovery Server and Client. Eureka Server is using Spring Cloud.

**AOP use cases**

* Logging and Tracing
* Log execution times
* Security
* Caching
* Error Handling
* Performance Monitoring
* Custom Business Rules
* Transaction Management

**Aspect**  
– A module that encapsulates pointcuts and advice

**Pointcut**  
– An expression that selects one or more Join Points. – Indicate which method should be intercept, by method name or regular expression pattern.  
**Join Point**  
– A point in the execution of a program such as a method call or exception thrown  
**Advice**  
– Code to be executed at each selected Join Point

**Around advice:** Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

**Spring Boot Annotations**

**@Configuration** Indicates that a class declares one or more @Bean methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime

**@Component** Indicates that an annotated class is a "component". Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.

**@Configuration** is meta-annotated with @Component, therefore @Configuration classes are candidates for component scanning.

**@Qualifier**

The @Qualifier annotation is used to control which bean should be autowire on a field. For example, bean configuration file with two similar person beans.

@SpringBootApplication

The @SpringBootApplication  equivalent to using @Configuration, @EnableAutoConfiguration and @ComponentScan

**How Autowring works in Spring?**

The autowiring happens when the application starts up. So, in Controller, which for arguments sake wants to use the UserServiceImpl class, you'd annotate it as follows:

public class Controller {

// You could also annotate the setUserService method instead of this

@Autowired

private UserService userService;

// rest of class goes here

}

When it sees @Autowired, Spring will look for a class that matches the property in the applicationContext, and inject it automatically. If you have more than one UserService bean, then you'll have to qualify which one it should use.

If you do the following:

UserService service = new UserServiceImpl();

It will not pick up the @Autowired unless you set it yourself.

Autowiring through @Autowired is performed by a BeanPostProcessor implementation, specifically [org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/AutowiredAnnotationBeanPostProcessor.html).

This BeanPostProcessor processes every bean, will scan its class (and superclasses) for any @Autowired annotations, and, depending on what is annotation (a constructor, field, or method), it will take appropriate action.

@ComponentScan to find your beans, in combination with @Autowired constructor injection works well.

If you structure your code as suggested above (locating your application class in a root package), you can add @ComponentScan without any arguments. All of your application components (@Component, @Service, @Repository, @Controller etc.) will be automatically registered as Spring Beans.

When it sees @Autowired, Spring will look for a class that matches the property in the applicationContext, and inject it automatically. If you have more than 1 UserService bean, then you'll have to qualify which one it should use.

By default, the @Autowired will perform the dependency checking to make sure the property has been wired properly. When Spring can’t find a matching bean to wire, it will throw an exception. To fix it, you can disable this checking feature by setting the “**required**” attribute of @Autowired to false.

**Spring boot annotations:**

### **@Bean**

The **@Bean** annotations are used at the method level and indicate that a method produces a bean that is to be managed by the Spring container. It is an alternative to the XML<bean> tag.

### **@Service**

It is used at the class level. It shows that the annotated class is a service class, such as business basic logic, and call external APIs.

### **@Repository**

It is a Data Access Object (DAO) that accesses the database directly. It indicates that the annotated class is a repository.

### **@Configuration**

It is used as a source of bean definitions. It is a **class-level annotation**.

### **@Controller**

The annotation is used to indicate that the class is a **web request handler**. It is often used to present web pages. It is most commonly used with @RequestMapping annotation.

### **@RequestMapping**

RequestMapping is used to map the HTTP request. It is used with the class as well as the method. It has many other optional elements like consumes, name, method, request, path, etc.

### **@Autowired**

This annotation is used to auto-wire spring bean on setter methods, constructor and instance variable. It injects object dependency implicitly. When we use this annotation, the spring container auto-wires the bean by its matching data type.

### **@Component**

It is a class-level annotation that turns the class into Spring bean at the auto-scan time.

### **@EnableAutoConfiguration**

### It is placed on the main application class. Based on classpath settings, other beans, and various property settings, this annotation instructs SpringBoot to start adding beans.

### **@ComponetScan**

It is used to scan a package of beans. It is used with the annotation @Configuration to allow Spring to know the packages to be scanned for annotated components.

### **@Required**

This annotation is applied to bean setter methods. It indicates that the required property must be filled at the configuration time in the affected bean, or else it throws an exception: BeanInitializationException.

### **@Qualifier**

It is used along with @Autowired annotation. It is used when more control is required over the dependency injection process. Individual constructor arguments or method parameters can be specified by using this annotation. Confusion arises when more than one bean of the same type is created, and only one of them is to be wired with a property, @Qualifier is used to get rid of the confusion.

### **@CookieValue**

It is used at the method parameter level as an argument of the request mapping method. For a given cookie name, the HTTP cookie is bound to a @CookieValue parameter.

### **@Lazy**

It is used in the component class. At startup, all auto-wired dependencies are created and configured. But a @Lazy annotation can be created if a bean is to be initialized lazily. This means that only if it is requested for a bean will be created. It can also be used on @Configuartion classes. It’s an indication that all @Bean methods within that @Configuration should be lazily initialized.

Micro Services communication Ways

REST API Based:

Inter Process Communication: (gRPC)

* **Synchronous communication**: (Mostly REST API based) In this pattern, a service calls an API exposed by another service and waits for the response

1. Rest API based
2. gRPC (Remote Procedure Call)

* **Asynchronous communication:** In this pattern, a service sends a message, without waiting for a response from another service. And, one or more services can process the message

**Example:** Queuing messaging technologies like RabbitMQ and Kafka

**gRPC**: **gRPC** is designed for low latency and high throughput communication. As discussed above, it works very well for microservices where efficiency and latency are critical.

* As gRPC heavily uses HTTP/2, it is impossible to call a gRPC service from a web browser directly
* Modern browsers do not provide the control needed over web requests to support a gRPC client. Hence, a proxy layer and gRPC-web are required to perform conversions between HTTP/1.1 and HTTP/2

Graphical user interface

Description automatically generated with medium confidence

* Service Definition
* gRPC Server
* gRPC Client

**Implement Spring Cloud Config Server:**

* It promotes code reusability and removes code repetition. By using Spring Cloud Config Server, we have a centralized configuration. Therefore, we don’t need to write the same set of properties in multiple microservices. Instead, write them at one place and access from any microservice.
* On the other hand, we don’t need to redeploy applications in case we modify any configuration.
* It acts as a central configuration server that provides configurations (properties) to each microservice connected to it.
* Moreover, it significantly simplifies the management of many microservices by centralizing their configuration in one location.
* We apply **@EnableConfigServer** annotation at the main class of the Application to recognize that this application will act as a **Spring Cloud Config Server**
* **@RefreshScope** annotation can be used to load properties in dynamically without restarting the server can be add to Controller class and main class of the application.
* Config server takes highest priority when same key is present in micro service configuration file and config server with different values.

**Eureka Server:**

### Apply Annotation **@EnableEurekaServer** at the main class

* **server.port=8761**
* **eureka.client.register-with-eureka=false**
* **eureka.client.fetch-registry=false**

**Config Server:**

### Apply Annotation **@EnableConfigServer** at the main class

* # Server port
* **server.port=8888**
* # Repository Location in Github
* **spring.cloud.config.server.git.uri=https://github.com/username/config\_repo.git**
* # Github username
* **spring.cloud.config.server.git.username=yourUserName**
* # Github Password
* **spring.cloud.config.server.git.password=yourPassword**
* # Github default branch
* **spring.cloud.config.server.git.default-label=main**

**Config Client:**

* Apply Annotation **@EnableEurekaClient** at the main class
* # port
* **server.port=9940**
* # serviceId (application-name)
* **spring.application.name=PRODUCT-SERVICE**
* # Eureka Location
* **eureka.client.service-url.defaultZone=http://localhost:8761/eureka**
* #Config Server location
* **spring.config.import=optional:configserver:http://localhost:8888**

Communicating micro services

* Using REST API using @RestTemplate
* Using Spring Cloud Feign Client
* Use @EnableFeignClients annotation to use this.

@FeignClient(value = "DEPARTMENT-SERVICE", url = "http://localhost:8080")

public interface APIClient {

@GetMapping(value = "/api/departments/{id}")

DepartmentDto getDepartmentById(@PathVariable("id") Long departmentId);}

**Monitoring Micro services**:

* Using Spring actuator health status

We can implement custom health Indicators only for specific services to maintain health statuses

@Component

@Slf4j

public class UrlShortenerServiceHealthIndicator

implements HealthIndicator {

private static final String URL

= "https://cleanuri.com/api/v1/shorten";

@Override

public Health health() {

// check if url shortener service url is reachable

try (Socket socket =

new Socket(new java.net.URL(URL).getHost(),80)) {

} catch (Exception e) {

log.warn("Failed to connect to: {}",URL);

return Health.down()

.withDetail("error", e.getMessage())

.build();

}

return Health.up().build();

}

}

* Using Hystrix Dashboard, Turbine, Actuator also can monitor micro services in the dashboard

@SpringBootApplication

@**EnableHystrixDashboard**

@**EnableTurbine**

public class MonitorApplication {

...

}

Remove the application.properties file and create a new bootstrap.yml file with the following content:

spring:

application.name: monitor-application

cloud.config:

discovery:

enabled: true

serviceId: config-server

eureka:

client:

serviceUrl:

defaultZone: http://localhost:7001/eureka/

registryFetchIntervalSeconds: 1

instance:

leaseRenewalIntervalInSeconds: 1

Add a new monitor-application.yml file to your [external configuration](https://vaadin.com/blog/microservices-externalized-configuration) with the following content:

endpoints.health.enabled: true

turbine:

appConfig: biz-application, admin-application, news-application, proxy-server

clusterNameExpression: "'default'"

instanceUrlSuffix: application/hystrix.stream

Point your browser to <http://localhost:9801/hystrix> and introduce <http://localhost:9801/turbine.stream>

* **Micrometer** is a library for collecting metrics from JVM-based applications and converting them in a format accepted by the monitoring tools
* <dependency>
* <groupId>io.micrometer</groupId>
* <artifactId>micrometer-registry-prometheus</artifactId>
* </dependency>

Add property in application.properties

**management.endpoints.web.exposure.include=health,info,prometheus**

Add configurations in [prometheus-config.yml](https://github.com/thombergs/code-examples/tree/master/spring-boot/spring-boot-health-check/prometheus-config.yml).

- job\_name: 'user sign up'

metrics\_path: '/actuator/prometheus'

scrape\_interval: 5s

static\_configs:

- targets: ['<HOST\_NAME>:8080']

Now we can check our application as a target in Prometheus by visiting the URL - **http://localhost:9090/targets**

* Using Hystrix Monitor
* Using by adding dependency with micrometer-registry-newrelic
* Using AppDynamics Tool where we need to implement health rule for all services
* Using DynaTrace Tool
* Using Watch Dog
* Integrate Prometheus into Kubernetes

In a microservices architecture, services often need to communicate with each other to fulfill client requests or exchange data. There are several approaches for service-to-service communication, each with its own benefits and use cases:

* **HTTP/REST**: The most common approach is using HTTP with a RESTful API. Services expose RESTful endpoints, and other services or clients make HTTP requests interact with them. This approach is simple, widely understood, and easy to implement, making it a popular choice for microservices communication.
* **gRPC:** gRPC is an RPC (remote procedure call) framework developed by Google. It uses Protocol Buffers for serialization and offers high performance, bi-directional streaming, and support for multiple programming languages. gRPC is well-suited for scenarios requiring high throughput and low latency.
* **Message brokers**: Message brokers like RabbitMQ and Apache Kafka facilitate asynchronous communication between services. Services publish messages to the broker and other services consume those messages. This decouples services and allows them to communicate in an event-driven manner.
* **GraphQL**: GraphQL is an alternative to REST that allows clients to request exactly the data they need, enabling efficient and flexible data retrieval. It reduces over-fetching and under-fetching of data which provides more control to clients.
* **Service mesh:** Service mesh solutions like Istio and Linkerd provide built-in service-to-service communication features including load balancing, service discovery, and encryption. They also offer advanced traffic management and observability
* **WebSocket** allows bidirectional, full-duplex communication between clients and services, making it suitable for real-time applications like chat, notifications, and collaborative tools.
* **Peer-to-peer**: In some cases, direct peer-to-peer communication between services may be appropriate, especially in small, tightly-coupled microservices environments.

The choice of communication approach depends on factors such as the nature of the application, scalability requirements, latency constraints, and the team's familiarity with the technology.

**Micro services Interview Questions**

Microservices architecture allows to avoid monolith application for large system. It provides loose coupling between collaborating processes which running independently in different environments with tight cohesion.

**Microservices Advantages**

* Smaller code base is easy to maintain.
* Easy to scale as individual component.
* Technology diversity i.e. we can mix libraries, databases, frameworks etc.
* Fault isolation i.e. a process of self-checking a system against threats and vulnerabilities.
* Better support for smaller and parallel team.
* Independent deployment
* Deployment time reduce

**Microservices Disadvantages**

* Difficult to achieve strong consistency across services
* ACID transactions do not span multiple processes.
* Distributed System so hard to debug and trace the issues
* Greater need for end to end testing
* Required cultural changes in across teams like Dev and Ops working together even in same team.
* Microservices architecture involves multiple tools and technologies to develop. So, architecture becomes very big and it takes a bit more development time as compared to the monolithic architecture.
* Maintenance of Microservices Architecture based applications is also very Complex and Cost Effective. Multiple Log files, Admin UI (Actuator), Communication chains, Debug Application etc
* Even Runtime Environment needs very high configuration and tools such as AWS Cloud, Google Cloud, Azure Cloud, PCF Cloud etc. in Microservices Architecture.

**Improve performance and Optimizations in Spring boot**

1. **Optimize Database Queries**

* **Use pagination**: When fetching large datasets, always paginate the results to avoid overwhelming the server with unnecessary data.
* **Indexing**: Ensure proper indexing on database tables for the most frequently queried columns to speed up read operations.
* **Lazy loading**: Use lazy loading for associations in JPA to avoid loading unnecessary data.
* **Efficient queries**: Optimize the queries themselves. Use @Query annotations or Criteria API to ensure that only the required data is being fetched.
* **Batching**: If you are inserting or updating large amounts of data, use batch processing to reduce the number of database hits.
* **Connection pooling**: Make sure you are using a connection pool like HikariCP (which is the default in Spring Boot) to reduce connection overhead.

2. **Use Caching Effectively**

* **Use Spring Cache**: Leverage Spring’s caching abstraction (@Cacheable, @CachePut, @CacheEvict) to cache method results.
* **Use distributed caching**: If you have a distributed application, use Redis, Memcached, or another distributed caching solution.
* **Evict cache appropriately**: Ensure that the cache is evicted or updated correctly when data changes.

3. **Optimize Spring Boot Startup Time**

* **Use Spring Boot DevTools**: DevTools helps with performance during development by enabling features like hot reloading.
* **Profile your application**: Use spring.profiles.active to load only the necessary configurations for the environment you're running in.
* **Use Lazy Initialization**: Spring Boot 2.2+ allows you to enable lazy initialization of beans to avoid initializing all beans at startup, reducing the startup time.

To enable lazy initialization in application.properties:

spring.main.lazy-initialization=true

4. **Asynchronous Processing**

* Use @Async to run methods asynchronously, such as sending emails or processing large tasks.
* Use ExecutorService or configure custom thread pools for better control over the concurrency of background tasks.

5. **Optimize Memory Usage**

* **Heap Dump Analysis**: Use tools like VisualVM or YourKit to analyze the heap dumps and identify memory leaks or unnecessary memory usage.
* **Avoid excessive object creation**: Avoid unnecessary object creation or use memory-efficient data structures.
* **Set JVM heap sizes**: Set appropriate JVM heap sizes for your application using -Xms and -Xmx flags.

6. **Optimize Logging**

* **Use asynchronous logging**: Use an asynchronous logging framework such as Logback’s AsyncAppender or the log4j2 async logger to avoid blocking the main application thread.
* **Reduce logging levels**: Set the logging level to INFO or WARN in production environments. Debug-level logging can significantly impact performance.

7. **Optimize Spring Boot Configuration**

* **Connection pool size**: For databases, set the connection pool size (e.g., for HikariCP) based on the expected load.
* **HTTP request handling**: Configure your server (Tomcat, Undertow, Jetty) for optimal performance by adjusting settings like thread pools, connection timeouts, and request handling.
* **JVM tuning**: Optimize the JVM’s garbage collection settings. You can experiment with different garbage collectors (e.g., G1, ZGC, etc.) for better performance based on your use case.

8. **Enable HTTP/2 and Optimize Network Communication**

* **Use HTTP/2**: Enable HTTP/2 on your server for better performance, especially for APIs that involve many small requests.
* server.http2.enabled=true in spring boot

9. **Use Microservices and Decompose the Application**

* Use tools like **Prometheus** and **Grafana** for monitoring.
* Profile your application regularly using **Spring Boot Actuator** to track performance metrics like response times, memory usage, and HTTP request statistics.

10. **Leverage the Right Hardware/Cloud Resources**

* Scale your application horizontally (add more instances).
* Optimize your cloud resources (e.g., autoscaling, right-sizing instances).
* Use load balancers and reverse proxies (e.g., Nginx, HAProxy)