**Can you explain the difference between @Qualifier and @Primary annotations in Spring?**

Sure, I'd be happy to provide some examples to help illustrate the difference between @Qualifier and @Primary annotations in Spring.

Can you explain the difference between @Qualifier and @Primary annotations in Spring?

@Qualifier and @Primary are both used to specify which bean to inject when multiple beans of the same type are defined in a Spring application context. The main difference between the two is that @Qualifier allows you to specify a unique identifier for a bean, while @Primary indicates a default or preferred bean of a certain type.

For example, suppose you have the following two beans defined in your Spring application context:

java

Copy code

@Bean

public BookService bookServiceA() {

return new BookServiceImpl();

}

@Bean

public BookService bookServiceB() {

return new BookServiceImpl();

}

Both of these beans are of type BookService, but they have different names (bookServiceA and bookServiceB). If you want to inject bookServiceA into another bean, you can use the @Qualifier annotation to specify its unique identifier:

java

Copy code

@Autowired

@Qualifier("bookServiceA")

private BookService bookService;

On the other hand, if you want to indicate a default or preferred BookService bean to be injected when no qualifier is specified, you can use the @Primary annotation:

java

Copy code

@Bean

@Primary

public BookService defaultBookService() {

return new BookServiceImpl();

}

Now, when you inject a BookService bean without specifying a qualifier, Spring will automatically inject the bean marked with @Primary.

**Can you give some tricky question on Configuration annotation in Spring?**

Sure, here are some tricky questions on @Configuration annotation in Spring:

Can you have more than one @Configuration class in a Spring application context? If so, how do you ensure that all configurations are loaded?

Yes, you can have more than one @Configuration class in a Spring application context. To ensure that all configurations are loaded, you can either:

Annotate your main application class with @SpringBootApplication, which will scan for and load all @Configuration classes in the same package and its sub-packages.

Create a @Configuration class that imports all other @Configuration classes using the @Import annotation.

For example:

@Configuration

@Import({DatabaseConfig.class, SecurityConfig.class, WebConfig.class})

public class AppConfig {

// ...

}

Here, AppConfig imports three other @Configuration classes (DatabaseConfig, SecurityConfig, and WebConfig), ensuring that all configurations are loaded.

Can you use @Autowired to inject a bean into a @Configuration class?

Yes, you can use @Autowired to inject a bean into a @Configuration class. However, the bean must be defined in the same or a parent application context. If the bean is defined in a child context, it will not be available for injection in the @Configuration class.

For example:

@Configuration

public class AppConfig {

@Autowired

private DataSource dataSource;

// ...

}

Here, AppConfig injects a DataSource bean using @Autowired.

Can you use @Value to inject a property into a @Configuration class?

Yes, you can use @Value to inject a property into a @Configuration class. However, the property must be defined in a properties file or environment variable, and the properties file must be loaded into the application context using the @PropertySource annotation.

For example:

@Configuration

@PropertySource("classpath:config.properties")

public class AppConfig {

@Value("${app.name}")

private String appName;

// ...

}

Here, AppConfig injects a property named app.name from the config.properties file using @Value.

Can you use @Bean to define a bean in a @Configuration class that depends on another bean defined in the same class?

Yes, you can use @Bean to define a bean in a @Configuration class that depends on another bean defined in the same class. However, you must ensure that the dependent bean is defined before the bean that depends on it.

For example:

@Configuration

public class AppConfig {

@Bean

public DataSource dataSource() {

// ...

}

@Bean

public JdbcTemplate jdbcTemplate() {

return new JdbcTemplate(dataSource());

}

// ...

}

Here, AppConfig defines a DataSource bean using the dataSource() method, and a JdbcTemplate bean using the jdbcTemplate() method, which depends on the dataSource() bean. By calling dataSource() within the jdbcTemplate() method, we ensure that the dataSource() bean is defined before the jdbcTemplate() bean that depends on it.

I hope these examples help clarify some of the intricacies of using @Configuration in Spring!

**Can you give me some tricky questions on Spring application context?**

Sure, here are some tricky questions on Spring application context:

What is the difference between a BeanFactory and an ApplicationContext?

Both BeanFactory and ApplicationContext are interfaces in Spring that provide the ability to manage beans. The main difference between them is that BeanFactory provides a basic set of functionality for managing beans, while ApplicationContext adds more advanced features such as internationalization, event propagation, and AOP.

ApplicationContext is a subinterface of BeanFactory, which means that all ApplicationContext implementations are also BeanFactory implementations. However, not all BeanFactory implementations are ApplicationContext implementations.

Can you have multiple application contexts in a Spring application?

Yes, you can have multiple application contexts in a Spring application. However, each context will be independent of the others, and beans defined in one context will not be visible to beans in another context, unless you explicitly configure them to share beans.

To share beans between multiple contexts, you can create a parent-child relationship between them using the parent attribute of the ContextLoaderListener or DispatcherServlet in your web.xml file, or by using the @ImportResource or @Import annotations in your configuration classes.

What is the difference between a singleton and a prototype bean scope?

In Spring, a bean scope determines how many instances of a bean are created and managed by the application context. The two most common bean scopes are singleton and prototype.

A singleton bean scope means that there is only one instance of the bean created and managed by the application context. This instance is shared by all other beans that reference it, and is created when the application context is started.

A prototype bean scope means that a new instance of the bean is created every time it is requested from the application context. This instance is not shared by other beans, and is created each time it is requested.

What is lazy initialization of beans in Spring?

Lazy initialization of beans means that the bean is not created until it is first requested from the application context. This can help to reduce startup time and memory usage, especially for applications with a large number of beans.

To enable lazy initialization for a bean, you can set the lazy-init attribute of the @Bean or @Component annotation to true, or set the default-lazy-init attribute of the <beans> element in your XML configuration file to true.

For example:

@Component

@Lazy(true)

public class MyLazyBean {

// ...

}

Here, MyLazyBean is annotated with @Lazy(true), which means that it will not be created until it is first requested from the application context.

I hope these examples help clarify some of the intricacies of working with Spring application context!

**Are IOC container and Spring application context the same thing?**

The terms "IOC container" and "Spring application context" are often used interchangeably, but they do have slightly different meanings.

In Spring, an IOC (Inversion of Control) container is responsible for managing the lifecycle of objects (beans) and their dependencies. It creates and manages the objects, and provides them to other parts of the application that need them. The IOC container is also responsible for injecting dependencies into the beans, using techniques such as constructor injection or setter injection.

The Spring application context is an implementation of the IOC container. It provides a way to manage and access beans within a Spring application. It is responsible for creating, configuring, and assembling the beans, and provides a way for the beans to interact with each other. The application context is also responsible for handling transactions, security, and other aspects of the application.

So, while the terms "IOC container" and "Spring application context" are often used interchangeably, the Spring application context is actually an implementation of the IOC container.

**What is Order annotation? Can you explain its real use case in Spring?**

In Spring, the @Order annotation is used to specify the order in which multiple beans should be processed, especially when they are defined by different configurations or modules.

When multiple beans of the same type are present in the application context, Spring needs a way to determine the order in which they should be processed. The @Order annotation provides a simple way to specify this ordering.

The @Order annotation can be applied to a class or method, and it takes a single integer value as a parameter. This integer value represents the order in which the bean should be processed. Lower values are processed first.

For example, let's say we have two beans of the same type:

@Bean

public MyBean myBean1() {

return new MyBean();

}

@Bean

public MyBean myBean2() {

return new MyBean();

}

If we want myBean1 to be processed before myBean2, we can add the @Order annotation to the MyBean class and specify the desired order:

@Order(1)

public class MyBean {

// ...

}

Alternatively, we can add the @Order annotation to the methods that create the beans:

@Bean

@Order(1)

public MyBean myBean1() {

return new MyBean();

}

@Bean

@Order(2)

public MyBean myBean2() {

return new MyBean();

}

In this case, the order is explicitly specified by the integer value passed to the @Order annotation.

The @Order annotation is often used in conjunction with the @Autowired annotation to specify the order in which dependencies should be injected. For example, if we have multiple beans of the same type and want to specify the order in which they are injected, we can use the @Order annotation on the constructor parameters:

@Component

public class MyComponent {

private final MyBean myBean1;

private final MyBean myBean2;

@Autowired

public MyComponent(@Order(1) MyBean myBean1, @Order(2) MyBean myBean2) {

this.myBean1 = myBean1;

this.myBean2 = myBean2;

}

// ...

}

In this example, the @Order annotation is used to specify the order in which myBean1 and myBean2 should be injected into the MyComponent constructor.

**Give me list of functional interfaces introduced in Java 8 with examples?**

Java 8 introduced several functional interfaces in the java.util.function package to support functional programming with lambda expressions and method references. Here's a list of some commonly used functional interfaces and their examples:

**1. Function /BiFunction**

Function<T, R> - takes a parameter of type T and returns a result of type R.

Function<String, Integer> strLength = (s) -> s.length();

Integer len = strLength.apply("Hello"); // returns 5

BiFunction<T, U, R> - takes two parameters of types T and U, and returns a result of type R.

BiFunction<Integer, Integer, Integer> add = (x, y) -> x + y;

int result = add.apply(2, 3); // returns 5

**2. Consumer / Supplier**

Consumer<T> - takes a parameter of type T and returns no result.

Consumer<String> printStr = (s) -> System.out.println(s);

printStr.accept("Hello"); // prints "Hello"

Supplier<T> - takes no parameters and returns a result of type T.

Supplier<Integer> randomNumber = () -> (int) (Math.random() \* 100);

int num = randomNumber.get(); // returns a random integer between 0 and 99

**3. Predicate**

Predicate<T> - takes a parameter of type T and returns a boolean value.

Predicate<Integer> isPositive = (i) -> i > 0;

boolean result = isPositive.test(5); // returns true

**4. UnaryOperator/ BinaryOperator**

UnaryOperator<T> - takes a parameter of type T and returns a result of type T.

UnaryOperator<Integer> square = (i) -> i \* i;

int result = square.apply(5); // returns 25

BinaryOperator<T> - takes two parameters of type T and returns a result of type T.

BinaryOperator<Integer> add = (x, y) -> x + y;

int result = add.apply(2, 3); // returns 5

These are just a few examples of functional interfaces introduced in Java 8. There are several others available in the java.util.function package that can be used for different purposes in functional programming.

**What are real use case where we can use annoation PostConstruct and interface InitializingBean ?**

Both @PostConstruct and InitializingBean are used to perform initialization logic for a bean after it has been instantiated and its dependencies have been injected. Here are some real-world use cases where you might use each of them:

@PostConstruct:

Setting default values for fields

Initializing a cache or a pool of resources

Registering a bean with a JMX server

Starting a thread

For example, let's say we have a MyBean class that needs to perform some initialization after it has been instantiated and its dependencies have been injected. We can use the @PostConstruct annotation to mark the initialization method:

@Component

public class MyBean {

private SomeDependency someDependency;

@PostConstruct

public void init() {

// Initialization logic

someDependency.initialize();

}

// ...

}

In this example, the init() method is called after the bean is instantiated and its dependencies are injected. It initializes the someDependency field by calling its initialize() method.

InitializingBean:

Performing validation on properties after they have been set

Connecting to a database or a remote service

Setting up a JMS listener container

Registering a message listener

For example, let's say we have a MyBean class that needs to perform some initialization after its properties have been set by Spring. We can implement the InitializingBean interface and override the afterPropertiesSet() method to perform the initialization logic:

@Component

public class MyBean implements InitializingBean {

private String someProperty;

@Override

public void afterPropertiesSet() throws Exception {

// Initialization logic

if (someProperty == null) {

throw new IllegalArgumentException("someProperty must be set");

}

}

// ...

}

In this example, the afterPropertiesSet() method is called after the bean's properties have been set by Spring. It performs validation on the someProperty field and throws an exception if it is null.

**What is the purpose of BeanPostProcessor interface in Spring? Can you give some real use cases?**

The BeanPostProcessor interface in Spring provides a mechanism for processing beans before and after they are instantiated by the Spring container. It is a callback interface that provides hooks for performing custom initialization and destruction logic for beans.

The BeanPostProcessor interface defines two methods:

postProcessBeforeInitialization(Object bean, String beanName): This method is called before the initialization of the bean. It allows you to perform any custom processing of the bean before it is fully initialized. You can modify the bean instance, or even return a different bean instance altogether.

postProcessAfterInitialization(Object bean, String beanName): This method is called after the initialization of the bean. It allows you to perform any custom processing of the bean after it is fully initialized. You can modify the bean instance or its properties.

Some real-world use cases where you might use a BeanPostProcessor are:

**Custom annotation processing**: You can use a BeanPostProcessor to process custom annotations that apply to your beans. For example, you can create a custom annotation that marks a bean as transactional, and then use a BeanPostProcessor to automatically wrap the bean in a transactional proxy.

**Property encryption**: You can use a BeanPostProcessor to automatically encrypt properties of a bean that contain sensitive information, such as passwords or API keys. You can define a custom annotation that marks a property as sensitive, and then use a BeanPostProcessor to automatically encrypt and decrypt the property value.

**Custom logging**: You can use a BeanPostProcessor to automatically log the creation and destruction of beans in your application. This can be useful for debugging and performance monitoring.

**Custom proxy creation**: You can use a BeanPostProcessor to create custom proxies for your beans. For example, you can create a custom proxy that caches the results of expensive method calls, or that adds additional security checks to method invocations.

Overall, the BeanPostProcessor interface is a powerful mechanism for customizing the behavior of your beans in Spring. It allows you to perform custom initialization and destruction logic, and to modify the beans before and after they are fully initialized.

**Can you explain putIfAbsent method in Java with an examples?**

putIfAbsent() is a method of the java.util.Map interface in Java that adds a key-value pair to the map only if the key does not already exist. If the key already exists, the method returns the existing value associated with that key.

The signature of the putIfAbsent() method is:

V putIfAbsent(K key, V value)

Here is an example of how to use putIfAbsent():

Map<String, Integer> map = new HashMap<>();

map.put("apple", 1);

map.put("banana", 2);

// add a new key-value pair to the map

Integer value = map.putIfAbsent("cherry", 3);

System.out.println(value); // prints "null"

// add a key-value pair to the map, but don't overwrite the existing value for "apple"

value = map.putIfAbsent("apple", 4);

System.out.println(value); // prints "1"

System.out.println(map); // prints "{apple=1, banana=2, cherry=3}"

In this example, we create a HashMap called map and add two key-value pairs to it. We then use the putIfAbsent() method to add a third key-value pair to the map. Since the key "cherry" does not already exist in the map, the method returns null.

We then use putIfAbsent() again to add a key-value pair to the map, but this time the key "apple" already exists in the map. Therefore, the method does not add a new key-value pair and instead returns the existing value associated with the key "apple".

Finally, we print out the contents of the map to confirm that it contains all three key-value pairs.

**What is the difference between compile and runtime scope in Maven?**

Certainly! Let's consider a hypothetical example of a Java web application built using Maven.

Compile scope example:

Suppose your web application uses the Apache Commons Lang library, which provides various utility classes for string manipulation, array handling, etc. You would declare this dependency in your pom.xml file with a compile scope:

<dependencies>

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-lang3</artifactId>

<version>3.12.0</version>

<scope>compile</scope>

</dependency>

</dependencies>

In this case, the Apache Commons Lang library will be required for the compilation of your source code and will also be included in the final artifact (e.g., JAR file) generated by Maven. Other dependencies of the Apache Commons Lang library, if any, will also be included.

Runtime scope example:

Let's assume that your web application uses the MySQL database for data storage. To connect to the MySQL database, you need a JDBC driver, which is not necessary for the compilation phase but is required during runtime. You can declare this dependency with a runtime scope:

<dependencies>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.27</version>

<scope>runtime</scope>

</dependency>

</dependencies>

With the runtime scope, the MySQL JDBC driver will not be included in the final artifact generated by Maven. However, when you run your web application, the MySQL JDBC driver will be required and should be available in the runtime environment.

By using different dependency scopes, you can manage your project's dependencies effectively. Compile scope ensures that the necessary dependencies are available during the compilation and building phases, while runtime scope allows you to include dependencies required only during the application's execution.

**Can you explain the difference between fail-fast and fail-safe iterator with examples?**

Certainly! The terms "fail-fast" and "fail-safe" refer to different approaches in handling concurrent modifications during iteration in programming. Let's explore each of them with examples:

1. Fail-Fast Iterator:

A fail-fast iterator is designed to immediately throw a `ConcurrentModificationException` if the underlying collection is modified while iterating. It detects any concurrent modifications and prevents the iteration from continuing, ensuring that the iteration operates on a consistent state of the collection.

Example of a Fail-Fast Iterator:

```java

List<String> list = new ArrayList<>();

list.add("Item 1");

list.add("Item 2");

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()) {

String item = iterator.next();

list.remove(item); // Concurrent modification

}

```

In the above example, when `list.remove(item)` is called inside the loop, it modifies the collection while the iterator is still active. As a result, a `ConcurrentModificationException` is thrown, indicating that the collection was modified during iteration.

2. Fail-Safe Iterator:

A fail-safe iterator, on the other hand, handles concurrent modifications gracefully by creating a copy of the original collection or working on a snapshot of the data. This allows the iterator to continue its traversal without throwing an exception. The copied or snapshot data ensures that the iterator operates on a consistent state, regardless of modifications made to the original collection.

Example of a Fail-Safe Iterator:

```java

ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

map.put("Key 1", 1);

map.put("Key 2", 2);

Iterator<Map.Entry<String, Integer>> iterator = map.entrySet().iterator();

while (iterator.hasNext()) {

Map.Entry<String, Integer> entry = iterator.next();

map.remove(entry.getKey()); // No ConcurrentModificationException

}

```

In the above example, `ConcurrentHashMap` provides a fail-safe iterator. Even though we are modifying the `ConcurrentHashMap` by removing entries while iterating, no `ConcurrentModificationException` is thrown. The iterator works on a snapshot of the data and does not encounter any issues due to concurrent modifications.

It's worth noting that fail-safe iterators may not reflect the latest modifications made to the collection during iteration, as they operate on a copy or snapshot. They are typically used in concurrent environments where multiple threads may modify the collection while it's being iterated, ensuring that the iteration is not affected by these modifications.

In summary, the main difference between fail-fast and fail-safe iterators lies in how they handle concurrent modifications. Fail-fast iterators immediately throw an exception when detecting modifications, while fail-safe iterators work on a separate copy or snapshot, allowing modifications to the underlying collection without throwing exceptions during iteration.

**How do you answer the question "Can you explain me architecture of your current project?".**

When answering the question "Can you explain the architecture of your current project?", you can provide a high-level overview of the architectural design and components of your project. Here's a structured approach to answering this question:

1. Start with a brief introduction: Provide a concise description of the project, its purpose, and its main goals. Mention any relevant background information to help set the context.

2. Describe the overall architecture: Explain the architectural style or pattern used in your project. Common architectural styles include layered architecture, microservices architecture, client-server architecture, or event-driven architecture. Briefly discuss why this architectural style was chosen and how it supports the project's requirements.

3. Discuss the key components: Identify and describe the main components or modules in your project. Highlight the responsibilities and interactions of each component. You can use diagrams or visual aids, if available, to illustrate the relationships between components.

4. Explain the data storage and access: Discuss how data is stored and accessed in your project. Mention the database management system (e.g., PostgreSQL, MySQL, MongoDB) and any caching mechanisms or data repositories utilized. Explain how data flows through the system and how different components interact with the data layer.

5. Discuss the communication mechanisms: Explain how different components or services communicate with each other. Discuss the protocols or technologies used for communication, such as RESTful APIs, message queues, or event buses. Highlight any patterns or frameworks used for inter-component communication, such as publish-subscribe or request-response patterns.

6. Mention external dependencies: Identify any external systems, services, or APIs that your project relies on. Discuss how your project integrates with these external dependencies and any considerations taken to ensure reliability and fault tolerance.

7. Address scalability and performance: Briefly touch upon how your project handles scalability and performance requirements. Mention any architectural decisions or techniques implemented to achieve scalability, such as load balancing, horizontal scaling, or caching strategies.

8. Discuss security and deployment considerations: Explain how security measures are implemented in your project, such as authentication, authorization, and data encryption. Discuss any deployment considerations, such as containerization using Docker, deployment on cloud platforms, or infrastructure as code.

9. Summarize key benefits: Conclude your explanation by summarizing the key benefits and advantages of the project's architecture. Highlight how the chosen architecture supports the project's goals, scalability, maintainability, and future extensibility.

Remember to tailor your explanation to the level of technical understanding of the audience. Provide sufficient detail to convey the architecture effectively without overwhelming the listener with unnecessary technical jargon.

**What are maven command options:**

mvn -T 1C clean install

-DskipTests

-Dmaven.test.skip=true

-Dmaven.site.skip=true

-Dmaven.javadoc.skip=true

**Give me steps to create singleton class in Java?**

To create a singleton class in Java, you can follow these steps:

1. Private Constructor: Make the class constructor private to prevent other classes from directly instantiating it.

2. Static Instance Variable: Declare a static variable of the class type within the class. This variable will hold the single instance of the class.

3. Static Factory Method: Create a static method that provides access to the single instance of the class. This method is responsible for creating the instance if it doesn't exist or returning the existing instance.

4. Lazy Initialization (optional): Optionally, apply lazy initialization to create the instance only when it is requested for the first time. This ensures that the instance is not created unnecessarily.

5. Thread Safety (optional): If your singleton needs to be thread-safe, you can apply synchronization techniques to ensure that only one thread can access the instance creation logic at a time. This prevents multiple instances from being created concurrently.

Here's an example implementation of a singleton class in Java:

```java

public class Singleton {

// Step 1: Private constructor

private Singleton() {

// Initialization logic (if any)

}

// Step 2: Static instance variable

private static Singleton instance;

// Step 3: Static factory method

public static Singleton getInstance() {

// Step 4: Lazy initialization

if (instance == null) {

// Step 5: Thread safety (synchronized block)

synchronized (Singleton.class) {

if (instance == null) {

instance = new Singleton();

}

}

}

return instance;

}

// Other methods and properties of the singleton class

public void doSomething() {

// ...

}

}

```

In the example above, the `getInstance()` method is responsible for creating or returning the single instance of the `Singleton` class. It applies lazy initialization and thread safety by using a double-checked locking mechanism. The first thread that accesses `getInstance()` will create the instance, and subsequent threads will skip the instance creation process.

Note that the double-checked locking technique shown above is suitable for Java versions before Java 5. In Java 5 and later, the volatile keyword can be used to ensure proper synchronization and visibility of the `instance` variable, removing the need for double-checked locking.

**What is advantage of using Enums over Constants in Java? Can you explain each point with an example?**

Certainly! Let's go through each advantage of using enums over constants in Java and illustrate them with examples:

1. Readability and Clarity:

```java

// Constants

public static final int MONDAY = 1;

public static final int TUESDAY = 2;

public static final int WEDNESDAY = 3;

// Enums

public enum Day {

MONDAY, TUESDAY, WEDNESDAY;

}

```

Using enums provides more descriptive and self-explanatory names, making the code more readable. In this case, it is clear that the enum values represent days of the week.

2. Type Safety:

```java

// Constants

public static final int RED = 1;

public static final int GREEN = 2;

public static final int BLUE = 3;

// Enums

public enum Color {

RED, GREEN, BLUE;

}

// Usage

int myColor = Color.RED; // Compilation error: incompatible types

Color myColor = Color.RED; // Correct usage

```

Enums ensure type safety by only allowing assignment of valid enum values. In this example, assigning `Color.RED` to an `int` variable results in a compilation error, preventing incorrect assignments.

3. Compile-Time Checking:

```java

// Constants

public static final int OPTION\_A = 1;

public static final int OPTION\_B = 2;

public static final int OPTION\_C = 3;

// Enums

public enum Option {

OPTION\_A, OPTION\_B, OPTION\_C;

}

// Usage

Option selectedOption = Option.OPTION\_D; // Compilation error: Option.OPTION\_D does not exist

```

Enums are checked at compile-time, catching errors early. Assigning a non-existent enum value like `Option.OPTION\_D` triggers a compilation error, helping identify mistakes during development.

4. Limited Set of Values:

```java

// Constants

public static final int STATUS\_OPEN = 1;

public static final int STATUS\_IN\_PROGRESS = 2;

public static final int STATUS\_COMPLETED = 3;

// Enums

public enum Status {

OPEN, IN\_PROGRESS, COMPLETED;

}

// Usage

int currentStatus = STATUS\_NEW; // Compilation error: STATUS\_NEW does not exist

Status currentStatus = Status.NEW; // Compilation error: Status.NEW does not exist

```

Enums restrict values to a predefined set, preventing assignment of invalid values. Attempting to assign a non-existent enum value results in a compilation error.

5. Improved Maintainability:

```java

// Constants

public static final int GRADE\_A = 1;

public static final int GRADE\_B = 2;

public static final int GRADE\_C = 3;

// Enums

public enum Grade {

A, B, C;

}

// Usage

// Constants

public static final String GRADE\_A\_LABEL = "Excellent";

public static final String GRADE\_B\_LABEL = "Good";

public static final String GRADE\_C\_LABEL = "Average";

// Enums

public enum Grade {

A("Excellent"), B("Good"), C("Average");

private final String label;

private Grade(String label) {

this.label = label;

}

public String getLabel() {

return label;

}

}

// Usage

String gradeALabel = GRADE\_A\_LABEL; // Constants usage

String gradeALabel = Grade.A.getLabel(); // Enums usage

```

Enums provide a more maintainable approach as additional functionality can be added within the enum itself. In this example, using an enum allows associating a label with each grade, enhancing readability and maintainability.

6. Enum Methods and Behavior:

```java

public enum MathOperation {

ADDITION {

@Override

public int calculate(int a, int b) {

return a + b;

}

},

SUBTRACTION {

@Override

public int calculate(int a, int b) {

return a - b;

}

};

public abstract int calculate(int a, int b);

}

// Usage

int result = MathOperation.ADDITION.calculate(5, 3); // 8

```

Enums can have their own methods and behavior. In this example, each enum value represents a math operation, and the `calculate()` method is implemented differently for each enum value.

7. Enhanced IDE Support:

IDEs provide improved support for enums compared to constants. Autocompletion, refactoring, and navigation features are more robust with enums, making development more efficient and error-resistant.

These examples demonstrate the advantages of using enums over constants in Java, showcasing how enums enhance readability, type safety, compile-time checking, maintainability, and functionality.

**What are the key component choices for the microservices fabric stack?**

Routing and discovery:

Netflix OSS:

Eureka, Zuul frameworks, and Ribbon provide this capability. Leverage this if you plan to build microservices in Java and leverage other parts of the Netflix OSS stack and Spring Cloud.

Logging and analytics:

Elastic Search, Logstash, and Kibana: ELK Stack is a popular stack for analytics and logging. This is a good choice for microservices analytics and logging.

Splunk: Fortune 100 enterprises leverage Splunk. Leverage Splunk if it's an organization standard.

Messaging

Kafka

RabbitMQ

Config

Spring Cloud Config

Security:

Basic security: Conventional HTTP Basic Auth that is still widely implemented in applications today, but poorly supported in a distributed microservices architecture.

OAuth2: An open-protocol security implementation that allows a consistent user experience, with common architecture building blocks, across web applications, mobile platforms, and IoT devices. OAuth requires unique architecture components to support implementation, but many open sources and enterprise offerings exist to build security from the ground up.

JSON Web Tokens (JWT): An extension to the OAuth concept and model; JWT is an open standard-based header that allows the actors in a microservices architecture to validate, verify, and generate authorization. This is based on JSON, and this specification is utilized across many non-JavaScript architectures.

Certificate-based: Originally implemented as Secure Sockets Layer (SSL) and more commonly implemented as Transport Layer Security (TLS), certificate-based security allows development teams to secure their architecture components using cryptographically secure keys that ensure communication is secret when going over network interface.

**What id difference between stubs and mocks?**

**What are AMQP tools available with us?**

Apache ActiveMQ

Apache Kafka

Apache RabbitMQ

**Check how retry is implemented in the project?**

**What is OAuth?**

OAuth is a delegated authorization framework for REST/APIs.

**What are server and client certificates?**

**Server certificate**: Server certificates also known as SSL certificates are leveraged to authenticate the identity of a server. When deployed on a website, an SSL certificate turns the protocol on the website from HTTP to HTTPS and installs indicators that guarantee the authenticity of the website. Thus, end users can know the website belongs to the said entity. Apart from authentication, SSL certificates also facilitate Encryption, that is, any information a user sends to the server is protected from any ill-intended entities/third parties.

**Client certificate**: Client certificates are leveraged to validate the identity of a user or the client. The user might be a website user or an email user. Simply, it works as a password, but without any intervention from the end user. In this manner, the server makes sure that it's connecting to the permitted user and that the party is safe to communicate.

**What does one mean by client side load balancing? How is it implemented in Spring Cloud?**

In SpringCloud, this can be implemented using Netflix Ribbon. Spring Cloud-Netflix Eureka + Ribbon Simple Example.

**How to enable server-side load balancing using Spring Cloud?**

Server-side load balancing can be enabled by leveraging the Netflix Zuul tool. Zuul is a JVM-based router and server-side load balancer from Netflix. It provides a single entry to the system, which allows a browser, mobile application, or user interface to consume services from multiple hosts without managing authentication and cross-origin resource sharing (CORS) for each one. We can integrate Zuul with other Netflix projects like Hystrix for fault tolerance and Eureka for service discovery, or leverage it to manage routing rules, filters, and load balancing across your system.

**How does Hystrix handles it when there multiple instances of an API is running in microservice architecture?**

When there are multiple instances of an API running in a microservice architecture, Hystrix handles the resilience and fault tolerance in the following manner:

1. Circuit Breaker Pattern: Hystrix implements the Circuit Breaker pattern, which monitors the calls to the API. If a certain threshold of failures (e.g., timeouts, exceptions, or high error rates) is reached within a defined time window, Hystrix opens the circuit. When the circuit is open, subsequent calls to the API will be short-circuited, meaning they won't even attempt to make the actual request. Instead, Hystrix will immediately return a fallback response without forwarding the request to the API instances. This helps prevent cascading failures and avoids overloading the API instances.

**2. Load Balancing: Hystrix, in combination with a service discovery mechanism like Eureka**, can perform load balancing across the multiple instances of the API. When a request is made to the API, Hystrix will distribute the requests across the available instances based on a load-balancing strategy (e.g., Round Robin or Random). This ensures that the load is distributed evenly among the instances, preventing any single instance from being overwhelmed.

3. Fallback Mechanism: Hystrix allows you to define a fallback mechanism for each API call. If the circuit is open, or if the request to the API instance fails for any reason, Hystrix will execute the fallback logic instead. The fallback logic could be a default value, a cached response, or an alternative service call to a different microservice that provides similar functionality. The fallback mechanism provides a graceful way to handle failures and degradation in the API instances.

4. Timeout Handling: Hystrix allows you to set a timeout for each API call. If a call takes longer than the specified timeout, Hystrix will consider it a failure and trigger the fallback mechanism. This prevents requests from being stuck indefinitely due to unresponsive API instances.

By incorporating Hystrix into a microservice architecture with multiple instances of APIs, you can enhance the resilience and fault tolerance of your system. Hystrix helps prevent failures from propagating across the system, balances the load between instances, provides a fallback mechanism for failed requests, and avoids unnecessary delays caused by unresponsive instances. This overall approach contributes to building more reliable and robust microservices.

**How do you identify the optimum microservices when you are refactoring an existing monolithic application to a microservices architecture?**

One can apply the DDD principles to identify microservices when refactoring an existing monolithic application. One of the initial goals is to identify bounded contexts. Bounded context is a conceptual boundary where a domain model is applicable.

Dependency analysis tools such as **JDepend** may assist in finding system boundaries. The next process step is the decomposition of the monolithic application into microservices components. The goal of this activity is to ensure that each service has a single responsibility and also services should be independently upgradeable and replaceable.

Good practices to follow when conducting this process are as follows:

Segregate databases before separating services.

Ensure existing transactions continue to reside within a single service.

Redesign to leverage compensation and/or eventual consistency.

Build new features as microservices around an existing monolithic application.

It is also important to understand when not to refactor a monolith application into microservices. The main driver to not refactor a monolithic application is when the cost of managing the service outweighs the benefits. Dysfunctional communication patterns and a cultural climate that does not support team autonomy are other factors that will make it hard to adopt microservices.

**How to use JDepend for my maven based project?**

JDepend is a popular tool for analyzing Java code dependencies. To use JDepend in your Maven-based project, you can follow these steps:

Step 1: Add JDepend Plugin to Your Maven Build

In your project's `pom.xml`, add the JDepend plugin to the `<plugins>` section within the `<build>` element. The plugin will be responsible for running JDepend and generating the dependency report.

```xml

<build>

<plugins>

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>jdepend-maven-plugin</artifactId>

<version>2.0.0</version>

<executions>

<execution>

<goals>

<goal>generate</goal>

</goals>

<phase>verify</phase>

</execution>

</executions>

<configuration>

<outputDirectory>${project.build.directory}/jdepend-report</outputDirectory>

</configuration>

</plugin>

</plugins>

</build>

```

Step 2: Run the JDepend Analysis

After adding the JDepend plugin to your `pom.xml`, you can run the JDepend analysis by executing the following Maven command:

```

mvn jdepend:generate

```

The command will generate a JDepend report in the specified output directory (`${project.build.directory}/jdepend-report` in the example above). The report will contain information about the package-level and class-level dependencies in your project.

Step 3: View the JDepend Report

After running the `jdepend:generate` command, you can view the generated JDepend report in the output directory. The report will typically be in HTML format and can be opened in a web browser.

For example, if you used the configuration above, you can find the report at `${project.build.directory}/jdepend-report/index.html`.

The JDepend report will display various metrics and visualizations to help you understand the dependencies in your project. It will show package-level metrics like afferent (incoming) and efferent (outgoing) dependencies, as well as class-level metrics like the number of methods, attributes, and the stability of each class.

By analyzing the JDepend report, you can gain insights into the structure of your project's dependencies, identify potential design issues, and improve the modularity and maintainability of your codebase.

**How do I test the resiliency of an application?**

One of the key aspects of the microservices architecture is that each microservice has its own lifecycle. Each microservice is operated and owned by an autonomous development team. Different teams can independently build, deploy, and manage their respective microservices as long as they ensure API compatibility. This agility, when combined with continuous integration and deployment processes, enables applications to be deployed tens to hundreds of times a day.

Chaos engineering: Chaos engineering addresses the uncertainty of distributed applications at scale. Chaos engineering is the facilitation of experiments to uncover systemic weaknesses. These experiments follow four steps:

Define steady state as a measurable output of an application that indicates normal behavior.

It hypothesizes that the steady state will continue in both, the experimental group and control group.

Introduce attributes for real-world events like hard drives that malfunction, servers that crash, and network connections.

Disprove the hypothesis by looking for a difference in the steady state between the experimental group and control group.

The harder it is to disrupt the steady state, the more confidence one can have in the application. In case a weakness is uncovered, one now has a target for improvement before that behavior manifests in the system at large. Automation is the key to resilience testing as well. Frameworks like Gremlin and Simian Army Chaos Monkey ensures that the application can tolerate random instance failures.

Chaos Monkey: The Chaos Monkey is the first entry in the Netflix technical team's Simian Army. Chaos Monkey randomly terminates virtual machine instances and containers that run inside an application landscape.

**What are the twelve-factor applications?**

Here is the outline of the 12 factors:

**Codebase**: One codebase is tracked in revision control with many deployments.

**Dependencies**: Explicitly declare and isolate dependencies.

**Configuration**: Store configuration in the environment.

**Backing service**: Treat backing services as attached resources.

**Build release run**: Strictly separate build and run stages.

**Processes**: Execute the app as one or more stage fewer processes.

**Port binding**: Export services with port binding.

**Concurrency**: Scale-out using a process model.

**Disposability**: Maximize robustness with fast startup and efficient shutdown.

**Development and production parity**: Keep development, staging, and production as similar as possible.

**Logs**: Treat logs as event streams.

**Admin processes**: Run administrative and management tasks as one-off processes.

What is difference between Sleuth and Xipkin?

Sleuth and Zipkin are both distributed tracing systems used in the field of software development to monitor and troubleshoot microservices-based architectures. They help developers gain insights into the flow of requests between various microservices and identify performance bottlenecks and errors. However, they are separate tools with some differences in their implementation and features.

1. Spring Cloud Sleuth: **Spring Cloud Sleuth is part of the Spring Cloud ecosystem**, which provides tools and libraries to build cloud-native applications. Sleuth is specifically designed for Java applications and integrates seamlessly with Spring Boot microservices. Its main function is to add unique identifiers (trace and span IDs) to each request as it flows through different microservices. These IDs allow developers to trace the entire path of a request as it travels through the microservices architecture.

Key features of Sleuth:

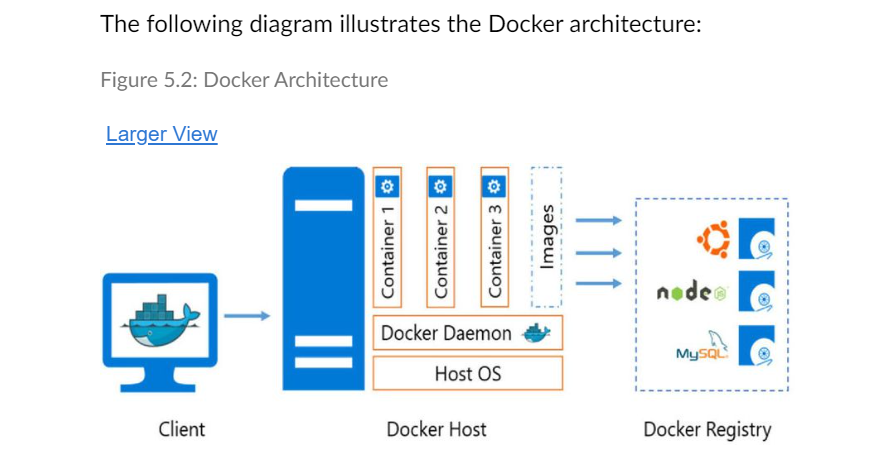
* Integration with Spring Cloud and Spring Boot projects.
* Automatic generation and propagation of trace and span IDs.
* Compatibility with various logging and monitoring tools, such as ELK (Elasticsearch, Logstash, Kibana), Zipkin, etc.

1. Zipkin: **Zipkin is a standalone distributed tracing system** that supports multiple programming languages. Unlike Sleuth, which is tightly integrated with Spring Cloud and Spring Boot, Zipkin can be used with various languages and frameworks. It provides a web-based user interface to visualize traces, spans, and their dependencies, which makes it easier for developers to understand and analyze the performance of their microservices.

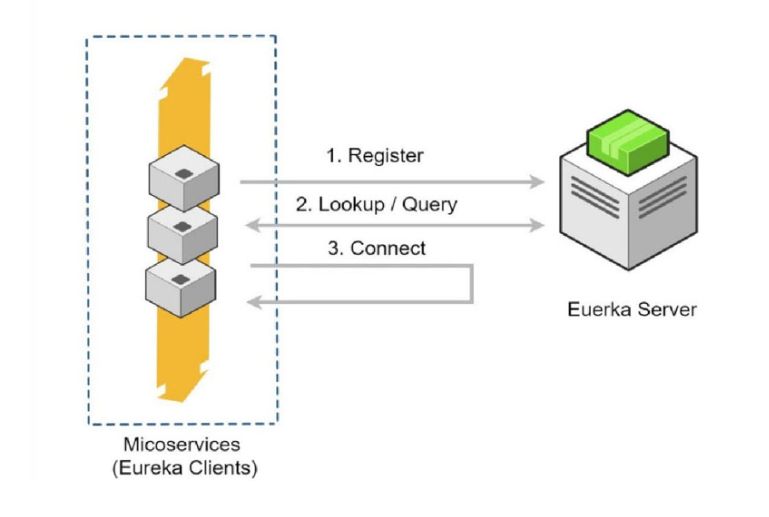
Key features of Zipkin:

* Support for multiple languages and frameworks.
* Centralized storage and visualization of tracing data.
* Ability to analyze traces and detect performance issues.
* Integration with various backends, such as Cassandra, MySQL, Elasticsearch, etc.

**Explain Docker Architecture**



**What is service discovery and registration? How is it enabled in Spring Cloud?**



**What is Spring boot?**

Spring boot provides a lot of boilerplate configurations and code that we were manually creating for application development. It abstracts the configurations, maven dependencies, application server configurations and reduces the development time. The following diagram is an overview of Spring Boot.

**What is a Dockerfile?**

Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user should call on the command line to assemble an image. Using the Docker build, users can create an automated build that executes several command line instructions in succession.

**Which are popular DevOps tools?**

The most popular DevOps tools are listed in the following table:

| **Environment** | **SDLC** | **Tool** | **Description** |
| --- | --- | --- | --- |
| **Development + Build + Test Environment** | Requirement | Jira | Plan, track, and manage agile and software development - requirements management. |
| Development | Jenkins | Orchestrator in the entire tool chain. build, test, and deploy the software. |
| Eclipse | Integrated development editor |
| Jira Service Desk | Service desk to receive, track, manage, and resolve requests. |
| Jira Insights | Manage all sorts of assets. |
| Microclimate | Tool to create Java/Spring cloud-native microservices |
| JUnit | API and endpoint testing |
| SonarQube | Static code analysis |
| PMD SCA | Whitebox security testing |
| Findfugs | Whitebox security testing |
| Swagger | Code documentation |
| Git, GitLab | Source code management |
| Build | Maven | Build automation tool |
| JFrog Artifactory | Built artefacts are stored and managed in the artifactory. Store/manage the container images. |
| Deploy | Ansible | Deployment automation |
| Release Management | Urban Code Release | Release management tool |
| Testing | Selenium | Test automation platform |
| **Staging and Prodution (Similar Tool-Set)** | Monitoring and Management | Prometheus | Application monitoring |
| Nagios | Infrastructure monitoring |
| Grafana | Dashboard and reporting |
| ELK | Application logging |
| Splunk | Security and event management |
| UCMDB | Configuration management database |
| Testing | OWASP ZAP | Blackbox security checks |

**Do you know how all sorting algorithms work?**

Bubble sort

Insertion sort

Quick sort

Merge sort

**What does the finalize method do?**

The finalize method is a protected method, inherited from Object. When the JVM is about to garbage collect the object, this finalize method is called first. The rationale for this method is to tie off any loose ends, to close off any resources that the garbage-collected object was dependent upon.

**How can you create Java objects using XSDs?**

Using JAXB library

XSD to Java compiler (xjc)

xjc -p com.wiley.acinginterview.chapter15.generated /path/to/team.xsd

**How can JSON be read in a Java application?**

Using Jackson

**Can JSON be used to serialize a Java object?**

The Jackson library can also be used to create JSON from a domain object.

@Test

public void writeJson() throws IOException {

final Player p1 = new Player();

p1.setName("Louise Mills");

p1.setPosition("Coach");

final Player p2 = new Player();

p2.setName("Liam Turner");

p2.setPosition("Attack");

final Team team = new Team();

team.setPlayers(Arrays.asList(p1, p2));

final ObjectMapper mapper = new ObjectMapper();

mapper.writeValue(new File("/tmp/newteam"), team);

}

**What is motivation behind Spring framework?**

The notion of POJOs comes as a reaction from the early approaches to **Enterprise Java Beans (EJBs), where your simple domain object needed to follow complicated patterns and implement several interfaces in order to be used within application servers**. **This code was often brittle and untestable.** In turn, this led to a rise in popularity of frameworks such as Spring, which concentrated on simple approaches around application configuration and focused on writing testable code.

**How do you test that an application context is configured correctly?**

**@ContextConfiguration**(

"classpath:com/wiley/javainterviewsexposed/chapter16/applicationContext.xml")

@RunWith(SpringJUnit4ClassRunner.class)

public class SpringIntegrationTest {

@Autowired

private SpellCheckApplication application;

@Test

public void checkWiring() {

assertNotNull(application);

}

@Test

public void useSpringIntegrationTests() {

final List<String> words = Arrays.asList(

"correct",

"valid",

"dfgluharg",

"acceptable");

final List<Integer> expectedInvalidIndices = Arrays.asList(2);

final List<Integer> actualInvalidIndices =

application.checkDocument(words);

assertEquals(expectedInvalidIndices, actualInvalidIndices);

}

}

The class is annotated to run with the SpringJUnit4ClassRunner, which understands how to load a Spring application context through the @ContextConfiguration class-level annotation.

Running tests this way allows the beans under test to be autowired for use in a particular test. Any @Autowired beans are reloaded from the application context for each test; similar to the way the @Before annotated methods are run before each test.

**How do you keep databases clean when running integration tests?**

The Spring integration testing functionality has a remarkably useful feature for working with databases and transactions. If you have a JUnit test suite, which extends AbstractTransactionalJUnit4SpringContextTests, then each of the tests run inside this suite will be run inside a database transaction, and will be rolled back at the end of each test.

This has the useful capability to check that your code meets any requirements specified by your database, such as referential integrity, or any complex knock-on effects such as those specified in a database trigger. Listing 16-18 shows a test that appears to insert data into a database. The test will fail if the precondition of any of the usernames to insert is already in the database.

Listing 16-18: Using Spring's transactional support for integration tests

@ContextConfiguration(

"classpath:com/wiley/javainterviewsexposed/" +

"chapter16/databaseTestApplicationContext.xml")

public class SpringDatabaseTransactionTest extends

AbstractTransactionalJUnit4SpringContextTests {

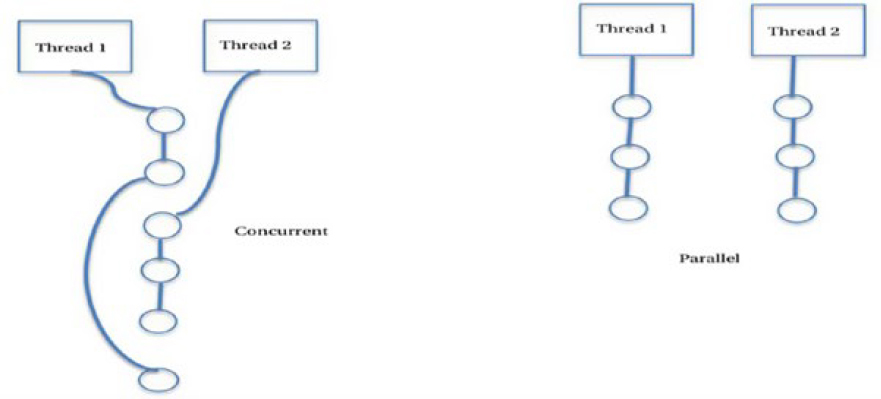
**What is difference between concurrency and parallelism?**

This brings us to the difference between concurrency and parallelism:

* *Concurrency*: The recognition that we can divide up a computation (an algorithm) into separate pieces where the *order of execution* of the pieces doesn’t matter. Concurrency is a function of the algorithm you use to solve a problem. This also means that even if you create a concurrent program, if you run it on a single processor it will still run serially.
* *Parallelism*: The mechanism used to execute a program on a particular machine or machine architecture in order to improve the performance of the program. Parallelism allows a concurrent program to execute on many different processors and thus potentially improve the overall performance of the program.

The left side of [Figure 12-1](https://cdn2.percipio.com/1690690820.36aa5552d480dd76e186ab77fae5a3437f8ab947/eod/books/137913/OEBPS/section-93.xhtml#ch12_fig1) illustrates the idea of executing a concurrent program broken up into two threads of execution and running on a single processor, whereas the right side illustrates the idea of using two processors to execute a program that’s divided up into two threads of execution.

Figure 12-1: Concurrency vs. parallelism



As an example, here’s a slightly different version of a way to tell the difference between concurrency and parallelism that appeared in Stack Overflow[[1]](https://cdn2.percipio.com/1690690820.36aa5552d480dd76e186ab77fae5a3437f8ab947/eod/books/137913/OEBPS/section-99.xhtml#ftn.ch012_fn1):

Assume that your local chess club organizes a demonstration tournament where 10 local players, all with approximately the same skill level, will be pitted against a chess Grandmaster. So the club organizers have to arrange 10 chess games and want to make them time efficient so everyone can go out and celebrate. There are several ways to organize the match:

* *Serial*: The boards are all lined up in a room and the chess master sits down at the first table to play. The game is played to completion and the chess master then moves to the next table to play the next game. If each game lasts 10 minutes, that’s a total of 100 minutes. If it takes the chess master about 6 seconds to move from one table to the next, then that’s about another minute (well, it’s 54 seconds, but we’ll fudge it a bit) for a total of 101 minutes. Not bad.
* *Concurrent*: The boards and players are all lined up the same way, but for this version the chess master plays differently. The chess master sits down at the first table to play, makes the first move in 6 seconds and then immediately moves to the next player to make the next first move. Once again, it takes 6 seconds to move between tables. This continues until after one minute (6 \* 10 seconds) the chess master is back at the first table for the second move. If we assume that each game will take the same 10 minutes that the serial games took, then we need to know how many rounds the chess master will have to make. If each of the local players takes about 50 seconds for their move, then we have each move taking 50 + 6 = 56 seconds. Each game takes 10 minutes or 600 seconds, so we have 600 seconds / 56 seconds per move = about 11 rounds to complete all 10 games. This will give us a total of 11 \* 56 + 11 \* 60 = 616 + 660 seconds = 1,276 seconds = 21.27 minutes to complete the match. Much better.
* *Parallel*: Lets say the chess club wants to add a bit more competition to the event and hires a second chess master. With two chess masters, the club organizers can have each master play just 5 games. If they use the serial method from above, then each chess master will sit down, play a complete game and then move to the next table. But this time since two games are being played simultaneously, the tournament is finished in half the time, 101 / 2 = 50.5 minutes. Better, but not as good as the concurrent approach.
* *Concurrent* and *Parallel*. In this case we have two chess masters, and each of them plays all 5 games at the same time. Each chess master sits down, makes a move in 6 seconds, gets up and moves to the next table also in 6 seconds. If the games still take 10 minutes each, and if each player still takes 50 seconds for each move, then we still end up with 11 rounds, but the total time to move between games only takes 30 seconds now (because there are only 5 games going on for each chess master). That gives us 11 \* 56 + 11 \* 30 = 616 + 330 = 946 seconds = 15.77 minutes for the entire match. This appears to be the minimum value that we get from the four different ways of playing the match, and it seems that creating a concurrent program and running it on a parallel machine is clearly better than the other possibilities.

Remember, concurrency is a property of the algorithm, whereas parallelism is a property of the machine.

**What are list of parallel pattern?**

Embarrassingly Parallel

Master/Worker

Map and Reduce

MapReduce

Divide & Conquer

Fork/Join

**What is order of execution of blocks in a terraform script?**

All the code for Terraform scripts can be authored in a single name having the extension ".tf". Terraform will load, parse, and interpret the file reading the blocks and then executing them logically. The order of execution of blocks in Terraform script is:

* terraform
* provider
* variable
* locals
* resource
* module
* output

**What is module in Terraform script?**

Modules can be consumed in Terraform scripts by making them part of the configuration. Just as resources are declared and used within Terraform, Terraform provides the module block along with its identifier. The only mandatory attribute for a module block is the source attribute. The source attribute defines the location of the module from which it should download its definition and provision resources. Additionally, and optionally, the module block accepts the values for the variables needed for the working of the module.

Modules are plugins that are downloaded and installed in the local .terraform folder managed by Terraform at the time of executing the terraform init command. The terraform init command initializes the modules after downloading it, and then it configures the local or remote backend alongside the plugins (providers) used within the configuration.

The references to the module for a Terraform deployment are stored within the modules folder of the .terraform directory. The is Terraform’s internal directory and should not be edited. Terraform uses the modules.json file to store the references and uses it to identify the modules, their sources, and the name of the module. See [Figure 5-5](https://cdn2.percipio.com/1690815426.3d3fabed4968947b1afc873ec83afe69fc4081a9/eod/books/157950/OEBPS/section-58.xhtml#ch05Fig5).

**What is terraform state file?**

resource "azurerm\_sql\_server" "book\_sql\_server" {

name = var.sql\_server\_name

location = azurerm\_resource\_group.common\_resource\_group.location

resource\_group\_name = azurerm\_resource\_group.common\_resource\_group.name

version = "12.0"

administrator\_login = var.sql\_username\_secret.value

administrator\_login\_password = var.sql\_password\_secret.value

tags = var.all\_tags

}

The resultant state file resource section after successful execution of the Terraform script containing this resource declaration is shown here:

{

"mode": "managed",

"type": "azurerm\_sql\_server",

"name": "book\_sql\_server",

"provider": "provider[\"registry.terraform.io/hashicorp/azurerm\"]",

"instances": [

{

"schema\_version": 0,

"attributes": {

**"administrator\_login": "SqlServeradmin",**

**"administrator\_login\_password": "SqlServer@1234 ",**

"connection\_policy": "Default",

"extended\_auditing\_policy": [],

"fully\_qualified\_domain\_name": "booktfsql1104.database.windows.net",

"id": "/subscriptions/Xxx/resourceGroups/bookappresourcegroup/

providers/Microsoft.Sql/servers/booktfsql1104",

"identity": [],

"location": "westeurope",

"name": "booktfsql1104",

"resource\_group\_name": "bookappresourcegroup",

"tags": {

"chapter": "terraform security",

"owner": "ritesh"

},

"timeouts": null,

"version": "12.0"

},

"sensitive\_attributes": [],

"private": "eyJlMmJmYjczMC1lY2FhLTExZTYtOGY4OC0zNDM2M2JjN2M0YzAi

OnsiY3JlYXRlIjozNjAwMDAwMDAwMDAwLCJkZWxldGUiOjM2MDAwMDAwMDAwMDAs

InJlYWQiOjMwMDAwMDAwMDAwMCwidXBkYXRlIjozNjAwMDAwMDAwMDAwfX0=",

}

]

}

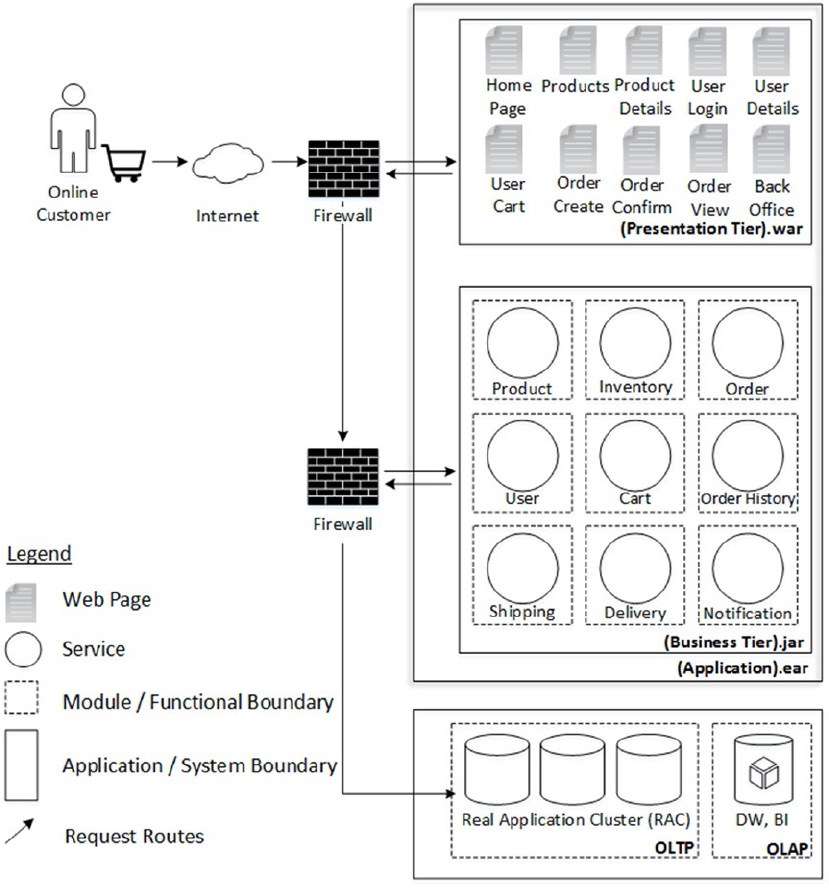
**How terraform state is being managed in your project?**

**What is terraform ?**

Basically, an infrastructure deployment tools. Also, termed as IaC tool.

**Explain me Typical application architecture?**

Figure 1-3: Typical application architecture



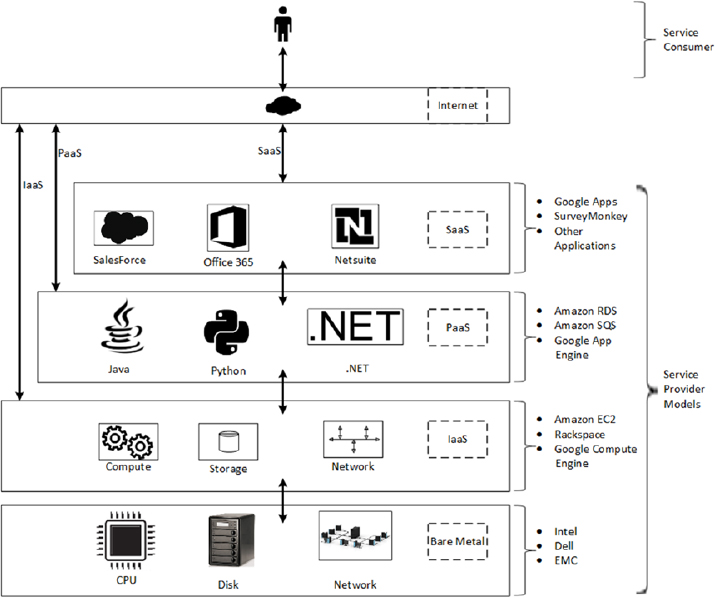
**Explain me Typical Deployment Architecture.**

Typical deployment architecture



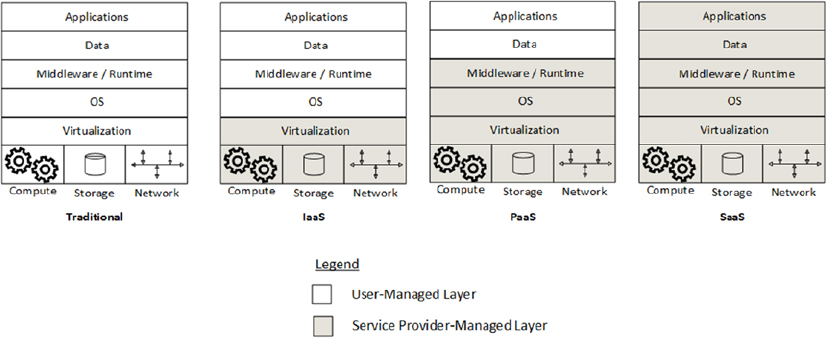
**What are typical cloud architecture components?**

 Cloud architecture components



**What is cloud service model?**

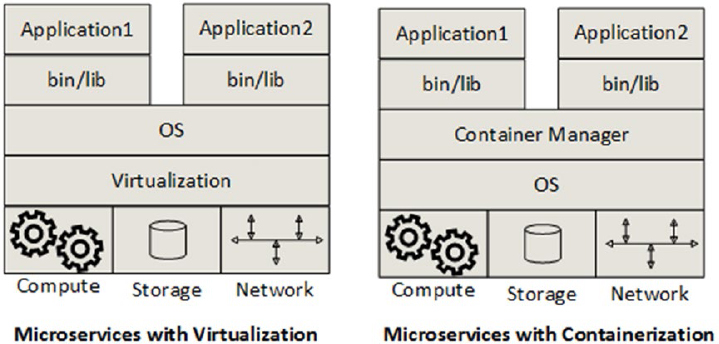
The cloud service model is based on the kind of control users may have over how the resources are being utilized.



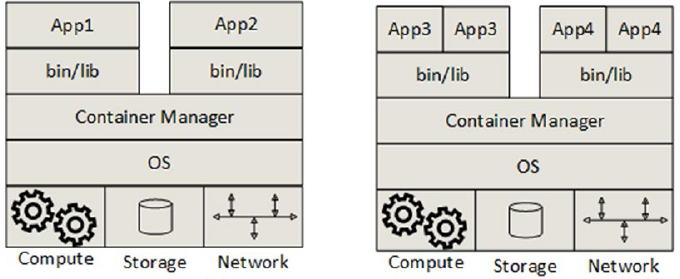
In the traditional service model, the user or the enterprise itself is responsible for managing the whole stack (e.g., hardware, data center facilities, software, and data) whereas in the cloud service model it varies as follows:

* **Infrastructure as a Service**: In the IaaS service model, the users request compute power, storage, network, and their associated resources alone and pay for what they use.
* **Platform as a Service**: In PaaS model, the users have zero control over the underlying infrastructure, such as CPU, network, and storage, as they are abstracted away below the platform. Instead, it allows application development platforms that allow the creation of applications with supported programming languages and related tools hosted in the cloud and accessed through an interface, mainly a browser. Many times the application runtime and middleware are also provided by the CSP, and the user develops, installs, manages, and operates the software application alone and its data.
* **Software as a Service**: This is the "forget everything" model where the user doesn't even own, manage, or operate the application. Applications are run on the cloud infrastructure and are accessible from various client devices. Limited user-configurations are available. Sometimes the same application instance serves the end users of more than one enterprise (tenant); such applications are called multi-tenant.

**What is difference between virtualized servers and containers?**



**Container instances sharing resources?**

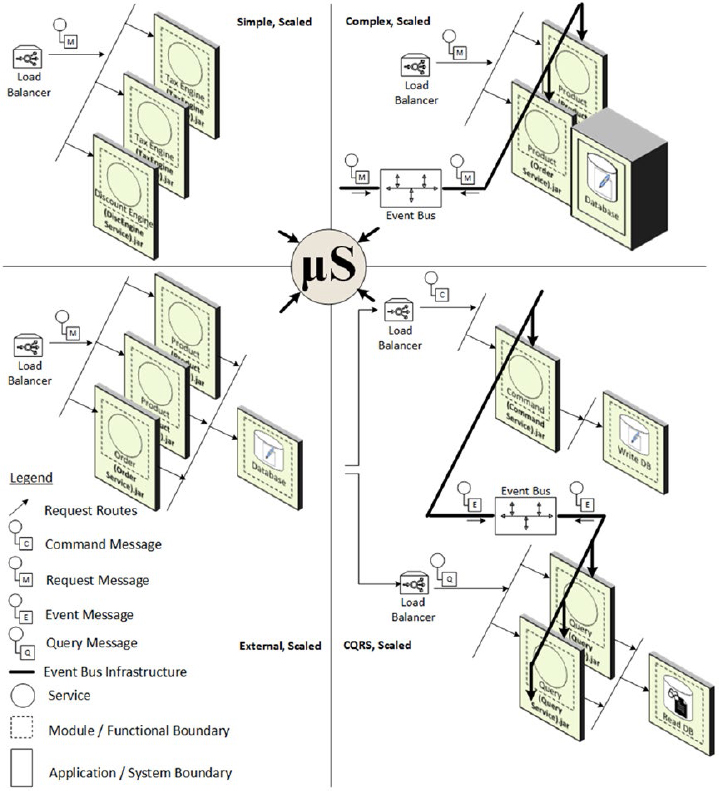


**What is Docker?**

Docker, which is now based on the Linux libcontainer, is a management system used to create, manage, and monitor containers in the Linux platform.

Docker is the most popular container provider system. Since it will run only on a Linux–based OS, if you have to set up Docker in Windows or Mac machines, you first need to start up a Linux VM using VirtualBox and then run the Docker containers inside this virtual machine.

**What is microservices inner architecture examples?**



**What is Microservices outer architecture?**

A diagram of a service

Description automatically generated

**What is difference between Traditional vs. CQRS-Based Software Systems?**

The Nomenclature in CQRS

CQRS is based on the notion of two concepts, commands and events. Commands and events are explained as follows:

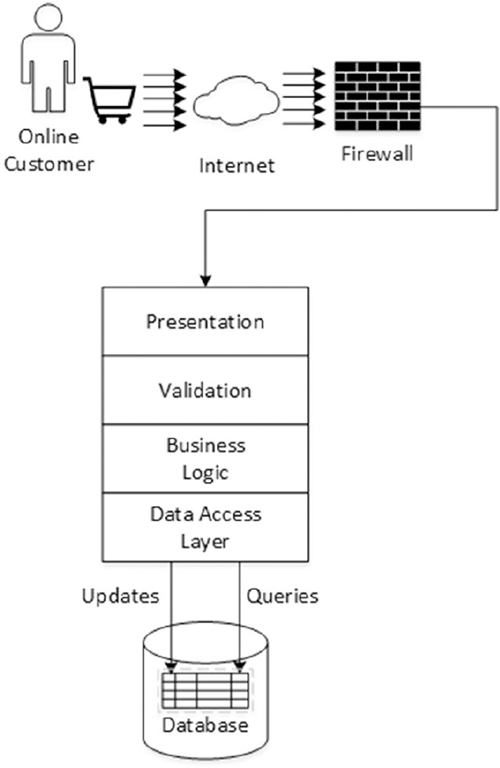
**Commands**: The intent to change the state of an entity is modelled as a command.

**Events**: Once there is a change in the state of an entity, events represent what has changed.

Referring to Figure 5-3, a write transaction coming from the client to the presentation tier is represented by commands, which will encapsulate all the information required to effect a state change. Any such state change will result in a corresponding write to the data persistent store. Due to this action or effect of state change, there can be components or services elsewhere in the application that are interested in what has changed and there should be a mechanism to propagate these changes to those interested services. Events come handy in this case. So, if a change has happened to the state of a write data entity, it can be propagated to the read data entity counterparts in terms of events.

Traditional

Same schema for read and write transactions



**Separate schemas for read and write transactions**

A diagram of a diagram

Description automatically generated

**Separate database nodes for read and write transactions**

Separate database nodes for read and write transactions

A diagram of a data processing process

Description automatically generated

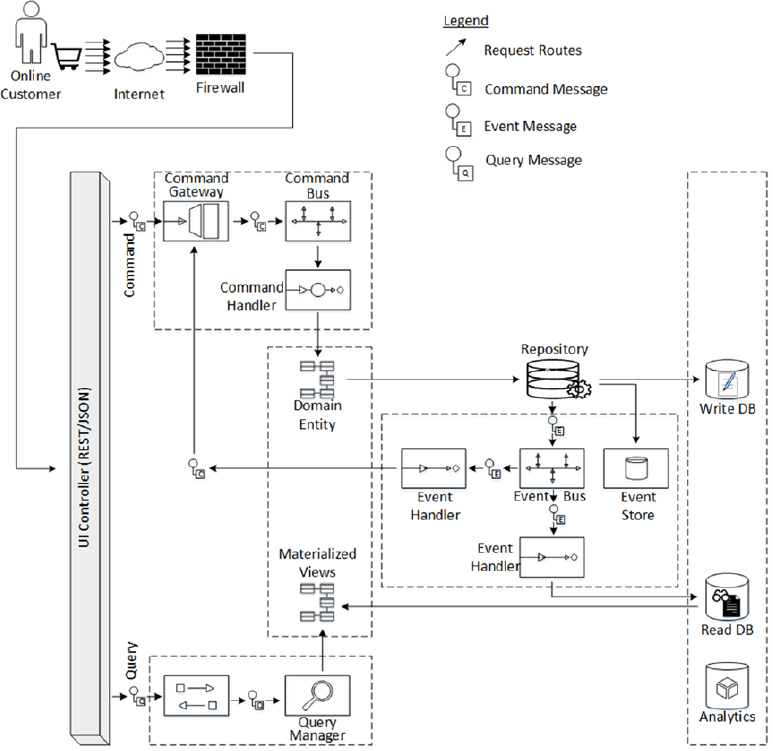
**Scale out using separate read and write processes**

A diagram of a data flow

Description automatically generated

**Explain me a typical Meta Model for Event-Based CQRS Design?**

A meta model for CQRS-based architecture



A meta model for event-based CQRS architecture

A diagram of a computer system

Description automatically generated

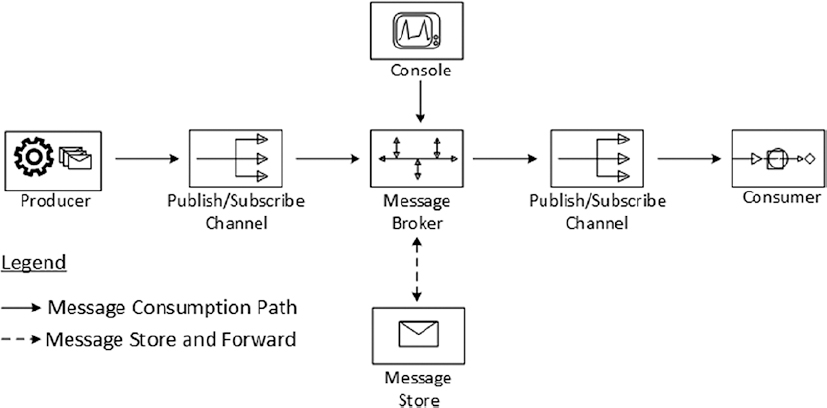
**Can you explain the importance of messaging in distributed software architectures, especially in the context of microservices?**

One of the most powerful mechanisms used in building enterprise grade software applications is messaging. Had we not invented messaging, all of the software applications around the globe would be like a complete workforce, needing to be working 24 x 7, 365 days a year without any room for downtime or failure.

But in practice we know that every software application is prone downtime or failure; rather, every software application should be designed to have downtime or failure. Since enterprise applications don't live in isolation, instead they talk to each other in a distributed manner, what an application tried to talk to an application that is down?

Messaging helps to gracefully accommodate unexpected downtime and failure of coordinating applications

**Explain how will you achieve Messaging for Resiliency in microservices architecture?**



Message persistence to balance operating characteristics

A diagram of a diagram

Description automatically generated

Clustering message brokers for resiliency

A diagram of a message exchange

Description automatically generated

**What is Spring Cloud?**

 An indispensable piece of Spring called Spring Cloud, which is built over Spring Boot itself.

There are a bunch of common patterns in distributed, microservices ecosystems that can help you integrate the core services as a loosely coupled continuum, and Spring Cloud provides many powerful tools that enhance the behaviour of Spring Boot applications to implement those patterns.

**What are different components in Spring Cloud?**

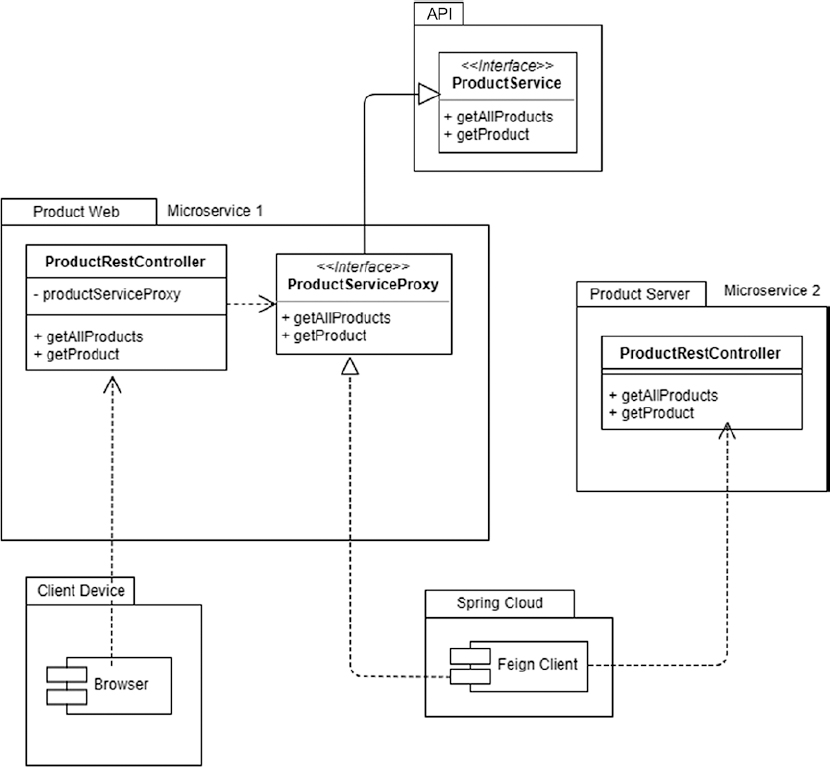
* The Feign client to make HTTP API calls
* Hystrix, the circuit breakers to gracefully degrade
* The Hystrix dashboard, showing a graphical overview of a circuit breaker in the system
* Ribbon, the client-side load balancer
* Eureka, a REST-based registry service
* Zuul, the API gateway, which is the front controller for your microservices
* The Config Server to externalize version control and manage configuration parameters for microservices

**What are typical use cases solved by Spring cloud?**

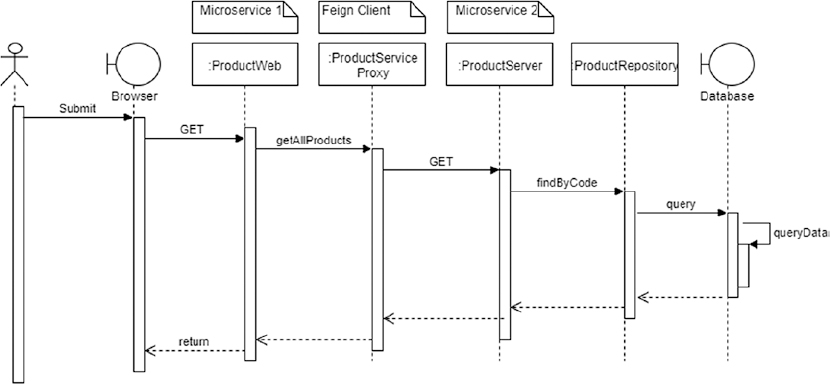
* Distributed/versioned configuration
* Service registration and discovery
* Routing and load balancing
* Interservice calls
* Circuit breakers
* Distributed messaging

**Explain a typical use case of Feign Client in Spring cloud?**

A Feign client usage scenario



Delegation of calls through the Feign client



**"Feign is a declarative HTTP client" . What does declarative mean here?**

In the context of Feign being a "declarative HTTP client," the term "declarative" refers to a style of programming where you specify what you want to achieve, rather than explicitly defining how to achieve it. In other words, you focus on describing the desired outcome or behavior, and the underlying implementation details are abstracted away.

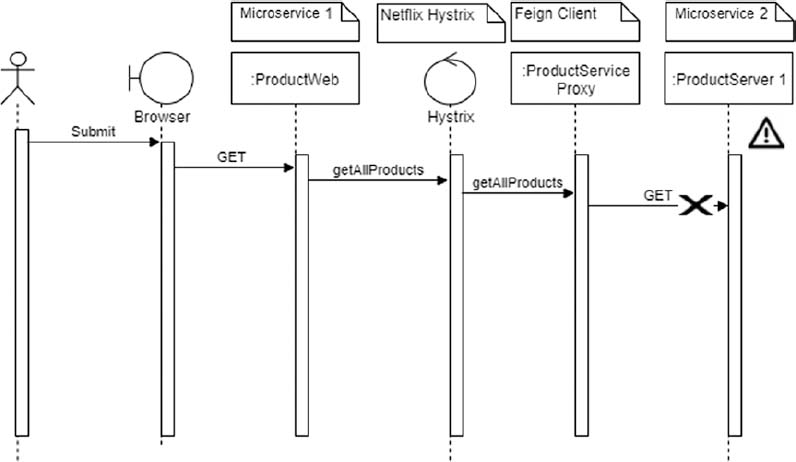
When using Feign, you define your API requests and interactions with remote HTTP services using a simple and concise interface, typically through annotations or configuration files. You describe the API endpoints, request parameters, headers, and other relevant details. The library then handles the process of making the actual HTTP requests, handling responses, and mapping the results to appropriate data structures.

The contrast to declarative programming is imperative programming, where you explicitly specify the step-by-step instructions on how to perform a task or achieve a goal. In declarative programming, you don't worry about the low-level details of the HTTP communication; instead, you focus on the high-level intent of the requests and responses.

This declarative approach can make the code cleaner, more readable, and less prone to errors. Feign, being a declarative HTTP client, allows developers to work at a higher level of abstraction when interacting with HTTP APIs.

**Can you explain how Spring cloud hystrix works?**

 Hystrix detects service non-availability



Hystrix retries the alternate service

A diagram of a product

Description automatically generated

**Explain typical syntax for Hystrix on method level.**

@HystrixCommand(fallbackMethod = "getAllTheProducts")

public ResponseEntity<Resources<Resource<Product>>> getAllProducts() {

return productServiceProxy.getAllProducts();

}

@HystrixCommand(fallbackMethod = "getTheProduct")

public ResponseEntity<Resource<Product>> getProduct(

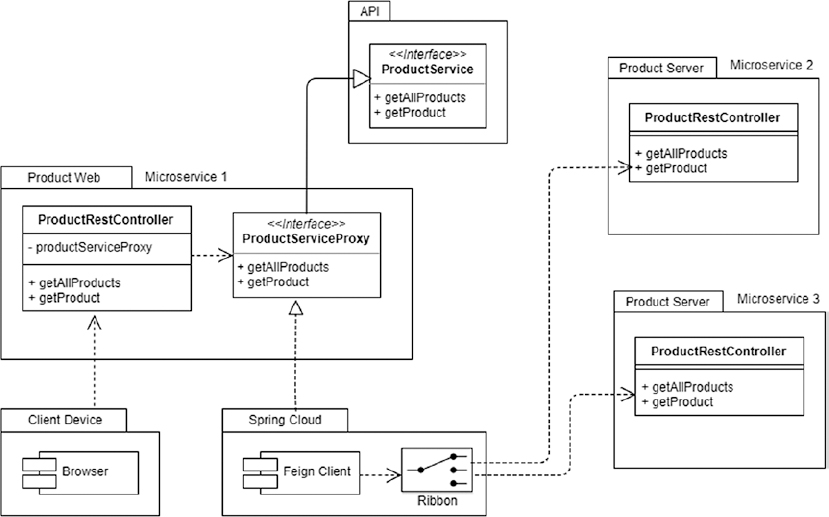
@PathVariable("productId") String productId) {

return productServiceProxy.getProduct(productId);

}

**Can you explain client side load balancer Ribbon how it works?**

Design with Ribbon



You now need to provide configuration values to Ribbon, which you will do in ch08\ch08-04\ProductWeb\src\main\resources\application.properties:

product-proxy.ribbon.listOfServers=localhost:8080,localhost:8081

**How do you configure Zuul API gateway in Spring cloud?**

you need Spring Cloud's @EnableZuulProxy to stand up an API gateway that other microservices can bind to. This is done in your regular Spring Boot application main class with one annotation added to enable the Zuul gateway.

@EnableZuulProxy

@EnableDiscoveryClient

@SpringBootApplication

public class ProductServerApiApplication {

public static void main(String[] args) {

SpringApplication.run(ProductServerApiApplication.class, args);

}

}

You now need to define the routes at the API gateway, which you will do in application.yml

spring:

application:

name: product-service-api

server:

port: 8082

zuul:

routes:

product-api:

path: /api/\*\*

service-id: product-service

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

hystrix:

command:

default:

execution:

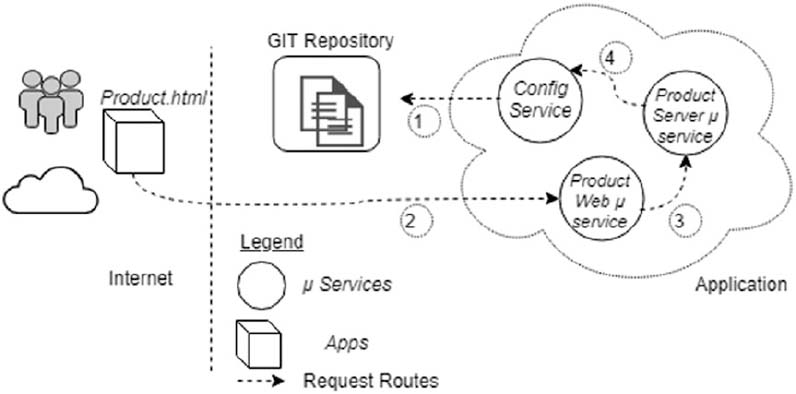
isolation:

thread:

timeoutInMilliseconds: 2000

**Explain Spring cloud config server?**

Design for a Config Server scenario



<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

@SpringBootApplication

**@EnableConfigServer**

public class EcomProductConfigApplication {

public static void main(String[] args) {

SpringApplication.run(EcomProductConfigApplication.class, args);

}

}

Next, configure the Config Server itself to point to the GIT URL.

server:

port: 8888

spring:

cloud:

config:

server:

git:

uri: file://D:/binil/gold/pack03/ch08/ch08-07/ConfigServer/

config-repo (Basically, this is GIT URL)

Here, since Port 8888 is the default port for the Config Server, even without explicit mention of server.port, the Config Server will bind to 8888. Next, you provide the GIT URL. It's in this repository where you need to place all your configuration files for the different microservices.

Next, you need to enable your microservice to access the Config Server

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

The Product Server microservice will act as the client for the Config Server. This is enabled by the spring-cloud-starter-config. You need to note the actuator dependency too, which is required for refreshing the configuration parameters when there is a change to the parameter in the Config Server

server:

port: 8081

spring:

application:

name: productservice

cloud:

config:

server:

uri: http://localhost:8888

management:

security:

enabled: false

The significant configuration parameters are

spring.application.name=productservice

spring.cloud.config.uri=http://localhost:8888

Here productservice is a logical name given to the Product Server microservice, which will also be treated as its service ID. The Config Server will look for productservice.yml in the repository to resolve any application configuration parameters.

**What is @RefreshScope annotation is Spring cloud config?**

When a bean is annotated with @RefreshScope, it means that Spring will monitor it for changes and refresh the bean's state if there are any changes to the configuration properties. This allows you to update the configuration without stopping the application and apply the changes dynamically.

Here's a brief overview of how it works:

1. You configure the Spring Cloud Config Server or a similar configuration management tool to provide the dynamic configuration properties.
2. Your Spring Boot application connects to the Config Server during startup and fetches the initial configuration.
3. Beans annotated with @RefreshScope will be created with the initial configuration.
4. If you change the configuration properties in the external configuration source (e.g., the Config Server), you can trigger a refresh by making a POST request to the /actuator/refresh endpoint of your Spring Boot application. This will cause the application context to re-evaluate the beans with @RefreshScope, and if there are changes in the configuration properties, the beans will be re-initialized with the updated values.

**Can you explain what is high availability in microservices architecture?**

High availability planning involves the identification of services that must be available for business continuity. Components that comprise each service should be identified, and list of possible points of failure for these systems should be made. A failure tolerance baseline should be established for each of them and failover strategies should be designed.

### Measuring High Availability

Software availability is expressed as a percentage of yearly uptime. Using the nearly unachievable ideal of 100% availability as a baseline, the goal of the highest levels of service availability is considered to be "five nines," or 99.999% availability.

| **Levels of Availability** | | | |
| --- | --- | --- | --- |
| **Availability** | **Unavailability** | **Five Nines** | **Downtime per Year** |
| 0.9 | 0.1 | 1 | 36 Days |
| 0.99 | 0.01 | 2 | 87.7 Hours |
| 0.999 | 0.001 | 3 | 8 Hours, 46 Minutes |
| 0.9999 | 0.0001 | 4 | 52.5 Minutes |
| 0.99999 | 0.00001 | 5 | 5 Minutes 16 Seconds |
| 0.999999 | 0.000001 | 6 | 32 Seconds |

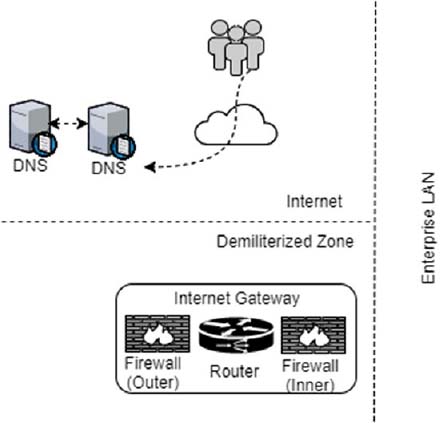
**What is Domain Name System?**

DNS serves as the phone book for the Internet by translating human-friendly computer hostnames into IP addresses, it serves as the bootstrap mechanism for most of the software-related operations through the Internet.

DNS outages may happen due to configuration errors, infrastructure failure, or a DDoS attack.

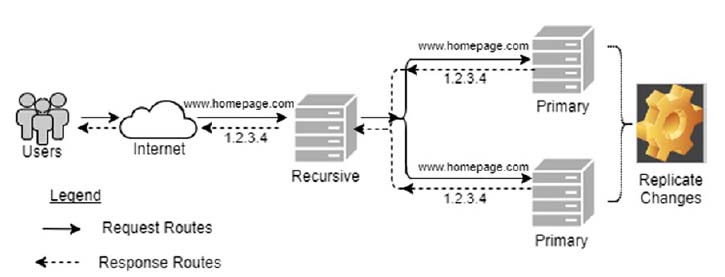
If the DNS name servers are overloaded with more requests than they can serve, or they are down or responding sluggishly due to any of the above reasons, this creates latency in the DNS network, making the resolution process sluggish and eventually the DNS server will be rendered unavailable for most users.

DNS redundancy



When evaluating multiple DNS providers for your multi-DNS network, consider the global point of presence (PoP) of the providers and make sure these providers on separate networks can be seamlessly integrated using APIs.

How DNS redundancy works

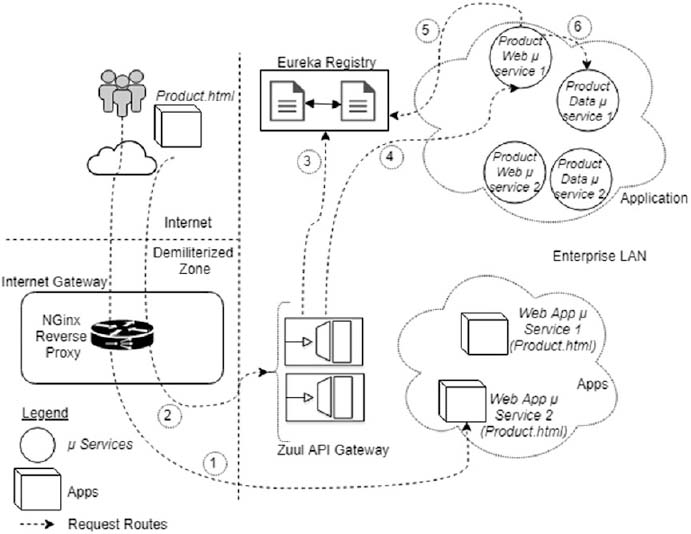


### What is SP Redundancy?

### What is Application Architecture Redundancy?

### What is Data and Storage Redundancy?

### Can you explain a typical design of a Highly Available Microservice Scenario?

Demonstration purpose of the microservices HA scenario, we can utilize the Eureka and Zuul components   


**What is Use of Nginx reverseproxy?**

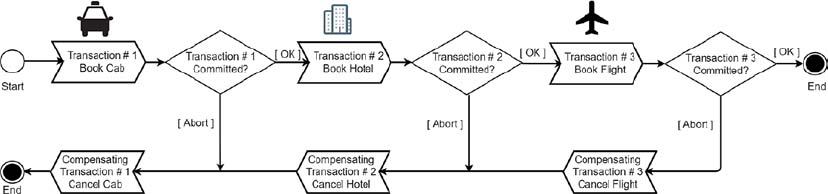
**Can you explain Saga design pattern?**

In sagas, each of these transactions has a corresponding compensating transaction. If any of the transactions in a saga fails, the compensating transactions for each transaction that was successfully run previously will be invoked so as to nullify the effect of previously successful transactions.

When two-phase commit transactions can't be used and when you still want to exercise control at each stage of a multi-step transaction, sagas will help you.

A classical example is when you book a flight, hotel, and rental cab. It's very unlikely that a single enterprise owns inventory for all these resources, but it's highly likely that end user wants to book one or more of these resources in a single transaction, because a confirmed hotel booking with a non-confirmed flight booking is not very useful for him. Another aspect here is that, if you look at the resource managers or the resource owners in such cases, they may fall outside trust boundaries, and in many cases they might be competing enterprises. My current firm is involved in building software systems for the travel and hospitality industry, and I can relate to many instances in our software design where similar transactions can and will only be designed to be eventually consistent.

State diagram illustrating saga



**Can you explain distributed saga?**

### Distributed Saga

The concept of saga as described in the previous section is rather trivial to build, since many of the workflow and Business Process Model and Notation (BPMN) frameworks are built using the same analogy where you want to orchestrate long-running transactions. If you want to coordinate the same in a microservices environment, this involves coordinating multiple decentralized nodes and processes, which is non-trivial.

You will first look at the various components needed for a distributed saga at the conceptual level. See [Figure 15-2](https://cdn2.percipio.com/1691083994.d7b40444ba15cb4d3a42f9d3b6666ab3c499a822/eod/books/146798/OEBPS/section-89.xhtml#ch15_fig2).

* Transactions and compensation transactions:

A saga transaction in a microservices context is better called a request (HTTP) or an event (commands and events); however, for the simplicity of our discussion, we will retain the term transactions.

A distributed saga is a collection of transactions:

T1, T2, ..., Tn

Each transaction has a compensating transaction:

C1, C2, ..., Cn

A compensating transaction (Cn) will semantically undo a previously completed corresponding transaction (Tn).

A distributed saga guarantees that either

T1, T2, ..., Tn

or

T1, T2, ..., Ti, Ci, ..., C2, C1

happens so that, as shown in [Figure 15-1](https://cdn2.percipio.com/1691083994.d7b40444ba15cb4d3a42f9d3b6666ab3c499a822/eod/books/146798/OEBPS/section-89.xhtml#ch15_fig1), if one of the transactions fails, the compensating transactions for each of the transactions that successfully ran will be executed.

* The saga log:

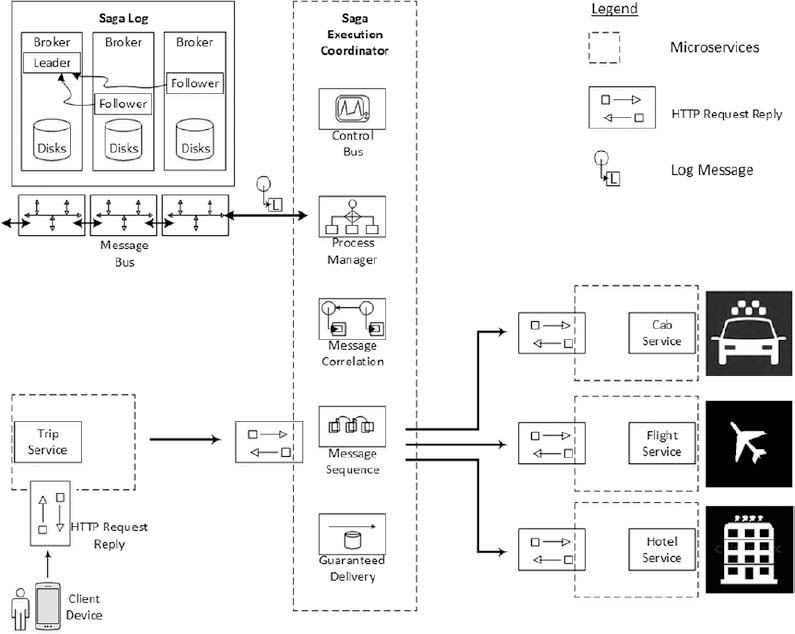
The saga log is a distributed log that's used to persist the state of every transaction/operation during the execution of a given saga. The saga log contains various state-changing operations, such as begin saga, end saga, abort saga, begin Ti, end Ti, begin Ci, and end Ci. The saga log is often implemented using a distributed log, and systems such as Kafka are commonly used for the implementation.

* The saga execution coordinator (SEC):

The SEC orchestrates the entire logic and is responsible for the execution of the saga. All of the steps in a given saga are recorded in the saga log and the SEC writes to and reads from and interprets the records of the saga log. A SEC has the following subcomponents:

* + Control bus: A control bus effectively administers a messaging system that is distributed across multiple nodes and processes.
  + Process manager: A process manager maintains the state of the sequence and determine the next processing step based on intermediate results.
  + Message correlation: The sender correlates the response it received to the request it sent originally. The MessageID and CorrelationID properties of the message are used to correlate the request and response messages. Thus a correlation identifier is a unique identifier that indicates which request message this reply is for.
  + Message sequence: This is required to retain the order of the messages send as per original intention.
  + Guaranteed delivery: With guaranteed delivery, the system uses built-in data stores to persist messages. Each node the messaging system is installed on has its own data store so that the messages can be stored locally. When the sender sends a message, the send operation does not complete successfully until the message is safely stored in the sender's data store. Subsequently, the message is not deleted from one data store until it is successfully forwarded to and stored in the next data store. In this way, once the sender successfully sends the message, it is always stored on disk on at least one computer until it is successfully delivered to and acknowledged by the receiver.

Distributed saga



The SEC is not a special process that has central control of the entire execution. It operates as a centralized runtime, but the runtime is dumb and the execution logic is kept out of the SEC in the distributed saga log. So when an SEC crashes, another SEC instance could be brought live and continue by looking at the saga log. It is required to make sure that SEC is up and running all the time. In the event of an SEC failure, a new SEC process should be started based on the same distributed saga log. The distributed saga log is also distributed and replicated so that there is no single point of data loss.