1. Oauth

**Key Roles in OAuth 2.0**

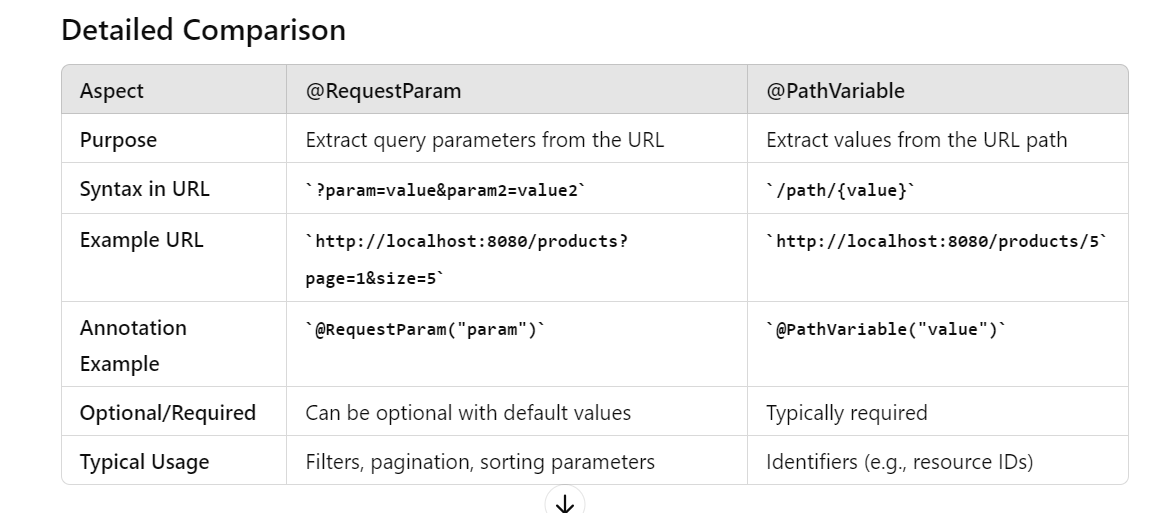
1. **Resource Owner**: The user who authorizes an application to access their account.
2. **Client**: The application that wants to access the user's account.
3. **Resource Server**: The server hosting the protected resources.
4. **Authorization Server**: The server that authenticates the user and issues access tokens.

2 step in jwt

### Steps in JWT Authentication

1. **Login Request**: The client sends a login request to the authentication server with the user's credentials (username and password).
2. **JWT Issued**: The authentication server verifies the credentials. If they are valid, it creates a JWT and sends it back to the client.
3. **Store JWT**: The client stores the JWT, typically in local storage or a cookie.
4. **Include JWT in Requests**: For each subsequent request to a protected resource, the client includes the JWT in the Authorization header using the Bearer schema.
5. **JWT Validation**: The resource server validates the JWT. If the token is valid, the server processes the request and sends the appropriate response back to the client.

3 difference between request param and path variable



**3 sqs vs kafka**





4- springbootapplication

**@SpringBootApplication Annotation**

The @SpringBootApplication annotation is a meta-annotation that includes the following three annotations:

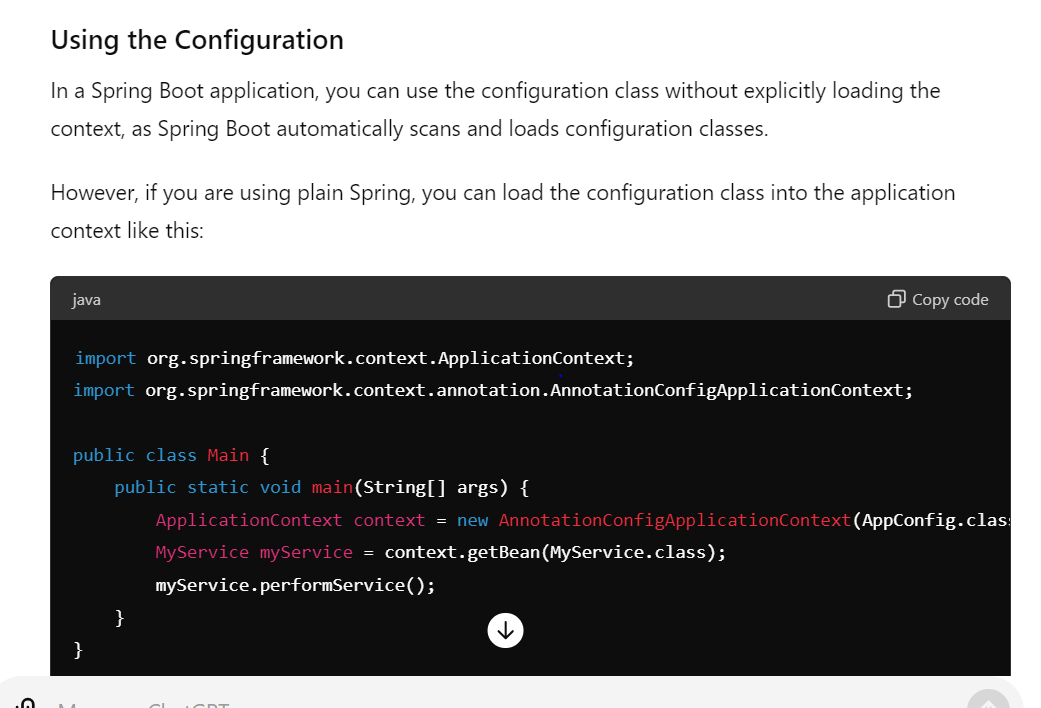
1. **@SpringBootConfiguration**:
   * This annotation is a specialization of @Configuration.
   * It indicates that the class is a source of bean definitions for the application context.
2. **@EnableAutoConfiguration**:
   * This annotation tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.
   * It automatically configures your Spring application based on the dependencies that you have added.
   * For example, if spring-boot-starter-web is on the classpath, this annotation flags the application as a web application and activates key behaviors such as setting up a DispatcherServlet.
3. **@ComponentScan**:
   * This annotation tells Spring to scan the current package and all its sub-packages for Spring components (i.e., @Component, @Service, @Repository, @Controller, etc.).
   * This ensures that all your Spring components are automatically discovered and registered as beans in the Spring application context.

5- @configuration

The @Configuration annotation in Spring is used to define configuration classes that serve as sources of bean definitions. It simplifies the configuration by allowing the use of Java code instead of XML. Methods within a @Configuration class annotated with @Bean create and configure beans that are managed by the Spring container.

**Explanation**

1. **@Configuration**: Marks the class as a source of bean definitions.
2. **@Bean**: Indicates that a method produces a bean to be managed by the Spring container. Each @Bean method is called, and its return value is registered as a bean within the Spring application context.
3. **Application Context**: The AnnotationConfigApplicationContext is used to load the AppConfig class, which contains the bean definitions.



6- application context

The ApplicationContext in Spring is a central interface for providing configuration information to an application. It manages the lifecycle of beans, handles their dependencies, supports internationalization, event propagation, resource loading, and environment abstraction. Different types of ApplicationContexts (such as AnnotationConfigApplicationContext, ClassPathXmlApplicationContext) cater to different configuration needs, making it a versatile and powerful component of the Spring Framework.

7- dispatcher servlet

The DispatcherServlet in Spring MVC is a key component that acts as the front controller for web applications. It intercepts incoming HTTP requests, delegates them to appropriate handlers, processes the requests, and generates responses. It integrates with various components of the Spring Framework to provide a comprehensive and flexible approach to web application development. In Spring Boot, the DispatcherServlet is automatically configured, simplifying the setup process.

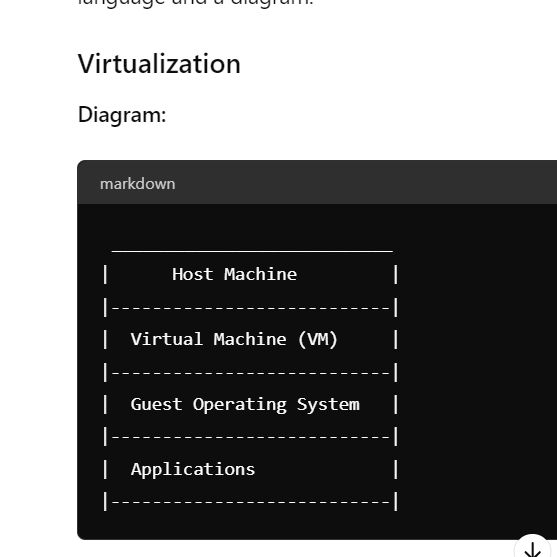
**Key Responsibilities of DispatcherServlet**

1. **Request Handling**: Intercepts incoming HTTP requests and maps them to appropriate handler methods in controller classes.
2. **Handler Mapping**: Uses handler mappings to determine which controller method should process the request.
3. **Request Processing**: Dispatches requests to controllers, invokes business logic, and generates responses.
4. **View Resolution**: Resolves logical view names returned by controller methods to actual view implementations (e.g., JSPs, Thymeleaf templates).
5. **Exception Handling**: Delegates exception handling to appropriate handlers.
6. **Interceptors**: Supports the use of interceptors for pre-processing and post-processing requests.

**Lifecycle of a Request in Spring MVC**

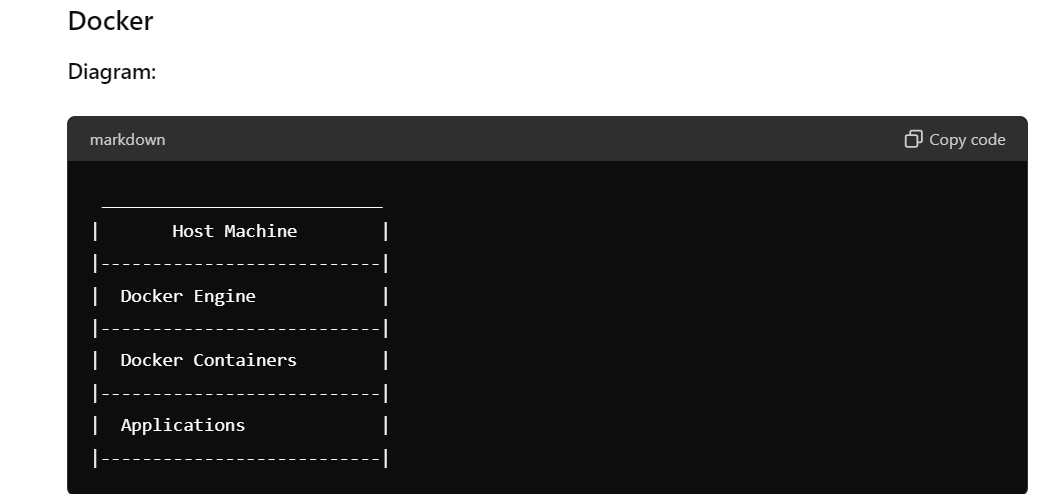
1. **Client Request**: A client sends an HTTP request to the web application.
2. **Front Controller**: The request is intercepted by the DispatcherServlet.
3. **Handler Mapping**: The DispatcherServlet consults the handler mappings to find the appropriate controller method to handle the request.
4. **Controller**: The controller processes the request, often interacting with the service layer to perform business logic.
5. **Model and View**: The controller returns a ModelAndView object, which contains the model data and the view name.
6. **View Resolution**: The DispatcherServlet resolves the view name to a specific view implementation.
7. **View Rendering**: The view is rendered with the model data and the response is sent back to the client.

**8- docker vs virtualization**

****

**Explanation:**

* In virtualization, the host machine runs a hypervisor (like VMware or VirtualBox), which allows it to host multiple virtual machines (VMs).
* Each VM has its own complete operating system (OS) installed (called the guest OS), which could be different from the host OS.
* Applications run inside these VMs as if they are running on separate physical machines.
* Virtualization provides isolation and flexibility but can be resource-intensive due to the overhead of running multiple OS instances.



**Explanation:**

* Docker uses containers instead of virtual machines.
* The host machine runs the Docker Engine, which manages and orchestrates containers.
* Containers share the host machine's OS kernel but are isolated processes.
* Each container runs a lightweight runtime environment with the necessary libraries and dependencies to run applications.
* Applications run inside these containers, and each container behaves like a small, isolated system.
* Docker containers are lightweight and faster to start compared to VMs, as they don't require booting an entire OS.

9:- What are the different types of references in Java?

In Java, there are four types of references, each with its own characteristics regarding how and when they are eligible for garbage collection:

**1. Strong References**

* **Default Reference Type**: When you create an object using new keyword, it creates a strong reference to the object.
* **Eligibility for Garbage Collection**: An object referenced by a strong reference is not eligible for garbage collection as long as the reference exists.
* **Example**:

java

Copy code

MyClass obj = new MyClass();

**2. Weak References**

* **Description**: Weak references allow objects to be garbage collected when they are only weakly reachable, meaning there are no strong references pointing to them.
* **Use Case**: Useful for implementing caches where you want to keep objects as long as they are actively used, but allow them to be garbage collected when memory is low.
* **Example**:

java

Copy code

WeakReference<MyClass> weakRef = new WeakReference<>(new MyClass());

MyClass obj = weakRef.get();

**3. Soft References**

* **Description**: Soft references allow objects to be garbage collected if they are only softly reachable, meaning they are eligible for garbage collection if the JVM determines memory is low.
* **Use Case**: Often used for memory-sensitive caches where you want to keep objects as long as possible but allow them to be reclaimed under memory pressure.
* **Example**:

java

Copy code

SoftReference<MyClass> softRef = new SoftReference<>(new MyClass());

MyClass obj = softRef.get();

**4. Phantom References**

* **Description**: Phantom references are the weakest form of reference and provide a way to clean up resources associated with an object before it is garbage collected.
* **Use Case**: Useful for managing direct memory or native resources, allowing you to perform cleanup actions when an object is about to be reclaimed by the garbage collector.
* **Example**:

java

Copy code

ReferenceQueue<MyClass> queue = new ReferenceQueue<>();

PhantomReference<MyClass> phantomRef = new PhantomReference<>(new MyClass(), queue);

MyClass obj = phantomRef.get(); // Always returns null

**Summary**

* **Strong references** are the default and most commonly used type.
* **Weak references** and **soft references** provide mechanisms for building more flexible caching strategies and managing memory more efficiently.
* **Phantom references** are used for more specialized scenarios involving resource cleanup or when you need to know precisely when an object is being reclaimed by the garbage collector.

Understanding and appropriately using these reference types can help optimize memory usage and manage resources effectively in Java applications, particularly when dealing with large-scale or resource-intensive systems.

3.5

Top of Form

Bottom of Form