**Core Spring**

* [@Bean](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Bean.html) - Annotated method produces a bean managed by the Spring IoC container
* Stereotype annotations
  + [@Component](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Component.html) - Marks annotated class as a bean found by the component-scanning and loaded into the application context
  + [@Controller](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Controller.html) - Marks annotated class as a bean for Spring MVC containing request handler
  + [@RestController](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html) - Marks annotated class as a @Controller bean and adds @ResponseBody to serialize returned results as messages
  + [@Configuration](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Configuration.html) - Marks annotated class as a Java configuration defining beans
  + [@Service](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Service.html) - Marks annotated class as a bean (as convention usually containing business logic)
  + [@Repository](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Repository.html) - Marks annotated class as a bean (as convention usually providing data access) and adds auto-translation from SQLException to DataAccessExceptions

**Bean state**

* [@PostConstruct](https://javaee.github.io/javaee-spec/javadocs/javax/annotation/PostConstruct.html) - Annotated method is executed after dependency injection is done to perform initialization
* [@PreDestroy](https://javaee.github.io/javaee-spec/javadocs/javax/annotation/PreDestroy.html) - Annotated method is executed before the bean is destroyed, e.g. on the shutdown

**Configuration**

* [@Import](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Import.html) - Imports one or more Java configuration classes @Configuration
* [@PropertySource](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/PropertySource.html) - Indicates the location of “**application.properties**” file to add key-value pairs to Spring Environment
* [@Value](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Value.html) - Annotated fields and parameters values will be injected.   
   **Eg:** @Value("${student.rollNo}")
* [@ComponentScan](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/ComponentScan.html) - Configures component scanning @Component, @Service, etc.

**Bean properties**

* [@Lazy](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Lazy.html) - Annotated bean will be lazily initialized on the first usage
* [@Profile](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Profile.html) - Indicates that beans will be only initialized if the defined profiles are active
* [@Scope](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Scope.html) - Defines bean creation scope, e.g. prototype, singleton, etc.  
   ***Singleton Scope is the default value if no other scope is specified***
* [@DependsOn](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/DependsOn.html) - Explicitly defines a dependency to other beans in terms of creation order
* [@Order](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/annotation/Order.html) - Defines sorting order if injecting a list of beans, but it does not resolve the priority if only a single bean is expected
* [@Primary](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Primary.html) - Annotated bean will be picked if multiple beans can be autowired
* [@Conditional](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Conditional.html) - Annotated bean is created only if conditions are satisfied
  + Additionally available in Spring Boot:
    - [@ConditionalOnBean](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/condition/ConditionalOnBean.html)
    - [@ConditionalOnMissingBean](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/condition/ConditionalOnMissingBean.html)
    - [@ConditionalOnClass](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/condition/ConditionalOnClass.html)
    - [@ConditionalOnMissingClass](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/condition/ConditionalOnMissingClass.html)
    - [@ConditionalOnProperty](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/condition/ConditionalOnProperty.html)
    - [@ConditionalOnMissingProperty](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/condition/ConditionalOnProperty.html)

**Bean injection**

* [@Autowired](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Autowired.html) - Beans are injected into annotated setters, fields, or constructor params.  
   Autowiring feature of spring framework **enables you to inject the object dependency implicitly**. It internally uses setter or constructor injection.
* [@Qualifier](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Qualifier.html) - Specifies the name of a bean as an additional condition to identify a unique candidate for autowiring

**Validation**

* [@Valid](https://javaee.github.io/javaee-spec/javadocs/javax/validation/Valid.html) - Mark a property, method parameters or return type for validation
* [@Validated](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/validation/annotation/Validated.html) - Variant of @Valid that allows validation of multiple groups, e.g. all fields of an annotated class
* [@NotNull](https://javaee.github.io/javaee-spec/javadocs/javax/validation/constraints/NotNull.html) - Must be not null
* [@NotEmpty](https://javaee.github.io/javaee-spec/javadocs/javax/validation/constraints/NotEmpty.html) - Must be not null nor empty
* [@NotBlank](https://javaee.github.io/javaee-spec/javadocs/javax/validation/constraints/NotBlank.html) - Must be not null and at least one non-whitespace character
* [@Digits](https://javaee.github.io/javaee-spec/javadocs/javax/validation/constraints/Digits.html) - Must be a number within accepted range
* [@Past](https://javaee.github.io/javaee-spec/javadocs/javax/validation/constraints/Past.html) - Must be an instant, date or time in the past
* [@Future](https://javaee.github.io/javaee-spec/javadocs/javax/validation/constraints/Future.html) - Must be an instant, date or time in the future

@Valid annotation for method level validation

@Validation apply constraints on a certain set of fields of the bean, and then later we want to apply constraints on another set of fields of the same bean.  
 **javax.validation.constraints.\***

@AssertFalse

@AssertTrue

@DecimalMax

@DecimalMin

@Digits

@Email

@Future

@FutureOrPresent

@Max

@Min

@Negative

@NegativeOrZero

@NotBlank

@NotEmpty

@NotNull

@Null

@Past

@PastOrPresent

@Pattern

@Positive

@PositiveOrZero

@Size

**Spring Boot**

* [@SpringBootConfiguration](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/SpringBootConfiguration.html) - Indicates Spring Boot application @Configuration
* [@EnableAutoConfiguration](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/EnableAutoConfiguration.html) - Enables application context auto-configuration to provide possibly needed beans based on the classpath
* [@ConfigurationProperties](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/context/properties/ConfigurationProperties.html) - Provides external binding of key value properties
* [@ConstructorBinding](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/context/properties/ConstructorBinding.html) - Bind properties by using constructor rather than setters
* [@ConfigurationPropertiesScan](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/context/properties/ConfigurationPropertiesScan.html) - Enables auto-detection of @ConfigurationProperties classes
* [@SpringBootApplication](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/SpringBootApplication.html) - Combination of @SpringBoot**Configuration**, @**EnableAutoConfiguration**, @ConfigurationPropertiesScan and @**ComponentScan**
* [@EntityScan](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/domain/EntityScan.html) - Configures base packages to scan for entity classes
* [@EnableJpaRepositories](https://docs.spring.io/spring-data/data-jpa/docs/current/api/org/springframework/data/jpa/repository/config/EnableJpaRepositories.html) - Enables auto-configuration of JPA repositories

**Spring Boot Tests**

* [@SpringBootTest](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/context/SpringBootTest.html) - Annotated test class will load the entire application context for integration tests
* [@WebMvcTest](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/autoconfigure/web/servlet/WebMvcTest.html) - Annotated test class will load only the web layer (service and data layer are ignored)
* [@DataJpaTest](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/autoconfigure/orm/jpa/DataJpaTest.html) - Annotated class will load only the JPA components
* [@MockBean](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/mock/mockito/MockBean.html) - Marks annotated field as a mock and loads it as a bean into the application context
* [@SpyBean](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/mock/mockito/SpyBean.html) - Allows partial mocking of beans
* [@Mock](https://www.javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mock.html) - Defines annotated field as a mock

**Spring Test**

* [@ContextConfiguration](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/context/ContextConfiguration.html) - Defines @Configuration to load application context for integration test
* [@ExtendWith](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/extension/ExtendWith.html) - Defines extensions to execute the tests with, e.g. MockitoExtension
* [@SpringJUnitConfig](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/context/junit/jupiter/SpringJUnitConfig.html) - Combines @ContextConfiguration and @ExtendWith(SpringExtension.class)
* [@TestPropertySource](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/test/context/TestPropertySource.html) - Defines the location of property files used in integration tests
* [@DirtiesContext](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/test/annotation/DirtiesContext.html) - Indicates that annotated tests dirty the application context and will be cleaned after each test
* [@ActiveProfiles](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/test/context/ActiveProfiles.html) - Defines which active bean definition should be loaded when initializing the test application context
* [@Sql](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/context/jdbc/Sql.html) - Allows defining SQL scripts and statements to be executed before and after tests

**Transactions**

* [@EnableTransactionManagement](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/EnableTransactionManagement.html) - Enables annotation-driven transaction declaration @Transactional
* [@Transactional](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html) - Annotated methods will be executed in a transactional manner

**Spring JPA and Hibernate**

* [@Id](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Id.html) - Marks annotated field as a primary key of an entity
* [@GeneratedValue](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/GeneratedValue.html) - Provides generation strategy of primary keys
* [@Entity](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Entity.html) - Marks annotated class as an entity
* [@Column](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Column.html) - Provides additional configuration for a field, e.g. column name
* [@Table](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Table.html) - Provides additional configuration for an entity, e.g. table name
* [@PersistenceContext](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/PersistenceContext.html) - EntityManger is injected into annotated setters and fields
* [@Embedded](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Embedded.html) - Annotated field is instantiated as a value of an Embeddable class
* [@Embeddable](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Embeddable.html) - Instances of an annotated class are stored as part of an entity
* [@EmbeddedId](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/EmbeddedId.html) - Marks annotated property as a composite key mapped by an embeddable class
* [@AttributeOverride](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/AttributeOverride.html) - Overrides the default mapping of a field
* [@Transient](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Transient.html) - Annotated field is not persistent
* [@CreationTimestamp](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/annotations/CreationTimestamp.html) - Annotated field contains the timestamp when an entity was stored for the first time
* [@UpdateTimestamp](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/annotations/UpdateTimestamp.html) - Annotated field contains the timestamp when an entity was updated last time
* [@ManyToOne](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/ManyToOne.html) - Indicates N:1 relationship, the entity containing annotated field has a single relation to an entity of other class, but the other class has multiple relations
* [@JoinColumn](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/JoinColumn.html) - Indicates a column for joining entities in @ManyToOne or @OneToOne relationships at the owning side or unidirectional @OneToMany
* [@OneToOne](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/OneToOne.html) - Indicates 1:1 relationship
* [@MapsId](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/MapsId.html) - References joining columns of owning side of @ManyToOne or @OneToOne relationships to be the primary key of referencing and referenced entities
* [@ManyToMany](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/ManyToMany.html) - Indicates N:M relationship
* [@JoinTable](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/JoinTable.html) - Specifies an association using a join table
* [@BatchSize](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/annotations/BatchSize.html) - Defines size to lazy load a collection of annotated entities
* [@FetchMode](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/annotations/FetchMode.html) - Defines fetching strategy for an association, e.g. loading all entities in a single subquery

**Spring Security**

* [@EnableWebSecurity](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/config/annotation/web/configuration/EnableWebSecurity.html) - Enables web security
* [@EnableGlobalMethodSecurity](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/config/annotation/method/configuration/EnableGlobalMethodSecurity.html) - Enables method security
* [@PreAuthorize](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/access/prepost/PreAuthorize.html) - Defines access-control expression using SpEL, which is evaluated before invoking a protected method
* [@PostAuthorize](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/access/prepost/PostAuthorize.html) - Defines access-control expression using SpEL, which is evaluated after invoking a protected method
* [@RolesAllowed](https://javaee.github.io/javaee-spec/javadocs/javax/annotation/security/RolesAllowed.html) - Specifies a list of security roles allowed to invoke protected method
* [@Secured](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/access/annotation/Secured.html) - Java 5 annotation for defining method level security

**Spring AOP**

* [@EnableAspectJAutoProxy](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/EnableAspectJAutoProxy.html) - Enables support for handling components marked with @Aspect
* [@Aspect](https://javadoc.io/static/org.aspectj/aspectjrt/1.9.5/org/aspectj/lang/annotation/Aspect.html) - Declares an annotated component as an aspect containing pointcuts and advices
* [@Before](https://javadoc.io/static/org.aspectj/aspectjrt/1.9.5/org/aspectj/lang/annotation/Before.html) - Declares a pointcut executed before the call is propagated to the join point
* [@AfterReturning](https://javadoc.io/static/org.aspectj/aspectjrt/1.9.5/org/aspectj/lang/annotation/AfterReturning.html) - Declares a pointcut executed if the join point successfully returns a result
* [@AfterThrowing](https://javadoc.io/static/org.aspectj/aspectjrt/1.9.5/org/aspectj/lang/annotation/AfterThrowing.html) - Declares a pointcut executed if the join point throws an exception
* [@After](https://javadoc.io/static/org.aspectj/aspectjrt/1.9.5/org/aspectj/lang/annotation/After.html) - Declares a pointcut executed if the join point successfully returns a result or throws an exception
* [@Around](https://javadoc.io/static/org.aspectj/aspectjrt/1.9.5/org/aspectj/lang/annotation/Around.html) - Declares a pointcut executed before the call giving control over the execution of the join point to the advice
* [@Pointcut](https://javadoc.io/static/org.aspectj/aspectjrt/1.9.5/org/aspectj/lang/annotation/Pointcut.html) - Externalized definition a pointcut expression

@GetMapping: It maps the HTTP GET requests on the specific handler method. It is used to create a web service endpoint that fetches It is used instead of using: @RequestMapping(method = RequestMethod.GET)

@PostMapping: It maps the HTTP POST requests on the specific handler method. It is used to create a web service endpoint that creates It is used instead of using: @RequestMapping(method = RequestMethod.POST)

@PutMapping: It maps the HTTP PUT requests on the specific handler method. It is used to create a web service endpoint that creates or updates It is used instead of using: @RequestMapping(method = RequestMethod.PUT)

@DeleteMapping: It maps the HTTP DELETE requests on the specific handler method. It is used to create a web service endpoint that deletes a resource. It is used instead of using: @RequestMapping(method = RequestMethod.DELETE)

@PatchMapping: It maps the HTTP PATCH requests on the specific handler method. It is used instead of using: @RequestMapping(method = RequestMethod.PATCH)  
  
**POST** **creates** a resource. **PUT** **replaces** a resource. **PATCH** **updates** a resource. DELETE **removes** a resource.

@RequestBody: It is used to bind HTTP request with an object in a method parameter. Internally it uses HTTP MessageConverters to convert the body of the request. When we annotate a method parameter with @RequestBody, the spring framework binds the incoming HTTP request body to that parameter.

@ResponseBody: It binds the method return value to the response body. It tells the Spring Boot Framework to serialize a return an object into JSON and XML format.

@PathVariable: It is used to extract the values from the URI. It is most suitable for the RESTful web service, where the URL contains a path variable. We can define multiple @PathVariable in a method.   
Eg: ***Setting @PathVariable as Not Required*** (@PathVariable*(required = false)* String id)

@RequestParam: It is used to extract the query parameters form the URL. It is also known as a query parameter. It is most suitable for web applications. It can specify default values if the query parameter is not present in the URL.

@RequestHeader: It is used to get the details about the HTTP request headers. We use this annotation as a method parameter. The optional elements of the annotation are name, required, value, defaultValue. For each detail in the header, we should specify separate annotations. We can use it multiple time in a method

@RestController: It can be considered as a combination of @Controller and @ResponseBody annotations. The @RestController annotation is itself annotated with the @ResponseBody annotation. It eliminates the need for annotating each method with @ResponseBody.

@RequestAttribute: It binds a method parameter to request attribute. It provides convenient access to the request attributes from a controller method. With the help of @RequestAttribute annotation, we can access objects that are populated on the server-side.

@RequestMapping: It is used to map the web requests. It has many optional elements like consumes, header, method, name, params, path, produces, and value. We use it with the class as well as the method.

@EnableAutoConfiguration: It auto-configures the bean that is present in the classpath and configures it to run the methods. The use of this annotation is reduced in Spring Boot 1.2.0 release because developers provided an alternative of the annotation, i.e. @SpringBootApplication.

@SpringBootApplication: It is a combination of three annotations **@EnableAutoConfiguration**, **@ComponentScan**, and **@Configuration**.

@Repository: It is a class-level annotation. The repository is a DAOs (Data Access Object) that access the database directly. The repository does all the operations related to the database.

@Service: It is also used at class level. It tells the spring that class contains the business logic.

@Controller: The @Controller is a class-level annotation. It is a specialization of @Component. It marks a class as a web request handler. It is often used to serve web pages. By default, it returns a string that indicates which route to redirect. It is mostly used with @RequestMapping annotation.

@Component: It is a class-level annotation. It is used to mark a Java class as a bean. A Java class annotated with @Component is found during the classpath. The Spring Framework pick it up and configure it in the application context as a Spring Bean.

@Bean: It is a method-level annotation. It is an alternative of XML <bean> tag. It tells the method to produce a bean to be managed by Spring Container.

@ComponentScan: It is used when we want to scan a package for beans. It is used with the annotation @Configuration. We can also specify the base packages to scan for Spring Components.

@Configuration: It is a class-level annotation. The class annotated with @Configuration used by Spring Containers as a source of bean definitions.

@Autowired: Spring provides annotation-based auto-wiring by providing @Autowired annotation. It is used to Autowire spring bean on setter methods, instance variable, and constructor. When we use @Autowired annotation, the spring container auto-wires the bean by matching data-type.

@Required: It applies to the bean setter method. It indicates that the annotated bean must be populated at configuration time with the required property, else it throws an exception BeanInitilizationException. **API Gateway** is a concept of having a single point of entry to access all of the services in the backend.



* So, when any device wants to access resources from the server, they make a call to the API-Gateway. API-Gateway then reaches out to rest of the services which actually take care of serving the user with what they need.
* To demonstrate this, we will create a Zuul based API-Gateway in Spring Boot. We then will use API Gateway to make a call to the Microservice, namely **Product** Service.

**Eureka vs Zookeeper: What are the differences?**

* **What is Eureka?** AWS Service registry for resilient mid-tier load balancing and failover. Eureka is a REST (Representational State Transfer) based service that is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers.
* **What is Zookeeper?** Because coordinating distributed systems is a Zoo. A centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications.
* Eureka and Zookeeper can be primarily classified as **"Open Source Service Discovery"** tools.
* **"Easy setup and integration with spring-cloud "** is the primary reason why developers consider Eureka over the competitors, whereas **"High performance, easy to generate node specific config"** was stated as the key factor in picking Zookeeper.
* Eureka is an open source tool with **7.98K** GitHub stars and **2.2K** GitHub forks. [Here's](https://github.com/Netflix/eureka) a link to Eureka's open source repository on GitHub.
* **Uber Technologies**, **Pinterest**, and **Shopify** are some of the popular companies that use Zookeeper, whereas Eureka is used by **Notify-e**, **Swingvy**, and **LabNetwork**. Zookeeper has a broader approval, being mentioned in **116** company stacks & **48** developer’s stacks; compared to Eureka, which is listed in **7** company stacks and **14** developer stacks.

|  |  |
| --- | --- |
| **Pros of Eureka** | **Pros of Zookeeper** |
| Easy setup and integration with spring-cloud | High performance , easy to generate node specific Config |
| Health checking | Kafka support |
| Circuit breaker | Java |
| Web UI | Spring Boot Support |
| Netflix battle tested components | Supports extensive distributed IPC |
| Service discovery | Supports DC/OS |
| Monitoring | Used in ClickHouse |
| Open Source | Curator |
|  | Embeddable In Java Service |
|  | Used in Hadoop |

**ZooKeeper** is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications.  
  
**ZooKeeper** follows a simple client-server model where clients are nodes (i.e., machines) that make use of the service, and servers are nodes that provide the service. A collection of ZooKeeper servers forms a ZooKeeper ensemble. Each ZooKeeper server can handle a large number of client connections at the same time.  
 **API access controls and gateways:** How to control the security of micro services.

**Application Programming Interface (API):** A software interface that allows users to configure and interact with other programs, usually by calling from a list of functions.

**Container:** Resource isolation at the OS (rather than machine) level, usually (in UNIX-based systems) in user space. Isolated elements vary by containerization strategy and often include file system, disk quota, CPU and memory, I/O rate, root privileges, and network access. Much lighter-weight than machine-level virtualization and sufficient for many isolation requirement sets.

**Continuous Delivery:** A software engineering approach in which continuous integration, automated testing, and automated deployment capabilities allow software to be developed and deployed rapidly, reliably, and repeatedly with minimal human intervention.

**Decouple:** Breaking monolithic applications into smaller components so legacy enterprises can pursue digital transformation.

**Distributed System:** Any system or application that operates across a wide network of services or nodes.

**Distributed Tracing:** A category of tools and practices that allow developers to analyze the behavior of a service and troubleshoot problems by creating services that record information about requests and operations that are performed.

**Domain-Driven Design:** A philosophy for developing software in which development is focused primarily on the business logic, the activities and issues that an application is supposed to perform or solve.

**Enterprise Service Bus (ESB):** A utility that combines a messaging system with middleware to provide comprehensive communication services for software applications.

**Eventual Consistency:** A data consistency model used to make distributed applications highly available by keeping data in sync and up-to-date across all services or nodes.

**Fault isolation:** Enables a microservice to crash and restart without causing a service outage.  
  
**Function as a service:** A category of cloud computing services that provides a platform allowing customers to develop, run, and manage application functionalities without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app.

**Holacracy:** A management practice for organizations that are separated into autonomous and independent departments based on roles, which can organize themselves and make decisions based on their duties. Holacracies are focused on rapidly iterating.

**Java Virtual Machine (JVM):** Abstracted software that allows a computer to run a Java program.

**Message Broker:**Middleware that translates a message sent by one piece of software to be read by another piece of software.

**Microservices architecture:**A development method of designing your applications as modular services that seamlessly adapt to a highly scalable and dynamic environment.

**Monolith:**A style of software architecture where the data and UI are all found on one platform, in a single program.

**Orchestration:** The method to automate the management and deployment of your applications and containers.

**OWAT2:** Microservices security best practice with OAuth.

**Serverless:** Cloud computing execution model in which the cloud provider dynamically manages the allocation of machine resources.

**Service Discovery:** The act of finding the network location of a service instance for further use.

**Service Mesh:** An infrastructure layer focused on service-to-service communication, primarily used for distributed systems and cloud-native applications.

**Sociocracy:** A mode of governance without a centralized power structure, aiming for less independence between teams to focus on organization-wide strategy.

**Twelve-factor application technology:** A methodology for building modern, scalable, maintainable software-as-a-service apps.

**Web Service:** A function that can be accessed over the web in a standardized way using APIs that are accessed via HTTP and executed on a remote system.

WHY MICROSERVICES ARE PREFERRED OVER MONOLITHIC?

Monolithic architecture is being used for a long time. It is the fundamental approach to building an enterprise software application that catered to numerous business requirements.

Monolith architecture at the starting phase is simple to develop and deploy. However, as the requirements and functionalities increase over time, it becomes complex to maintain. Monolithic has the following disadvantages:

To update a part of the system, the entire application has to be redeployed.

Adopting new technologies or frameworks is very difficult because all the functionalities are built on homogeneous technologies.

The whole application might halt even if one service goes unstable.

Using agile methodologies in monolithic application is difficult.

To overcome the disadvantages of monolithic, non-overlapping services such as microservices are preferred. Now let us look at the advantages of microservices in detail.

***ADVANTAGES OF MICROSERVICES***

***INDEPENDENTLY REDEPLOYABLE***

Microservices are independently re-deployable. When a single component of the application needs to be changed, that particular microservice alone can be redeployed. This aspect helps during agile development when making frequent code changes to the application and deployment to production.

CROSS-FUNCTIONAL TEAMS

Microservices approach is to split teams into micro teams that can function independently. For instance, each team will have a mix of UI specialists, middleware specialists and DBAs rather than having all UI specialists in one team, database specialists in another team and a third team full of middleware specialists.

DECENTRALIZED DATA MANAGEMENT

Microservices architecture has a private database for each microservice to implement a particular business function. In case, one particular microservice needs to access the data available with other microservice, it can do so via API calls.

DECENTRALIZED GOVERNANCE

Microservices are self-governing entities in a software application. Whether it is design or implementation of functionalities, microservices can function independently without a common standard of governance. When the need arises to use common services like security requirements, monitoring, throttling, etc., microservices communicate through APIs.

SECURITY PROTOCOLS

Standard API-security protocols, OAuth2, and OpenID Connect are used in a microservice architecture. OAuth2 allows access when a ‘by-reference token’ or ‘Access token’ is provided during authentication. OpenID Connect uses both the access token and ID token for authentication. ID token, as the name suggests, contains user information.

SERVICE REGISTRY AND SERVICE DISCOVERY

Microservices are registered with the help of service registry. Why service registry is required? When teams develop microservices independently there are chances that they have developed duplicate microservices. When a service registry is available, the teams can share and use microservices optimally.

The registry contains the location of the microservice and the various instances. To identify a specific microservice in the Service Registry, a discovery mechanism is required. Client-side Discovery and Server-side Discovery are mechanisms used to identify a microservice.

In client-side discovery, the API gets the location by querying the Service Registry directly. In Server-side discovery, the request is first sent to the load balancer running on a known location and then the load balancer calls the service registry to determine the microservice location.

CODES AND LANGUAGES OF YOUR CHOICE

Microservices approach solves development problems by allowing different teams to pick their own codes/languages (Java, C++, J Script).

BRANDS USING MICROSERVICES

Microservices are used by major players in the tech industry including Amazon, Netflix, Twitter, PayPal, eBay and Uber and the list goes on.

HOW TO DESIGN A MICROSERVICE?

Defining the boundaries of microservices and aligning them with the business requirements is essential during the design phase.

A microservice should be designed in such a way that each microservice should have focus towards only one specific functionality in the software. Assigning more than one functionality to a microservice is generally considered a bad design. Microservices architecture helps accommodate agile development and delivery methodologies.

To design a microservices, you need to get the following aspects right:

Messaging

Integrating

Deployment

MESSAGING

To achieve efficient ‘messaging’ in a microservices architecture, you need lightweight mechanisms. REST (Representational State Transfer) and Thrift architectural types are for synchronous messaging and AMQP, STOMP or MQTT are for asynchronous messaging. With regard to message formats, JSON, XML and for service interfaces, Swagger, RAML and Thrift IDL (Interface Description Language) are available.

INTEGRATING

To integrate microservices in a software application, you need protocols and gateways. There are many approaches to implement gateways such as point-to-point and API-Gateway.

In a point-to-point approach, standard protocols like HTTP and JSON (JavaScript Object Notation) are used for integrating microservices along with REST APIs. However, point-to-point connectivity has limitations such as:

Communication between the services and clients cannot be monitored, traced or filtered.

Implementation of common functionalities like end-user authentication has to be repeated for each and every microservice.

API-Gateway (API-GW) approach combines both microservices and API management and acts as a bridge between the client and microservices. The advantages of API gateway are:

Ability to expose different APIs to different clients.

Non-functional capabilities like security, monitoring are implemented at the Gateway- level and not repeated at every microservice-level.

Lightweight and quick communication is achieved at microservices-level.

DEPLOYMENT

Microservices deployment should satisfy requirements such as:

Deploy or undeploy each microservice independently.

Building and deploying microservices should be easy and quick.

Ability to scale at microservice-level to handle more traffic for specific service.

Even if one microservice fails it should not affect the functioning of other microservices.

**INTRODUCING MICROSERVICES TOOLS & TERMINOLOGIES**

There are various tools & terminologies required for designing a Microservices architecture. Let’s explore some of them here.

BROKER/MESSAGING

RabbitMQ is the widely used open source message broker tool. It can be used to connect two or microservices in a microservices architecture to scale applications and can be used to exchange events between services as well.

APACHE KAFKA

Apache Kafka is a distributed event streaming platform and can be used for communication between microservices. The advantage of Kafka is that; the sender is oblivious of the receiver. Further, it helps achieve a centralized security control because you can set Access Control Lists (ACL) limiting specific senders and receivers to access specific data in the system.

APACHE ZOOKEEPER

Apache Zookeeper can be used to manage microservices especially when migrating from monolithic architecture. Both Kafka and Zookeeper can be used for message queuing between microservices and external sources.

The basic flow here is to start Zookeeper and kafka server as a broker that mediates messages between publisher and subscriber. Then comes the producer API is called to create message and the consumer API that consumes message from Kafka queue.

SERVICE REGISTRY – EUREKA, ZOOKEEPER, CONSUL

Eureka is a service directory where every microservice is registered. When a client microservice needs to communicate with another microservice, it first contacts the Eureka server and then through the servier, the dependent microservice is contacted. Consul is a service directory like Eureka. Microservices can communicate using Consul service directory.

CIRCUIT BREAK

For microservices communication, circuit breakers are needed to trace any failure between distributed services. Hystrix and Jrugged are among the preferred circuit breakers.

JRUGGED  
Jrugged is a Java. Fault-tolerant library used to build production-ready server code in Java. It provides simple circuit breaker implementation with a few monitoring capabilities. However, it lacks the much expected features from fault tolerance library such as fallbacks or bulkheads.

HYSTRIX  
Hystrix or Netflix is implemented to trace and control latency/failure between distributed services. It continuously monitors the calls and when any dependent service response exceeds a threshold limit, it breaks the circuit.

A fallback policy is put in place so that all the requests go through that fallback path while the dependent source is given time to recover. After a preset time, the circuit is closed for the request to flow normally.

GATEWAY

**ZUUL**  
Zuul is a microservice acting as an API gateway. Also known as Edge service, it receives requests from the front-end UI and allocates the requests to appropriate internal microservices at the backend.

**NETTY**   
Netty is an event-driven client/server framework/gateway used for developing high-performance protocol servers and clients. It supports FTP, SMTP, HTTP, UDP and TCP applications.

**FINAGLE**  
Finagle is an Apache-licensed RPC stack, developed by Twitter and is written in Scala. It is based on asynchronous NIO APIs and both its servers and clients are built on top of Netty.

IDENTITY AND ACCESS MANAGEMENT

Identity and access management (IAM) ensures that the internal, intra-application communications among microservices are secure.

KERBEROS

Kerberos is an authentication tool that can secure big data products including Apache Kafka. In microservices architecture, Kerberos is used to achieve single sign (authentication) on functionality.

OAUTH3  
OAuth3 is an authorization protocol that allows users to access specific data using web/desktop/mobile apps.

OPENID  
OpenID Connect is based on OAuth3 and used for authorization. It uses UserInfo, ID Token and other parameters over OAuth3. However, it cannot be used for authentication.

DASHBOARD  
Microservices Dashboard app allows to visualize links between Microservices and other ecosystems that encompass it. It consists of UI, Resources, Backends and Microservices as columns. Elastic search can analyze large data volumes.

KIBANA  
Kibana is a plugin that helps visualize Elastic search data. Together, Kibana and Elastic search can be used to host a cloud service on Amazon Web Services.

SPLUNK

Splunk fetches a data pile and let’s people search the pile to extract the information they require. The three key components of Splunk are: forwarder, indexers and search head. Forwarder sends the data to remote indexers. Indexers, as the name suggests, is responsible for storing and indexing data. Search head is the front-end of the web interface.

LOG AGGREGATION

Log Aggregation, as the name suggests, is the process of aggregating log files in order to organize the data for easy searching.

LOGSTASH

Logstash, like Splunk, is also a log aggregator. It can take logs from different sources and with the help of input plugins, filter the data and sends the desired output.

METRICS

Metrics help understand how your microservices architecture performs. When service traffic is passed through a service mesh that gathers telemetry data, it automatically collects fine-grained, high-level app information. These metrics include data like Success rates, Request volume, Request duration, Request size. Request and error counts, Latency and HTTP Error codes. Based on the metrics, you can optimize the performance of your architecture.

DROPWIZARD

Dropwizard has a collection of best-in-class libraries to support application metrics, logging etc., In microservices, Dropwizard can be used for its various features like standalone server and single JAR.

ACTUATOR

Actuator or Spring Boot Actuator offers various production-grade services to the application such as logs, metrics, environment variables etc., This information can be exposed by Actuator endpoints via HTTP, JMX or by logging in directly through SSH.

PROMETHEUS

Prometheus is an open source project by SoundCloud and in microservices context, it support multi-dimensional data collection. The main strength of Prometheus is its ‘querying’ capabilities with which it can help quickly diagnose problems. Prometheus server runs independently and so, even if a few parts in the architecture fails, it can still rely on the server to view the system statistics.

CONTAINERS

Containers are lightweight envelops assigned for each Microservice making the software portable. It has the required code to execute the specific Microservice instance. The advantage of Container is that it can be dynamically created and destroyed which allows microservices to be scalable and highly available.

DOCKER

Docker is a **containerization platform**. A Docker container encapsulates a Microservice and responsible for a particular business functionality. With the help of Dockers, encapsulate each microservice in the architecture and make the application run regardless of the host environment. Docker can integrate popular tools and services including Amazon Web Services and Microsoft Azure.

ROCKET

Rocket or CoreOS rkt is a container runtime, an alternate to the Docker. Rocket consists of two elements, Actool, the first element is responsible for building container, handling its validation and discovery. The second element rkt (pronounced as Rocket) fetches and runs container images.

The advantages of Rocket container are:

Independent composition (download/install/run) of containers

Security isolation that is pluggable and includes image auditing

Simple discovery of container images and independent deployment

Well-specified runtime so that tools are implemented consistently

ORCHESTRATOR

Orchestrator is responsible for coordinating front-end and back-end tasks especially in a microservices architecture.

KUBERNETES

Kubernetes is an **orchestration platform** created by Google, used to solve challenges in designing a microservice architecture such as traffic routing, and distributed monitoring, service discoveries and so on. Kubernetes is language agnostic and open source as well.

MESOS

Mesos or Apache Mesos is a cluster manager. It separates data centre resources in order to ease the deployment and management of distributed applications. Mesos, along with DC/OS (mesosphere-backed framework) built on top of Mesos and Apache Kafka, can achieve highly-scalable streaming infrastructure and cloud-native platform.

DOCKER SWARM

Docker Swarm is a container orchestration platform. It is a container management service and a native clustering engine for and by Docker. Any tool or service running with Docker containers will run in Docker Swarm as well. Docker Swarm is easy to setup and helps quick container deployment.

CONTINUOUS DELIVERY (CD)

Continuous Delivery (CD) in microservice architecture is challenging because of four major reasons:

Complexity of the distributed system makes it difficult to maintain the integrity.

Managing frequent feature releases especially when many microservices are involved.

Deploying disparate technology stacks is challenging.

Best-suitable tools for CD workflows can be hard to find.

To achieve CD on Microservices, you need to have an effective test strategy, evaluate your current CD practices, a proper plan to use various environments, managing configuration and preparation for instability and bugs.

JENKINS

Jenkins or Jenkins X is an open source system to achieve continuous integration, continuous delivery (CI/CD) and automated testing in Kubernetes. It is responsible for microservices deployment pipeline which saves time to create pipelines for each microservice.

ASGARD

Asgard is a web interface for cloud management and application deployments. It is part of the open source Netflix projects and released under the Apache Licence (v 2.0).

AMINATOR

Aminator is one of the AMI creation tools and used for creating custom Amazon Machine Images (AMIs).

EXTERNALIZED CONFIGURATION

Externalization Configuration is a pattern in which an application is run in multiple environments like development, testing, production without any code modifications in the application. Externalized configuration stores the configuration information including database, file system, environmental variables in an external store.

During the start of the application, microservices will load the configuration from the external store. When running, microservices have an option to reload the same configuration. This prevents any service restart.

CONSUL

Consul is a distributed service mesh. It can be used to connect and configure services in any runtime platform and cloud. It provides service discovery, health monitoring and config services. In short, it can be termed as a combination of DNS server plus ZooKeeper for service discovery and Nagios for health monitoring all rolled up into one system.

DECIDER

Decider is a state machine service and monitors the workflow events such as task completion and failure. It checks the current state of the workflow and identifies the next state. Later, it schedules the tasks and updates the status of the workflow. To schedule tasks, Decider uses a distributed queue (eg: dyno-queues).

***The Twelve Factors Applied to Microservices***

**1 – Codebase**

#### One codebase per service, tracked in revision control; many deploys

The Twelve‑Factor App recommends [one codebase per app](https://12factor.net/codebase). In a microservices architecture, the correct approach is actually one codebase per service. Additionally, we strongly recommend the use of Git as a repository, because of its rich feature set and enormous ecosystem. GitHub has become the default Git hosting platform in the open source community, but there are many other excellent Git hosting options, depending on the needs of your organization.

**2 – Dependencies**

#### Explicitly declare and isolate dependencies

As suggested in The Twelve‑Factor App, regardless of what platform your application is running on, [use the dependency manager](https://12factor.net/dependencies) included with your language or framework. How you install operating system or platform dependencies depends on the platform:

* In non-containerized environments, use a configuration management tool (Chef, Puppet, Ansible) to install system dependencies.
* In a containerized environment, do this in the Dockerfile.

**Note:** We recommend that you choose a dependency management mechanism in the context of your comprehensive Infrastructure‑as‑Code strategy, not as an isolated decision. See Martin Fowler’s writings on [Infrastructure‑as‑Code](https://martinfowler.com/bliki/InfrastructureAsCode.html) and download the O’Reilly report [Infrastructure as Code](https://www.nginx.com/resources/library/infrastructure-as-code/) by Kief Morris.

**3 – Config**

#### Store configuration in the environment

Anything that varies between deployments can be considered configuration. The Twelve‑Factor App guidelines recommend [storing all configuration in the environment](https://12factor.net/config), rather than committing it to the repository. We recommend the following specific practices:

* Use non‑version controlled **.env** files for local development. Docker supports the loading of these files at runtime.
* Keep all **.env** files in a secure storage system, such as [Vault](https://www.vaultproject.io/), to keep the files available to the development teams, but not commited to Git.
* Use an environment variable for anything that can change at runtime, and for any secrets that should not be committed to the shared repository.
* Once you have deployed your application to a delivery platform, use the delivery platform’s mechanism for managing environment variables.

### **4 –ervices**

**4 - Treat backing services as attached resources**

The Twelve‑Factor App guidelines define a [backing service](https://12factor.net/backing-services) as “any service the app consumes over the network as part of its normal operation.” The implication for microservices is that anything external to a service is treated as an attached resource, including other services. This ensures that every service is completely portable and loosely coupled to the other resources in the system. Additionally, the strict separation increases flexibility during development – developers only need to run the service(s) they are modifying, not others.

**5 – Build, Release, Run**

#### Strictly separate build and run stages

To support [strict separation of build, release, and run stages](https://12factor.net/build-release-run), as recommended by The Twelve‑Factor App, we recommend the use of a continuous integration/continuous delivery (CI/CD) tool to automate builds. Docker images make it easy to separate the build and run stages. Ideally, images are created from every commit and treated as deployment artifacts.

### **6 – Processes**

#### Execute the app in one or more stateless processes

For microservices, the important point in the [Processes factor](https://12factor.net/processes) is that your application needs to be stateless. This makes it easy to scale a service horizontally by simply adding more instances of that service. Store any stateful data, or data that needs to be shared between instances, in a backing service.

### **7 – Data Isolation**

#### Each service manages its own data

As a modification to make the [Port binding factor](https://12factor.net/port-binding) more useful for microservices, we recommend that you allow access to thepersistent data owned by a service only via the service’s API. This prevents implicit service contracts between microservices and ensures that microservices can’t become tightly coupled. Data isolation also allows the developer to choose, for each service, the type of data store that best suits its needs.

### **8 – Concurrency**

#### Scale out via the process model

The [Unix process model](https://devcenter.heroku.com/articles/process-model) is largely a predecessor to a true microservices architecture, insofar as it allows specialization and resource sharing for different tasks within a monolithic application. In a microservices architecture, you can horizontally scale each service independently, to the extent supported by the underlying infrastructure. With containerized services, you further get the [concurrency](https://12factor.net/concurrency) recommended in the Twelve‑Factor App, for free.

### **9 – Disposability**

#### Maximize robustness with fast startup and graceful shutdown

Instances of a service need [to be disposable](https://12factor.net/disposability) so they can be started, stopped, and redeployed quickly, and with no loss of data. Services deployed in Docker containers satisfy this requirement automatically, as it’s an inherent feature of containers that they can be stopped and started instantly. Storing state or session data in queues or other backing services ensures that a request is handled seamlessly in the event of a container crash. We are also proponents of using a backing store to support [crash‑only design](https://lwn.net/Articles/191059/).

### **10 – Dev/Prod Parity**

#### Keep development, staging, and production as similar as possible

Keep all of your environments – development, staging, production, and so on – [as identical as possible](https://12factor.net/dev-prod-parity), to reduce the risk that bugs show up only in some environments. To support this principle, we recommend, again, the use of containers – a very powerful tool here, as they enable you to run exactly the same execution environment all the way from local development through production. Keep in mind, however, that differences in the underlying data can still cause differences at runtime.

### **11 – Logs**

#### Treat logs as event streams

Instead of including code in a microservice for routing or storing logs, use one of the many good log‑management solutions on the market, several of which are listed in the [Twelve‑Factor App](https://12factor.net/logs). Further, deciding how you work with logs needs to be part of a larger APM and/or PaaS strategy.

### **12 – Admin Processes**

#### Run admin and management tasks as one‑off processes

In a production environment, run [administrative and maintenance tasks](https://12factor.net/admin-processes) separately from the app. Containers make this very easy, as you can spin up a container just to run a task and then shut it down.

***The Twelve Factors Applied to Microservices***

1 – ***Codebase*** : One codebase per service, tracked in revision control; many deploys  
2 – ***Dependencies*** : Explicitly declare and isolate dependencies  
3 – ***Config*** : Store configuration in the environment  
4 – ***Backing* *Services*** : Treat backing services as attached resources  
5 – ***Build, Release, Run*** : Strictly separate build and run stages  
6 – ***Processes*** : Execute the app in one or more stateless processes  
7 – ***Data Isolation*** : Each service manages its own data  
8 – ***Concurrency*** : Scale out via the process model  
9 – ***Disposability*** : Maximize robustness with fast startup and graceful shutdown  
10 – ***Dev/Prod Parity*** : Keep development, staging, and production as similar as possible  
11 – ***Logs*** : Treat logs as event streams  
12 – ***Admin Processes*** : Run admin and management tasks as one-off processes  
  
<https://dzone.com/articles/design-patterns-for-microservices>

So **circuit breaker** is a kind of a wrapper of the method which is doing the service call and it monitors the service health and once it gets some issue, the circuit breaker trips and all further calls go to the circuit breaker fall back and finally restores automatically once the service came back.  
  
<https://howtodoinjava.com/spring-cloud/spring-hystrix-circuit-breaker-tutorial/>

### Saga Pattern

**Problem** When each service has its own database and a business transaction spans multiple services, how do we ensure data consistency across services? For example, for an e-commerce application where customers have a credit limit, the application must ensure that a new order will not exceed the customer’s credit limit. Since Orders and Customers are in different databases, the application cannot simply use a local ACID transaction.

**Solution** A Saga represents a high-level business process that consists of several sub requests, which each update data within a single service. Each request has a compensating request that is executed when the request fails. It can be implemented in two ways:

1. Choreography — When there is no central coordination, each service produces and listens to another service’s events and decides if an action should be taken or not.
2. Orchestration — an orchestrator (object) takes responsibility for a saga’s decision making and sequencing business logic.

# **HTTP response status codes**

HTTP response status codes indicate whether a specific [HTTP](https://developer.mozilla.org/en-US/docs/Web/HTTP) request has been successfully completed. Responses are grouped in five classes

1. Informational responses (100–199)
2. Successful responses (200–299)
3. Redirects (300–399)
4. Client errors (400–499)
5. Server errors (500–599)

The below status codes are defined by [section 10 of RFC 2616](https://datatracker.ietf.org/doc/html/rfc2616#section-10). You can find an updated specification in [RFC 7231](https://datatracker.ietf.org/doc/html/rfc7231#section-6.5.1).

If you receive a response that is not in this list, it is a non-standard response, possibly custom to the server's software.

[**Information responses**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#information_responses)[**100 Continue**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/100)

This interim response indicates that everything so far is OK and that the client should continue the request, or ignore the response if the request is already finished.

[**101 Switching Protocol**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/101)

This code is sent in response to an [Upgrade](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Upgrade) request header from the client, and indicates the protocol the server is switching to.

[**102 Processing**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/102)**(**[**WebDAV**](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV)**)**

This code indicates that the server has received and is processing the request, but no response is available yet.

[**103 Early Hints**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/103)

This status code is primarily intended to be used with the [Link](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Link) header, letting the user agent start [preloading](https://developer.mozilla.org/en-US/docs/Web/HTML/Preloading_content) resources while the server prepares a response.

[**Successful responses**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#successful_responses)

[**200 OK**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200)

The request has succeeded. The meaning of the success depends on the HTTP method:

* GET: The resource has been fetched and is transmitted in the message body.
* HEAD: The entity headers are in the message body.
* PUT or POST: The resource describing the result of the action is transmitted in the message body.
* TRACE: The message body contains the request message as received by the server.

[**201 Created**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/201)

The request has succeeded and a new resource has been created as a result. This is typically the response sent after POST requests, or some PUT requests.

[**202 Accepted**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/202)

The request has been received but not yet acted upon. It is noncommittal, since there is no way in HTTP to later send an asynchronous response indicating the outcome of the request. It is intended for cases where another process or server handles the request, or for batch processing.

[**203 Non-Authoritative Information**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/203)

This response code means the returned meta-information is not exactly the same as is available from the origin server, but is collected from a local or a third-party copy. This is mostly used for mirrors or backups of another resource. Except for that specific case, the "200 OK" response is preferred to this status.

[**204 No Content**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204)

There is no content to send for this request, but the headers may be useful. The user-agent may update its cached headers for this resource with the new ones.

[**205 Reset Content**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/205)

Tells the user-agent to reset the document which sent this request.

[**206 Partial Content**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/206)

This response code is used when the [Range](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Range) header is sent from the client to request only part of a resource.

[**207 Multi-Status**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/207)**(**[**WebDAV**](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV)**)**

Conveys information about multiple resources, for situations where multiple status codes might be appropriate.

[**208 Already Reported**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/208)**(**[**WebDAV**](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV)**)**

Used inside a <dav:propstat> response element to avoid repeatedly enumerating the internal members of multiple bindings to the same collection.

[**226 IM Used**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/226)**(**[**HTTP Delta encoding**](https://datatracker.ietf.org/doc/html/rfc3229)**)**

The server has fulfilled a GET request for the resource, and the response is a representation of the result of one or more instance-manipulations applied to the current instance.

## [Redirection messages](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#redirection_messages)

[**300 Multiple Choice**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/300)

The request has more than one possible response. The user-agent or user should choose one of them. (There is no standardized way of choosing one of the responses, but HTML links to the possibilities are recommended so the user can pick.)

[**301 Moved Permanently**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/301)

The URL of the requested resource has been changed permanently. The new URL is given in the response.

[**302 Found**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/302)

This response code means that the URI of requested resource has been changed temporarily. Further changes in the URI might be made in the future. Therefore, this same URI should be used by the client in future requests.

[**303 See Other**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/303)

The server sent this response to direct the client to get the requested resource at another URI with a GET request.

[**304 Not Modified**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/304)

This is used for caching purposes. It tells the client that the response has not been modified, so the client can continue to use the same cached version of the response.

**305 Use Proxy**

Defined in a previous version of the HTTP specification to indicate that a requested response must be accessed by a proxy. It has been deprecated due to security concerns regarding in-band configuration of a proxy.

**306 unused**

This response code is no longer used; it is just reserved. It was used in a previous version of the HTTP/1.1 specification.

[**307 Temporary Redirect**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/307)

The server sends this response to direct the client to get the requested resource at another URI with same method that was used in the prior request. This has the same semantics as the 302 Found HTTP response code, with the exception that the user agent must not change the HTTP method used: If a POST was used in the first request, a POST must be used in the second request.

[**308 Permanent Redirect**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/308)

This means that the resource is now permanently located at another URI, specified by the Location: HTTP Response header. This has the same semantics as the 301 Moved Permanently HTTP response code, with the exception that the user agent must not change the HTTP method used: If a POST was used in the first request, a POST must be used in the second request.

## [Client error responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#client_error_responses)

[**400 Bad Request**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/400)

The server could not understand the request due to invalid syntax.

[**401 Unauthorized**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/401)

Although the HTTP standard specifies "unauthorized", semantically this response means "unauthenticated". That is, the client must authenticate itself to get the requested response.

[**402 Payment Required**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/402)

This response code is reserved for future use. The initial aim for creating this code was using it for digital payment systems, however this status code is used very rarely and no standard convention exists.

[**403 Forbidden**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/403)

The client does not have access rights to the content; that is, it is unauthorized, so the server is refusing to give the requested resource. Unlike 401, the client's identity is known to the server.

[**404 Not Found**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404)

The server can not find the requested resource. In the browser, this means the URL is not recognized. In an API, this can also mean that the endpoint is valid but the resource itself does not exist. Servers may also send this response instead of 403 to hide the existence of a resource from an unauthorized client. This response code is probably the most famous one due to its frequent occurrence on the web.

[**405 Method Not Allowed**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/405)

The request method is known by the server but has been disabled and cannot be used. For example, an API may forbid DELETE-ing a resource. The two mandatory methods, GET and HEAD, must never be disabled and should not return this error code.

[**406 Not Acceptable**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/406)

This response is sent when the web server, after performing [server-driven content negotiation](https://developer.mozilla.org/en-US/docs/Web/HTTP/Content_negotiation#server-driven_negotiation), doesn't find any content that conforms to the criteria given by the user agent.

[**407 Proxy Authentication Required**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/407)

This is similar to 401 but authentication is needed to be done by a proxy.

[**408 Request Timeout**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/408)

This response is sent on an idle connection by some servers, even without any previous request by the client. It means that the server would like to shut down this unused connection. This response is used much more since some browsers, like Chrome, Firefox 27+, or IE9, use HTTP pre-connection mechanisms to speed up surfing. Also note that some servers merely shut down the connection without sending this message.

[**409 Conflict**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/409)

This response is sent when a request conflicts with the current state of the server.

[**410 Gone**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/410)

This response is sent when the requested content has been permanently deleted from server, with no forwarding address. Clients are expected to remove their caches and links to the resource. The HTTP specification intends this status code to be used for "limited-time, promotional services". APIs should not feel compelled to indicate resources that have been deleted with this status code.

[**411 Length Required**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/411)

Server rejected the request because the Content-Length header field is not defined and the server requires it.

[**412 Precondition Failed**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/412)

The client has indicated preconditions in its headers which the server does not meet.

[**413 Payload Too Large**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/413)

Request entity is larger than limits defined by server; the server might close the connection or return an Retry-After header field.

[**414 URI Too Long**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/414)

The URI requested by the client is longer than the server is willing to interpret.

[**415 Unsupported Media Type**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/415)

The media format of the requested data is not supported by the server, so the server is rejecting the request.

[**416 Range Not Satisfiable**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/416)

The range specified by the Range header field in the request can't be fulfilled; it's possible that the range is outside the size of the target URI's data.

[**417 Expectation Failed**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/417)

This response code means the expectation indicated by the Expect request header field can't be met by the server.

[**418 I'm a teapot**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/418)

The server refuses the attempt to brew coffee with a teapot.

[**421 Misdirected Request**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/421)

The request was directed at a server that is not able to produce a response. This can be sent by a server that is not configured to produce responses for the combination of scheme and authority that are included in the request URI.

[**422 Unprocessable Entity**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422)**(**[**WebDAV**](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV)**)**

The request was well-formed but was unable to be followed due to semantic errors.

[**423 Locked**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/423)**(**[**WebDAV**](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV)**)**

The resource that is being accessed is locked.

[**424 Failed Dependency**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/424)**(**[**WebDAV**](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV)**)**

The request failed due to failure of a previous request.

[**425 Too Early**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/425)

Indicates that the server is unwilling to risk processing a request that might be replayed.

[**426 Upgrade Required**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/426)

The server refuses to perform the request using the current protocol but might be willing to do so after the client upgrades to a different protocol. The server sends an [Upgrade](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Upgrade) header in a 426 response to indicate the required protocol(s).

[**428 Precondition Required**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/428)

The origin server requires the request to be conditional. This response is intended to prevent the 'lost update' problem, where a client GETs a resource's state, modifies it, and PUTs it back to the server, when meanwhile a third party has modified the state on the server, leading to a conflict.

[**429 Too Many Requests**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/429)

The user has sent too many requests in a given amount of time ("rate limiting").

[**431 Request Header Fields Too Large**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/431)

The server is unwilling to process the request because its header fields are too large. The request may be resubmitted after reducing the size of the request header fields.

[**451 Unavailable For Legal Reasons**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/451)

The user-agent requested a resource that cannot legally be provided, such as a web page censored by a government.

## [Server error responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#server_error_responses)

[**500 Internal Server Error**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/500)

The server has encountered a situation it doesn't know how to handle.

[**501 Not Implemented**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/501)

The request method is not supported by the server and cannot be handled. The only methods that servers are required to support (and therefore that must not return this code) are GET and HEAD.

[**502 Bad Gateway**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/502)

This error response means that the server, while working as a gateway to get a response needed to handle the request, got an invalid response.

[**503 Service Unavailable**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/503)

The server is not ready to handle the request. Common causes are a server that is down for maintenance or that is overloaded. Note that together with this response, a user-friendly page explaining the problem should be sent. This responses should be used for temporary conditions and the Retry-After: HTTP header should, if possible, contain the estimated time before the recovery of the service. The webmaster must also take care about the caching-related headers that are sent along with this response, as these temporary condition responses should usually not be cached.

[**504 Gateway Timeout**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/504)

This error response is given when the server is acting as a gateway and cannot get a response in time.

[**505 HTTP Version Not Supported**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/505)

The HTTP version used in the request is not supported by the server.

[**506 Variant Also Negotiates**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/506)

The server has an internal configuration error: the chosen variant resource is configured to engage in transparent content negotiation itself, and is therefore not a proper end point in the negotiation process.

[**507 Insufficient Storage**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/507)**(**[**WebDAV**](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV)**)**

The method could not be performed on the resource because the server is unable to store the representation needed to successfully complete the request.

[**508 Loop Detected**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/508)**(**[**WebDAV**](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV)**)**

The server detected an infinite loop while processing the request.

[**510 Not Extended**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/510)

Further extensions to the request are required for the server to fulfill it.

[**511 Network Authentication Required**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/511)

The 511 status code indicates that the client needs to authenticate to gain network access.

* [Zookeeper](https://zookeeper.apache.org/) is a distributed key value store. It can be used as the basis to implement service discovery (similar to etcd).
* [Eureka](https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance) is primarily a service locator used as part of Netflix’s load balancers and failover(allow finding the right service targets for distributing client calls to members of an application cluster).
* [Kubernetes](https://kubernetes.io/) is a container orchestration solution that includes the deployment, discovery and self-healing of services. For a complete list of features, check the link above. The service discovery in kubernetes is based on DNS in the virtual network it spans and builds upon etcd.
* [Consul](https://github.com/hashicorp/consul) (mentioned in the other answer) is a service discovery framework with a REST interface and some additional features (Health Checking, Service Segmentation,..). It has its own internal distributed key value store that can be used as well.

**Junit Annotations**

<https://devqa.io/junit-5-annotations/>  
<https://junit.org/junit5/docs/current/user-guide/#writing-tests-classes-and-methods>

JUnit Jupiter supports the following annotations for configuring tests and extending the framework.  
Unless otherwise stated, all core annotations are located in the [org.junit.jupiter.api](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/package-summary.html) package   
in the junit-jupiter-api module.

| **Annotation** | **Description** |
| --- | --- |
| @Test | Denotes that a method is a test method. Unlike JUnit 4’s @Test annotation, this annotation does not declare any attributes, since test extensions in JUnit Jupiter operate based on their own dedicated annotations. Such methods are inherited unless they are overridden. |
| @ParameterizedTest | Denotes that a method is a [parameterized test](https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests). Such methods are inherited unless they are overridden. |
| @RepeatedTest | Denotes that a method is a test template for a [repeated test](https://junit.org/junit5/docs/current/user-guide/#writing-tests-repeated-tests). Such methods are inherited unless they are overridden. |
| @TestFactory | Denotes that a method is a test factory for [dynamic tests](https://junit.org/junit5/docs/current/user-guide/#writing-tests-dynamic-tests). Such methods are inherited unless they are overridden. |
| @TestTemplate | Denotes that a method is a [template for test cases](https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-templates) designed to be invoked multiple times depending on the number of invocation contexts returned by the registered [providers](https://junit.org/junit5/docs/current/user-guide/#extensions-test-templates). Such methods are inherited unless they are overridden. |
| @TestMethodOrder | Used to configure the [test method execution order](https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-execution-order) for the annotated test class; similar to JUnit 4’s @FixMethodOrder. Such annotations are inherited. |
| @TestInstance | Used to configure the [test instance lifecycle](https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-instance-lifecycle) for the annotated test class. Such annotations are inherited. |
| @DisplayName | Declares a custom [display name](https://junit.org/junit5/docs/current/user-guide/#writing-tests-display-names) for the test class or test method. Such annotations are not inherited. |
| @DisplayNameGeneration | Declares a custom [display name generator](https://junit.org/junit5/docs/current/user-guide/#writing-tests-display-name-generator) for the test class. Such annotations are inherited. |
| @BeforeEach | Denotes that the annotated method should be executed before **each** @Test, @RepeatedTest, @ParameterizedTest, or @TestFactory method in the current class; analogous to JUnit 4’s @Before. Such methods are inherited unless they are overridden. |
| @AfterEach | Denotes that the annotated method should be executed after **each** @Test, @RepeatedTest, @ParameterizedTest, or @TestFactory method in the current class; analogous to JUnit 4’s @After. Such methods are inherited unless they are overridden. |
| @BeforeAll | Denotes that the annotated method should be executed before **all** @Test, @RepeatedTest, @ParameterizedTest, and @TestFactory methods in the current class; analogous to JUnit 4’s @BeforeClass. Such methods are inherited (unless they are hidden or overridden) and must be static (unless the "per-class" [test instance lifecycle](https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-instance-lifecycle) is used). |
| @AfterAll | Denotes that the annotated method should be executed after **all** @Test, @RepeatedTest, @ParameterizedTest, and @TestFactory methods in the current class; analogous to JUnit 4’s @AfterClass. Such methods are inherited (unless they are hidden or overridden) and must be static (unless the "per-class" [test instance lifecycle](https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-instance-lifecycle) is used). |
| @Nested | Denotes that the annotated class is a non-static [nested test class](https://junit.org/junit5/docs/current/user-guide/#writing-tests-nested). @BeforeAll and @AfterAll methods cannot be used directly in a @Nested test class unless the "per-class" [test instance lifecycle](https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-instance-lifecycle) is used. Such annotations are not inherited. |
| @Tag | Used to declare [tags for filtering tests](https://junit.org/junit5/docs/current/user-guide/#writing-tests-tagging-and-filtering), either at the class or method level; analogous to test groups in TestNG or Categories in JUnit 4. Such annotations are inherited at the class level but not at the method level. |
| @Disabled | Used to [disable](https://junit.org/junit5/docs/current/user-guide/#writing-tests-disabling) a test class or test method; analogous to JUnit 4’s @Ignore. Such annotations are not inherited. |
| @Timeout | Used to fail a test, test factory, test template, or lifecycle method if its execution exceeds a given duration. Such annotations are inherited. |
| @ExtendWith | Used to [register extensions declaratively](https://junit.org/junit5/docs/current/user-guide/#extensions-registration-declarative). Such annotations are inherited. |
| @RegisterExtension | Used to [register extensions programmatically](https://junit.org/junit5/docs/current/user-guide/#extensions-registration-programmatic) via fields. Such fields are inherited unless they are shadowed. |
| @TempDir | Used to supply a [temporary directory](https://junit.org/junit5/docs/current/user-guide/#writing-tests-built-in-extensions-TempDirectory) via field injection or parameter injection in a lifecycle method or test method; located in the org.junit.jupiter.api.io package. |

***OAuth2 : JWT Token***   
  
<https://youtu.be/p6MXb0GtXwg>  
<https://www.javainuse.com/spring/boot-jwt>  








***Spring MVC 3.2 Execution Flow***

Step **1**: First request will be received by DispatcherServlet  
Step **2**: DispatcherServlet will take the help of HandlerMapping and get to know the Controller class name associated with the given request  
Step **3**: So request transfer to the Controller, and then controller will process the request by executing appropriate methods and returns ModeAndView object (contains Model data and View name) back to the DispatcherServlet  
Step 4: Now DispatcherServlet send the model object to the ViewResolver to get the actual view page  
Step **5**: Finally DispatcherServlet will pass the Model object to the View page to display the result

**Cross-Origin Resource Sharing (CORS)** is a security concept that allows restricting the resources implemented in web browsers. It prevents the JavaScript code producing or consuming the requests against different origin.

For example, your web application is running on 8080 port and by using JavaScript you are trying to consuming RESTful web services from 9090 port. Under such situations, you will face the Cross-Origin Resource Sharing security issue on your web browsers.

Two requirements are needed to handle this issue −

* RESTful web services should support the Cross-Origin Resource Sharing.
* RESTful web service application should allow accessing the API(s) from the 8080 port.

In this chapter, we are going to learn in detail about how to enable Cross-Origin Requests for a RESTful Web Service application.

***Enable CORS in Controller Method*** We need to set the origins for RESTful web service by using **@CrossOrigin** annotation for the controller method. This @CrossOrigin annotation supports specific REST API, and not for the entire application.

@RequestMapping(value = "/products")

@CrossOrigin(origins = "http://localhost:8080")

public ResponseEntity<Object> getProduct() {

return null;   
}

***Global CORS Configuration*** We need to define the shown @Bean configuration to set the CORS configuration support globally to your Spring Boot application.

@Bean

public WebMvcConfigurer corsConfigurer() {

return new WebMvcConfigurerAdapter() {

@Override

public void addCorsMappings(CorsRegistry registry) {

registry.addMapping("/products").allowedOrigins("http://localhost:9000");

}

};

}

To code to set the CORS configuration globally in main Spring Boot application is given below.

package com.tutorialspoint.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

import org.springframework.web.servlet.config.annotation.CorsRegistry;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@SpringBootApplication

public class DemoApplication {

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

@Bean

public WebMvcConfigurer corsConfigurer() {

return new WebMvcConfigurerAdapter() {

@Override

public void addCorsMappings(CorsRegistry registry) {

registry.addMapping("/products").allowedOrigins("http://localhost:8080");

}

};

}

}

Now, you can create a Spring Boot web application that runs on 8080 port and your RESTful web service application that can run on the 9090 port.  
  
<https://www.tutorialspoint.com/spring_boot/spring_boot_cors_support.htm>  
<https://www.educative.io/blog/getting-started-cors>  
  
what is CORS?

**Cross-origin resource sharing (CORS)** is a browser mechanism that allows a web page to use assets and data from other pages or domains.  
  
Most sites need to use resources and images to run their scripts. These embedded assets present a security risk as the assets could contain viruses or allow server access to a hacker.  
  
Security policies mitigate the security risks of asset use. The policy rules what assets a requesting site can load based on origin or contents and regulates the amount of access given to the requesting site. Each policy must have enough restrictions to secure the web server but not enough to hurt functionality.  
  
Same-origin is the most secure type of policy that prevents access to any outside server. All assets for a site must come from the same origin. Most of the time, same-origin is a good choice as most scripts can function with only local resources. However, sometimes we’ll want to allow access to outside assets such as videos, live-streams, or pictures.  
  
What is an origin?  
***Origin refers to 3 parts: a protocol, a host, and port number***. Protocol refers to the application layer protocol, often HTTP. The host is the main site domain that all pages fall under, like [Educative.io](http://educative.io/). Finally, the port number is the communication endpoint for the request, which defaults to port 80.  
  
Many sites use a form of cross-origin policy called cross-origin resource sharing (CORS) that defines a way for a web page and the host server to interact and determine if it is safe for the server to allow access to the web page.  
  
CORS is a middle ground policy between security and functionality as the server can approve certain outside requests without the insecurity of approving all requests.  
  
Lived Example of CORS   
The most prevalent example of CORS are advertisements on non-native sites.  
  
For example, imagine you’re watching a YouTube video and you see an Android advertisement. YouTube’s servers are reserved for their essential resources and cannot locally store every possible advertisement.  
  
Instead, all ads are stored on the advertisement company’s servers. The advertisement company has allowed viewing access to YouTube to allow a YouTube web page to play the stored Android advertisement video.  
  
The benefit of this system is that YouTube can use content from another server without using local storage. Also, it allows the advertisement company to roll out new advertisements quickly as they only need to update what ad is passed to YouTube from their server.  
  
**Spring JPA dynamic query example**  
<https://javadeveloperzone.com/spring/spring-jpa-dynamic-query-example/>  
 **Steps to Generate Dynamic Query in Spring JPA**

1. Extends JpaSpecificationExecutor in Repository as like
2. Write Specification for the query

**Spring JPA dynamic query examples**

1. JPA Dynamic Criteria with equal
2. JPA dynamic with equal and like
3. JPA dynamic like for multiple fields
4. JPA dynamic Like and between criteria
5. JPA dynamic query with Paging or Pagination
6. JPA Dynamic Order

**Extends JpaSpecificationExecutor in Repository as like:**

@Repository  
 @Transactional  
 **public** **interface** EmployeeDAO **extends** CrudRepository<Employee, **Long**>, JpaSpecificationExecutor<Employee> { }  
 **Write Specification for the query:**

Here we have created List of Predicate, We can add criteria to predicates based on our requirements and using those predicates criteria spring JPA will generate a dynamic query. Here is employeeName will be null then we haven’t added Predicate so it will fetch all the Employees otherwise match to specific names.  
 **public** List<Employee> findByCriteria(String employeeName) {

**return** employeeDAO.findAll(**new** Specification<Employee>() {

@Override

**public** Predicate toPredicate(Root<Employee> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder) {

List<Predicate> predicates = **new** ArrayList<>();

**if** (employeeName != **null**) {

**predicates.add(criteriaBuilder.and(criteriaBuilder.equal(root.get("employeeName"), employeeName)));**

}

**return** criteriaBuilder.and(predicates.toArray(**new** Predicate[predicates.size()]));

}

});

}  
  
**2.1 JPA Dynamic Criteria with equal**

**public** List<Employee> findByCriteria(**String** employeeName){

**return** employeeDAO.findAll(new Specification<Employee>() {

@Override

**public** Predicate toPredicate(Root<Employee> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder) {

List<Predicate> predicates = new ArrayList<>();

**if**(employeeName!=**null**) {

predicates.add(criteriaBuilder.and(criteriaBuilder.equal(root.get("employeeName"), employeeName)));

}

**return** criteriaBuilder.and(predicates.toArray(new Predicate[predicates.size()]));

}

});

}

employeeService.findByCriteria("Harry");                             // Fetch only those whose name is Harry

**SQL Query:** select \* from employee where employeeName=?

employeeService.findByCriteria(null);                               // Fetch all employees

**SQL Query:** select \* from employee

### **2.2 JPA dynamic with equal and like**

**public** List<Employee> findByCriteria(**String** employeeName,**String** employeeRole){

**return** employeeDAO.findAll(new Specification<Employee>() {

@Override

**public** Predicate toPredicate(Root<Employee> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder) {

List<Predicate> predicates = new ArrayList<>();

**if**(employeeName!=**null**) {

predicates.add(criteriaBuilder.and(criteriaBuilder.like(root.get("employeeName"), "%"+employeeName+"%")));

}

**if**(employeeRole!=**null**){

predicates.add(criteriaBuilder.and(criteriaBuilder.equal(root.get("employeeRole"), employeeRole)));

}

**return** criteriaBuilder.and(predicates.toArray(new Predicate[predicates.size()]));

}

});

}

employeeService.findByCriteria("Jo","ADMIN");                // Fetch all names like \*Jo\* and role is ADMIN

**SQL Query:**select \* from employee where (employeeName like ?) and employeeRole=?

employeeService.findByCriteria(null,"ADMIN");                // Fetch employee whose role is ADMIN

**SQL Query:** select \* from employee where employeeRole=?

employeeService.findByCriteria(null,null);                       // Fetch all employees

SQL Query: select \* from employee

**2.3 JPA dynamic like for multiple fields**

**public** List<Employee> findByLikeCriteria(**String** text){

**return** employeeDAO.findAll(new Specification<Employee>() {

@Override

**public** Predicate toPredicate(Root<Employee> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder) {

List<Predicate> predicates = new ArrayList<>();

**if**(text!=**null**) {

predicates.add(criteriaBuilder.or(criteriaBuilder.like(root.get("employeeName"), "%" + text + "%"), criteriaBuilder.like(root.get("employeeEmail"), "%" + text + "%")));

}

**return** criteriaBuilder.and(predicates.toArray(new Predicate[predicates.size()]));

}

});

}

employeeService.findByLikeCriteria("info");                          // Fetch all employees whole email or name contains “info”

**SQL Query:** select \* from employee where employeeName like ? or employeeEmail like ?

employeeService.findByLikeCriteria(null);                              // Fetch all employees

**SQL Query:** select \* from employee

**2.4 JPA dynamic Like and between criteria**

**public** List<Employee> findByLikeAndBetweenCriteria(**String** text, **int** employeeIdStart, **int** employeeIdEnd){

**return** employeeDAO.findAll(new Specification<Employee>() {

@Override

**public** Predicate toPredicate(Root<Employee> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder) {

List<Predicate> predicates = new ArrayList<>();

**if**(text!=**null**) {

predicates.add(criteriaBuilder.or(criteriaBuilder.like(root.get("employeeName"), "%" + text + "%"), criteriaBuilder.like(root.get("employeeEmail"), "%" + text + "%")));

}

**if**(employeeIdStart!=0 && employeeIdEnd!=0){

predicates.add(criteriaBuilder.between(root.get("employeeId"),employeeIdStart,employeeIdEnd));

}

**return** criteriaBuilder.and(predicates.toArray(new Predicate[predicates.size()]));

}

});

}

List<Employee> findByLikeAndBetweenCriteria(String employeeName, int employeeIdStart, int employeeIdEnd);

**SQL Query:** select \* from employee where (employeeName like ? or employeeEmail like ?) and (employeeId between 15 and 20)

employeeService.findByLikeAndBetweenCriteria("info",0,0);         // Fetch all employees whole email or name contains “info”

**SQL Query:** select \*  from employee where employeeName like ? or employeeEmail like ?

employeeService.findByLikeAndBetweenCriteria(null,0,0);               // Fetch all employees

**SQL Query:** select \* from employee  
  
**2.5 JPA dynamic query with Paging or Pagination**

**public** List<Employee> findByPagingCriteria(**String** employeeName,Pageable pageable){

Page page = employeeDAO.findAll(new Specification<Employee>() {

@Override

**public** Predicate toPredicate(Root<Employee> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder) {

List<Predicate> predicates = new ArrayList<>();

**if**(employeeName!=**null**) {

predicates.add(criteriaBuilder.and(criteriaBuilder.equal(root.get("employeeName"), employeeName)));

}

**return** criteriaBuilder.and(predicates.toArray(new Predicate[predicates.size()]));

}

}, pageable);

page.getTotalElements(); // get total elements

page.getTotalPages(); // get total pages

**return** page.getContent(); // get List of Employee

}

employeeService.findByPagingCriteria("Jone", new PageRequest(0,10));   In pageRequest, First Param is page number, Second for page size

**SQL Query:**select \* from employee where employeeName=? limit ?

### **2.6 JPA Dynamic Order**

**public** List<Employee> findByCriteria(**String** employeeName,**String** fieldName){

**return** employeeRepository.findAll(new Specification<Employee>() {

@Override

**public** Predicate toPredicate(Root<Employee> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder) {

List<Predicate> predicates = new ArrayList<>();

**if**(employeeName!=**null**) {

predicates.add(criteriaBuilder.and(criteriaBuilder.equal(root.get("employeeName"), employeeName)));

}

query.orderBy(criteriaBuilder.desc(root.get(fieldName))); // for desc

// query.orderBy(criteriaBuilder.asc(root.get(fieldName))); // for asc

**return** criteriaBuilder.and(predicates.toArray(new Predicate[predicates.size()]));

}

});

}

employeeService.findByCriteria("Harry","joinedDate");

**SQL Query:**select \* from employee where employeeName=? order by joinedDate desc  
  
**Multiple Data Sources with Spring Boot  
  
DataSource Configurations**   
***Spring Boot*** provides a way to configure our project properties using application.properties or .yml file.   
  
spring.jpa.generate-ddl=true

spring.datasource.url = jdbc:mysql://localhost:3306/db1

spring.datasource.username = [username]

spring.datasource.password = [password]

spring.datasource.driverClassName = com.mysql.jdbc.Driver

#second db2 ...

db2.datasource.url = jdbc:mysql://localhost:3306/db2

db2.datasource.username = [username]

db2.datasource.password = [password]

db2.datasource.driverClassName = com.mysql.jdbc.Driver

**JPA Entities**

Let’s define the following 2 JPA entities for our post. ***Product & Customer***

3.1 **Product Entity**

@Entity

public class ProductModel {

@Id

@GeneratedValue (strategy = GenerationType.IDENTITY)

private int id;

@Column (nullable = false)

private String code;

private String name;

private double price;

// *Setter Getter*

}

3.2 **Customer Entity**

@Entity

public class CustomerModel {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Integer id;

@Column(unique = true, nullable = false)

private String email;

private String firstName;

private String lastName;

// *Setter Getter*

}

**Package Structure**

This is important while creating configuration classes. While creating config classes, we need to specify the base packages which will be used by Spring to read different configurations and create multiple Datasource.   
  
src/main/java

- com.javadevjournal

- product

- data

- repo

- config

- customer

- data

- repo

- config  
  
**JPA Repositories**

Let’s create the JPA repositories for our Customer and Product entities. ***Spring Data JPA*** focuses on using JPA to store data in a relational database. Its most compelling feature is the ability to create repository implementations automatically, at runtime, from a repository interface.

5.1 **Product Repository**

@Repository

public interface ProductRepository extends JpaRepository<ProductModel,Integer> {

}

5.2 **Customer Repository**

@Repository

public interface CustomerRepository extends JpaRepository < CustomerModel, Integer > {

}  
  
**Spring Configuration Classes**

In the final step, we will create two Spring configuration classes whose responsibilities are to read the configurations and make sure all necessary setup/classes are available to our project on the application startup. We are creating configuration classes for the Product and Customer with the following details:

**DataSource** *details*

**EntityManagerFactory**

**TransactionManager**

To put it in simple words, we will have these separate configurations for both Customer and Product class.

6.1 **Customer Configuration**

@Configuration

@EnableTransactionManagement

@EnableJpaRepositories(

entityManagerFactoryRef = "customerEntityManagerFactory",

transactionManagerRef = "customerTransactionManager",

basePackages = {

"com.javadevjournal.customer.repo"

}

)

public class CustomerConfig {

@Primary

@Bean(name = "**customerDataSource**")

@ConfigurationProperties(prefix = "spring.datasource")

public DataSource customerDataSource() {

return DataSourceBuilder.create().build();

}

@Primary

@Bean(name = "**customerEntityManagerFactory**")

public LocalContainerEntityManagerFactoryBean entityManagerFactory (

EntityManagerFactoryBuilder builder, @Qualifier("customerDataSource") DataSource dataSource){

return builder

 .dataSource(dataSource)

  .packages("com.javadevjournal.customer.data")

  .persistenceUnit("db1")

  .build();

}

@Primary

@Bean(name = "**customerTransactionManager**")

public PlatformTransactionManager customerTransactionManager(

@Qualifier("customerEntityManagerFactory") EntityManagerFactory customerEntityManagerFactory) {

return new JpaTransactionManager(customerEntityManagerFactory);

}

}

We are setting customerTranscationManager as the primary manager using @Primary annotation. This is important in case we are injecting transaction manager without specifying it. Spring will pick the primary bean in case multiple instances found during injection. 

6.2 **Product Configuration**

@Configuration

@EnableTransactionManagement

@EnableJpaRepositories(

entityManagerFactoryRef = "productEntityManagerFactory",

transactionManagerRef = "productTransactionManager",

basePackages = {

"com.javadevjournal.product.repo"

}

)

public class ProductConfig {

@Bean(name = "**productDataSource**")

@ConfigurationProperties(prefix = "db2.datasource")

public DataSource dataSource() {

return DataSourceBuilder.create().build();

}

@Bean(name = "**productEntityManagerFactory**")

public LocalContainerEntityManagerFactoryBean barEntityManagerFactory(

EntityManagerFactoryBuilder builder, @Qualifier("productDataSource") DataSource dataSource){

return

builder

 .dataSource(dataSource)

 .packages("com.javadevjournal.product.data")

 .persistenceUnit("db2")

 .build();

}

@Bean(name = "**productTransactionManager**")

public PlatformTransactionManager productTransactionManager(

@Qualifier("productEntityManagerFactory") EntityManagerFactory productEntityManagerFactory) {  
 return new JpaTransactionManager(productEntityManagerFactory);

}   
}

**Testing**

@RunWith(SpringRunner.class)

@SpringBootTest

public class MultipleDataSourcesProductTests {

@Autowired

private ProductRepository productRepository;

@Test

@Transactional("productTransactionManager")

public void create\_check\_product() {

ProductModel product = new ProductModel("228781","Running Shoes", 20.0);

product = productRepository.save(product);

assertNotNull(productRepository.findById(product.getId()));

}

}

//Customer test

@RunWith(SpringRunner.class)

@SpringBootTest

public class MultipleDataSourcesCustomerTests {

@Autowired

private CustomerRepository customerRepository;

@Test

@Transactional("customerTransactionManager")

public void create\_check\_customer() {

CustomerModel customer = new CustomerModel("user@www.javadevjournal.com","Robert","Hickle");

customer = customerRepository.save(customer);

assertNotNull(customerRepository.findById(customer.getId()));

assertEquals(customerRepository.findById(customer.getId()).get().getEmail() ,"user@www.javadevjournal.com");

}

}

***Spring @Async rest controller example – Spring @EnableAsync***Spring comes with @EnableAsync annotation and can be applied on application classes for asynchronous behavior. This annotation will look for methods marked with @Async annotation and run in background thread pools. The @Async annotated methods can return CompletableFuture to hold the result of an asynchronous computation.

<https://howtodoinjava.com/spring-boot2/rest/enableasync-async-controller>   
  
**Create async thread pool**

***AsyncConfiguration.java***

@Configuration

@EnableAsync

public class AsynchConfiguration{

@Bean(name = "asyncExecutor")

public Executor asyncExecutor(){

ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();

executor.setCorePoolSize(3);

executor.setMaxPoolSize(3);

executor.setQueueCapacity(100);

executor.setThreadNamePrefix("AsynchThread-");

executor.initialize();

return executor;

}

}

**@Async controller methods**

Methods which shall run asynchronously, annotate them with @Async annotation and method return type should return

@Async("asyncExecutor")

public CompletableFuture<EmployeeNames> methodOne() throws InterruptedException {

//code

}

**Combine async method results**

Inside REST Controller

CompletableFuture.allOf(methodOne, methodTwo, methodThree).join();  
  
**Horizontal Scaling** means scaling by adding more machines to your pool of resources (also described as "scaling out"), whereas **Vertical Scaling** refers to scaling by adding more power (e.g. CPU, RAM) to an existing machine (also described as "scaling up").  
  
  
  
  
  
**API Request contains two different types of data sets and how to handle it?**

**Eg**:   
{

animals:[

{type:'Cat', whiskers-length:'3'},

{type:'Dog', name:'Fancy'}

]

}

You should use the Jackson annotations @**JsonTypeInfo** and @**JsonSubTypes** to achieve **polymorphic json**. The annotations go on the Animal base class.

@**JsonTypeInfo** (use = Id.NAME, include = As.PROPERTY, property = "type")

@**JsonSubTypes** ({

@JsonSubTypes.Type (value = Dog.class, name = "Dog"),

@JsonSubTypes.Type (value = Cat.class, name = "Cat")

})

public abstract class Animal { }  
  
[**https://stackoverflow.com/questions/17247189/spring-requestbody-containing-a-list-of-different-types-but-same-interface**](https://stackoverflow.com/questions/17247189/spring-requestbody-containing-a-list-of-different-types-but-same-interface)

<https://www.linkedin.com/pulse/micro-service-chassis-akshaya-pandey/>

**Micro service: Chassis**any typical application based on micro service architecture has a number of small independent services split based on functionality it offers. However, there are a number of cross cutting concerns each micro service needs to provide which gets repeated across all these services. Various cross cutting concerns encountered in a micro service can include:

1. Logging.
2. Health Check.
3. Service Registry.
4. Security.
5. Monitoring.
6. Metrics collection.

As we create more and more services, it results in code duplication and redundancy. To manage such common concerns, a good idea is to create a skeleton application which provides all the mentioned features and can be used to start adding business functionality to it. Some of the features which are required by few (not all) micro services can be maintained by having a set of independent shared libraries and reusing those across these services.  Frameworks like spring Boot, Spring cloud are classic examples of chassis without us realizing that it is what we are actually doing.

### But what is a Chassis?

**Chassis**can be thought of as a skeleton application. ***In a micro services context, it can be the base framework or even another service which can be reused across different services***. It ensures that the base framework of cross cutting concerns can be reused and we need to focus only the building the business functionality which the micro service needs to provide. Thus, it provides a ready platform for us to start focusing on business functionality and not get bogged down by building the common features across all micro services.



Above diagram depicts a typical micro service development using Chassis pattern. All the Services , service A, service B and Service use the skeleton chassis. Since, only Service A requires message support and DB connectivity, it uses the independent shared libraries developed for that purpose.

### Advantages of a Chassis

1. Decoupling of Business Logic from the cross cutting concerns.
2. Available of a configurable base framework which can be easily configured as per required of micro service.
3. Stable code base which can be reused across other micro services.
4. Supports RAD (Rapid Application Development) by enabling the developers to pull in only the desired libraries in the micro service.
5. New features can be independently added to chassis and needs to be updated only for the services which require these features. Thus, has no/minimal impact on other services

### Design Issues to consider while developing a Chassis:

**Keep the Skeleton Chassis application Lightweight**: There are number of features which are required by all micro services like Logging, Health check which are required by all Microservices. However, there could be some features like **session handling, authentication, messaging support** etc. which might not be required by all services. Thus, it makes sense to abstract out such concerns and have them as separate library instead of including them in skeleton.  In other words keep your skeleton application lightweight.

**Services Required by some micro services** there could be some features like session handling, database connectivity, asynchronous communication which would be required by some micro services (not all). Thus, it makes sense to keep them as a part of separate reusable shared library.

**Chassis should be adequately decoupled** from the business domain of the application. Some exceptions could be some common domain objects which need to be used across your application.  
  
**How to call stored procedures from Spring Data JPA Repositories.**

**Stored Procedure Creation**

**A stored procedure can have parameters** so that we can get different results based on the input. For example, we can create a stored procedure that takes an input parameter of integer type and returns a list of cars:

**AD**

**CREATE** **PROCEDURE** FIND\_CARS\_AFTER\_YEAR(**IN** year\_in **INT**)

**BEGIN**

**SELECT** \* **FROM** car **WHERE** **year** >= year\_in **ORDER** **BY** **year**;

**END**

A stored procedure **can also use output parameters to return data** to the calling applications. For example, we can create a stored procedure that takes an input parameter of string type and stores the query result into an output parameter:

**CREATE** **PROCEDURE** GET\_TOTAL\_CARS\_BY\_MODEL(**IN** model\_in **VARCHAR**(50), **OUT** count\_out **INT**)

**BEGIN**

**SELECT** COUNT(\*) **into** count\_out **from** car **WHERE** model = model\_in;

**END**

5. **Reference Stored Procedures in Repository**

**In Spring Data JPA, repositories are where we provide database operations.** We can construct a repository for the database operations on the Car entity, and reference stored procedures in this repository:

@Repository

**public** **interface** **CarRepository** **extends** **JpaRepository**<Car, Integer> {

// ...

}

Next, let's add some methods to our repository that call stored procedures.

5.1. **Map a Stored Procedure Name Directly**

**We can define a stored procedure method using the @Procedure annotation, and map the stored procedure name directly.**

There are four equivalent ways to do that. For example, we can use the stored procedure name directly as the method name:

@Procedure

**int** **GET\_TOTAL\_CARS\_BY\_MODEL**(String model);

If we want to define a different method name, we can put the stored procedure name as the element of the @Procedure annotation:

@Procedure("GET\_TOTAL\_CARS\_BY\_MODEL")

**int** **getTotalCarsByModel**(String model);

**We can also use the procedureName attribute to map the stored procedure name:**

@Procedure(procedureName = "GET\_TOTAL\_CARS\_BY\_MODEL")

**int** **getTotalCarsByModelProcedureName**(String model);

Finally, we can use the value attribute to map the stored procedure name:

@Procedure(value = "GET\_TOTAL\_CARS\_BY\_MODEL")

**int** **getTotalCarsByModelValue**(String model);

5.2. **Reference a Stored Procedure Defined in Entity**

**We can also use the @NamedStoredProcedureQuery annotation to define a stored procedure in the entity class:**

@Entity

@NamedStoredProcedureQuery (name = "Car.getTotalCardsbyModelEntity",

procedureName = "GET\_TOTAL\_CARS\_BY\_MODEL", parameters = {

@StoredProcedureParameter(mode = ParameterMode.IN, name = "model\_in", type = String.class),

@StoredProcedureParameter(mode = ParameterMode.OUT, name = "count\_out", type = Integer.class)})

**public** **class** **Car** {

// class definition

}

Then we can reference this definition in the repository:

@Procedure(name = "Car.getTotalCardsbyModelEntity")

**int** **getTotalCarsByModelEntiy**(@Param("model\_in") String model);

**We use the name attribute to reference the stored procedure defined in the entity class.** For the repository method, we use @Param to match the input parameter of the stored procedure. We also match the output parameter of the stored procedure to the return value of the repository method.

5.3. **Reference a Stored Procedure with the @Query Annotation**

We can also call a stored procedure directly with the @Query annotation:

@Query(value = "CALL FIND\_CARS\_AFTER\_YEAR(:year\_in);", nativeQuery = true)

List<Car> **findCarsAfterYear**(@Param("year\_in") Integer year\_in);

In this method, we use a native query to call the stored procedure. We store the query in the value attribute of the annotation.   
  
Similarly, we use @Param to match the input parameter of the stored procedure. We also map the stored procedure output to the list of entity Car objects.

<https://www.baeldung.com/spring-data-jpa-stored-procedures>

**@EntityListeners** 🡪 <https://www.baeldung.com/jpa-entity-lifecycle-events>

JPA specifies seven optional lifecycle events that are called:

* before persist is called for a new entity – *@PrePersist*
* after persist is called for a new entity – *@PostPersist*
* before an entity is removed – *@PreRemove*
* after an entity has been deleted – *@PostRemove*
* before the update operation – *@PreUpdate*
* after an entity is updated – *@PostUpdate*
* after an entity has been loaded – *@PostLoad*

@EntityListeners(AuditTrailListener.class)   
@Entity   
**public** **class** **User** { //... }  
  
  
public class AuditTrailListener {

private static Log log = LogFactory.getLog(AuditTrailListener.class);

@PrePersist

@PreUpdate

@PreRemove

private void beforeAnyUpdate(User user) {

if (user.getId() == 0) {

log.info("[USER AUDIT] About to add a user");

} else {

log.info("[USER AUDIT] About to update/delete user: " + user.getId());

}

}

@PostPersist

@PostUpdate

@PostRemove

private void afterAnyUpdate(User user) {

log.info("[USER AUDIT] add/update/delete complete for user: " + user.getId());

}

@PostLoad

private void afterLoad(User user) {

log.info("[USER AUDIT] user loaded from database: " + user.getId());

}

}  
  
<https://howtodoinjava.com/spring-boot2/pagination-sorting-example/>

**PagingAndSortingRepository**PagingAndSortingRepository is an extension of CrudRepository to provide additional methods to retrieve entities using the pagination and sorting abstraction. It provides two methods:

**Page** findAll (Pageable pageable) – returns a Page of entities meeting the paging restriction provided in the Pageable object.

**Iterable** findAll (Sort sort) – returns all entities sorted by the given options. No paging is applied here.

Generally, paging and sorting parameters are optional and thus part of request URL as query parameters. If any API supports paging and sorting, ALWAYS provide default values to its parameters – to be used when client does not choose specify any paging or sorting preferences.

The default paging and sorting values shall be clearly documented in API docs. In UI, these default values can be highlighted with separate colors.

Page number values start with 0. So in UI, if you are displaying page number from 1, then do not forget to subtracting ‘1’ while fetching records.

In below spring mvc controller, we are accepting paging and sorting parameters using pageNo, pageSize and sortBy query parameters. Also, by default '10' employees will be fetched from database in page number '0', and employee records will be sorted based on 'id' field.

<http://localhost:8080/employees?pageSize=5&pageNo=1&sortBy=email>

  
  
  


**Hibernate Vs JPA**

The major difference between Hibernate and JPA is that Hibernate is a framework while JPA is API specifications. Hibernate is the implementation of all the JPA guidelines.

Some key features of JPA which are as follows:

* JPA is only a specification, it is not an implementation.
* It is a set of rules and guidelines to set interfaces for implementing object-relational mapping, .
* It needs a few classes and interfaces.
* It supports simple, cleaner, and assimilated object-relational mapping.
* It supports polymorphism and inheritance.
* Dynamic and named queries can be included in JPA.

The main feature of **Hibernate** is to map the Java classes to database tables. Following are some key features of Hibernate:

* Hibernate is an implementation of JPA guidelines.
* It helps in mapping Java data types to SQL data types.
* It is the contributor of JPA.

https://www.geeksforgeeks.org/java-jpa-vs-hibernate/  
  
Following are the differences between JPA and Hibernate:

| **JPA** | **Hibernate** |
| --- | --- |
| JPA is described in **javax.persistence** package. | Hibernate is described in **org.hibernate** package. |
| It describes the handling of relational data in Java applications. | Hibernate is an Object-Relational Mapping (ORM) tool that is used to save the Java objects in the relational database system. |
| It is not an implementation. It is only a Java specification. | Hibernate is an implementation of JPA. Hence, the common standard which **is** given by JPA is followed by Hibernate. |
| It is a standard API that permits to perform database operations. | It is used in mapping Java data types with SQL data types and database tables. |
| As an object-oriented query language, it uses **Java Persistence Query Language (JPQL)** to execute database operations. | As an object-oriented query language, it uses **Hibernate Query Language (HQL)** to execute database operations. |
| To interconnect with the entity manager factory for the persistence unit, it uses **EntityManagerFactory** interface. Thus, it gives an entity manager. | To create Session instances, it uses **SessionFactory** interface. |
| To make, read, and remove actions for instances of mapped entity classes, it uses **EntityManager** interface. This interface interconnect**s** with the persistence condition. | To make, read, and remove actions for instances of mapped entity classes, it uses **Session**interface. It acts as a runtime interface between a Java application and Hibernate. |

How to convert Response objects to JSON and XML in a Spring Boot REST API  
  
<https://linkedin.com/pulse/converting-objects-json-xml-spring-boot-rest-api-punyakeerthi-bl-xzjvf/>  
<https://www.youtube.com/watch?v=fyc5Tez0CY8&ab_channel=KOBSTechnologies>

1. Add dependency of Jackson data format in pom.xml

|  |
| --- |
| <dependency>  <groupId>com.fasterxml.jackson.dataformat</groupId>  <artifactId>jackson-dataformat-xml</artifactId>  <version>2.12.4</version>  </dependency> |

1. Add in controller produces attribute in Mapping annotation

Eg: @GetMapping(path = **"/customers"**, ***produces = {MediType.APPLICATION\_JSON\_VALUE, MediaType.APPLICATION\_XML\_VALUE}***)

1. In the response object add Annotation @JacksonXmlRootElement(locaName=”**Tag-Name**”)

@JacksonXmlRootElement(localName = "person")

public class Person {

@JacksonXmlProperty

private String name;

@JacksonXmlProperty

private int age;

// Getters and setters

}

1. In the Postman add Accept in headers as application/xml