Spring

The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform. Spring makes it easy to create Java enterprise applications. It provides everything you need to embrace the Java language in an enterprise environment.

Inversion of Control/Dependency Injection

- IoC is also known as dependency injection (DI). It is a process whereby objects define their dependencies (that is, the other objects they work with) only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method.
- The container then injects those dependencies when it creates the bean.
- This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes or a mechanism such as the Service Locator pattern.

The ***org.springframework.beans*** and ***org.springframework.context*** packages are the basis for Spring Framework's IoC container. The ***BeanFactory*** interface provides an advanced configuration mechanism capable of managing any type of object. ***ApplicationContext*** is a sub-interface of ***BeanFactory***. It adds:

- Easier integration with Spring's AOP features
- Message resource handling (for use in internationalization)
- Event publication
- Application-layer specific contexts such as the ***WebApplicationContext*** for use in web applications.

DI exists in two major variants: *Constructor-based* dependency injection and *Setter-based* dependency injection.

- Constructor-based DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency.
- Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or a no-argument static factory method to instantiate your bean.

REST API

REST is an acronym for REpresentational State Transfer and an architectural style for distributed hypermedia systems. REST is not a protocol or a standard, it is an architectural style. The six guiding principles or constraints of the RESTful architecture are:

1. Uniform Interface
2. Client-Server
3. Stateless
4. Cacheable
5. Layered System
6. Code on Demand

1. Uniform Interface

By applying the principle of generality to the components interface, we can simplify the overall system architecture and improve the visibility of interactions. Multiple architectural constraints help in obtaining a uniform interface and guiding the behavior of components.

The following four constraints can achieve a uniform REST interface:

Identification of resources – The interface must uniquely identify each resource involved in the interaction between the client and the server.

Manipulation of resources through representations – The resources should have uniform representations in the server response. API consumers should use these representations to modify the resource state in the server.

Self-descriptive messages – Each resource representation should carry enough information to describe how to process the message. It should also provide information of the additional actions that the client can perform on the resource.

Hypermedia as the engine of application state – The client should have only the initial URI of the application. The client application should dynamically drive all other resources and interactions with the use of hyperlinks.

In simpler words, REST defines a consistent and uniform interface for interactions between clients and servers. For example, the HTTP-based REST APIs make use of the standard HTTP methods (GET, POST, PUT, DELETE, etc.) and the URIs (Uniform Resource Identifiers) to identify resources.

2. **Client-Server**

The client-server design pattern enforces the separation of concerns, which helps the client, and the server components evolve independently.

By separating the user interface concerns (client) from the data storage concerns (server), we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.

While the client and the server evolve, we have to make sure that the interface/contract between the client and the server does not break.

3. **Stateless**

Statelessness mandates that each request from the client to the server must contain all the information necessary to understand and complete the request.

The server cannot take advantage of any previously stored context information on the server.

For this reason, the client application must entirely keep the session state.

4. **Cacheable**

The cacheable constraint requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable.

If the response is cacheable, the client application gets the right to reuse the response data later for equivalent requests and a specified period.

5. **Layered System**

The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior. In a layered system, each component cannot see beyond the immediate layer they are interacting with.

A layman’s example of a layered system is the MVC pattern. The MVC pattern allows for a clear separation of concerns, making it easier to develop, maintain, and scale the application.

6. **Code on Demand (Optional)**

REST also allows client functionality to extend by downloading and executing code in the form of applets or scripts.

The downloaded code simplifies clients by reducing the number of features required to be pre-implemented. Servers can provide part of features delivered to the client in the form of code, and the client only needs to execute the code.

REST(Representational State Transfer) vs SOAP(Simple Object Access Protocol)

REST	SOAP
REST (Representational State Transfer) is a software architecture style aimed at distributed hypermedia systems such as the web. This term refers specifically to a collection of principles for the design of network architectures.	SOAP is a protocol for the exchange of messages over computer networks, generally using HTTP. This protocol is based on XML, making it easier to read, even if messages are longer and therefore considerably slower to transfer. These could be some reasons to use SOAP, instead of REST.

These are some of the advantages of REST:

- It is usually simple to build and adapt.
- Low use of resources.
- Process instances are created explicitly.
- With the initial URI, the client does not require routing information.
- Clients can have a generic ‘listener’ interface for notifications.

Transaction Management

Data access and The interaction between the data access layer and the business or service layer.

The Spring Framework provides a Consistent programming model across different transaction management APIs such as Java Transaction API (JTA), JDBC, Hibernate, and Java Persistence API (JPA).

Hibernate ORM

Hibernate ORM is concerned with helping your application to achieve persistence. So what is persistence? Persistence simply means that

we would like our application’s data to outlive the applications process. In Java terms, we would like the state of (some of) our objects to live beyond the scope of the JVM so that the same state is available later.

CrudRepository	JpaRepository
It is a base interface and extends Repository Interface. It contains methods for CRUD operations. For example save(), saveAll(), findById(), findAll(), etc.	It extends PagingAndSortingRepository that extends CrudRepository. It contains the full API of CrudRepository and PagingAndSortingRepository. For example, it contains flush(), saveAndFlush(), saveAllAndFlush(), deleteInBatch(), etc along with the methods that are available in CrudRepository.
It doesn’t provide methods for implementing pagination and sorting	It provides all the methods for which are useful for implementing pagination.
It works as a marker interface.	It extends both CrudRepository and PagingAndSortingRepository.
To perform CRUD operations, define repository extending CrudRepository. **Syntax:** public interface CrudRepository<T, ID> extends Repository<T, ID>	To perform CRUD as well as batch operations, define repository extends JpaRepository. **Syntax:** public interface JpaRepository<T,ID> extends PagingAndSortingRepository<T,ID>, QueryByExampleExecutor<T>

JPA Queries

JPQL-

```
Collection<User> findAllActiveUsers();
```

Native-

```
@Query( value = "SELECT * FROM USERS u WHERE u.status = 1",
nativeQuery = true) Collection<User>
findAllActiveUsersNative();
```

Aspect-oriented Programming (AOP)

AOP is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. It does this by adding additional behaviour to existing code without modifying the code itself.

Aspect

An aspect is a modularization of a concern that cuts across multiple classes. Unified logging can be an example of such cross-cutting concern.

Joinpoint

A Joinpoint is a point during the execution of a program, such as the execution of a method or the handling of an exception.

Pointcut

A Pointcut is a predicate that helps match an Advice to be applied by an Aspect at a particular JoinPoint.

We often associate the Advice with a Pointcut expression, and it runs at any Joinpoint matched by the Pointcut.

Advice

An Advice is an action taken by an aspect at a particular Joinpoint. Different types of advice include “around,” “before,” and “after.”

In Spring, an Advice is modelled as an interceptor, maintaining a chain of interceptors around the Joinpoint.

Spring Boot Caching

Cache is a part of temporary memory (RAM). It lies between the application and the persistent database.

Caching is a mechanism used to increase the performance of a system. It is a process to store and access data from the cache.

@EnableCaching

It is a class level annotation. It is used to enable caching in spring boot application.

@Cacheable

It is a method level annotation. It is used in the method whose response is to be cached. The Spring boot manages the request and response of the method to the cache that is specified in the annotation attribute.

1. cacheNames / value :

The cacheNames is used to specify the name of the cache while value specifies the alias for the cacheNames.

2. key :

This is the key with which object will be cached. It uniquely identifies each entry in the cache. If we do not specify the key then Spring uses the default mechanism to create the key.

@CachePut

It is a method level annotation. It is used to update the cache before invoking the method.

@CacheEvict

It is a method level annotation. It is used to remove the data from the cache. When the method is annotated with this annotation then the method is executed and the cache will be removed / evicted.

The Twelve Factors

1. **Codebase**

One codebase tracked in revision control, many deploys

2. **Dependencies**

Explicitly declare and isolate dependencies

3. **Config**

Store config 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 as one or more stateless processes

7. **Port binding**

Export services via port binding

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/management tasks as one-off processes

Microservices

- Microservices are self-contained, independent deployment module.
- The cost of scaling is comparatively less than the monolithic architecture.
- Microservices are independently manageable services. It can enable more and more services as the need arises. It minimizes the impact on existing service.
- It is possible to change or upgrade each service individually rather than upgrading in the entire application.
- Microservices allows us to develop an application which is organic (an application which latterly upgrades by adding more functions or modules) in nature.
- It enables event streaming technology to enable easy integration in comparison to heavyweight interposes communication.
- Microservices follows the single responsibility principle.
- The demanding service can be deployed on multiple servers to enhance performance.
- Less dependency and easy to test.
- Dynamic scaling.
- Faster release cycle.

Microservices Design Patterns

The principles used to design Microservices are as follows:

1. Independent & Autonomous Services
2. Scalability
3. Decentralization
4. Resilient Services
5. Real-Time Load Balancing
6. Availability
7. Continuous delivery through DevOps Integration
8. Seamless API Integration and Continuous Monitoring
9. Isolation from Failures
10. Auto-Provisioning

API Gateway:

An API Gateway helps to address many concerns raised by microservice implementation, not limited to the ones above.

1. An API Gateway is the single point of entry for any microservice call.

2. It can work as a proxy service to route a request to the concerned microservice, abstracting the producer details.

3. It can fan out a request to multiple services and aggregate the results to send back to the consumer.

4. One-size-fits-all APIs cannot solve all the consumer's requirements; this solution can create a fine-grained API for each specific type of client.

5. It can also convert the protocol request (e.g. AMQP) to another protocol (e.g. HTTP) and vice versa so that the producer and consumer can handle it.

6. It can also offload the authentication/authorization responsibility of the microservice.

Service Discovery Pattern:

1. A service registry needs to be created which will keep the metadata of each producer service.
2. A service instance should register to the registry when starting and should de-register when shutting down.
3. The consumer or router should query the registry and find out the location of the service. The registry also needs to do a health check of the producer service to ensure that only

working instances of the services are available to be consumed through it.
4. There are two types of service discovery: client-side and server-side. An example of client-side discovery is Netflix Eureka and an example of server-side discovery is AWS ALB.

Circuit Breaker Pattern:

1. The consumer should invoke a remote service via a proxy that behaves in a similar fashion to an electrical circuit breaker.
2. When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period, all attempts to invoke the remote service will fail immediately.
3. After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed, the circuit breaker resumes normal operation. Otherwise, if there is a failure, the timeout period begins again.
4. Netflix Hystrix is a good implementation of the circuit breaker pattern. It also helps you to define a fallback mechanism which can be used when the circuit breaker trips. That provides a better user experience.

Controller Vs RestController

@Controller	@RestController
@Controller is used to mark classes as Spring MVC Controller.	@RestController annotation is a special controller used in RESTful Web services, and it's the combination of @Controller and @ResponseBody annotation.
It is a specialized version of @Component annotation.	It is a specialized version of @Controller annotation.
In @Controller, we can return a view in Spring Web MVC.	In @RestController, we can not return a view.

@Controller	@RestController
@Controller annotation indicates that the class is a “controller” like a web controller.	@RestController annotation indicates that class is a controller where @RequestMapping methods assume @ResponseBody semantics by default.
In @Controller, we need to use @ResponseBody on every handler method.	In @RestController, we don’t need to use @ResponseBody on every handler method.
It was added to Spring 2.5 version.	It was added to Spring 4.0 version.

Bean Life Cycle

- The IoC container instantiates the bean from the bean’s definition in the XML file. Spring then populates all of the properties using the dependency injection as specified in the bean definition.
- The bean factory container calls setBeanName() which take the bean ID and the corresponding bean has to implement BeanNameAware interface.
- The factory then calls setBeanFactory() by passing an instance of itself (if BeanFactoryAware interface is implemented in the bean).
- If BeanPostProcessors is associated with a bean, then the preProcessBeforeInitialization() methods are invoked.
- If an init-method is specified, then it will be called.
- Lastly, postProcessAfterInitialization() methods will be called if there are any BeanPostProcessors associated with the bean that needs to be run post creation.

ResponseEntity<>

A Generic class, which bundles the HTTP response sent back to Client. With

ResponseEntity you can specify

a. Http status code

b. Http Headers

c. Response Body

Spring Actuator

An actuator is an additional feature of spring that helps you to monitor and manage your application when you push it to production. These actuators include auditing, health, CPU usage, HTTP hits, and metric gathering, and many more that are automatically applied to your application.

Profiles

While developing the application we deal with multiple environments such as dev, QA, Prod, and each environment requires a different configuration.

For e.g., we might be using an embedded H2 database for dev but for prod, we might have proprietary Oracle or DB2. Even if DBMS is the same across the environment, the URLs will be different.

To make this easy and clean, spring has the provision of Profiles to keep the separate configuration of environments.

Stereotype Annotations

Spring Framework provides us with some special annotations. These annotations are used to create Spring beans automatically in the application context. @Component annotation is the main Stereotype Annotation. There are some Stereotype meta-annotations which is derived from @Component those are,

@Service

@Repository

@Controller

1: **@Service**: We specify a class with @Service to indicate that they’re holding the business logic. Besides being used in the service layer, there isn’t any other special use for this annotation. The utility classes can be marked as Service classes.

2: **@Repository**: We specify a class with @Repository to indicate that they’re dealing with CRUD operations, usually, it’s used with DAO (Data Access Object) or Repository implementations that deal with database tables.

3: **@Controller**: We specify a class with @Controller to indicate that they’re front controllers and responsible to handle user requests and return the appropriate response. It is mostly used with REST Web Services.

Annotations

@Bean – method annotated with @Bean creates and returns Bean. Spring Container calls such methods, automatically.

@PostConstruct & **@PreDestroy** – indicates Bean life cycle methods

@Configuration – Class annotated with @Configuration has methods annotated with @Bean or has data members annotated with @Value

@Scope– indicates Scope of a Bean such as Singleton, Prototype, Session, etc…

@Lazy – indicates that Bean needs to be created on Demand only, i.e. when there is explicit request.

@Autowired – indicates Bean needs to be automatically created by Spring Container. 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.

```
class EmployeeController{

@Autowired

EmployeeService eService;

//Remaining code which directly uses eService, as it's automatically created using Autowired

}
```

@Qualifier – used along with @Bean or @Autowired to avoid ambiguity during Bean creation by Spring Container

@Primary – When there are multiple qualified Beans, priority is given to the Bean annotated with @Primary

@Component – indicates a class as Component, so that it can be recognized by @ComponentScan, automatically. As known, all Component classes are automatically scanned and loaded by Spring Container.

@ComponentScan – scans one or more packages/subpackages for Components.This annotation is part of @SpringBootApplication, however it may be used separately as well.

@Service – Components in Service Layer need to be annotated with @Service.It is also used at class level. It tells the spring that class contains the business logic.

```
@Service
public class EmployeeService{
//define methods exposed by the Service Layer
}
```

@Repository – Components in Repository Layer need to be annotated with @Repository. The repository is a DAOs (Data Access Object) that access the database directly. The repository does all the operations related to the database.

@SpringBootApplication – This annotation is used with main class of Spring Boot Application.@SpringBootApplication is combination of @ComponentScan, @EnableAutoConfiguration and @Configuration

@Value – Data members of a Configuration class are automatically loaded from Configuration file (such as application.properties) or initialized to a specific value,as shown below.

```
@Service
public class EmployeeService{
//cname property is automatically retrieved from application.properties file
@Value("${cname}")
private String company_name;
//initialize tax_id with ABCDEF123
@Value("ABCDEF123")
private String tax_id;
//methods which uses above fields...
}
```

@ConfigurationProperties – Class annotated with @ConfigurationProperties automatically loads bunch of data members (with matching property names)from Configuration file(such as application.properties), as shown below

```
@Component
@ConfigurationProperties
public class CompanyDetails{
private String company_name;
private String company_ceo;
private String head_office_city;
```

```
//methods defined by this class...

}
```

```
#contents of application.properties file

company_name = WXYZ Company

company_ceo = Some One

head_office_city = Bangalore

#other configuration properties can be specified here
```

@PropertySource – Class Level Annotation, generally along with @Configuration annotation. @PropertySource lets developers to specify custom Property filename(s)(other than application.properties), from which Configuration properties can be loaded, in runtime, as shown below.

```
@Configuration

@PropertySource("classpath:myfilename.properties")

class SomeConfigurations{

//Spring Boot automatically looks for myfile.properties file, for required Configuration properties

}
```

@Profile – can be used with Configuration or Component classes, to indicate this specific class is available, when application is running in specific profile mode, such as dev, test or production.

REST API related Annotations:

@RestController – Class annotated with @RestController exposes REST End points, as shown below

```
@RestController

class EmployeeController{

@GetMapping("/allemployees")

List<Employee> getEmployees(){

//return List of Employees

}

//More REST end points can be exposed

}
```

@RequestBody – used with method parameter of REST end point. This annotation automatically deserializes the body(of Http request) into a Model or Entity object.

```
@RestController
```

```
class EmployeeController{
@PostMapping("/create")
ResponseEntity<Employee> createEmployee(@RequestBody Employee emp){
//code to save emp object in DB
}
//other REST end points
}
```

@PathVariable – used with method parameter of REST end point. It automatically retrieves a Path variable into the method parameter of REST endpoint.

```
@RestController
class EmployeeController{
@GetMapping("/employee/{eid}")
Employee getEmployee(@PathVariable("eid") Integer empid){
//code to fetch Employee from DB
}
//other REST end points
}
```

@RequestParam – used with method parameter of REST end point. It automatically retrieves a Query parameter into the method parameter of REST end point.

```
@RestController
class EmployeeController{
@GetMapping(“/emp”)
Employee getEmployee(@RequestParam Integer empid){
//code to fetch Employee from DB
}
//other REST end points
}
```

@RequestHeader – used with method parameter of REST end point. It automatically retrieved value from a specified HTTP header and populates the value into the method parameter. REST End points are annotated with any of below annotation, to indicate specific HTTP method

1. @RequestMapping
2. @GetMapping – to retrieve one or more resource(such as Employee)

details

3. @PostMapping – to create a new resource

4. @PutMapping – to update an existing resource

5. @DeleteMapping – to delete an existing resource

```
@RestController
class EmployeeController{
@DeleteMapping("/emp")
Employee removeEmployee(@RequestParam Integer empid){
//code to delete Employee from DB
}
//other REST end points
}
```

REST API Exception Handling annotations

@ExceptionHandler –method annotated with this annotation, is automatically called whenever a specific Exception (either inbuilt or custom) occurs. This method returns appropriate Exception details to the Client.

@ControllerAdvice – class annotated with this annotation, has methods annotated with @ExceptionHandler

@Valid – used with @RequestBody, to automatically validate the data members during deserialization. This annotation works along with Validation rules such as

@NotNull, @Max, etc… used with fields of Entity class

```
@RestController
class EmployeeController{
@PostMapping("/emp")
Employee createEmployee(@Valid @RequestParam Employee emp){
//code to create new Employee in DB
}
//other REST end points
}
```

Spring Boot Data JPA related annotations

@Entity – class which need to be mapped with underlying DB Table

@Table – Used along with @Entity annotated, to specify custom name for DB

Table (by default DB Table has same name as Entity Class name)

@Column – Used with Data members of Entity class, to indicate a Column of DB Table.

Data field Validation related – @NotNull, @Max, @Min, @Positive, @Negative,etc…

@Query – to specify Custom Query String(native or JPQL query), along with method declaration in Repository interface.

Entity class relationships – @OnetoOne, @OnetoMany, @ManytoOne,

@ManytoMany

Security related Annotations

@CrossOrigin – Can be used with Class or method(s), indicating by which Origins(domain name or domain name patterns) the REST end points can be invoked.

Below annotations used for method level Security

1. @Secured
2. @PreAuthorize
3. @PermitAll

AOP related Annotations: Aspect Oriented Programming is used to separate Cross Cutting concerns (such as Logging, Security, etc…), from Business Logic. AOP is used only in selected Spring Boot Projects.

@Aspect – to specify that a class is Aspect, which holds Cross cutting concerns

@Pointcut – to specify Pointcut expressions

@Before – to specify a method is Before Advice

@After – to specify a method is After Advice

@Around – to specify a method is around Advice

As known, all advice methods are in an Aspect class.

Caching related Annotations

@EnableCaching – Used along with @SpringBootApplication, which enables the application to perform Cache related operations

@Cacheable – Adds an entry to the Cache

@CachePut – Updates an existing entry in the Cache

@CacheEvict – Removes one or more entries from the Cache

Scheduling related Annotations: It's quite common that an Enterprise application may need some functionality to be executed periodically. For such requirements, below Scheduling related Annotations are used

@EnableScheduling – Enables the Application to use Scheduler. This annotation is used along with @SpringBootApplication.

@Scheduled – this annotation is used with a method, which needs to be automatically executed periodically at specific points of time.

Transaction related Annotations:

@Transactional – Used by class/interface or method, indicates the method(s) is executed under a Transaction.