

Spring Boot Interview Questions and Answers

1. What is Spring Boot?

- not a framework
- ease to create stand-alone application with minimal or zero configurations.
- It is approach to develop spring based application with very less configuration.
- develop production ready applications.
- It provides a set of Starter Pom's or gradle build files which one can use to add required dependencies and also facilitate auto configuration.
- Spring Boot automatically configures required classes depending on the libraries on its classpath.
- Suppose your application want to interact with DB, if there are Spring Data libraries on class path then it automatically sets up connection to DB along with the Data Source class.

2. What are the advantages of using Spring Boot?

- It is very easy to develop Spring Based applications with Java or Groovy.
- It reduces lots of development time and increases productivity.
- It avoids writing lots of boilerplate Code, Annotations and XML Configuration.
- It is very easy to integrate Spring Boot Application with its Spring Ecosystem like Spring JDBC, Spring ORM, Spring Data, Spring Security etc.
- It follows “Opinionated Defaults Configuration” Approach to reduce Developer effort
- It provides Embedded HTTP servers like Tomcat, Jetty etc. to develop and test our web applications very easily.

- It provides CLI (Command Line Interface) tool to develop and test Spring Boot (Java or Groovy) Applications from command prompt very easily and quickly.
- It provides lots of plugins to develop and test Spring Boot Applications very easily using Build Tools like Maven and Gradle
- It provides lots of plugins to work with embedded and in-memory Databases very easily.

3. What are the disadvantages of using Spring Boot?

- It is very tough and time consuming process to convert existing or legacy Spring Framework projects into Spring Boot Applications. It is applicable only for brand new/Greenfield Spring Projects.

4. Why is it “opinionated”?

- It follows “Opinionated Defaults Configuration” Approach to reduce Developer effort. Due to opinionated view of spring boot, what is required to get started but also we can get out if not suitable for application.
 - Spring Boot uses sensible defaults, “opinions”, mostly based on the classpath contents.
 - For example
 - Sets up a JPA Entity Manager Factory if a JPA implementation is on the classpath.
 - Creates a default Spring MVC setup, if Spring MVC is on the classpath.
 - Everything can be overridden easily
 - But most of the time not needed

5. How does it work? How does it know what to configure?

- Auto-configuration works by analyzing the classpath
 - If you forget a dependency, Spring Boot can't configure it
 - A dependency management tool is recommended
 - Spring Boot parent and starters make it much easier
- Spring Boot works with Maven, Gradle, Ant/Ivy
 - Our content here will show Maven

6. How are properties defined? Where?

In spring boot, we have to define properties in the application.properties file exists in classpath of application as follow.

Example: configure default DataSource bean

database.host=localhost

database.user=admin

7. What is the difference between an embedded container and a WAR?

There is no force to go container less

- Embedded container is just one feature of Spring Boot
- Traditional WAR also benefits a lot from Spring Boot
- Automatic Spring MVC setup, including DispatcherServlet
- Sensible defaults based on the classpath content
- Embedded container can be used during development

8. What embedded containers does Spring Boot support?

Spring Boot includes support for embedded Tomcat, Jetty, and Undertow servers.

By default the embedded server will listen for HTTP requests on port 8080.

9. What does `@EnableAutoConfiguration` do? What about `@SpringBootApplication`?

`@EnableAutoConfiguration` annotation on a Spring Java configuration class

- Causes Spring Boot to automatically create beans it thinks you need
- Usually based on classpath contents, can easily override

`@Configuration`

`@EnableAutoConfiguration`

```
public class MyAppConfig {  
    public static void main(String[] args) {  
        SpringApplication.run(MyAppConfig.class, args);  
    }  
}
```

`@SpringBootApplication` was available from Spring Boot 1.2

It is very common to use `@EnableAutoConfiguration`, `@Configuration`, and `@ComponentScan` together.

```
@Configuration
@ComponentScan
@EnableAutoConfiguration
public class MyAppConfig {
...
}
```

With @SpringBootApplication annotation

```
@SpringBootApplication
public class MyAppConfig {
...
}
```

10. What is a Spring Boot starter POM? Why is it useful?

Starters are a set of convenient dependency descriptors that you can include in your application. The starters contain a lot of the dependencies that you need to get a project up and running quickly and with a consistent, supported set of managed transitive dependencies.

The starter POMs are convenient dependency descriptors that can be added to your application's Maven. In simple words, if you are developing a project that uses Spring Batch for batch processing, you just have to include `spring-boot-starter-batch` that will import all the required dependencies for the Spring Batch application. This reduces the burden of searching and configuring all the dependencies required for a framework.

11. Spring Boot supports both Java properties and YML files. Would you recognize and understand them if you saw them?

spring boot application java property file name is
`application.properties`

spring boot application YML file name is `application.yml`

12. Can you control logging with Spring Boot? How?

Yes, we can control logging with spring boot.

Customizing default Configuration for Logging:

By adding logback.xml file to the application we can override the default logging configuration providing by the Spring Boot. This file place in the classpath (src/main/resources) of the application for Spring Boot to pick the custom configuration.

Spring Boot can control the logging level

- Just set it in application.properties
- Works with most logging frameworks
- Java Util Logging, Logback, Log4J, Log4J2

logging.level.org.springframework=DEBUG

logging.level.com.acme.your.code=INFO

13. How to reload my changes on Spring Boot without having to restart server?

Include following maven dependency in the application.

```
<dependency>  
<groupId>org.springframework</groupId>  
<artifactId>springloaded</artifactId>  
<version>1.2.6.RELEASE</version>  
</dependency>
```

Automatic restart

Applications that use spring-boot-devtools will automatically restart whenever files on the classpath change. This can be a useful feature when working in an IDE as it gives a very fast feedback loop for code changes. By default, any entry on the classpath that points to a folder will be monitored for changes.

```
<dependency>  
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-devtools</artifactId>
```

```
<optional>true</optional>  
</dependency>
```

This can be achieved using DEV Tools. With this dependency any changes you save, the embedded tomcat will restart. Spring Boot has a Developer tools (DevTools) module which helps to improve the productivity of developers. One of the key challenge for the Java developers is to auto deploy the file changes to server and auto restart the server. Developers can reload changes on Spring Boot without having to restart my server. This will eliminates the need for manually deploying the changes every time. Spring Boot doesn't have this feature when it has released it's first version. This was a most requested features for the developers. The module DevTools does exactly what is needed for the developers. This module will be disabled in the production environment.

14. What is Actuator in Spring Boot?

Spring Boot Actuator is a sub-project of Spring Boot. It adds several production grade services to your application with little effort on your part. There are also many features added to your application out-of-the-box for managing the service in a production (or other) environment. They're mainly used to expose different types of information about the running application – health, metrics, info, dump, env etc.

15. How to run Spring boot application to custom port ?

In application.properties, add following property.
server.port = 8181

16. How to implement security for Spring boot application ?

Add spring security starter to the boot application

<dependency>

```
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-security</artifactId>  
</dependency>
```

17. What is the configuration file name used by Spring Boot?

The configuration file used in spring boot projects is application.properties. This file is very important where we would over write all the default configurations. Normally we have to keep this file under the resources folder of the project.

18. How to implement Spring web using Spring boot?

Web Application Convenience

- Boot automatically configures
 - A DispatcherServlet & ContextLoaderListener
 - Spring MVC using same defaults as @EnableWebMvc
- Plus many useful extra features:
 - Static resources served from classpath
- /static, /public, /resources or /META-INF/resources

- Templates served from /templates
- If Velocity, Freemarker, Thymeleaf, or Groovy on classpath
- Provides default /error mapping
- Easily overridden
- Default MessageSource for I18N

19. How to configure datasource using Spring boot?

- Use either spring-boot-starter-jdbc or spring-boot-starter-data-jpa and include a JDBC driver on classpath
- Declare properties

```
spring.datasource.url=jdbc:mysql://localhost/test
```

```
spring.datasource.username=dbuser
```

```
spring.datasource.password=dbpass
```

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

- Spring Boot will create a DataSource with properties set

- Will even use a connection pool if the library is found on the classpath!

20. What is YAML?

Yaml Ain't a Markup Language

- Recursive acronym
 - Created in 2001
 - Alternative to .properties files
- Allows hierarchical configuration
 - Java parser for YAML is called SnakeYAML
- Must be in the classpath
- Provided by spring-boot-starters

YAML for Properties

- Spring Boot support YAML for Properties
- An alternative to properties files
application.properties

database.host = localhost

database.user = admin

application.yml

database:

host: localhost

user: admin

- YAML is convenient for hierarchical configuration data
 - Spring Boot properties are organized in groups
 - Examples: server, database, etc

1.1 What is Microservices Architecture?

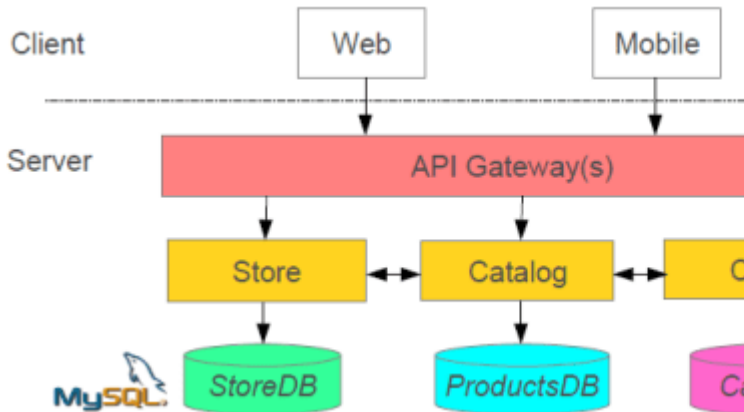
Microservices architecture allows to ***avoid monolith application*** for large system.

It provide loose coupling between collaborating processes which running independently in different environments with tight cohesion.

So lets discuss it by an example as below.

For example imagine an online shop with separate microservices for user-accounts, product-catalog order-processing and shopping carts. So these components are inevitably important for such a large online

shopping portal. For online shopping system we could use following architectures.



Microservices Architecture

Spring enables separation-of-concerns

Loose Coupling– Effect of changes isolated

Tight Cohesion– Code perform a single well defined task

Microservices provide the same strength as Spring provide

Loose Coupling– Application build from collaboration services or processes, so any process change without effecting another processes.

Tight Cohesion-An individual service or process that deals with a single view of data.

Microservices Benefits

- Smaller code base is easy to maintain.
- Easy to scale as individual component.
- Technology diversity i.e. we can mix libraries, databases, frameworks etc.
- Fault isolation i.e. a process failure should not bring whole system down.
- Better support for smaller and parallel team.
- Independent deployment
- Deployment time reduce

Micro services Challenges

- Difficult to achieve strong consistency across services
- ACID transactions do not span multiple processes.
- Distributed System so hard to debug and trace the issues
- Greater need for end to end testing
- Required cultural changes in across teams like Dev and Ops working together even in same team.

Adding Spring Cloud and Discovery server

What is Spring Cloud?

1. It is building blocks for Cloud and Microservices
2. It provides microservices infrastructure like provide use services such as Service Discovery, Configuration server and Monitoring.
3. It provides several other open source projects like Netflix OSS.

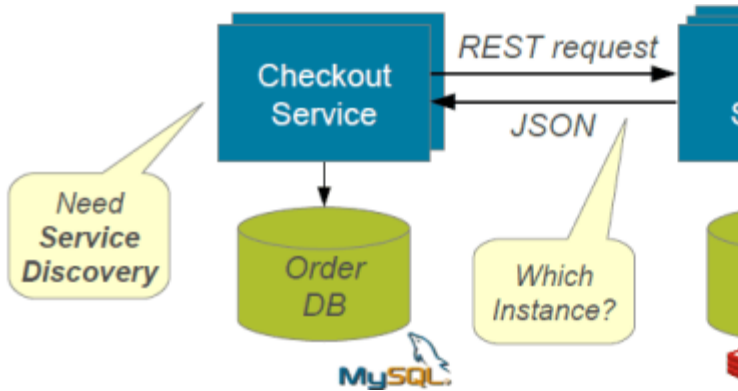
4. It provides PaaS like Cloud Foundry, AWS and Heroku.
5. It uses Spring Boot style starters

There are many use-cases supported by Spring Cloud like Cloud Integration, Dynamic Reconfiguration, Service Discovery, Security, Client side Load Balancing etc. But in this post we concentrate on following microservices support

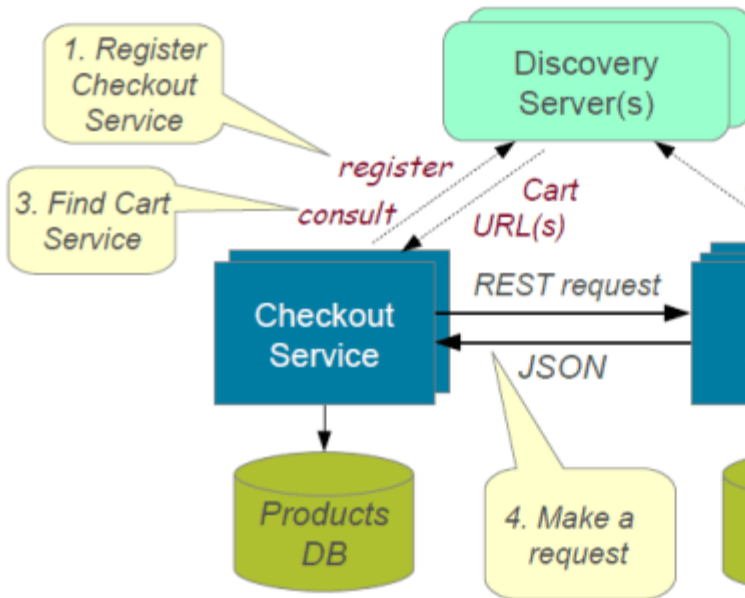
Problem without discovery

Service Discovery (How do services find each other?)

Client-side Load Balancing (How do we decide which service instance to use?)



Resolution with service discovery



Implementing Service Discovery

Spring Cloud support several ways to implement service discovery but for this I am going to use Eureka created

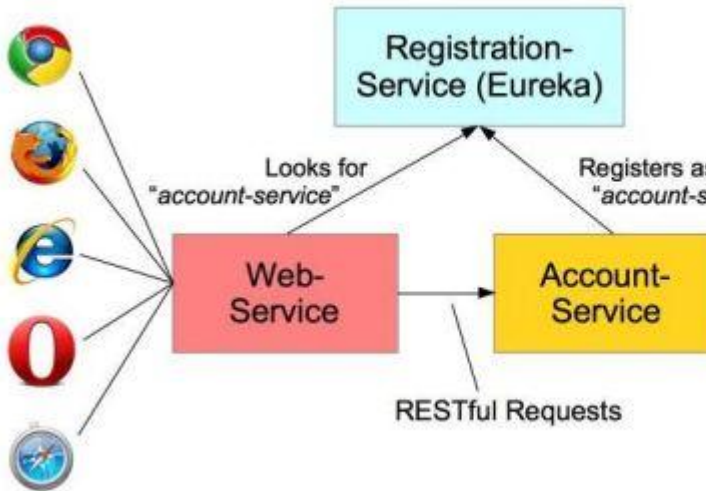
by Netflix. Spring Cloud provide several annotation to make it use easy and hiding lots of complexity.

Client-side Load Balancing

Each service typically deployed as multiple instances for fault tolerance and load sharing. But there is problem how to decide which instance to use?

Implementing Client-Side Load Balancing

We will use Netflix Ribbon, it provide several algorithm for Client-Side Load Balancing. Spring provide smart RestTemplate for service discovery and load balancing by using **@LoadBalanced** annotation with RestTemplate instance.



```
<!-- Eureka registration server -->
```

```
<dependency>
```

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

    <artifactId>spring-cloud-starter-
eureka-server</artifactId>

</dependency>
```

DiscoveryMicroserviceServerApplication.java

```
package com.doj.discovery;
```

```
import
org.springframework.boot.SpringApplication;
```

```
import
org.springframework.boot.autoconfigure.SpringBootApplication;

import
org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication

@EnableEurekaServer
```

```
public class
DiscoveryMicroserviceServerApplication {

    public static void main(String[] args) {
```

```
SpringApplication.run(DiscoveryMicroserviceSer  
verApplication.class, args);  
  
}  
  
}
```

application.yml

```
# Configure this Discovery Server  
  
eureka:  
  
  instance:  
  
    hostname: localhost  
  
  client: #Not a client  
  
    registerWithEureka: false
```

```
    fetchRegistry: false

# HTTP (Tomcat) port

server:

    port: 1111
```

in pom.xml we need these dependency

```
<dependency>

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

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

```
</dependency>
```

```
<dependency>
```

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

```
    <artifactId>spring-cloud-starter-  
eureka</artifactId>
```

```
</dependency>
```

```
<dependency>
```

Creating Account Producer MicroService

Microservice declares itself as an available service and register to Discovery Server created in Step 1.

Using ***@EnableDiscoveryClient***

Registers using its application name

@EnableDiscoveryClient annotation also allows us to query Discovery server to find microservices

Load Balanced *RestTemplate*

Create using @LoadBalanced– Spring enhances it to service lookup & load balancing

```
@Bean

@LoadBalanced

public RestTemplate restTemplate() {

    return new RestTemplate();

}
```

Core Spring annotations

1. @Required : This annotation is applied on bean setter methods. Consider a scenario where you need to enforce a required property. The @Required annotation indicates that the affected bean must be populated at configuration time with the required property. Otherwise an exception of type `BeanInitializationException` is thrown.

2. @Autowired

This annotation is applied on fields, setter methods, and constructors. The @Autowired annotation injects object dependency implicitly.

When you use @Autowired on fields and pass the values for the fields using the property name, Spring will automatically assign the fields with the passed values.

You can even use `@Autowired` on private properties, as shown below. (This is a very poor practice though!)

```
public class Customer {  
    @Autowired  
    private Person person;  
    private int type;  
}
```

When you use `@Autowired` on setter methods, Spring tries to perform the by Type autowiring on the method. You are instructing Spring that it should initiate this property using setter method where you can add your custom code, like initializing any other property with this property.

```
public class Customer {  
    private Person person;  
    @Autowired
```

```
    public void setPerson (Person person) {  
        this.person=person;  
    }  
}  
public class Customer {  
    private Person person;  
    @Autowired  
    public void setPerson (Person person) {  
        this.person=person;  
    }  
}
```

Consider a scenario where you need instance of class A, but you do not store A in the field of the class. You just use A to obtain instance of B, and you are storing B in this field. In this case setter method autowiring will better suite you. You will not have class level unused fields.

When you ***use @Autowired on a constructor, constructor injection happens at the time of object***

creation. It indicates the constructor to autowire when used as a bean. One thing to note here is that only one constructor of any bean class can carry the @Autowired annotation.

@Component

```
public class Customer {  
    private Person person;  
    @Autowired  
    public Customer (Person person) {  
  
        this.person=person;  
    }  
}
```

@Component

```
public class Customer {  
    private Person person;  
    @Autowired  
    public Customer (Person person) {
```

```
        this.person=person;
    }
}
```

NOTE: *As of Spring 4.3, @Autowired became optional on classes with a single constructor.* In the above example, Spring would still inject an instance of the Person class if you omitted the @Autowired annotation.

@Qualifier

This annotation is used along with @Autowired annotation. When you need more control of the dependency injection process, @Qualifier can be used. @Qualifier can be specified on individual constructor arguments or method parameters. This annotation is used to avoid confusion which occurs when you create more than one bean of the same type and want to wire only one of them with a property.

Consider an example where an interface BeanInterface is implemented by two beans BeanB1 and BeanB2.

```
@Component
```

```
public class BeanB1 implements BeanInterface {  
    //  
}
```

```
@Component
```

```
public class BeanB2 implements BeanInterface {  
    //  
}
```

Now if BeanA autowires this interface, Spring will not know which one of the two implementations to inject. One solution to this problem is the use of the @Qualifier annotation.

```
@Component
```

```
public class BeanA {  
    @Autowired
```

```
@Qualifier("beanB2")  
private BeanInterface dependency;  
...  
}
```

With the `@Qualifier` annotation added, Spring will now know which bean to autowire where beanB2 is the name of BeanB2.

@Configuration

It is used on classes which define beans.

`@Configuration` is an analog for XML configuration file – it is configuration using Java class. J

ava class annotated with `@Configuration` is a configuration by itself and will have methods to instantiate and configure the dependencies.

Here is an example:

@Configuration

```
public class DataConfig{
```

```
@Bean
public DataSource source(){
    DataSource source = new OracleDataSource();
    source.setURL();
    source.setUser();
    return source;
}

@Bean
public PlatformTransactionManager manager(){
    PlatformTransactionManager manager = new
BasicDataSourceTransactionManager();
    manager.setDataSource(source());
    return manager;
}
}
```

@ComponentScan

This annotation is used with @Configuration annotation to allow Spring to know the packages to scan for annotated components.

It is also used to specify base packages using `basePackageClasses` or `basePackage` attributes to scan. If specific packages are not defined, scanning will occur from the package of the class that declares this annotation

@Bean

This annotation is used at the method level. `@Bean` annotation works with `@Configuration` to create Spring beans. As mentioned earlier, `@Configuration` will have methods to instantiate and configure dependencies. Such methods will be annotated with `@Bean`. The method annotated with this annotation works as bean ID and it creates and returns the actual bean.

@Lazy

This annotation is used on component classes. By default all autowired dependencies are created and configured at startup.

But if you want to initialize a bean lazily, you can use **@Lazy** annotation over the class.

This means that the bean will be created and initialized only when it is first requested for.

You can also use this annotation on @Configuration classes.

This indicates that all @Bean methods within that @Configuration should be lazily initialized.

@Value

This annotation is used at the field, constructor parameter, and method parameter level. The @Value annotation indicates a default value expression for the field or parameter to initialize the property with. As the @Autowired annotation tells Spring to inject object into another when it loads your application context, you can also use @Value annotation to inject values from a property file into a bean's attribute. It supports both #{...} and \${...} placeholders.

Spring Framework

Stereotype Annotations

@Component

This annotation is used on classes to indicate a Spring component. The `@Component` annotation marks the Java class as a bean or say component so that the component-scanning mechanism of Spring can add into the application context.

@Controller

The `@Controller` annotation is used to indicate the class is a Spring controller. This annotation can be used to identify controllers for Spring MVC or Spring WebFlux.

@Service

This annotation is used on a class. The `@Service` marks a Java class that performs some service, such as execute business logic, perform calculations and call external APIs.

This annotation is a specialized form of the `@Component` annotation intended to be used in the service layer.

@Repository

This annotation is used on Java classes which directly access the database. The `@Repository` annotation works as marker for any class that fulfills the role of repository or Data Access Object.

This annotation has a automatic translation feature. For example, when an exception occurs in the `@Repository` there is a handler for that exception and there is no need to add a try catch block.

Spring Boot Annotations

@EnableAutoConfiguration

This annotation is usually placed on the main application class. The `@EnableAutoConfiguration` annotation implicitly defines a base “search package”. This annotation tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.

@SpringBootApplication

This annotation is used on the application class while setting up a Spring Boot project. The class that is annotated with the `@SpringBootApplication` must be kept in the base package. The one thing that the `@SpringBootApplication` does is a component scan. But it will scan only its sub-packages. As an example, if you put the class annotated with `@SpringBootApplication` in `com.example` then `@SpringBootApplication` will scan all its sub-packages, such as `com.example.a`, `com.example.b`, and `com.example.a.x`. The `@SpringBootApplication` is a convenient annotation that adds all the following:

- `@Configuration`
- `@EnableAutoConfiguration`
- `@ComponentScan`

Spring MVC and REST Annotations

@Controller

This annotation is used on Java classes that play the role of controller in your application. The `@Controller` annotation allows autodetection of component classes in the classpath and auto-registering bean definitions for them. To enable autodetection of such annotated controllers, you can add component scanning to your configuration. The Java class annotated with `@Controller` is capable of handling multiple request mappings.

This annotation can be used with Spring MVC and Spring WebFlux.

@RequestMapping

This annotation is used both at class and method level. The `@RequestMapping` annotation is used to map web requests onto specific handler classes and handler methods. When `@RequestMapping` is used on class level it creates a

base URI for which the controller will be used. When this annotation is used on methods it will give you the URI on which the handler methods will be executed. From this you can infer that the class level request mapping will remain the same whereas each handler method will have their own request mapping.

Sometimes you may want to perform different operations based on the HTTP method used, even though the request URI may remain the same. In such situations, you can use the `method` attribute of `@RequestMapping` with an HTTP method value to narrow down the HTTP methods in order to invoke the methods of your class.

Here is a basic example on how a controller along with request mappings work:

```
@Controller
```

```
@RequestMapping("/welcome")
```

```
public class WelcomeController{
```

```
    @RequestMapping(method = RequestMethod.GET)
```

```
    public String welcomeAll(){
```

```
        return "welcome all";
```

```
    }
```

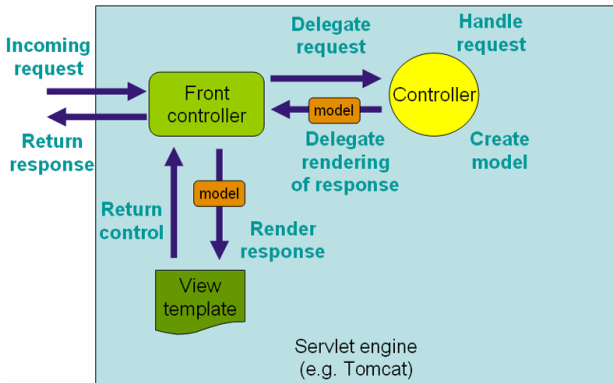
```
}
```

USING THE SPRING @REQUESTMAPPING ANNOTATION

@RequestMapping is one of the most common annotation used in Spring Web applications. This annotation maps HTTP requests to handler methods of MVC and REST controllers.

In Spring MVC applications, the RequestDispatcher (Front Controller Below) servlet is responsible for routing incoming HTTP requests to handler methods of controllers.

When configuring Spring MVC, you need to specify the mappings between the requests and handler methods.



To configure the mapping of web requests, you use the `@RequestMapping` annotation.

The `@RequestMapping` annotation can be applied to class-level and/or method-level in a controller.

The class-level annotation maps a specific request path or pattern onto a controller. You can then apply additional method-level annotations to make mappings more specific to handler methods.

Here is an example of the `@RequestMapping` annotation applied to both class and methods.

```
@RestController
@RequestMapping("/home")
public class IndexController {
    @RequestMapping("/")
    String get(){
        //mapped to hostname:port/home/
    }
}
```

```
        return "Hello from get";
    }
    @RequestMapping("/index")
    String index(){
        //mapped to hostname:port/home/index/
        return "Hello from index";
    }
}
```

With the preceding code, requests to /home will be handled by get() while request to /home/index will be handled by index().

@RequestMapping with Multiple URIs

You can have multiple request mappings for a method. For that add one @RequestMapping annotation with a list of values.

```
@RestController
@RequestMapping("/home")
public class IndexController {
```

```
@RequestMapping(value={"", "/page",  
"page*", "view/*, **/msg"})  
String indexMultipleMapping(){  
    return "Hello from index multiple mapping."  
}  
}
```

As you can see in this code, `@RequestMapping` supports wildcards and ant-style paths. For the preceding code, all these URLs will be handled by `indexMultipleMapping()`.

- localhost:8080/home
- localhost:8080/home/
- localhost:8080/home/page
- localhost:8080/home/pageabc
- localhost:8080/home/view/
- localhost:8080/home/view/view

@RequestMapping with @RequestParam

The `@RequestParam` annotation is used with `@RequestMapping` to bind a web request parameter to the parameter of the handler method.

The `@RequestParam` annotation can be used with or without a value. The value specifies the request param that needs to be mapped to the handler method parameter, as shown in this code snippet.

```
@RestController
@RequestMapping("/home")
public class IndexController {

    @RequestMapping(value = "/id")

    String getIdByValue(@RequestParam("id") String
personId){

        System.out.println("ID is "+personId);
```

```
    return "Get ID from query string of URL with value  
element";
```

```
}
```

```
@RequestMapping(value = "/personId")
```

```
String getId(@RequestParam String personId){
```

```
    System.out.println("ID is "+personId);
```

```
    return "Get ID from query string of URL without value  
element";
```

```
}
```

```
}
```

In Line 6 of this code, the request param id will be mapped to the personId parameter personId of the getIdByValue() handler method.

An example URL is this:

localhost:8090/home/id?id=5

The value element of `@RequestParam` can be omitted if the request param and handler method parameter names are same, as shown in Line 11.

An example URL is this:

`localhost:8090/home/personId?personId=5`

The required element of `@RequestParam` defines whether the parameter value is required or not.

```
@RestController
@RequestMapping("/home")
public class IndexController {
    @RequestMapping(value = "/name")
    String getName(@RequestParam(value = "person",
        required = false) String personName){
        return "Required element of request param";
    }
}
```

In this code snippet, as the required element is specified as false, the getName() handler method will be called for both of these URLs:

/home/name?person=xyz

/home/name

The default value of the @RequestParam is used to provide a default value when the request param is not provided or is empty

```
@RestController
```

```
@RequestMapping("/home")
```

```
public class IndexController {
```

```
    @RequestMapping(value = "/name")
```

```
    String getName(@RequestParam(value = "person",  
defaultValue = "John") String personName ) {
```

```
        return "Required element of request param";
```

```
    }
```

```
}
```

In this code, if the `person` request param is empty in a request, the `getName()` handler method will receive the default value `John` as its parameter

Using `@RequestMapping` with HTTP Method

The Spring MVC `@RequestMapping` annotation is capable of handling HTTP request methods, such as GET, PUT, POST, DELETE, and PATCH.

By default all requests are assumed to be of HTTP GET type.

In order to define a request mapping with a specific HTTP method, you need to declare the HTTP method in `@RequestMapping` using the `method` element as follows.

`@RestController`


```
@RequestMapping("/home")
public class IndexController {
    @RequestMapping(method = RequestMethod.GET)
    String get(){
        return "Hello from get";
    }
    @RequestMapping(method = RequestMethod.DELETE)
    String delete(){
        return "Hello from delete";
    }
    @RequestMapping(method = RequestMethod.POST)
    String post(){
        return "Hello from post";
    }
    @RequestMapping(method = RequestMethod.PUT)
    String put(){
        return "Hello from put";
    }
    @RequestMapping(method = RequestMethod.PATCH)
```

```
String patch(){  
    return "Hello from patch";  
}  
}
```

In the code snippet above, the method element of the `@RequestMapping` annotations indicates the HTTP method type of the HTTP request.

All the handler methods will handle requests coming to the same URL (`/home`), but will depend on the HTTP method being used.

For example, a POST request to `/home` will be handled by the `post()` method. While a DELETE request to `/home` will be handled by the `delete()` method.

Using @RequestMapping with Producible and Consumable

The request mapping types can be narrowed down using the `produces` and `consumes` elements of the `@RequestMapping` annotation.

In order to produce the object in the requested media type, you use the produces element of @RequestMapping in combination with the @ResponseBody annotation.

You can also consume the object with the requested media type using the consumes element of @RequestMapping in combination with the @RequestBody annotation.

The code to use producible and consumable with @RequestMapping is this.

```
@RestController
```

```
@RequestMapping("/home")
```

```
public class IndexController {
```

```
    @RequestMapping(value = "/prod", produces =  
    {"application/JSON"})
```

```
    @ResponseBody
```

```
    String getProduces(){
```

```
    return "Produces attribute";  
  
}
```

```
@RequestMapping(value = "/cons", consumes =  
{ "application/JSON", "application/XML" })
```

```
String getConsumes(){  
  
    return "Consumes attribute";  
  
}  
  
}
```

In this code, the `getProduces()` handler method produces a JSON response. The `getConsumes()` handler method consumes JSON as well as XML present in requests.

@RequestMapping with Headers

The `@RequestMapping` annotation provides a header element to narrow down the request mapping based on headers present in the request.

You can specify the header element as `myHeader = myValue`.

```
@RestController
```

```
@RequestMapping("/home")
```

```
public class IndexController {
```

```
    @RequestMapping(value = "/head", headers = {"content-type=text/plain"})
```

```
    String post(){
```

```
        return "Mapping applied along with headers";
```

```
    }
```

```
}
```

In the above code snippet, the headers attribute of the `@RequestMapping` annotation narrows down the mapping to the `post()` method. With this, the `post()` method will handle requests to `/home/head` whose content-type header specifies plain text as the value.

You can also indicate multiple header values like this:

```
@RestController
@RequestMapping("/home")
public class IndexController {
    @RequestMapping(value = "/head", headers =
{"content-type=text/plain", "content-type=text/html"})
    String post(){
        return "Mapping applied along with headers";
    }
}
```

Here it implies that both text/plain as well as text/html are accepted by the post() handler method.

@RequestMapping with Request Parameters

The params element of the @RequestMapping annotation further helps to narrow down request mapping. Using the params element, you can have multiple handler methods handling requests to the same URL, but with different parameters.

You can define params as myParams = myValue. You can also use the negation operator to specify that a particular parameter value is not supported in the request.

```
@RestController
@RequestMapping("/home")
public class IndexController {
    @RequestMapping(value = "/fetch", params =
{"personId=10"})
```

```

String getParams(@RequestParam("personId") String
id){
    return "Fetched parameter using params attribute =
"+id;
}
@RequestMapping(value = "/fetch", params =
{"personId=20"})
String
getParamsDifferent(@RequestParam("personId") String
id){
    return "Fetched parameter using params attribute =
"+id;
}
}

```

In this code snippet, both the `getParams()` and `getParamsDifferent()` methods will handle requests coming to the same URL (`/home/fetch`) but will execute depending on the `params` element.

For example, when the URL is /home/fetch?id=10 the `getParams()` handler method will be executed with the id value 10.. For the URL, `localhost:8080/home/fetch?personId=20`, the `getParamsDifferent()` handler method gets executed with the id value 20.

Using `@RequestMapping` with Dynamic URIs

The `@RequestMapping` annotation is used in combination with the `@PathVariable` annotation to handle dynamic URIs. In this use case, the URI values can act as the parameter of the handler methods in the controller. You can also use regular expressions to only accept the dynamic URI values that match the regular expression.

```
@RestController
@RequestMapping("/home")
public class IndexController {
    @RequestMapping(value = "/fetch/{id}", method =
RequestMethod.GET)
    String getDynamicUriValue(@PathVariable String id){
        System.out.println("ID is "+id);
        return "Dynamic URI parameter fetched";

    }
    @RequestMapping(value = "/fetch/{id:[a-z]+}/{name}",
method = RequestMethod.GET)
    String
getDynamicUriValueRegex(@PathVariable("name")
String name){
        System.out.println("Name is "+name);
        return "Dynamic URI parameter fetched using regex";

    }
}
```

In this code, the method `getDynamicUriValue()` will execute for a request to `localhost:8080/home/fetch/10`. Also, the `id` parameter of the `getDynamicUriValue()` handler method will be populated with the value 10 dynamically.

The method `getDynamicUriValueRegex()` will execute for a request to `localhost:8080/home/fetch/category/shirt`. However, an exception will be thrown for a request to `/home/fetch/10/shirt` as it does not match the regular expression.

`@PathVariable` works differently from `@RequestParam`. You use `@RequestParam` to obtain the values of the query parameters from the URI. On the other hand, you use `@PathVariable` to obtain the parameter values from the URI template.

The @RequestMapping Default Handler Method

In the controller class you can have default handler method that gets executed when there is a request for a default URI.

Here is an example of a default handler method.

```
@RestController
@RequestMapping("/home")
public class IndexController {
    @RequestMapping()
    String default(){

        return "This is a default method for the class";
    }
}
```

In this code, A request to /home will be handled by the default() method as the annotation does not specify any value.

@RequestMapping Shortcuts

Spring 4.3 introduced method-level variants, also known as composed annotations of @RequestMapping. The composed annotations better express the semantics of the annotated methods. They act as wrapper to @RequestMapping and have become the standard ways of defining the endpoints.

For example, @GetMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET).

The method level variants are:

@GetMapping

@PostMapping

@PutMapping

@DeleteMapping

@PatchMapping

The following code shows using the composed annotations.

@RestController

@RequestMapping("/home")

public class IndexController {

 @GetMapping("/person")

 public @ResponseBody ResponseEntity<String>

 getPerson() {

 return new ResponseEntity<String>("Response from GET", HttpStatus.OK);

 }

 @GetMapping("/person/{id}")

 public @ResponseBody ResponseEntity<String>

 getPersonById(@PathVariable String id){

 return new ResponseEntity<String>("Response from GET with id " +id,HttpStatus.OK);

 }

```
@PostMapping("/person")
public @ResponseBody ResponseEntity<String>
postPerson() {
    return new ResponseEntity<String>("Response from
POST method", HttpStatus.OK);
}
@PutMapping("/person")
public @ResponseBody ResponseEntity<String>
putPerson() {
    return new ResponseEntity<String>("Response from
PUT method", HttpStatus.OK);
}
@DeleteMapping("/person")
public @ResponseBody ResponseEntity<String>
deletePerson() {
    return new ResponseEntity<String>("Response from
DELETE method", HttpStatus.OK);
}
@PatchMapping("/person")
```

```
public @ResponseBody ResponseEntity<String>
patchPerson() {
    return new ResponseEntity<String>("Response from
PATCH method", HttpStatus.OK);
}
}
```

In this code, each of the handler methods are annotated with the composed variants of **@RequestMapping**. Although, each variant can be interchangeably used with **@RequestMapping** with the method attribute, it's considered a best practice to use the composed variant. Primarily because the composed annotations reduce the configuration metadata on the application side and the code is more readable.

@RequestMapping Conclusion

As you can see in this post, the **@RequestMapping** annotation is very versatile. You can use this annotation to configure Spring MVC to handle a variety of use cases. It can be used to configure traditional web page

requests, and well as RESTful web services in Spring MVC.

Spring boot JPA Example

```
-----  
package hello;  
import javax.persistence.*;  
@Entity  
public class Customer {  
    @Id  
    @GeneratedValue(strategy=GenerationType.AUTO)  
    private Long id;  
    private String firstName;  
    private String lastName;  
  
    protected Customer() {}  
  
    public Customer(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
}
```

```
}

@Override
public String toString() {
    return String.format(
        "Customer[id=%d, firstName='%s',
lastName='%s']",
        id, firstName, lastName);
}
}
```

The `Customer` class is annotated with `@Entity`, indicating that it is a JPA entity. For lack of a `@Table` annotation, it is assumed that this entity will be mapped to a table named `Customer`.

The Customer's id property is annotated with `@Id` so that JPA will recognize it as the object's ID. The id property is also annotated with `@GeneratedValue` to indicate that the ID should be generated automatically.

The other two properties, `firstName` and `lastName` are left unannotated. It is assumed that they'll be mapped to columns that share the same name as the properties themselves.

Create simple queries

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.

To see how this works, create a repository interface that works with `Customer` entities:

```
import
```

```
org.springframework.data.repository.CrudRepository;
```

```
public interface CustomerRepository extends  
    CrudRepository<Customer, Long> {
```

```
List<Customer> findByLastName(String lastName);  
}
```

CustomerRepository extends the CrudRepository interface. The type of entity and ID that it works with, Customer and Long, are specified in the generic parameters on CrudRepository.

By extending CrudRepository, CustomerRepository inherits several methods for working with Customer persistence, including methods for saving, deleting, and finding Customer entities.

Spring Data JPA also allows you to define other query methods by simply declaring their method signature. In the case of CustomerRepository, this is shown with a findByLastName() method.

You don't have to write an implementation of the repository interface. Spring Data JPA creates an

implementation on the fly when you run the application.

@SpringBootApplication is a convenience annotation that adds all of the following:

@Configuration tags the class as a source of bean definitions for the application context.

@EnableAutoConfiguration tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.

Normally you would add **@EnableWebMvc** for a Spring MVC app, but Spring Boot adds it automatically when it sees spring-webmvc on the classpath. This flags the application as a web application and activates key behaviors such as setting up a DispatcherServlet.

@ComponentScan tells Spring to look for other components, configurations, and services in the hello package, allowing it to find the controllers.

application.properties

```
spring.jpa.hibernate.ddl-auto=create  
  
spring.datasource.url=jdbc:mysql://localhost:3306/  
db_example  
  
spring.datasource.username=springuser  
  
spring.datasource.password=ThePassword
```

Many To One Example

@JsonIgnore

@ManyToOne

private Account account;

REST is a style of software architecture for distributed hypermedia systems

REST stands for REpresentational State Transfer. The definitions for REST can be vague. So, let's understand the important concepts.

Key abstraction in REST is a Resource. There is no restriction on what can be a resource. A todo is a resource. A person on facebook is a resource.

A resource has an URI (Uniform Resource Identifier):

/user/Ranga/todos/1

/person/Ranga

A resource will have representations

XML

HTML

JSON

A resource will have state. The representation of a resource should capture its current state.

When a resource is requested, we provide the representation of the resource.

REST and HTTP

REST builds on top of HTTP (Hypertext Transfer Protocol). HTTP is the language of the web.

HTTP has a few important verbs.

- POST - Create a new resource
- GET - Read a resource
- PUT - Update an existing resource
- DELETE - Delete a resource

HTTP also defines standard response codes.

- 200 - SUCCESS
- 404 - RESOURCE NOT FOUND
- 400 - BAD REQUEST
- 201 - CREATED
- 401 - UNAUTHORIZED
- 415 - UNSUPPORTED TYPE -
Representation not supported for the
resource
- 500 - SERVER ERROR

Restful Service Constraints

- Client - Server : There should be a
service producer and a service
consumer.

- The interface (URL) is uniform and exposing resources. Interface uses nouns (not actions)
- The service is stateless. Even if the service is called 10 times, the result must be the same.
- The service result should be Cacheable. HTTP cache, for example.
- Service should assume a Layered architecture. Client should not assume direct connection to server - it might be getting info from a middle layer - cache.

When do you use JPA?

- SQL Database
- Static Domain Model
- Mostly CRUD
- Mostly Simple Queries/Mappings

Bootstrapping with Spring Initializr

Creating a REST service with Spring Initializr is a cake walk. We will use Spring Web MVC as our web framework.

Spring Initializr <http://start.spring.io/> is great tool to bootstrap your Spring Boot projects.

As shown in the image above, following steps have to be done

- Launch Spring Initializr and choose the following
 - Choose `com.in28minutes.springboot.rest.example` as Group
 - Choose `spring-boot-2-rest-service-basic` as Artifact
 - Choose following dependencies
 - Web
 - JPA

- H2
- DevTools
- TODO
- Click Generate Project.
- Import the project into Eclipse. File -> Import -> Existing Maven Project.

Do not forget to add JPA and H2 as dependencies

Create Your First JPA Entity

The first step is to create a JPA Entity. Lets create a simple Student Entity with a primary key id.

```
@Entity
```

```
public class Student {
```

```
    @Id
```

```
    @GeneratedValue
```

```
private Long id;  
  
private String name;  
  
private String passportNumber;
```

Important things to note:

- `@Entity`: Specifies that the class is an entity. This annotation is applied to the entity class.
- `@Id`: Specifies the primary key of an entity.
- `@GeneratedValue`: Provides for the specification of generation strategies for the values of primary keys.
- `public Student()`: Default constructor to make JPA Happy

When the application reloads, you can launch H2 console at <http://localhost:8080/h2-console>.

You will see that a new table called 'student' is created in H2 Console.

How did the Student table get created?

Spring Boot Auto Configuration detects that we are using an in-memory database H2. It autoconfigures the property to create the database tables based on the Entities.

Let's now populate some data into the student table.

/src/main/resources/data.sql










```
insert into student
values(10001,'Ranga', 'E1234567');

insert into student
values(10002,'Ravi', 'A1234568');
```

When the application reloads you would see following statements in the log indicating that the sql files are picked up.

```
Executing SQL script from URL  
[file:/in28Minutes/git/spring-boot-  
examples/spring-boot-2-jdbc-with-  
h2/target/classes/data.sql]
```

After App Reload, When you login to H2 Console (<http://localhost:8080/h2-console>) you can see that the student table is created and the data is populated.

-  jdbc:h2:mem:testdb
-  STUDENT
 -  ID
 -  NAME
 -  PASSPORT_NUMBER
 -  Indexes
 -  INFORMATION_SCHEMA
 -  Users
-  H2 1.4.196 (2017-06-10)

Run Run Selected Auto com

SELECT * FROM STUDENT

SELECT * FROM STUDENT;

ID	NAME	PASSPORT_
10001	Ranga	E1234567
10002	Ravi	A1234568

(2 rows, 6 ms)

Create JPA Repository class to Read Student from Database

/src/main/java/com/in28minutes/springboot/jpa/hibernate/h2/example/student/StudentRepository.java

We create a simple interface StudentRepository extending JpaRepository.

```
@Repository
```

```
public interface StudentRepository extends  
JpaRepository<Student, Long>{
```

Notes

- We will talk about all the methods in the JpaRepository a little later.
- `public interface StudentRepository extends JpaRepository<Student, Long>` - We are extending JpaRepository using two generics - Student & Long. Student

is the entity that is being managed and the primary key of Student is Long.

JpaRepository

JpaRepository (Defined in Spring Data JPA) is the JPA specific Repository interface.

```
public interface JpaRepository<T, ID extends
```

```
Serializable>
```

```
extends
```

```
PagingAndSortingRepository<T, ID>,  
QueryByExampleExecutor<T> {
```

JpaRepository extends PagingAndSortingRepository which in turn extends CrudRepository interface. So, JpaRepository inherits all the methods from the two interfaces shown below.

PagingAndSortingRepository

```
public abstract interface
PagingAndSortingRepository extends
CrudRepository {

    public abstract Iterable findAll(Sort
arg0);

    public abstract Page findAll(Pageable
arg0);

}
```

CrudRepository

```
public interface CrudRepository<T, ID extends
Serializable>

    extends Repository<T, ID> {
```

```
<S extends T> S save(S entity);

T findOne(ID primaryKey);

Iterable<T> findAll();

Long count();

void delete(T entity);

boolean exists(ID primaryKey);

// ... more functionality omitted.
}
```

Exposing Resources using StudentResource

Lets start with setting up the StudentResource class and then move into creating methods to handle different kinds of request methods to the Student Resouce.

Setting up Student Resource

```
@RestController
```

```
public class StudentResource {
```

```
    @Autowired
```

```
    private StudentRepository  
studentRepository;
```

Notes

- `@RestController` : Combination of `@Controller` and `@ResponseBody`.

Beans returned are converted to/from JSON/XML.

- `@Autowired private StudentRepository studentRepository` : Autowire the `StudentRepository` so that we can retrieve and save data to database.

Exposing GET methods on Student Resource

Let's create the method to expose the details of all students.

```
@GetMapping("/students")  
  
public List<Student> retrieveAllStudents() {  
    return studentRepository.findAll();  
}
```

Below picture shows how we can execute a Get Request Method on a Resource using Postman - my favorite tool to run rest

services.

http://localhost:8080/ ×



GET ▼

http://localhost:8080

Authorization

Headers

Body

Type

No A

Body

Cookies (1)

Headers (3)

Pretty

Raw

Preview

JSON

1 ▾ [

2 ▾ {

"i": "10001"

- URL - `http://localhost:8080/students`
- Request Method - GET

Response

```
[  
  {  
    "id": 10001,  
    "name": "Ranga",  
    "passportNumber": "E1234567"  
  },  
  {  
    "id": 10002,  
    "name": "Ravi",  
    "passportNumber": "A1234568"  
  }  
]
```

```
]
```

Let's create a method to expose the details of a specific student.

```
@GetMapping("/students/{id}")

public Student retrieveStudent(@PathVariable
long id) {

    Optional<Student> student =
studentRepository.findById(id);

    if (!student.isPresent())

        throw new
StudentNotFoundException("id-" + id);

    return student.get();
}
```

Let's execute another GET
request

http://localhost:8080/ ×



GET ▼

http://localhost:8080

Authorization

Headers

Body

Type

No A

Body

Cookies (1)

Headers (3)

Pretty

Raw

Preview

JSON

1 ▾ [

2 ▾ {

{

"i": "10001"

- URL -
`http://localhost:8080/students/10002`
- Request Method - GET

Response

```
{  
  "id": 10002,  
  "name": "Ravi",  
  "passportNumber": "A1234568"  
}
```

Expose DELETE Method on Student Resource

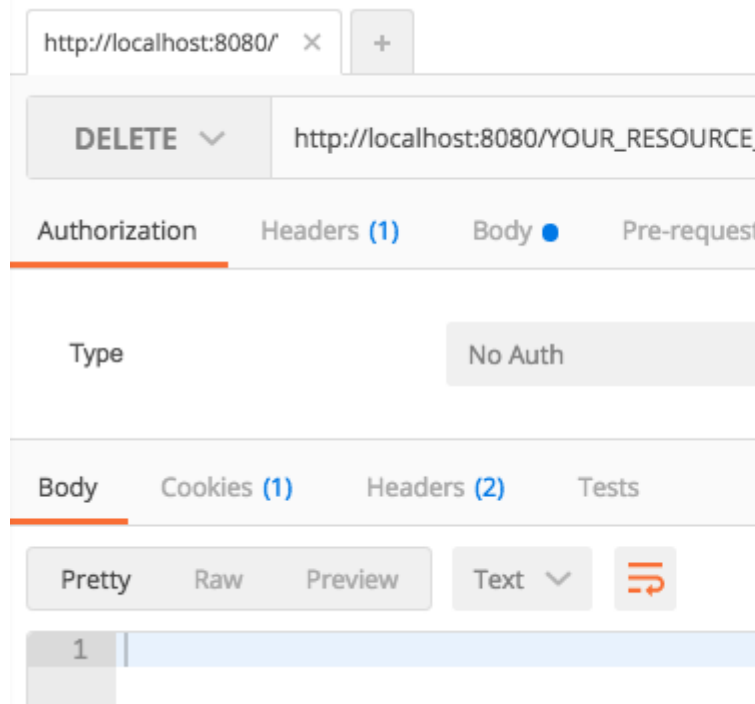
Delete method is simple. All you have to do is to call `studentRepository.deleteById(id)`.

```
@DeleteMapping("/students/{id}")
```

```
public void deleteStudent(@PathVariable long id) {  
  
    studentRepository.deleteById(id);  
  
}
```

Below picture shows how we can execute a DELETE Request method on a Resource from Postman - my favorite tool to run rest

services.



- URL -
`http://localhost:8080/students/10002`
- Request Method - DELETE

Request - Empty Body Response with status 200 - Successful

Expose POST Method to create a new student

POST method is simple. All you have to do is to call `studentRepository.save(student)`. Note that we are using `@RequestBody` to map the student details from request to bean. We are also returning a `ResponseEntity` with a header containing the URL of the created resource.

```
@PostMapping("/students")  
  
public ResponseEntity<Object>  
createStudent(@RequestBody Student student) {
```



```
        Student savedStudent =
studentRepository.save(student);

        URI location =
ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")

        .buildAndExpand(savedStudent.getId())
        .toUri();

        return
ResponseEntity.created(location).build();

    }
```

Below picture shows how we can execute a POST Request method on a Resource from Postman - my favorite tool to run rest

services.

http://localhost:8080/ ×



POST ▼

http://localhost:8080/YOUR_URL

Authorization

Headers (1)

Body ●

Pre-request



form-data



x-www-form-urlencoded



raw



```
1 {  
2   "your_content": "your_json_content"  
3 }
```

Body

Cookies (1)

Headers (3)

Tests

Pretty

Raw

Preview

Text ▼



```
1
```

- URL - <http://localhost:8080/students>
- Request Method - POST

Request

```
{  
  "name": "Tom",  
  "passportNumber": "Z1234567"  
}
```

Response

- Status 201 - CREATED
- Header Location
→ <http://localhost:8080/students/2>

Expose PUT Method to update existing student

Before updating the student we check if the student exists. If the student does not exist,

we return a not found status. Otherwise, we save the student details using `studentRepository.save` method.

```
@PutMapping("/students/{id}")

public ResponseEntity<Object>
updateStudent(@RequestBody Student student,
@PathVariable long id) {

    Optional<Student> studentOptional =
studentRepository.findById(id);

    if (!studentOptional.isPresent())

        return
ResponseEntity.notFound().build();

    student.setId(id);
```

```
        studentRepository.save(student);

        return
        ResponseEntity.noContent().build();
    }
```

Below picture shows how we can execute a PUT Request method on a Resource from Postman - my favorite tool to run rest

services.

http://localhost:8080/ x

+

PUT v

http://localhost:8080/YOUR_RESOURCE

Authorization

Headers (1)

Body ●

Pre-request

☐ form-data

☐ x-www-form-urlencoded

☒ raw

1 {

2 "Your_JSON_CONTENT": "Your_JSON_CONTENT"

3 }

Body

Cookies (1)

Headers (2)

Tests

- URL →
http://localhost:8080/students/10002
- Request
 - Method → PUT

Request

```
{  
    "name": "Tom",  
    "passportNumber": "Z1234567"  
}
```

Response with status 200 - Successful

EXAMPLE

```
@GetMapping("/students")  
public List<Student> retrieveAllStudents() {  
    return studentRepository.findAll();  
}
```

```
    @GetMapping("/students/{id}")
    public Student retrieveStudent(@PathVariable
long id) {
        Optional<Student> student =
studentRepository.findById(id);
        if (!student.isPresent())
            throw new
StudentNotFoundException("id-" + id);
        return student.get();
    }
```

```
    @DeleteMapping("/students/{id}")
    public void deleteStudent(@PathVariable long
id) {
        studentRepository.deleteById(id);
    }
```

```
    @PostMapping("/students")
    public ResponseEntity<Object>
createStudent(@RequestBody Student student) {
```

```

        Student savedStudent =
studentRepository.save(student);
        URI location =
ServletUriComponentsBuilder.fromCurrentRequest().pa
th("/{id}")

        .buildAndExpand(savedStudent.getId()).toUri();
        return
ResponseEntity.created(location).build();

}

```

```

@PutMapping("/students/{id}")
public ResponseEntity<Object>
updateStudent(@RequestBody Student student,
@PathVariable long id) {
        Optional<Student> studentOptional =
studentRepository.findById(id);
        if (!studentOptional.isPresent())

```

```
        return  
        ResponseEntity.notFound().build();  
        student.setId(id);  
        studentRepository.save(student);  
        return  
        ResponseEntity.noContent().build();  
    }  
}
```

Most important feature of Spring Framework is Dependency Injection. At the core of all Spring Modules is Dependency Injection or IOC Inversion of Control.

Spring Boot Starter Project Options

As we see from Spring Boot Starter Web, starter projects help us in quickly getting started with developing specific types of applications.

- spring-boot-starter-web-services - SOAP Web Services
- spring-boot-starter-web - Web & RESTful applications
- spring-boot-starter-test - Unit testing and Integration Testing
- spring-boot-starter-jdbc - Traditional JDBC
- spring-boot-starter-hateoas - Add HATEOAS features to your services
- spring-boot-starter-security - Authentication and Authorization using Spring Security

- `spring-boot-starter-data-jpa` - Spring Data JPA with Hibernate
- `spring-boot-starter-cache` - Enabling Spring Framework's caching support
- `spring-boot-starter-data-rest` - Expose Simple REST Services using Spring Data REST

Other Goals of Spring Boot

There are a few starters for technical stuff as well

- `spring-boot-starter-actuator` - To use advanced features like monitoring & tracing to your application out of the box
- `spring-boot-starter-undertow`, `spring-boot-starter-jetty`, `spring-boot-starter-tomcat` - To pick your specific choice of Embedded Servlet Container

- spring-boot-starter-logging - For Logging using logback
- spring-boot-starter-log4j2 - Logging using Log4j2

Spring Boot aims to enable production ready applications in quick time.

- Actuator : Enables Advanced Monitoring and Tracing of applications.
- Embedded Server Integrations - Since server is integrated into the application, I would NOT need to have a separate application server installed on the server.
- Default Error Handling

Hibernate Interview Questions

1. How to avoid constraint violation exception parent key not found?

2. What are the ORM level?

The ORM levels are:

- * Pure relational (stored procedure.)
- * Light objects mapping (JDBC)
- * Medium object mapping
- * Full object Mapping (composition, inheritance, polymorphism, persistence by reachability)

3. What does ORM consists of?

An ORM solution consists of the following four pieces:

- * API for performing basic CRUD operations
- * API to express queries referring to classes
- * Facilities to specify metadata
- * Optimization facilities : dirty checking, lazy associations fetching

4. What are the general considerations or best practices for defining your Hibernate persistent classes?

1. You must have a default no-argument constructor for your persistent classes and there should be getXXX() (i.e. accessor/getter) and setXXX() (i.e. mutator/setter) methods for all your persistable instance variables.

2. You should implement the equals() and hashCode() methods based on your business key and it is important not to use the id field in your equals() and hashCode() definition if the id field is a surrogate key (i.e. Hibernate managed identifier). This is because the Hibernate only generates and sets the field when saving the object.

3. It is recommended to implement the Serializable interface. This is potentially useful if you want to migrate around a multi-processor cluster.

4. The persistent class should not be final because if it is final then lazy loading cannot be used by creating proxy objects.

5. Use XDoclet tags for generating your *.hbm.xml files or Annotations (JDK 1.5 onwards), which are less verbose than *.hbm.xml files.

5. How would you reattach detached objects to a session when the same object has already been loaded into the session?

You can use the `session.merge()` method call.

6. What is the difference between the `session.update()` method and the `session.lock()` method?

Both of these methods and `saveOrUpdate()` method are intended for reattaching a detached object. The `session.lock()` method simply reattaches the object to the session without checking or updating the database on the assumption that the database is sync with the detached object. It is the best practice to use either `session.update(..)` or `session.saveOrUpdate()`. Use `session.lock()` only if you are absolutely sure that the detached object is in sync with your detached object or if it does not matter because you will be overwriting all the columns that would have changed later on within the same transaction.

Note: When you reattach detached objects you need to make sure that the dependent objects are reattached as well.

7. What is the difference between the `session.get()` method and the `session.load()` method?

Both the `session.get(..)` and `session.load()` methods create a persistent object by loading the required object from the database. But if there was not such object in the database then the method `session.load(..)` throws an exception whereas `session.get(&)` returns null.

8. How does Hibernate distinguish between transient (i.e. newly instantiated) and detached objects?

- " Hibernate uses the `version` property, if there is one.
- " If not uses the identifier value. No identifier value means a new object. This does work only for Hibernate managed surrogate keys. Does not work for natural keys and assigned (i.e. not managed by Hibernate) surrogate keys.
- " Write your own strategy with `Interceptor.isUnsaved()`.

9. What are the pros and cons of detached objects?

Pros:

- " When long transactions are required due to user think-time, it is the best practice to break the long transaction up into two or more transactions. You can use detached objects from the first transaction to carry data all the way up to the presentation layer. These detached objects get modified outside a transaction and later on re-attached to a new transaction via another session.

Cons

" In general, working with detached objects is quite cumbersome, and better to not clutter up the session with them if possible. It is better to discard them and re-fetch them on subsequent requests. This approach is not only more portable but also more efficient because - the objects hang around in Hibernate's cache anyway.

" Also from pure rich domain driven design perspective it is recommended to use DTOs (DataTransferObjects) and DOs (DomainObjects) to maintain the separation between Service and UI tiers.

10. What are the benefits of detached objects?

Detached objects can be passed across layers all the way up to the presentation layer without having to use any DTOs (Data Transfer Objects). You can later on re-attach the detached objects to another session.

[Download Interview PDF](#)

11. What is a Session? Can you share a session object between different threads?

Session is a light weight and a non-threadsafe object (No, you cannot share it between threads) that represents a single unit-of-work with the database. Sessions are opened by a SessionFactory and then are closed when all work is complete. Session is the primary interface for the persistence service. A session obtains a database connection lazily (i.e. only when required). To avoid creating too many sessions ThreadLocal class can be used as shown below to get the current session no matter how many times you make call to the currentSession() method.

12. What is a SessionFactory? Is it a thread-safe object?

SessionFactory is Hibernate's concept of a single datastore and is threadsafe so that many threads can access it concurrently and request for sessions and immutable cache of compiled mappings for a single database. A SessionFactory is usually only built once at startup. SessionFactory should be wrapped in some kind of singleton so that it can be easily accessed in an application code.

13. How will you configure Hibernate?

The configuration files hibernate.cfg.xml (or hibernate.properties) and mapping files *.hbm.xml are used

by the Configuration class to create (i.e. configure and bootstrap hibernate) the SessionFactory, which in turn creates the Session instances. Session instances are the primary interface for the persistence service.

" hibernate.cfg.xml (alternatively can use hibernate.properties): These two files are used to configure the hibernate service (connection driver class, connection URL, connection username, connection password, dialect etc). If both files are present in the classpath then hibernate.cfg.xml file overrides the settings found in the hibernate.properties file.

" Mapping files (*.hbm.xml): These files are used to map persistent objects to a relational database. It is the best practice to store each object in an individual mapping file (i.e mapping file per class) because storing large number of persistent classes into one mapping file can be difficult to manage and maintain. The naming convention is to use the same name as the persistent (POJO) class name. For example Account.class will have a mapping file named Account.hbm.xml. Alternatively hibernate annotations can be used as part of your persistent class code instead of the *.hbm.xml files.

14. What is the difference between sorted and ordered collection in hibernate?

sorted collection vs. order collection

sorted collection :-

A sorted collection is sorting a collection by utilizing the sorting features provided by the Java collections framework. The sorting occurs in the memory of JVM which running Hibernate, after the data being read from database using java comparator.

If your collection is not large, it will be more efficient way to sort it.

order collection :-

Order collection is sorting a collection by specifying the order-by clause for sorting this collection when retrieval.

If your collection is very large, it will be more efficient way to sort it .

15. What is component mapping in Hibernate?

- * A component is an object saved as a value, not as a reference
- * A component can be saved directly without needing to declare interfaces or identifier properties
- * Required to define an empty constructor
- * Shared references not supported

16. What are derived properties?

The properties that are not mapped to a column, but calculated at runtime by evaluation of an expression are called derived properties. The expression can be defined using the formula attribute of the element.

People who read this also read:

Core Java Questions

Spring Questions

SCJP 6.0 Certification

EJB Interview Questions

Servlets Questions

17. If you want to see the Hibernate generated SQL statements on console, what should we do?

In Hibernate configuration file set as follows:

```
<property name="show_sql">true</property>
```

18. How do you switch between relational databases without code changes?

Using Hibernate SQL Dialects , we can switch databases. Hibernate will generate appropriate hql queries based on the dialect defined.

19. What are the benefits does HibernateTemplate provide?

The benefits of HibernateTemplate are :

- * HibernateTemplate, a Spring Template class simplifies interactions with Hibernate Session.
- * Common functions are simplified to single method calls.
- * Sessions are automatically closed.
- * Exceptions are automatically caught and converted to runtime exceptions.

20. Define HibernateTemplate?

org.springframework.orm.hibernate.HibernateTemplate is a helper class which provides different methods for querying/retrieving data from the database. It also converts checked HibernateExceptions into unchecked DataAccessExceptions.

[Download Interview PDF](#)

21. Explain Criteria API?

Criteria is a simplified API for retrieving entities by composing Criterion objects. This is a very convenient approach for functionality like “search” screens where there is a variable number of conditions to be placed upon the result set.

Example :

```
List employees = session.createCriteria(Employee.class)
.add(Restrictions.like("name", "a%") )
.add(Restrictions.like("address", "Boston"))
.addOrder(Order.asc("name") )
.list();
```

22. How do you invoke Stored Procedures?

```
<sql-query name="selectAllEmployees_SP" callable="true">
<return alias="emp" class="employee">
<return-property name="empid" column="EMP_ID"/>
<return-property name="name" column="EMP_NAME"/>
<return-property name="address"
column="EMP_ADDRESS"/>
{ ? = call selectAllEmployees() }
</return>
</sql-query>
```

23. What do you mean by Named - SQL query?

Named SQL queries are defined in the mapping xml document and called wherever required.

Example:

```
<sql-query name = "empdetails">
<return alias="emp" class="com.test.Employee"/>
SELECT emp.EMP_ID AS {emp.empid},
emp.EMP_ADDRESS AS {emp.address},
```

```
emp.EMP_NAME AS {emp.name}  
FROM Employee EMP WHERE emp.NAME LIKE :name  
</sql-query>  
Invoke Named Query :  
List people = session.getNamedQuery("empdetails")  
.setString("TomBrady", name)  
.setMaxResults(50)  
.list();
```

24. What does it mean to be inverse?

It informs hibernate to ignore that end of the relationship. If the one-to-many was marked as inverse, hibernate would create a child->parent relationship (child.getParent). If the one-to-many was marked as non-inverse then a child->parent relationship would be created.

25. Define cascade and inverse option in one-many mapping?

cascade - enable operations to cascade to child entities.
cascade="all|none|save-update|delete|all-delete-orphan"

inverse - mark this collection as the "inverse" end of a bidirectional association.

inverse="true|false"

Essentially "inverse" indicates which end of a relationship should be ignored, so when persisting a parent who has a

collection of children, should you ask the parent for its list of children, or ask the children who the parents are?

26. How do you define sequence generated primary key in hibernate?

Using <generator> tag.

Example:-

```
<id column="USER_ID" name="id" type="java.lang.Long">  
<generator class="sequence">  
<param name="table">SEQUENCE_NAME</param>  
</generator>  
</id>
```

27. What is the difference between and merge and update?

Use update() if you are sure that the session does not contain an already persistent instance with the same identifier, and merge() if you want to merge your modifications at any time without consideration of the state of the session.

28. What is the difference between load() and get()?

load() vs. get()

load() :-

Only use the load() method if you are sure that the object exists.

load() method will throw an exception if the unique id is not found in the database. load() just returns a proxy by default and database won't be hit until the proxy is first invoked.

get():-

If you are not sure that the object exists, then use one of the get() methods.

get() method will return null if the unique id is not found in the database.

get() will hit the database immediately.

29. What Does Hibernate Simplify?

Hibernate simplifies:

- * Saving and retrieving your domain objects
- * Making database column and table name changes
- * Centralizing pre save and post retrieve logic
- * Complex joins for retrieving related items
- * Schema creation from object model

30. How do you map Java Objects with Database tables?

- * First we need to write Java domain objects (beans with setter and getter). The variables should be same as database columns.
- * Write hbm.xml, where we map java class to table and database columns to Java class variables.

Example :

```
<hibernate-mapping>
<class name="com.test.User" table="user">
<property column="USER_NAME" length="255"
name="userName" not-null="true" type="java.lang.String"/>
<property column="USER_PASSWORD" length="255"
name="userPassword" not-null="true"
type="java.lang.String"/>
</class>
</hibernate-mapping>
```

[Download Interview PDF](#)

31. What is Hibernate Query Language (HQL)?

Hibernate offers a query language that embodies a very powerful and flexible mechanism to query, store, update, and retrieve objects from a database. This language, the Hibernate query Language (HQL), is an object-oriented extension to SQL.

32. What is the general flow of Hibernate communication with RDBMS?

The general flow of Hibernate communication with RDBMS is :

- * Load the Hibernate configuration file and create configuration object. It will automatically load all hbm mapping files
- * Create session factory from configuration object
- * Get one session from this session factory
- * Create HQL Query
- * Execute query to get list containing Java objects

33. What role does the SessionFactory interface play in Hibernate?

The application obtains Session instances from a SessionFactory. There is typically a single SessionFactory for the whole application-created during application initialization. The SessionFactory caches generate SQL statements and other mapping metadata that Hibernate uses at runtime. It also holds cached data that has been read in one unit of work and may be reused in a future unit of work

SessionFactory sessionFactory =
configuration.buildSessionFactory();

34. What role does the Session interface play in Hibernate?

The Session interface is the primary interface used by Hibernate applications. It is a single-threaded, short-lived object representing a conversation between the application and the persistent store. It allows you to create query objects to retrieve persistent objects.

Session session = sessionFactory.openSession();
Session interface role:

- * Wraps a JDBC connection
- * Factory for Transaction
- * Holds a mandatory (first-level) cache of persistent objects, used when navigating the object graph or looking up objects by identifier

35. What are the Core interfaces are of Hibernate framework?

People who read this also read:

The five core interfaces are used in just about every Hibernate application. Using these interfaces, you can store and retrieve persistent objects and control transactions.

- * Session interface
- * SessionFactory interface
- * Configuration interface

- * Transaction interface
- * Query and Criteria interfaces

36. What are the important tags of hibernate.cfg.xml?

An Action Class is an adapter between the contents of an incoming HTTP request and the corresponding business logic that should be executed to process this request.

37. What are the most common methods of Hibernate configuration?

The most common methods of Hibernate configuration are:

- * Programmatic configuration
- * XML configuration (hibernate.cfg.xml)

38. What is HQL?

HQL stands for Hibernate Query Language. Hibernate allows the user to express queries in its own portable SQL extension and this is called as HQL. It also allows the user to express in native SQL.

39. What is object/relational mapping metadata?

ORM tools require a metadata format for the application to specify the mapping between classes and tables, properties and columns, associations and foreign keys, Java types and SQL types. This information is called the object/relational mapping metadata. It defines the transformation between the different data type systems and relationship representations.

40. What are POJOs?

POJO stands for plain old java objects. These are just basic JavaBeans that have defined setter and getter methods for all the properties that are there in that bean. Besides they can also have some business logic related to that property. Hibernate applications works efficiently with POJOs rather than simple java classes.

[Download Interview PDF](#)

41. What should SessionFactory be placed so that it can be easily accessed?

As far as it is compared to J2EE environment, if the SessionFactory is placed in JNDI then it can be easily accessed and shared between different threads and various components that are hibernate aware. You can set the

SessionFactory to a JNDI by configuring a property `hibernate.session_factory_name` in the `hibernate.properties` file.

42. What are Callback interfaces?

These interfaces are used in the application to receive a notification when some object events occur. Like when an object is loaded, saved or deleted. There is no need to implement callbacks in hibernate applications, but they're useful for implementing certain kinds of generic functionality.

43. What the Core interfaces are of hibernate framework?

There are many benefits from these. Out of which the following are the most important one.

Session Interface : This is the primary interface used by hibernate applications. The instances of this interface are lightweight and are inexpensive to create and destroy.

Hibernate sessions are not thread safe.

SessionFactory Interface : This is a factory that delivers the session objects to hibernate application. Generally there will be a single SessionFactory for the whole application and it will be shared among all the application threads.

Configuration Interface : This interface is used to configure

and bootstrap hibernate. The instance of this interface is used by the application in order to specify the location of hibernate specific mapping documents.

Transaction Interface : This is an optional interface but the above three interfaces are mandatory in each and every application. This interface abstracts the code from any kind of transaction implementations such as JDBC transaction, JTA transaction.

Query and Criteria Interface : This interface allows the user to perform queries and also control the flow of the query execution.

44. What are the benefits of ORM and Hibernate?

There are many benefits from these. Out of which the following are the most important one.

Productivity : Hibernate reduces the burden of developer by providing much of the functionality and let the developer to concentrate on business logic.

Maintainability :As hibernate provides most of the functionality, the LOC for the application will be reduced and it is easy to maintain. By automated object/relational persistence it even reduces the LOC.

Performance : Hand-coded persistence provided greater performance than automated one. But this is not true all the times. But in hibernate, it provides more optimization that works all the time there by increasing the performance. If it is automated persistence then it still increases the

performance.

Vendor independence : Irrespective of the different types of databases that are there, hibernate provides a much easier way to develop a cross platform application.

45. What is meant by full object mapping?

Full object mapping supports sophisticated object modeling: composition, inheritance, polymorphism and persistence. The persistence layer implements transparent persistence; persistent classes do not inherit any special base class or have to implement a special interface. Efficient fetching strategies and caching strategies are implemented transparently to the application.

46. What is a meant by medium object mapping?

The application is designed around an object model. The SQL code is generated at build time. And the associations between objects are supported by the persistence mechanism, and queries are specified using an object-oriented expression language. This is best suited for medium-sized applications with some complex transactions. Used when the mapping exceeds 25 different database products at a time.

47. What is a meant by light object mapping?

The entities are represented as classes that are mapped manually to the relational tables. The code is hidden from the business logic using specific design patterns. This approach is successful for applications with a less number of entities, or applications with common, metadata-driven data models. This approach is most known to all.

48. Why do you need ORM tools like hibernate?

The main advantage of ORM like hibernate is that it shields developers from messy SQL. Apart from this, ORM provides following benefits:

- ▶ Improved productivity
 - * High-level object-oriented API
 - * Less Java code to write
 - * No SQL to write

- ▶ Improved performance
 - * Sophisticated caching
 - * Eager loading

- ▶ Improved maintainability
 - * A lot less code to write

- ▶ Improved portability
 - * ORM framework generates database-specific SQL for you

49. What is a pure relational ORM?

The entire application, including the user interface, is designed around the relational model and SQL-based relational operations.

50. What are the different levels of ORM quality?

There are four levels defined for ORM quality.

Pure relational

object mapping

Medium object mapping

Full object mapping

51. What does an ORM solution comprises of?

It should have an API for performing basic CRUD (Create, Read, Update, Delete) operations on objects of persistent classes Should have a language or an API for specifying queries that refer to the classes and the properties of classes An ability for specifying mapping metadata It should have a technique for ORM implementation to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimization functions

52. What is ORM?

ORM stands for Object/Relational mapping. It is the programmed and translucent perseverance of objects in a

Java application in to the tables of a relational database using the metadata that describes the mapping between the objects and the database. It works by transforming the data from one representation to another.

53. What is Hibernate?

Hibernate is a pure Java object-relational mapping (ORM) and persistence framework that allows you to map plain old Java objects to relational database tables using (XML) configuration files. Its purpose is to relieve the developer from a significant amount of relational data persistence-related programming tasks.

1. In spring bean class uses singleton or prototype?

By Default spring uses singleton or mention singleton="true" for singleton else false for prototype inside bean tag.

2. What is prototype?

Having multiple instances or having clones

3. Explain servlet life cycle?

init(),service(),destroy()

4. What is jsp life cycle?

jspinit(),_jspService(),jspdestroy()

5. What is AOP(assepct oriented programing)?

Separating your business logic with other functionalit y such as services,Loggers etc,Making classess more cohesive.

6. What are the diff types of exception?

Unchecked and Checked exceptions

7. What are the oops concept?

Inhertitance, Encapsulation, Polymorphism, Data Abstraction

8. Explain spring framework?

Spring is lightweight,Inversion controlled,Aspect oriented ,Container framework.

9. What is IOC concept & explain it?

Injecting dependencies to object itself instead of depending on container.

10. Write a singleton program?

Having single instance through out the application.eg.:
Loggers

[Download Interview PDF](#)

11. Explain what is orm?

Object Relational Mapping ,its a tool for transaction management that needs to be integrated with Spring,Struts etc.Eg : Hibernate,iBatis,JDO etc

12. Write a program to show synchronization?

```
public someClass{  
    public synchronised methodA(){  
        //write your code  
    }  
}
```

13. Explain what is synchronization?

When multiple threads working, Synchronization is to lock a method for a particular object.

14. Different between Struts and Spring? or Why use Spring, if you are already using Struts?

Struts:

1. Struts is only for web Applications. We can not develop any type of Java, J2EE applications by using Struts Framework.
2. We can not Integrate other Frameworks with Any other Java Oriented Frameworks.

Spring:

1. Spring is for developing any kind of Java, J2EE applications.
2. It is Layered architecture. We can integrate any no of Frameworks with Spring.
3. It has So many features like AOP, IOC.

15. Does Java pass arguments by value or reference?

Pass by value.

1. When passing primitives, it passes a copy of the variable to the method. Any change made in the method

does
not reflect in the calling method.

2. When dealing with objects, a copy of their reference/address is passed. Thus the change made to the object is reflected in the calling method.

16. Will it be called overriding if I do not change the parameters or return type, instead throw a different exception in the method signature?

yes, you will be overriding to throw a different exception

17. How will the struts know which action class to call when you submit a form?

struts-config.xml in this file.

under the tag <type> absolute path of Action class

under the tag <name> name of the action form class

Both of this will be called when the desired action mentioned under the <path> tag will be called and struts-config.xml will call action class mentioned in the <type> tag and also populate the fields of form class mentioned under <name> tag.

18. If I define a method in JSP scriptlet <%..%>, where will it go after translation into a servlet?

It will give compilation error. In order to define a method in JSP, it should be defined in the JSP declarations `<%!..%>`, and it can be called in the JSP expression. On translation, the method will be added outside all methods in the servlet class.

19. Why use a datasource when you can directly specify a connection details? (in a J2EE application)?

Because, it would be really difficult to specify the connection details in every method that access the database. Instead, if we create a data source, it can be used to connect to the database in every method we want.

20. Can we have more than one action servlet?

yes you can have and if you specify different url patter like

- *.do for one action servlet
 - *.abc for other action servlet
- in your web.xml file

[Download Interview PDF](#)

21. What is difference between object state and behavior?

If you change the state of an object, you ask it to perform a behavior. An object stores its states in a field e.g. variables, and demonstrates its behavior through methods.

22. If i learn Java, what kind of applications can i create that will help Banking, Retail, Hotel, Logistics industry?

When learned the java with the advanced concepts ,the application can be created for all the domain.Using the J2EE will be more friendly and efficiency in the code maintenance, since part of the code will be from the framework

23. Java is fully object oriented languages or not? Why?

No,java is not fully object oriented language because it does not support "multiple inheritance" and "pointers" which are used in C++.But,by using Interfaces we can implement multiple inheritance.Also,due to presence of Primitive datatypes,which are used in (AutoBoxing)...we can say it is not fully object oriented language.

24. What we can not do in jdbc but can do hibernate?

There are many thing we can do in hibernate automatically by

hibernate tools or setting hibernate properties. some I giving below:

(1) We can migrate database just change database dielect.

(2)we can used caching

(3)if user do not know ant any SQL language then they can use criteria query.

(4)There is one scnerio where SQL query are failed when want fetch obejct from db but do not know there ID.

Hibernate provide this solution using HSQL.