**Questions on SpringBoot:**

# [Difference between @SpringBootApplication vs @EnableAutoConfiguration annotations in Spring Boot? Example](https://www.java67.com/2018/05/difference-between-springbootapplication-vs-EnableAutoConfiguration-annotations-Spring-Boot.html)

Even though both **@SpringBootApplication** and **@EnableAutoConfiguration** can be used to enable the **auto-configuration feature of Spring Boot,** there is a subtle difference between them. The @SpringBootApplication does much more than what @EnableAutoConfiguration does. It's actually a combination of three annotations: **@Configuration,** which is used in Java-based configuration on Spring framework, **@ComponentScan** to enable component scanning of components you write like [@Controller](http://javarevisited.blogspot.sg/2017/11/difference-between-component-service.html) classes, and **@EnableAutoConfgiuration** itself, which is used to allow for auto-configuration in [Spring Boot](https://javarevisited.blogspot.sg/2018/02/top-5-spring-microservices-courses-with-spring-boot-and-spring-cloud.html)application.

**What is @Query used for? (**[**example**](https://javarevisited.blogspot.com/2021/09/spring-data-jpa-query-example-tutorial.html)**)**  
Answer:  Spring Data API provides many ways to define SQL query which can be executed and Query annotations one of them. The @Query is an annotation that is used to execute both JPQL and native SQL queries.  
Here is an example of @Query annotation from Spring Data Application which returns all active orders from the database:

@Query("select e from Employee e where se.name = ?1")

List<Employee> getEmployees(String name);

**What is SQL?**

Ans. Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties. HQL queries are translated by Hibernate into conventional SQL queries, which in turns perform action on database.

Although you can use SQL statements directly with Hibernate using Native SQL, but I would recommend to use HQL whenever possible to avoid database portability hassles, and to take advantage of Hibernate's SQL generation and caching strategies.

Keywords like SELECT, FROM, and WHERE, etc., are not case sensitive, but properties like table and column names are case sensitive in HQL.

The **SELECT** clause provides more control over the result set then the from clause. If you want to obtain few properties of objects instead of the complete object, use the SELECT clause. Following is the simple syntax of using SELECT clause to get just first\_name field of the Employee object −

String hql = "SELECT E.firstName FROM Employee E";

Query query = session.createQuery(hql);

List results = query.list();

It is notable here that **Employee.firstName** is a property of Employee object rather than a field of the EMPLOYEE table.

**Eceptions ->**

* **IllegalStateException ->** Because of not writing @Entity in top of Entity class
* **org.hibernate.AnnotationException: No identifier specified for entity -> B**ecause of not writting the Annotations like @ID when the “id” variable, when you have declared ID as Primary Key in DB

difference between group and package:

makaut is a group ->com.demo\_1,

Package is a subgroup of main group, written like, com.demo\_1.crudexaple.

Artifact is project name that you want.

\*\*Earlier when we want to connect our Project to MySQL DB, then, we had to integrate JAR file, i.e.,MySQL connector, and we download it ,extract it and configure it. But that’s very painful. Because there are so many JAR File out there, like email API.

**API->** Any 3rd party software if we are integrating in my application to perform some specific task. Ex, email Sending API, or Payment Gate Way.

**MAVEN Dependencies:** It helps us to download Dependencies. We use “**pom.xml”** file and add the dependency codes to download dependencies. In “**pom.xml”** file dependency tags are used to download project Dependencies. It’s a big Repository for Developers containing some default JAR Files , and if we need some JAR like email API, MySQL Connector, then the Process is:-

* Go to Google & type **->** <https://mvnrepository.com/> **->** here search for required JAR Files like “MySQL Connector” **->** click on the latest Version **->** Copy the **Maven{<dependency>..…</dependency>}** Code.
* Now in STS Tool, click on “**pom.xml”,** which is used to communicate or **command Maven** to download any JAR File, defined in the {<dependency>…</dependency>} **Tags ->** Maven Only understands the code or **command** defined in <**dependency**> **Tags** which is pasted in **pom.xml file ->** After this we save “pom.xml” file, and STS Tool starts downloading the MySQL Connector JAR File inside it. So, easy.

How are you performing unit Testing in your Project:

Developers perform one type of Testing called the White Box Testing done at code level. We will check if the complete code is running fine by giving own values done by developers only.

Now all these kind of Testing done at code Level is achieved **Using Junit**, which is a framework.

**ANNOTATIONS:**

**@Test:** This Annotations helps to perform any number of Method Testing here. There is no need to manually call a method, nor any Object creation is required to run these @Test on a method. All these are done manually by "JUnit" Testing.

It's work is, to check the method code , for any error, without calling it. Just write the annotation above any method. If the method code runs successfully, the @Test will return **true**. If any line of code inside method fails, then the @Test Annotation will report a failure or false. The Job of @Test annotation here is to keep checking the code inside the test method. If is it running properly. If any line of code is not running properly, it will report a failure.

Now, we have special Annotation, with which we can control which method will be executed first OR last, etc., Below are few Examples:-

* **@Before:** This Annotation is used on that method which is to be executed first before all other methods.

Practical Use: Suppose we have one method for Inserting to DB, and another to delete. Now before this methods are executed, we need to execute the DriverConnection method for connecting to Database. In that case, we can use **@Before** Annotation before DriverConnection() method, to execute it before all other methods. Runs the method twice.

* **@After:** This Annotation is used on that method which is to be executed last after all the methods. Runs the method twice.

Practical Use: Suppose “**con.close”** for closing a DB connection, is kept in a method, and is to be executed at last, after all methods.

* **@BeforeClass: I**t’s an annotation used with only static methods to run a specific method only once Instead of twice before all other methods, even before @Before Annotation.
* **@AfterClass: I**t’s an annotation used with only static methods to run a specific method only once Instead of twice after all other methods, even before @After Annotation.

Assert.*assertEquals*(expectedValue, actual\_Value);//Asserts that two long values are equal. If they are not, an AssertionError is thrown. This method compares the expected and actual value, if they are same, they returns true, else false. Based on the value returned by the method, we decide if the test has passed or failed.

**Ex. Code:-**

**public** **class** AddTestAnnotation {

@Test //this annotation will automatically run this static method without Object Creation.

**public** **void** testAddNumbers1() {

Add\_Calc a1 = **new** Add\_Calc();

**int** actual\_Value = a1.add(10,20);

**int** expectedValue = 40;

Assert.*assertEquals*(expectedValue, actual\_Value); //Here an error is Thrown, as value is different, and "JUnit" thus fails to execute as error will be thrown inside the method.

}}

Class add{

Public int add(x,y){ return x+y; } }

Which unit Testing did you use in your Project?

Ans. Junit, a testing framework, for testing different methods with Annotations. If passed, then Junit is passed. Or else, if any error occurs, Junit fails to pass.

JPA -> Java Persistent Api Jakarda

Every Entity class should be same name as that of the database table. Like if Database Table name is “Student”, then class name is also “Student”, but it’s not **CaseSensitive**. Now, there are column name also like name, email, phone number.

Now in the Student class we create Private variables of the column names like name, roll no. Thus we achieve Data Hiding by preventing direct access from outside class. Here Encapsulation concept is used to fetch this variables.

Suppose DB Team gives an column name “s\_id” OR “name”. Now the column name and the name of private variables for determining the column should have same name. I can create a variable with the name as “sId”.Both meaning same, but don’t change anything more, else shows error.

Just like Oops concept used in c++, JAVA,etc.,

In same way **JPA** is also a concept used in “**Hibernate JPA”, “eclipseLink”, “OpenLink”.**

**Hibernate:** It’s an ORM(Object-Relation Mapping) Tool.

**ORM Tool:** It’s Object-Relation Mapping, i.e., we are creating Objects and storing data into it using setters, and then mapping the Object with the DBTable.

Spring Initializer-> It helps us to give a Spring Boot Project Structure inside Eclipse including pom.xml and the others.

**JPA** is used to map the Entity Class Object(containing row data) with the Database Schema, and fill rows of the Table.

To Perform CRUD Operation,, steps are:-

1. **Create Database Schema(**creating a table)
2. **Create Spring Boot Project(** JPA Dependencies**)//**In **SpringBoot**, **@Entity** is the Annotation for **JPA. @Entity** is responsible for mapping Objects with DBTable in **Spring**. But for activating the @Entity, we have to download things like:

* JPA Dependencies first-> (Implementing Hibernate JPA)
* then download MySqlConnector->
* then Create Entity Class(same name As DBTable)->
* Update Application.properties file-> mention username, url, password to connect to DB
* Create Repository Layer->
* Perform JUnit Testing->

**Create a Spring Starter Project -> Next-> Select “MySQL Driver“ && “Spring Data JPA Driver” ->**

**Spring Data JPA -> It gives you all Hibernate JARS and dependencies like @Entity, @Id, @GeneratedValue ( strategy = GenerationType.IDENTITY).**

**Which version of Hibernate you are using in your Project? Where in your Project you have used Hibernate?**

**Ans.** Using 5.6.1 Version of Hibernate. I have used Hibernate in my Project for mapping Objects, of a Student Entity Class, to the DBTable.

**Annotations☹**

3 Things we have to import first for JPA Package

i)import javax.persistence.Entity, for the class

ii)import javax.persistence.GeneratedValue, for incrementing Primary key ID automatically

iii)import javax.persistence.Id, for telling this is the Primary Key of our Table.

**@Id -> If we write this Annotation on a variable “private int id” then T**his Annotation means that variable is **PRIMARY KEY** and it is mapped to the Primary Key column of the Database Table.

**@GeneratedValue(strategy = GenerationType.IDENTITY) ->** Now we want the “id” section of column to be Autoincremented. So to make Spring Compiler know that we want “private int id” to be Auto Incremented and not manually done. Then the @Generated… Annotation is used on the top of that variable to denote “Autoincrement” of the DBTable in MySQL. Due to this **Annotation** Spring Compiler will automatically set values for the “id” section, while creating the Object, everytime.

**Where to write Database Codes for connecting to DB in SpringBoot:**

Now earlier to connect with the database, we had to write DriverManager.getConnection() and many things by creating a separate folder and separate method in JDBC. BUT, now in Spring Boot, there is a place inside SpringBoot Project folder, named by “Application.properties”, where we define all the database Connectivity Codes and that’s done. So easy nah!!

Spring will see the connectivity code and connects with the Database. Everything we write in “Application.properties” is shown in Key-Value Pair.

**Interview: How you are connecting the SpringBoot to the database?**

Ans. In “Application.properties” file, I will specify JDBC URL, username, password. SpringBoot automatically reads the file and connect with the Database.

**Repository Layer: ☹**

* **We will create two repositories i) JPARepositories ii) CRUDRepository. These two Repositories are Interfaces. Now we can create our own Repository Interface “StudentRepository” and extend it to CrudRepository<EntityType(Student), PrimaryKeyType(Long)>.**
* **Repository Layer It helps us to develop the application and has lot of built-in methods to perform CRUD Operation.**

**CREATING OBECTS: ☹**

**Earlier**, we used to create Object by creating an Object of class name, and then using setters to set the values of Entity Class to map to DB. **Now,** we use @Autowired.

Now In spring Boot, **Objects** are termed or called as **Beans.**

**Now In our code we perform something called as Dependency Injection**

**@AutoWired : Object creation is done using @AutoWired**, which is present in SpringBoot Package. When we have a class “StudentDemo” implementing “Student ”.Now a reference variable of Interface “private Student s1”, is created and write “@AutoWired” on it, Then Spring will search for that class which is implementing that Interface, here “StudentDemo”, and creates an object of the StudentDemo Type automatically and return it’s address to the Reference Variable.

Now reference variable of an Interface(Student) can be created. But Object cannot be created. But **SpringBoot is clever,** it will create an Object of Class “StudentDemo” because implements “Student”, and class Upcasting Occurs, and returns the address of StudentDemo to the reference variable.

Ex. Interface A{}

Class B implements A{}

A a1 = new B();

**Spring IOC?? Dependency Injection????**

An IoC container is a common characteristic of frameworks that implement IoC. In the Spring framework, the interface ApplicationContext represents the IoC container. The Spring container is responsible for instantiating, configuring and assembling objects known as beans, as well as managing their life cycles.

**BEANS:** In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called **beans**. A bean is **an object that is instantiated, assembled, and otherwise managed by** a Spring IoC container. Otherwise, a bean is simply one of many objects in your application.

**What is Spring Container?**

Ans. The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring container uses DI to manage the components that make up an application. These objects are called Spring Beans, which we will discuss in the next chapter.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram represents a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.

In “application\_context.xml” File or a Container we were defining the Bean Codes with <bean> Tags. Then inside that Bean code, we were alos defining the <property> Tags and <Constructor type=”int” value=”5”> Tags..

Now inside our Controller class, we call the ApplicationContext reference variable and create an Object of “ClassPathXmlApplicationContext(“Application\_context.xml”). Thus all Bean Configuration is done.

Spring Container initializes the instance with the values, and then init() method is called where different database initialization or connectivity codes are written. Destroy() method will destroy the Bean.

**Autowiring** feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection. **Autowiring** can't be used to inject primitive and string values. It works with reference only.

The Dependency Injection is a design pattern that removes the dependency of the programs. In such case we provide the information from the external source such as XML file. It makes our code loosely coupled and easier for testing.

**In Hibernate, it works on the concept of Serialization and Deserialization.**

**Interview Question: Where Serialization and Deserialization have you seen practically?**

**Ans**. In hibernate. When I am injecting or Inserting the Object Content to the Database, Serialization will take place. When I am taking the Object of DB(Containing row and putting that into an Object, I am doing Deserialization.

@Test

**void** saveStudentInfo() {

//Inserting the Record

Here Serialization and Deserialization taking place. First "Serialization" happens **while** Inserting Data ->Here the Objects(containing Data to be Inserted) are converted into Binaries -> then Binaries to Database Content.

Student s = **new** Student();//creating a Student Entity class Object to access the Setter methods, to initialize into it.

s.setName("Bhaskkjhar");//calling the setName() method in Student Class

s.setCourse("Autosadmation Tester");

s.setFee(20320);

//Now the Object Address which "studentrepo" reference is holding, comes into picture. That Bean(Object) contains a method save().

//save() is used to take all the content of the "Student" Entity Class and put them in the databaseTable. Behind the scene, Hibernate internally creates the Sql Query for Insertion.

studentRepo.save(s);

}

//Read the Data from DBTable needs Desiarialization

@Test

**void** getStudentInfo() {

//Here Deserialization taking place. "Deserialization" while fetching Data from DB -> Database Content are converted to Binaries -> then Binaries are converted to Objects->That Object is converted into Student Object from Optional Object-> then use the “Student type reference and use get() methods to print them.

Optional<Student> findById = studentRepo.findById(1L);//Retrieves a row with the given ID(Long),by putting the row, after Deserialization, in an "Optional type" Object and returning the address -> Now that object

Now the reference "findById" contains the address of an Object containing row Data.

}

**Where have you seen Optional Class while implementing Project?**

I have used it In SpringBoot Hibernate JPA, where when I had to read and fetch a row from the DB , by storing into an Object. The Address value is returned and stored in the Reference Variable of “Optional<T> Class”.

Inside Optional reference variable I hold the Object Address containing the row. Now I use get() method of Optional class to fetch the content(Optional[com.demo2.studentEntities.Student@338270ea]) inside the “Optional[]”. And store it into the Student type reference variable and get to access all the values.

InterView Question: What is Auto Increment? Without AutoIncrement how will you create the Database.

**InterviewQuestion: How you are using Hibernate in your Project?**

**Ans. First I was Adding “Spring JPA Jar” Dependencies in my SpringBoot. Then I create an entity Class, then I update Application.properties File. And thatshow Integration was done. And when we write save() method, then internally Hibernate runs a query to Insert or Update values.**

**CREATING WEB APPLICATION USING SPRINGBOOT:**

STEPS:

1. Create Database Schema -> Location
2. Create project with JPA, WEB, MySQL Dependencies -> Spring WEB, Spring Data Web, MySQL Driver
3. We create Entity Class -> Locations (@Table = Location)
4. Update Application.properties File -> Creating Driver
5. Create View Layer -> src/main/webapp/WEB-INF/view/JSP File(saveLocation.jsp)
6. Create Controller Layer(Containing Servlets, and corresponding doGet() or doPost() methods)
7. Create Repository Layer
8. Create Services Layer
9. Add Embedded Jasper Tomcat(For running Hibernate, Springboot we configure it)

Without any configuration Spring Boot expects the views to be stored inside /webapp, the view page may be of any format depends on application.properties settings(like html or jsp) to set .jsp as view page at /views/ folder

spring.mvc.view.prefix=/views/

spring.mvc.view.suffix=.jsp //for .html change it to .html

“spring.mvc.view.suffix = .jsp” This Line is written in “application.properties”, which means all the view that you are developing written in “**return** "saveLocation" should have the extension “.jsp”. So, all the files we will create as the view will automatically have the extension of “.jsp”. We don’t have to write it everyTime

spring.mvc.view.prefix = /WEB-INF/jsps/ -> tHis Line means that all the jsp files written in “**return** "saveLocation" will have the paths as the Link above. It’s extension is denoted by suffix.We don’t have to write the Path everytime.

JpaRepository<Locations, Long> -> It’s an advanced version of CRUDRepository, containing more features

From where SpringBoot Execution Process Begins in Project?

Inside the Project named “locationweb2” there is “src/main/java” -> which contains “com.location” package -> Inside it there is a Java file named “Locationweb2Application. This Java file is automatically created. Now, from this file begins the execution and it’s the starting point. It’s marked with Annotation “@SpringBootApplication”,.

“@SpringBootApplication” -> That defines the starting point of execution in SpringBoot Projects. It’s like main() method.

When we write “localhost:8080/saveLocation” it’s the frontend Page -> From here we will go to Controller Class denoted with @Controller Annotation -> Inside that class contains small Servlet denoted by “RequestMapping(“/saveLocation”)” , acts like @WebServlet in JDBC, used for mapping frontEnd with Servlet->then call it’s corresponding method -> “return” statement inside that method acts Like Request Dispatcher and takes us to the Front EndPage inside Quotation (return “saveLocation”)

@RequestMapping(value = "/saveLocation", method = RequestMethod.***GET***) -> This GET method should be called only when the value = saveLocation is found in URL= <http://localhost:8080/saveLocation>.

**SPRING MVC:**

We cannot create an Web Application until we use MVC Architecture. You have to segregate the layer. The Controller Layer has to defined with the notation @Controller . Only Controller Layer can interact with the view(containing JSP Files). JSP files are Present in src/main/webapp/WEB-INF/view/”JSP File”. When I submit the data from the View, then the Controller will be called. Then the Controller will interact with the Database to perform CRUD on the data.

As described in the Spring MVC documentation - the **@ModelAttribute** annotation can be used on **methods** or on **method arguments**. And of course we can have both use at the same time in one controller.

**2.Method argument**

public String findPerson(@ModelAttriute(value="person") Person person) {

//..Some logic with person

return "person.jsp";

}

When the form is called/initialized their getters are invoked. On form submit their setters are invoked, and their values are transferred in the bean**(Object)**

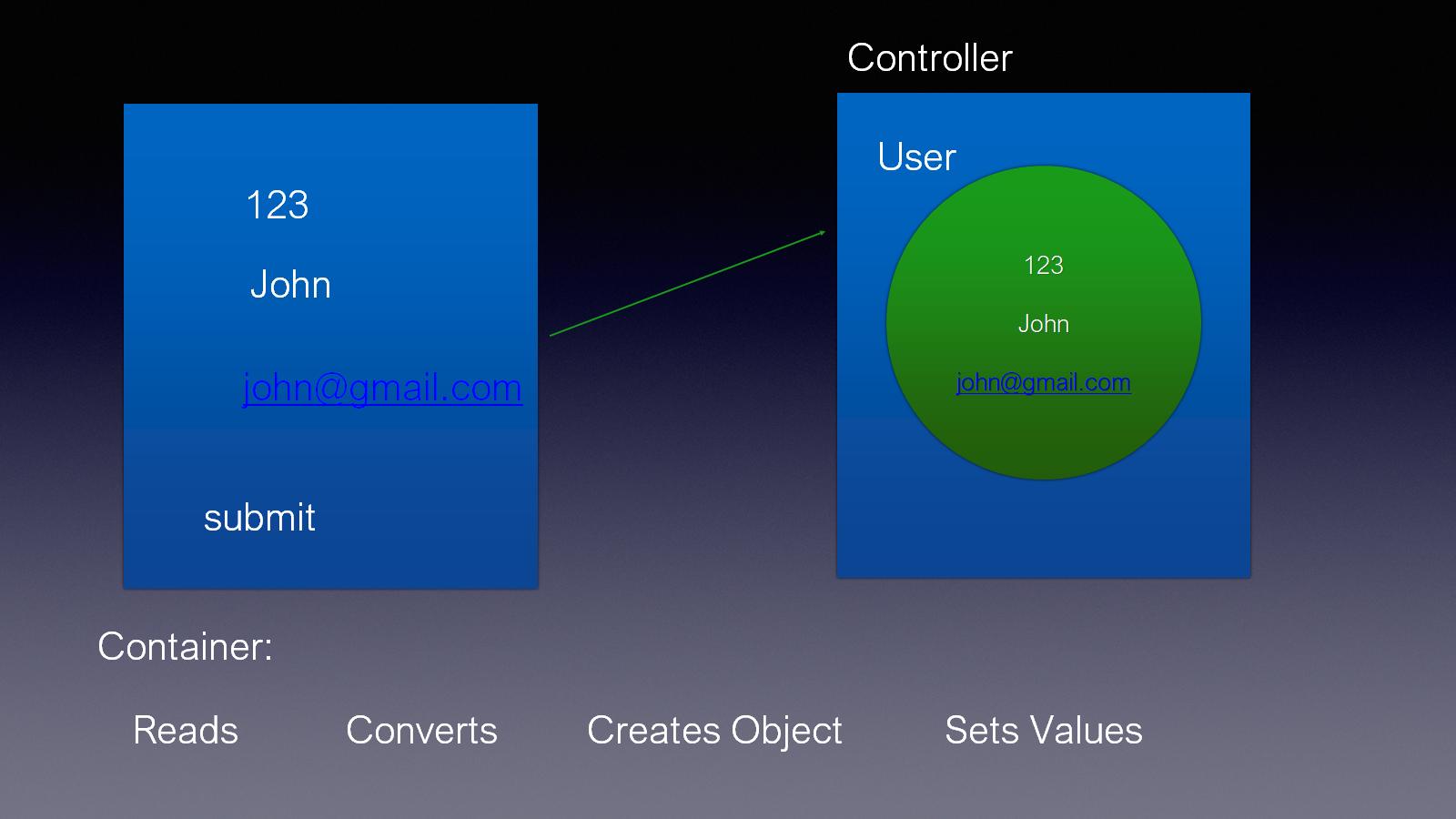
An @ModelAttribute on a method argument indicates the argument should be retrieved from the model. Then will take the request parameters and try to data bind them in the Person object using their names.

What we are interested here is how the data is communicated from the UI to Controller. This can also be done in 2 ways:

1. Using an HTML Form
2. Using Query Parameters.

Using an HTML Form: Consider the below scenario,

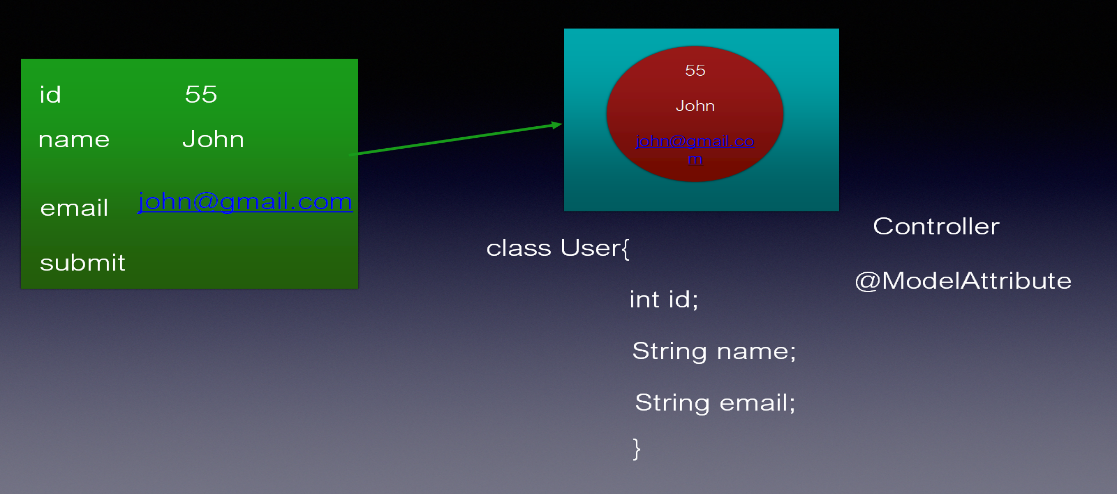
**Here “User” Entity will be replaced by “Location” Entity.**

[](https://i.stack.imgur.com/fLdHj.jpg)

When we submit the form data from the web browser, we can access that from data in our Controller class as an object. When we submit an HTML form, the Spring Container does four things. It will,

1. first read all the data that is submitted that comes in the request using the **request.getParameter** method.
2. once it reads them, it will convert them into the appropriate Java type using **integer.parseInt**, **double.parseDouble** and all the other parse methods that are available based on the data type of the data.
3. once parsed, it will create an object of the model(**Entity)** class that we created. For example, in this scenario, it is the user(Location) information(like id,name,code,type) that is being submitted from the **Form**. We had the Entity Class as **USER**(Location) and **Spring Container** will create an object of the class and set all the values received from the Form that enters automatically into that object.
4. it will then handover that object by **setting** the values with **SETTERS** to the Controller Class.

To get this whole thing to work, we'll have to follow certain steps.

[](https://i.stack.imgur.com/38qzM.png)

We first create a model(Entity) class, like User(Location), in which the number of fields should exactly match the number of fields in the HTML form. Also, the “name” field that we use in the HTML form should match the variable names present in the Java class. These two are very important. Names should match, the number of fields in the form should match the number of fields in the class that we create. Once we do that, the **SpringContainer** will automatically read the FORM data that comes in, creates an object of this model, sets the values and it hands it over to the Controller. To read those values inside the Controller, we use the **@ModelAttribute** annotation on the method parameters. When we create methods in the Controller, we are going to use the @ModelAttribute and add a parameter to it which will automatically have this object given by the Container.

Model means Entity Class, Attribute means the Model Object, it will initialize this Object Attributes now. When I click on **SUBMIT BUTTON** -> @ModelAttribute(“locate”) -> then automatically one Entity class “Location” Object is created and values are Set Inside the Object using **SETTERS**, and the Address will be stored in the **key** variable “**locate**”, or **null also,** inside **@ModelAttribute(“locate”).** This @**ModelAttribute()** is binded with the Method Arguments like “@**ModelAttribute**(“locate”) **Locations location”.** Then that Address is further copied to the **Reference Variable(Method Argument=“Locations location”)**  **->** Now using this Reference variable **“location”,** we can fetch the form Data by calling the **“Getter” M**ethods like getName() or getType().

It takes the form data and initializes entity object with the data

It takes the form data and stores the data into the Entity Class Object.

The Entity class Object will be further given to the Method Argument.

**Interview Question: Where you use Encapsulation practically in Project?**

**Ans.** While Creating an Entity Class(Location), there I have used Encapsulation, while submitting a form, all the data of the Form are set into Object Using **SETTERS** and address of the Object is sent to @ModelAttribute(“locate”), and the **Reference** Variable(**Locations location**), which is binded with @ModelAttribute(“locate”), is used to fetch the Data from Object using **GETTERS.** There I have used **Encapsulation.**

**\*\*\* ENCAPSULATION also occurs internally in DTO, we will see it below..\*\*\***

**Interview Question: Where you use Inheritance practically in Project?**

**Ans.** I have used it in Services Layer of “Location” Project. I have created an Interface named “**LocationService”**  and created another class named “**LocationServiceImpl” .** Now I am **Inheriting** the “**LocationService” Parent Class** inside “**LocationServiceImpl”** Child Class by **implements keyword.**

**Interview Question: Where you use Polymorphism practically in Project?**

Now I am **Inheriting** the same incomplete Method “**saveLocation**” of “**LocationService” Parent** and completing them inside “**LocationServiceImpl”** Child Class. Same Method having two Different Forms. So **Polymorphism** occurs here.

In SpringBoot, Service Layer is the Model Layer, where business Logics are to be defined.

**There are 3 ways to Read the Form Data:**

* One is by using @ModelAttribute(“”) and binding it with Method Argument Reference variable “EntityName variableName”.
* Another way to read Form Data is by @RequestParam(“”) and binding it with an Argument of Variable Name like “long id”. **Example Shown below:-**

ii) Another way is by using @RequestParam("") and binding it with Argument Variables. Each form Data is fetched and stored into their corresponding @RequestParam("") and later copied to the corresponding Arguments. Ex. “id” fetched from the form is stored into its corresponding @RequestMapping(“id”) and later copied to its corresponding binded variable “long id”.

@RequestMapping(value ="/saveLocationData", method = RequestMethod.***POST***)

**public** String saveLocationData(@RequestParam("id") **long** id, @RequestParam("codes") String codes, @RequestParam("name") String name, @RequestParam("type") String type ) {

// Now after the value is fetched directly from the form data int Argument of this method, we create an Object of Locations class, and now set the data received from the form into the variables of Locations Class using SETTER Methods.

Locations location = **new** Locations();

location.setId(id);

location.setCodes(codes);

location.setName(name);

location.setType(type);

locationService.saveLocation(location);

**return** "saveLocation";

**modelmap.addAttribute(“keymsg”,”message body”) : Now if we want to show any message to the “view”, after data is inserted to DB, then we use reference of “ModelMap modelmap” of ModelMap Class. “modelmap.addAttribute()” has same functions as that of “request.setAttribute()” to create a “message” with a “key”.**

**${} -> Spring Expression, used in .jsp files to print the message Body using “keymsg” by writing “${keymsg}.**

**What is Data Transfer Object(DTO)?**

**Ans. It’s a technique in which we take the Form Data, and then put that into the Java Class Object, and then that Object Data we do an ORM to store it to the Database.**

**DTO acts as a middle layer and helps to communicate from the Client Side to the Server and from the Server back to the Client Side.**

**For Ex. If a client sends a request for inserting User Information, then it is first send to the UserDTO Class for mapping with the variables, and the address is sent to the “ShowLocation**(LocationData locationData)” reference inside **ShowLocation**() inside Controller. Then we get those Values and then again set those Values after creating “Location location = new Location” Object, and using “location” reference we set those values as “location.setId(locationData.getId())“, and send them to DB.

Again a **CLIENT can send a request for fetching some values, so there first values will be fetched from DB and then we set those values, and send them to the front end.**

**@ModelAttribute:** Here process of **ENCAPSULATION** is happening Internally. **Here we are taking the Data from the Form ->creating an Object of DTO Class “LocationData” -> Setting it’s values using SETTER methods -> returning the address of the Object to the Reference Variable “**LocationData locationData” **present as an argument in showLocation() method in LocationController:**

**public** String **ShowLocation**(LocationData locationData) {

**Now These data received from the DTO Class stored inside “**LocationData locationData” reference is mapped with the Entity Class “Location” as shown below:

Location location = new Location();

location.setId(locationData.getId());

location.setCodes(locationData.getCodes());

//Here we bind or map the values(code, name, type) of “locationData” reference inside “location” reference and then send it to Database.

**JSTL(JSP Standard Template Library):**

**Defination**: **It helps us to write JAVA Code in the form of Tags.**

JSP Standard Tag Library represents a set of Tags to simplify the JSP development. Earlier we used JSP so that we can use both html and Java codes together, but the Java codes inserted were much bulky size, and code lines is **reduced** in **JSTL. JSTL** are just like Html Tags

**Printing “Output in JSP” -> <%=10%>**

**Printing “Output in JSTL” -> <c:out value=”10”> </c:out>**

**Interview Question: Where are we using DataStructure in our Project?**

Ans. When we are fetching Data from the DataBase using findAll() method, then multiple rows are fetched and each rows of the tables is stored inside object and multiple Address is returned. So to store this Multiple Objects of Locations type, we use “List<Location>”, for storing them.

**InterviewQuestion: Why in Project we are using List ?**

Ans. Because the sequence in which we are reading data from the table, In that same sequence we need to fetch that Data also, and we know List follows the same Insertion Order.

**InterviewQuestion: Where do we use List and Collection Practical Example?**

Ans. List is Collection, and Collection are Collections of Objects. Now when we fetch Data from the DBTable then, the each rows are stored in single Objects. Now if there are 4 rows, 4 Objects will be created and each rows will get one Object, and the Address is returned to the List. When we print the List, we get 4 Object Address. **Shown Below:**

[com.Location.entity.Locations@1ec86b06, [com.Location.entity.Locations@81a9f6e](mailto:com.Location.entity.Locations@81a9f6e), [com.Location.entity.Locations@8301c](mailto:com.Location.entity.Locations@8301c), [com.Location.entity.Locations@11d154f9](mailto:com.Location.entity.Locations@11d154f9)]

**METHODS OF JPA Repository:**

1. findAll() method is a method of LocationRepository which is extending JPARepository tp fetch all the Data from the DBTable.
2. deleteById(id) -> is a method which is used to delete a row from Table with respective “id”.
3. findById(id) -> is a method to find a particular row from the DBTable.

**WebServices:**

**1st Step:** Whenever any flight details is added in AirAsia application, then it will generate an **XML File**.

Difference between XML & HTML?

HTML has predefined Tags, like Html Tags, Input Tag, Break Tag.

XML Tags are user-defined, which means we can give the TagName, the way we want to give it. And now what happens is, whenever we add the flight details in “AirAsia Application”, then those files will also be stored by us in XML File. **Why storing XML File only?** Because **XML Files**  are platform Independent.

**What does that mean?** Whateveryou put in Xml file, like any programming Language or any Hardware Configuration or any details related to Booking flight, **XML File** can understand that and can be used with any other Applications(makemyTrip) built with Different Language. So, after AirAsia(.net) Application is copied into XML, can now be integrated with any Application.

\*\*XML File can be understood by every Application made with Different Languages.

<flight-details>

<!-- this are self-made tags defined by us. -->

<!-- metaData defines what kind of data it is contained inside this Tags.-->

<!-- Like inside <flight-details>> Tag we will have flight related Details -->

<operating-airlines> <!-- "operating-airlines" Tags is also a meta data containing AIRlINES nAME -->Air Asia</operating-airlines>

<departure-city>Chennai</departure-city>

</flight-details>

**DEFINATION:** When we are exchanging Data between two Heterogeneous Application made with two different Languages, Then that Heterogeneous Data will do it with the concept of **WEBSERVICES**. WebServices helps us to integrate both Heterogeneous and homogeneous application like two Application made with same **Programming** language or different Language.

**@RestController** is a convenience annotation for creating Restful controllers. It is a specialization of @Component and is autodetected through classpath scanning. It adds the @Controller and @ResponseBody annotations. It converts the response to JSON or XML. It does not work with the view technology, so the methods cannot return ModelAndView. It is typically used in combination with annotated handler methods based on the @RequestMapping annotation.

The @Controller annotation is used with the view technology.

### Spring RestController annotation is a convenience annotation that is itself annotated with [@Controller](https://www.journaldev.com/21515/spring-controller-spring-mvc-controller) and @ResponseBody. This annotation is applied to a class to mark it as a request handler.

### Spring RestController annotation is used to create RESTful web services using Spring MVC. Spring RestController takes care of mapping request data to the defined request handler method. Once response body is generated from the handler method, it converts it to JSON or XML response.

Difference between @Controller and @RestController?

**with @RestControler**  
@RestController

public class Book{

@RequestMapping(value={"/book"})

public Book getBook(){

//...

return book;

}

}

**without @RestController**  
@Controller

public class Book{

@RequestMapping(value={"/book"})

@ResponseBody

public Book getBook(){

//...

return book;

}

}

Read more: <https://javarevisited.blogspot.com/2017/08/difference-between-restcontroller-and-controller-annotations-spring-mvc-rest.html#ixzz7HJ622O2y>

WEBSERVICES is of 2 types:-

1. SOAP -> Here we exchange Data between 2 different Applications using XML File. Disadvantage -> Implementing SOAP is much complex because Parsing or Reading XML Files is difficult to implement.
2. REST -> Using REST we can achieve WEBSERVICE Technic of converting many updates to JSON/XML File. Here we came up with something called JSON(Java Script Object Notation) Object with which we exchange Data between 2 applications -> It’s lot easier. Here the content will be stored as a key-value Pair.

REST also supports exchange of Data using XML File.

JSON Syntax -> In key-Value Pair, and can store any type of value(StringArray,Object,Boolean,String,number), and keys must be “String” only. Key & Value Separated by “:”.

{

“id” : 1,

“code” : ”KA”,

“name” : “Karnataka”

“Bikes” : {“Himalayan”,”Igniter”,”Unicorn”} //String Array

}

Now How WEBSERVICES works?

Ans. AirAsia File is exposing Data to the XML File, and “makemyTrip” is consuming XML File Data.

**WEBSERVICES** are called an API, because, whenever we Integrate a 3rd party Software into Our Application then it becomes an API. Now when for “AirAsia” we have some updates like Flight Update Details ,we Integrate those Updates into “MakeMyTrip” using XML. So for “MakeMyTrip” , “AirAsia” is a 3rd Party API, which is to be Integrated inside “MakeMyTrip”.

## @ResponseBody

The @ResponseBody annotation tells a controller that the object returned is automatically serialized into **JSON** and passed back into the HttpResponse object.

Next, the associated controller can be implemented:

@Controller

@RequestMapping("/post")

**public** **class** **ExamplePostController** {

@Autowired

ExampleService exampleService;

@PostMapping("/response")

@ResponseBody

**public** ResponseTransfer **postResponseController**(

@RequestBody LoginForm loginForm) {

**return** **new** ResponseTransfer("Thanks For Posting!!!");

}

}

In the developer console of our browser or using a tool like Postman, we can see the following response:

{"text":"Thanks For Posting!!!"}

**Remember, we don't need to annotate the @RestController-annotated controllers with the @ResponseBody annotation** since Spring does it by default.

***@RequestBody***

Simply put, **the*@RequestBody* annotation maps the *HttpRequest* body, coming from POSTMAN or other Application, to a transfer or domain object(Location Entity Object), enabling automatic deserialization** of the inbound *HttpRequest* body into a Java object.

First, let's have a look at a Spring controller method:

@PostMapping("/request")

**public** ResponseEntity **postController**(

@RequestBody LoginForm loginForm) {

exampleService.fakeAuthenticate(loginForm);

**return** ResponseEntity.ok(HttpStatus.OK);

}

## ****Query Parameter vs URI Path****

## @RequestParam and @PathVariable can both be used to extract values from the request URI, but they are a bit different.

While **@RequestParams extract values from the query string, @PathVariables extract values from the URI path**:

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

@ResponseBody

**public** String **getFooById**(@PathVariable String id) {

**return** "ID: " + id;

}

Then we can map based on the path:

http://localhost:8080/ getLocation /2

----

ID: 2

And for @RequestParam, it will be:

@GetMapping("/getLocation ")

@ResponseBody

**public** String **getFooByIdUsingQueryParam**(@RequestParam String id) {

**return** "ID: " + id;

}

which would give us the same response, just a different URI:

<http://localhost:8080/getLocation?id=3>

----

ID: 3

Spring automatically deserializes the JSON into a Java type value, assuming an appropriate one is specified.

By default, **the type we annotate with the *@RequestBody* annotation must correspond to the JSON sent from our client-side controller:**

**POSTMAN ->** It’s used to supply data in JSON Format inside Database.

**InterviewQuestion: Which WEBSERVICES Tool did you use to test your WEBSERVICES? How you were testing REST Webservices?**

Ans. I was using **POSTMAN.**

**InterviewQuestion: What Is the use of @RequestBody Annotation**

**InterviewQuestion:** Which all API have you used in your Project?

Like Email API, which someone has already created and we Integrate it into our Appilcation to send Emails.

@RestController defines it's webservices Layer, when I call this "getAllLocations()" method it will interact with database, get me all the rows, each rows stored into one java objects and all Addresses are returned and stored inside a List. Java object is then converted into JSON object and displayed in your browser.

**Difference between @Controller and @RestController?**

**Ans**. @Controller defines it is the controller layer of the Project, responsible to interact with the view and the backend buisness logic.**abstract**@Rest -> the webservices layer in your project.

**@GetMapping->** It’s used to handle **Get Requests** only. It also converts all the java Objects containing rows, received using **findAll()** into **JSON** Objects. Json Objects contains Key-Value Pair. WEBSERVICES is nothing but we are creating an URL(http://localhost:8080/locations) and putting it inside URL box to interact with the Database

The method-level annotations may override the default values by providing their own set of values.

For example, in *HomeController.java*, *@RequestMapping* annotation at line no. 4 provides the default values of produces attribute. It means that all the request handler methods in this controller class will, by default, will return the JSON response.

But addMember\_V2() method at line no. 12, overrides the produces attribute and it will return the XML response to the clients.

Note that addMember\_V1() method will produce the content in default media type i.e. application/json.

**HomeController.java**

package com.howtodoinjava.web;

@Controller

@RequestMapping(path = "/", produces = MediaType.APPLICATION\_JSON\_VALUE)

public class HomeController{

@PostMapping(path = "/members")

public void addMember\_V1(@RequestBody Member member) {

//code

}

@PostMapping(path = "/members", produces = MediaType.APPLICATION\_XML\_VALUE)

public void addMember\_V2(@RequestBody Member member) {

//code

}

}

***MICROSERVICES:***

**DEFINATION:** Here we break the bigger application into smaller mini projects and then we establish the communication between them using WEBSERVICES. **Practical Ex.** Suppose we create a Hospital Management System, and there every module is splitted into **MicroServices** and each MicroService Module have different functionalities. Like one Module is “**Registration”** for inseting the names of Patients in DB. Another Module is “**Medicines”,**  where a nurse wants to see which medicines are assigned to a patient with ID=5. Now **Medicine** Module will communicate with the **Registration** Module using **WEBSERVICES”** to fetch that patient’s id present in the Registration Database .

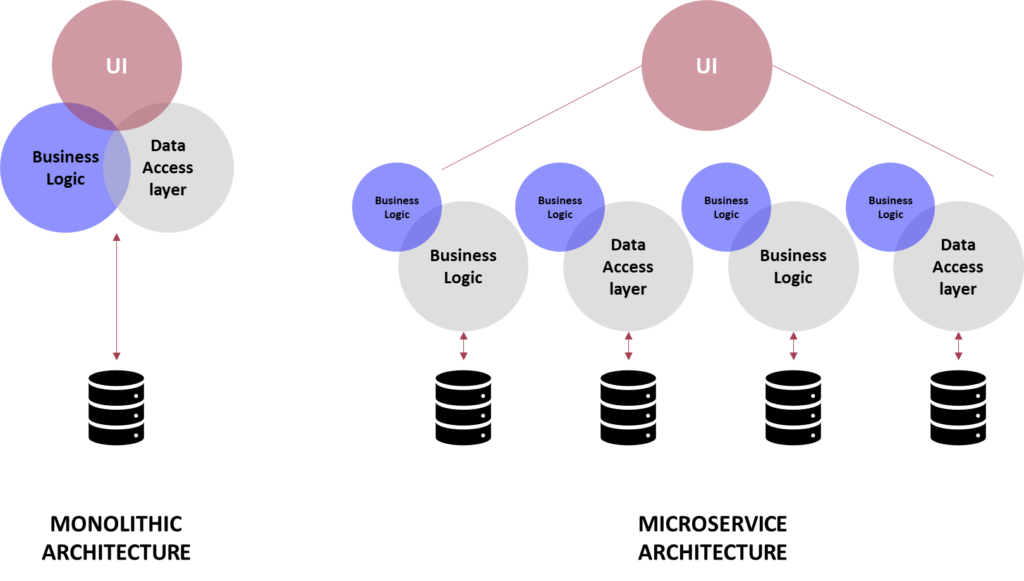
In “**Bed Allocation”** Module, we book a particular Bed for a Patient, and fetch the details again from “**Registration”** module.

In “**Reports”** we connect with the “**Registration”** Module to fetch the reports of a particular Patient with id=5.

Suppose we have a **Hospital Management System.** Now we have different **Section like i) Registration(PID) ii) Bed Allocation iii) Reports iv) Medicines.**

Now earlier we used to create **MONOLITHIC ARCHITECTURE** where we add all the Sections of our Application in same place like **UI, Business Logic or Service Layer,** and **Data Access Layer**. And the code used to get very lengthy and bigger, so maintenance becomes challenge. And also if our Application runs only on one  **SERVER.** Then it will arise many Complexities like whole of Application will be on one server. Now if Server is down someday, then whole Application will not run.

In **MICROSERVICES**  we divide the Bigger Application into smaller mini parts. Each parts will run on Different Servers and having their own Database.



**InterviewQuestion:What is the Difference between MicroServices and Monolithic Applications?**

**Ans.**

**DOUBTS, CHECK BROWSER TOMORROW:** Exposing WEBSERVICES and developing end Points in WEBSERVICES: End-Point URL means with this URL now I can get the record one by one from the Database. Where we will be inserting an ID with the URL and based on that we will be getting records rom DB.

server.port = 9090 //you run this project on tomcat by creating a new instance of TOMCAT on port 9090. Now two TomCat server will run on two different ports, and we can host two different Application Projects on those 2 servers.

Also add: <dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<version>1.3.156</version>

</dependency>

**Microservices Project Explanation ->**

Firstly in our first application named “**LocationWeb20-MicroServices**” , we had created a file named “**LocationRestController**” where in that file we had a method named “**getLocation(@PathVariable long id)”** whose work is to fetch a row from DATABASE with given ID, and return the row, in form JSON Format due to presence of @RestController Annotation, back to the URL Browser(**https://localhost:8080/ getLocation**) which send this request for fetching. Now this is E**XPOSING** of Data is happening over here. The Application named “LocationWeb20-MicroServices” sends data or response to another Application by using WEBSERVICES i.e., exposing data in JSON format, and send it to “searchLocation20MicroService” Application.

Now, the 2nd Application “searchLocation20MicroService” sends a request from the UI(SearchLocation.jsp) for an ID=5 suppose. Now, we will go to the Controller “SearchLocation”, and then create an Object of “LocationRestClient” and call its getLocation() method. Now we will create an inbuilt class “RestTemplate” and use its method named “getForObject (URL, Class<T> responseType)“ to call the controller method (“getLocation”+id) with an ID bind to it.

Now, that “URL” **CONSUMES** the data(JSON Content) fetched from the 1st Application. And return it to controller. And controller then returns it to VIEW using “ModelMap ModelMap”.

Here a **DTO Class “Location”**  is used for mapping the JSON properties with the variables inside DTO class using SETTERS.

**getForObject(URL,)-> “URL”** is the1stargument which we want to call. It contains The names of these methods clearly indicate which HTTP method they invoke, while the second part of the name indicates what is returned. For instance, getForObject() will perform a GET, convert the HTTP response into an object type of your choice, and returns that object. postForLocation will do a POST, converting the given object into a HTTP request, and returns the response HTTP Location header where the newly created object can be found.

Now this URL will call another **Microservice Application**, to fetch **Location** row from DB and return a String of **JSON** Format. Now another argument “ is for the DTO class where the properties will be defined using “**Enacapsultation” ,** and contain only those properties which is present in JSON(**as KEY-Value pair)**

Now that 2nd argument will initialize its properties with the values received ass JSON Format from URL, and initialize them with SETTER Methods and returns a fully fledged Object containing all the properties(id, name, type, code) with their values(“1”,”Bapi”,”Male”,”007”) and the address is returned to the reference(Location locations) of type Location. Now we can just use the reference and get the values with GETTERS.

CODE: **public** Location getLocation(**long** id) {

//RestTemplate is a built in class used for consuming WEBSERVICES. This is the class that will take the JSON Content and put that in JAVA Object

//In its argument, there is a place of URL, which was used to fetch the JSON content from our 1st Application. And that content wil be copied to the next argument which is the Location Object

RestTemplate restTemplate = **new** RestTemplate();

Location locations = restTemplate.getForObject("http://localhost:8080/getLocation/"+id, Location.**class**);//"getForObject will automatically fetch the JSON Content and copy that to the "Location.class" DTO Object Argument, and the Location Object is what we get here.

**return** locations;//now after the Data types of Location DTO is set ,using SETTER, with the values of JSON fetched from the URL.

}

**EMAIL SENDING API**

spring.mail.host=smtp.gmail.com //This is the Host where Gmail is running just like Localhost

spring.mail.port=587 //That’s the port number on the server where it is running.

**Below Lines should be declared inside Application.properties**

* spring.mail.username =<login user to smtp server> debanjansarkar9981@gmail.com spring.mail.password = <login password to smtp server>(Password of Gmail)
* spring.mail.properties.mail.smtp.auth=true spring.mail.properties.mail.smtp.starttls.enable=true

**InterviewQuestion: Which API have you used in your Project?**

**Ans.** Email API I have used for sending mails in my Project.

**IMPORTANT CONCEPT TO KNOW FOR INTERVIEW:**

Core Java, SpringBoot, WEBSERVICES, Hibernate JPA.

**SPRINGBOOT ANNOTATIONS DEFINATION:**

* **@AutoWired:** It helps us to perform Dependency Injection
* **@Controller :** It helps us to define Controller Layer in SpringBoot Projects which is responsible to interact with view and the BackEnd business Logic.
* **@RequestMapping:** It binds the URL/Forms with Controller Layer. We enter the URL and it calls the method in the Controller Layer. Next, we develop a Form, and the action Attribute of the Form calls the Controller Methods. That Binding work of the URL and the form happens with help of @ResquestMapping(“”).
* **@ModelAttribute:** It binds Form Data with Entity Class Object.
* **@RequestParam:** It takes the form Data and initializes Controller Method Arguments.
* **@ModelMap :** It’s a **in-built class** and not an Annotation. It helps us to exchange the Data between Controller and View Layer.
* **@Service:** It helps us to define Services(Model) Layer, containing Business Logic, in our SpringBoot Project.
* **@Component:** It is a stereotype**.????? !! RESEARCH MORE**
* **@DateTimeFormat: ????**
* **@Query: ????**
* **@Param: ???**

**WEBSERVICES ANNOTATIONS:**

* **@RestController:** It defines **WEBSERVICES** of our Project.
* **@PostMapping:** It helps us to store the Data in Database using **WEBSERVICES.**
* **@PutMapping:** It helps us to update the Data in Database using **WEBSERVICES.**
* **@DeleteMapping:** It helps us to delete a record in Database using **WEBSERVICES.**
* **@GetMAPPING :** It helps us to read all the Data from the Database using **WEBSERVICES.**

**InterviewQuestion: What is Hibernate? Which version of Hibernate did you use in your Project? How did you use Hibernate in your Project?**

**Ans. F**irst we have to add the Dependency “Spring Data JPA” -> then we update “Application.Properties” File, then we create an Entity Class -> then we create a Repository Layer which will allow **CRUD O**perations to perform, and Hibernate will be internally Automatically applied to this **CRUD** Operations.

**JPA ANNOTATIONS:**

All the Annotation that I have used on Entity class like “**@Entity, @Id, @Table,@Column**” are Hibernate Annotations. Hibernate is an implementation of JPA. Hibernate is a ORM Tool.

* **@Entity:** It helps us to Map Java Class with Class with Database Table.
* **@Table:** **I**t helps us to Map Java Class Entity name with Database Table name in cases where their names are not same.
* **@Column:** It helps us to Map Entity Class Variable name with Database Column name if both names are not same.
* **@GenratedValue(“”):** It helps us to Auto-Increment the value when we save the data in the Database.
* **@MappedSuperclass: ?????? ->** In our Project I **created** an **@ID “private int id”,** in a **super class** because that **“id”** was used in many places, so I declared it **once inside Parent Super class mapped with this Annotation(@MappedSuperclass),** and later **inherited Super Class** to all the **Child Entity Class** where **“id”** was required**.** So thus we achieved **reusability** ofthat **“id” variable.** Don’t need to declare it everywhere multiple Times.
* **@PathVariable:** It maps the URL Data with WEBSERVICES method Argument.
* **@RequestBody: It** maps the JSON Object Content with the Java Class Object.

**InterviewQuestion: Where I have used Encapsulation in our Project?**

Ans. While creating Entity Class, used Encapsulation.

**InterviewQuestion: Where I have used Abstraction in our Project?**

Ans. I created a **Service** **Layer** named **EmailService** **Interface** where I created **Incomplete Abstract Methods** for achieving **Abstraction, i.e., Hiding implementation Details.**

**InterviewQuestion: What is Session Factory?**

**Ans.** It helps us to connect with the Database in Hibernate. It gives us built-in method to perform the **CRUD** operation in Hibernate.

**Singleton design pattern ->** It means we need to design a class in such a way that only one Object of the class can be created throughout the Program Execution. The ways are:-

* By declaring the constructor of that class private.
* By creating a static reference variable and

**CRM TOOL:**

It’s a end-to-end marketing activities for my client. Here client is my showroom. It’s a tool used for keeping all the records of the visitors who has visited the showroom for purchasing a car.

**CRM** belongs to **Marketing DOMAIN.**

Now the Customer Detailsare filled in a form, who wants to purchase cars, and stored in their Database. The Marketing term of storing the Customer data in a Form is termed as **Leads Capturing.**

Now after Leads are captured(form is filled in website) and we click on submit button, then an email is send to that Customer for a Test Drive.

Now after email is sent and the Customer is ready for TestDrive also and will purchase the car, then his details is moved to  **Contacts Section from Leads Section. Difference** between  **Leads and Contacts Section are:**

**Leads** are the places where the details of all those Customers are stored, who has just visited the show room.

**Contacts Section** is the place in DB where details of only those Customers are kept from the **Leads** Section, who actually want to purchase a car.

**PROJECT FLOW:**

* Created the Database Schema
* Created a SpringStarter Project, and installing Dependencies like Mysql Driver, Spring Data JPA and Spring WEB.
* Create an Entity class “Leads”.
* Created Repository Layer extending JPA Repository for performing CRUD Operation.
* Configured application.properties File for connecting with Database
* Then created Controller Layer
* Created Services Layer
* Created View Layer for Storing Leads Info.

Then the View will call Controller-> then Controller will call the Services Layer-> and Services Layer will save the Data.

Which all modules did you develop in your Project?

Ans. Leads, contacts, Billing Module, Reports, Email Modules.

Implementing Microservices and Webservices?

First create a Search Option in Billing Module which is suppose running on port 8080. Then in the Search Bar, give the ID of the customer or Contacts, whose Billing is to be done after purchasing a car.

Now in Search bar we give the id=3, and then click “submit” -> It will send a request to the “Contacts” Section running on port 8081 to fetch all the details of that Customer with id=3, using  **Consuming and Exposing** Process. And after getting the details, we return to the Billing Section Port, and create a Bill of the individual.

Where did you use MicroServices in your Project?

Developing Billing Module.

**What is CRM?**

**Ans.** It helps our client to perform end-to-end marketing activities. It is a very big complex application that consists of core modules like:

* Leads
* Contacts
* Reports
* Email API
* Billing, etc.

ProjectName: Emarketing 5.4

Duration: 1.1

Technology: Core Java, Hibernate JPA, Restful Webservices etc.,

Tools: STS, MySQL Workbench, PostMan, etc.

Process: Agile Scrum Process

Description:

Roles and Responsibilities:

Explain your Project?

Ans. I was responsible to develop end-to-end Marketing Software for my client which is called as “xxxProject\_Name 10.5”. It was a very big and complex application which consisted of core modules like leads, reports, email api, Contacts, Billing, etc.

My roles and responsibilities consisted of:

Firstly, we have to create an application\_context.xml File, and inside it we write our Bean Factory codes. Now, In demo class we import the application\_context.xml File.

**JSP IMPLICIT OBJECTS:**

* + 1. Out.println(“yo”);
    2. Request.getAttribute(“Keyname”);

**iii)**<% HttpSession ses = request.getSession();

**if**(ses.getAttribute("keyName")!=**null**){

%>

* + 1. Through this object the JSP programmer can add new cookies or date stamps, HTTP status codes, etc. response.sendRedirect("geeksforgeeks.org");

I am not a permanent employee of capgeminay. I am working on Contract basis in Capgeminai. My pay roll company is "X".

**Java is Pass by Value because java becauses in a code, we created Java Object, and then pass the refernce address in form of argument values to another method. So the the addressis passed, but its in the from of value. And refernce points**

Many Spring Boot developers like their apps to use auto-configuration, component scan and be able to define extra configuration on their "application class". A single @SpringBootApplication annotation can be used to enable those three features, that is:

* @EnableAutoConfiguration: enable [Spring Boot’s auto-configuration mechanism](https://docs.spring.io/spring-boot/docs/2.0.x/reference/html/using-boot-auto-configuration.html)
* @ComponentScan: enable @Component scan on the package where the application is located (see [the best practices](https://docs.spring.io/spring-boot/docs/2.0.x/reference/html/using-boot-structuring-your-code.html))
* @Configuration: allow to register extra beans in the context or import additional configuration classes

The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration, and @ComponentScan with their default attributes, as shown in the following example:

**@ComponentScan is always written with @Configuration.** With Spring, **we use the @ComponentScan annotation along with the @Configuration annotation to specify the packages that we want to be scanned**. @ComponentScan without arguments tells Spring to scan the current package and all of its sub-packages.

**Hibernate Configuration XML File:**

You will specify a cache provider in hibernate.cfg.xml configuration file. We choose EHCache as our second-level cache provider −

<?xml version = "1.0" encoding = "utf-8"?>

<!DOCTYPE hibernate-configuration SYSTEM

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<property name = "hibernate.dialect">

org.hibernate.dialect.MySQLDialect

</property>

<property name = "hibernate.connection.driver\_class">

com.mysql.jdbc.Driver

</property>

<!-- Assume students is the database name -->

<property name = "hibernate.connection.url">

jdbc:mysql://localhost/test

</property>

<property name = "hibernate.connection.username">

root

</property>

<property name = "hibernate.connection.password">

root123

</property>

<property name = "hibernate.cache.provider\_class">

org.hibernate.cache.EhCacheProvider

</property>

<!-- List of XML mapping files -->

<mapping resource = "Employee.hbm.xml"/>

</session-factory>

</hibernate-configuration>

**Spring Initializr:**

It helps us to create Spring Project Structure. Earlier we didn’t had STS, so we have to do it in Eclipse. So we use to search Spring Initializr, and there we give “artifact”,”projectName”,etc, and create the Spring Application Structure, and download the file, and import it in Eclipse. Then we see the entire project structure is integrated along with pom.xml in eclipse.

**Difference between Spring and SpringBoot:**

**Spring Boot** is built on top of the conventional spring framework. So, it provides all the features of spring and is yet easier to use than spring. Spring Boot is a microservice-based framework and making a production-ready application in very less time. In Spring Boot everything is auto-configured. We just need to use proper configuration for utilizing a particular functionality. Spring Boot is very useful if we want to develop REST API.

Spring :1)Configuration is done using “.xml” file. Like if we want to connect to Database then we create “applicationContext.xml” which contains beans structures and configuration Files. All these configuration files has to be created for Spring Projects. If we want to create Object, we have to create “bean.xml” file and write the Depenencies.

SpringBoot: 1)Here configuration is done in application.properties File like configuring email API 3rd party.

Spring: 2) All Dependencies Jar like Hibernate JPA, MySQL driver, spring Web, all this dependency JARs have to be downloaded externally. Even we had to download Tomcat Server separately from outside. Here in pom.xml nothing was their as “Spring Starter Web” in <dependency> tag. We had to externally download every dependencies like “Spring-core” or “Spring-context <dependency>.

SpringBoot: 2)Spring Boot provides embedded servers such as embedded Jasper Tomcat and Jetty etc. Inside pom.xml, it contains starter tags like “**spring-boot-starter-web” or “spring-boot-starter-data-jpa”** , which downloads all the Spring JARS automatically. Has feature of Auto-Configuration for everything like Hibernate, web, MySQL database.

It gives us Starter Dependency Tags.

**Handling Exceptions using SpringBoot and not try-catch:**

First we have to create a class named **“LocationNotFound”** which will be extending **“RuntimeException”,** to create our own **user-defined Exception “LocationNotFound” using** “throw” keyword.

Now Inside the Controller class, we create a **getLocationById(@PathVariable(“id”) long id)** method, to fetch the request url containing an id like [**http://localhost:8080/location/id**](http://localhost:8080/location/id)**,** where id=0,1,2, can be anything. Now we fetch that id from URL, and use a **if block** to check if the id fetched from URL is == 0, then we will throw “**LocationNotFound”** Exception using “throw” keyword.

**But,** now after throwing the Exception, we need to catch the Exception to suppress it. So, in place of try-catch Block, we create a class named “LocationExceptionController”, which will be marked with **“@ControllerAdvice”.** This Annotation will give the **class** a **superpower** to make that class a kind-of Catch Block to **handle** the Exceptions.

Moment we create the class using @ControlerAdvice, SpringBoot knows that the moment any user-defined Exceptions arise, we will give it to the class marked with **@ControllerAdvice. Now, @ControllerAdvice** has a method named **@ExceptionHandler(value=LocationNotFound.class)** ,which will suppress the Exception defined in it’s argument containing **“value=LocationNotFound.class”.**

**Project Structure for @ControllerAdvisor Exception Class:**

src

├───main

│ ├───java

│ │ └───com

│ │ └───zetcode

│ │ │ Application.java

│ │ │ MyRunner.java

│ │ ├───controller

│ │ │ MyController.java

│ │ ├───exception

│ │ │ CityNotFoundException.java

│ │ │ ControllerAdvisor.java

│ │ │ NoDataFoundException.java

│ │ ├───model

│ │ │ City.java

│ │ ├───repository

│ │ │ CityRepository.java

│ │ └───service

│ │ CityService.java

│ │ ICityService.java

If @ControllerAdvice is marked above a class, then that means whenever we get an Exception, then we bring that Exception to that class marked with the Annotation.

ExceptionHandler(value= LocationNotFound.class) will handle the expression if the Exception name is “LocationNotFound”.

**Exception Annotations:**

**@ControllerAdvice:** This Annotation is used to define a class where Exception can be handled

**@ExceptionHandler:** When an Exception occurs, then the Exception is given to that method which has the Annotation @ExceptionHandler(value=“locationNotFound.class”), and handles only that Exception only written inside value=””.

**ResponseEntity<Object>:** It’s an Object that helps us to give the http response(**like 404NotFound OR 200OK)** and a “**message”** is sent back to the user when an Exception Occurs.

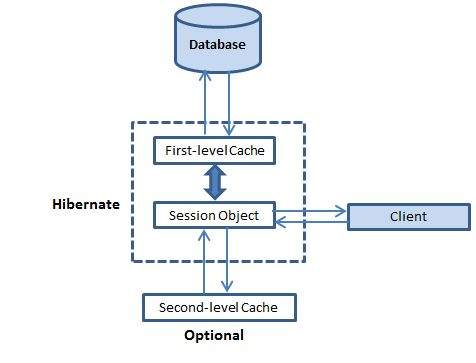
**InterviewQuestion: In your Project when you handled Exception?**

**Ans.** When the id number given was invalid(like id=0), like we are fetching that row which doesn’t exist. This would arise an abrupt abnormal Exception. So, using “throw keyword I created our User-defined Exception Class, and declaring a class to handle Exception using @ControllerAdvice, and @ExceptionHandler is used to suppress the Exception and print a meaningful “**message”** back to theuser who requested for id=0.

**Hbernate Caching File:**

Caching is a mechanism to enhance the performance of a system. It is a buffer memorythat lies between the application and the database. Cache memory stores recently used data items in order to reduce the number of database hits as much as possible.

Caching is important to Hibernate as well. It utilizes a multilevel caching scheme as explained below −



## First-level Cache

The first-level cache is the Session cache and is a mandatory cache through which all requests must pass. The Session object keeps an object under its own power before committing it to the database.

If you issue multiple updates to an object, Hibernate tries to delay doing the update as long as possible to reduce the number of update SQL statements issued. If you close the session, all the objects being cached are lost and either persisted or updated in the database.

## Second-level Cache

Second level cache is an optional cache and first-level cache will always be consulted before any attempt is made to locate an object in the second-level cache. The second level cache can be configured on a per-class and per-collection basis and mainly responsible for caching objects across sessions.

Any third-party cache can be used with Hibernate. An **org.hibernate.cache.CacheProvider** interface is provided, which must be implemented to provide Hibernate with a handle to the cache implementation.

## Query-level Cache

Hibernate also implements a cache for query resultsets that integrates closely with the second-level cache.

This is an optional feature and requires two additional physical cache regions that hold the cached query results and the timestamps when a table was last updated. This is only useful for queries that are run frequently with the same parameters.

## The Second Level Cache

Hibernate uses first-level cache by default and you have nothing to do to use first-level cache. Let's go straight to the optional second-level cache. Not all classes benefit from caching, so it's important to be able to disable the second-level cache.

The Hibernate second-level cache is set up in two steps. First, you have to decide which concurrency strategy to use. After that, you configure cache expiration and physical cache attributes using the cache provider.