* **Annotation:** Java Annotation is a tag that represents the metadata i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.
* In most typical applications, we have distinct layers like data access, presentation, service, business, etc.
* Additionally, in each layer we have various beans**. To detect these beans automatically, Spring uses class path scanning annotations**. **Then it registers each bean in the Application Context**.

**Bean**: **bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.**

1. **@Component Annotation**:

* Indicates that an annotated class is a "bean/ component". Such classes are considered as candidates for auto-detection when using annotation-based configuration and class path scanning.
* We can use @Component across the application to mark the beans as Spring’s managed components
* In other words, without having to write any explicit code, Spring will:
* Scan our application for classes annotated with @Component
* Instantiate them and inject any specified dependencies into them
* Inject them wherever needed

**Example:**

**ComponentDemo.java**

import org.springframework.stereotype.Component;  
  
// Annotation  
@Component  
  
// Class  
public class ComponentDemo {  
  
 // Method  
 public void demoFunction()  
 {  
 // Print statement when method is called  
 System.*out*.println("Hello GeeksForGeeks");  
 }  
}

**DemoApplication.java**

// Java Program to Illustrate Application class  
  
// Importing required classes  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
// Annotation  
@SpringBootApplication  
  
// Class  
public class DemoApplication {  
  
 // Main driver method  
 public static void main(String[] args)  
 {  
  
 // Annotation based spring context  
 AnnotationConfigApplicationContext context  
 = new AnnotationConfigApplicationContext();  
 context.scan("com.example.demo");  
 context.refresh();  
  
 // Getting the Bean from the component class  
 ComponentDemo componentDemo  
 = context.getBean(ComponentDemo.class);  
 componentDemo.demoFunction();  
  
 // Closing the context  
 // using close() method  
 context.close();  
 }  
}

**Spring Stereotype Annotations:** Spring has provided a few specialized stereotype annotations: @Controller, @Service and @Repository. They all provide the same function as @Component.

1. **@Controller Annotation:**

* @Controller annotation indicates that the annotated class is a controller. It is a specialization of @Component and is auto detected through class path scanning.
* We specify a class with @Controller to indicate that they’re front controllers and responsible to handle user requests and return the appropriate response
* It is typically used in combination with annotated handler methods based on the @RequestMapping annotation.

// Java Program to Illustrate DemoController File  
// Importing required classes  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.ResponseBody;  
  
// Annotation  
@Controller  
// Main class  
public class DemoController {  
  
 @RequestMapping("/hello")  
 @ResponseBody  
  
 // Method  
 public String helloGFG()  
 {  
 return "Hello GeeksForGeeks";  
 }  
}

1. **@RestController Annotation:**

* @RestController is the combination of the @controller and @ResponseBody annotations.
* The controller is annotated with the @RestController annotation; therefore, the @ResponseBody isn't required.
* Every request handling method of the controller class automatically serializes return objects into HttpResponse.

@RestController  
@RequestMapping("books-rest")  
public class SimpleBookRestController {  
  
 @GetMapping("/{id}", produces = "application/json")  
 public Book getBook(@PathVariable int id) {  
 return findBookById(id);  
 }  
}

**@Controller**

**@ResponseBody**

**class Controller**

**{**

**------**

**------**

**------**

**}**

It is equivalent to

**@RestController**

**class RestController**

**{**

**------**

**------**

**------**

**}**

| **@Controller** | **@RestController** |
| --- | --- |
| @Controller is used to mark classes as Spring MVC Controller. | @RestController annotation is a special controller used in RESTful Web services, and it’s the combination of @Controller and @ResponseBody annotation. |
| **It is a specialized version of @Component annotation.** | **It is a specialized version of @Controller annotation.** |
| In @Controller, we can return a view in Spring Web MVC. | In @RestController, we cannot return a view. |
| @Controller annotation indicates that the class is a “controller” like a web controller. | @RestController annotation indicates that class is a controller where @RequestMapping methods assume @ResponseBody semantics by default. |
| **In @Controller, we need to use @ResponseBody on every handler method.** | **In @RestController, we don’t need to use @ResponseBody on every handler method.** |
| It was added to Spring 2.5 version. | It was added to Spring 4.0 version. |

* For example Controller returning HTML view:

@Controller  
public class HtmlController {  
 @GetMapping(value = "/welcome", produces = MediaType.TEXT\_HTML\_VALUE)  
 @ResponseBody  
 public String welcomeAsHTML() {  
 return "<html>\n" + "<header><title>Welcome</title></header>\n" +  
 "<body>\n" + "Hello world\n" + "</body>\n" + "</html>";  
 }  
}

1. **@Service Annotation:**

* We mark beans with @Service to indicate that they're holding the business logic and **it also enables annotated classes to be discovered and registered with the application context.**
* Besides being used in the service layer, there isn’t any other special use for this annotation

1. **@Repository Annotation:**

* @Repository serves as a specialization of @Component, **it also enables annotated classes to be discovered and registered with the application context**
* We specify a class with @Repository to indicate that they’re dealing with CRUD operations, usually, it’s used with DAO (Data Access Object) or Repository implementations that deal with database tables.

**Spring Core related Annotations:**

1. **@Bean Annotation( Method level Annotation):**

* Spring @Bean Annotation is **applied on a method** **to specify that it returns a bean to be** **managed by spring context**.
* Spring Bean annotation is usually declared in Configuration classes methods
* During Java configuration (@Configuration), the method is executed and its return value is registered as a bean within a Bean Factory.

1. **@Configuration Annotation:**

* Designates the class as a configuration class for [**Java configuration**](https://learnjava.co.in/how-spring-works-under-the-hood/#Configuration_metadata).
* Spring Configuration annotation indicates that the class has **@Bean** definition methods. So spring container can process the class and generate Spring Beans to be used in the application.

**CollegeConfig.java**

// Java Program to Illustrate Configuration Class  
// Importing required classes  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
@Configuration  
public class CollegeConfig {  
  
 // Using Bean annotation to create College class Bean  
 @Bean  
 // Here the method name is the bean id/bean name  
 public College collegeBean() {  
  
 // Return the College object  
 return new College();  
 }  
  
}

**Main.java**

// Java Program to Illustrate Application Class  
// Importing required classes  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
// Application class  
public class Main {  
 public static void main(String[] args)  
 {  
  
 // Use AnnotationConfigApplicationContext  
 // instead of ClassPathXmlApplicationContext  
 // because we are not using XML Configuration  
 ApplicationContext context  
 = new AnnotationConfigApplicationContext(  
 CollegeConfig.class);  
 // Getting the bean  
 College college  
 = context.getBean("collegeBean", College.class);  
 // Invoking the method inside main() method  
 college.test();  
 }  
**}**

1. **@EnableAutoConfiguration Annotation:**

* This enables Spring Boot’s [auto configuration mechanism](https://learnjava.co.in/springboot-what-and-why/#Autoconfiguration). Auto-configuration refers to creating beans automatically by scanning the class path.

1. **@ComponentScan Annotation:**

* Typically, in a [spring](https://click.linksynergy.com/deeplink?id=MnzIZAZNE5Y&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Fspring-tutorial-for-beginners%2F) application, annotations like @Component, @Configuration, @Service, @Repository are specified on classes to mark them as spring beans.
* **The @ComponentScan annotation basically tells Spring Boot to scan the current package and its sub-packages in order to identify annotated classes and configure them as spring beans**.
* Thus, it designates the current package as the root package for component scanning.

1. **@SpringBootApplication Annotation:**

* The @SpringBootApplication annotation is a convenience annotation that combines **@EnableAutoConfiguration**, **@Configuration** and the **@ComponentScan** annotations in a [Spring Boot](https://click.linksynergy.com/deeplink?id=MnzIZAZNE5Y&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Fspringbootfundamentals%2F) application.

1. **@Autowired Annotation:**

* **It allows spring to resolve and inject collaborating beans into our bean.**
* When we run this Spring Boot application, **@SpringBootApplication annotation will automatically scan the components in the current package and its sub-packages**. Thus it will register them in spring's Application Context, and allow us to inject beans using @Autowired.
* **we can use auto wiring on properties, setters, and constructors**:

1. **@Autowired on properties:**

@Component  
public class FooService {  
 @Autowired  
 private FooFormatter fooFormatter;  
}

1. **@Autowired on Setters:**

public class FooService {  
 private FooFormatter fooFormatter;  
 @Autowired  
 public void setFormatter(FooFormatter fooFormatter) {  
 this.fooFormatter = fooFormatter;  
 }  
}

1. **@Autowired on Constructors:**

public class FooService {  
 private FooFormatter fooFormatter;  
 @Autowired  
 public FooService(FooFormatter fooFormatter) {  
 this.fooFormatter = fooFormatter;  
 }  
}

* When a bean is being constructed, the @Autowired dependencies should be available. Otherwise, **if spring cannot resolve a bean for wiring, it will throw an exception named “NoSuchBeanException”**.

To fix this, we need to declare a bean of the required type:

public class FooService {  
 @Autowired(required = false)  
 private FooDAO dataAccessor;  
}

1. **@Qualifier Annotation:**

* By default, spring resolves @Autowired entries by type. **If more than one bean of the same type is available in the container, the framework will throw a fatal exception.**

**For Example:**

**FooFormatter.java**

@Component("fooFormatter")  
public class FooFormatter implements Formatter {  
 public String format() {  
 return "foo";  
 }  
}

**BarFormatter.java**

@Component("barFormatter")  
public class BarFormatter implements Formatter {  
 public String format() {  
 return "bar";  
 }  
}

**FooService.java**

public class FooService {  
 @Autowired  
 private Formatter formatter;  
}

* In our example, there are two concrete implementations of Formatter available for the spring container. As a result, **spring will throw a “NoUniqueBeanDefinitionException” exception when constructing the FooService**.
* When there are multiple beans of the same type, it's a good idea to**use @Qualifier to avoid ambiguity.**

public class FooService {  
 @Autowired  
 @Qualifier("fooFormatter")  
 private Formatter formatter;  
}

1. **@Lazy Annotation:**

* **By default, spring creates all singleton beans eagerly at the startup/bootstrapping of the application context.** The reason behind this is to avoid and detect all possible errors immediately rather than at runtime.
* **When we put @Lazy annotation over the @Configuration class, it indicates that all the methods with @Bean annotation should be loaded lazily.**
* A @Lazy bean is not initialized until referenced by another bean or explicitly retrieved from BeanFactory

**Example: To make a single bean lazy**

**The Bean that we want to load lazily:**

@Lazy  
@Component  
public class City {  
 public City() {  
 System.*out*.println("City bean initialized");  
 }  
}

It’s Reference

public class Region {  
  
 @Lazy  
 @Autowired  
 private City city;  
  
 public Region() {  
 System.*out*.println("Region bean initialized");  
 }  
  
 public City getCityInstance() {  
 return city;  
 }  
}

## **Autowiring @Lazy Beans**: Generally, beans are injected into other components using [***@Autowired***](https://howtodoinjava.com/spring-core/spring-beans-autowiring-concepts/) annotation. In this case, **we must use the @Lazy annotation at both places**:

* The bean definition which we want to lazy load
* The place it is injected along with @Autowired annotation

@Lazy  
@Service  
public class EmployeeManagerImpl implements EmployeeManager {  
 //  
}

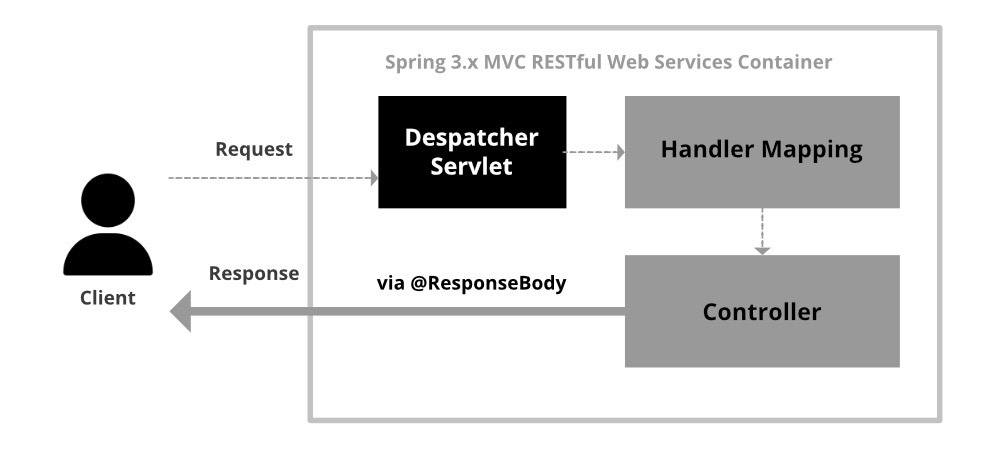
@Controller  
public class EmployeeController {  
 @Lazy  
 @Autowired  
 EmployeeManager employeeManager;  
}

**Rest API related Annotations:**

1. **@ResponseBody Annotation:**
   * @ResponseBody is a spring annotation which binds a method return value to the web response body.
   * It is not interpreted as a view name. It uses HTTP Message converters to convert the return value to HTTP response body, based on the content-type in the request HTTP header.
   * **The @ResponseBody annotation tells a controller that the object returned is automatically serialized into JSON and passed back into the HttpResponse obj**ect.
   * When we use the @ResponseBody annotation, we're still able to explicitly set the content type that our method returns.
   * For that, **we can use the @RequestMapping‘s produces attribute.**
2. **@RequestBody Annoatation:**

* **The @RequestBody annotation maps the HttpRequest body to a transfer or domain object, enabling automatic deserialization** of the inbound HttpRequest body onto a Java object.
* Spring automatically deserializes the JSON into a Java type, 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**

**Diagram depicting @ResponseBody Annotations:**



Example for @RequestBody and @ResponseBody Annotations:

@PostMapping(value = "/content", produces = MediaType.APPLICATION\_JSON\_VALUE)  
@ResponseBody  
public ResponseTransfer postResponseJsonContent(@RequestBody LoginForm loginForm) {  
 return new ResponseTransfer("JSON Content!");  
}

1. **@RequestMapping Annotation:**

* RequestMapping annotation is used to map web requests onto specific handler classes and/or handler methods.
* @RequestMapping can be applied to the controller class as well as methods.
* We can use a single method for handling multiple URIs

@RequestMapping(value={"/method1","/method1/second"})  
@ResponseBody  
public String method1(){  
 return "method1";  
}

* [@RequestMapping](https://www.digitalocean.com/community/users/requestmapping) provides **produces** and **consumes** variables where we can specify the request content-type for which method will be invoked and the response content type.

@RequestMapping(value="/method6", produces={"application/json","application/xml"}, consumes="text/html")  
@ResponseBody  
public String method6(){  
 return "method6";  
}

1. **@PathVariable Annotation:**

* **The @PathVariable annotation can be used to handle template variables in the request URI mapping**, and set them as method parameters

@RequestMapping(path="/{name}/{age}")  
public String getMessage(@PathVariable("name") String name,  
@PathVariable("age") String age){  
}

1. **@RequestParam Annotation:**

* **We can use @RequestParam to extract query parameters, form parameters, and even files from the request.**
* When the parameter isn't specified, the method parameter is bound to null.
* **we can configure the @RequestParam name using the name attribute**

@PostMapping("/api/foos")  
@ResponseBody  
public String addFoo(@RequestParam(name = "id") String fooId, @RequestParam String name) {  
 return "ID: " + fooId + " Name: " + name;  
}

* We can configure our @RequestParam to be optional with the required attribute or we can even make request param of type Optional.

@GetMapping("/api/foos")  
@ResponseBody  
public String getFoos(@RequestParam(required = false) String id) {  
 return "ID: " + id;  
}

@GetMapping("/api/foos")  
@ResponseBody  
public String getFoos(@RequestParam Optional<String> id){  
 return "ID: " + id.orElseGet(() -> "not provided");  
}

**Multiple value Query Parameter:**

@GetMapping("/api/foos")  
@ResponseBody  
public String getFoos(@RequestParam List<String> id) {  
 return "IDs are " + id;  
}

* **Spring MVC will map a comma-delimited id parameter or a list of separate id parameters**.

So, you can call api in above example using:

* <http://localhost:8080/spring-mvc-basics/api/foos?id=1,2,3>
* <http://localhost:8080/spring-mvc-basics/api/foos?id=1&id=2>

@RequestParam vs @PathVariable:

|  |  |
| --- | --- |
| **@RequestParam** | **@PathVariable** |
| 1. @RequestParams extract values from the querystring | 1. @PathVariables extract values from the URI path |

1. **@GetMapping Annotation:**

* The @GetMapping annotation is a composed version of @RequestMapping annotation that acts as a **shortcut for @RequestMapping(uri,method = RequestMethod. GET).**
* The @GetMapping annotated methods handle the HTTP GET requests matched with the given URI expression.

1. **@PostMapping Annotation:**

* The @PostMapping is a specialized version of @RequestMapping annotation that acts as a **shortcut for @RequestMapping(uri,method = RequestMethod. POST)**.
* The @PostMapping annotated methods handle the HTTP POST requests matched with the given URI expression.

1. **@PutMapping Annotation:**

* As we know PUT HTTP method is used to update/modify the resource so the @PutMapping annotation is used for mapping HTTP PUT requests onto specific handler methods.
* Specifically, @PutMapping is a composed annotation that acts as **a shortcut for @RequestMapping(uri,method = RequestMethod. PUT)**.

1. **@DeleteMapping Annotation:**

* @DeleteMapping annotation maps HTTP DELETE requests onto specific handler methods.
* It is a composed annotation that acts as a **shortcut for @RequestMapping(method = RequestMethod. DELETE)**.

1. **@PatchMapping Annotation:**

* **When a client needs to replace an existing Resource entirely, they can use PUT. When they're doing a partial update, they can use HTTP PATCH.**
* For instance, when updating a single field of the Resource, sending the complete Resource representation can be cumbersome and uses a lot of unnecessary bandwidth. In such cases, the semantics of PATCH make a lot more sense.

**Spring Data JPA related annotations:**

* **Entity: Entities in JPA are nothing but POJOs representing data** that can be persisted to the database. **An entity represents a table stored in a database**. Every instance of an entity represents a row in the table.

1. **@Entity Annotation:**

* The @Entity annotation specifies that the class is an entity and is mapped to a database table.
* Each JPA entity must have a primary key that uniquely identifies it. [**The @Id annotation**](https://www.baeldung.com/hibernate-identifiers) defines the primary key.
* We can generate the identifiers in different ways, which are specified by the **@GeneratedValue annotation**.
* We can choose from four id generation strategies with the strategy element. **The value can be AUTO, TABLE, SEQUENCE, or IDENTITY**

**1. AUTO:** the persistence provider will determine values based on the type of the primary key attribute. This type can be numerical or UUID.  
**2. IDENTITY:** **This type of generation relies on the IdentityGenerator, which expects values generated by an identity column in the database. This means they are auto-incremented.  
3. SEQUENCE:** A sequence specify a database object that can be used as a source of primary key values. It uses @SequenceGenerator.  
**4. TABLE:** The TableGenerator uses an underlying database table that holds segments of identifier generation values. It keeps a separate table with the primary key values. It uses @TableGenerator.

* **Default value of Strategy attribute is AUTO.**

@Id  
@GeneratedValue(strategy=GenerationType.TABLE , generator="student\_generator")

* The Generator attribute is used to specify the name of the primary key generator to use as specified in the SequenceGenerator or TableGenerator annotation. It is an optional attribute.

1. **@Table Annotation:**

* **The name of the table in the database and the name of the entity won't be the same.** In these cases, we can specify the table name using the @Table annotation

1. **@Column Annotation:**

* we can use the @Column annotation to mention the details of a column in the table
* The @Column annotation has many elements such as name, length, nullable, and unique.
* The name element specifies the name of the column in the table. The length element specifies its length. The nullable element specifies whether the column is nullable or not, and the unique element specifies whether the column is unique.
* **If we don't specify this annotation, the name of the column in the table will be the name of the field.**

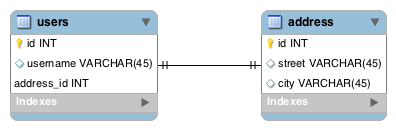
1. **@Transactional Annotation:**

**Entity Class relationship:**

**Foreign Key:** In the relational databases, a foreign key is a field or a column that is used to establish a link between two tables.

In simple words you can say that, a foreign key in one table used to point primary key in another table.

1. **@OnetoOne Annotation:**



The address\_id column in users is the [foreign key](https://en.wikipedia.org/wiki/Foreign_key) to address

**Users.java**

@Entity  
@Table(name = "users")  
public class User {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.AUTO)  
 @Column(name = "id")  
 private Long id;  
 //...   
  
 **@OneToOne(cascade = CascadeType.ALL)  
 @JoinColumn(name = "address\_id", referencedColumnName = "id")  
 private Address address;**  
 // ... getters and setters  
}

* We place the @OneToOne annotation on the related entity field, Address.
* Also, we need to place [**the @JoinColumn annotation**](https://www.baeldung.com/jpa-join-column)**to configure the name of the column in the users table that maps to the primary key in the address table.**
* Note in the next entity we won't use the @JoinColumn annotation. This is because we only need it on the owning side of the foreign key relationship.
* **Simply whoever owns the foreign key column gets the @JoinColumn annotation.**

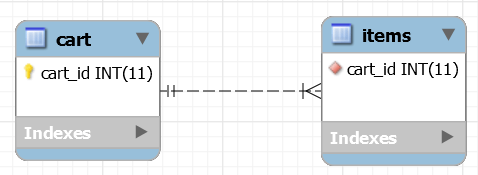
**Address.java**

@Entity  
@Table(name = "address")  
public class Address {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.AUTO)  
 @Column(name = "id")  
 private Long id;  
 //...  
  
 **@OneToOne(mappedBy = "address")  
 private User user;**  
  
 //... getters and setters  
}

* We also need to place the @OneToOne annotation here too. That's because this is a [bidirectional relationship](https://docs.oracle.com/cd/E19798-01/821-1841/bnbqj/index.html). **The address side of the relationship is called the non-owning side**.

**Note: In unidirectional case, no need to place @OneToOne annotation in non-owning side**

1. **@OnetoMany Annotation:**
   * **one-to-many mapping means that one row in a table is mapped to multiple rows in another table.**



* + For this example, we'll implement a cart system where we have a table for each cart and another table for each item.
  + **One cart can have many items, so here we have a one-to-many mapping.**
  + The way this works at the database level is we have a cart\_id as a primary key in the cart table and also a cart\_id as a foreign key in items

**Cart.java**

@Entity

public class Cart {  
  
 //...   
  
 @OneToMany(mappedBy="cart")  
 private Set<Item> items;  
  
 //...  
}

**Item.java**

@Entity  
@Table(name="ITEMS")  
public class Item {  
  
 //...  
 @ManyToOne  
 @JoinColumn(name="cart\_id", nullable=false)  
 private Cart cart;

// getters and setters  
}

* The @ManyToOne annotation is associated with the Cart class variable. @JoinColumn annotation references the mapped column.
* Note this is bidirectional, **But in unidirectional No need to place @ManyToOne or @JoinColumn in non-owning side**

1. **@ManytoOne Annotation:**
   * In many to one relationship, multiple source objects can have relationship with same target object.
   * Let’s consider CD and Artist. So multiple CDs can be written by same Artist or multiple Artists can write the same CD.

**Cd.java**

@Table  
@Entity  
public class Cd {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private int cdId;  
  
 @Column  
 private String cdTitle;  
  
 @JoinColumn(name = "artistId")  
 @ManyToOne(cascade = CascadeType.ALL, optional = false)  
 private Artist artist;  
}

* The @ManyToOne annotation defines the relationship between entity classes or tables in database. This annotation will define many Cd can have single artist or a single artist can have many Cd.

**Artist.java**

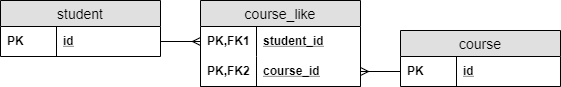
@Table  
@Entity  
public class Artist {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private int artistId;  
  
 @Column  
 private String artistName;  
  
 @OneToMany(mappedBy = "artist", cascade = CascadeType.ALL)  
 private Set<Cd> cds = new HashSet<>();  
}

* **This is bidirectional, In unidirectional no need to place @OneToMany on non-owning side.**

1. **@ManytoMany Annotation:**

* **A relationship is a connection between two types of entities. In the case of a many-to-many relationship, both sides can relate to multiple instances of the other side.**

A student can like **many** courses, and **many** students can like the same course



* Since both sides should be able to reference the other, **we need to create a separate table to hold the foreign keys**. Such a table is called a **join table.** In a join table, the combination of the foreign keys will be its composite primary key.
* Every Many-to-Many relationship has two sides: The owning side & the non-owning (inverse side)
* The join table is specified on the owning side using [@JoinTable](https://docs.oracle.com/javaee/7/api/javax/persistence/JoinTable.html) annotation. This relationship is bidirectional, the inverse side must use the mappedBy element to specify the relationship field or property of the owning side.

**Student.java**

@Entity  
class Student {  
  
 @Id  
 Long id;  
  
 @ManyToMany  
 @JoinTable(name = "course\_like",  
 joinColumns = @JoinColumn(name = "student\_id"),  
 inverseJoinColumns = @JoinColumn(name = "course\_id"))  
 Set<Course> likedCourses;  
  
 // additional properties  
 // standard constructors, getters, and setters  
}

**Course.java**

@Entity  
class Course {  
  
 @Id  
 Long id;  
  
 @ManyToMany  
 Set<Student> likes;  
  
 // additional properties  
 // standard constructors, getters, and setters  
}