**Design Documentation for Quiz Web App**

**Goal**:

This is a **Web Application,** which serves users with a set of MCQ questions, along with 4 options. For each question, user has to choose one option. Based on the choices received from the user, they will be graded and which will be displayed accordingly.

Other functionalities that can be performed:

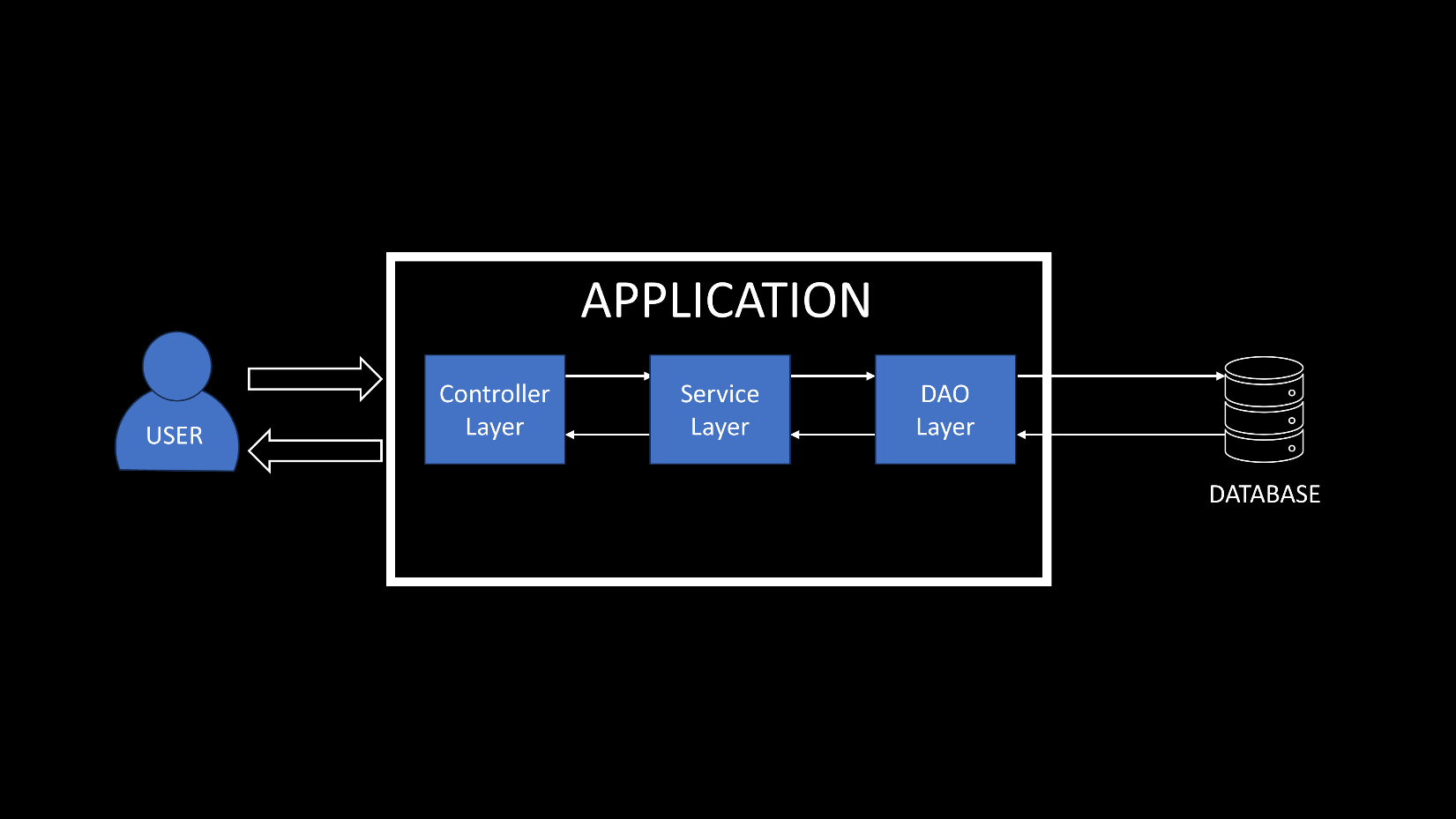
1. User can submit additional questions to the application as well as update and delete questions already present.
2. User can create quizzes from the questions present in the DB.

In short, **CRUD** **operations can be performed on this Web Application**. All of the CRUD operations are performed by using **RESTful Web Services**.

**Basic Design Overview**:

This web application is mainly divided into 3 layers:

1. **Controller Layer**: [Package name: *com.example.quizwebapp.****controlle****r*]
   1. This layer is responsible for interacting with the users.
   2. Accepts user requests and sends them over to the Service Layer.
   3. Returns the output from the service layer to the User.
   4. Above-mentioned package contains all of our controller classes.
2. **Service Layer**: [Package name: *com.example.quizwebapp.****service***]
   1. All the business logic required by the application stays here.
   2. Processing is done here.
   3. Gets data from Controller Layer. Once done, passes the output to the controller layer.
   4. Above-mentioned package contains all of our Service classes.
3. **DAO/Repository Layer**: [Package name: *com.example.quizwebapp.****dao***]
   1. This layer is responsible for interacting with the DB.
   2. DAO layer is used to decouple the Service layer from the DB.
   3. Whenever Service layer needs to fetch data, it calls the DAO layer and it supplies the requested data back.
   4. Above-mentioned package contains all of our DAO classes.



**Models**:

Since our web application used the **MVC framework**, we have a few model classes. They are stored in the package: *com.example.quizwebapp.model*

Data usually doesn’t care about the application we are creating and it’s needs. So, we need to mold the data to fit our application’s needs. Model classes do exactly that. They contain the code required for storing, retrieving and organizing the data used in application. Also, they are independent of controllers/views.

**Database**

We have used PostgreSQL DB for this project.  
This DB contains 3 tables by the end of the project:

1. **question\_copy\_table**: This table contains all of the questions.
2. **quiz:** Contains the list of quizzes created by the user.
3. **quiz\_questions**: Contains the list of questions that each of the quizzes contain.

**question\_copy\_table:**

This is the table we start with. All the questions are stored here. Column names are written in **snake\_casing**. The columns are as follows:

1. id (Primary key and Auto-incremented)
2. category
3. difficulty\_level
4. option1
5. option2
6. option3
7. option4
8. question\_title
9. right\_answer

We can choose to create the table before or during the application. We’ll go through all the steps for both before and after. Let’s start with creating the table **before** the application.

1. **Scenario: We don’t have any CSV file to import data from, into our table**.

CREATE TABLE public.question\_copy\_table  
(

id SERIAL PRIMARY KEY,  
category character varying(255),  
difficulty\_level character varying(255),  
option1 character varying(255),  
option2 character varying(255),  
option3 character varying(255),  
option4 character varying(255),  
question\_title character varying(255),  
right\_answer character varying(255)

);

We can then add values to the table created using our application or by queries in Table.

1. **Scenario: We HAVE a CSV file to import data from, into our table**.

Name of the CSV file: *questionData.csv*  
Location of the CSV file in our local directory: *E:\New Folder\questionData.csv*  
CSV file contains all the above-mentioned columns **EXCEPT** the column **id**.

Query to create the table:

CREATE TABLE public.question\_copy\_table  
(

category character varying(255),  
difficulty\_level character varying(255),  
option1 character varying(255),  
option2 character varying(255),  
option3 character varying(255),  
option4 character varying(255),  
question\_title character varying(255),  
right\_answer character varying(255)

);

COPY public.question\_copy\_table FROM 'E:\New Folder\questionData.csv' DELIMITER ',' CSV HEADER;

ALTER TABLE question\_copy\_table ADD COLUMN id SERIAL PRIMARY KEY;

This would create the table with the id column at the end. We are doing this to prevent conflict while adding values into the table.

Now, if we want our application to create the table and update the data it running, we have to mention the following details in the **application.properties** file located in *src/main/resources*:

spring.datasource.driver-class-name=org.postgresql.Driver  
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres  
spring.datasource.username= <Username\_for\_postgreDB >  
spring.datasource.password=<Password\_for\_postgreDB >  
spring.jpa.hibernate.ddl-auto=update  
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

**quiz:**

This table is created by the application. All the quizzes made by users are stored here. The columns are as follows:

1. id (Primary key and Auto-incremented)
2. title

**quiz\_questions:**

This table is created by the application as well, because of **Many-to-Many** relationship in Quiz Entity class. This table maps the questions present in each of the quizzes. The columns are as follows:

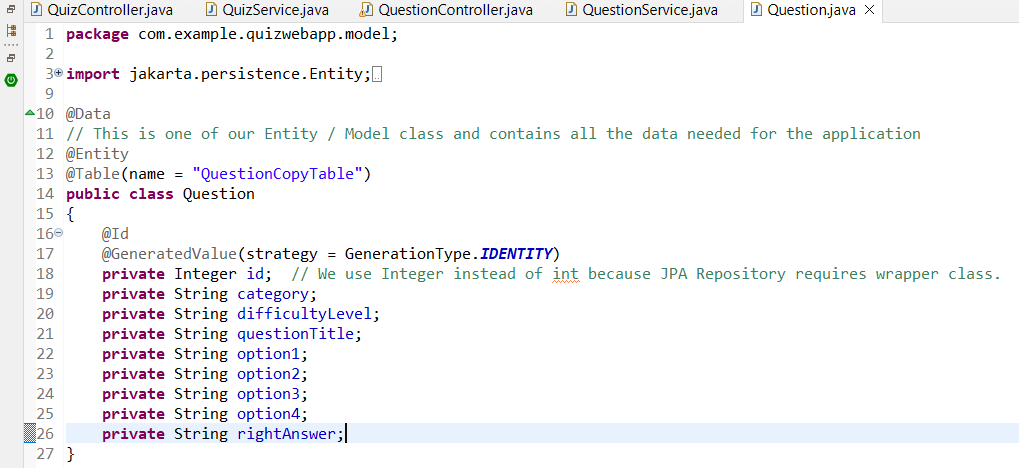
1. quiz\_id: Foreign key from table: *question\_copy\_table*.
2. questions\_id: Foreign key from table: *quiz*.

**Code Explanation:**

This project has two services: **Question Service** and **Quiz Service**.   
In both of the cases, the user interacts with the Controller Layers – **QuestionController.java** and **QuizController.java**. The flow of information starts from the controller layer.   
They pass on the information to the Service Layers – **QuestionService.java** and **QuizService.java**. Here all the processing takes place.   
Any data required by the Service Layers are fetched from the DAO Layers – **QuestionDAO.java** and **QuizDAO.java** respectively. They interact with the Database, fetches the required data and sends it back to the Service Layer.

For shaping the data which is received from the user as well as the data being sent back to the user, we have **Model Classes** in the model package. We have two **Entity model classes** - **Question** and **Quiz**. Entity indicates these classes are associated with a table in the Database. If not already present, they’ll create one:

* Question.java -> question\_copy\_table
* Quiz.java -> quiz

**Question.java**:  


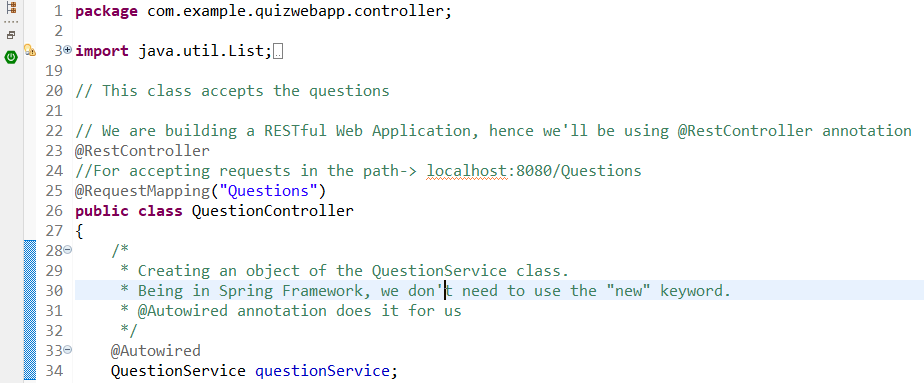
**@Data** annotation is used so that we don’t need to write getters and setters for each variable present in the class. **@Table** annotation maps the class **Question.java** to the already present table: **question\_copy\_table** in DB.

Each of the variable names (as well as the Table Name) here are written in **camelCasing** and they represent a column in the table. That way they can automatically map to the respective **snake\_casing** columns present in the table. SQL uses **snake\_casing** while Java uses "camelCasing". This is becasue we are using JPA, whose object-relational mapping (ORM) framework makes sure this happens.

**@Id** and **@GeneratedValue ( strategy = GenerationType.IDENTITY)** annotations are mapped to the variable **id**. The former annotation makes id as the primary key while the latter ensures that the value is generated and incremented automatically.

Now that we have seen the Question model class, next we’ll dive into the Question Service. This is not to be confused with the service layer class: QuestionService.java. We are talking about the part of the program that handles questions.

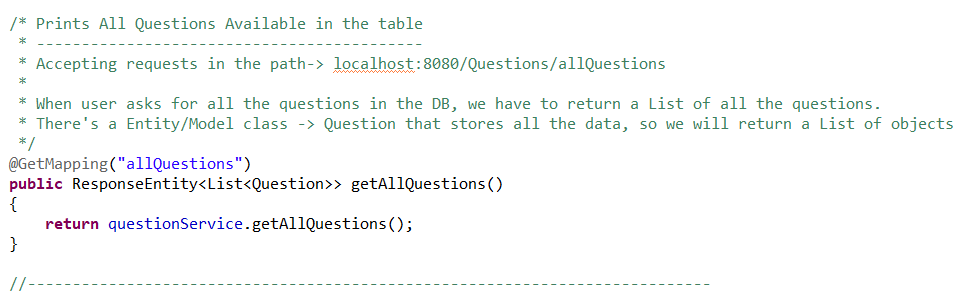
**Question Service**:

As mentioned previously, the flow of data starts from the Controller layer, we’ll take a look into **QuestionController.java**. 

The path is: **localhost:8080/Questions  
@RestController** annotation is used to indicate the class is a controller and it will handle RESTful APIs.  
**@Requestmapping(“Questions”)** annotation makes the above-mentioned path for the this controller.  
**@Autowired** annotation ensures we don’t need to write/create new objects every time we need to use methods from another class. It is mapped to **questionService,** which is areference of theclass **QuestionService.java**.

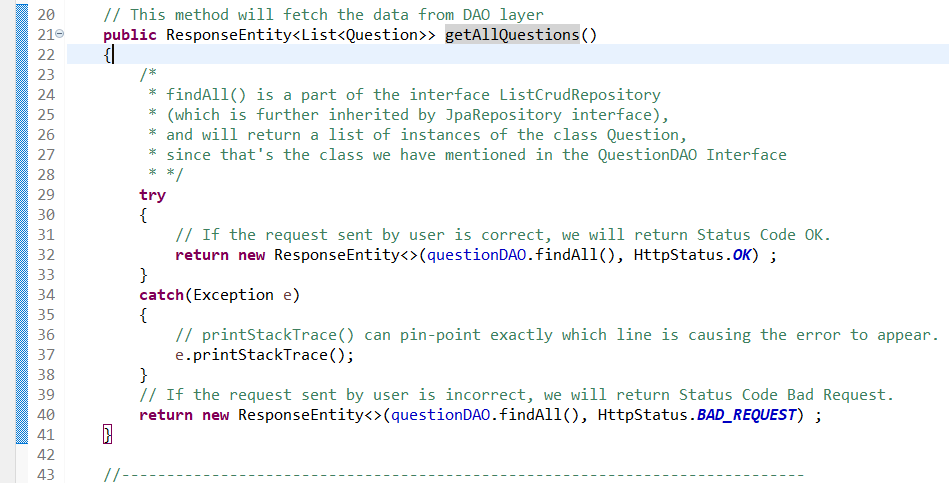
Here, we have 4 functions:

1. **getAllQuestions()**

**Path**: localhost:8080/Questions/allQuestions  


**@GetMapping** annotation handles all the GET requests.

**@GetMapping(“allQuestions”)** maps the method to the above-mentioned path, since it is a function of **QuestionController** class.

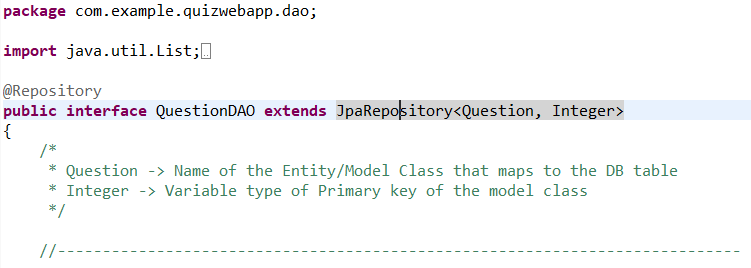
This method returns the List of All questions present in table: *question\_copy\_table*.   
It calls QuestionService’s **getAllQuestion()**.  


The method’s return type is **ResponseEntity**. It returns two things:

* List of Questions from the table (*questionDAO.findAll()*)
* Status Code (*HttpStatus.OK*)

ResponseEntity is used here because we want a proper status code to display what is the error in case the request doesn’t go through. The Response Status code is sent by the ***HttpStatusCode***.

Also, in the method, there’s **try-catch block**. This is done to handle exceptions.

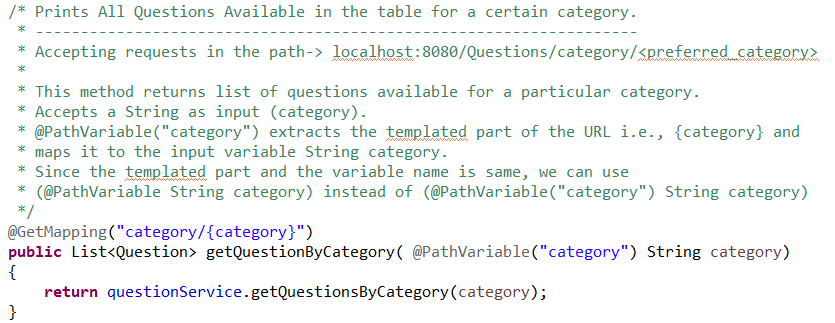
Coming to the first thing it returns, it is: ***questionDAO.findAll()***.  
Here, **questionDAO** is a reference of the interface, **QuestionDAO.java** which is autowired.  


Interface QuestionDAO extends interface **JpaRespository** which contains all the **APIs for basic CRUD operations**. JpaRespository extends **ListCrudRepository** which further extends **CrudRepository**. CrudRespository extends **Repository** interface.

In short, by extending JpaRepository, we can access all of the methods available in all 4 of these interfaces: **Repository**, **CrudRepository**, **ListRepository** and **JpaRepository**.

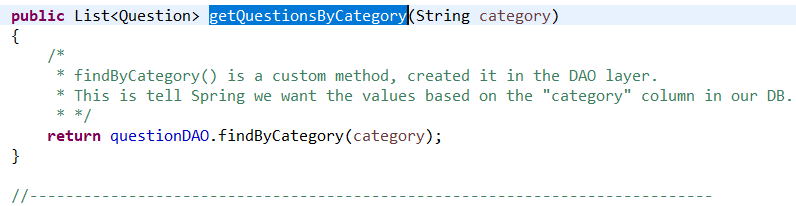
Here, QuestionService is calling the **findAll()** method which returns the List of Questions from the table. findAll() is inbuilt method present in the ListCrudRepository interface.

1. **getQuestionByCategory()**

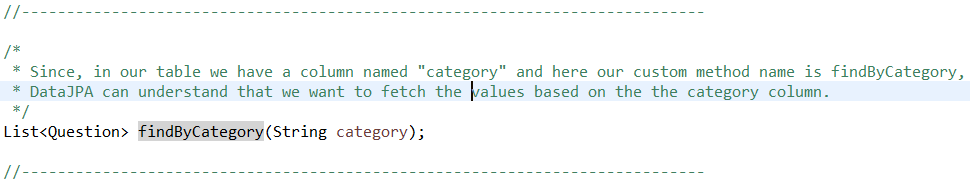
**Path**: localhost:8080/Questions/allQuestions/category/<preferred\_category>  


**@GetMapping(“category/{category}”)** maps the method to the above-mentioned path. Here, {category} is the templated part of the path.  
**(@Pathvariable(“category”) String category)** maps the templatedpart to the String category.

This method accepts the category from the user and passes it to the **QuestionService**’s getQuestionsByCategory().



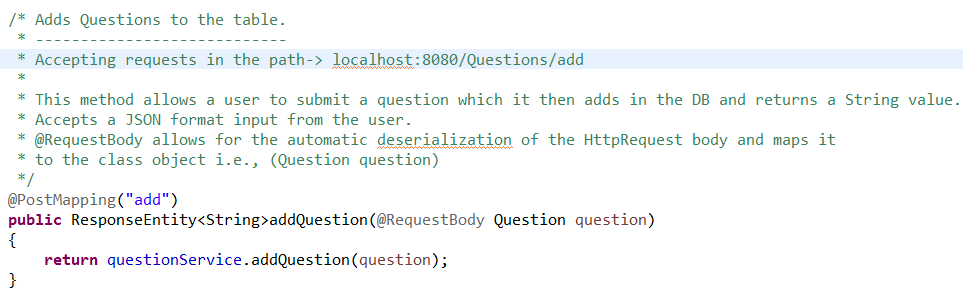
Since this is also supposed to return a List of Questions, hence the return type is set to that. This method calls the method **QuestionDAO**’s **findByCategory()** and send the category.



findByCategory() is a custom method, not an inbuilt one.

The table we are fetching data from is: question\_copy\_table and we already have a column “category” in there. Hence, when we create this method, DataJPA automatically knows what to do.

1. **addQuestion()**

**Path**: localhost:8080/Questions/allQuestions/category/add  


This function accepts a question from the User. It accepts an **object** of the Entity Model class, **Question.java**.

The question needs to be added in this JSON format as shown below:

{

"category": "JAVA",

"difficultyLevel": "Easy",

"questionTitle": "How many primitive data types are there in Java?",

"option1": "6",

"option2": "7",

"option3": "8",

"option4": "9",

"rightAnswer": "8. They are: int, short, double, float, long, char, boolean, byte"

}

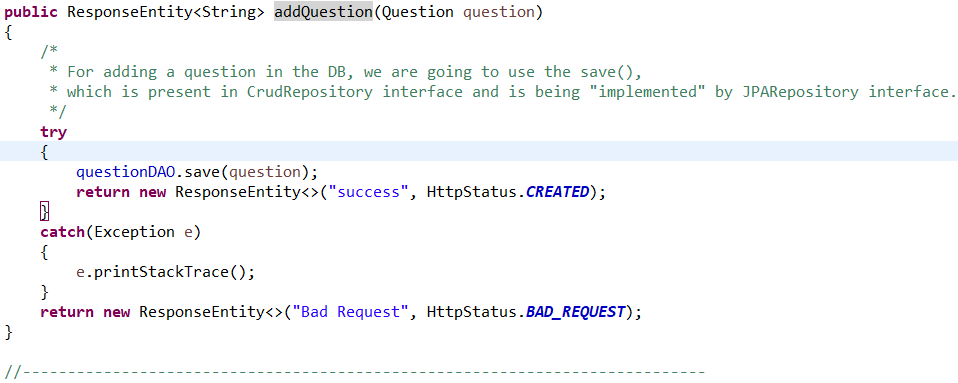
**@PostMapping** annotation handles the POST requests. Also, marks this method as a POST method.

We don’t need to mention **id**, since it is auto-generated and auto-incremented. The data we entered; forms the body of the HttpRequest, we are sending to the application.

This is an example of us, molding the data to fit our application’s needs.

@RequestBody annotation does two things:

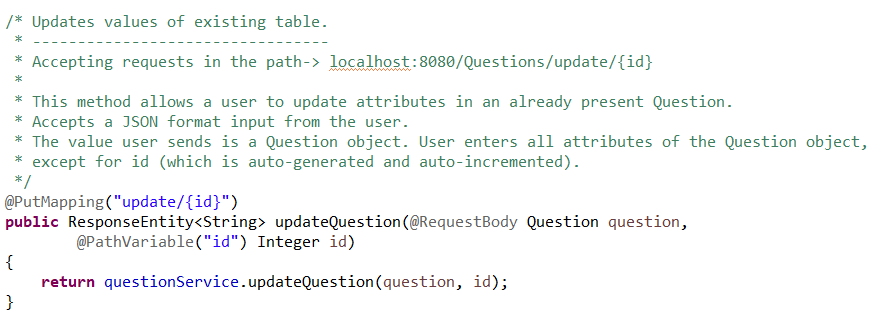
* Automatically deserializes the HttpRequest’s body, which means extracting the information out of the body and converting it to a class object.
* Here, **question** is the object of the class **Question.java**. The annotation maps the data to this class object.

This method calls **addQuestion()** of **QuestionController.java** and passes the object. 

It calls the **save()** in **QuestionDAO**.   
save() is inherited by QuestionDAO from the **CrudRepository** interface. It saves the entities/objects to the table *question\_copy\_table*. It makes the object persist in the table.

1. **updateQuestion()**

**Path:** localhost:8080/Questions/update/<preferred\_id>



**@PutMapping** annotation is used to handle UPDATE requests.

This method allows user to update a certain attribute of the Question object, already present in the table: question\_copy\_table.

User needs to send the **id** for question they want to update and the data in JSON format.

For example, we want to the update the “category” attribute of the question with id = 16. Presently the category is “JAVA”, and we want it to be “Java”. We can update individual or multiple attributes at once.

We would need to send the data in this format:

{

"category": "Java",

"difficultyLevel": "Easy",

"questionTitle": "How many primitive data types are there in Java?",

"option1": "6",

"option2": "7",

"option3": "8",

"option4": "9",

"rightAnswer": "8. They are: int, short, double, float, long, char, boolean, byte"

}

We can’t change the attribute id because it’s auto-generated and auto-incremented. This method accepts the id and the Question object, which it then passes on to **QuestionController**’s **updateQuestion()**.



Here, first we need to find if the id mentioned by user, even exists in the table or not.

For this, we need to create an object of **Optional** class, **optionalObject**.

The point of using Optional here is to prevent and handle **NullPointerException**.

Its job is to store a non-null object. In this case, if the id is present in the table, it will store the corresponding **Question** object or else it’ll be empty. **findById()** searches the table for the mentioned id.

We can check if **optionalObject** stored has any value by using ***isPresent()***. If ***isPresent()*** returns true we can use ***get()*** to fetch the object.

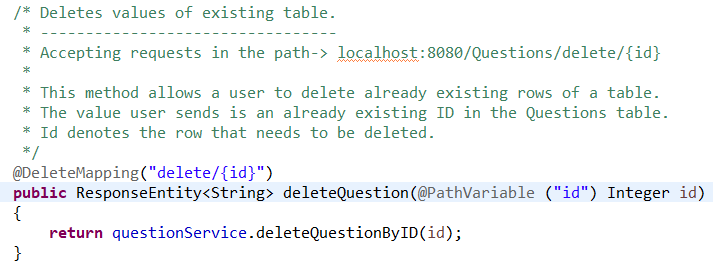
Here, since we have used try-catch blocks, we don’t need to check using *isPresent()*. If optionalObject is indeed empty, control will flow to the catch block and a Bad Request with Status Code 400 will be returned.

Once we find that the **id** exists, we make sure to set the present value of the attributes to the ones provided by the user.

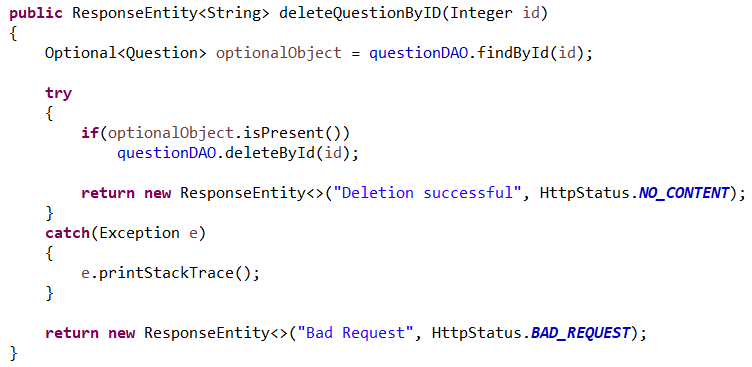
After that, we need to make sure the changes persist in the table. So we save the object using ***save()***.

1. **deleteQuestion()**

**Path:** localhost:8080/Questions/delete/<preferred\_id>



**@DeleteMapping** annotations are used to handle DELETE requests. It takes the already existing id and sends the id over to **QuestionService**’s **deleteQuestionByID()**.



Here, we first search for the id in the table. If present, we delete the Question object associated with it and return the respective status code. If it is not present, we return Status Code 400 and Bad Request message.

We’ll start with Quiz Service next.

**Quiz Service**:

This is Quiz Model class. This is an entity class as denoted by the **@Entity** annotation.

