**Trial Project: Backend (BookMyAppointment)**

**You are allowed to use ChatGPT / any other Gen AI platforms to generate the Code.**

**This project uses following software’s:**

* **SQL and relational databases**
* **Spring core and Spring-boot**
* **Data access layer and service layer**
* **Microservices**
* **Controller layer**
* **Exception handling and application security**
* **Communication between microservices**

**As you see in the above segment that, at its core, this application allows anyone from anywhere in the world to consult a medical specialist. To elaborate this further, the application has the following functionalities:**

* **Admin can add various medical specialists to the application**
* **Admin can get user details**
* **Users (someone who’s looking to consult a doctor) can register by filling in their details**
* **Users can log in using email and password**
* **Users can look for medical specialists and their available time slots**
* **Users can book appointments**
* **Users can look at all the appointments that they have made**
* **Users can rate a doctor**
* **Users can log out**

**Your tasks would revolve around writing the code that would improve your understanding of the following topics:**

* **Controller layer**
* **Data access layer**
* **Service layer**
* **Exception handling**

**In the next segment, you will get an overview of the project code.**

**Code Guide**

**In this segment, you will get an overview of the project code.**

**Reference:** [**https://github.com/ganeshsonar90/bookmyconsultation/blob/master/src/main/java/com/upgrad/bookmyconsultation/service/AppointmentService.java**](https://github.com/ganeshsonar90/bookmyconsultation/blob/master/src/main/java/com/upgrad/bookmyconsultation/service/AppointmentService.java)

**The package structure and its purpose in life are as follows:**

* ***com.upgrad.bookmyconsultation***
  + **Contains the main class that starts the application**

* ***com.upgrad.bookmyconsultation.config***
  + ***ApiConfiguration.java*: Tells Spring-boot to scan the *com.upgrad.bookmyconsultation* package along with all the sub-packages.**
  + ***SwaggerConfiguration.java*: Generates API endpoint documentation**

* ***com.upgrad.bookmyconsultation.constants***
  + **Contains constants that will be used throughout the project**

* ***com.upgrad.bookmyconsultation.controller***
  + **Contains endpoints and their corresponding responses can be found below**

* ***com.upgrad.bookmyconsultation.entity***
  + **Contains entities that are used throughout the project**

* ***com.upgrad.bookmyconsultation.enums***
  + ***Specialty.java*: Contains an enum with various specialties.**
  + ***UserAuthTokenStatus.java*: Contains an enum with authentication token status values**

* ***com.upgrad.bookmyconsultation.exception***
  + **Contains classes that handle various types of exceptions**

* ***com.upgrad.bookmyconsultation.handler***
  + ***RestExceptionHandler.java*: Handles specific exceptions and sends responses to the user**
* ***com.upgrad.bookmyconsultation.model***
  + ***AuthorizedUser.java*: POJO for a model that contains authorised user’s information**
  + ***TimeSlot.java*: The TimeSlot constructor returns a list of available time slots (returns a random number of time slots with random time slot values)**

* ***com.upgrad.bookmyconsultation.provider***
  + ***BasicAuthDecoder.java*: Contains a method to decode an email and password**
  + ***BearerAuthDecoder.java*: Contains a method to decode the token**
  + ***PasswordCryptographyProvider.java*: Contains methods that hash and salt the passwords to secure them**
  + ***TokenCrytographyProvider.java*: Contains methods to encrypt and decrypt the tokens**

* ***com.upgrad.bookmyconsultation.provider.token***
  + ***JwtTokenProvider.java*: Contains a method to generate the token**
  + ***Token.java*: Contains the specification that defines the generated token**
  + ***TokenProvider.java*: Contains the declaration for methods that serialise and deserialise tokens**

* ***com.upgrad.bookmyconsultation.repository***
  + **Provides CRUD operations**

* ***com.upgrad.bookmyconsultation.service***
  + ***AppointmentService.java*: Contains a method that saves an appointment to the database. Also contains a method that allows retrieving appointments from the database**
  + ***AuthenticationService.java*: Allows the user to be authenticated based on email id and password. Generates a token for the user once the email id and password are authenticated**
  + ***AuthTokenService.java*: Contains the actual logic for token generation. Also contains methods to validate and invalidate the tokens**
  + ***DoctorService.java*: Contains methods that register a doctor, returns a doctor object based on the id, and returns doctors belonging to a specific speciality. Also contains methods that return doctors that are sorted by ratings. Finally, contains a method that returns time slots.**
  + ***RatingsService.java*: Saves rating to the database. Adds the newly added rating to the average ratings and recalculates the average**
  + ***UserAuthTokenVerifier.java*: Stores the token status**
  + ***UserService.java*: Contains various methods for the user such as registering a user to the database. Contains the *getUser* and *getAllUsers* methods to extract the information for a single user and all users from the database, respectively.**

* ***com.upgrad.bookmyconsultation.util***
  + ***ValidationUtils.java*: Contains code for validating user, doctor, and appointment details. For example, ensuring that a phone number contains only 10 digits.**

**In the next session, you will learn the tasks that you need to complete.**

**Overview**

**Welcome to the session on ‘Tasks and Approach’.**

**In this session, you will learn the tasks that you need to complete. You will also understand the approach that you must adopt to complete these tasks.**

**You are required to complete six tasks:**

* **Project setup**
* **Database schema**
* **Controller layer**
* **Data access layer**
* **Service layer**
* **Exception handling**

**In the next segment, you will learn the approach that you need to adopt to complete the project setup task.**

**Project setup**

**So, first things first, you need to set up the project on your system before you can start coding.**

**You will require the following tools to set up the project:**

* **IDE**
* [**MySQL Workbench**](https://airlock-on-edge.woolf.university/?url=https%3A%2F%2Fdev.mysql.com%2Fdownloads%2Fmysql%2F&resourceId=5649970&studentId=37746437-4865-4301-b4c6-2ec28dc342ab&token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjM3NzQ2NDM3LTQ4NjUtNDMwMS1iNGM2LTJlYzI4ZGMzNDJhYiIsImlzVmVyaWZpZWQiOnRydWUsImtpbmQiOiJvYXV0aCIsIm9yZyI6eyJncm91cHMiOltdLCJpZCI6Ijk3ZjhjNTRmLWVjZjctNGY1YS1iNGQ3LWM4NWEwMDI2ZGEwNyJ9LCJzY29wZSI6IioiLCJpYXQiOjE3NDg5Mzk0NDcsImV4cCI6MTc0OTU0NDI0NywiaXNzIjoidXJuOldvb2xmVW5pdmVyc2l0eTpzZXJ2ZXIvdXNlci9hY2Nlc3MifQ.aElHxxz91wB3BCqj0ffkje01OwYpc0OZjD1CwKHuSkg)
* **Java 11 SDK or higher (LTS version preferred)**
* **Apache Maven 3.3 build tool or higher**
* **Postman (optional)**
* **Coffee**
* **Project file**

**Project file**

**Download**

**Once you have the tools mentioned above, you can proceed to set up your machine.**

1. **The first step is to import the project into your IDE.**
2. **Once you have imported the project, head over to this file named ‘DBLoadScript.sql’ inside the folder path ‘src/main/resources’.**
3. **Inside this file, you will find MySQL scripts.**
4. **These scripts will ensure that the database schema created on your machine is consistent with that of our graders.**
5. **Open Workbench, create a database named ‘consultation’, and execute the scripts from the file ‘DBLoadScript.sql’ to create various tables and fill some of those tables with data.**
6. **You can use the ‘application.properties’ file inside the ‘src/main/resources’ to update the ‘username’ and the ‘password’ for the database you just created. You can also use the ‘application.properties’ file to add/edit other properties.**
7. **The next step is to head back to your IDE and build the project using the maven command ‘mvn clean install -DskipTests’. This will install all the dependencies required to run the project.**
8. **You can run the project using the command ‘mvn spring-boot:run’.**

**The project probably will show issues as it is incomplete and will only work end-to-end once you complete the ‘TODOs’ inside the project.**

**You can also refer to the file named ‘README.md’ inside ‘src/main/resources’ for more information.**

**Now that you have completed the project setup, you can begin coding.**

**In the next segment, you will learn the approach you need to adopt to complete the database schema task.**

**Database Schema**

**Your first task is to create documentation for the database schema by referring to the consultation database and all its relations inside the Workbench tool.**

**You should use the following format:**

**relation-name(column1: type, column2: type, ...columnN:type)**

**In the next segment, you will learn the approach required to complete the controller layer task.**

**Controller Layer**

**Inside the controller package, there are five essential files, namely:**

* ***AppointmentController.java***
* ***AuthenticationController.java***
* ***DoctorController.java***
* ***RatingsController.java***
* ***UserAdminController.java***

**You will create methods for the classes *AppointmentController,***

***RatingsController*and*UserAdminController*.**

**Head over to the file *AppointmentController.java*. You are required to create two endpoints */appointments* and */appointmentsId* that allow a user to book an appointment and get their existing appointments, respectively.**

**For the endpoint */appointments*, create a post method named *bookAppointment* that returns an object of type *ResponseEntity<String>***

**and uses *@RequestBody* to have your values mapped to the model we created for you. Head over to the entity package and select an appropriate model to be used for this method. The body of this newly created method should ensure that the appointment is saved to the database. Now, head over to the *AppointmentService* class inside the *service* package. Here, the *appointment* method might come in handy.**

**Similarly, for the endpoint */appointmentsId*, you are required to create a method named *getAppointment*. The*getAppointment* method inside the *AppointmentService* class might come in handy to complete this method.**

**Great! Now that you have completed the *AppointmentController* class, let’s move on to the *RatingsController* class. Here, you are required to create one endpoint named */ratings* that will allow a user to submit ratings.**

**Create a method named *submitRatings* that takes an object of type rating as the parameter and returns a *ResponseEntity* object. Now, head over to the *Rating* class inside the *entity* package. Here, you would notice that the model itself is incomplete. You will have to create a model as well, which we will discuss later. For now, use the *Rating* model without worrying about its underlying details. Inside the *submitRatings* method, you need to call in the method that has the actual logic for submitting a rating. The *submitRatings* method inside the *RatingService* class in the *service* package might come in handy.**

**Great! The *RatingsController* class is sorted as well.**

**Next, move on to the file named *UserAdminController.java*. Here, you are supposed to create a */register* endpoint that registers the user with the details that the user puts in. You will also need to create a model for the *User* entity in the *User.java* file inside the*entity* package which will be similar to the *Rating* model. We will let you figure out this endpoint without providing any additional details.**

**In the next segment, you will learn the approach required to complete the data access layer task.**

**Data Access Layer**

**Remember that you created a database and added tables to the database? Now you need some mechanism to be able to fire DML (Data Manipulation Language) queries from your project to the database. Let’s use *CrudRepository* for this purpose.**

**Before we dive further into the*repository* package, let’s first complete the*Rating*entity inside the *Rating.java* file in the entity package. The *Rating* entity also needs to be mapped to the *rating* table inside the database. The *Rating*entity should have five private properties, namely:**

* ***id* (of type String)**
* ***appointmentId*(of type String)**
* ***doctorId*(of type String)**
* ***rating* (of type Integer)**
* ***comments* (of type String)**

**Make sure you use *@Entity* to mark this class as an *entity* class. The *id* member variable needs to be marked as a primary key, so use an appropriate annotation for this as well.**

**Next, complete the *User* entity. The User entity should have eight private properties, namely:**

* ***firstName* (of type String)**
* ***lastName* (of type String)**
* ***dob* (of type String)**
* ***mobile* (of type String)**
* ***emailId* (of type String)**
* ***password* (of type String)**
* ***createDate* (of type String)**
* ***salt* (of type String)**

**Alright! Now, move on to the *repository* package. The *repository* package has six files, namely:**

* ***AddressRepository.java***
* ***AppointmentRepository.java***
* ***DoctorRepository.java***
* ***RatingsRepository.java***
* ***UserAuthTokenRepository.java***
* ***UserRepository.java***

**Now, head over to the *RatingsRepository.java*, create an interface named *RatingsRepository* that extends*CrudRepository*. Provide the type information, such as the *entity* type, which is *Rating* in this case, as well as the *id* type. That’s it, you have now enabled your application to make CRUD operations to the rating table. However, we want you to add a functionality that returns *Rating* details for a specific doctor. You need to find this specific doctor by their *id* and retrieve their *Rating* details.**

**Similarly, add the CRUD methods for the *AddressRepository* and the *UserRepository* interfaces. For *UserRepository*, you need to add two additional functionalities: one will allow you to get all the users and the other will allow you to find and get one user using an email id. So, go ahead and open those respective files and add the mentioned functionalities.**

**In the next segment, you will learn the approach required to complete the service layer task.**

**Service Layer**

**For the previous tasks, you used the *CRUDRepository* to easily implement the data layer. Now, you are going to create business services. The *service*package contains seven files, namely:**

* ***AppointmentService.java***
* ***AuthenticationService.java***
* ***AuthTokenService.java***
* ***DoctorService.java***
* ***RatingsService.java***
* ***UserAuthTokenVerifier.java***
* ***UserService.java***

**Here, you shall implement parts of *AppointmentService*, *DoctorService, RatingsService,*and*UserService* classes. You may skip the exception handling part for now. The exception handling part of the project will be discussed in detail in the exception handling task.**

**So, let’s begin with the *AppointmentService* class. The first thing you will need to do is to get the *AppointmentRepository*instance into the *AppointmentService* class so that you can make the calls to the *AppointmentRepository*from the *Appointment* Service. The best way to get an instance into a Spring service class is by *autowiring*. So, go ahead and create a new member variable called *appointmentRepository* of type *AppointmentRepository* and mark it as *Autowired*.**

**Next, you will create a method named appointment with a return type String and a method parameter of type *Appointment*. The purpose of this method is to book an appointment.**

**The method parameter is basically the appointment details that the user would send. You would need to validate these details first. The validate method from the *ValidateUtils* class in the*util* package might come in handy. Another validation that you need to ensure is to check if the doctor already has any existing appointment with the same doctor for that specific time and date. Your newly created *AppointmentRepository*instance might help check this. If there is no existing appointment, save the appointment details to the database.**

**Next, you need to create a method named *getAppointment* that returns appointment details after getting an appointment id. We will leave the rest for you to figure out.**

**Well that was a lot of fun, right?**

**Next move on to *DoctorService.java*. Here, you will write two methods. The first method is named as *register* and it returns as well as has a parameter of type *Doctor*. The method, as the name suggests, will allow the admin to register doctors to the application. So, it is similar to the appointment method where you validate the details. You also need to assign an ID to every new doctor that you add so that you can uniquely identify each doctor later. The *setId* method will help set a unique ID to a doctor. The *UUID* functionality within the *java.util* library might make your life easier by implementing UUIDs for the doctors. Another check that you need to make is that if the admin does not specify any speciality for the doctor, it should be set to ‘GENERAL\_PHYSICIAN’. You might want to take a look at the *enums* package, where we have implemented enums for you. Make sure that you save the doctor’s address details in the address table and the rest of the doctor’s details in the doctor table.**

**Another method that you will write in the *DoctorService* class is a method named *getDoctor* that returns doctor details after getting the doctor id as the input.**

**Done with the *DoctorService* class? Great!**

**Now, open the file named *RatingsService.java* in the service package. In the *RatingsService* class, you need to write a method named *submitRatings* that gets a parameter of type *Ratings*. The goal of this method is to save the doctor ratings in the ratings table as well as calculate and update the new average rating (in the doctor table) when a new rating is submitted for that specific doctor.**

**Finally, in the *UserService* class, write a method named*getAllUsers* that returns all the users present in the database.**

**In the next segment, you will learn the approach required to complete the exception handling task.**

**Exception Handling**

**So far, you have implemented the layers and have assumed that things are going to work as if you are in an ideal world. But what if something does not work as it is expected to work or if there is some exception? You would want some kind of an error code or an error message that is a proper JSON response and not some random HTML page that the Tomcat server throws out.**

**Your end goal is to throw exceptions in the *AppointmentService* class if certain specific conditions occur.**

**So, head over to the *AppointmentService* class and modify the *appointment* method. The *appointment*method should throw two exceptions, namely *SlotUnavailableException* and *InvalidInputException*. The *throws* keyword might come in handy.**

**Remember that inside the appointment method, you are checking if an existing appointment exists for a specific slot. If it does exist, you should not let the user book another appointment for the same slot and throw an exception. This is the part where you will throw the exception using *SlotUnavailableException()* and the *throw* keyword.**

**Next, head over to the *handler* package and open *RestExceptionHandler.java*. We have annotated the *RestExceptionHandler*class with *ControllerAdvice* as we want the exception techniques to be applied across the whole application. Inside this class, you shall create exception handling methods annotated with *@ExceptionHandler*. Now, create a new method named *handleSlotUnavailableException* with*ResponseEntity* as its return type. You need to mention the type of exception that you are handling by mentioning it in the parameter of the *ExceptionHandler* annotation. We have already created a class for you named *SlotUnavailableException* that extends *RuntimeException* in the*exception* package. You need to specify this exception here. In the body of this method, you need to return a custom HTTP response using *ResponseEntity*. The following coding snippet might come in handy here.**

**return ResponseEntity .badRequest() .body(new ErrorResponse(HttpStatus.BAD\_REQUEST, e.getDescription()));**

**We have defined the *ErrorResponse* class in the *controller.ext* package that defines the structure of the response.**

**Task Summary**

**Once you have completed this Task.**

**Share the following files:**

* **Answer.pdf: It is a document that contains images/screenshots for the API responses.**
* **Solution Code**