Lecture Notes

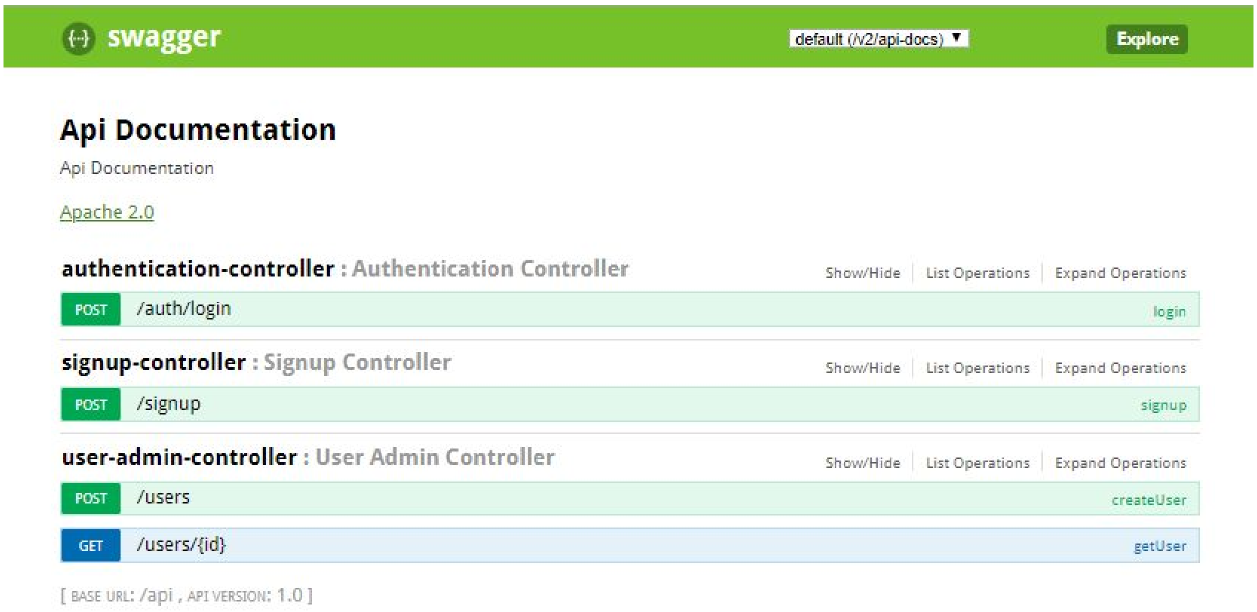
Creating an API Backend - II

Step I: Introduction

Introduction to RESTful APIs

- RESTful API is an application program interface using HTTP Requests to POST, PUT, GET and DELETE data.
- An API endpoint is like a remote function that you can call from another application, with a backend server. For example, a client application, like a web browser, can call an API endpoint to interact with data resources represented and managed by the backend server.
- Why are we using RESTful APIs? The major advantage of using API endpoints as the backend is that it allows you to build different websites and applications on the top of the same backend.

Here is an image of all the API endpoints that we have implemented in the 'Proman' project



Api Documentation

Api Documentation

Apache 2.0

authentication-controller : Authentication Controller — Show/Hide | List Operations | Expand Operations

POST /auth/login login

signup-controller : Signup Controller — Show/Hide | List Operations | Expand Operations

POST /signup signup

user-admin-controller : User Admin Controller — Show/Hide | List Operations | Expand Operations

POST /users createUser

GET /users/{id} getUser

[BASE URL: /api , API VERSION: 1.0]

Figure 1 - Endpoints on Swagger

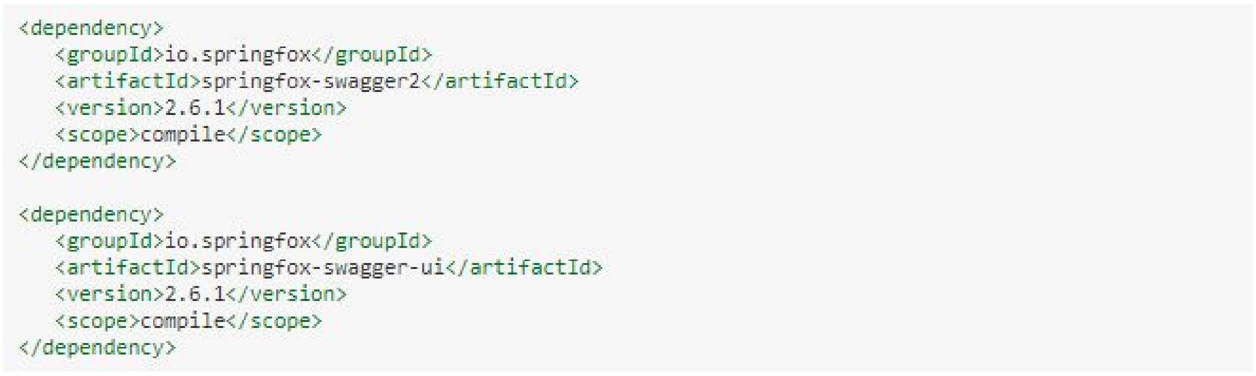
You can build different websites using the same API endpoints and hit any API endpoint to obtain the required functionality.

Introduction to Swagger and JSON

- A swagger is a tool used by product managers and developers to define API endpoints, after which the development team will develop the APIs based on the provided definitions.
- Product managers generally create the schema in JSON format , as in the ‘Proman’ project, for API endpoints to be developed specifying the request patterns, Http status codes, request and response parameters, etc. These JSON files are then read by the developers using Swagger and the API endpoints are developed accordingly.

‘Signup.json’ file in the ‘Proman’ project is provided by the Product Manager and contains all the details about the signup endpoint. As a developer, you need to observe all the details by reading this JSON file on Swagger. This file is then used to generate Request and Response models using Swagger. Request models are used to map the input JSON fields to the Java attributes and Response model maps the Java attributes to the JSON output fields. Import this file in ‘proman-api’ submodule, build the project and you will see the request and response models generated.

The dependencies to add Swagger configuration in Spring are as shown below:



```
<dependency>
   <groupId>io.springfox</groupId>
   <artifactId>springfox-swagger2</artifactId>
   <version>2.6.1</version>
   <scope>compile</scope>
</dependency>

<dependency>
   <groupId>io.springfox</groupId>
   <artifactId>springfox-swagger-ui</artifactId>
   <version>2.6.1</version>
   <scope>compile</scope>
</dependency>
```

Figure 2 - Dependencies for Swagger

Introduction to POM

- A project contains configuration files, the project’s dependencies and all other little pieces that come into play to give code life.
- POM is actually a one-stop-shop for all the things that are related to the project. It declares the project resources like 3rd party library dependencies (jar files used in the project as the library) using XML format.
- While a task or a goal (a basic unit of work in Maven representing a specific task, thus contributing to build the project) is executed, maven reads the POM in the current directory, gets the needed configuration information and then executes the goal.

We will be creating three submodules in the project and all of them will be having individual pom files. We will be adding the required plugins in the pom file as we proceed with the project.

Step II: Creating signup endpoint

Initial Basic controller class for signup endpoint

What role does the Request and Response models play?
When the controller class is executed to respond to a client request, we need to pass a java object (request model) to the controller method. These Request models then come into picture which has already been generated using JSON files. These Request models get instantiated (with all the attributes mapped using JSON properties filled by the user in JSON request) and the instance is passed to the controller request mapping method.
Similarly, response models are used to send JSON response. The response model type object is declared and passed as a response after setting all its attributes.

What is a Controller class?
As you may be familiar with Spring MVC concepts, the Dispatcher Servlet scans for all the classes annotated with the @Controller annotation when a client sends a particular request. Therefore, it is essential to have a Controller class in the application such that Dispatcher Servlet scans that controller class and looks for a particular method in the class to serve the request.

Now that you have the response and request models generated in the project, the next step is to work on the **signup controller class**.

Annotations used:

1. **@RestController** - This annotation combines two annotations @Controller and @ResponseBody.
2. **@RequestMapping** - This annotation is used with classes and methods to redirect a client request to a specific controller class/method with which the request matches.

The initial code for the basic controller class is as shown below:

```
@RequestMapping(method= RequestMethod.POST, path="/signup", consumes=
MediaType.APPLICATION_JSON_UTF8_VALUE, produces=MediaType.APPLICATION_JSON_UTF8_VALUE)
public ResponseEntity<SignupUserResponse> signup(final SignupUserRequest signupUserRequest)
{
   return new ResponseEntity(HttpStatus.OK);
}
```

Database and JPA properties

Now since we are storing all the data in the database, therefore we need to mention all the database configuration properties. Also, we need to mention the JPA properties since we are using JPA for object-relational mapping. YAML is a superset of JSON and, hence, is a convenient syntax for storing external properties in a hierarchical format.

The external properties stored in ‘application.yaml’ file are as shown below:

```yaml
server:
 servlet:
   port: 8080
   contextPath: /api

spring:

 application:
   name: proman-api

 datasource:
   driverClassName: org.postgresql.Driver
   url: jdbc:postgresql://localhost:5432/postgres
   username: postgres
   password: postgres@123

 jpa:
   properties:
     hibernate:
       temp:
         use_jdbc_metadata_defaults: false
   database-platform: org.hibernate.dialect.PostgreSQL9Dialect
```

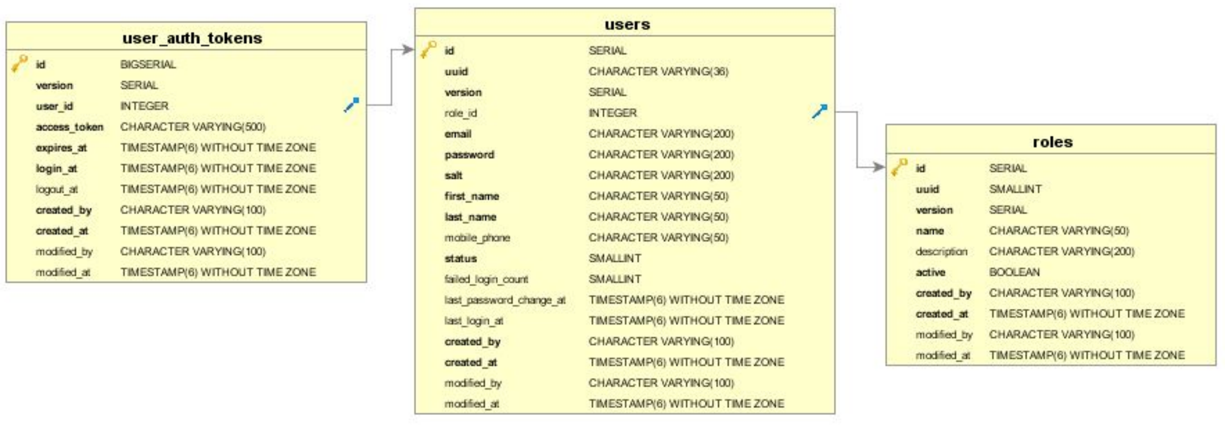
Creating the database

Now, let us focus on creating the database for the Proman project and configure it with the project. Make a submodule (proman-db) for handling the database and its configuration in your project.

The configuration file is implemented with the name 'localhost.properties' which are subject to the change based on the username and password of your database.

```
# environment properties for local development

server.host = localhost
server.port = 5432

database.name = postgres
database.user = postgres
database.password = postgres@123
```

Let us discuss the database schema which is to be implemented in the 'Proman' project. 'users' table stores all the details of the user, 'user_auth_tokens' table stores the authentication details of the signed in user, and 'roles' table stores the details of the possible role a user can have.
The user_id column in 'user_auth_tokens' table references the primary key id in the 'users' table. The role_id column in 'users' table references the primary key id in the 'roles' table. Observe the columns in all the tables in the database as shown below:



user_auth_tokens	
id	BIGSERIAL
version	SERIAL
user_id	INTEGER
access_token	CHARACTER VARYING(500)
expires_at	TIMESTAMP(6) WITHOUT TIME ZONE
login_at	TIMESTAMP(6) WITHOUT TIME ZONE
logout_at	TIMESTAMP(6) WITHOUT TIME ZONE
created_by	CHARACTER VARYING(100)
created_at	TIMESTAMP(6) WITHOUT TIME ZONE
modified_by	CHARACTER VARYING(100)
modified_at	TIMESTAMP(6) WITHOUT TIME ZONE

users	
id	SERIAL
uuid	CHARACTER VARYING(36)
version	SERIAL
role_id	INTEGER
email	CHARACTER VARYING(200)
password	CHARACTER VARYING(200)
salt	CHARACTER VARYING(200)
first_name	CHARACTER VARYING(50)
last_name	CHARACTER VARYING(50)
mobile_phone	CHARACTER VARYING(50)
status	SMALLINT
failed_login_count	SMALLINT
last_password_change_at	TIMESTAMP(6) WITHOUT TIME ZONE
last_login_at	TIMESTAMP(6) WITHOUT TIME ZONE
created_by	CHARACTER VARYING(100)
created_at	TIMESTAMP(6) WITHOUT TIME ZONE
modified_by	CHARACTER VARYING(100)
modified_at	TIMESTAMP(6) WITHOUT TIME ZONE

roles	
id	SERIAL
uuid	SMALLINT
version	SERIAL
name	CHARACTER VARYING(50)
description	CHARACTER VARYING(200)
active	BOOLEAN
created_by	CHARACTER VARYING(100)
created_at	TIMESTAMP(6) WITHOUT TIME ZONE
modified_by	CHARACTER VARYING(100)
modified_at	TIMESTAMP(6) WITHOUT TIME ZONE

Figure 3 - Proman Database

We have implemented all the SQL queries. You need to run these SQL queries against the database. The option -P is used to activate the profile in pom file under 'profile' tag. The command 'mvn clean install -Psetup' will be used to activate the profile with id 'setup' whenever you need to seed the database. You may observe the database and the corresponding tables generated.

Add the required Maven plugins, configurations in the pom.xml file of ‘proman-db’ and generate the database tables on the basis of SQL queries.

Entity classes for signup endpoint

Now that you have created the database and controller class, its time to create the Java entity classes to map them with the database
Make a submodule ‘proman-service’ and import the pom file in the ‘proman-service’ submodule.
Now in order to map the three tables in the database, we will create three entity classes to map the attributes with the database fields. The entity classes to be created are UserEntity, UserAuthTokenEntity, and RoleEntity.

Annotations used:

1. **@Entity** annotation denotes this class as a JPA entity.
2. **@Table** annotation allows you to specify the details of the table.
3. **@Id** annotation is to specify a particular column as the primary key for the table.
4. **@NotNull** is used for the validation of the field specifying that this field cannot remain null.
5. **@Size** is used for the validation of the value range for a field.
6. **@Version** is used to ensure the data consistency.
7. **@ToStringExclude** is used to prevent the sensitive information from being displayed when you call the toString() on the entity.

The code for UserEntity class is as shown below:

```
@Entity
@Table(name = "users", schema = "proman")
@NamedQueries(
       {
               @NamedQuery(name = "userByUuid", query = "select u from UserEntity u where
u.uuid = :uuid"),
               @NamedQuery(name = "userByEmail", query = "select u from UserEntity u where
u.email =:email")
       }
)


public class UserEntity implements Serializable {

   @Id
   @Column(name = "ID")
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Integer id;

   @Column(name = "UUID")
   @Size(max = 64)
   private String uuid;

   @ManyToOne
   @JoinColumn(name = "ROLE_ID")
   private RoleEntity role;

   @Column(name = "EMAIL")
   @NotNull
   @Size(max = 200)
   private String email;

   //@ToStringExclude
```

```
@Column(name = "PASSWORD")
private String password;

@Column(name = "FIRST_NAME")
@NotNull
@Size(max = 200)
private String firstName;

@Column(name = "LAST_NAME")
@NotNull
@Size(max = 200)
private String lastName;

@Column(name = "MOBILE_PHONE")
@NotNull
@Size(max = 50)
private String mobilePhone;

@Column(name = "STATUS")
@NotNull
private int status;

@Column(name = "FAILED_LOGIN_COUNT")
@Min(0)
@Max(5)
private int failedLoginCount;

@Column(name = "LAST_PASSWORD_CHANGE_AT")
private ZonedDateTime lastPasswordChangeAt;

@Column(name = "LAST_LOGIN_AT")
private ZonedDateTime lastLoginAt;

@Column(name = "SALT")
@NotNull
@Size(max = 200)
//@ToStringExclude
private String salt;

@Version
@Column(name = "VERSION", length = 19, nullable = false)
private Long version;


@Column(name = "CREATED_BY")
@NotNull
private String createdBy;


@Column(name = "CREATED_AT")
@NotNull
private ZonedDateTime createdAt;

@Column(name = "MODIFIED_BY")
private String modifiedBy;

@Column(name = "MODIFIED_AT")
private ZonedDateTime modifiedAt;


@Override
public boolean equals(Object obj) {
    return new EqualsBuilder().append(this, obj).isEquals();
}

public Integer getId() {
    return id;
}

public void setId(Integer id) {
    this.id = id;
}

public String getUuid() {
    return uuid;
}

public void setUuid(String uuid) {
    this.uuid = uuid;
}

public RoleEntity getRole() {
    return role;
}

public void setRole(RoleEntity role) {
    this.role = role;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

public String getMobilePhone() {
    return mobilePhone;
}

public void setMobilePhone(String mobilePhone) {
    this.mobilePhone = mobilePhone;
}

public int getStatus() {
    return status;
}

public void setStatus(int status) {
    this.status = status;
}

public int getFailedLoginCount() {
    return failedLoginCount;
}

public void setFailedLoginCount(int failedLoginCount) {
    this.failedLoginCount = failedLoginCount;
}

public ZonedDateTime getLastPasswordChangeAt() {
    return lastPasswordChangeAt;
}

public void setLastPasswordChangeAt(ZonedDateTime lastPasswordChangeAt) {
    this.lastPasswordChangeAt = lastPasswordChangeAt;
}

public ZonedDateTime getLastLoginAt() {
    return lastLoginAt;
}

public void setLastLoginAt(ZonedDateTime lastLoginAt) {
    this.lastLoginAt = lastLoginAt;
}

public String getSalt() {
    return salt;
}

public void setSalt(String salt) {
    this.salt = salt;
}

public Long getVersion() {
    return version;
}

public void setVersion(Long version) {
    this.version = version;
}

public String getCreatedBy() {
    return createdBy;
}

public void setCreatedBy(String createdBy) {
    this.createdBy = createdBy;
}

public ZonedDateTime getCreatedAt() {
    return createdAt;
}

public void setCreatedAt(ZonedDateTime createdAt) {
    this.createdAt = createdAt;
}

public String getModifiedBy() {
    return modifiedBy;
}

public void setModifiedBy(String modifiedBy) {
    this.modifiedBy = modifiedBy;
}
public ZonedDateTime getModifiedAt() {
    return modifiedAt;
}

public void setModifiedAt(ZonedDateTime modifiedAt) {
    this.modifiedAt = modifiedAt;
}

@Override
public int hashCode() {
    return new HashCodeBuilder().append(this).hashCode();
}

@Override
public String toString() {
    return ToStringBuilder.reflectionToString(this, ToStringStyle.MULTI_LINE_STYLE);
}


}
```

The code for UserAuthTokenEntity class is as shown below:

```
@Entity
@Table(name = "user_auth_tokens", schema = "proman")
@NamedQueries({
        @NamedQuery(name = "userAuthTokenByAccessToken", query = "select ut from
UserAuthTokenEntity ut where ut.accessToken = :accessToken ")
})
public class UserAuthTokenEntity implements Serializable {


    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @ManyToOne
    @JoinColumn(name = "USER_ID")
    private UserEntity user;

    @Column(name = "ACCESS_TOKEN")
    @NotNull
    @Size(max = 500)
    private String accessToken;

    @Column(name = "LOGIN_AT")
    @NotNull
    private ZonedDateTime loginAt;

    @Column(name = "EXPIRES_AT")
    @NotNull
    private ZonedDateTime expiresAt;

    @Column(name = "LOGOUT_AT")
    private ZonedDateTime logoutAt;


    @Version
    @Column(name = "VERSION", length = 19, nullable = false)
    private Long version;
```

```
@Column(name = "CREATED_BY")
@NotNull
private String createdBy;


@Column(name = "CREATED_AT")
@NotNull
private ZonedDateTime createdAt;

@Column(name = "MODIFIED_BY")
private String modifiedBy;

@Column(name = "MODIFIED_AT")
private ZonedDateTime modifiedAt;

public Integer getId() {
    return id;
}

public void setId(Integer id) {
    this.id = id;
}

public UserEntity getUser() {
    return user;
}

public void setUser(UserEntity user) {
    this.user = user;
}

public String getAccessToken() {
    return accessToken;
}

public void setAccessToken(String accessToken) {
    this.accessToken = accessToken;
}

public ZonedDateTime getLoginAt() {
    return loginAt;
}

public void setLoginAt(ZonedDateTime loginAt) {
    this.loginAt = loginAt;
}

public ZonedDateTime getExpiresAt() {
    return expiresAt;
}

public void setExpiresAt(ZonedDateTime expiresAt) {
    this.expiresAt = expiresAt;
}

public ZonedDateTime getLogoutAt() {
    return logoutAt;
}

public void setLogoutAt(ZonedDateTime logoutAt) {
    this.logoutAt = logoutAt;
}

public Long getVersion() {
    return version;
    }

    public void setVersion(Long version) {
        this.version = version;
    }

    public String getCreatedBy() {
        return createdBy;
    }

    public void setCreatedBy(String createdBy) {
        this.createdBy = createdBy;
    }

    public ZonedDateTime getCreatedAt() {
        return createdAt;
    }

    public void setCreatedAt(ZonedDateTime createdAt) {
        this.createdAt = createdAt;
    }

    public String getModifiedBy() {
        return modifiedBy;
    }

    public void setModifiedBy(String modifiedBy) {
        this.modifiedBy = modifiedBy;
    }

    public ZonedDateTime getModifiedAt() {
        return modifiedAt;
    }

    public void setModifiedAt(ZonedDateTime modifiedAt) {
        this.modifiedAt = modifiedAt;
    }

    @Override
    public boolean equals(Object obj) {
        return new EqualsBuilder().append(this, obj).isEquals();
    }

    @Override
    public int hashCode() {
        return new HashCodeBuilder().append(this).hashCode();
    }

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this, ToStringStyle.MULTI_LINE_STYLE);
    }
}
```

The code for RoleEntity class is as shown below:

```
@Entity
@Table(name = "roles", schema = "proman")
public class RoleEntity implements Serializable {

   @Id
   @Column(name = "ID")
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Integer id;

   @Column(name = "UUID")
   @NotNull
   private int uuid;

   @Column(name = "NAME")
   @NotNull
   @Size(max = 50)
   private String name;

   @Column(name = "DESCRIPTION")
   @Size(max = 200)
   private String description;

   @Column(name = "ACTIVE")
   @NotNull
   private boolean active;


   @Version
   @Column(name = "VERSION", length = 19, nullable = false)
   private Long version;


   @Column(name = "CREATED_BY")
   @NotNull
   private String createdBy;


   @Column(name = "CREATED_AT")
   @NotNull
   private ZonedDateTime createdAt;

   @Column(name = "MODIFIED_BY")
   private String modifiedBy;

   @Column(name = "MODIFIED_AT")
   private ZonedDateTime modifiedAt;


   public int getId() {
       return id;
   }

   public void setId(int id) {
       this.id = id;
   }

   public int getUuid() {
       return uuid;
   }

   public void setUuid(int uuid) {
       this.uuid = uuid;
```

```
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public boolean isActive() {
        return active;
    }

    public void setActive(boolean active) {
        this.active = active;
    }

    public Long getVersion() {
        return version;
    }

    public void setVersion(Long version) {
        this.version = version;
    }

    public String getCreatedBy() {
        return createdBy;
    }

    public void setCreatedBy(String createdBy) {
        this.createdBy = createdBy;
    }

    public ZonedDateTime getCreatedAt() {
        return createdAt;
    }

    public void setCreatedAt(ZonedDateTime createdAt) {
        this.createdAt = createdAt;
    }

    public String getModifiedBy() {
        return modifiedBy;
    }

    public void setModifiedBy(String modifiedBy) {
        this.modifiedBy = modifiedBy;
    }

    public ZonedDateTime getModifiedAt() {
        return modifiedAt;
    }

    public void setModifiedAt(ZonedDateTime modifiedAt) {
        this.modifiedAt = modifiedAt;
    }

   /* public RoleEntity(@NotNull int uuid) {
        this.uuid = uuid;
    }*/

    @Override
    public boolean equals(Object obj) {
        return new EqualsBuilder().append(this, obj).isEquals();
    }

    @Override
    public int hashCode() {
        return new HashCodeBuilder().append(this).hashCode();
    }

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this, ToStringStyle.MULTI_LINE_STYLE);
    }
}
```

Business logic

Now the database is implemented and the entity classes are also implemented, we need to implement the code to map the Java objects to the database tables. The controller class receives the data from the request, transforms the data into the corresponding entities and pass them to the business layer.

Annotations used:

1. **@Service** annotation is a specialized version of @Component. Although @Component annotation can be used here in place of @Service but @Service specifies the special functionality as a service layer handling the business logic.

The code for the Service logic is as shown below:

```
@Service
public class SignupBusinessService {
   @Autowired
   private UserAdminBusinessService userAdminBusinessService; @Transactional(propagation =
Propagation.REQUIRED)
   public UserEntity signup(UserEntity userEntity) {
       return userAdminBusinessService.createUser(userEntity);
   }
}
```

The above code starts a transaction to persist a user in the database and simply calls the DAO logic passing the user entity to be persisted in the database.

Dealing with the database

After implementing the business logic, the service class needs to interact with the database to persist the data and save the data to the database. Let us understand another important aspect called the data access layer, which interacts with the database and it is a good coding practice to make a separate layer to interact with the database.

Let us implement the DAO logic
Annotations used:

1. **@Repository** is a specialized version of @Component, which gives the DAO class the functionalities to interact with the database.

The code for the Service logic is as shown below:

```
@Repository
public class UserDao {


   @PersistenceContext
   private EntityManager entityManager;

   public UserEntity createUser(UserEntity userEntity) {
       entityManager.persist(userEntity);
       return userEntity;
   }
```

The above code declares an instance of Spring-managed Entity Manager and uses the entity manager to persist the user entity in the database using the persist() method. An entity manager instance is required to manage the entity objects.

Controller class

Now the code for the Controller class can be modified as shown below:

```
@RestController
@RequestMapping("/")
public class SignupController {

   @Autowired
   private SignupBusinessService signupBusinessService;
   @RequestMapping(method= RequestMethod.POST, path="/signup", consumes=
MediaType.APPLICATION_JSON_UTF8_VALUE, produces=MediaType.APPLICATION_JSON_UTF8_VALUE)
   public ResponseEntity<SignupUserResponse> signup(final SignupUserRequest signupUserRequest)
   {
       final UserEntity userEntity = new UserEntity();
       userEntity.setUuid(UUID.randomUUID().toString());
       userEntity.setFirstName(signupUserRequest.getFirstName());
       userEntity.setLastName(signupUserRequest.getLastName());
       userEntity.setEmail(signupUserRequest.getEmailAddress());
       userEntity.setPassword(signupUserRequest.getPassword());
       userEntity.setMobilePhone(signupUserRequest.getMobileNumber());
       userEntity.setSalt("1234abc");
       userEntity.setStatus(4);
       userEntity.setCreatedAt(ZonedDateTime.now());
       userEntity.setCreatedBy("api-backend");
       final UserEntity createdUserEntity = signupBusinessService.signup(userEntity);
       SignupUserResponse userResponse = new
SignupUserResponse().id(createdUserEntity.getUuid()).status("REGISTERED");
       return new ResponseEntity<SignupUserResponse>(userResponse,HttpStatus.CREATED);
   }
}
```

The above code receives the POST type request to register a user in the database. The method receives the SignupUserRequest type Java object (mapped to all fields in the JSON), declares an instance of UserEntity, sets all the attributes and pass this object to the business logic to be persisted in the database.

Swagger configuration

Since we are using Swagger to convert the JSON to Java, therefore we need to configure Swagger in the application.

Annotations used:

1. **@Configuration** annotation indicates that a class declares one or more @Bean methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime.
2. **@Bean** annotation is applied in a method to specify that it returns a bean to be managed by the Spring context.

The code for the SwaggerConfiguration class is shown below:

```
@Configuration
@EnableSwagger2
public class SwaggerConfiguration {

   @Bean
   public Docket swagger() {
       return new
Docket(DocumentationType.SWAGGER_2).select().apis(RequestHandlerSelectors.basePackage("com.upg
rad.proman.api.controller")).paths(PathSelectors.any()).build();
   }
}
```

Configuration class

This configuration class is used to load all the project components when you run the application. Annotations used:

1. **@Configuration** annotation indicates that the class has @Bean definition methods.
2. **@ComponentScan** annotation tells the Spring to search for the components/beans in all the classes under ‘com.upgrad.proman.service’ package. All the classes annotated with @Component will have their objects declared on running the application.
3. **@EntityScan** on the other hand does not declare the object. It only identifies the classes to be used by the persistence context.

The code for the Configuration class is shown below:

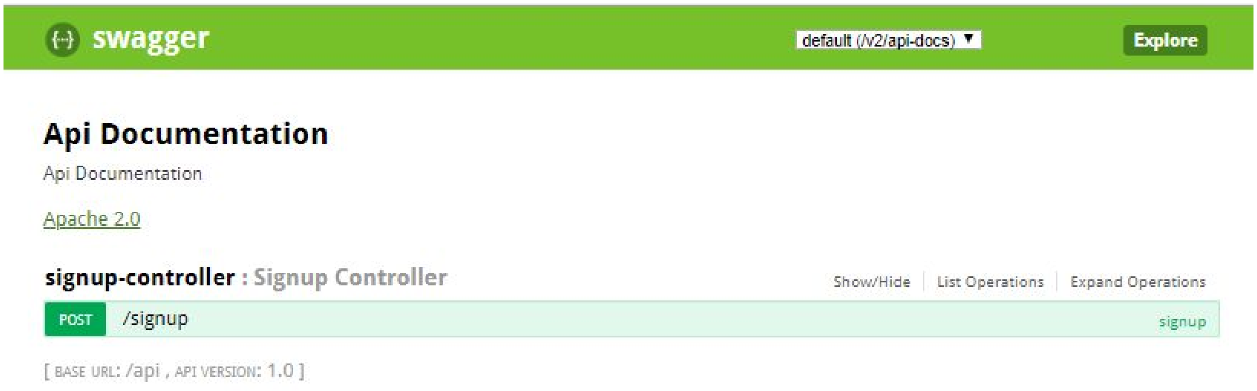
```
@Configuration
@ComponentScan("com.upgrad.proman.service")
@EntityScan("com.upgrad.proman.service.entity")
public class ServiceConfiguration {
}
```

Main class to run the application

An application needs a point where it can start from. The code for the main class for the application is shown below:

```
@SpringBootApplication
@Import(ServiceConfiguration.class)
public class PromanApiApplication {


   public static void main(String[] args){

       SpringApplication.run(PromanApiApplication.class,args);
   }
}
```

Now you are ready to run the application on UI. Run the application and you will observe the following output:



Api Documentation

Api Documentation

Apache 2.0

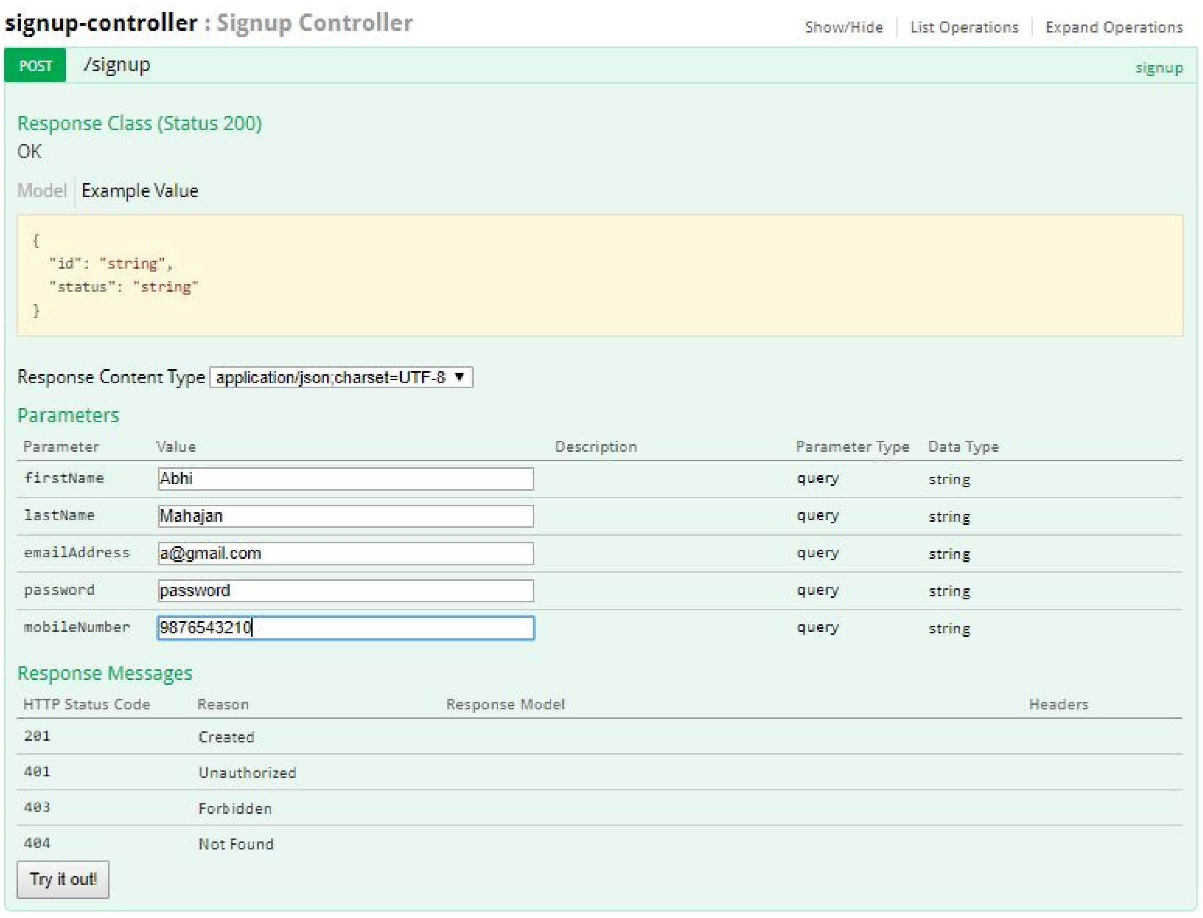
signup-controller : Signup Controller

POST /signup signup

[BASE URL: /api , API VERSION: 1.0]

Figure 4 - Endpoints on Swagger

You may enter the details of the user and register in the application using this endpoint as shown below:



signup-controller : Signup Controller

Show/Hide | List Operations | Expand Operations

POST /signup signup

Response Class (Status 200)

OK

Model | Example Value

```
{
  "id": "string",
  "status": "string"
}
```

Response Content Type application/json;charset=UTF-8

Parameters

Parameter	Value	Description	Parameter Type	Data Type
firstName	Abhi		query	string
lastName	Mahajan		query	string
emailAddress	a@gmail.com		query	string
password	password		query	string
mobileNumber	9876543210		query	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

Figure 5 - Registration form

Also, observe the data being stored in the PostgreSQL database.

Step III: Creating getUser endpoint

Controller class for getUser endpoint

Let us try to get the details of the registered user from the database.

- You need to enter the id of the user whose information is to be retrieved. Implement a new ‘users/{id}’ endpoint to get the details of the user.
- Observe that the request URL contains the dynamic parameter ‘id’ which specifies the id of the user whose information is to be retrieved and we use @PathVariable annotation to get the value of the dynamic parameter in the URL.

The endpoint definition for the userAdmin is provided by the product manager in the form of a JSON file ‘useradmin.json’. Build the project again and you will see the request and response models generated for this endpoint in the target folder.

The code for the Controller class is as shown below:

```
public class UserAdminController {

public ResponseEntity<UserDetailsResponse> getUser(final String userUuid) {

  }
}
```

Service class for get User endpoint

Create a new service class ‘UserBusinessService’. This class will have a getUser() method that will take a user Uuid as the input and return the corresponding UserEntity with the Uuid, if Uuid exists in the database.

The code for the service class is as shown below:

```
@Service
public class UserAdminBusinessService {

@Autowired
private UserDao userDao;

public UserEntity getUser(final String userUuid){
return userDao.getUser(userUuid);
   }
}
```

getUser() method calls a method in the user DAO class and receives an object of type UserEntity from the DAO. This method then returns the UserEntity type object received with the corresponding uuid to the controller class.

JPQL vs SQL

You need to write the SQL query to retrieve the data of a given user's Uuid in the DAO class, However, as we are implementing JPA, you also have an advantage of writing JPQL which is platform independent object oriented query language instead of writing the queries in plain SQL. **Unlike SQL, JPQL operate against Java entity objects rather than directly with database**. These queries are written in the Entity classes as named queries.

Let us write the named query in UserEntity class to fetch the user details.

```
@NamedQueries(
        {
                @NamedQuery(name = "userByUuid", query = "select u from UserEntity u where
u.uuid = :uuid")
        }
)
```

The query returns the user entity with uuid as 'uuid'.

DAO for get User endpoint

The code for DAO method which interacts with the database is shown below:

```
public UserEntity getUser(final String userUuid){
   return entityManager.createNamedQuery("userByUuid", UserEntity.class).setParameter("uuid",
userUuid)
           .getSingleResult();
}
```

- Define a getUser() method which takes the user uuid as an argument from service class.
- Calling the createNamedQuery() method for entityManager **creates** a query which is defined by using the 'javax.persistence.NamedQuery' annotation.
- Pass the name of the query as the first argument and the second argument is the **entity_class_name.class**.
- setParameter() method sets the value of the uuid parameter in the named query. getSingleResult() **executes** the select query named “userByUuid” returning a single result.
- After the query is executed, DAO returns the User details corresponding to the userUuid given.

Controller class for get User endpoint

The modified code for the Controller class is as shown below:

```
@RestController
@RequestMapping
public class UserAdminController {

   @Autowired
   private UserAdminBusinessService userAdminBusinessService;

   @RequestMapping(method = RequestMethod.GET, path = "/users/{id}", produces =
MediaType.APPLICATION_JSON_UTF8_VALUE)
   public ResponseEntity<UserDetailsResponse> getUser(@PathVariable("id") final String
userUuid) {
       final UserEntity userEntity = userAdminBusinessService.getUser(userUuid);
       UserDetailsResponse userDetailsResponse = new
UserDetailsResponse().id(userEntity.getUuid()).firstName(userEntity.getFirstName())
               .lastName(userEntity.getLastName()).emailAddress(userEntity.getEmail())

.mobileNumber(userEntity.getMobilePhone()).status(UserStatusType.valueOf(UserStatus.getEnum(us
erEntity.getStatus()).name()));
       return new ResponseEntity<UserDetailsResponse>(userDetailsResponse, HttpStatus.OK);
   }
}
```

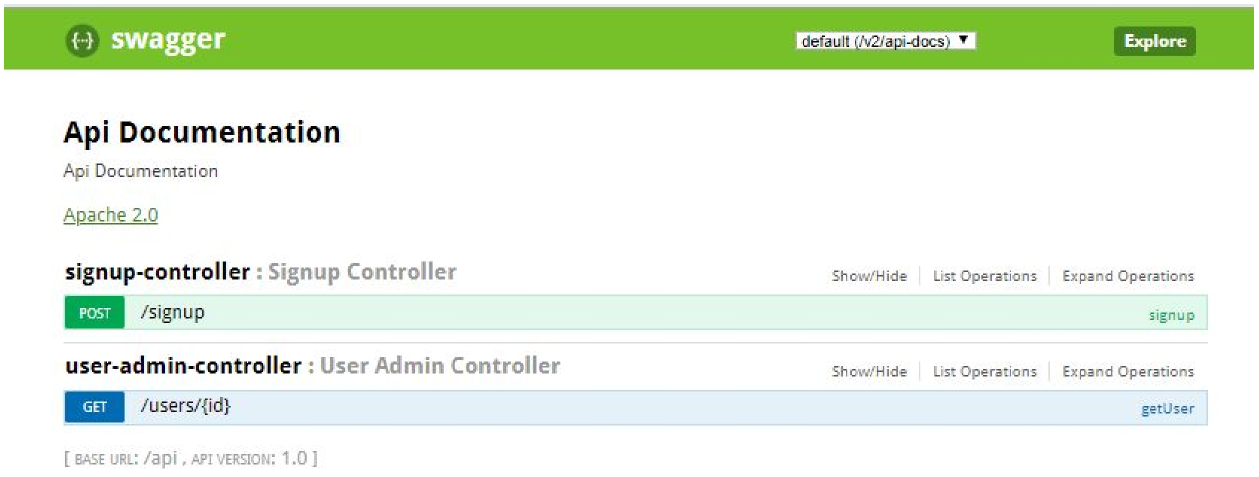
- This method listens for an Http request to GET type with “/users/{id}” pattern.
- The URI pattern contains dynamic parameter “id” and we need to map this dynamic parameter to one of the method argument, therefore @PathVariable annotation is used to map the value of the dynamic parameter “id”. @PathVariable(“id”) final String userUuid maps the value of ‘id’ dynamic parameter to the userUuid argument of the method.
- After the method gets the uuid of the user whose details are to be retrieved, it calls the business logic and gets all the details of the user with the corresponding uuid. The UserDetailsResponse type object is declared and all the attributes are set as received from the business logic.
- Then the response object is returned along with Http status OK, which specifies that the request was fulfilled.
- The status stored in the database when the user signs up is of integer type. When an admin gets the user details, we want the status to be readable and developer friendly. Therefore, we will define the enumeration constants for all four types of status.
- The value 0 is associated with ‘INACTIVE’, 1 is associated with ‘ACTIVE’, 2 is associated with ‘LOCKED’ and 4 is associated with ‘REGISTERED’.
- After we get the integer type status value from the database, we get the enumeration constant corresponding to the integer type status in the database. After we get the string, the status field of UserDetailsResponse is set after converting it to the UserStatusType type object using valueOf() method.

The code for the enum class is as shown below:

```
public enum UserStatus {

   ACTIVE(1) , INACTIVE(0), LOCKED(2),REGISTERED(4);

   private static final Map<Integer, UserStatus> Lookup = new HashMap<>();

   static{
       for(UserStatus userStatus : UserStatus.values()){
           Lookup.put(userStatus.getCode(), userStatus);
       }
   }
   private final int code;
   private  UserStatus(final int code){
       this.code = code;
   }

   public int getCode(){
       return code;
   }

   public static UserStatus getEnum(final int code){
       return Lookup.get(code);
   }
}
```

Add the enumeration class in **proman-service->src->main->java->com->upgrad->proman->service->type**

Now you are ready to run the application on UI. Run the application and you will observe the following output:



Api Documentation

Api Documentation

Apache 2.0

signup-controller : Signup Controller — Show/Hide | List Operations | Expand Operations

POST /signup signup

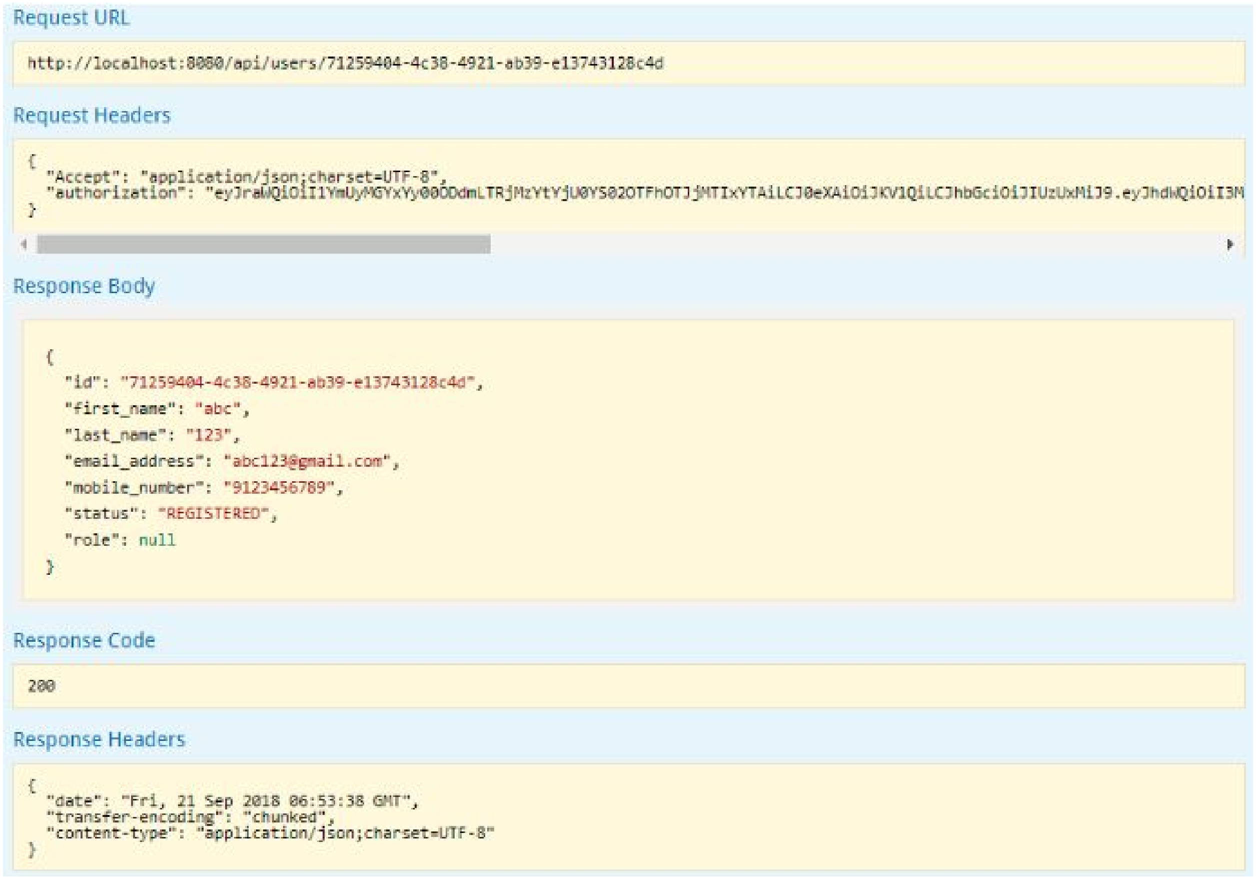
user-admin-controller : User Admin Controller — Show/Hide | List Operations | Expand Operations

GET /users/{id} getUser

[BASE URL: /api , API VERSION: 1.0]

Figure 6 - Endpoints on Swagger

Enter the id of the user whose details are to be fetched and you will get all the details of the user as shown below:



Request URL

```
http://localhost:8080/api/users/71259404-4c38-4921-ab39-e13743128c4d
```

Request Headers

```
{
  "Accept": "application/json;charset=UTF-8",
  "authorization": "eyJraWQiOiI1YmUyMGYxYy00ODdmLTRjMzYtYjU0YS02OTFhOTJjMTIxYTAiLCJ0eXAiOiJKV1QiLCJhbGciOiJIUzUxMiJ9.eyJhdWQiOiI3M
}
```

Response Body

```
{
  "id": "71259404-4c38-4921-ab39-e13743128c4d",
  "first_name": "abc",
  "last_name": "123",
  "email_address": "abc123@gmail.com",
  "mobile_number": "9123456789",
  "status": "REGISTERED",
  "role": null
}
```

Response Code

```
200
```

Response Headers

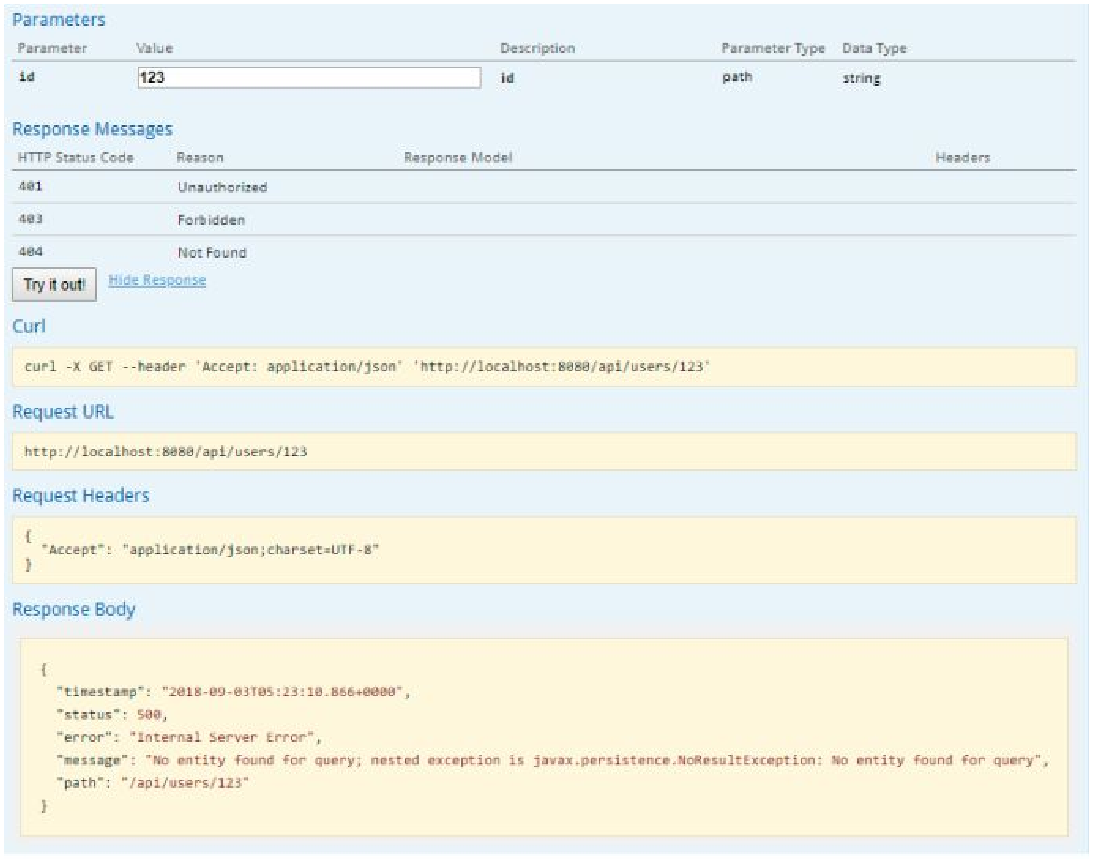
```
{
  "date": "Fri, 21 Sep 2018 06:53:38 GMT",
  "transfer-encoding": "chunked",
  "content-type": "application/json;charset=UTF-8"
}
```

Figure 7 - User details response

Step IV: Exception Handling

Need of Exception Handling

The registered user information is required by the admin of the application to validate the signup process. But imagine a case where the admin asks for the details of the user who has not yet registered i.e., the input (uuid) provided by the admin is not available in the database.
Observe the output given by the code when you enter the wrong uuid of the user whose details are to be fetched as shown below:



Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	123	id	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Curl

```
curl -X GET --header 'Accept: application/json' 'http://localhost:8080/api/users/123'
```

Request URL

```
http://localhost:8080/api/users/123
```

Request Headers

```
{
  "Accept": "application/json;charset=UTF-8"
}
```

Response Body

```
{
  "timestamp": "2018-09-03T05:23:10.866+0000",
  "status": 500,
  "error": "Internal Server Error",
  "message": "No entity found for query; nested exception is javax.persistence.NoResultException: No entity found for query",
  "path": "/api/users/123"
}
```

Figure 8 - Endpoints on Swagger

This is where exception handling comes into the picture in implementing your project. We need to display some user-friendly message indicating the failure to fetch the user details. Exception handling would allow us to return more specific errors with user-friendly message and code to the end developer or users, so they would have more information to identify the root issue and potentially fix the issue if possible.

Handling the Exception in getUser endpoint

How we will handle the exception using Exception handling?
You need to give end users a more specific error information through exception handling. Specifically, you have to handle the exception where the record for the entity is not found in the database and throw a user-friendly **error code and an error message** along with corresponding **Http status code**.

- The error code and error message are the part of the business logic and you have to include it in the service class of the endpoint.
- For which, you have to describe the case in DAO where the user does not exist by returning null (empty response).
- Then the business logic should implement exception handling by throwing an error code and error message for the case when no record is returned by the DAO.

Modified DAO class

The modified code to handle the exception for DAO class is as shown below:

```
public UserEntity getUser(final String userUuid){
   try {
       return entityManager.createNamedQuery("userByUuid",
UserEntity.class).setParameter("uuid", userUuid)
               .getSingleResult();
   }
   catch (NoResultException nre){
       return null;
   }
}
```

Note that the exception is handled using try-catch block. If the user record is found in the database, it is returned to the Service class in try block, otherwise the NoResultException is handled in the catch block.

Modified service class

Now we need to modify the service class such that it can handle the null value if DAO logic returns null.
The modified code for the business logic is as shown below:

```
public class UserAdminBusinessService {

   @Autowired
   private UserDao userDao;


   public UserEntity getUser(final String userUuid) throws ResourceNotFoundException {
       UserEntity userEntity =  userDao.getUser(userUuid);
       if(userEntity == null){
           throw new ResourceNotFoundException("USR-001", "User not found");
       }
       return userEntity;
   }
}
```

The code handles the case when DAO class returns null, and throws a user defined exception ‘ResourceNotFoundException’ along with suitable error code and error message. Now we also need to define user defined exception class and a handler to handle the user defined exception.

Defining user-defined exception and its handler

The code for ResourceNotFoundException is as shown below:

```
public class ResourceNotFoundException extends Exception {

   private final String code;
   private final String errorMessage;

   public ResourceNotFoundException(final String code, final String errorMessage){
       this.code = code;
       this.errorMessage = errorMessage;
   }

   @Override
   public void printStackTrace() {
       super.printStackTrace();
   }

   @Override
   public void printStackTrace(PrintStream s) {
       super.printStackTrace(s);
   }

   @Override
   public void printStackTrace(PrintWriter s) {
       super.printStackTrace(s);
   }

   public String (){
       return code;
   }

   public String getErrorMessage(){
       return errorMessage;
   }
}
```

The above class defines a custom exception. The structure of the code is very simple as:

- It consists of a constructor for the class where the error code and error message are passed as the arguments.
- There are two getter methods for the error code and error response.
- And a few methods are overridden from the Exception superclass, which the ResourceNotFoundException class extended from.

Now you have defined an exception but there is no method which decides the action to be taken when that exception is encountered. Let us now define a suitable handler for this user defined exception.
Annotations used:

1. **@ControllerAdvice** is a Spring specific annotation that allows you to write a piece of code which will be made available to all classes annotated with @Controller annotation.

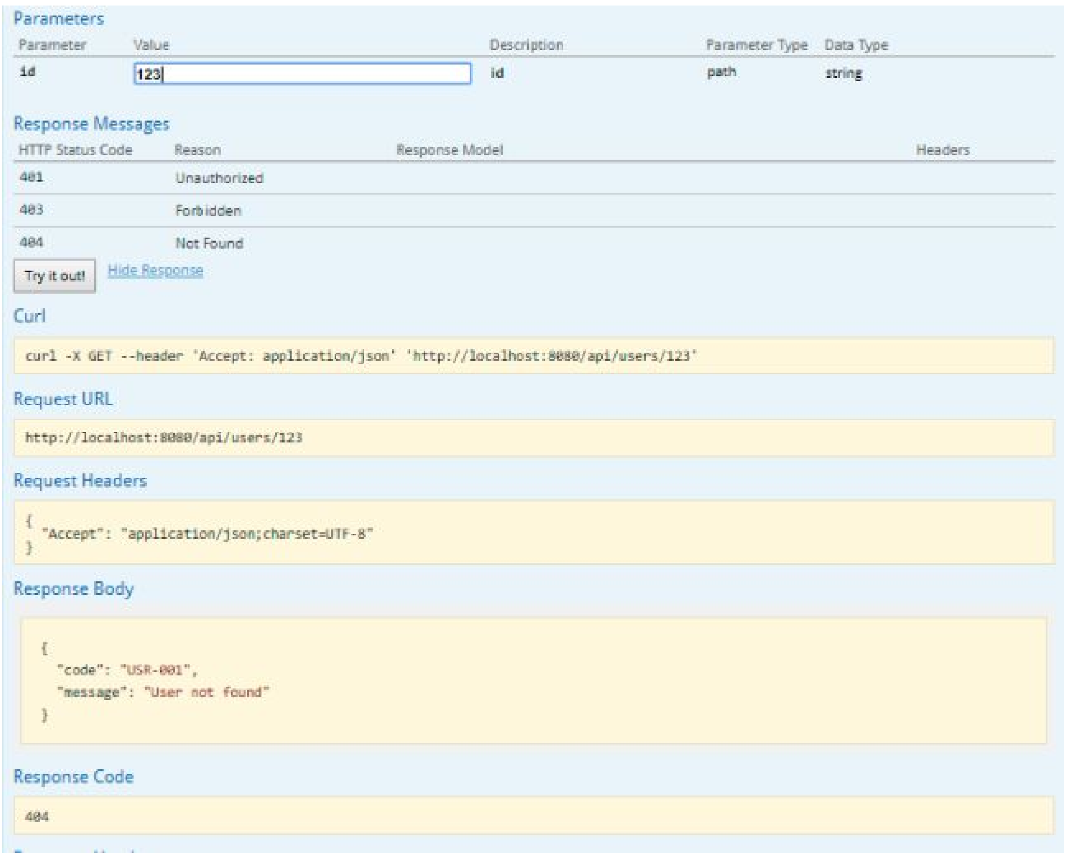
2. **@ExceptionHandler** annotation allows you to define a method to handle specific exceptions.

The code for the handler is as shown below:

```
@ControllerAdvice
public class RestExceptionHandler {

   @ExceptionHandler(ResourceNotFoundException.class)
   public ResponseEntity<ErrorResponse> resourceNotFoundException(ResourceNotFoundException
exe, WebRequest request){
       return new ResponseEntity<ErrorResponse>(
               new ErrorResponse().code(exe.getCode()).message(exe.getErrorMessage()),
HttpStatus.NOT_FOUND
       );
   }
}
```

The exception handling method above takes the two arguments: the exception object and a web request object. An exception object is an object of that exception class for which this exception handler is defined. The web request object gives access to all the request parameters.

Now try to run the application and fetch the user details for a user which does not exist in the database. You will observe the following output:



Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	123	id	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Curl

```
curl -X GET --header 'Accept: application/json' 'http://localhost:8080/api/users/123'
```

Request URL

```
http://localhost:8080/api/users/123
```

Request Headers

```
{
  "Accept": "application/json;charset=UTF-8"
}
```

Response Body

```
{
  "code": "USR-001",
  "message": "User not found"
}
```

Response Code

```
404
```

Figure 9 - Get details of non existing user

Step V: Password encryption

Need of password encryption

The details, including the user’s password, entered by the user in signup endpoint are directly stored in the database. You need to store the password in the database in an encrypted manner to add an additional layer of security as the user’s password is sensitive information. More specifically, in the event that a hacker gains unauthorized access to your database, password encryption helps to protect the database from revealing the actual user passwords to the hacker as the hacker can only see the encrypted version of the passwords. Hence, it is an important task to save the password in an encrypted form in the database.

Encrypting the password

Add the ‘PasswordCryptographyProvider’ class to the business package in proman service submodule which provides suitable methods to encrypt and decrypt the password.

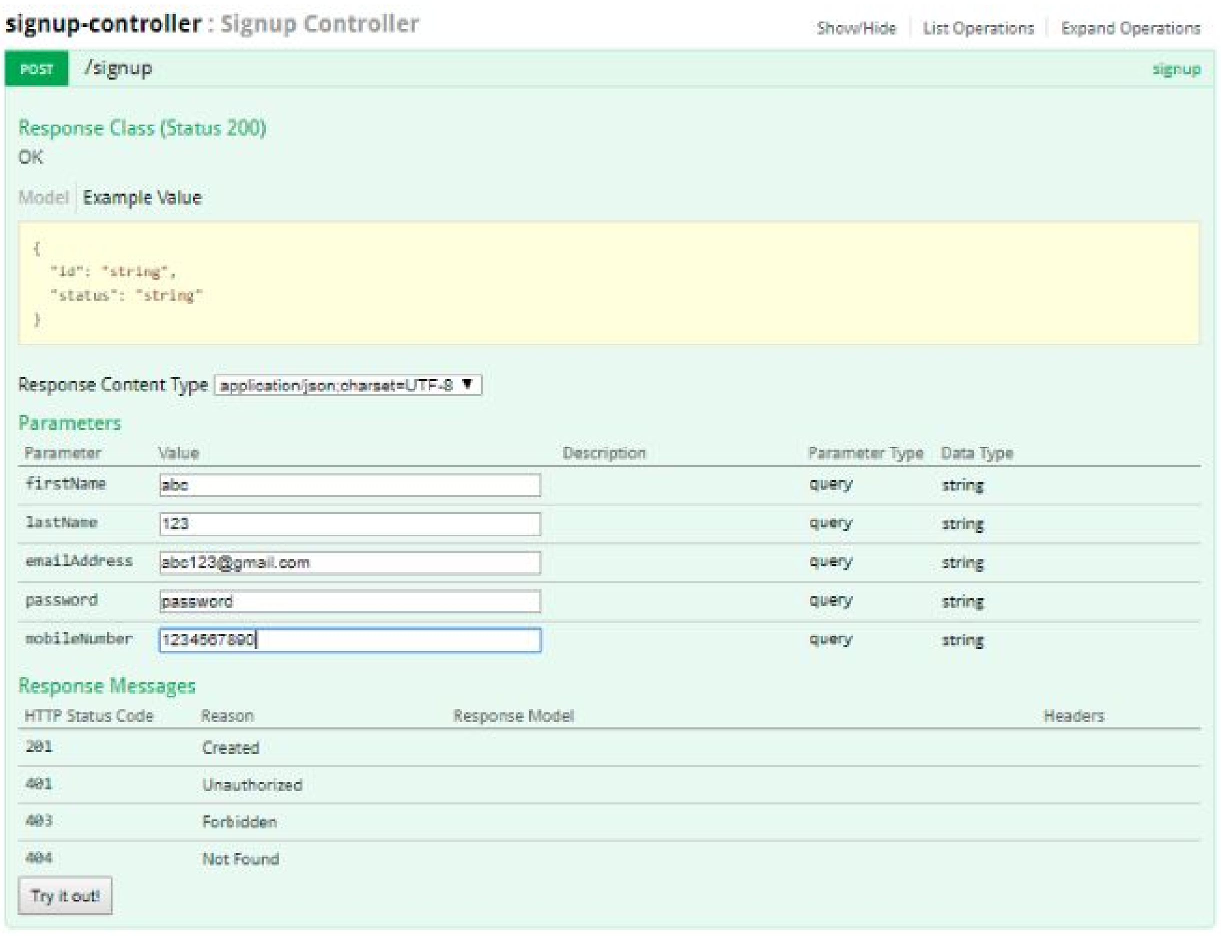
Now you need to add a business logic in the Service class to encrypt the password using the ‘PasswordCryptographyProvider’ class.
The code for the Service class is as shown below:

```java
@Service
public class SignupBusinessService {

   @Autowired
   private UserDao userDao;

   @Autowired
   private PasswordCryptographyProvider cryptographyProvider;

   @Transactional(propagation = Propagation.REQUIRED)
   public UserEntity signup(UserEntity userEntity)
   {
       String[] encryptedText = cryptographyProvider.encrypt(userEntity.getPassword());
       userEntity.setSalt(encryptedText[0]);
       userEntity.setPassword(encryptedText[1]);

       return userDao.createUser(userEntity);
   }
}
```

The above code encrypts the raw password and generates the salt which are then persisted in the database. The encrypt(final String password) method is used when you sign up in the application. The method takes the string i.e, the password entered by the user in plain text format as an argument, encrypts the password and returns an array of two strings. The first string is the salt and the second string is the encrypted password.

What is salt?
The salt adds an extra layer of cryptographic security and makes it more difficult for a hacker to decrypt the encrypted password if a hacker does indeed gain unauthorized access to your database. Therefore, when a user signs in the application, the user provides the username and password in plain text format. The application takes salt from the database corresponding to the same username and then encrypts the password provided by the user using this salt by calling encrypt (final String password, String salt) method.

Try to register in the application as shown below:


signup-controller : Signup Controller

/signup

Response Class (Status 200)

OK

```
{
  "id": "string",
  "status": "string"
}
```

Response Content Type: application/json;charset=UTF-8

Parameters

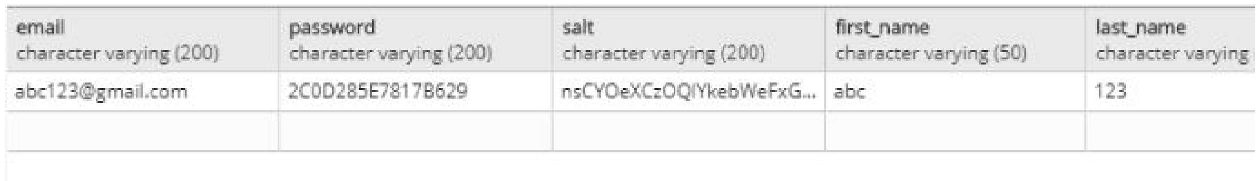
Parameter	Value	Description	Parameter Type	Data Type
firstName	abc		query	string
lastName	123		query	string
emailAddress	abc123@gmail.com		query	string
password	password		query	string
mobileNumber	1234567890		query	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

Figure 10 - Registration form

Observe the record stored in the database. Observe the password in encrypted format and also the salt as shown below:



email character varying (200)	password character varying (200)	salt character varying (200)	first_name character varying (50)	last_name character varying
abc123@gmail.com	2C0D285E7817B629	nsCYOeXCzOQIYkebWeFxG...	abc	123

Figure 11 - PostgreSQL

Step VI: Authentication

What is Authentication?

In an application, you may want to restrict the accessibility of certain endpoints or functionality to registered users. Therefore, the application will have to validate these users on the basis of some specific identity such as username and password. More specifically, the application has to check the login details provided i.e., username and password against the information stored in the database. This process of validating user credentials is called as authentication.

How the authorization header is entered?
The username (email) and password are concatenated with a colon. For example, a username of 'abhi' and a password of 'password' becomes the string 'abhi:password' and then this string is encoded to Base64 format to '**YWJoaTpwYXNzd29yZA==**' and then '**Basic YWJoaTpwYXNzd29yZA==**' is entered in the authorization header.

Why username and password are not passed simply rather in the authorization header?
This format was defined for HTTP basic authentication by the "Internet Engineering Task Force" specifically. HTTP basic authentication states that:

1. The username and password should be transmitted in the format of a String such as "username:password"
2. The string should be base64 encoded, not for security, but to encode non-HTTP-compatible characters into HTTP-compatible characters that may be in the username or password.
3. And the encoded String should be stored under the authorization field in the HTTP header under the format:

Authorization: Basic [insert base64 encoded string here]

Authentication scenario

- Firstly you need to generate the response model using the 'authentication.json' file. After the build is successful you can see the model class for AuthorisedUserResponse generated in the target folder. This model class contains the information about the user like first name, last name, last login time, role etc.
- Add the required plugins in the pom file of the proman-api module to generate the response model of the login endpoint.
- Now create the skeleton (method signature and the return type) of the controller class, service class and add a method in the DAO for authentication endpoint.

Let's discuss the possible scenarios:

- If the username (email) entered by the user is not available in the database, the named query will not be able to retrieve any result and DAO will return null to the service class
- If the username (email) is correct, the DAO will return the UserEntity object corresponding to the username(email) entered. Here you have to encrypt the password with the correct salt and compare the encrypted password with the encrypted password stored in the database.
- If these credentials match, you should return the UserAuthEntity containing JWT token and other necessary information from the Service class. (implemented in further segments).
- However, If the password does not match, you have to throw a user-friendly error code and error message to the user and handle the exceptions in the service and the controller class.

Controller class for authentication

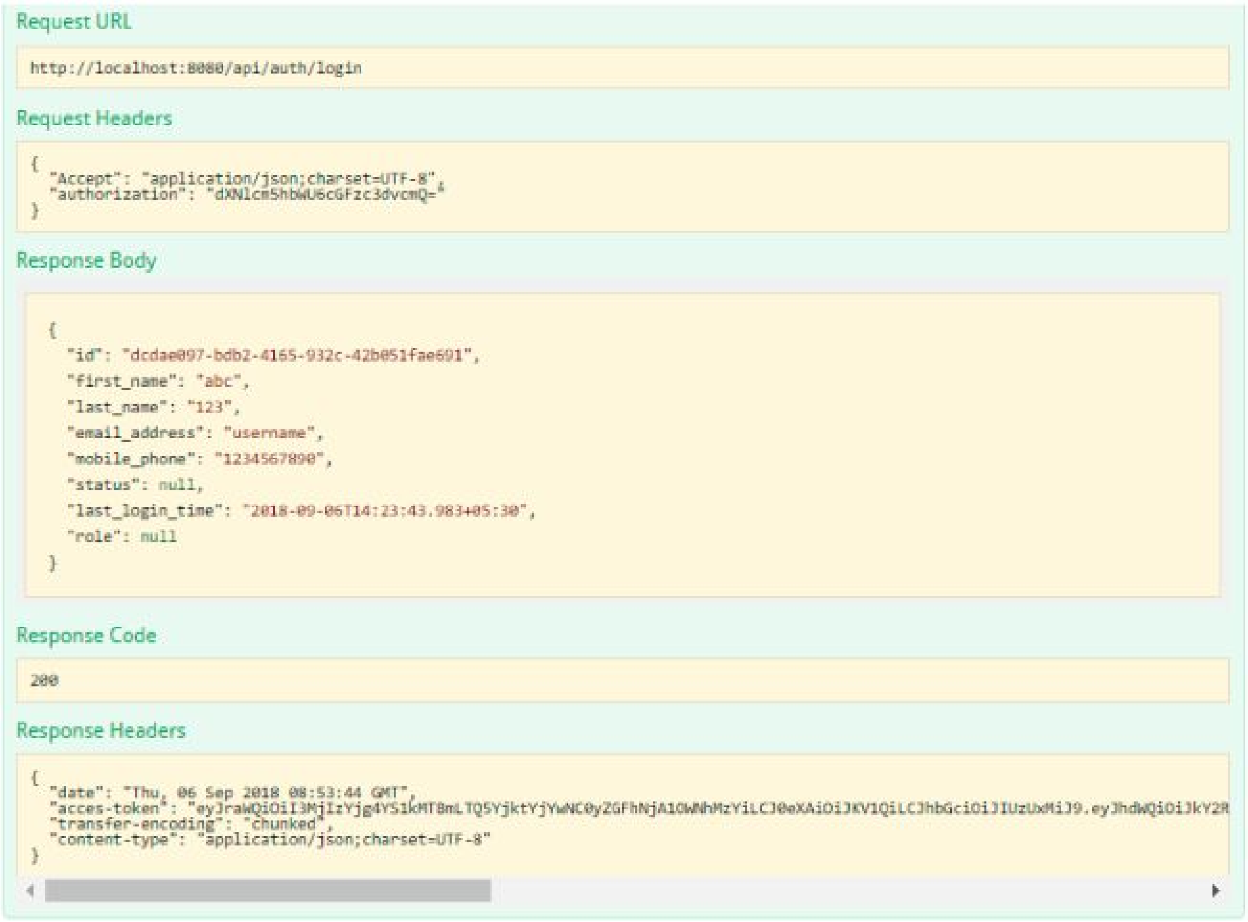
Let us start with the controller logic.
The code for the controller class is as shown below:

```
@RestController
@RequestMapping("/")
public class AuthenticationController {


   @Autowired
   private AuthenticationService authenticationService;

   @RequestMapping(method = RequestMethod.POST, path = "auth/login", produces =
MediaType.APPLICATION_JSON_UTF8_VALUE)
   public ResponseEntity<AuthorizedUserResponse> login(@RequestHeader("authorization") final
String authorization) throws AuthenticationFailedException {
       //Basic dXNlcm5hbWU6cGFzc3dvcmQ=
       //above is a sample encoded text where the username is "username" and password is
"password" seperated by a ":"
       byte[] decode = Base64.getDecoder().decode(authorization.split("Basic ")[1]);
       String decodedText = new String(decode);
       String[] decodedArray = decodedText.split(":");

       UserAuthTokenEntity userAuthToken =
authenticationService.authenticate(decodedArray[0],decodedArray[1]);
       UserEntity user = userAuthToken.getUser();

       AuthorizedUserResponse authorizedUserResponse =  new
AuthorizedUserResponse().id(UUID.fromString(user.getUuid()))
               .firstName(user.getFirstName()).lastName(user.getLastName())
               .emailAddress(user.getEmail()).mobilePhone(user.getMobilePhone())
               .lastLoginTime(user.getLastLoginAt());

       HttpHeaders headers = new HttpHeaders();
       headers.add("access-token", userAuthToken.getAccessToken());
       return new ResponseEntity<AuthorizedUserResponse>(authorizedUserResponse,headers,
HttpStatus.OK);
   }
}
```

Observe the argument passed in method signature for login. The argument passed here is a request header (string) called ‘authorization’. As already discussed, you need to pass the Basic authorization token in the above request header.

- First, we remove “Basic” from the authorization string
- The decode() method decodes the Base64 the rest of the encoded authorization string
- The decode() method, which we called to decode the authorization String, returns its results as a byte[]. We would then convert the array of bytes to a string by creating a String and passing in the byte array as the parameter into the String constructors. This decoded String is stored in the variable ‘decodedText’.
- Then, the split() method is used to split the decoded string with the colon (:) into two separate strings.
- First of which is the username and the second element is the password

- Now that you have got the username and password, you may pass this information to the ‘authenticate’ method in the service class.
- After the business logic successfully works, the user is signed in the application and the user details along with JWT token are returned in the response as shown below:



Request URL

```
http://localhost:8080/api/auth/login
```

Request Headers

```
{
  "Accept": "application/json;charset=UTF-8",
  "authorization": "dXNlcm5hbWU6cGFzc3dvcmQ="
}
```

Response Body

```
{
  "id": "dcdae097-bdb2-4165-932c-42b051fae691",
  "first_name": "abc",
  "last_name": "123",
  "email_address": "username",
  "mobile_phone": "1234567890",
  "status": null,
  "last_login_time": "2018-09-06T14:23:43.983+05:30",
  "role": null
}
```

Response Code

```
200
```

Response Headers

```
{
  "date": "Thu, 06 Sep 2018 08:53:44 GMT",
  "acces-token": "eyJraWQiOiI3MjIzYjg4YS1kMTBmLTQ5YjktYjYwNC0yZGFhNjA1OWNhMzYiLCJ0eXAiOiJKV1QiLCJhbGciOiJIUzUxMiJ9.eyJhdWQiOiJkY2R
  "transfer-encoding": "chunked",
  "content-type": "application/json;charset=UTF-8"
}
```

Figure 12 - Authentication response

Service class for authentication

The code for the Service class is as shown below:

```
@Service
public class AuthenticationService {

   @Autowired
   private UserDao userDao;

   @Autowired
   private PasswordCryptographyProvider cryptographyProvider;


   @Transactional(propagation = Propagation.REQUIRED)
   public UserAuthTokenEntity authenticate(final String username, final String password)
throws AuthenticationFailedException {
       UserEntity userEntity = userDao.getUserByEmail(username);
       if (userEntity == null) {
           throw new AuthenticationFailedException("ATH-001", "User with email not found");
       }

       final String encryptedPassword = cryptographyProvider.encrypt(password,
userEntity.getSalt());
       if (encryptedPassword.equals(userEntity.getPassword())) {
           JwtTokenProvider jwtTokenProvider = new JwtTokenProvider(encryptedPassword);
           UserAuthTokenEntity userAuthToken = new UserAuthTokenEntity();
           userAuthToken.setUser(userEntity);
           final ZonedDateTime now = ZonedDateTime.now();
           final ZonedDateTime expiresAt = now.plusHours(8);
           userAuthToken.setAccessToken(jwtTokenProvider.generateToken(userEntity.getUuid(),
now, expiresAt));
           userAuthToken.setLoginAt(now);
           userAuthToken.setExpiresAt(expiresAt);
           userAuthToken.setCreatedBy("api-backend");
           userAuthToken.setCreatedAt(now);

           userDao.createAuthToken(userAuthToken);
           userDao.updateUser(userEntity);
           userEntity.setLastLoginAt(now);

           return userAuthToken;
       } else {
           throw new AuthenticationFailedException("ATH-002", "Password Failed");
       }

   }
}
```

- The authenticate() method in the service class takes username and password as the arguments. As you saw in the controller class, the request method takes string argument ‘authorization’. So, the controller has to decode this encoded ‘authorization’ string to the separate username and password that it represents. And then pass the username and password to the service class.
- The code then checks whether the user with entered email exists in the database or not? If the user does not exist in the database, throw ‘AuthenticationFailedException’.

- If user exists in the database, the service class should now encrypt the password with the appropriate salt and compare the encrypted password against the encrypted password stored in the database.
- If the password does not match, the service class should throw ‘AuthenticationFailedException’.
- If the password matches, service class generates the JWT token, sets all the attributes of the ‘UserAuthTokenEntity’ and calls the DAO logic to persist the user authentication details in the database.
- It also updates the Last logout time of the user in the ‘UserEntity’ using the merge() method.

DAO class for authentication

When a user enters a username (or user email in our case) the DAO should search for the UserEntity object in the database corresponding to the given username. To do this, you will have to write a JPQL query in the DAO to search for and return an UserEntity object with the given username (or user email in our case).
The JPQL query is as shown below:

```
@Entity
@Table(name = "USERS", schema = "proman")
@NamedQueries(
        {
                @NamedQuery(name = "userByUuid", query = "select u from UserEntity u where
u.uuid = :uuid"),
                @NamedQuery(name = "userByEmail", query = "select u from UserEntity u where
u.email = :email")
        }
)
```

Now you need to implement the DAO logic to interact with the database. The DAO class is as shown below:

```
public UserEntity getUserByEmail(final String email) {
    try {
        return entityManager.createNamedQuery("userByEmail",
UserEntity.class).setParameter("email", email)
                .getSingleResult();
    } catch (NoResultException nre) {
        return null;
    }
}

public UserAuthTokenEntity createAuthToken(final UserAuthTokenEntity userAuthTokenEntity) {
    entityManager.persist(userAuthTokenEntity);
    return userAuthTokenEntity;
}

public void updateUser(final UserEntity updatedUserEntity) {
    entityManager.merge(updatedUserEntity);
}
```

- After the first method executes the named query, it will retrieve the user with corresponding email from the database.
- The second method persists the 'UserAuthTokenEntity' in the database after successful login.
- The third method updates the last login time in the 'UserEntity' using merge() method.

Exception and its handler

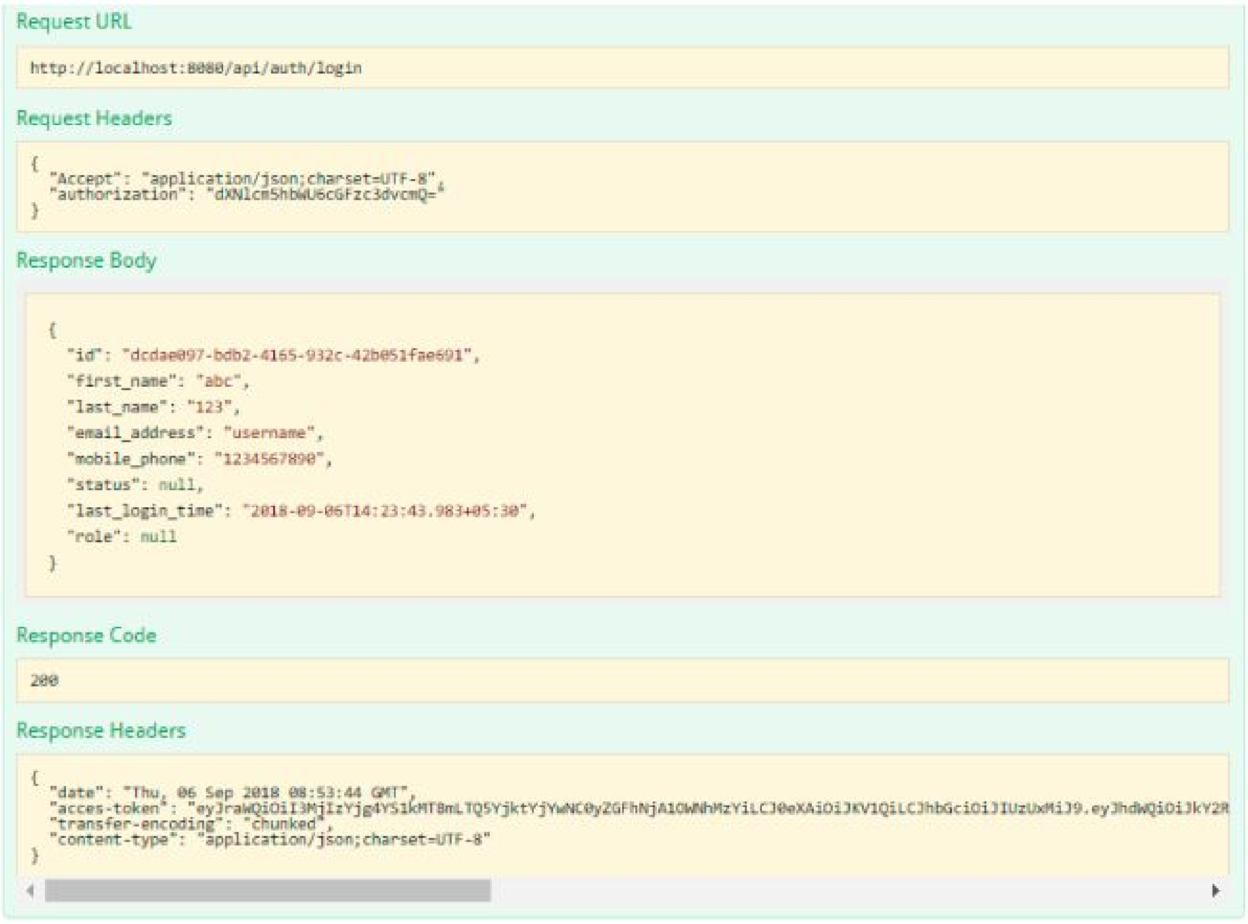
In the case when the username entered by the user is invalid i.e. no user record is found with the given username, the getUserByEmail(username) method will return a null object to the userEntity variable. Therefore, to follow how we created the custom exception for the getUser endpoint, let's first create a new user-defined exception named **'AuthenticationFailedException'**. The code for the exception is as follows:

```
public class AuthenticationFailedException extends Exception {

   private final String code;
   private final String errorMessage;

   public AuthenticationFailedException(final String code, final String errorMessage){
       this.code = code;
       this.errorMessage = errorMessage;
   }

   @Override
   public void printStackTrace() {
       super.printStackTrace();
   }

   @Override
   public void printStackTrace(PrintStream s) {
       super.printStackTrace(s);
   }

   @Override
   public void printStackTrace(PrintWriter s) {
       super.printStackTrace(s);
   }

   public String getCode(){
       return code;
   }

   public String getErrorMessage(){
       return errorMessage;
   }
}
```

You also need to define a handler for this exception. The code for exception handler is as follows:

```
@ExceptionHandler(AuthenticationFailedException.class)
public ResponseEntity<ErrorResponse>
authenticationFailedException(AuthenticationFailedException exe, WebRequest request){
   return new ResponseEntity<ErrorResponse>(
           new ErrorResponse().code(exe.getCode()).message(exe.getErrorMessage()),
HttpStatus.NOT_FOUND
   );
}
```

Now you can enter the Basic authorization header in the authorization request header and observe the output as shown below:



Request URL

```
http://localhost:8080/api/auth/login
```

Request Headers

```
{
  "Accept": "application/json;charset=UTF-8",
  "authorization": "dXNlcm5hbWU6cGFzc3dvcmQ="
}
```

Response Body

```
{
  "id": "dcdae097-bdb2-4165-932c-42b051fae691",
  "first_name": "abc",
  "last_name": "123",
  "email_address": "username",
  "mobile_phone": "1234567890",
  "status": null,
  "last_login_time": "2018-09-06T14:23:43.983+05:30",
  "role": null
}
```

Response Code

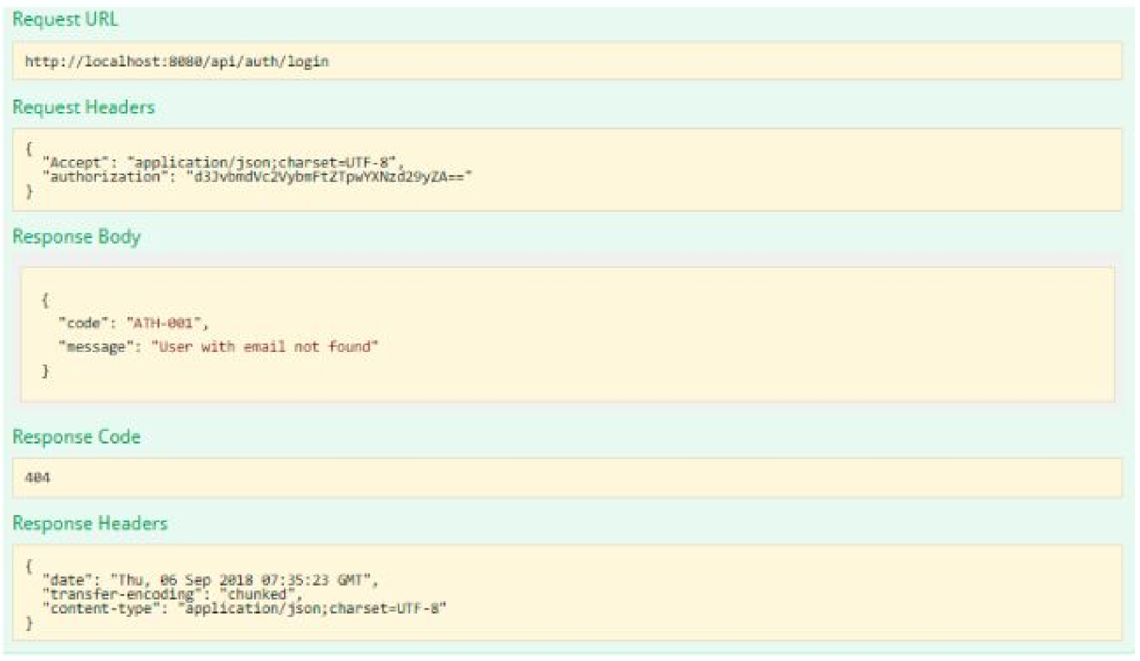
```
200
```

Response Headers

```
{
  "date": "Thu, 06 Sep 2018 08:53:44 GMT",
  "acces-token": "eyJraWQiOiI3MjIzYjg4YS1kMTBmLTQ5YjktYjYwNC0yZGFhNjA1OWNhMzYiLCJ0eXAiOiJKV1QiLCJhbGciOiJIUzUxMiJ9.eyJhdWQiOiJkY2R
  "transfer-encoding": "chunked",
  "content-type": "application/json;charset=UTF-8"
}
```

Figure 13 - Authentication response

Try to enter the invalid username and you will observe the following output:



Request URL

```
http://localhost:8080/api/auth/login
```

Request Headers

```
{
  "Accept": "application/json;charset=UTF-8",
  "authorization": "d3JvbmdVc2VybmFtZTpwYXNzd29yZA=="
}
```

Response Body

```
{
  "code": "ATH-001",
  "message": "User with email not found"
}
```

Response Code

```
404
```

Response Headers

```
{
  "date": "Thu, 06 Sep 2018 07:35:23 GMT",
  "transfer-encoding": "chunked",
  "content-type": "application/json;charset=UTF-8"
}
```

Figure 14 - Invalid credentials while authentication

Step VII: Create User endpoint

Controller class for create user endpoint

We will now implement the ‘createUser’ endpoint. This endpoint functionality is provided by the product manager as part of the ‘userAdmin.json’ file and it helps to generate the request and response models.
The code for controller class is as shown below:

```
@RequestMapping(method = RequestMethod.POST, path = "/users", consumes =
MediaType.APPLICATION_JSON_UTF8_VALUE, produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
public ResponseEntity<CreateUserResponse> createUser(final CreateUserRequest userRequest){
   UserEntity userEntity = new UserEntity();
   userEntity.setUuid(UUID.randomUUID().toString());
   userEntity.setFirstName(userRequest.getFirstName());
   userEntity.setLastName(userRequest.getLastName());
   userEntity.setEmail(userRequest.getEmailAddress());
   userEntity.setMobilePhone(userRequest.getMobileNumber());
   userEntity.setStatus(UserStatus.ACTIVE.getCode());
   userEntity.setCreatedAt(ZonedDateTime.now());
   userEntity.setCreatedBy("api-backend");

   final UserEntity createdUser = userAdminBusinessService.createUser(userEntity);

   final CreateUserResponse userResponse = new
CreateUserResponse().id(createdUser.getUuid()).status(UserStatusType.ACTIVE);

   return new ResponseEntity<CreateUserResponse>(userResponse, HttpStatus.CREATED);

}
```

The createUser and signup controller method implementation remain same except for the user status of the newly created user. When the administrator creates a new user through the create user endpoint, the user status will be set as ‘Active’, whereas when the user registers using signup endpoint, the user status of the newly created user will be set as ‘Registered’.

- The above code receives the request model, declares an object of UserEntity type to be persisted in the database, sets all the attributes and calls the business logic to persist this record in the database.
- The code then returns the details of the user persisted in the database in the response body.

Service class for create user endpoint

The code for the service class is as follows:

```
@Transactional(propagation = Propagation.REQUIRED)
public UserEntity createUser(final UserEntity userEntity){

   String password = userEntity.getPassword();
   if(password == null){
       userEntity.setPassword("proman@123");
   }
   String[] encryptedText = cryptographyProvider.encrypt(userEntity.getPassword());
   userEntity.setSalt(encryptedText[0]);
   userEntity.setPassword(encryptedText[1]);
   return userDao.createUser(userEntity);

}
```

- The above code first sets the default password for all the users created by the administrator as ‘proman@123’.
- Encrypts the password and generates the salt.
- Calls the DAO logic to persist the record in the database.

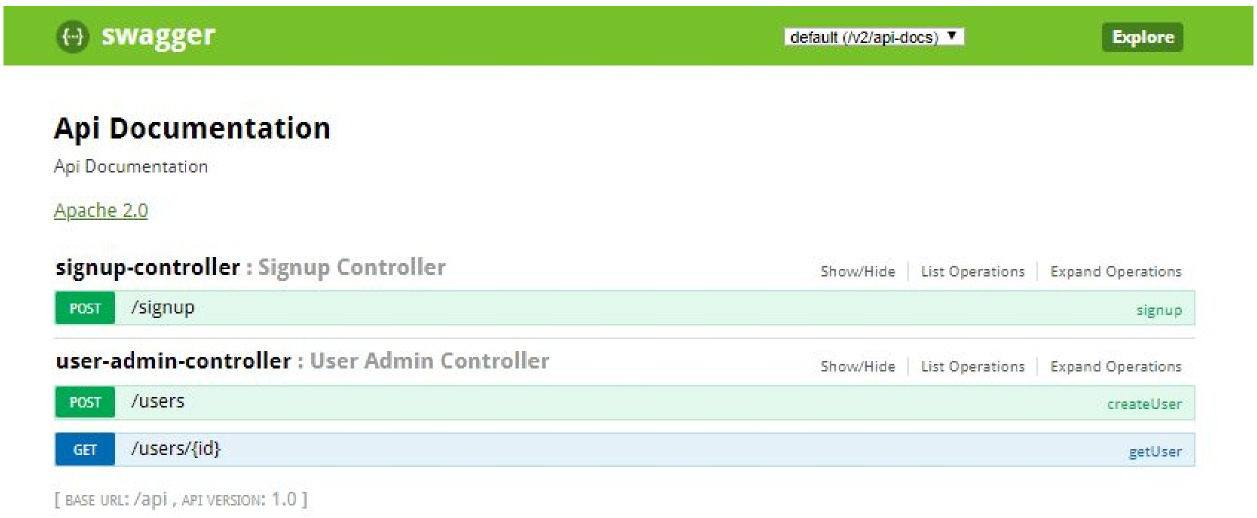
DAO logic for create user endpoint

The DAO class is as shown below:

```
public UserEntity createUser(UserEntity userEntity) {
   entityManager.persist(userEntity);
   return userEntity;
}
```

The above code simply persists the user record in the database.

Run the application and observe the endpoint as shown below:


swagger
default (/v2/api-docs)
Explore

Api Documentation

Api Documentation

Apache 2.0

signup-controller : Signup Controller

Show/Hide | List Operations | Expand Operations

POST /signup signup

user-admin-controller : User Admin Controller

Show/Hide | List Operations | Expand Operations

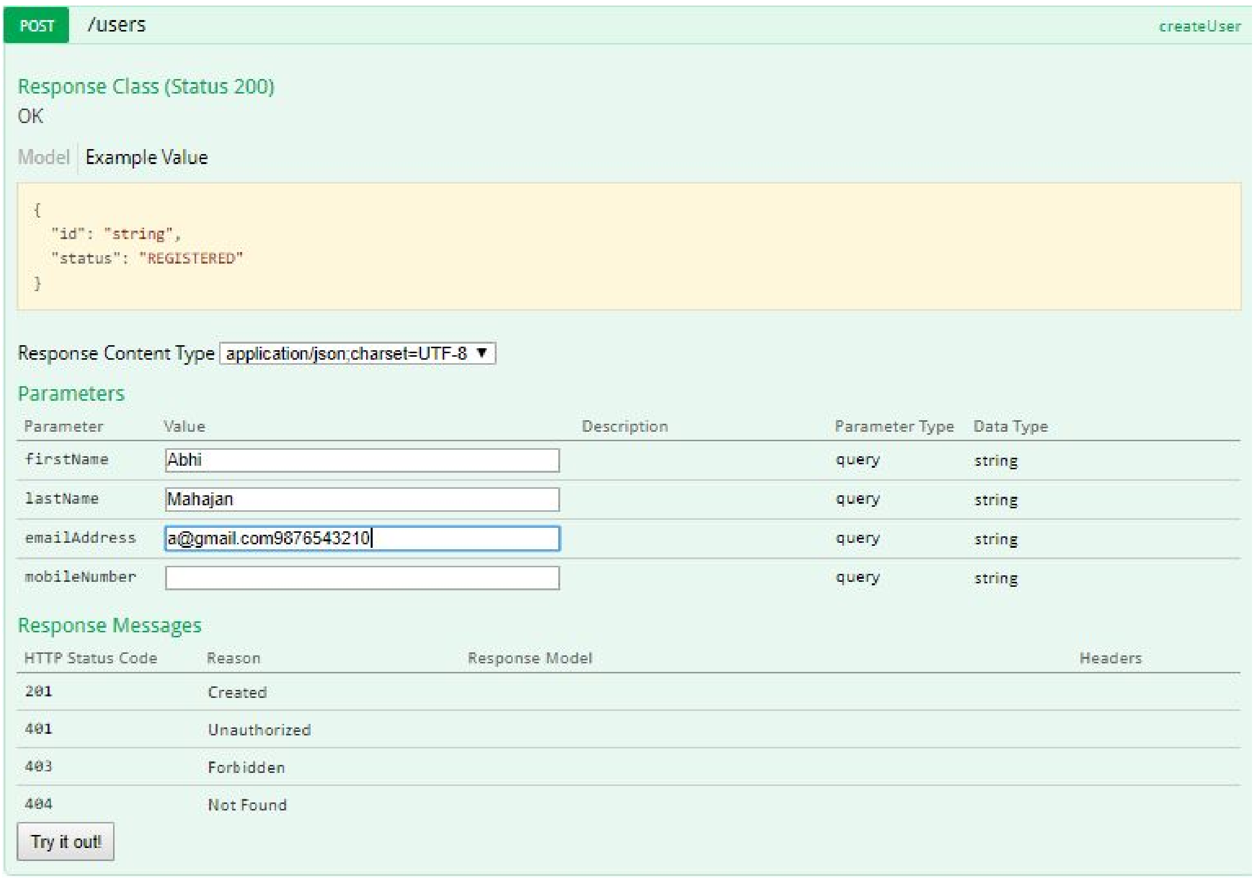
POST /users createUser

GET /users/{id} getUser

[BASE URL: /api , API VERSION: 1.0]

Figure 15 - Endpoints on Swagger

Fill the details and the record will be saved in the database as shown below:



POST /users createUser

Response Class (Status 200)
OK

Model | Example Value

```
{
  "id": "string",
  "status": "REGISTERED"
}
```

Response Content Type application/json;charset=UTF-8

Parameters

Parameter	Value	Description	Parameter Type	Data Type
firstName	Abhi		query	string
lastName	Mahajan		query	string
emailAddress	a@gmail.com9876543210		query	string
mobileNumber			query	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

Figure 16 - Create a user

Step VIII: Authorization

Need and mechanism

You have implemented the createUser endpoint. Now it's time to learn to implement authorization to a get user endpoint such that only the authenticated user with the role as 'administrator' can access that endpoint. The authorization of a user is validated using the JWT tokens generated when the user logs into the application.

In order to add authorization to the getUser endpoint, you need to check and validate the JWT token of the authenticated user:
First, you need to retrieve the JWT token from the 'authorization' field in request header and the access token is passed to the service class from the controller to fetch the user details (role).

- If the role of the user with the access token is 'administrator', then the getUser endpoint business logic is executed.
- Else, the service must throw an unauthorized user exception to indicate the user does not have the permission or authorization to accomplish the requested action in the application.

Controller class for authorized get user endpoint

Annotations used:

1. **@RequestHeader** - This annotation maps a controller method parameter to a specific request header value as indicated in the annotation. Here in the code, the value of the "authorization" field in the request header is mapped to the "authorization" parameter of the getUser() method.

The code for the controller class is as shown below:

```
@RequestMapping(method = RequestMethod.GET, path = "/users/{id}", produces =
MediaType.APPLICATION_JSON_UTF8_VALUE)
public ResponseEntity<UserDetailsResponse> getUser(@PathVariable("id") final String userUuid,
                                                    @RequestHeader("authorization") final String
authorization) throws ResourceNotFoundException, UnauthorizedException {
   final UserEntity userEntity = userAdminBusinessService.getUser(userUuid, authorization);
   UserDetailsResponse userDetailsResponse = new
UserDetailsResponse().id(userEntity.getUuid()).firstName(userEntity.getFirstName())
           .lastName(userEntity.getLastName()).emailAddress(userEntity.getEmail())
           .mobileNumber(userEntity.getMobilePhone())
           .status(UserStatusType.valueOf(UserStatus.getEnum(userEntity.getStatus()).name()));
   return new ResponseEntity<UserDetailsResponse>(userDetailsResponse, HttpStatus.OK);
}
```

The controller class passes the authorization token to the service class and the authorization part is implemented by the service class.

Service class for authorized get user endpoint

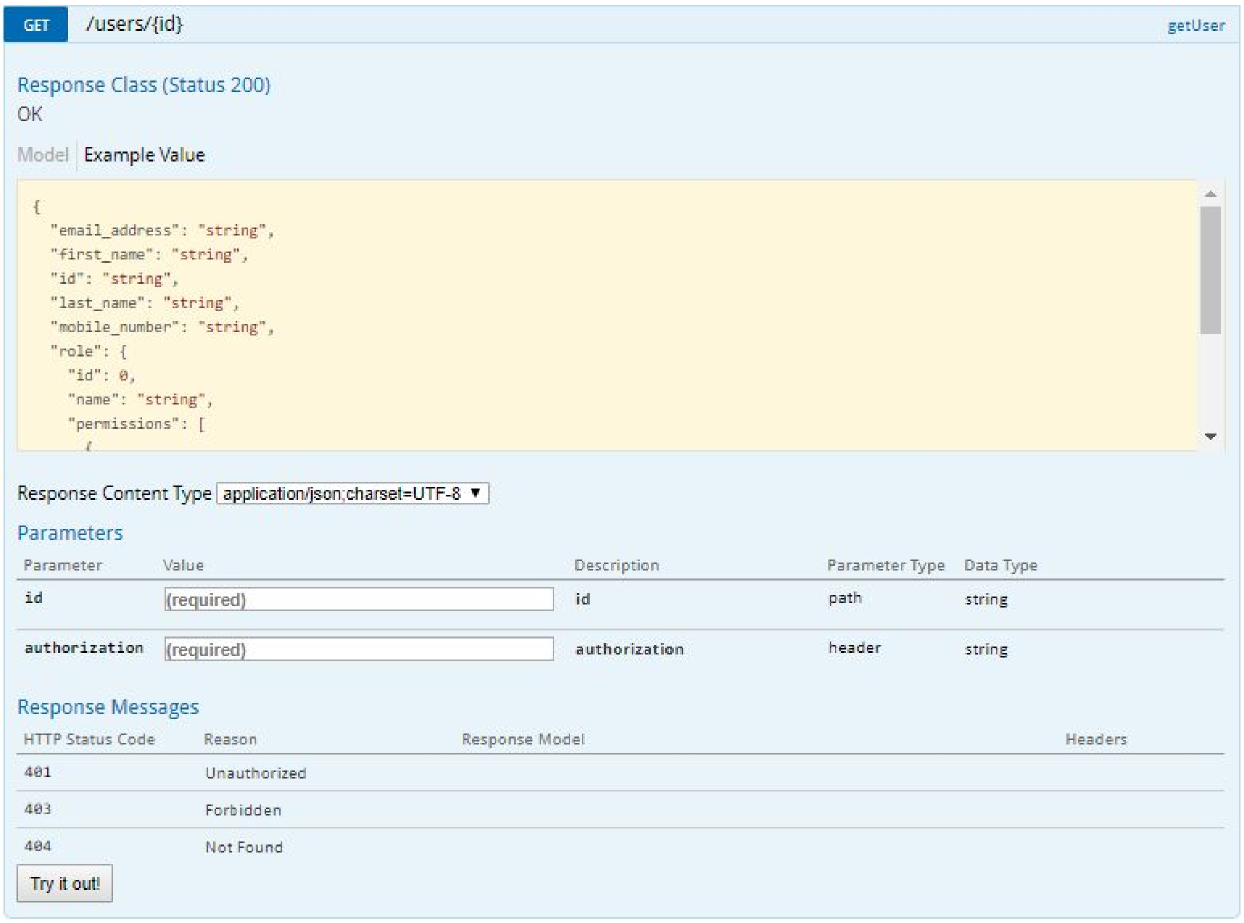
The modified code for the authorized get user endpoint is as shown below:

```
public UserEntity getUser(final String userUuid, final String authorizationToken) throws
ResourceNotFoundException,
        UnauthorizedException {


   UserAuthTokenEntity userAuthTokenEntity = userDao.getUserAuthToken(authorizationToken);
   RoleEntity role = userAuthTokenEntity.getUser().getRole();
   if (role != null && role.getUuid() == 101) {
        UserEntity userEntity = userDao.getUser(userUuid);
        if (userEntity == null) {
            throw new ResourceNotFoundException("USR-001", "User not found");
        }
        return userEntity;
   }
   throw new UnauthorizedException("ATH-002", "you are not authorized to fetch user details");
}
```

How the authorization is implemented in business logic?

- Service class receives the authorization token passed by the Controller class.
- This token is used to retrieve the user information from the database.
- The role of the user is retrieved and if the role of the user is not an admin, UnauthorizedException is thrown, indicating that the corresponding user is restricted to fetch the user details.
- Else, the user can fetch the the details of any user in the database.
- The uuid of the user is passed whose details are to be fetched from the database.
    - If the user with corresponding uuid does not exist in the database, throw ‘ResourceNotFoundException’, indicating that the user with corresponding uuid does not exist in the database.
    - Else, retrieve all the details of the corresponding user from the database.

Observe the authorization header for get user endpoint as shown below:



GET /users/{id} getUser

Response Class (Status 200)
OK

Model | Example Value

```
{
  "email_address": "string",
  "first_name": "string",
  "id": "string",
  "last_name": "string",
  "mobile_number": "string",
  "role": {
    "id": 0,
    "name": "string",
    "permissions": [
```

Response Content Type application/json;charset=UTF-8

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	(required)	id	path	string
authorization	(required)	authorization	header	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Figure 17 - Authorization

Now we have covered all the concepts used in creating the Backend API endpoints. In this project, all the endpoints are not authorized. This project is just to make you capable of implementing any functionality in creating the Backend APIs. After completing this, you may implement the ‘ImageHoster’ project with complete functionality. Also, you may not be 100% sure about the flow of the code. Implementing the ‘ImageHoster’ project will give you complete understanding and the confidence with the flow of the code. After that, you are ready to create Backend APIs of your own projects.