create an entity class to represent our data model.

create a repository interface to handle database operations for the entity.

create a service class to handle the business logic for our REST API.

create a controller class to handle HTTP requests for our REST API.

`RepairController` class:

1. **getAllRepairs()**:

   - This function handles HTTP GET requests to `/repairs`.

   - It retrieves all repair requests from the database using the `findAll()` method provided by `RepairRepo`.

   - The list of repair requests is returned as the response body.

2. **getRepairsByCustomerId(Long customerId)**:

   - This function handles HTTP GET requests to `/repairs/register/{customerId}`.

   - It takes a `customerId` as a path variable and retrieves all repair requests associated with that customer ID from the database using the findByCustomerCustomerId().

   - The list of repair requests is returned as the response body within a `ResponseEntity`. setting the response status to OK (200)

3. **createRepair(Repair repair)**:

   - This function handles HTTP POST requests to `/repairs`.

   - It creates a new repair request by accepting a `Repair` object in the request body.

   - The initial status of the repair request is set to "booked".

   - saves the repair request to the database using the save() method of the RepairRepo interface and returns the saved repair object.

   - The created repair request is returned as the response body.

4. **getRepairById(Long id)**:

   - This function handles HTTP GET requests to `/repairs/{id}`.

   - It takes a `id` as a path variable and retrieves the repair request with the specified ID from the database using the `findById()` method provided by `RepairRepo`.

   - If the repair request is found, it is returned as the response body within a `ResponseEntity`.

  - throws a ResourceNotFoundException if the requested repair request does not exist in the database.

5. **updateRepair(Long id, Repair repairDetails)**:

- This function handles HTTP PUT requests to `/repairs/{id}`.

- It takes an `id` as a path variable and the updated repair details as a `Repair` object in the request body.

- It retrieves the repair request with the specified ID from the database using the `findById()` method provided by `RepairRepo`.

- If the repair request is found, it update the details of the existing repair request with the details provided in the repairDetails object and saved back to the database using the `save()` method provided by `RepairRepo`.

- The updated repair request is returned as the response body within a `ResponseEntity`.

- throws a ResourceNotFoundException if the requested repair request does not exist in the database.

6. **deleteRepair(Long id)**:

- This function handles HTTP DELETE requests to `/repairs/{id}`.

- It takes an `id` as a path variable and deletes the repair request with the specified ID from the database using the `delete()` method provided by `RepairRepo`.

- If the repair request is successfully deleted, a map containing a message indicating successful deletion is returned as the response body within a `ResponseEntity`. setting the response status to OK (200)

- throws a ResourceNotFoundException if the requested repair request does not exist in the database.

7. **numberOfRepairs()**:

- This function handles HTTP GET requests to `/repair-number`.

- It calls the `numberOfRepairs()` method from the `RepairService`.

- The total number of repair requests is returned as the response body within a `ResponseEntity`.

- setting the response status to OK (200).

Repair Service :

this RepairService class acts as a bridge between the controller layer and the data access layer. It provides a method (numberOfRepairs()) to fetch the total number of repair requests. It calls the count() method on the repairRepository object. This counts the number of repairs in the Repair table.

Repair Repository :

JpaRepository provides various methods for performing database operations on entities, including saving, updating, deleting, and querying entities. specifying the type of entity class (Repair). It retrieves a list of repairs associated with a given customer ID.

Repair Model :

java.util.Date - specific instant in time.

javax.persistence -related to the Java Persistence API (JPA), used for working with databases in Java.

Lombok annotations used to automatically generate constructor, getter, setter, toString(), and other methods, reducing boilerplate code.

@Entity: Indicates that this class is a JPA entity, meaning it will be mapped to a database table.

@Table(name = "repairs"): Specifies the name of the database table to which this entity is mapped.

@Data: Lombok annotation to generate getter, setter, equals(), hashCode(), and toString() methods.

@AllArgsConstructor: Lombok annotation to generate a constructor with all fields as arguments.

@NoArgsConstructor: Lombok annotation to generate a no-argument constructor.

@id: Unique identifier for each repair, annotated with @Id  - Primary key

@GeneratedValue to indicate it's the primary key and its generation strategy.

@ManyToOne to establish a many-to-one relationship with the Register entity.

customer: Represents the customer associated with this repair.

@Column to specify the column names in the database.

Register Controller :

@RestController: This annotation indicates that the class defines a REST controller, which provides RESTful web services.will handle HTTP requests and produce HTTP responses.

@RequestMapping("/register"): This annotation specifies the base URI path for all the endpoints defined in this controller.

private RegisterRepo registerRepository;: This variable holds an instance of a repository (presumably for data persistence operations related to user registration).

private RegisterService registerService;: This variable holds an instance of a service class responsible for business logic related to user registration and authentication.

public RegisterController(RegisterService registerService): Constructor injection is used to inject an instance of RegisterService into the controller. This allows the controller to delegate business logic to the service layer.

1. **getAllCustomers()**:

- This is a GET request handler mapped to the root endpoint i.e. base url

- It retrieves all registered customers from the `registerService` and returns them as a list of `Register` objects.

- If there are no customers, it returns a 204 No Content response; otherwise, it returns a 200 OK response with the list of customers.

2. **updateUser(Long customerId, Register updatedUser)**:

This method updates the details of a registered customer identified by their customerId.

It first retrieves the existing user by calling registerService.findUserById(customerId).

If the customer with the specified ID doesn't exist, it returns a 404 Not Found response.

If the customer exists, it updates the customer's name, email, and phone number with the information provided in the updatedUser object.

It then calls registerService.updateUser(existingUser) to persist the updated user details.

Finally, it returns a 200 OK response with the updated user details in the response body.

3. **addregisteruser(Register register)**:

This method handles a POST request to create a new customer.

It receives the customer details in the request body as a Register object and adds the new customer using the registerService.

It returns a 201 Created response with the newly created customer's details.

4. **getCustomerById(Long customerId)**:

This method handles a GET request to retrieve the details of a registered customer by their customerId. It returns a ResponseEntity containing the customer details if the customer is found (200 OK). If the customer does not exist, a 404 Not Found response is returned.

5. **loginService(String email, String password)**:

This method handles a GET request to perform a login operation.

It validates the provided email and password using the loginValidation() method from registerService.

If the login is successful, it returns a map containing customer information along with a customerId.

If the login fails (invalid email/password), a 404 Not Found response is returned.

6. **deleteUser(Long customerId)**:

This method handles a DELETE request to delete a registered customer by their customerId. It calls the deleteUser() method from registerService to delete the customer. If the customer is successfully deleted, a 204 No Content response is returned. If the customer does not exist, a 404 Not Found response is returned..

7. **forgetPassword(String email, String newPassword)**:

This method resets a customer's password by their email.

It receives the email and a new password as request parameters.

It calls registerService.resetPassword(email, newPassword) to reset the password for the customer associated with the provided email.

If the password reset is successful, it returns a 200 OK response with a success message.

If an error occurs during the password reset process, it returns a 500 Internal Server Error response.

8. **numberOfCustomers()**:

This method retrieves the total number of registered customers.

It calls registerService.numberOfCustomers() to get the total count of customers.

It returns a 200 OK response with the total number of customers in the response body.

Register Service :

```
@Autowired
    private RegisterRepo registerRepository;
```

@Autowired annotation injects the RegisterRepo bean into this class, enabling the service to interact with the data layer through this repository.

addregister method - adds a new registration entry if both email and phone number provided in the Register object are unique. It checks for existing users with the same email and phone number using repository methods. If no user with the same email and phone exists, it saves the new registration entry and returns it; otherwise, it returns null.

loginValidation method - validates user credentials by checking if a user with the provided email exists and if the provided password matches the stored password. It returns a map containing the customerId and a status indicating whether the login was successful.

getAllCustomers retrieves all registered customers from the database using the repository's findAll method and returns them as a list.

resetPassword method - resets the password for the user associated with the provided email. If the email exists, it updates the password and returns a success message; otherwise, it returns a failure message.

updateUser method updates the user information in the database by saving the provided Register object.

deleteUser method deletes a user with the specified customerId if it exists in the database. It returns true if the deletion is successful, otherwise false.

findUserById method retrieves a user with the specified customerId from the database. It returns the user if found, otherwise null.

numberOfCustomers method returns the total number of registered customers in the database by utilizing the count method provided by the repository.

This RegisterService class encapsulates various operations related to user registration, authentication, updating user information, and managing user accounts in a Spring application. It utilizes a RegisterRepo for interacting with the underlying database.

It creates a Map object named response to construct the response body. In this case, it sets a key-value pair where the key is "deleted" and the value is Boolean.TRUE, indicating that the deletion was successful.

Exception class:

ResponseStatus annotation from the Spring Framework. This annotation allows us to specify the HTTP status code to be returned when this exception is thrown.

indicates that when an instance of ResourceNotFoundException is thrown, the HTTP response status should be set to "404 Not Found".

private static final long serialVersionUID = 1L;: This line declares a serialVersionUID to ensure compatibility of the class during serialization and deserialization.

requested resource is not found in a Spring application. When this exception is thrown, it automatically sets the HTTP response status to "404 Not Found", providing a clear indication to the client that the requested resource could not be found.

spring.jpa.hibernate.ddl-auto=update

the behavior of Hibernate's Data Definition Language (DDL) auto generation.

Hibernate will automatically update the database schema based on the entity classes when the application starts.

spring.datasource.url=jdbc:mysql://localhost:3306/acservice:

it's connecting to a MySQL database

spring.datasource.username=root:

Sets the username for connecting to the MySQL database.

spring.datasource.password=Pavithra@28:

Specifies the password for the MySQL for authenticate

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect:

Specifies the Hibernate dialect to use for MySQL database.

spring.jpa.show-sql=true:

Spring Data JPA to output SQL statements to the console. When set to "true".

enable.swagger.plugin=true:  enables the Swagger plugin.

Swagger is a framework for documenting APIs, and enabling this plugin allows generating API documentation automatically.

spring.mvc.pathmatch.matching-strategy=ant-path-matcher:

the strategy used to match incoming request URLs to controller methods.

application properties :

file are used to customize the behavior of the Spring application and its interaction with the MySQL database.