2.Relational database

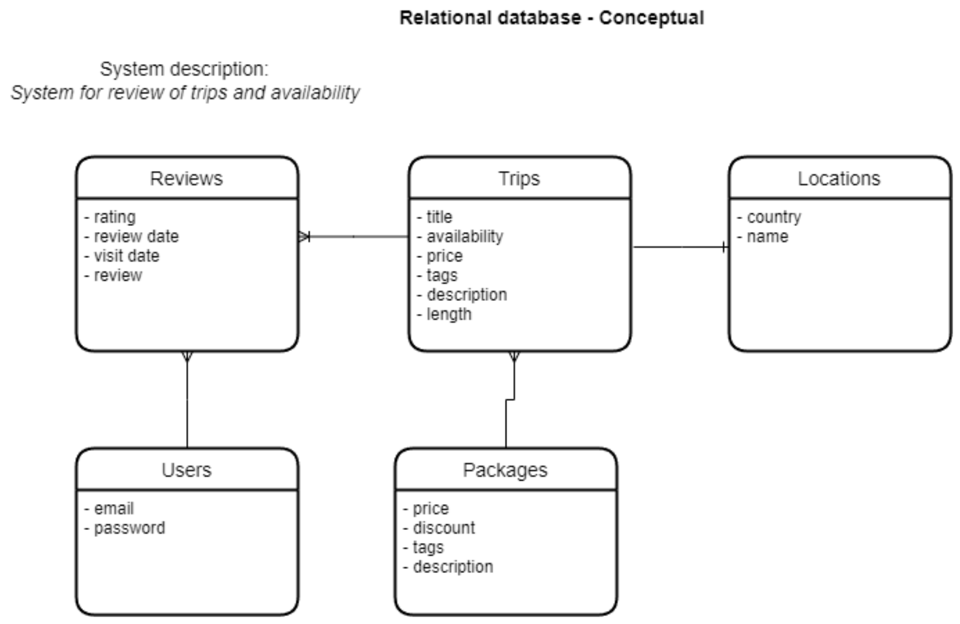
2.1. Intro to relational databases

A relational database is a database that follows the relational model and store data in a table format with rows, columns and unique keys for each data point.

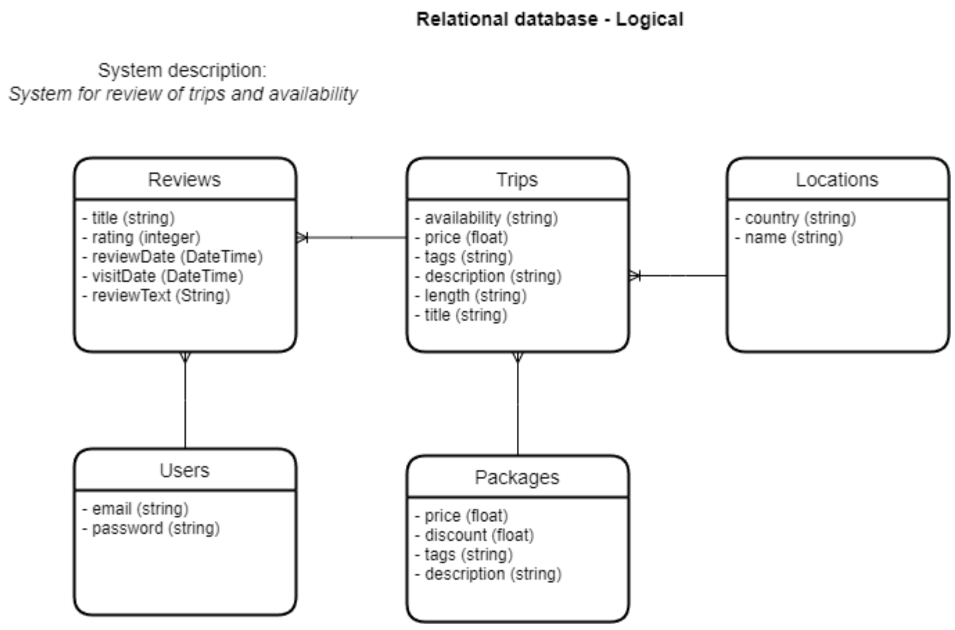
2.2. Database design

2.2.1. Entity/Relationship Model (Conceptual -> Logical -> Physical model)

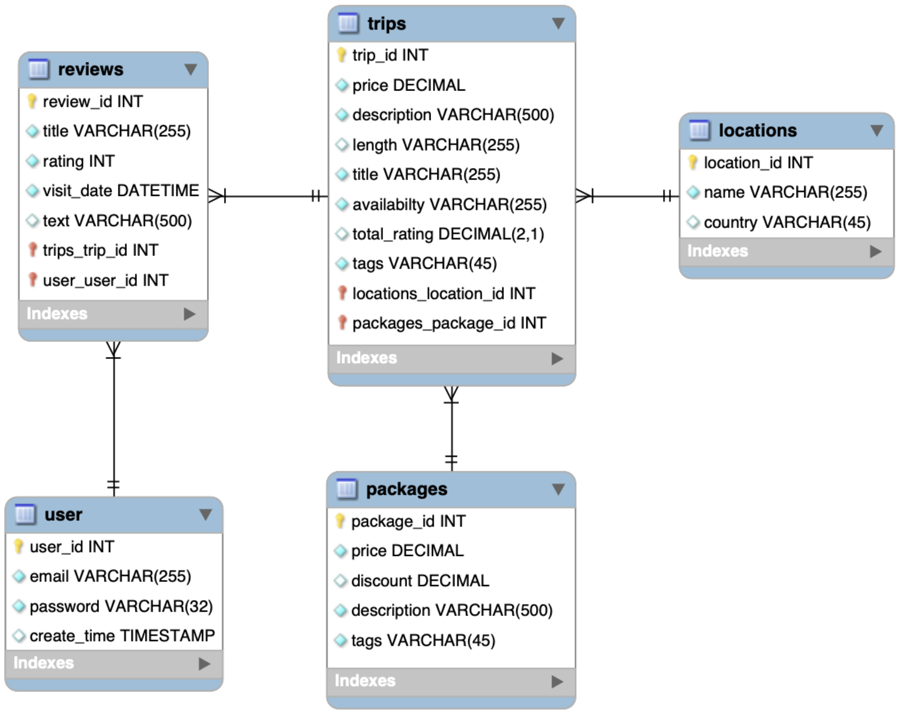
2.2.1.1 Conceptual model



2.2.1.2 Logical model



2.2.1.3 Physical model

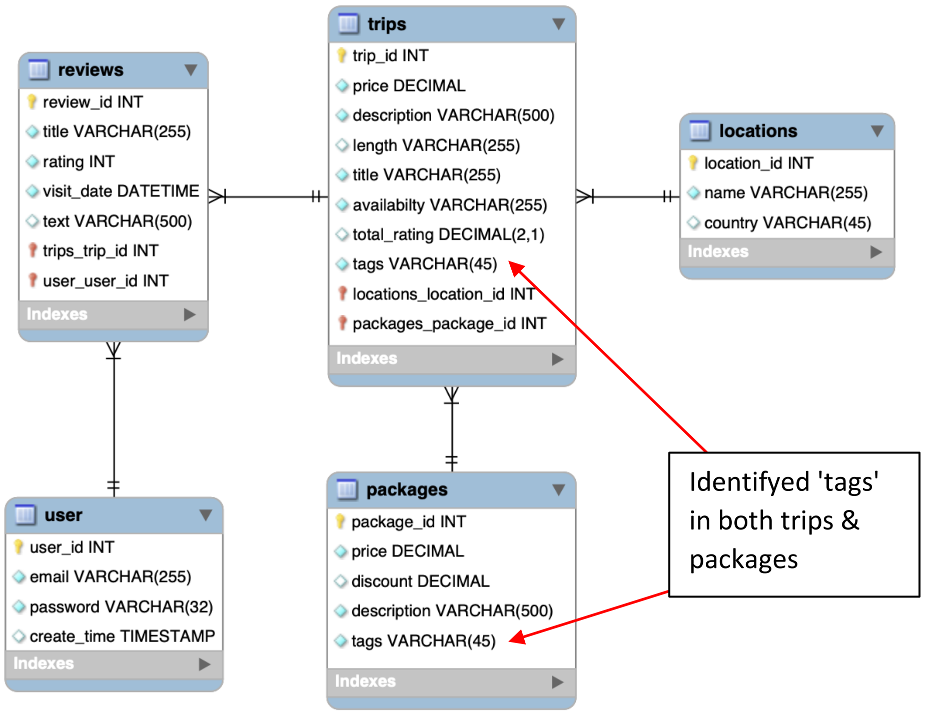


2.2.3. Normalization process

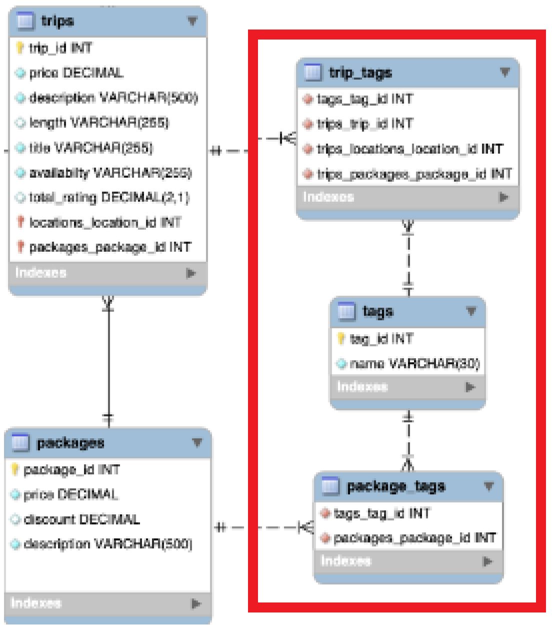
From Physical model in section 2.2.1.3, we started identifying 1. normal form, that says:

- Each cell should have a single value

- We cannot have repeated columns



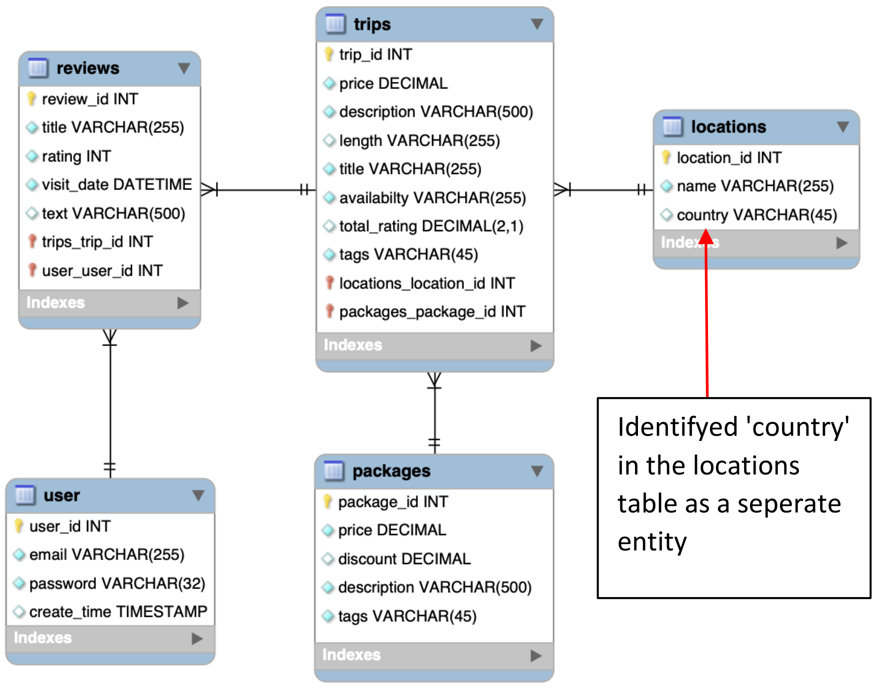
We then spit ***tags*** up in three different columns: trip tags, package tags which hold different kind of tags from the last column tags. Here is what It looks like in the final physical data model:



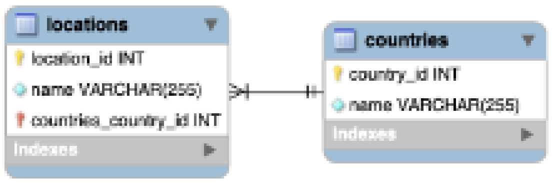
After 1. normal form, we moved to 2. normal form which says:

- Each table should only describe 1 entity

- Every column in that table should describe that entity



Here is what it looks like after we put them into two tables:

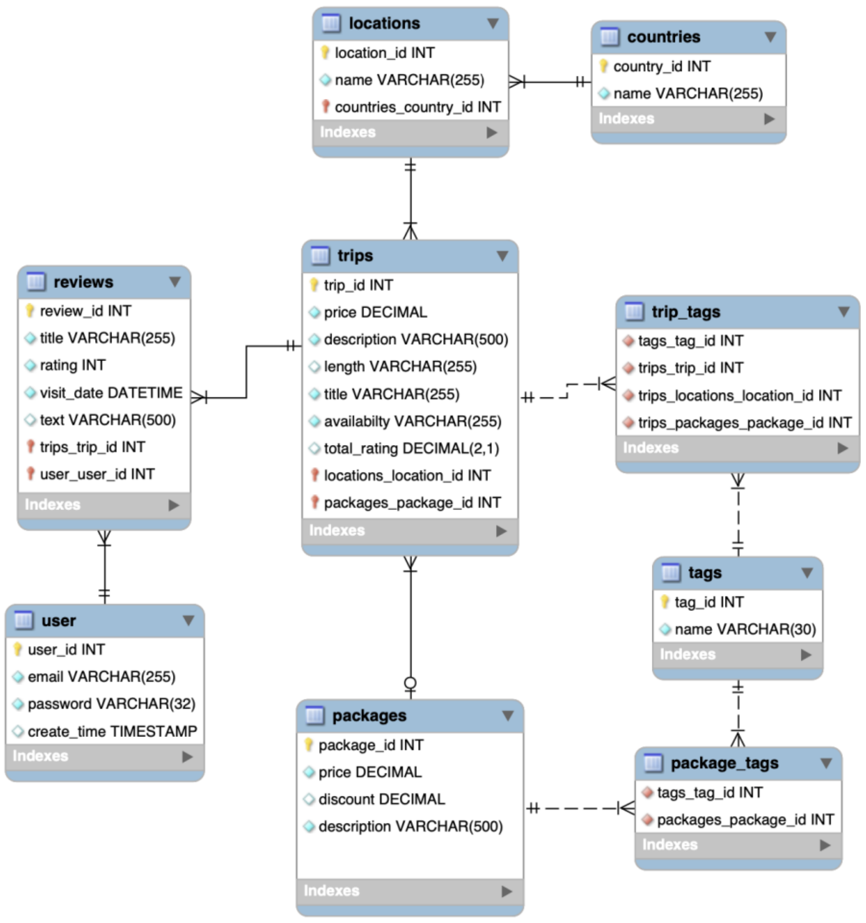


To complete our normalizing process, we looked at the tables to see if it fulfilled the 3. normal form which says:

- To be in 3. normal form, a column should not be derived from other columns and must be in 2. normal form

To identify transient dependencies, we looked to see if any of the fields in our tables had higher dependency on another column than the primary key. Here none were identified, and our data model was done which is shown in section 2.3 down below.

2.3. Physical data model



2.3.1. Data types

- Datetime
- Int
- Varchar
- Decimal
- Timestamp

2.3.2. Primary and foreign keys

Primary keys:

- Location_id
- User_id
- Package_id
- Tag_id
- Reviews_id

- Trips_id

Foreign keys:

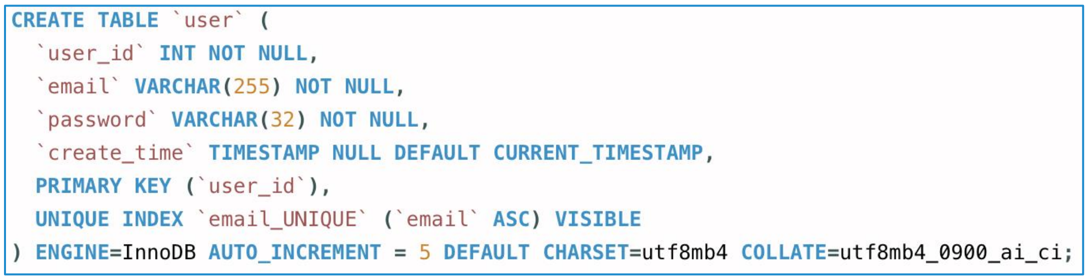
- Locations_location_id -> the trips table
- Packages_package_id -> the trips table
- Trips_trip_id -> the reviews table
- User_user_id -> the reviews table
- Countries_country_id -> the location table

2.3.3. Indexes(skal ikke laves denne gang)

2.3.4. Constraints and referential integrity

Constraints

- **Foreign key** constraints are used throughout the database. Here all foreign keys follow the convention: location_id references locations.
- **Not null** is used to ensure an attribute must be assigned. This is important for the user entity since it cannot exist either without an id, email, or password.
- The email attribute in user is **unique** since we only want one user registered per email address.
- The create_time attribute has a **default** constraint as timestamp, which automatically provides it with a date and time value when created.



```sql
CREATE TABLE `user` (
  `user_id` INT NOT NULL,
  `email` VARCHAR(255) NOT NULL,
  `password` VARCHAR(32) NOT NULL,
  `create_time` TIMESTAMP NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`user_id`),
  UNIQUE INDEX `email_UNIQUE` (`email` ASC) VISIBLE
) ENGINE=InnoDB AUTO_INCREMENT = 5 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

The different constraints for user table.

Referential integrity

We tried to avoid having many-to-many relations with creating composite entity.

- **Countries** and **locations** have a **one-to-many** referential integrity, meaning that one country can have one or more locations.
- **Locations** and **trips** have a **one-to-many** referential integrity, meaning that trip can only have one location, but location can have multiple trips.
- **Trips** and **reviews** have a **one-to-many** referential integrity, meaning that one trip can have many reviews, but each review can only be on one trip

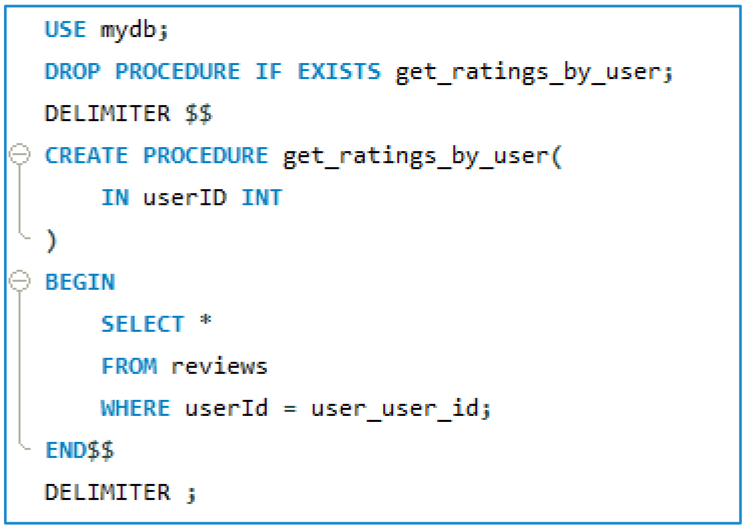
- **Review** and **user** have a **one-to-many** referential integrity, meaning that one user can have many reviews, but each review can only be made by one user.
- **Trips** and **packages** have a **one-to-many** referential integrity with modality is not mandatory, meaning that a trip can be part of zero or many packages, but package can have one or more trips.
- **Trips** and **trip_tags** have a **one-to-many** referential integrity, meaning trips can have one or more tags but each tag can only be on one trip
- **Packages** and **package_tags** have a **one-to-many** referential integrity, meaning package can have one or more tags but each tag can only be on one package
- **trip_tags** and **package_tags** have a **one-to-many** referential integrity with **tags**, meaning that **tags** can have many **trip_tags** or **package_tags**.

2.4. Stored objects –stored procedures/functions, views, triggers, events

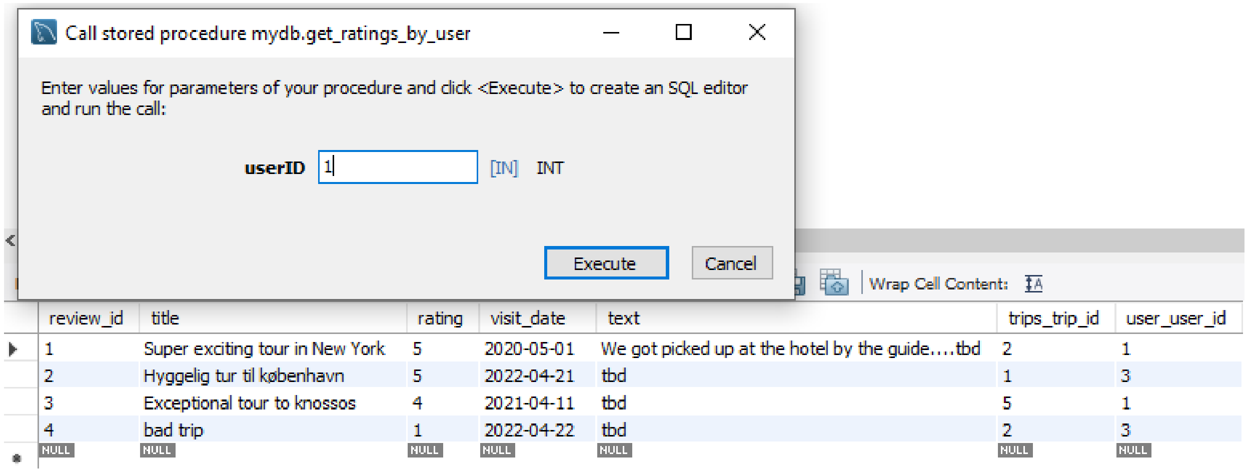
2.4.1 Stored procedures

Instead of writing the same statement repeatedly the stored procedures can be used. With stored procedures the most used statements can be saved and executed when there is a need for them.

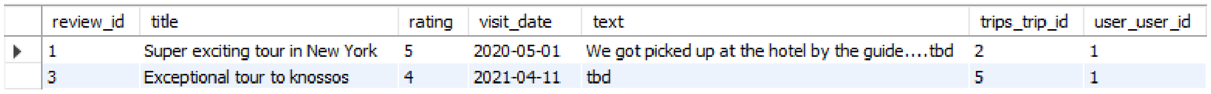
In this example we are making a stored procedure which gets the ratings by user id.



```
USE mydb;
DROP PROCEDURE IF EXISTS get_ratings_by_user;
DELIMITER $$
CREATE PROCEDURE get_ratings_by_user(
    IN userID INT
)
BEGIN
    SELECT *
    FROM reviews
    WHERE userId = user_user_id;
END$$
DELIMITER ;
```


Call stored procedure mydb.get_ratings_by_user
Enter values for parameters of your procedure and click <Execute> to create an SQL editor and run the call:
userID 1 [IN] INT
Execute
Cancel
Wrap Cell Content:

review_id	title	rating	visit_date	text	trips_trip_id	user_user_id
1	Super exciting tour in New York	5	2020-05-01	We got picked up at the hotel by the guide....tbd	2	1
2	Hyggelig tur til københavn	5	2022-04-21	tbd	1	3
3	Exceptional tour to knossos	4	2021-04-11	tbd	5	1
4	bad trip	1	2022-04-22	tbd	2	3
NULL	NULL	NULL	NULL	NULL	NULL	NULL



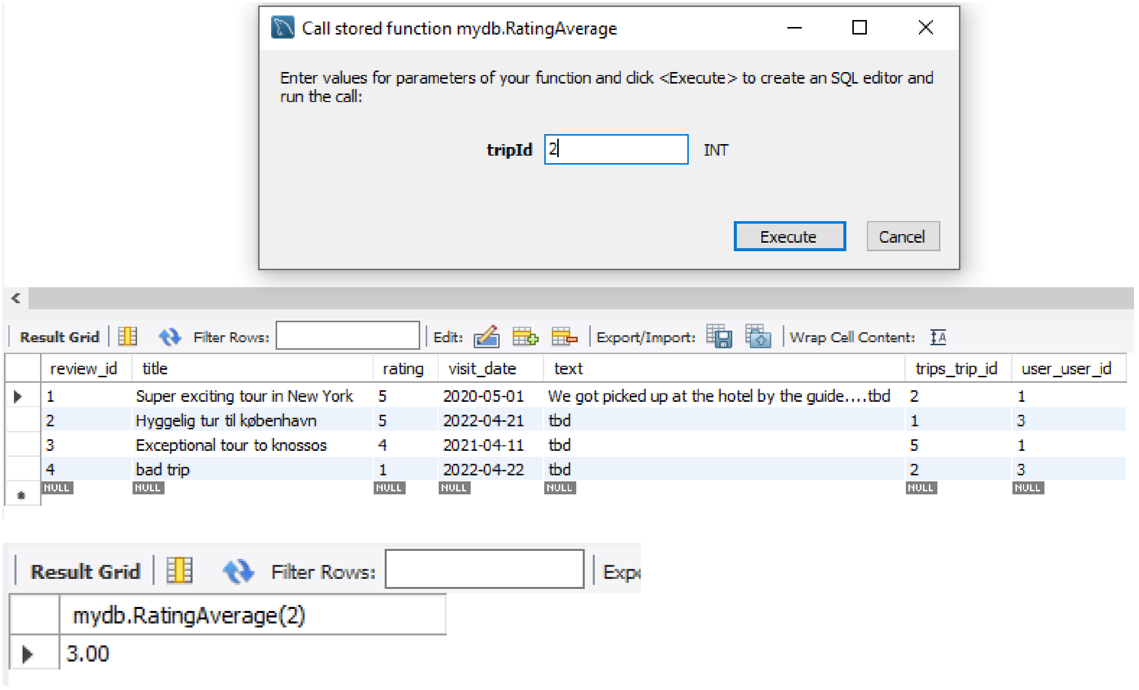
review_id	title	rating	visit_date	text	trips_trip_id	user_user_id
1	Super exciting tour in New York	5	2020-05-01	We got picked up at the hotel by the guide....tbd	2	1
3	Exceptional tour to knossos	4	2021-04-11	tbd	5	1

2.4.2 Stored functions

A stored function is a stored program. The code for creating a stored function is in many ways like creating a stored procedure, the most important difference is that the stored function returns a single value.

Using this stored function, we can calculate the average of a single trip rating.

```
USE MYDB;
DROP FUNCTION IF EXISTS RatingAverage;

DELIMITER $$
CREATE FUNCTION RatingAverage(tripId INT)
RETURNS decimal(9,2)
DETERMINISTIC
BEGIN
    DECLARE ratingAverage INT;
    SET ratingAverage = (SELECT AVG(`rating`) FROM reviews WHERE tripId = trips_trip_id);
    RETURN ratingAverage;
END$$

DELIMITER ;
```



Enter values for parameters of your function and click <Execute> to create an SQL editor and run the call:

tripId 2 INT

review_id	title	rating	visit_date	text	trips_trip_id	user_user_id
1	Super exciting tour in New York	5	2020-05-01	We got picked up at the hotel by the guide....tbd	2	1
2	Hyggelig tur til københavn	5	2022-04-21	tbd	1	3
3	Exceptional tour to knossos	4	2021-04-11	tbd	5	1
4	bad trip	1	2022-04-22	tbd	2	3
NULL	NULL	NULL	NULL	NULL	NULL	NULL

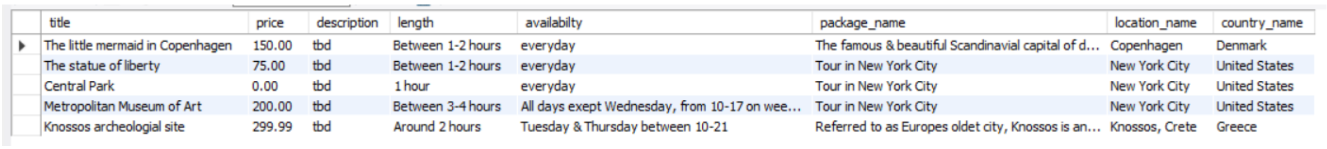
mydb.RatingAverage(2)
3.00

2.4.3 Views

We use views to display our database entities with the relevant data for the end user. For example, in our trips entity we display the name variable of package, location and country instead of the id variable. We end up with a view which contains the data we want to display from the object.

```
VIEW `mydb`.`trip_view` AS
    SELECT
        `mydb`.`trips`.`title` AS `title`,
        `mydb`.`trips`.`price` AS `price`,
        `mydb`.`trips`.`description` AS `description`,
        `mydb`.`trips`.`length` AS `length`,
        `mydb`.`trips`.`availabilty` AS `availabilty`,
        `mydb`.`packages`.`description` AS `package_name`,
        `mydb`.`locations`.`name` AS `location_name`,
        `mydb`.`countries`.`name` AS `country_name`
    FROM
        (((`mydb`.`trips`
        JOIN `mydb`.`locations` ON ((`mydb`.`trips`.`locations_location_id` = `mydb`.`locations`.`location_id`)))
        JOIN `mydb`.`packages` ON ((`mydb`.`trips`.`packages_package_id` = `mydb`.`packages`.`package_id`)))
        JOIN `mydb`.`countries` ON ((`mydb`.`locations`.`countries_country_id` = `mydb`.`countries`.`country_id`)))
```

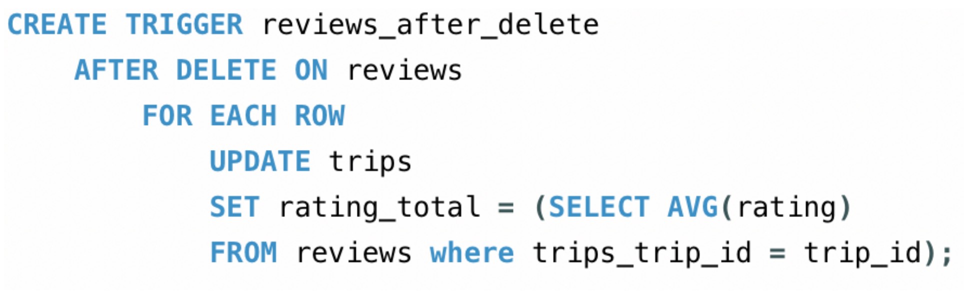
In the script we display the trips table where location_name and package_name are joined from their id's. Country_name is also joined with reference to countries_country_id from the locations table.



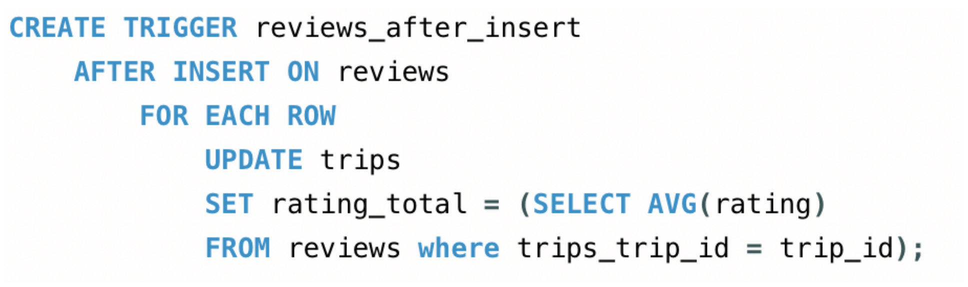
title	price	description	length	availabilty	package_name	location_name	country_name
The little mermaid in Copenhagen	150.00	tbd	Between 1-2 hours	everyday	The famous & beautiful Scandinavial capital of d...	Copenhagen	Denmark
The statue of liberty	75.00	tbd	Between 1-2 hours	everyday	Tour in New York City	New York City	United States
Central Park	0.00	tbd	1 hour	everyday	Tour in New York City	New York City	United States
Metropolitan Museum of Art	200.00	tbd	Between 3-4 hours	All days exept Wednesday, from 10-17 on wee...	Tour in New York City	New York City	United States
Knossos archeologial site	299.99	tbd	Around 2 hours	Tuesday & Thursday between 10-21	Referred to as Europes oldet city, Knossos is an...	Knossos, Crete	Greece

2.4.4 Triggers

For triggers we made two triggers that automatically update trips rating total after either an insert on reviews or delete on reviews.

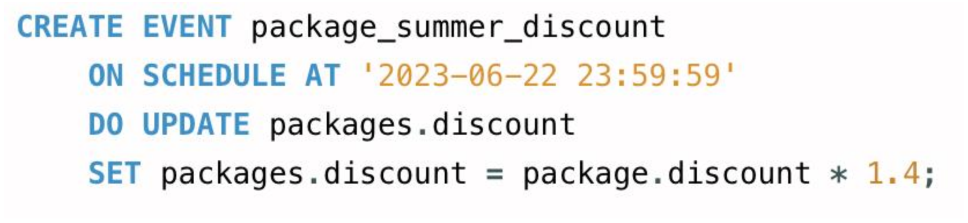


```
CREATE TRIGGER reviews_after_delete
    AFTER DELETE ON reviews
        FOR EACH ROW
            UPDATE trips
            SET rating_total = (SELECT AVG(rating)
            FROM reviews where trips_trip_id = trip_id);
```



```
CREATE TRIGGER reviews_after_insert
    AFTER INSERT ON reviews
        FOR EACH ROW
            UPDATE trips
            SET rating_total = (SELECT AVG(rating)
            FROM reviews where trips_trip_id = trip_id);
```

2.4.5 Events



```
CREATE EVENT package_summer_discount
    ON SCHEDULE AT '2023-06-22 23:59:59'
    DO UPDATE packages.discount
    SET packages.discount = package.discount * 1.4;
```

2.5. Transactions. Explanation of the structure and implementation of transactions

Have not implemented transactions yet.

2.6. Auditing. Explanation of the audit structure implemented with triggers

Have not implemented transactions yet.

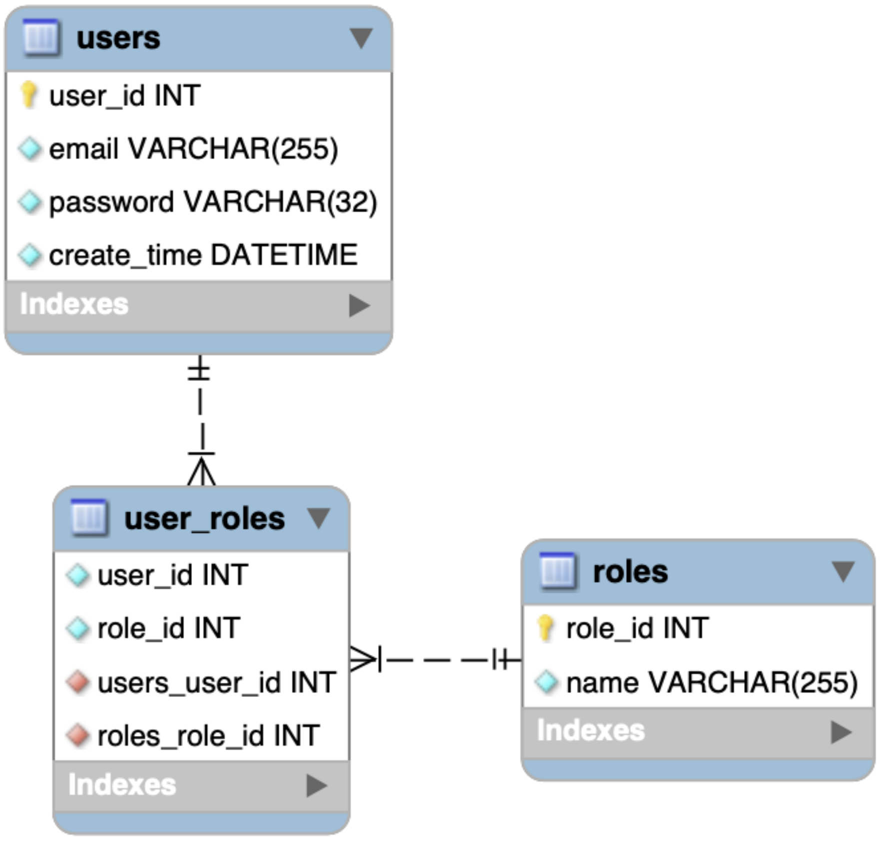
2.7. Security.

2.7.1. Explanation of users and privileges

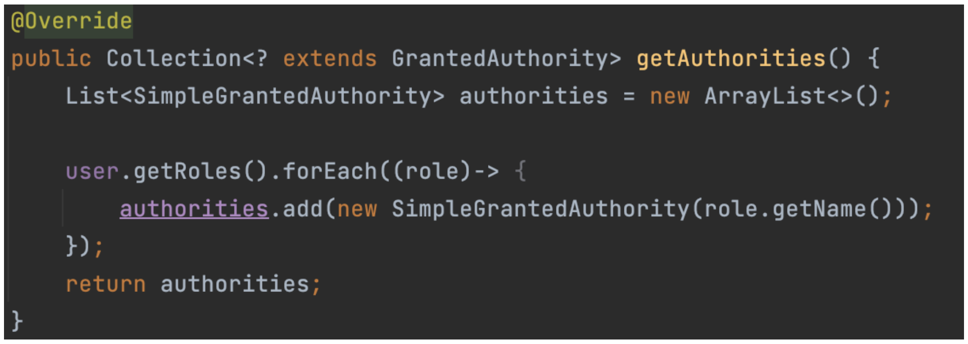
Users and privileges security is where we give each user a role that defines their privileges.

A role could either be Administrator or default user, where the administrator can delete or modify default users in at a specific endpoint, where if default user tries to access the same endpoint, an error would occur because of different privileges. This is also called Role-based access control.

In our application we use RBAC (Role-based access control) by using spring boot security dependency and adding roles and user roles tables to our database from the figure of the physical model in section 2.3. Here is the relationship between each table:

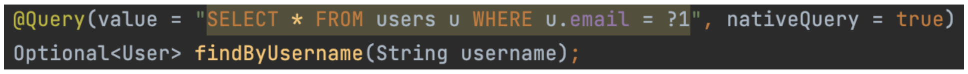


For giving users different privileges the first we did was to give each role an authority in the MyUserDetails class that Inherit UserDetails.



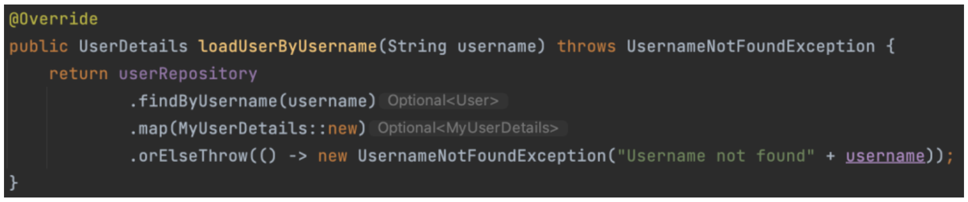
```
@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    List<SimpleGrantedAuthority> authorities = new ArrayList<>();

    user.getRoles().forEach((role)-> {
        authorities.add(new SimpleGrantedAuthority(role.getName()));
    });
    return authorities;
}
```

Then we created a Query to find user by username with a parameter, this I later used for loading the security login we added with spring boot security



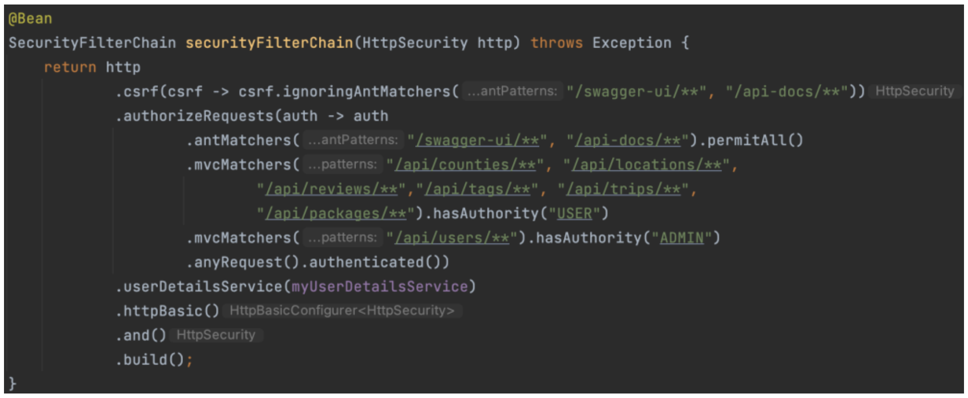
```java
@Query(value = "SELECT * FROM users u WHERE u.email = ?1", nativeQuery = true)
Optional<User> findByUsername(String username);
```

In UserDetails service we then used our query to load username and map MyUserDetails that contains the authority.



```java
@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
    return userRepository
            .findByUsername(username) Optional<User>
            .map(MyUserDetails::new) Optional<MyUserDetails>
            .orElseThrow(() -> new UsernameNotFoundException("Username not found" + username));
}
```

Then in our Security configurations we set the privileges for each role, where it's only admin that can access the user's endpoint.

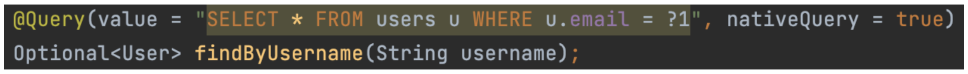


```java
@Bean
SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    return http
            .csrf(csrf -> csrf.ignoringAntMatchers( ...antPatterns: "/swagger-ui/**", "/api-docs/**")) HttpSecurity
            .authorizeRequests(auth -> auth
                    .antMatchers( ...antPatterns: "/swagger-ui/**", "/api-docs/**").permitAll()
                    .mvcMatchers( ...patterns: "/api/counties/**", "/api/locations/**",
                            "/api/reviews/**","/api/tags/**", "/api/trips/**",
                            "/api/packages/**").hasAuthority("USER")
                    .mvcMatchers( ...patterns: "/api/users/**").hasAuthority("ADMIN")
                    .anyRequest().authenticated())
            .userDetailsService(myUserDetailsService)
            .httpBasic() HttpBasicConfigurer<HttpSecurity>
            .and() HttpSecurity
            .build();
}
```

2.7.2. SQL Injection – what is it and how it is dealt with in the project?

SQL injection is a vulnerability that allows attackers to access data that they should not be able to retrieve, such as data belonging to other users by injecting SQL statements in a form field or a URL. In other cases, an SQL injection might be able to modify or delete data in the database.

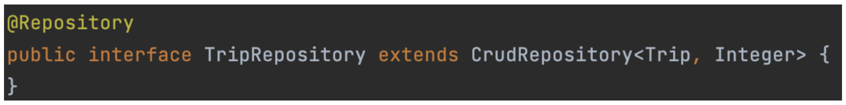
In the application we deal with SQL injection in different ways, such as we use @Query from spring data JPA which is parameterized by default, here is an example from our code:



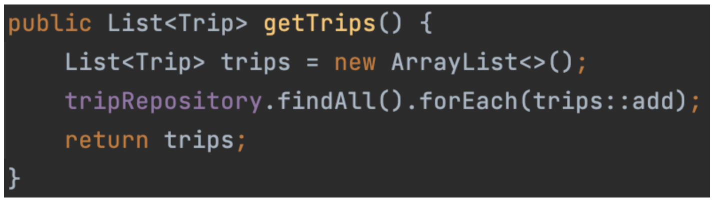
```java
@Query(value = "SELECT * FROM users u WHERE u.email = ?1", nativeQuery = true)
Optional<User> findByUsername(String username);
```

Parameterized query is another name for prepared statements and uses a placeholder for a parameter, the parameter value is then supplied at runtime and is safe against SQL injection.

Another thing we do to avoid SQL injection is to use the default queries from a CRUD repository where our repositories inherit from the CrudRepository that contains methods such as: findAll, findAllById, findById, deleteAll, deleteById, save, saveAll. Here is an example of how we use the method findAll to create a list of all trips in our database.



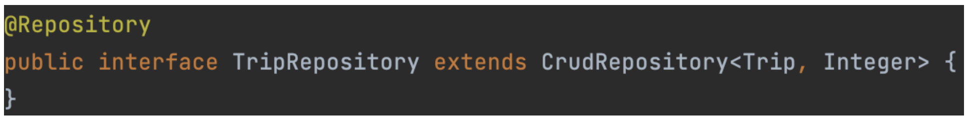
```java
@Repository
public interface TripRepository extends CrudRepository<Trip, Integer> {
}
```



```java
public List<Trip> getTrips() {
    List<Trip> trips = new ArrayList<>();
    tripRepository.findAll().forEach(trips::add);
    return trips;
}
```

2.8. Description of the CRUD application for RDBMS – REST APIs, service layer, security – registration / login, transactions, etc.

For the CRUD application, we used interfaces that implemented CrudRepository, that has different methods such as: findAll, findAllById, findById, deleteAll, deleteById, save, saveAll, for create, read, update & delete.

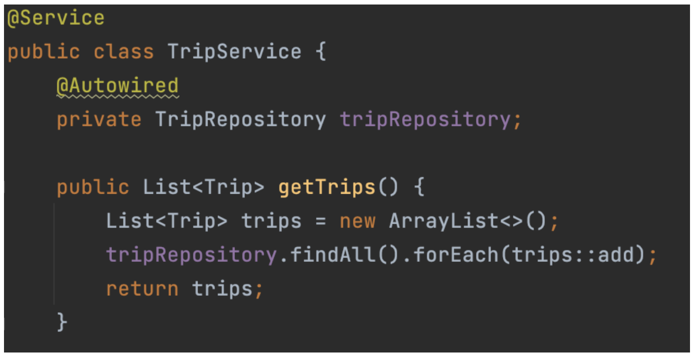


```java
@Repository
public interface TripRepository extends CrudRepository<Trip, Integer> {
}
```

These methods are then used in the service layer, which is described in section 2.8.1 service layer.

2.8.1. Service layer

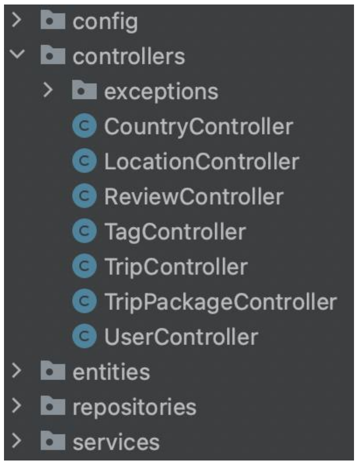
The service layer in our CRUD application for RDBMS is responsible for any logic with data from the controller and facilitates communication between the controller and the persistence layer. Here is an example from the code:



```java
@Service
public class TripService {
    @Autowired
    private TripRepository tripRepository;

    public List<Trip> getTrips() {
        List<Trip> trips = new ArrayList<>();
        tripRepository.findAll().forEach(trips::add);
        return trips;
    }
```

2.8.2. REST API

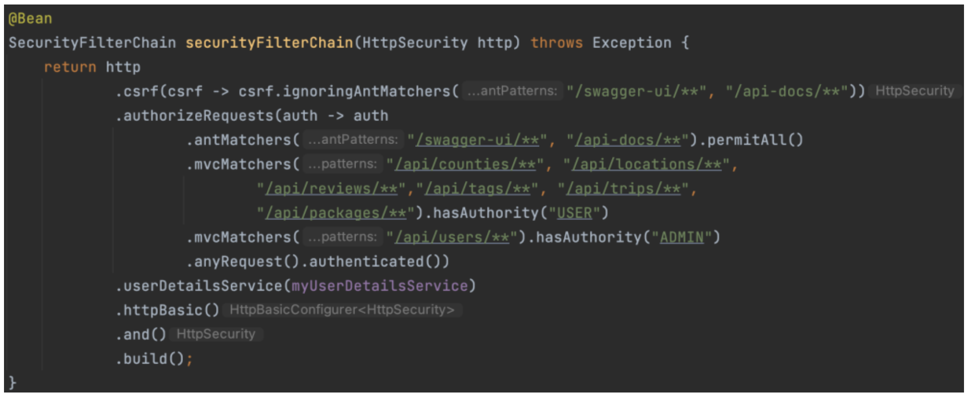
In the CRUD application for RDBMS we spring boot, spring annotation @RestController to build REST API. The Annotation allows the class to handle requests by the client.



```
> config
v controllers
    > exceptions
      CountryController
      LocationController
      ReviewController
      TagController
      TripController
      TripPackageController
      UserController
> entities
> repositories
> services
```

2.8.3. Security login

For implementing security login in spring boot, we added the dependency spring boot security which creates a login page. Here are which users can login to different endpoints:



```
@Bean
SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    return http
            .csrf(csrf -> csrf.ignoringAntMatchers( ...antPatterns: "/swagger-ui/**", "/api-docs/**")) HttpSecurity
            .authorizeRequests(auth -> auth
                    .antMatchers( ...antPatterns: "/swagger-ui/**", "/api-docs/**").permitAll()
                    .mvcMatchers( ...patterns: "/api/counties/**", "/api/locations/**",
                            "/api/reviews/**","/api/tags/**", "/api/trips/**",
                            "/api/packages/**").hasAuthority("USER")
                    .mvcMatchers( ...patterns: "/api/users/**").hasAuthority("ADMIN")
                    .anyRequest().authenticated())
            .userDetailsService(myUserDetailsService)
            .httpBasic() HttpBasicConfigurer<HttpSecurity>
            .and() HttpSecurity
            .build();
}
```

2.8.4. Transactions

Have not implemented transactions yet.

2.8.5 ORM (Object-relational mapping)

JPA:

Stands for Jakarta Persistence Api originally Java Persistence APi. It provides a specification for persisting, reading and managing data from your Java object to relational tables in the database.

Hibernate:

It is an Object-Relational-Mapping solution for Java environments that provides a framework for mapping.

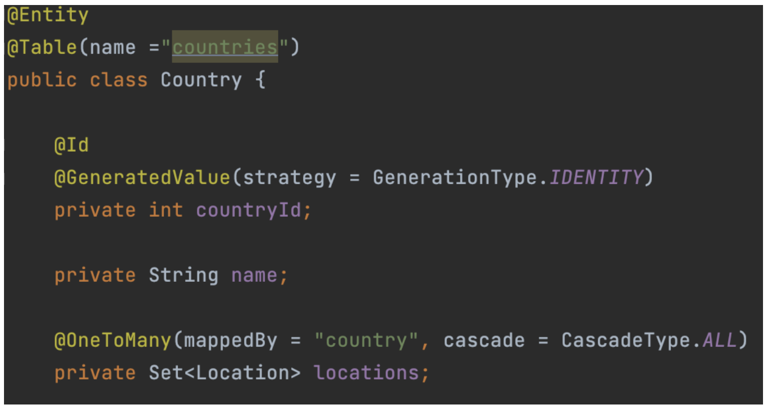
Object-Relational-Mapping or the abbreviation ORM is the programming technique to map application domain model objects to the relational database tables.

Spring Data JPA

Is a Library/framework that adds an extra layer of abstraction on the top layer of JPA provider like Hibernate.

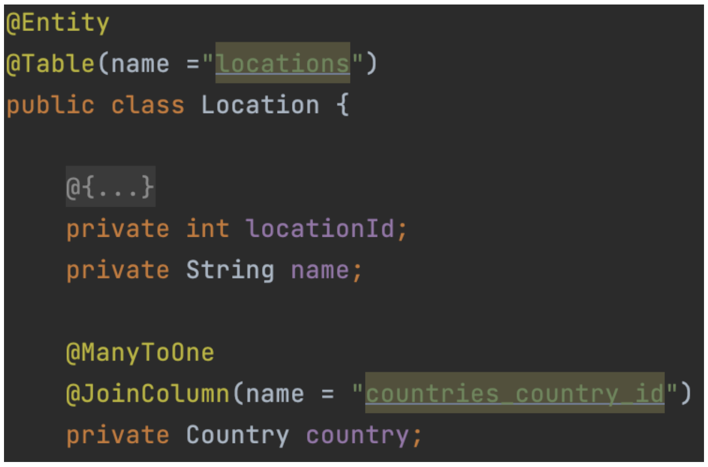
The difference between Hibernate and Spring Data JPA is that Hibernate is a JPA implementation while Spring Data JPA is a Data Access Abstraction. Spring Data JPA always requires the JPA provider such as Hibernate or others to work.

In the application for RDBMS, we used mapping for our entities. Here is an example of the countries table, that is mapped to have one to many locations.



```java
@Entity
@Table(name ="countries")
public class Country {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int countryId;

    private String name;

    @OneToMany(mappedBy = "country", cascade = CascadeType.ALL)
    private Set<Location> locations;
```

In the location entity, we can then see how the association is setup with many to one country. This means that if the country is deleted from the database, it removes all locations the country has.



```java
@Entity
@Table(name ="locations")
public class Location {

    @{...}
    private int locationId;
    private String name;

    @ManyToOne
    @JoinColumn(name = "countries_country_id")
    private Country country;
```