# Application Description

**Introduction**

Application simulates information system operation in some company, which provides passenger rail transportation. There is *four tier architecture* in this application:

1. Data access layer
2. Service layer
3. Controller layer
4. Presentation layer

Application has 2 Maven-based modules – “core” (contains data access and service layers) and “ui” (contains controller and presentation layers).

**“Core” Module**

**Data Access Layer**

Data access layer contains methods and POJO-classes for accessing the database data. Database schema is presented on the figure 1.

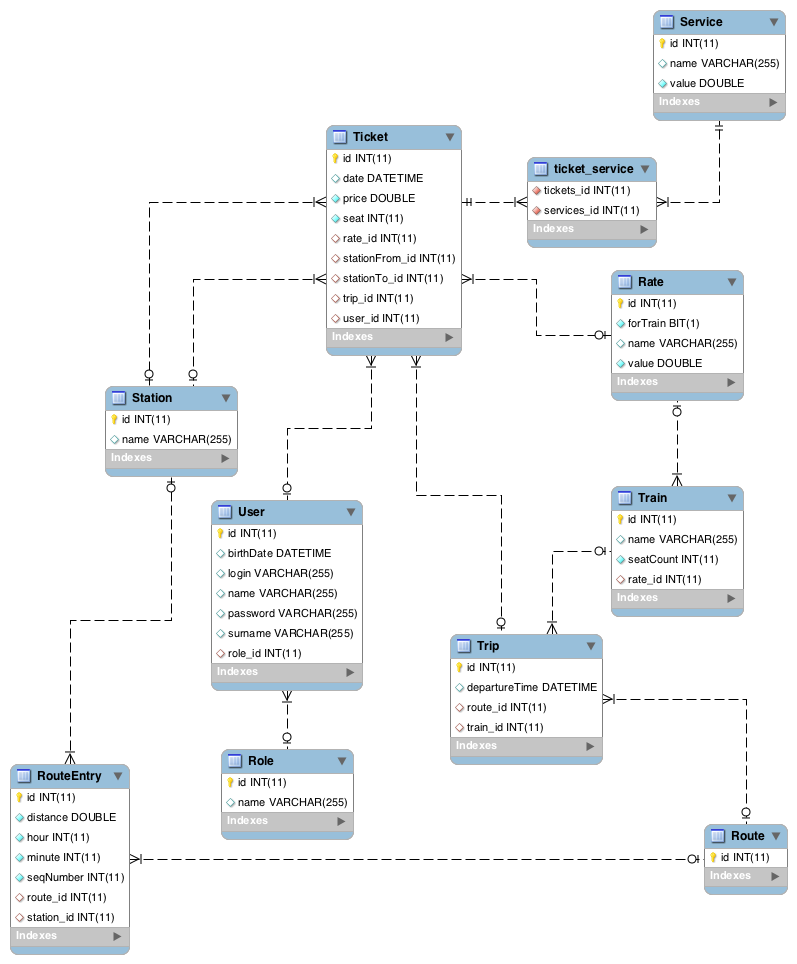


Figure 1. Database schema

I have worked with relational data as objects via Hibernate ORM. There are following entity classes that matches tables in database:

1. User (information about users of information system)
2. Trip (trains trips on 45-days period)
3. Train (information about trains)
4. Ticket (purchase information about ticket on trip)
5. Station (information about stations)
6. Service (information about additional services)
7. RouteEntry (information about route points)
8. Route (route number)
9. Role (user roles for securiry purposes)
10. Rate (types of trains, types of clients)

Class diagram is represented on the figure 3. Interaction between data access layer and service layer is provided by repository interfaces (figure 2) that automatically implemented by Spring Data JPA.

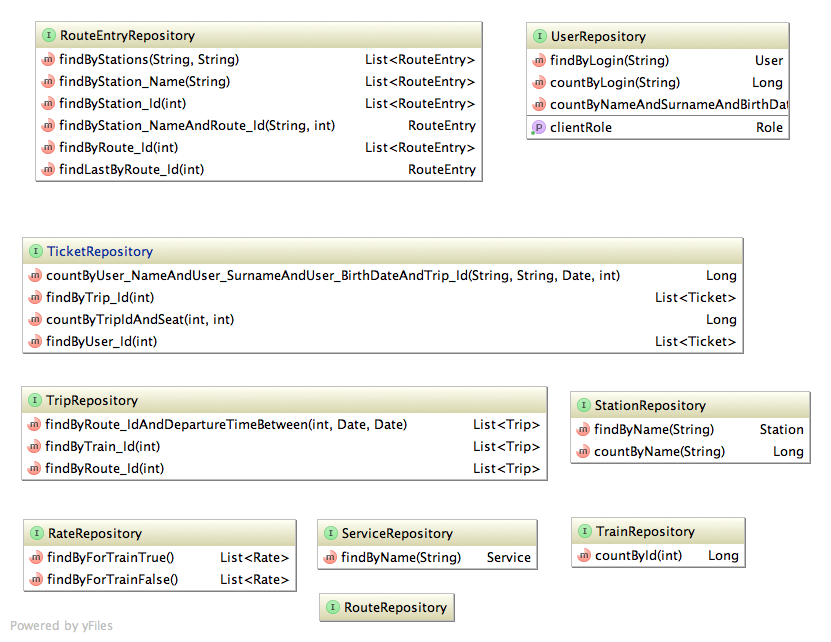


Figure 2. Repository interfaces

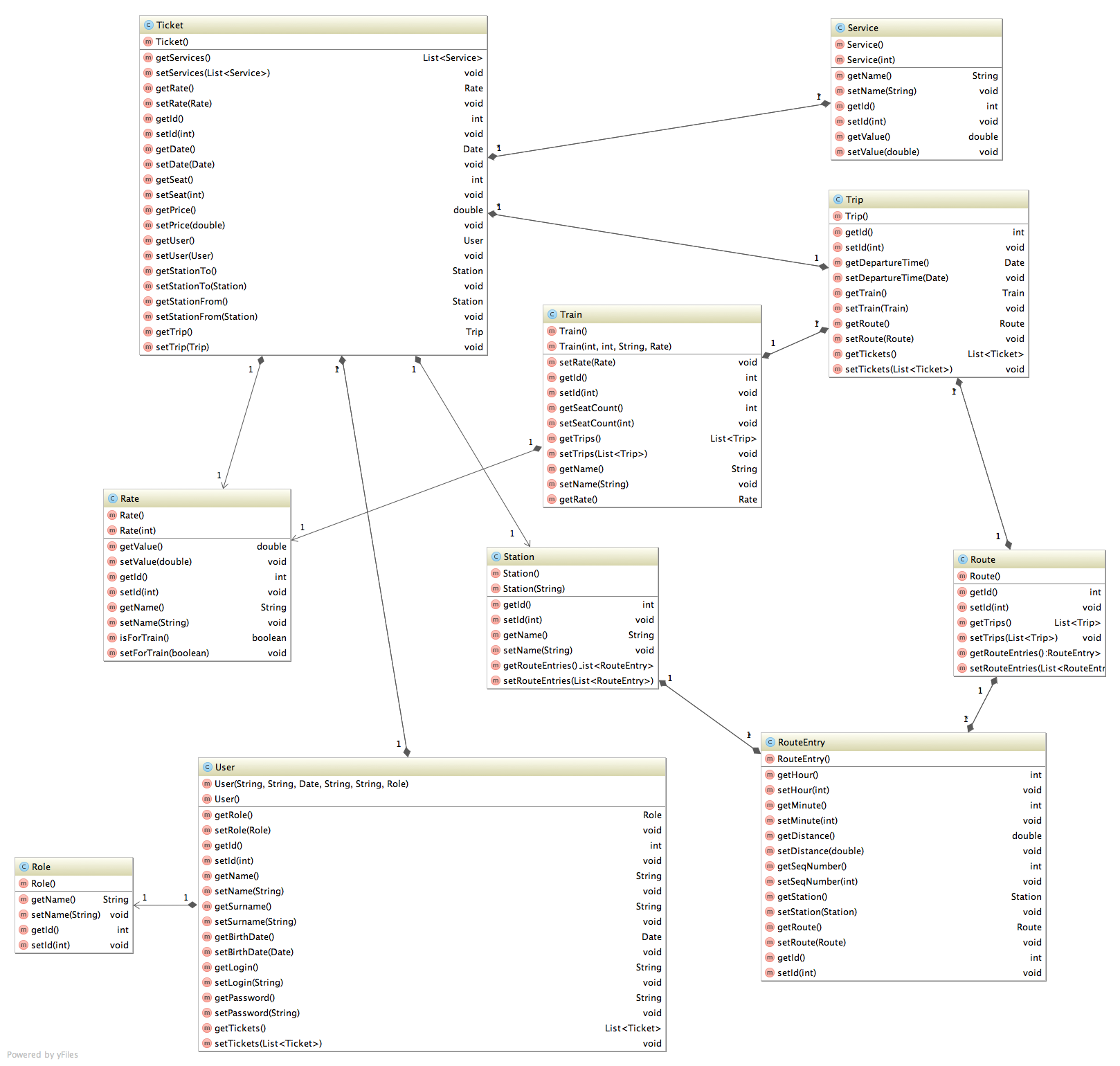


Figure 3. Database model

**Service Layer**

Service layer encapsulates the business logic of the application, controlling transactions and managing responses in the implementation of these operations. All classes are marked with @Transactional annotation that means that Spring will manage transactions. To use dependency injection, data access repositories are declared with @Inject annotation.

All services make conversion between two kind of entities – transfer objects that pass into service from client and entities of data access layer for communication with database. Application services are listed on the figure 3.

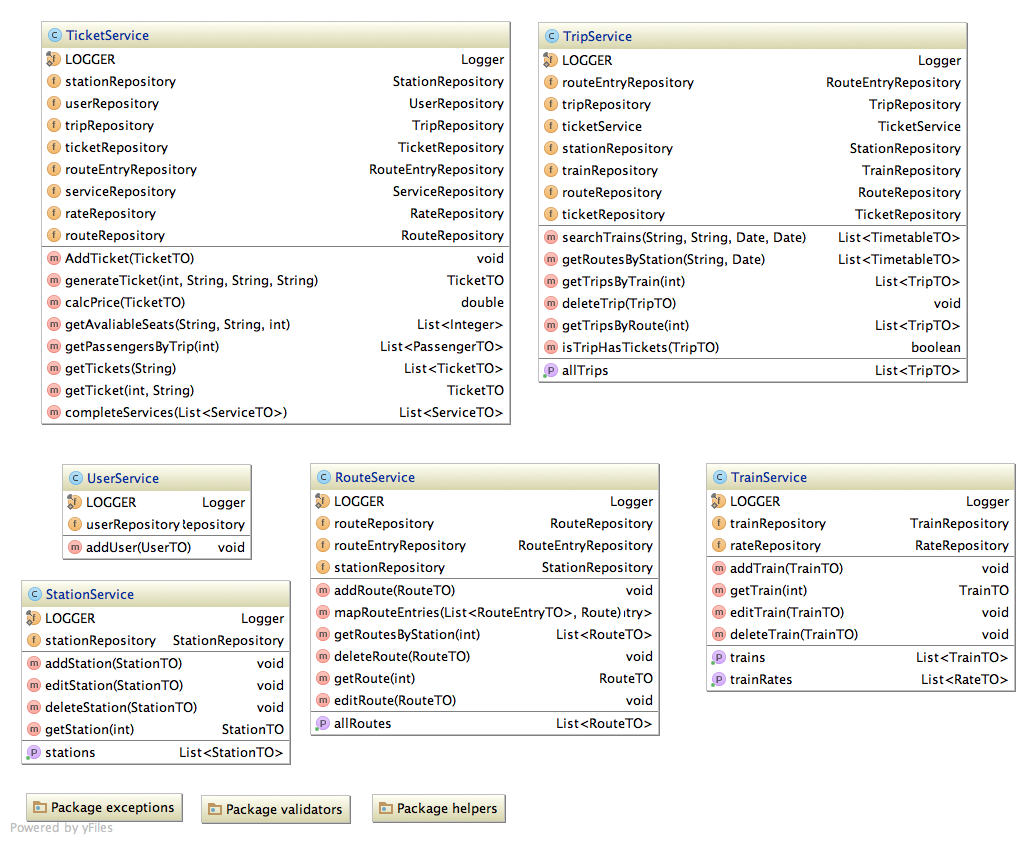


Figure 4. Service layer classes

There is exception handling at the Service layer. Application has following custom exceptions:

* NoAvailableSeats
* RouteNotFoundException
* SeatAlreadyRegisteredException
* StationAlreadyExistException
* StationNotFoundException
* TicketNotFoundException
* TimeConstraintException
* TrainAlreadyExistException
* TrainNotFoundException
* TripDetailsNotFoundException
* TripNotFoundException
* UserAlreadyExistException
* UserAlreadyRegisteredException
* UserNotFoundException

These exceptions are being forwarded to the next level of controllers and being processed together with other possible errors. There is a logging with log4j on this level.

**“UI” Module**

**Controller layer**

Controllers handle incoming requests and return the model with requested data that renders into view and returns as response back to the client. In application this is done with Spring MVC. Configuration of the central Servlet that dispatches requests to controllers and resolves views is presented in dispatcher-servlet.xml.

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"

p:prefix="/WEB-INF/jsp/"

p:suffix=".jsp" />

This lines means that dispatcher-servlet will resolve a string, returned by the controller, into jsp page with location in /WEB-INF/jsp/ subfolder. The dispatcher servlet is an actual servlet declared in the web.xml in the application. Requests are mapped for dispatcher servlet handling, by using a URL mapping in the same web.xml file. The list of controllers is presented on the figure 4 below.

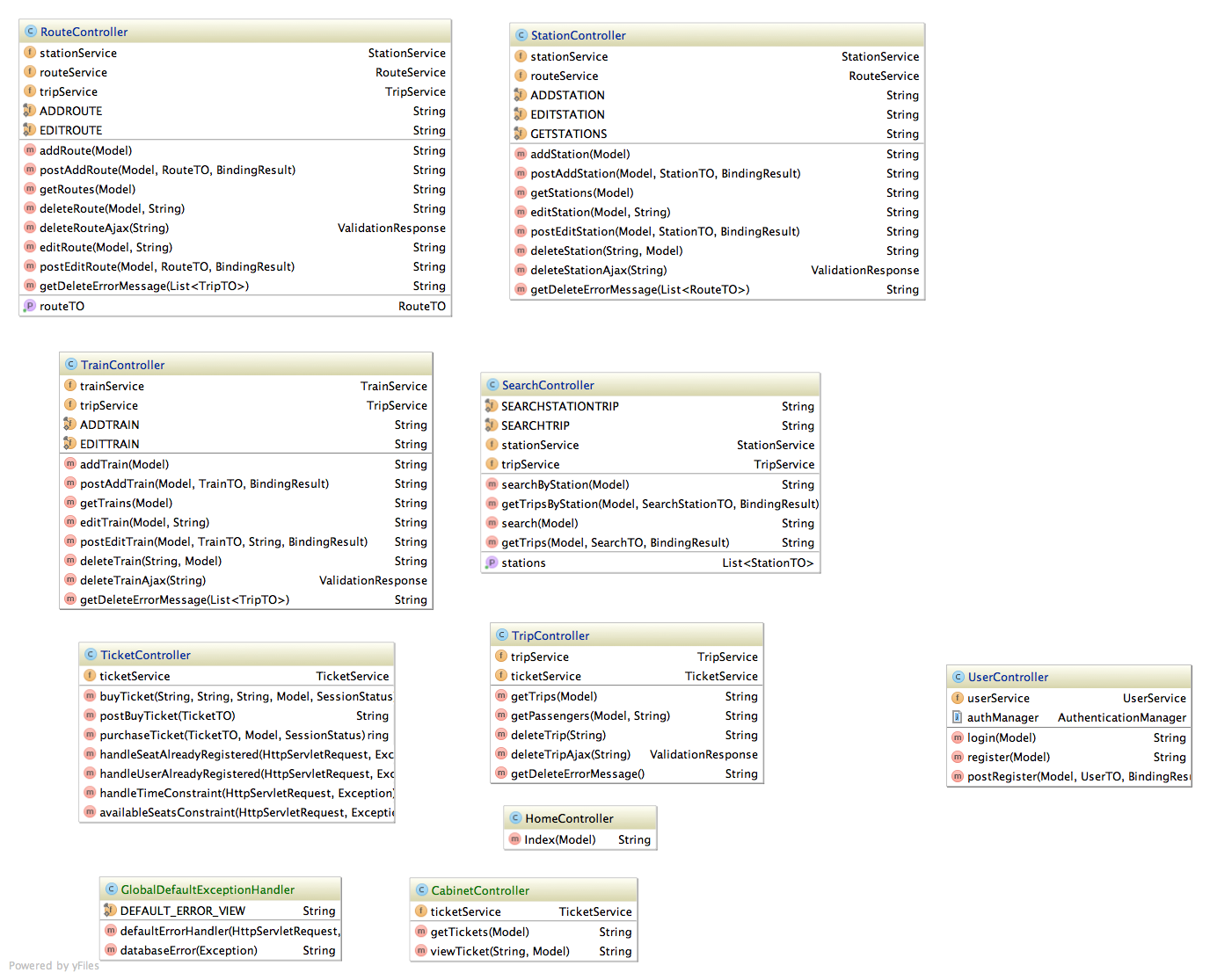


Figure 5. Controller layer

There is an exception handling at controller layer. All possible exceptions are handled with Spring MVC Exception Handling here. Service layer ObjectNotFound exceptions are catched with try/catch construction and throws controller layer PageNotFoundException with response status 404. In this case error pages (404, 400 and 500) are configured in web.xml. Thus, when server returns 404 status, then appropriate error page is sent to client as response. Inappropriate parameter in address bar calls page 400. Unexpected shut down database server calls 500. Server error is handled by global exception handler which implements as GlobalDefaultExceptionHandler class. This class reacts to SQLException and PersistenceException errors by returning specified error page, and to other exceptions by returning common error page with default message.

Service layer can return exceptions that specific only to the TicketController controller. There are controller handlers in these cases. They are marked with @ExceptionHandler annotation.

There are some features when user information maps from jsp page form to specified POJO-class represented by transfer objects from «core» module. Firstly, there is unknown count of adding route points (station, time, distance), when user adds a route. I solved this problem by using Spring AutoPopulatingList object that allows for elements to be automatically populated as they are requested. Secondly, form that represents additional services inforamtion on the page, is different than expected result of mapping. Implementing Converter interface in ServiceConverter class solves this case. It returns appropriate object type list.

**Presentation Layer**

There is a set of jsp pages organized in folders. Folder names are appropriate to controller mapping values. Some jsp pages use expression language custom functions to date, price and services information formatting. They are defined in WEB-INF/functions.tld file.

**Ajax technology**

Deleting of stations, routes, trains and trips is done with Ajax technology. There is JavaScript function in /resources/js/ajax-delete-items.js file. It receives path to controller method, which is a part of the url where ajax request is sent. Controllers delete methods are marked with @ResponseBody annotation that causes return value to be bound to the web response body. Methods return ValidationObject that will be automatically converted to json format. ValidationObject contains status field. If requested object would be deleted from database succesfully , then status field will have «SUCCESS» value, in other case it will have «FAIL» value. There is a check on the client, when controller method is finished. If server returns «SUCCESS» then requested row will be deleted from the page, else – error messages will be displayed.

**Security**

Application access security is provided by Spring Security technologies. There are three kinds of users: anonymous, client and admin. Admins are company employees, they are stored in database and there is no other way to create such kind of user. Clients are created via registration form. Role authorities are represented on picture below.

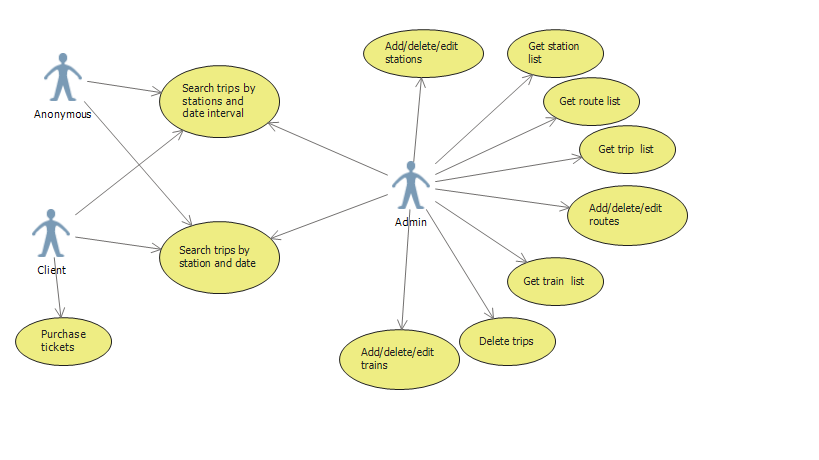


Figure . Roles and authorities

Access to different addresses is depended on user role. Configuration of constrains for this addresses are represented in /WEB-INF/security.xml.

I use such user information like first name, surname and birth date in the application. To have this opportunity I adapt my User object to the Spring Security UserDetails interface and UserService to UserDetailsService and specify link on new UserDetailsService in Spring configuration file. These two classes-adaptors are located in ru.tsystems.tsproject.sbb.secure.adapters package. Security is also provided in the service layer. @PreAuthorize annotation sets constraint on permission to execute these methods.

Content of pages also depends on user role. There is special library, which provides role check functions on jsp page.

**Validation**

I use JSR-303 validation to check correctness of client data in transfer object fields. Each field that uses to get users data is marked with needed annotations. I have also custom checks. They are implement ConstraintValidator interface, and are represented as annotations. Custom validation annotations are located in ru.tsystems.tsproject.sbb.services.validators package.