Technical project design

eCare

By German Urikh

October — December 2014

1. Introduction

1.1 Task

The task given was to implement an application that emulates the work of the mobile operator. The application should have the following entities:

- Tariff
  - Name
  - Price
  - List of possible options
- Option
  - Name
  - Price
  - Initial price (the cost paid immediately after attaching the option to the tariff)
- Client
  - Name
  - Surname
  - Birthday
  - Passport data
  - Address
  - Numbers of contracts
  - E-mail
  - Password to the account control panel
- Contract
  - Number
  - Tariff
  - List of selected options

The clients of the company should be able to do the following:

- view the contract in the account control panel;
- view all possible tariffs to switch their contract to;
- view all possible options to select to their tariff, change the existing options;
- block/unblock their contract (if the contract is blocked, no changes with it should be possible; if it is blocked by the employee, the client should not be able to unblock it).

The employees of the company should be able to do the following:

- create a contract with a new client: to select a new contracts number with a tariff and options. The number should be unique;
- view all the clients and the contracts;
- block/unblock the client;

- find a user by contracts number;
- change the existing contract, add new options to it and delete the old ones;
- add new tariffs and delete the old ones;
- add/delete options for the tariff;
- control the options. Some options can only go together with other ones, and some are incompatible with each other – the employee adds and deletes these rules.

The cart with selected changes to the contract should be available while the user is making changes to it.

1.2 Instruments used

I used the following instruments during my work:

- Intellij IDEA Ultimate 13.1.5;
- jdk 1.7.0_67;
- Apache Maven 3.0.4;
- Wildfly 8.1.0;
- MySQL 5.5.38;
- MySQL Workbench 5.2.38
- SonarQube 4.5.1 with sonar-runner 2.4.

The framework selected was Spring Framework.

2. Implementation

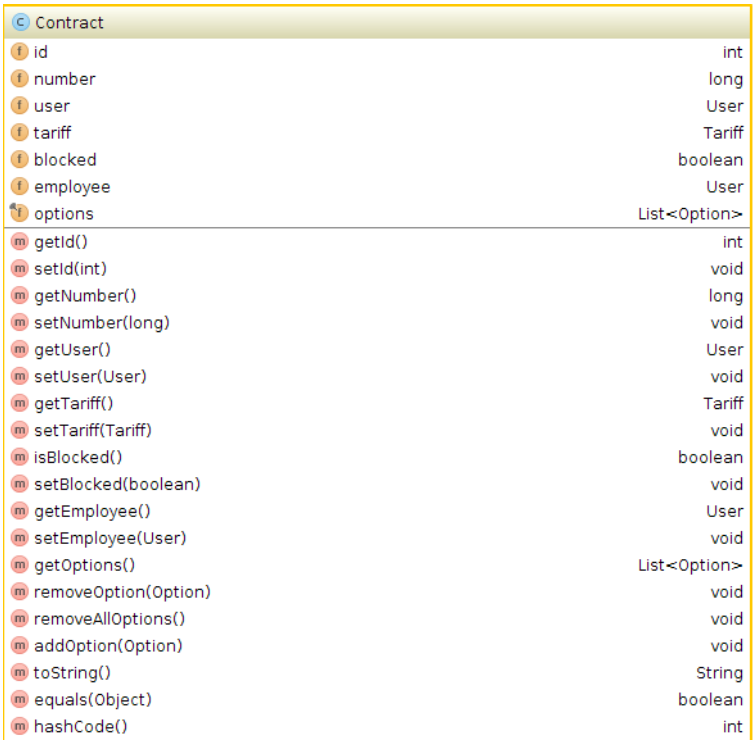
2.1 Entities

My application has 5 entities, defined in the corresponding classes:

- Contract;
- Option;
- Role;
- Tariff;
- User.

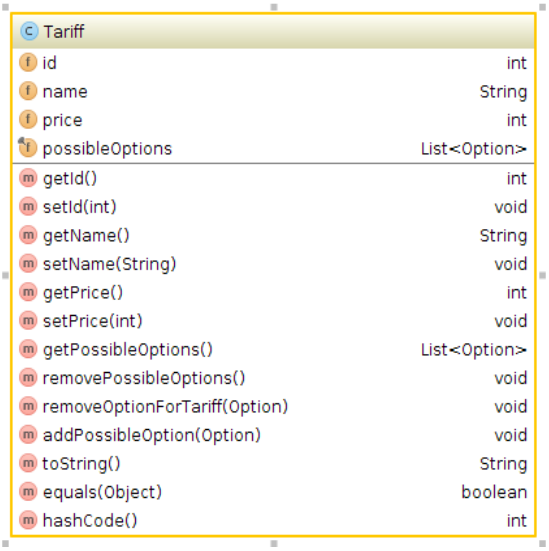
The fields and methods are illustrated below.

Contract

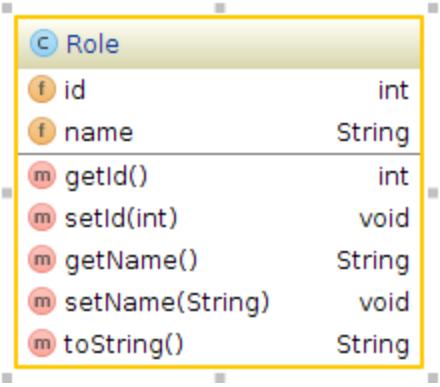


Option

Tariff

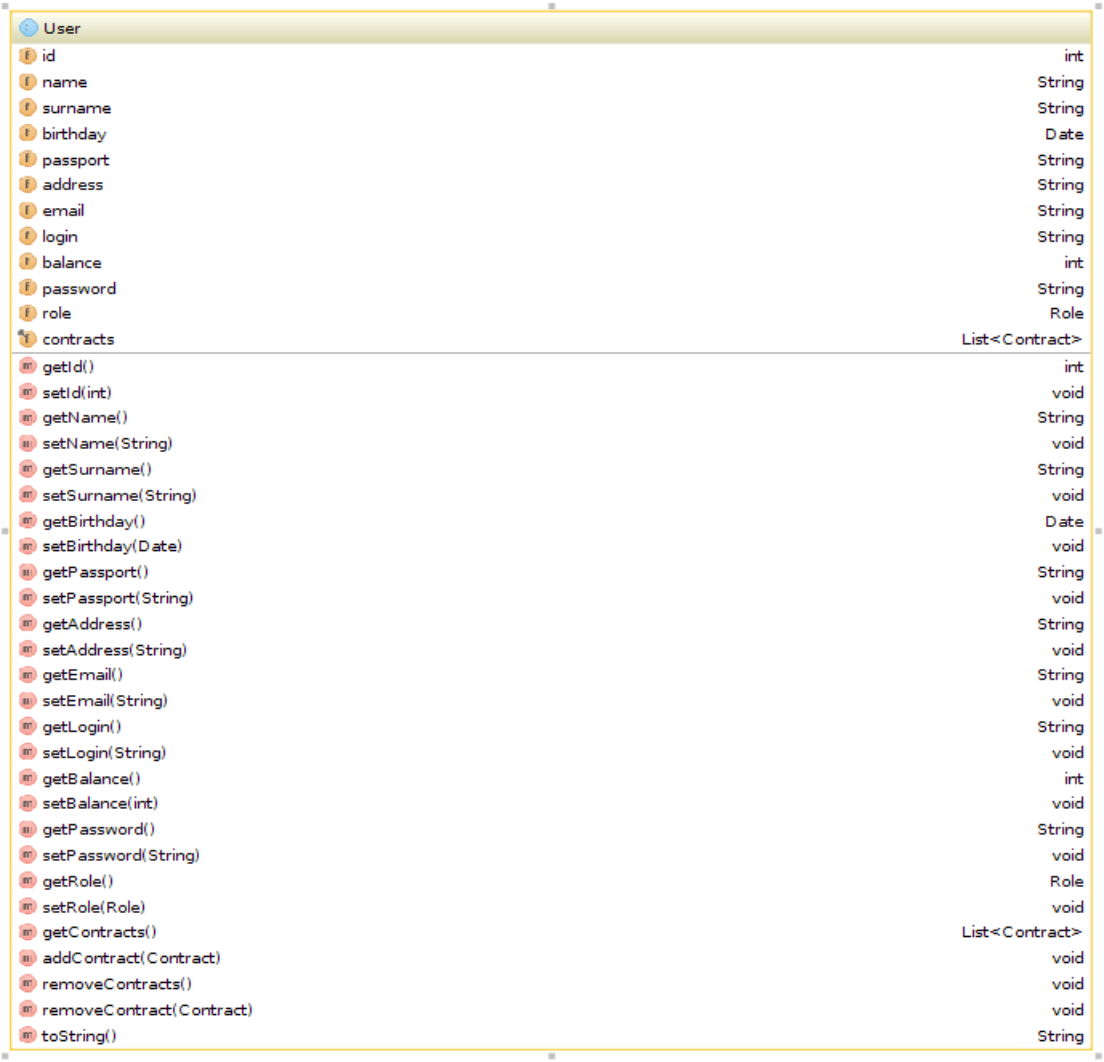


Role

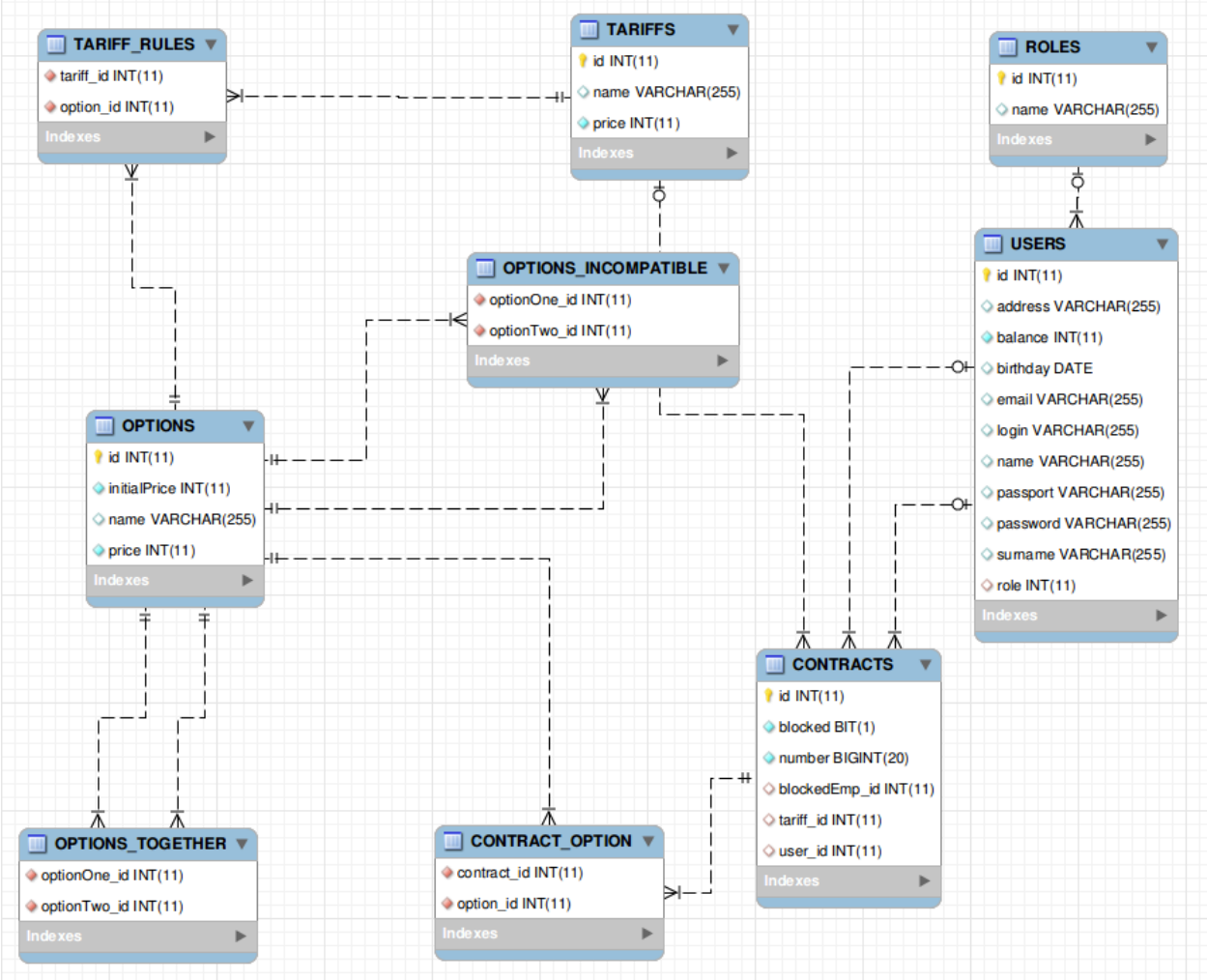


The Role class is defined so that we had the roles described in the task. The roles are created before using the application, with ID 1 being the User and 2 being the employee.

User



The entities' relation can be seen in the following diagram created in MySQL Workbench.



All the tables have been generated automatically by Hibernate due to proper @Entity and @Table naming of the entities. The tables CONTRACT_OPTION, TARIFF_RULES, OPTIONS_INCOMPATIBLE and OPTIONS_TOGETHER were generated to hold the lists of related entities - a list of options attached for the contract, a list of options that are possible to attach to a tariff, a list of options that do not go together with a specified option and a list of options that should go together with an option correspondingly.

2.2 Architecture

The application has 5 layers defining its architecture:

1. DAO layer
2. Service layer implemented to work with DAO and to hold the exception processing logic.
3. Business logic layer which holds the sophisticated logic of, for example, validating the options selected for a tariff or making necessary changes while deleting an entity.
4. Controller layer that works with the parameters received from the UI and dispatches the queries.

5. View layer which consists of JSP-pages, scripts, fonts, styles and images.

2.2.1 DAO

DAO layer, in its term, consists of API package and implementation package.

2.2.1.1 API

There is a generic API interface holding the 5 main operations: CRUD and the getAllEntities one. All other DAO API interfaces (one for each of the entity) implement this generic API as well as holding their specific methods. They are as follows:

1. ContractDAO
    a. *public Contract getContractByNumber(long number) throws ContractNotFoundException;*
    b. public List<Contract> getAllContractsForUser(int id) throws ContractsForEntityNotGotException.
2. OptionDAO
    a. public List<Option> getAllOptionsForTariff(int id) throws OptionsForEntityNotGotException;
    b. public List<Option> getAllOptionsForContract(int id) throws OptionsForEntityNotGotException;
    c. public List<Option> getAllOptionsTogetherForOption(int id) throws OptionsForEntityNotGotException;
    d. *public List<Option> getAllOptionsIncompatibleForOption(int id) throws OptionsForEntityNotGotException*.
3. RoleDAO doesn't have any specific methods.
4. TariffDAO also doesn't have any specific methods.
5. UserDAO
    a. public User getUserByNumber(long number) throws UserNotFoundException;
    b. public User getUserByLogin(String login) throws UserNotFoundException.

All the methods above are self-documentary and do exactly what they are thought to do.

2.2.1.2 Implementation

There is an implementation with the realization of the methods for each API interface.
The realisation of the CRUD methods is done using the JPA + entitymanager in the GenericImplementation class, while the methods specified above are realized in the corresponding implementation classes.
Entitymanager is being injected by the @Autowiring annotation using Spring DI.
Each of the DAO implementation classes is annotated with a @Repository annotation so that Spring didn't try to work with exceptions on this layer and threw it forward to service classes.

2.2.2 Services

Service layer also consists of an API package and the implementation package.

2.2.2.1 API

There is a generic API for services holding the CRUD methods and the getAllEntities one. Other API service interfaces implement this one and hold their specific methods which are exactly the ones as in DAO layer.

2.2.2.2 Implementation

There is a realisation for each of the service API interfaces in the corresponding implementation class.
Each of the classes is annotated with the @Service annotation.
The implementation is made using the DAO API, with DAO implementation being injected by Spring DI with the help of an @Autowired annotation.
The methods are annotated with @Transactional annotation so that we didn't bother about committing or rollbacking the transactions - it will be done automatically by Spring.

2.2.3 Business logic layer

The business logic layer is represented in the ru.tsystems.tproject.integration package and it consists of 3 classes:

- ContractValidator
- EntityRemoval
- UserUpdater

2.2.3.1 ContractValidator

This class holds the methods that are carried out during the changing of the contract.

- *public List validateOptions(int[] array, List<Exception> exceptionsList, int userId)* - this method checks if the entered options are correct and they can go with each other and without other options;
- *public int balanceCheck(int userId, List<Option> optionList)* - this method checks the balance of the client that will remain after adding the options to the tariff;
- *public void priceCheck(int price, String priceName) throws IOException* - this method checks if the price is higher than the maximum price set in our logic.

2.2.3.2 EntityRemoval

This class holds the logic to be applied while removing a tariff or an option.
When a tariff is removed, all the users that have this tariff in their contract get it switched to the one with id BASE_TARIFF, which is defined in this class. The compensation in the amount of TARIFF_COMPENSATION is added to their balance.
When an option is removed, all the users that have it attached to their balance get the compensation added to their balance in the amount of OPTION_COMPENSATION.

2.2.3.3. UserUpdater

This class holds the logic applied to the user's creation or when it is updated. It sets the fields which are null as a result of no entered data.

2.2.4 Controllers

This layer holds the classes which are responsible for the interaction between the UI and the service classes. The controllers are based on the Spring MVC framework, have the @Controller annotation and have the services @Autowired. Those are:

- LoginController - a controller that dispatches the queries when the user is not logged in or when he has just logged (to the main page);
- EmployeeController - a controller to dispatch the queries to the employee pages;
- ClientController - a controller to dispatch the queries to the client pages;
- AjaxDispatcher - a controller to hold the methods that respond to the AJAX queries;
- GlobalExceptionHandler - an exception handler to navigate when an exception occurs.

The responsible spring context is webapp/WEB-INF/spring/mvcDispatcher-servlet.xml

2.2.5 Views

This layer is responsible for UI. It is based on the webapp folder with the following subfolders which are self-explanatory:

- css;
- fonts;
- images;
- scripts;
- WEB-INF/views.

2.3 Other features

Not mentioned above, there are several other features in the application worth mentioning.

2.3.1 Logging

Each step made on the UI is logged. The logging in my application is implemented in two classes: MainAspect in the package ru.tsystems.tproject.aspect and RequestInterceptor in ru.tsystems.tproject.utils.

2.3.1.1 MainAspect

This class has 4 poincuts:

- execution of get* method;
- execution of create* method;
- execution of update* method;
- execution of delete* method.

All together they intercept all the queries to the database.
The method called and the arguments passed are logged before each query with the @Before annotation; and the result of fulfilling the method is also logged with the @After annotation
This class also has the @AfterThrowing annotation which helps us log in details the exception if one occurs during the interaction with the database

2.3.1.2 RequestInterceptor

This class is responsible for intercepting all the queries to get different views on the UI level. It is also based on AspectJ, but it is implemented by Spring.
Three methods are overriden: what happens before the request (preHandle), what happens after starting answering to the request (postHandle) and what happens after completing the request (afterCompletion). Together they log the user's login that has sent the request, the url that has been asked for, the url sent as an answer and the time spent on dispatching the request.

2.3.2 Security

The security in the application (the authentification and the authorisation) is handled by Spring Security. The responsible context is webapp/WEB-INF/spring/spring-security.xml
The security is based on the roles got from the database from the ROLE table. The necessary logic for receiving the necessary data is in ru.tsystems.tproject.security package in AppUserDetailsServiceDAO class.
The main features are the following:

- neither the client, nor the employee have access to any pages but for login and 403 pages if they are not authorised;
- the client does not have access to employee pages;
- the employee does not have access to client pages;
- the maximum number of sessions of one user is 1;
- the time a user is remembered if "remember me" is checked on the login is 1200 seconds;
- the password is encoded with the help of custom converter in ru.tsystems.tproject.utils.Converter class implementing the Spring BasePasswordEncoder class. This converter uses the base MD5 algorythm, but it also addes two words in front of and in the end of the password. The result is stored in the database as a password to be checked while logging.

2.3.3 Localisation

The client's account control panel has two languages to choose from - English and Russian. The logic responsible for setting an object containing the locale info is in LoginController, while the implementation of such an object is in the package ru.tsystems.tproject.utils.locale.

The interface Translatable has to be implemented while creating a new locale class, and all the getter methods have to be overriden with proper values returned. As a result, an object containing the fields with necessary locale is set in the session and got in the JSP page.

2.3.4 Custom exceptions

The exceptions are detalised where possible so that the name of exception could tell a lot. A CustomDaoException is an exception to be caught, while all other custom exceptions extend this one. They are self-explanatory and are held in ru.tsystems.tproject.exceptions package:

- ContractNotFoundException;
- ContractsForEntityNotGotException;
- EntityNotDeletedException;
- OptionsForEntityNotGotException;
- ScriptViolationException;
- UserNotFoundException.

2.3.5 Other features

The constants with the view addresses for the controller classes are held in the specific classes and are statically imported. They are in ru.tsystems.tproject.utils.pages:

- ClientPages;
- EmployeePages;
- SharedPages.

3. Quality control

The quality of the application is controlled with two powerful instruments:

- JUnit tests;
- SonarQube application.

3.1 JUnit

3.1.1 Dependency injection

A simple dependency injection test is run in the integration.injection package to check that our @Autowired works in a correct way.

3.1.2 Services

All the services are tested in the corresponding test classes. The main feature of these tests is that:

- SQL-script is executed before each test method so that we had the data to work with;
- another SQL-script is executed after each test method so that the database remained in the same condition as it was before the tests.

This is done intentionally with no mocking as these tests are almost integrational as they have autowiring of DAO dependencies. We do check the correct work with the database here.

3.1.3 Business logic

Our integration package contains the classes that test the classes described as the business-logic holders. Here we do not work with the database, as this is not what we need to test, so Mockito is widely used there.

3.1.4 Controllers

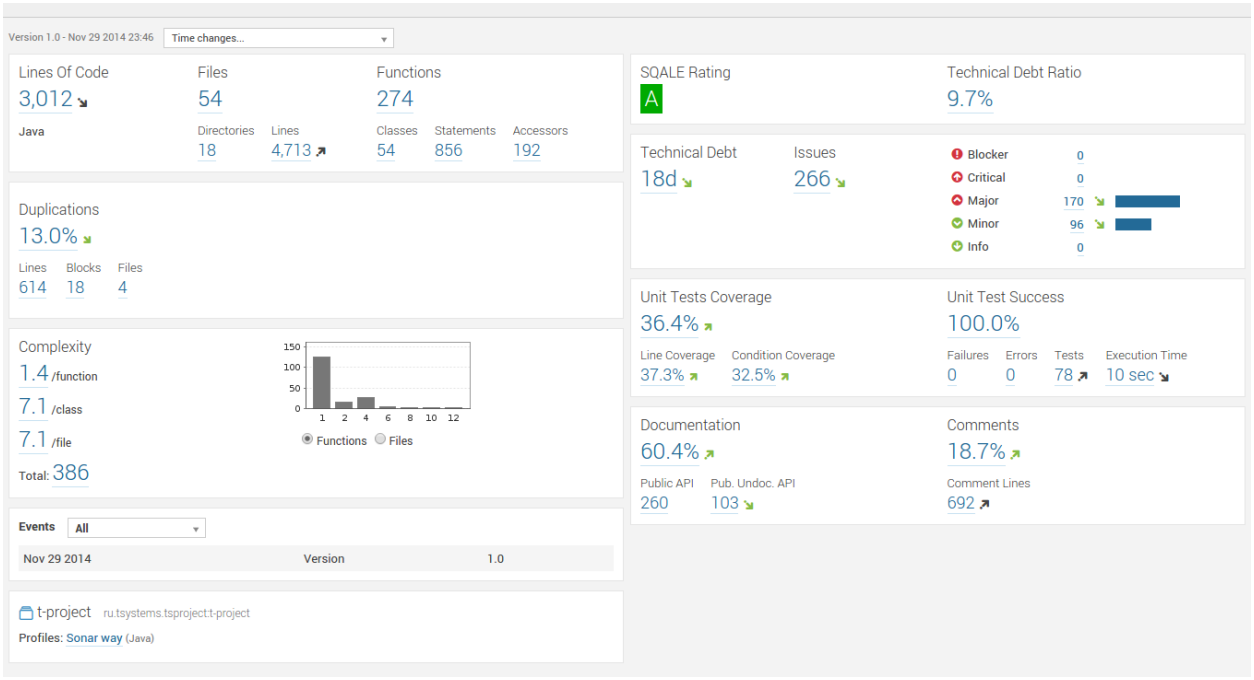
The controllers package has the tests to check the navigation between the pages - that it goes as expected.

3.1.5 Utils

The utils package has some other tests to check the correct working of other utils such as Converter class and Parser class.

3.2 SonarQube

The application was analysed by SonarQube - a static code quality analyzer. The results are below.



We can note the following:

- The overall SQALE Rating is A. This is the best possible score. It shows that the time needed to solve the technical debt is relatively very low to the time already spent;
- The Unit Tests Coverage is 36,4%. It could be higher but the Spring controller methods are not very simple to test while being at the same time reliable. It is now low, though.
- The documentation percent is 60,4%. Every method in the application is documented, so this rate is almost the highest possibly approachable.
- There are no critical or blocker errors, and the remaining major ones are not the ones urgent to fix - they are not errors at all.

To sum up, the quality of the application can be estimated as high.