**4.** ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В текущем разделе будут рассмотрены более подробно функциональные блоки, из которых состоит разрабатываемое программное средство. Так же будут рассмотрены спасобы реализации функциональных блоков.

4.1 Модуль аутентификации

Данный модуль реализуется с использованием модуля Spring Security фреймворка Spring. Для реализации модуля необходимо всего три класса реализующих логику: *DefaultUserDetailsService*, *SecurityConfiguration, DatabaseUserDao*. И один класс сущности: *User*.

Класс *DefaultUserDetailsService* является реализацией интерфейса *UserDetailsService* модуля Spring Security и предоставляет метод *loadUserByUsername,* который принимает имя пользователя и возвращает объект *User* в случае успешной аутентификации или выбрасывает исключение *UserNotFoundException* если пользователя с таким именем не существует. Данный метод в свою очередь обращается к методу *getUserByUsername* класса *DatabaseUserDao*, который возвращает объект *User*, если пользователь найден или *null* в обратном случае.

Класс *SecurityConfiguration* является местом для настройки процесса аутентификации. Класс унаследован от класса *WebSecurityConfigurerAdapter* и переопределяет метод *configure,* который принимает объект *HttpSecurity* для настройки процесса аутентификации. В листинге 4.1 приведена настройка аутентификации. Здесь указывается, что при *post*-запросе на url */login* будет осуществлятся аутентификация пользователя. Параметр запроса с именем *username* содержит в себе имя пользователя, а параметр *password* содержит пароль пользователя. При успешной аутентификации кленту придет ответ с кодом *202* (принято), иначе ответ с кодом *401* (не авторизован)*.*

*http.csrf().disable().formLogin().loginPage("/login"). usernameParameter("username").passwordParameter("password").failureHandler(authenticationFailureHandler()).successHandler(authenticationSuccessHandler())*

Листинг 4.1 ­- Настройка аутентификации

Второй метод класса *SecurityConfiguration configure* принимает объект типа *AuthenticationManagerBuilder.* В этом методе указывается *UserDetailsService,* который будет использован при аутентификации, так же указывается алгоритм хэширования пароля. В листинге 4.2 приведен данный метод.

protected void configure(AuthenticationManagerBuilder auth) throws Exception { auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());

}

Листинг 4.2 ­- Метод *configure.*

Класс *SecurityConfiguration* помечен аннотацией *@Configuration*, тем самым указывается что данный класс является классом для настройки безопасности приложения. На рисунке 4.1 представлена схема реализации механизма аутентификации в Spring Security.



Рисунок 4.1 – Схема работы аутентификации Spring Security

Так же в данном модуле реализуется так называемая функция «remember me», которая позволяет пользователю не авторизовываться каждый раз при входе в приложение, посредством сохранения ключа авторизации в cookies пользовательского браузера.

Весь вышеизложенный функционал реализован с использованием Spring Security. Данный фреймворк необходимо включить в зависимости Maven.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

Помимо аутентификации пользователей данный модуль выполняет следующие функции: регистрация пользователей, редактирование профиля пользователей, выход из системы (log out) и удаление пользователя. Для реализации данного функционала используются классы классы: *AuthenticationController*, *DefaultUserDetailsService* и *DatabaseUserDao*.

Класс *UserDetailsService* был рассмотрен выше, однако с точки зрения реализации интерфейса *UserDetailsService.* Здесь этот класс рассматривается как реализация интерфейса *UserService.*

Класс *AuthenticationController* содержит в себе методы: *logout*, *update*, *delete.* Метод *logout* удаляет текущего пользователя из сессии и закрывает саму сессию.

Метод *update* принимает объект класса *User*, валидирует его и передает методу *updateUser* класса  *DefaultUserDetailsService.* После этот объект будет передан методу *udpateUser* класса *DatabaseUserDao* и будет произведено обновление записи о пользователе в базе данных.

Метод *delete* принимает логин пользователя, далее вызывается метод *deleteUser* класса DefaultUserService и метод *deleteUser* класса *DatabaseUserDao.*

**4.2** Модуль отправки уведомлений

Данный модуль предназначен для отправки уведомлений ползователям по электронной почте. Непосредственным отправителем письма будет являтся класс *MailSender*, с единственным методом *send,* который принимает три параметра: электронный адрес получателя, тему и тело сообщения. Так же в отдельный файл должны быть вынесены следующие свойства: smtp-хост, номер порта, электронныйй адрес и пароль отправителя сообщений.

Данный модуль реализуется с помощью библиотеки JavaMail, которая подключается с помощью добавления зависимости в файл *pom.xml* (см. листинг 4.3).

<dependency>

    <groupId>javax.mail</groupId>

    <artifactId>mail</artifactId>

    <version>1.4</version>

</dependency>

Листинг 4.3 – Подключение JavaMail

JavaMail позволяет производить отправку сообщений по следующим протоколам: SMTP, POP, IMAP, MIME.



Рисунок 4.2 – Схема работы библиоткеки JavaMail

На рисунке 4.2 приведена схема работы библиотеки JavaMail. Здесь класс *Session* является базовым представлением сессии и отвечает за соединение с smtp-сервером. Класс *Message* представляет сообщение, которое может быть принято или отправлено, является абстрактным классом. Класс *Address* представляет собой адрес получателя или отправителя сообщения, являетсяя абстрактным классом. Класс *Transport* отвечает за работу по выбранному протоколу. Модуль *Store* и *Folder* создают сессию, соединяются с хранилищем в соответствии с указанными именем пользователя и паролем.

**4.3** Модуль построения отчетов

Данный модуль предназначен для получения отчета о проделанной работе по проекту. Суть построения отчета заключается в том, что пользователь получает информацию о задачах находящихся в том или ином статусе.

Для реализации функционала необходимо выделить класс *ReportController*. Все методы класса возвращают список задач выбранных по тому или иному критерию.

Метод класса *ReportController getTasks* принимает параметр *taskStatus*, таким образом метод возвращает список объектов класса *Task* с определенным статусом. Метод *getHistoryInfo* возвращает список объектов класса *TaskHistory.* В конечном итоге пользователь сможет увидеть изменения касающиеся задач, которые были произведены на проекте. Так же имеет смысл добавить перегруженную версию метода *getHistoryInfo,* которая принимает параметры *endDate* и *startDate*, что позволяет получить историю изменения задач за определенный период. Метод *getFullStatastic* позволят узнать сколько задач находится в каждом из статусов.

Методы с аналогичными названиями присутствуют в классах *DefaultReportService* (класс уровня бизнес логики) и *ReportDao* (класс уровня доступа к данным).

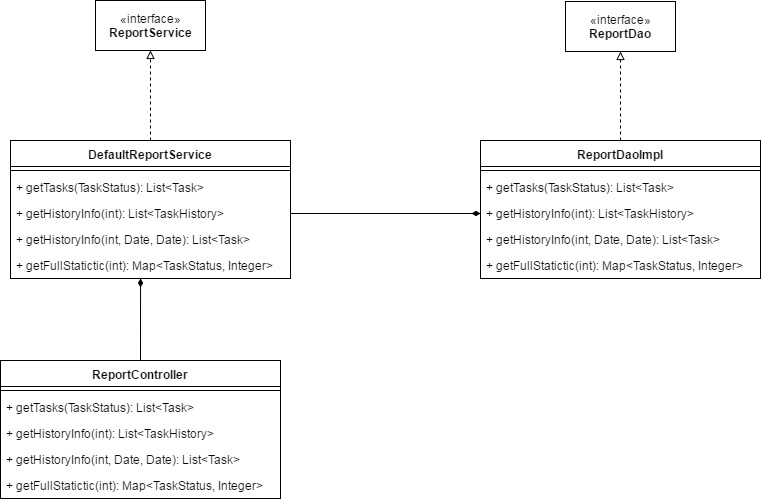


Рисунок 4.3 – Модель классов модуля построения отчетов

Отдельно выделяется класс *TimeTrackController.* Этот класс предназначен для отчета пользователя о проделанной работе. Метод *logWork* принимает экземпляр класса *Work* который представляет собой отчет о проделанной работе. Метод *getLoggedWork* принимает имя пользователя и позволяет получить отчет о проделанной работе для конкретного пользователя. Перегруженная версия этого метода принимает *endDate* и *startDate*, что позволяет получить отчет за определенный временной отрезок.

Методы с аналогичными названиями присутствуют в классах *DefaultLogWorkService* и *LogWorkDao.*