**3.** ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

Рассмотрим подробнее функциональные блоки, из которых состоит разрабатываемое программное средство, а также способы их реализации.

**3.1** Модуль аутентификации

Модуль аутентификации необходим для входа пользователей в систему. Аутентификация производится по логину и паролю, введенными пользователем. Так же данный модуль занимается управлением пользователями системы: регистрацией, обновлением профиля, удалением пользователей.

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 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.

Для настройки процесса аутентификации разработан класс *SecurityConfiguration,* которыйявляется местом для настройки процесса аутентификации. Класс унаследован от класса *WebSecurityConfigurerAdapter* и переопределяет метод *configure,* который принимает объект *HttpSecurity* для настройки процесса аутентификации. В листинге 3.1 приведена настройка аутентификации. Здесь указывается, что при *post*-запросе на url */login* будет осуществляться аутентификация пользователя. Параметр запроса с именем *username* содержит в себе имя пользователя, а параметр *password* содержит пароль пользователя. При успешной аутентификации клиенту 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());

Листинг 3.1 – 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 3.2 приведен данный метод.

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());

}

Листинг 3.2 – Метод *configure*

Класс *SecurityConfiguration* помечен аннотацией *@Configuration*, тем самым указывается что данный класс является классом для настройки безопасности приложения. На рисунке 3.1 представлена схема реализации механизма аутентификации в Spring Security.



Рисунок 3.1 – 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. Чтобы этого достичь ключ авторизации сохраняется в cookies пользовательского браузера.

Весь вышеизложенный функционал реализован с использованием Spring Security. Данный фреймворк необходимо включить в зависимости Maven (см. листинг 3.3).

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*.

<dependency>

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

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

</dependency>

Листинг 3.3 – Подключение Spring Security в Maven

Класс *DefaultUserDetailsService* 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 в метод *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.*

Диаграмма классов настоящего модуля представлена на рисунке 3.2.

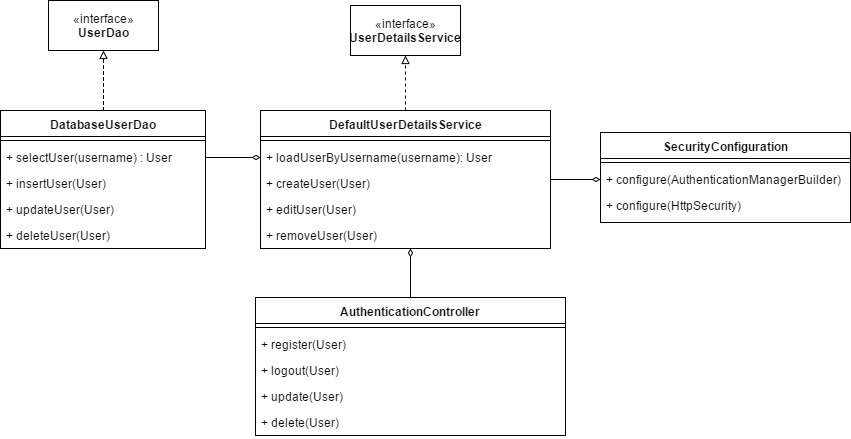


Рисунок 3.2 – Диаграмма классов модуля аутентификации

**3.2** Модуль отправки уведомлений

Данный модуль предназначен для отправки уведомлений пользователям по электронной почте. Класс *EmailNotificationService* реализует интерфейс *NotificationService.* В классе переопределены следующие методы: *send* (метод принимает объект класса *EmailMessage* и производит отправку сообщения по электронной почте), *sendUserRegistration* (метод принимает объект класса *User* и отправляет уведомление пользователю об успешной регистрации), *sendAddedToProjectMessage* (метод принимает объекты классов *Project* и *User* и отправляет пользователю уведомление, о том, что он был добавлен к проекту).

Так же в отдельный файл должны быть вынесены следующие свойства: smtp-хост, номер порта, электронныйй адрес и пароль отправителя сообщений. JavaMail позволяет производить отправку сообщений по следующим протоколам: SMTP, POP, IMAP, MIME.

Данный модуль реализуется с помощью библиотеки JavaMail, которая подключается с помощью добавления зависимости в файл *pom.xml* (см. листинг 3.4).

<dependency>

     <groupId>javax.mail</groupId>

     <artifactId>mail</artifactId>

     <version>1.4</version>

</dependency>

Листинг 3.4 – Подключение JavaMail



Рисунок 3.3 – Схема работы JavaMail

На рисунке 3.3 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. На рисунке 3.4 представлена диаграмма классов данного модуля.

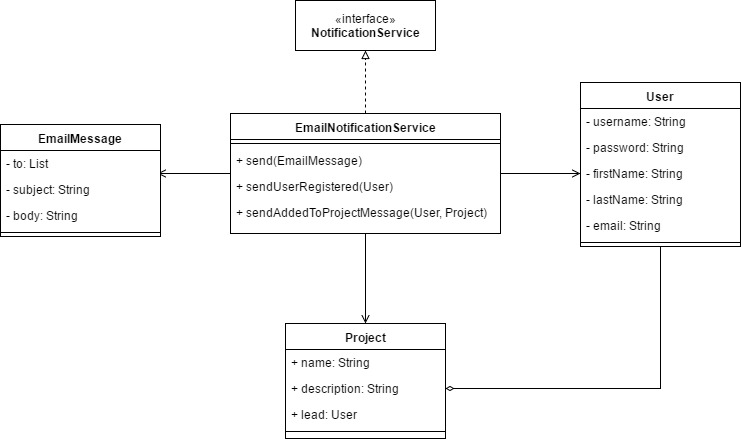
****

Рисунок 3.4 – Диаграмма классов модуля отправки уведомлений

**3.3** 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*, и возвращает список объектов класса *Task* с этим 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 классе *DefaultReportService.* На рисунке 3.5 представлена диаграмма классов модуля построения отчетов.

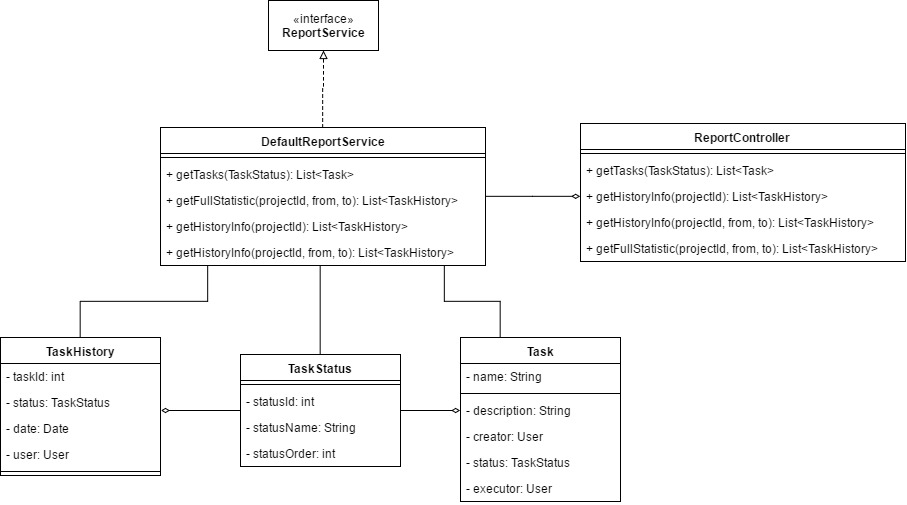


Рисунок 3.5 – Диаграмма классов модуля построения отчетов

Отдельно выделяется класс *TimeTrackController.* Этот класс предназначен для отчета пользователя о проделанной работе. Метод *logWork* принимает экземпляр класса *WorkItem* 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. На рисунке 3.6 представлена диаграмма классов модуля отправки отчетов.

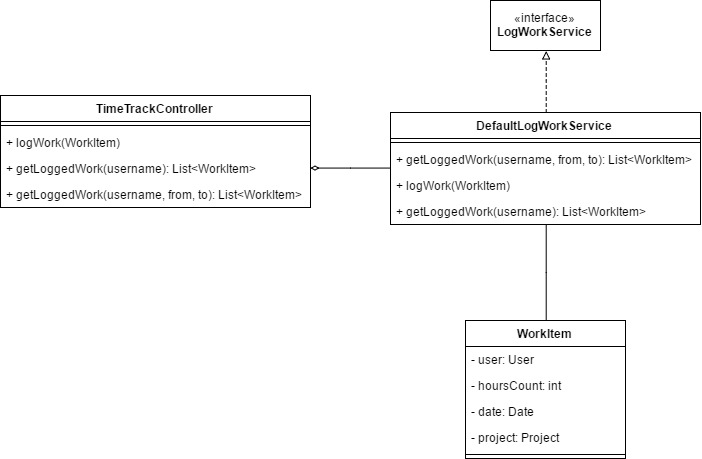


Рисунок 3.6 – Диаграмма классов модуля отправки отчетов

**3.4** Модуль доступа к данным

Модуль доступа к данным является своего рода адаптером между бизнес логикой приложения и базой данных. В данном модуле можно выделить следующие классы: *DatabaseUserDao*, *DatabaseTaskDao*, *DatabaseProjectDao*, *DatabasePermissionDao*, *DatabaseTaskHistoryDao*, *DatabaseLogWorkDao*, *DatabaseTaskStatusDao*, *DatabaseGroupDao.*

Класс *DatabaseUserDao* является точкой доступа к данным связанных с информацией о пользователе. Данный класс реализует следующие методы: *getUserByUsername* (метод принимает строку с именем пользователя и возвращает объект *User*), *updateUser* (метод принимает экземпляр класса *User* и обновляет информацию о нём в базе данных), метод *createUser* (принимает экземпляр класса *User* и создаёт новую запись о пользователе в базе данных), *deleteUser* (принимает экземпляр класса User и удаляет соответствующую запись из базы данных).

Класс *DatabaseGroupDao* предоставляет интерфейс для доступа к данным, касающихся проектной группы. Класс реализует следующие методы: *createGroup* (метод принимает экземпляр класса *Group* и создает новую группу пользователей)*, addUserToGroup* (метод принимает идентификатор группы и имя пользователя и производит добавление пользователя в группу проекта)*, removeUserFromGroup* (метод принимет идентификатор группы и имя пользователя и удаляет пользователя из проектной группы)*, deleteGroup* (метод принимает экземпляр класса *Group* и удаляет группу пользователей), *getProjectGroups* (метод принимает идентификатор проекта и возвращает все группы на проекте)*.*  На рисунке 3.7 представлена диаграмма классов *DatabaseUserDao* и *DatabaseGroupDao*.

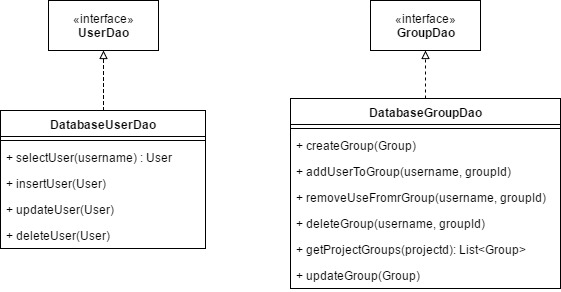


Рисунок 3.7 – Диаграмма классов DatabaseUserDao и DatabaseGroupDao

Класс *DatabaseTaskDao* (см. рис. 3.8) является точкой доступа к данным касающихся задачи. Данный класс реализует следующие методы: *getTasksWithStatus* (метод принимает экземпляр класса *Status* и возвращает список с указанным статусом), *getTask* (метод принимает идентификатор задачи и возвращает экземпляр класса *Task*), *updateTask* (метод принимает экземпляр класса *Task* и обновляет соответствующее поля в базе данных), *deleteTask* (метод принимает экземпляр класса *Task* и удаляет соответствующую задачу из базы данных), *addAttachement* (метод принимает экземпляр класса *Attachement* тем самым создается приложение к задаче), *addComment* (метод принимает экземпляр класса *Comment,* таким образом добавляется комментарий к задаче), *getTaskDetails* (метод принимает идентификатор задачи, и возвращает полную информацию о ней), *getProjectTasks* (метод принимает идентификатор проекта и идентификатор статуса задачи и возвращает задачи соответствующие переданным параметрам).

Класс *DatabaseProjectDao* (см. рис. 3.8)представляет собой точку для доступа к данным касающихся проекта. Данный класс реализует следующие методы методы: *getUsersProjects* (метод принимает экземпляр класса *User* и возвращает список проектов, над которыми работает указанный пользователь), *getProject* (метод принимает идентификатор проекта и возвращает экземпляр класса *Project*), *createProject* (метод принимает экземпляр класса *Project* и создает соответсвующую запись в базе данных), *deleteProject* (метод принимает экземпляр класса *Project* и удаляет проект из базы данных), *getUsersOnProject* (метод принимает идентификатор проекта и возвращает список пользователей участвующих в его разработке), *getLead* (метод принимает идентификатор проекта и возвращает экземпляр класса *User*, который символизирует руководителя проекта).

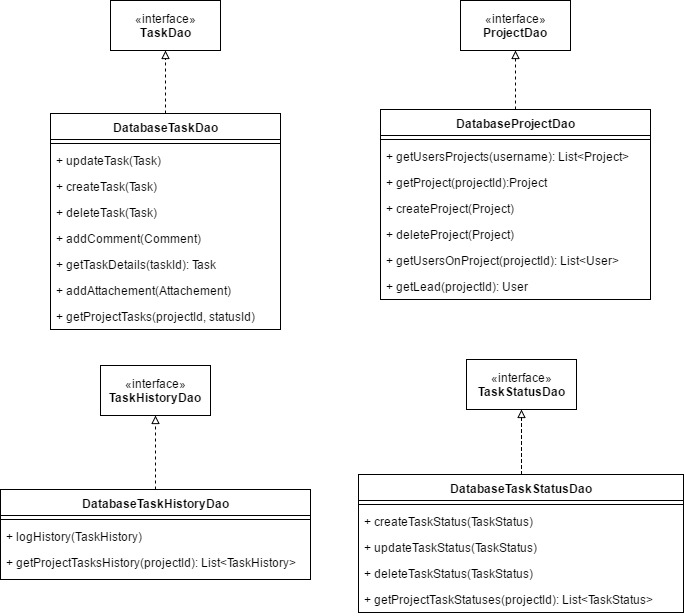
**

Рисунок 3.8 – Диаграмма классов для DatabaseTaskDao, DatabaseProjectDao, DatabaseTaskHistoryDao

Класс *DatabaseTaskHistoryDao* (см. рис. 3.8)предоставляет интерфейс для доступа к данным касающихся истории изменения задачи. Данный класс предоставляет следующие методы: *logHistory* (метод принимет экземпляр класса *History* и занимается созданием новой записи в истории задач), *getProjectTasksHistory* (метод принимает идентификатор проекта и возвращает список изменений задач).

Класс *DatabaseTaskStatusDao* (см. рис. 3.8)предоставляет интерфейс, который позволяет манипулировать данными, которые касаются статуса задачи. Данный класс реализует следующие методы: *createTaskStatus* (метод принимает экземпляр класса *Status* и создает новый статус задачи)*, updateTaskStatus* (метод принимает экземпляр класса *Status* и обновляет информацию о статусе задачи)*, deleteTaskStatus* (метод принимет экземпляр класса *Status* и удаляет статус задачи)*, getProjectTaskStatuses* (метод принимает идентификатор пректа и возвращает список статусов, в которых может находится задача).

Класс *DatabasePermissionDao* (см. рис. 3.9)предоставляет интерфейс по управлению правами пользователей на проекте. Класс реализует следующие методы: *createPermission* (метод принимает экземпляр класса *Permission* и создает новое право в базе данных), *deletePermission* (метод принимает экземпляр класса *Permission* и удаляет соответствующее право из базы), *assignPermissionToGroup* (метод принимает экземпляр класса *Permission* и идентификатор группы, тем самым предоставляя группе пользователей определеноое право), *takeOffPermissionFromGroup* (метод принимает экземпляр класса *Permission* и идентификатор проектной группы, тем самым удаляя права пользователя), *getGroupPermissions* (метод принимает идентификатор группы и возвращает список прав для нее).

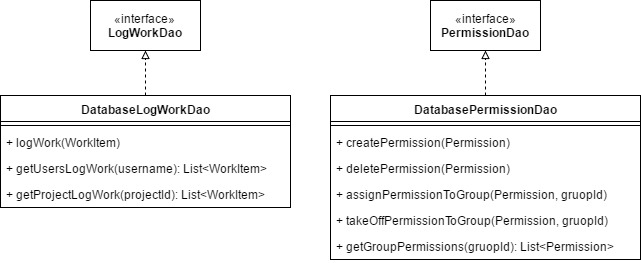


Рисунок 3.9 – Диаграмма классов для DatabaseLogWorkDao и DatabasePermissionDao

Класс *DatabaseLogWorkDao* (см. рис. 3.9)предоставляет интерфейс для доступа к данным, касающихся отчета о проделанной работе. Данный класс реализует следующие методы: *logWork* (метод принимает экземпляр класса *WorkItem* и создает запись о проделанной работе в базе данных), *getUsersLogWork* (метод принимает имя пользователя и возвращает список отчетов по временным затратам для конкретного пользователя), *getProjectLogWork* (метод принимает идентификатор проекта и возвращает список отчетов по проекту).

**3.5** Модуль API веб-сервиса

Модуль API веб-сервиса является своего рода ядром разрабатываемой системы. Данный модуль принимает запросы от модуля маршрутизации фреймворка Spring, затем обращается, по необходимости, к другим модулям веб-сервиса, производит предварительную обработку ответа и посылает ответ модулю маршрутизации.

Разрабатываемый модуль представлен следующим множеством классов: *AuthenticationController, TemplateController, ProjectController, TaskController,*

*LogWorkController* (см. рис. 3.10)*.*

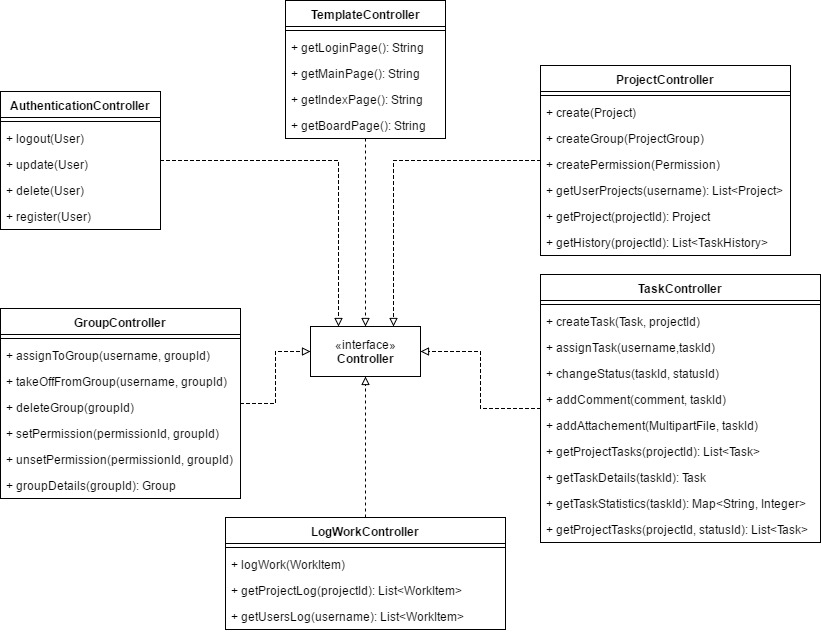


Рисунок 3.10 – Диаграмма классов модуля API веб-сервиса

Класс *AuthenticationController* предоставляет интерфейс по управлению пользователями. Класс реализует следующие методы: *logout* (метод принимает текущего пользователя и закрывает его сессию, таким образом пользователь выходит из системы), *updateProfile* (принимает экземпляр класса *User* и обновляет профиль пользователя), *deleteUser* (принимает экземпляр класса *User* и удаляет пользователя из системы).

Класс *TemplateController* предоставляет интерфейс по получению шаблонов веб-страниц. Метод *getLoginPage* возвращает название шаблона со страницей входа в систему. Метод *getMainPage* возвращает название шаблона с главной страницей приложения. Метод *getIndexPage* возвращает возвращает название шаблона главной страницы приложения. Метод *getBoardPage* возвращает название шаблона с доской проекта.

Класс *ProjectController* предоставляет интерфейс по управлению проектом. Метод *create* принимает экземпляр класса Project и создает новый проект в системе. Метод *createGroup* принимает объект класса ProjectGroup и добавляет новую группу в проект. Метод *assignToGroup* принимает имя пользователя и идентификатор группы, таким образом пользователь добавляется в проектную группу. Метод *takeOffFromGroup* принимает имя пользователя и идентификатор группы, таким образом пользователь удаляется из проектной группы. Метод *createPermission* принимает идентификатор проекта и экземпляр класса Permission, таким образом на проекте создается новый тип права. Метод *getUserProject* принимает имя пользователя и возвращает список проектов, над которыми работает пользователь. Метод *getProject* принимает идентификатор проекта ивозвращает полную информацию по проекту. Метод *getHistory* возвращает историю изменений в проекте.

Класс *GroupController* предоставляет интерфейс по управлению группами пользователей. Метод *assignToGroup* принимает имя пользователя и идентификатор группы, тем самым пользователь присоединятся к группе проекта. Метод *takeOffFromGroup* принимает имя пользователя и идентификатор группы и удаляет пользователя из группы. Метод *deleteGroup* принимает идентификатор группы, таким образом проектна группа удаляется. Методы *setPermission* и *unsetPermission* принимают идентификаторы группы и права и задают либо лишают права проектную группу. Метод *groupDetails* принимает принимает идентификатор группы и возвращает полную информацию о группе.

Класс *TaskController* предоставляет интерфейс по управлению задачами на проекте. Метод *createTask* принимает объект класса Task и идентификатор проекта, таким образом создается новая задача. Метод *assignTask* принимает имя пользователя и идентификатор задачи, тем самым пользователю назначается задача к исполнению. Метод *addComment* принимает содержимое комментария и идентификатор задачи, тем самым к задаче добавляется новый комментарий. Метод *addAttachement* принимает файл и идентификатор задачи, таким образом к задаче добавляется приложение. Метод *getProjectTasks* принимает идентификатор проекта и возвращает список задач на проекте, а его перегруженная версия вдобавок принимает идентификатор статуса задачи. Метод *getTaskStatistics* принимает идентификатор проекта и возвращает отображение где ключ – это название статуса задачи, а значение – количество задач в этом статусе.

Класс *LogWorkController* предоставляет интерфейс по управлению отчетами о проделанной работе. Метод *logWork* принимает экземпляр класса *WorkItem,* таким образом создается отчет о проделанной пользователем работе. Метод *getProjectLog* принимает идентификатор проекта и возвращает все отчеты по проекту. Метод *getUsersLog* принимает имя пользователя и возвращает список отчетов для конкретного пользователя.

**3.6** Модуль маршрутизации фреймворка Spring

Данный модуль является своего рода точкой входа в веб-сервис со строны веб-сайта. Модуль представляет собой сервлет, который принимает http-запрос и делигирует его одному из контроллеров модуля API веб-сервиса.

Ключевую роль в данном модуле играет класс *DispatcherServlet.* Данный класс является сервлетом и способен принимать запросы по протоколу http. После того, как на этот сервлет поступил запрос с помощью класса *HandlerMapping* производится выбор: какому контроллеру делегировать полученный запрос. Контроллер представляет собой класс, помеченный аннотацией *@Controller.* Выбор контроллера основан на url по которому произошло обращение, параметрах запроса и http-методе. Чтобы указать соответствие между запросом и вызываемым методом используется аннотация *@RequestMapping.* Данной анотацией можно пометить как методы контроллера, так и сам класс. Здесь с помощью атрибута value можно указать url, с помощью атрибута method указывается метод http-запроса и с мощью атрибута requestParams можно указать параметры запроса при которых метод будет вызван.

**3.7** Модуль управления задачами

Данный модуль предназначен для создания задач, смены статуса задачи поиска задач и т. д. Данный модуль представлен в виде следующих классов: *TaskController, DefaultTaskService, DatabaseTaskDao.* Диаграмма классов разрабатываемого модуля представлена на рисунке 3.11.

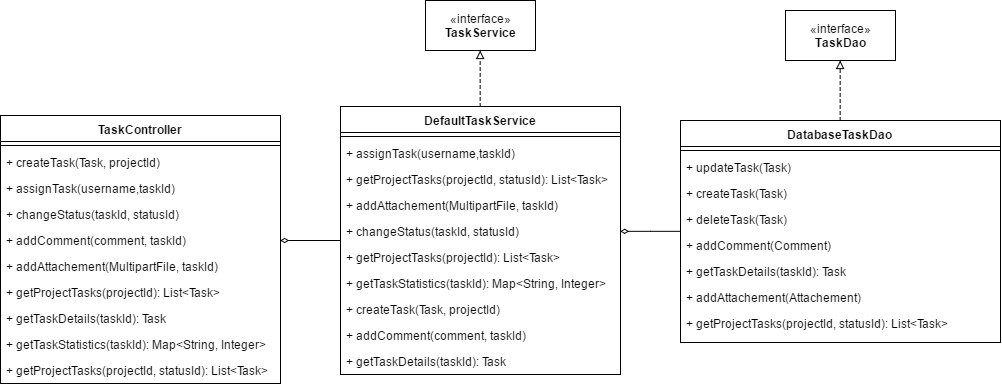


Рисунок 3.11 – Диаграмма классов модуля управления задачами

Класс *TaskController* является является контроллером, и при вызове его методов обработка запроса сразу же делегируется методу с такой же сигнатурой из класса *DefaultTaskService*.

Класс *DefaultTaskService* предоставляет интерфейс по управлению задачами на проекте. Метод *createTask* принимает объект класса Task и идентификатор проекта, таким образом создается новая задача. Метод *assignTask* принимает имя пользователя и идентификатор задачи, тем самым пользователю назначается задача к исполнению. Метод *addComment* принимает содержимое комментария и идентификатор задачи, тем самым к задаче добавляется новый комментарий. Метод *addAttachement* принимает файл и идентификатор задачи, таким образом к задаче добавляется приложение. Метод *getProjectTasks* принимает идентификатор проекта и возвращает список задач на проекте, а его перегруженная версия вдобавок принимает идентификатор статуса задачи. Метод *getTaskStatistics* принимает идентификатор проекта и возвращает отображение где ключ – это название статуса задачи, а значение – количество задач в этом статусе.

Класс *DatabaseTaskDao* является точкой доступа к данным касающихся задачи. Данный класс реализует следующие методы: *getTasksWithStatus* (метод принимает экземпляр класса *Status* и возвращает список с указанным статусом), *getTask* (метод принимает идентификатор задачи и возвращает экземпляр класса *Task*), *updateTask* (метод принимает экземпляр класса *Task* и обновляет соответствующее поля в базе данных), *deleteTask* (метод принимает экземпляр класса *Task* и удаляет соответствующую задачу из базы данных), *addAttachement* (метод принимает экземпляр класса Attachement тем самым создается приложение к задаче), *addComment* (метод принимает экземпляр класса *Comment,* таким образом добавляется комментарий к задаче), *getTaskDetails* (метод принимает идентификатор задачи, и возвращает полную информацию о ней), *getProjectTasks* (метод принимает идентификатор проекта и идентификатор статуса задачи и возвращает задачи соответствующие переданным параметрам).

**3.8** Модуль администрирования и авторизации

В данном модуле сосредоточены функции получения прав пользователя, назначения прав пользователям, а также для администрирования системы в целом (создание проекта, назначение руководителя проекта). Авторизация производится с использованием модуля Spring Security фреймворка Spring. После создания проекта его руководитель создает проектную группу. Для каждой группы указываются её права: возможность создавать задачу, право на смену статуса задач, может ли пользователь выбирать задачу самостоятельно, может ли пользователь назначать задачи другим пользователям.

Авторизация осуществляется фреймворком Spring Security. Для его настройки необходимо унаследоваться от класса *WebSecurityConfigurerAdapter* и переопределить метод *configure.* Этот процесс описан в пункте 3.1 данного раздела. Здесь необходимо на объекте класса *HttpSecurity* вызвать метод *antMatchers* и передать в него url, и роль пользователя, который сможет пройти по этому url.

Однако необходимо защитить некоторые методы сервисов от доступа пользователей, которые не имеют права на это. Для этого над методом необходимо поставить анотацию *@PreAuthorize* и в ней в атрибуте *value* написать выражение на языке *SpEL,* которое проверяет, может данный пользователь вызывать этот метод или нет. В случает если выражение вернуло *false* клиенту придет ответ с кодом 403 (Доступ запрещен), иначе метод выполнится и, в случае необходимости, будет возвращен результат его выполнения.

Администрирование системы: создание проектов, создание групп проектов, создание прав пользователей др. Эти процессы сосредоточены в классах *ProjectController* и *TaskController.* Болееподробно эти классы и их методы рассмотрены в пункте 3.5 данной главы.

**3.9** Модуль взаимодействия с веб-сервисом

Модуль взаимодействия с веб-сервисом находится на стороне веб-сайта и посылает запросы веб-сервису по протоколу http. Таким образом веб-сайт и веб-сервис обмениваются данными в формате JSON. На рисунке 3.12 представлена схема взаимодействия веб-сайта и веб-сервиса.

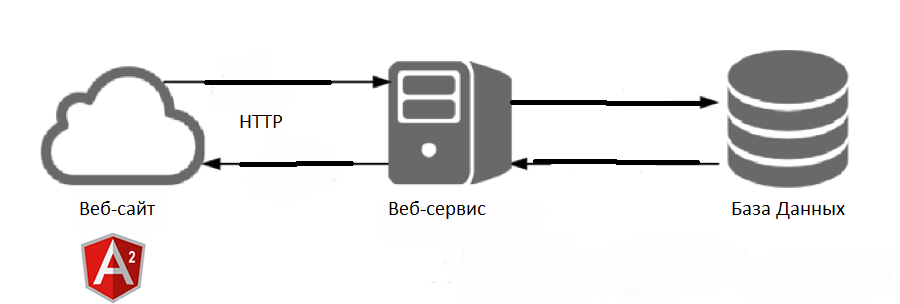


Рисунок 3.12 – Схема взаимодействия в веб-сервисом

В Angular взаимодействие с веб-сервисом происходит с помощю зависимости *$http.* Данная зависимость позволяет отправлять сервису http-запросы по выбранному URL и с учетом http-метода. Например, при получении данных с веб сервиса вызывается метод $http.get(), в него передается URL, по этому URL к веб-сервису совершается запрос с http-методом GET. Для обработки ответа сервиса необходимо в метод then передать 2 функции. Первая функция вызовется при ответе сервиса с кодом 2xx, вторая вызовется при ответе сервиса с кодом 4xx или же при внутренней

ошибке сервера 5xx.

**3.10** Модуль рендеринга html-страниц

Модуль рендеринга html-страниц предназначен для отрисовки html-страниц по заданному шаблону. Шаблон страницы загружается с веб-сервиса один раз при первом обращении к шаблону. После этого шаблон сохраняется на стороне клиент и дальнейшая его выгрузка с веб-сервиса уже не требуется.

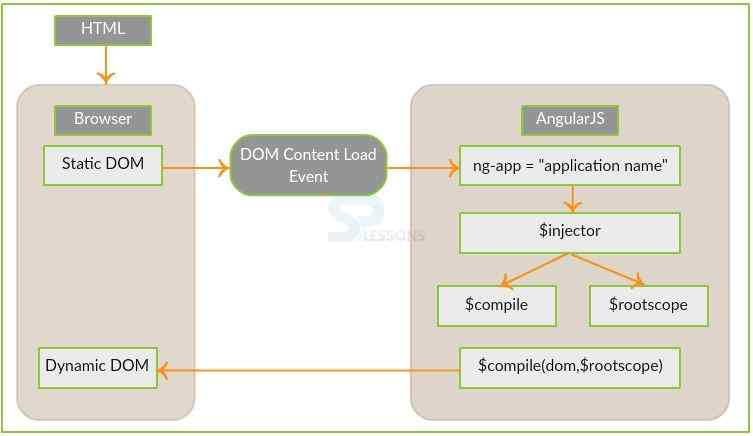


Рисунок 3.13 – Рендеринг страниц в Angular

На рисунке 3.13 представлена схема рендеринга html-страниц в фреймворке Angular.

Рендеринг html-страницы происходит следующим образом: веб-клиент выгружает с веб-сервиса html-страницу, которая является шаблоном. В местах, где нужно вставлять контент ставятся своего рода метки в виде двойных фигурных скобках и идентификатора объекта из модели ({{varName}}). После получения ответа веб-сервиса данные в формате JSON сохраняются в модели и автоматически переносятся на html-страницу.

**3.11** Фреймворк Angular

Фреймворк Angular является основным модулем на стороне веб-сайта. Данный модуль управляет как процессом взаимодействия веб-сайта с веб-сервисом, так и процессом рендеринга html-страниц. В данном модуле подключаются следующие зависимости: *ngRoute* (необходима для динамической маршрутизации между страницами см. листинг 3.4) и *dragular* для реализации функционала по перетаскиванию задач (см. листинг 4.5).

app.config(['$routeProvider', function ($routeProvider) {  
$routeProvider.when("/",  
{template : '<h1>Hello</h1>',  
controller : 'StartController'  
}).when("/main", {templateUrl : '/main',  
controller : 'MainPageController'  
}).when('/index', {templateUrl : '/index'});  
}]);

Листинг 3.4 – Маршрутизация в AngularJS

.controller('RemoveOnSpillWithModel', ['$scope', '$element', 'dragularService', function TodoCtrl($scope, $element, dragularService)

{

$scope.items1 = [{content: 'Move me, but you can only drop me in containers.'}, {content: 'If you try to drop me somewhere other than containers, I\'ll die a fiery death.'}, {content: 'Item 3'}, {content: 'Item 4'}];

$scope.items2 = [{content: 'You can drop me in the left container.'}, {content: 'Item 6'}, {content: 'Item 7'}, {content: 'Item 8'}];

var containers = $element.children().eq(0).children(); dragularService.cleanEnviroment(); dragularService([containers[0],containers[1]],{ containersModel: [$scope.items1, $scope.items2], removeOnSpill: true });

}]);

Листинг 3.5 – Пример использования библиотеки dragular

В данной главе были рассмотрены более детально модули, составляющие разрабатываемое программное средство, так же была спроектирована диаграмма классов всего программного средства.