МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ им. В. Г. Шухова»
(БГТУ им. В. Г. Шухова)

КУРСОВАЯ РАБОТА

по дисциплине «Интерфейсы ВС»

Тема: “ Создание клиент-серверного приложения с использованием Spring Framework”

Автор работы _____ Черных А.В.
(подпись) ВТ-42

Руководитель проекта _____ ст. пр.
(подпись) Торопчин Д.А.

Оценка _____

Белгород
2021 г.

Оглавление

Введение

Целью курсового проекта ставится разработка клиент-серверного приложения, серверная составляющая которого представляет собой приложение, реализованное на языке Java с использованием Spring Framework, а в качестве клиентской части – приложение, реализованное на языке JavaScript.

Предметной областью для проекта был выбран сервис просмотра досок Agile. Приложение должно предоставлять пользователям возможность регистрации и авторизации; пользователю предоставляются средства просмотра задач на доске и их описания.

Глава 1. Теоретические сведения

1.1. Java

Java - строго типизированный объектно-ориентированный язык программирования общего назначения, разработанный компанией Sun Microsystems (в последующем приобретённой компанией Oracle). Разработка ведётся сообществом, организованным через Java Community Process; язык и основные реализующие его технологии распространяются по лицензии GPL. Права на торговую марку принадлежат корпорации Oracle [1].

Программы на Java транслируются в байт-код Java, выполняемый виртуальной машиной Java (JVM) — программой, обрабатывающей байтовый код и передающей инструкции оборудованию как интерпретатор.

Достоинством подобного способа выполнения программ является полная независимость байт-кода от операционной системы и оборудования, что позволяет выполнять Java-приложения на любом устройстве, для которого существует соответствующая виртуальная машина. Другой важной особенностью технологии Java является гибкая система безопасности, в рамках которой исполнение программы полностью контролируется виртуальной машиной. Любые операции, которые превышают установленные полномочия программы (например, попытка несанкционированного доступа к данным или соединения с другим компьютером), вызывают немедленное прерывание.

На данный момент, конструкция Java состоит из трех основных компонентов:

- JVM (Java Virtual Machine) – виртуальная машина Java, которая исполняет байт-код, созданный из исходного кода программы компилятором javac.
- JRE (Java Runtime Environment) – минимальная реализация виртуальной машины, необходимая для исполнения Java-приложений. В ее состав не входит компилятор и другие средства разработки.
- JDK (Java Development Kit) – комплект разработчика приложений на языке Java. Он включает в себя компилятор, стандартные библиотеки классов Java, документацию, исполнительную систему JRE и прочее.

1.2. Spring Framework

Spring Framework - универсальный фреймворк с открытым исходным кодом для Java-платформы. Также существует форк для платформы .NET Framework, названный Spring.NET [2].

Несмотря на то, что Spring не обеспечивал какую-либо конкретную модель программирования, он стал широко распространённым в Java-сообществе главным образом как альтернатива и замена модели Enterprise JavaBeans. Spring предоставляет большую свободу Java-разработчикам в проектировании; кроме того, он предоставляет хорошо документированные и лёгкие в использовании средства решения проблем, возникающих при создании приложений корпоративного масштаба.

Между тем, особенности ядра Spring применимы в любом Java-приложении, и существует множество расширений и усовершенствований для построения веб-приложений на Java Enterprise платформе. Из-за широкой функциональности трудно определить наиболее значимые структурные элементы, из которых он состоит. Spring не всецело связан с платформой Java Enterprise, несмотря на его масштабную интеграцию с ней, что является важной причиной его популярности и признания разработчиков.

Данный фреймворк можно рассмотреть как коллекцию меньших фреймворков, большая часть которых может работать независимо друг от друга, однако они обеспечивают большую функциональность при совместном их использовании. Вышеуказанные фреймворки делятся на структурные элементы (иначе говоря, модули) типовых комплексных приложений:

- Inversion of Control-контейнер
- Фреймворк аспектно-ориентированного программирования
- Фреймворк доступа к данным
- Фреймворк управления транзакциями
- Фреймворк MVC
- Фреймворк удалённого доступа
- Фреймворк аутентификации и авторизации
- Фреймворк удалённого управления
- Фреймворк работы с сообщениями
- Тестирование

Рассмотрим подробнее фреймворк MVC. Spring MVC является веб-средой Spring. Это позволяет создавать все, что связанно с сетью, от небольших веб-сайтов до сложных веб-сервисов.

MVC расшифровывается как Модель-Представление-Контроллер (Model View Controller), что как раз-таки означает три основные части данного фреймворка. Рассмотрим их поподробнее:

- Модель – содержит данные, которые необходимо отобразить, представляют собой Java-объекты.

- Представление – отвечает за вывод данных пользователю.
- Контроллер – отвечает за обработку запросов пользователей и передачу данных модулю представления для обработки.

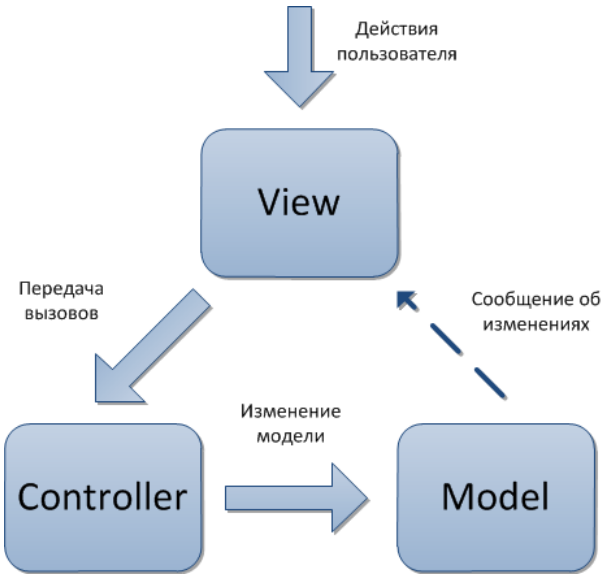

Рис. 1.2.1. Схема MVC фреймворка

Spring Boot представляет собой проект, целью которого является упрощения создания приложений на основе Spring. Он позволяет наиболее простым способом создать веб-приложение, требуя от разработчиков минимум усилий по настройке и написанию кода.

Среди особенностей Spring Boot можно отметить такие, как простота управления зависимостями, автоматическая конфигурация проекта и встроенная поддержка сервера приложений.

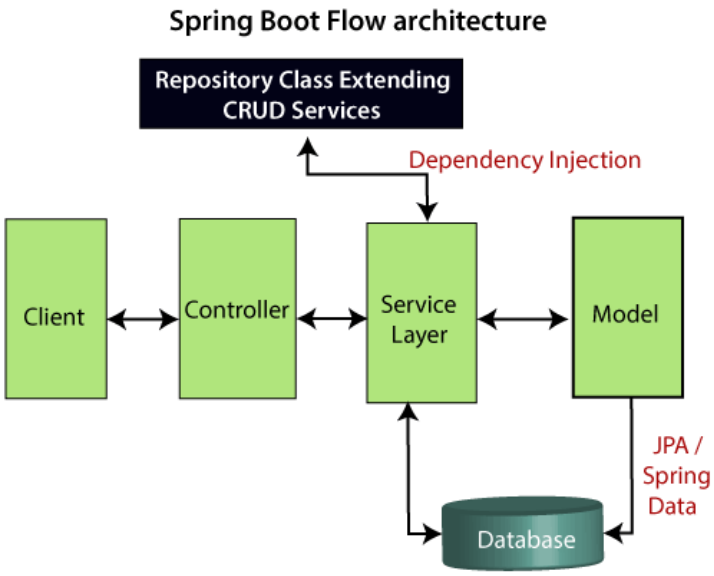

Рис. 1.2.2. Архитектура Spring Boot

1.3. Maven

Maven — фреймворк для автоматизации сборки проектов на основе описания их структуры в файлах на языке POM (англ. Project Object Model), являющемся подмножеством XML. Проект Maven издаётся сообществом Apache Software Foundation [3].

Maven обеспечивает декларативную сборку проекта. Это значит, что разработчику не нужно уделять внимание каждому аспекту сборки — все необходимые параметры настроены по умолчанию. Изменения нужно вносить лишь в том объёме, в котором программист хочет отклониться от стандартных настроек. В файлах описания проекта содержится его спецификация, а не отдельные команды выполнения. Все задачи по обработке файлов, описанные в спецификации, Maven выполняет посредством их обработки последовательностью встроенных и внешних плагинов. В основном, Maven используется для построения и управления проектами, написанными на Java, но есть и плагины для интеграции с C/C++, Ruby, Scala, PHP и другими языками.

Конечно же, нельзя не отметить, что Maven ценят за декларативную сборку проекта. Но есть еще одно огромное достоинство проекта — гибкое управление зависимостями. Maven умеет подгружать в свой локальный репозиторий сторонние библиотеки, выбирать необходимую версию пакета, обрабатывать транзитивные зависимости. Разработчики также подчёркивают независимость фреймворка от ОС. При работе из командной строки параметры зависят от платформы, но Maven позволяет не обращать внимания на этот аспект.

1.4. Java Persistence API (JPA)

Java Persistence API (JPA) — спецификация API Java EE, предоставляет возможность сохранять в удобном виде Java-объекты в базе данных [4].

Сама Java не содержит реализации JPA, однако существует множество реализаций данной спецификации от разных компаний (открытых и нет). JPA реализует концепцию ORM.

ORM (Object-Relational Mapping или же объектно-реляционное отображение) – технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая виртуальную объектную базу данных.

Данная библиотека позволяет решить задачу связи классов Java и таблиц данных в реляционной базе данных, а также типов данных Java и базы данных.

Помимо этого, предоставляются возможности по автоматической генерации и обновлению набора таблиц, построения запросов к базе данных, обработки полученных результатов, что приводит к значительному уменьшению времени разработки.

ORM — это, по сути, концепция о том, что Java объект можно представить как данные в БД (и наоборот). Она нашла воплощение в виде спецификации JPA — Java Persistence API. Спецификация — это уже описание Java API, которое выражает эту концепцию. Спецификация рассказывает, какими средствами мы должны быть обеспечены (т.е. через какие интерфейсы мы сможем работать), чтобы работать по концепции ORM. И как использовать эти средства. Реализацию средств спецификация не описывает. Это даёт возможность использовать для одной спецификации разные реализации. Можно упростить и сказать, что спецификация — это описание API. Следовательно, чтобы использоать JPA нам требуется некоторая реализацию, при помощи которой мы будем пользоваться технологией. Одна из самых популярных реализаций JPA является Hibernate.

Hibernate — библиотека для языка программирования Java, предназначенная для решения задач объектно-реляционного отображения, самая популярная реализация спецификации JPA. Распространяется свободно на условиях GNU Lesser General Public License [5].

Главным достоинством данной библиотеки является возможность сократить объемы низкоуровневого программирования при работе с реляционными базами данных.

1.5. REST

REST (от англ. Representational State Transfer — «передача состояния представления») — архитектурный стиль взаимодействия компонентов распределённого приложения в сети. REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой гипермедиа-системы. В определённых случаях (интернет-магазины, поисковые системы, прочие системы, основанные на данных) это приводит к повышению производительности и упрощению архитектуры.

Свойства архитектуры, которые зависят от ограничений, наложенных на REST-системы:

- Производительность — взаимодействие компонентов системы может являться доминирующим фактором производительности и эффективности сети с точки зрения пользователя;
- Масштабируемость для обеспечения большого числа компонентов и взаимодействий компонентов.

Существует шесть обязательных ограничений для построения распределённых REST-приложений по Филдингу.

Выполнение этих ограничительных требований обязательно для REST-систем. Накладываемые ограничения определяют работу сервера в том, как он может обрабатывать и отвечать на запросы клиентов. Действуя в рамках этих ограничений, система приобретает такие желательные свойства как производительность, масштабируемость, простота, способность к изменениям, переносимость, отслеживаемость и надёжность.

Если сервис-приложение нарушает любое из этих ограничительных условий, данную систему нельзя считать REST-системой.

Обязательными условиями-ограничениями являются:

- Модель клиент-сервер
- Отсутствие состояния
- Кэширование
- Единообразие интерфейса
- Слои
- Код по требованию (необязательное ограничение)

Глава 2. Проектирование и разработка приложения

Целью работы является создание клиент-серверного приложения, поэтому разработку проекта можно разделить на три основные составляющие:

- разработка и проектирование базы данных;
- разработка клиентской части проекта - frontend-а;
- разработка серверной части проекта - backend-а.

2.1. Проектирование БД

Прежде всего необходимо спроектировать структуру базы данных - выделить сущности и выявить связи между ними, а также определить наборы атрибутов.

В БД имеются 4 таблицы:

1. Пользователи (users): имя, отображаемое имя, адрес электронной почты, пароль;
2. Доски (boards): наименование, описание;
3. Стадии (stages): наименование, описание, доска-родитель;
4. Задачи (tasks): наименование, описание, стадия-родитель.

Также в БД были занесены некоторые тестовые данные.

Для создания данной структуры БД был написан SQL-скрипт, исходный код которого доступен в приложении.

В результате получим следующую диаграмму:

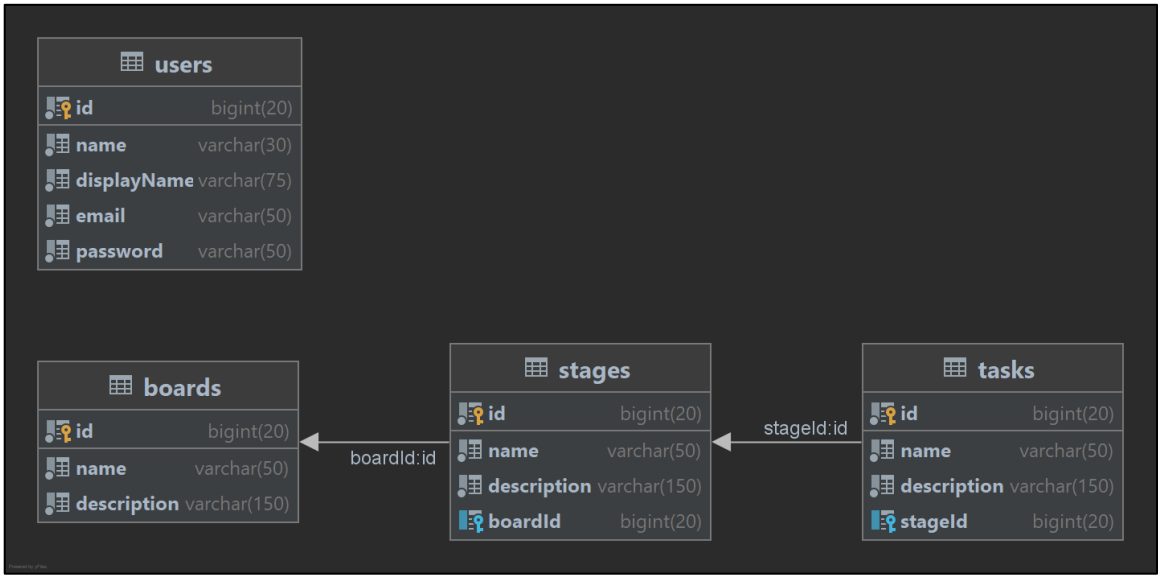

Рис 2.1.1. Диаграмма БД

Spring Boot базируется на паттерне проектирования MVC, следовательно необходимо реализовать модели соответствующие сущностям БД.

Исходный код модели User

```
package com.cherdev.teamly.entities.users;

import com.sun.istack.NotNull;

import javax.persistence.*;

@Entity
@Table(name = "users")
public class User
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String name;
    private String displayName;
    private String email;
    private String password;

    public User()
    {
    }

    public User(String name, String displayName, String email, String
password)
    {
        this.name = name;
        this.displayName = displayName;
        this.email = email;
        this.password = password;
    }

    public User(long id, String name, String displayName, String email, String
password)
    {
        this(name, displayName, email, password);

        this.id = id;
    }

    public long getId()
    {
        return id;
    }

    public String getName()
    {
        return name;
    }

    public String getDisplayName()
    {
        return displayName;
    }

    public String getEmail()
    {
        return email;
    }

    public String getPassword()
    {
```

```
        return password;
    }

    public void setId(long id)
    {
        this.id = id;
    }

    public void setName(String name)
    {
        this.name = name;
    }

    public void setDisplayName(String displayName)
    {
        this.displayName = displayName;
    }

    public void setEmail(String email)
    {
        this.email = email;
    }

    public void setPassword(String password)
    {
        this.password = password;
    }
}
```

Аннотация @Entity указывает, что данный объект реализует сущность БД, при этом его поля являются столбцами таблиц, а для установки ограничений, нужно использовать дополнительные аннотации.

К первичному ключу применяются аннотации @Id, означающая, что поле является первичным ключом, и @GeneratedValue, задающая стратегию генерирования значения ключа.

Также к атрибутам применимы аннотации @OneToOne, @ManyToOne, @OneToMany и @ManyToMany, создающие связи между сущностями типа один к одному, многие к одному, один ко многим и многие ко многим соответственно.

Исходный код модели Stage

```
package com.cherdev.teamly.entities.boards;

import com.fasterxml.jackson.annotation.JsonIgnore;

import javax.persistence.*;
import java.util.List;

@Entity
@Table(name = "stages")
public class Stage
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String name;
```

```
    private String description;
    @OneToMany(mappedBy = "stage")
    private List<Task> tasks;
    @ManyToOne
    @JoinColumn(name = "board_id")
    private Board board;

    public Stage()
    {

    }

    public Stage(String name, String description, Board board)
    {
        this.name = name;
        this.description = description;
        this.board = board;
    }

    public Stage(long id, String name, String description, Board board)
    {
        this(name, description, board);

        this.id = id;
    }

    public long getId()
    {
        return id;
    }

    public String getName()
    {
        return name;
    }

    public List<Task> getTasks()
    {
        return tasks;
    }

    @JsonIgnore
    public Board getBoard()
    {
        return board;
    }
}
```

2.2. Разработка клиентской части приложения

Клиентская часть проекта представляет собой приложение, написанное с помощью фреймворка для JS – Vue JS. Функционал frontend-приложения включает в себя регистрацию и авторизацию пользователей, просмотр задач и их описания.

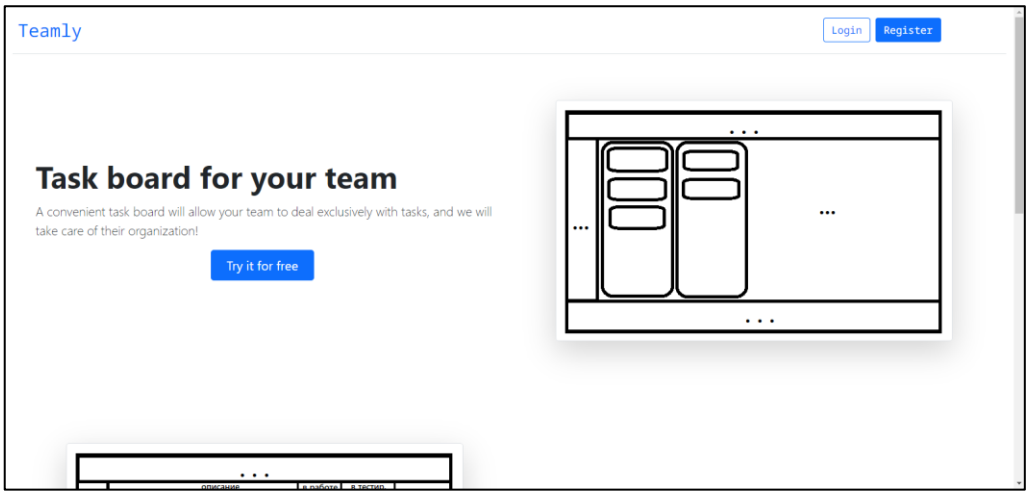

Рисунок 2.2.1. Главная страница

Данные для выпадающих списков запрашиваются у backend-приложения с помощью запроса GET.

Пример GET-запросов с использованием библиотеки Axios

```
this.$http.get("/boards/1")
    .then(response => {
      this.data = response.data
    })
    .catch(e => {
    })
```

Рисунок 2.2.2. Страница доски Agile

Данная страница получает данные о задачах конкретной доски.

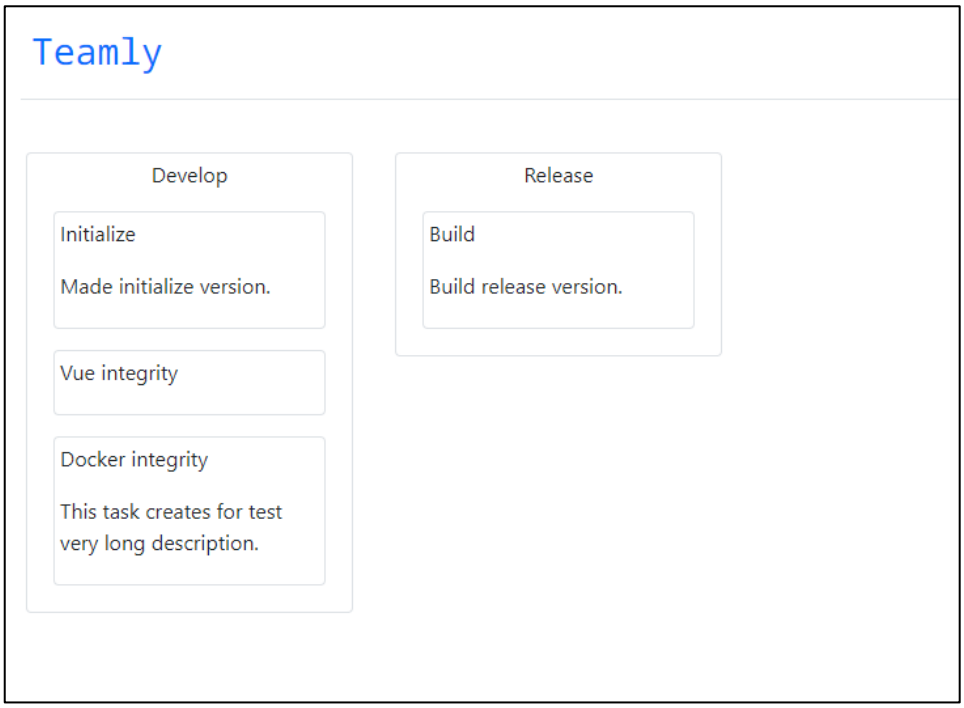

Рисунок 2.2.3. Доска Agile с описаниями задач

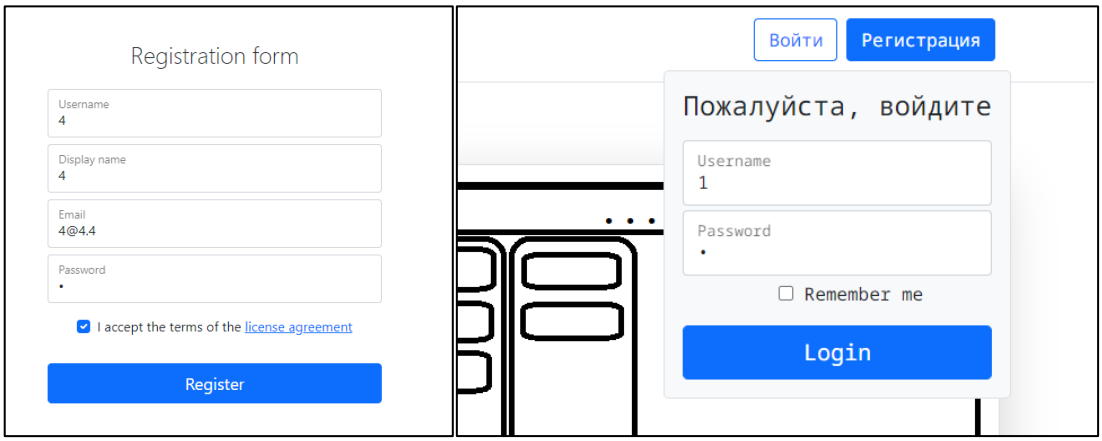

Рисунок 2.2.4. а) Страница регистрации; б) Страница авторизации

При регистрации пользователя отправляется POST-запрос с информацией о пользователе, при авторизации – GET-запрос.

Пример POST-запроса

```
this.$http.post(
        "/users/register",
        qs.stringify(this.form),
        {
          dataType: "x-www-form-
urlencoded",
          headers: { 'content-type':
'application/x-www-form-urlencoded' }
      })
      .then(response => {
        if (response.data ===
"Success")
          this.$router.push({
```

```
            name: "Home"
          })
      })
      .catch(e => {
      })
```

2.3. Разработка серверной части приложения

Разработка backend-приложения началась с создания шаблона проекта на Spring Boot с помощью фирменной утилиты Spring Initializer. В результате был создан файл зависимостей Maven – Pom.xml, папка main с главным классом TeamlyApplication и файлом конфигурации application.property.

Для проекта была выбрана СУБД MySQL. Настройка подключения к СУБД осуществляется в файле application.property.

Исходный код файла application.property

```
spring.datasource.url=jdbc:mysql://localhost:3306/teamlydb
spring.datasource.username=root
spring.datasource.password=superpass321!
```

Один из этапов разработки backend-приложения был рассмотрен в разработке БД при создании моделей. Модели являются лишь представлением записей таблиц БД, их необходимо дополнить функционалом, с помощью которого можно будет получать и заполнять данные базы данных.

Для этого реализуем интерфейсы к разработанным моделям (репозитории) и наследуем их от класса JpaRepository, реализующего необходимые методы.

Стоит отметить, что базового функционала часто бывает недостаточно, т.к. он не позволяет делать выборку по нужным полям. В таком случае можно определить собственные методы с именем в определённом формате, тогда реализацию методов возьмёт на себя фреймворк.

Пример реализации интерфейса модели User с определением собственного метода

```
package com.cherdev.teamly.repositories.users;

import com.cherdev.teamly.entities.users.User;
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long>
{
    User findByName(String name);
}
```

Определённый метод имеет формат имени findBy<атрибут> и принимает соответствующий параметр. Таким образом будет реализован метод выборки записей пользователей по имени.

Доступ к функционалу серверной составляющей осуществляется через REST API, которое реализуется контроллерами.

Исходный код контроллера UserController

```
package com.cherdev.teamly.controllers.users;

import com.cherdev.teamly.entities.users.User;
import com.cherdev.teamly.repositories.users.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import javax.persistence.EntityManager;
import java.util.List;

@RestController
@RequestMapping("/api/users")
public class UserController
{
    @Autowired
    private UserRepository userRepository;
    @Autowired
    private EntityManager entityManager;

    @GetMapping
    public List<User> list()
    {
        return userRepository.findAll();
    }

    @GetMapping("/{id}")
    public User user(@PathVariable long id)
    {
        return userRepository.findById(id).orElse(null);
    }

    @GetMapping(value = "/login")
    public ResponseEntity<String> login(@RequestParam String name,
                                        @RequestParam String password)
    {
        User user = userRepository.findByName(name);

        if (user != null)
        {
            if (!password.equals(user.getPassword()))
            {
                return new ResponseEntity<>("Invalid password",
HttpStatus.BAD_REQUEST);
            }
        }
        else
        {
            return new ResponseEntity<>("User doesn't exists",
HttpStatus.BAD_REQUEST);
```

```
        }

        return new ResponseEntity<>("Success", HttpStatus.OK);
    }

    @PostMapping(value = "/register", consumes = {
MediaType.APPLICATION_FORM_URLENCODED_VALUE })
    public ResponseEntity<String> register(User user)
    {
        User findedUser = userRepository.findByName(user.getName());

        if (findedUser != null)
        {
            return new ResponseEntity<>("User already contains",
HttpStatus.BAD_REQUEST);
        }
        else
        {
            userRepository.saveAndFlush(user);
        }

        return new ResponseEntity<>("Success", HttpStatus.CREATED);
    }
}
```

Аннотация @RestController сообщает фреймворку, что данный класс является REST контроллером и в данном классе будет реализована логика обработки клиентских запросов. @Autowired означает, что для атрибута необходимо внедрить зависимости. Аннотации @GetMapping и @PostMapping реализуют соответственно GET и POST запросы, при этом параметры GET запроса передаются через URL, а POST запроса – в его теле.

2.4. Документация к API

Документация доступна на сайте: ссылка.

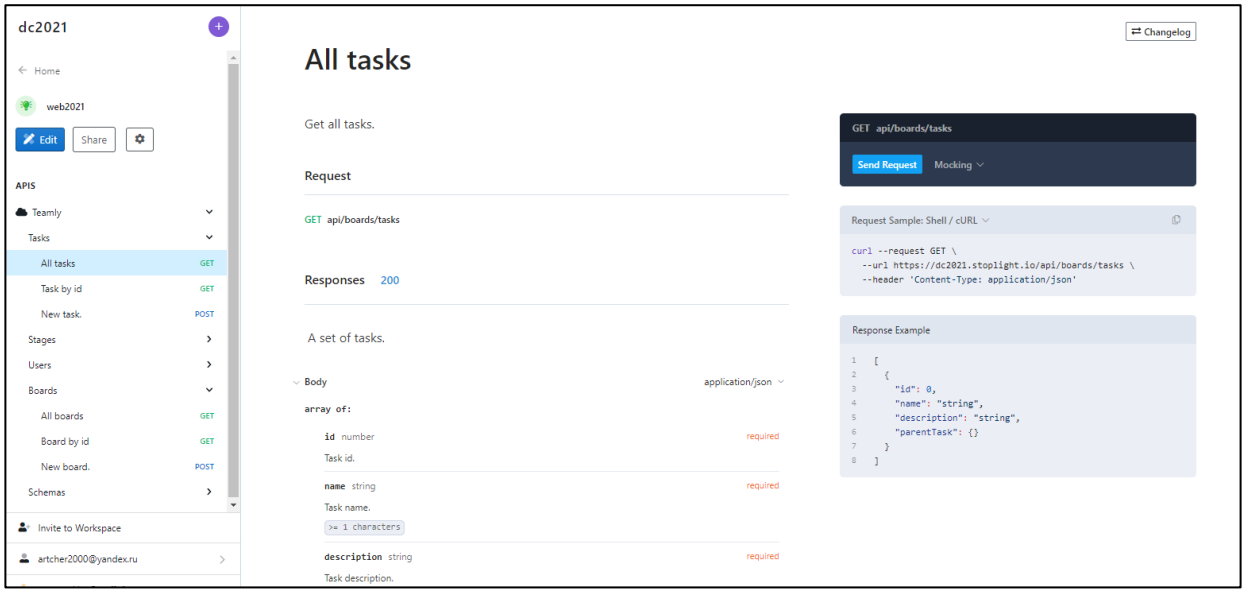

Рисунок 2.4.1. Внешний вид документации

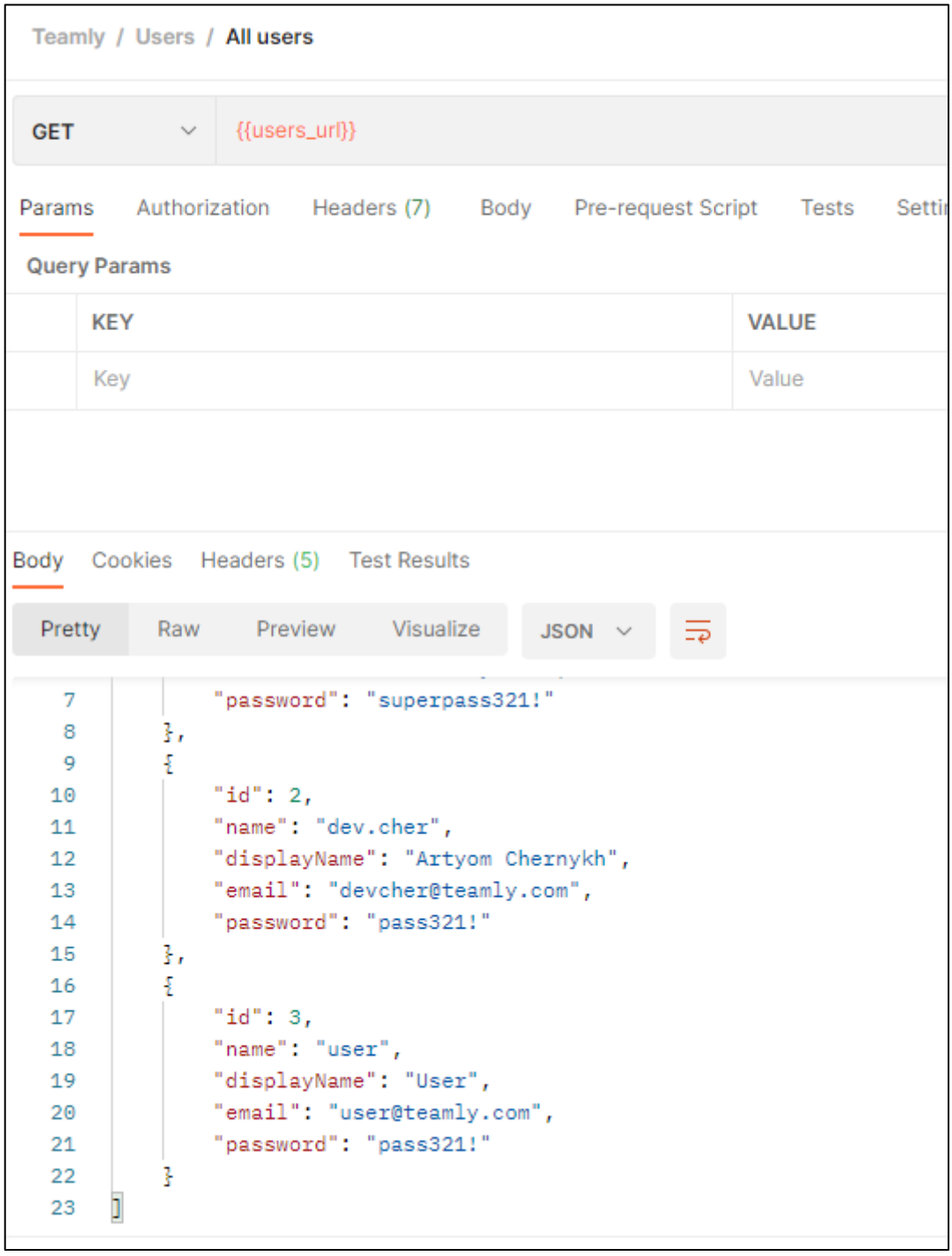

Рисунок 2.4.2. Пример запроса в Postman

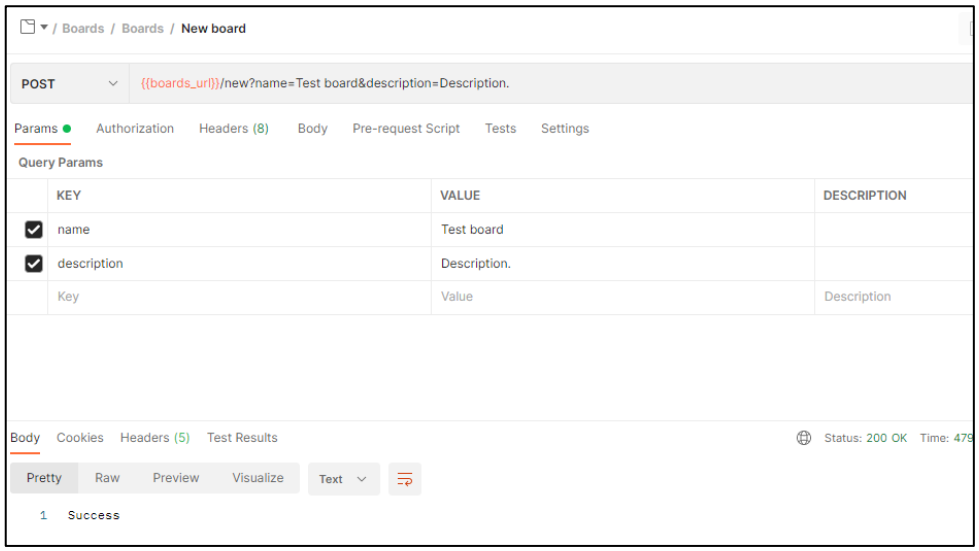

Рисунок 2.4.3. Пример запроса в Postman

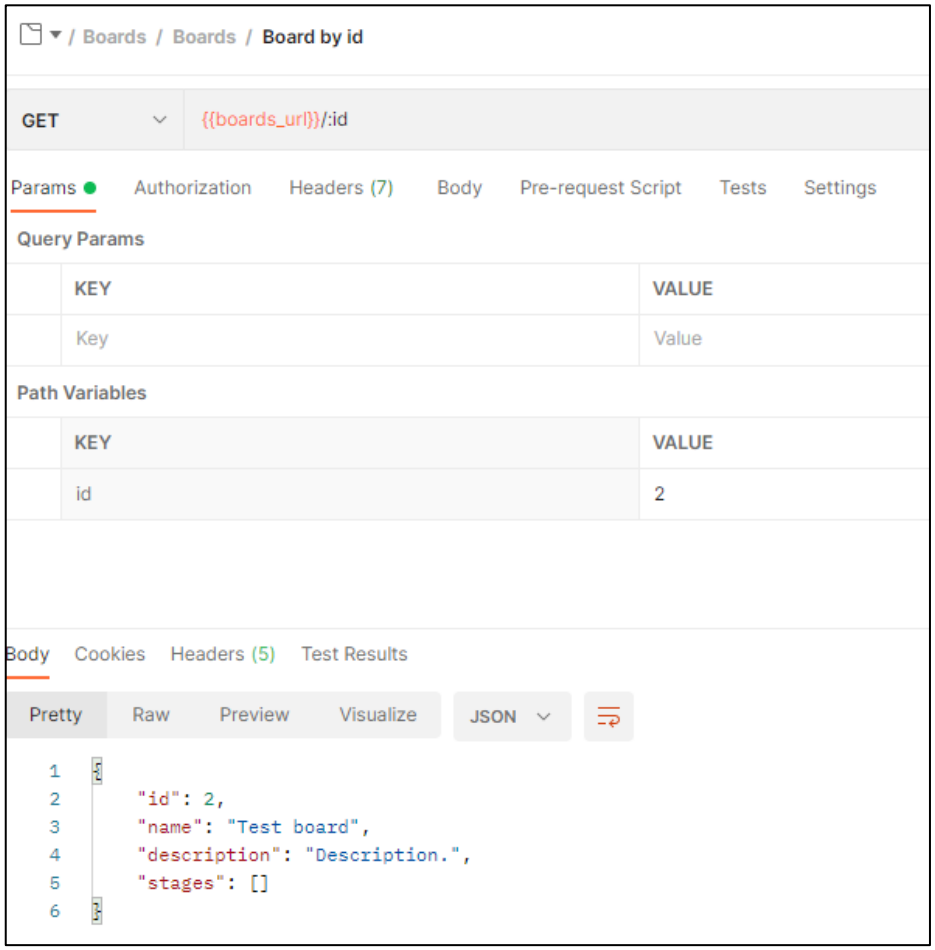

Рисунок 2.4.4. Пример запроса в Postman

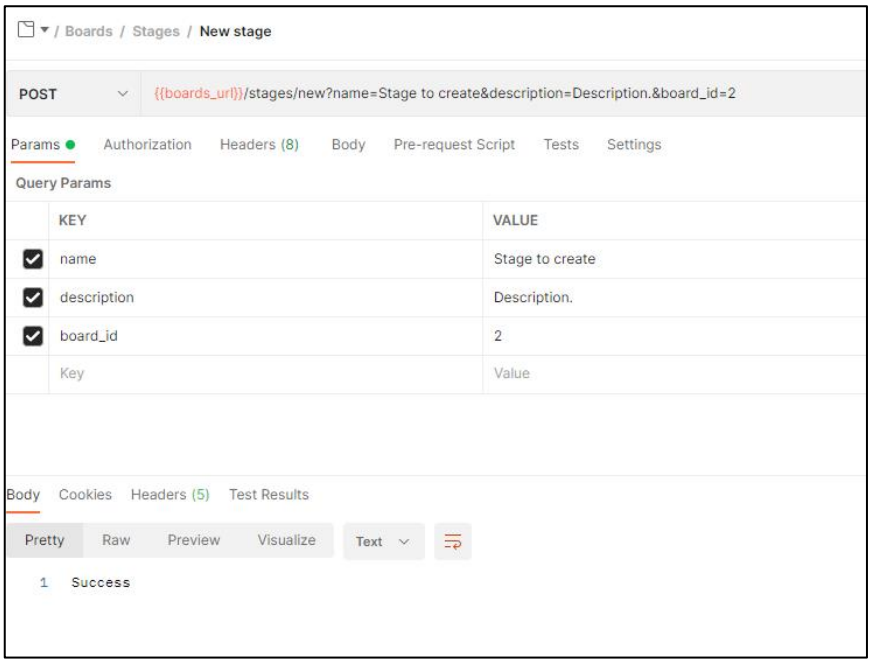

Рисунок 2.4.5. Пример запроса в Postman

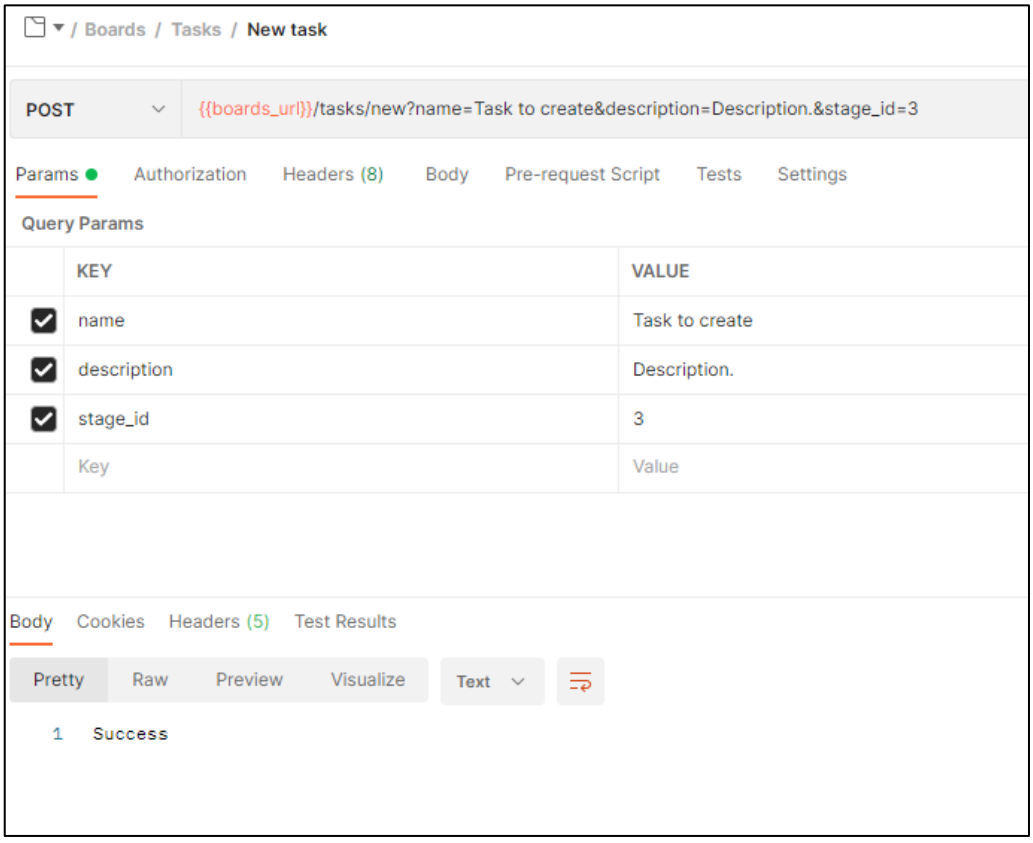

Рисунок 2.4.6. Пример запроса в Postman

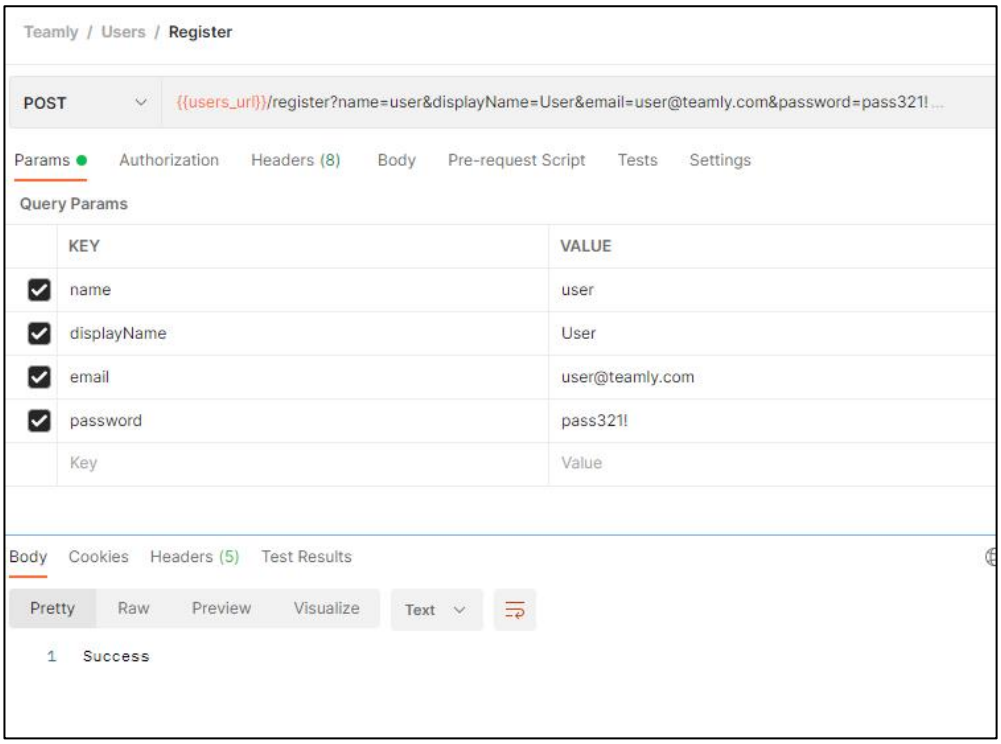

Рисунок 2.4.7. Пример запроса в Postman

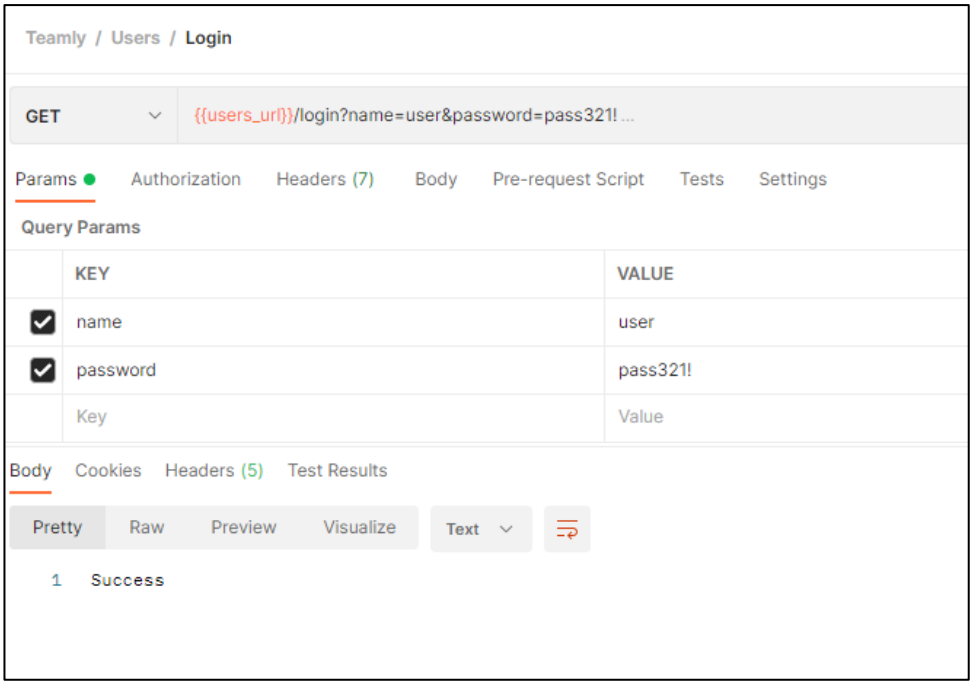

Рисунок 2.4.8. Пример запроса в Postman

Заключение

В ходе выполнения курсового проекта были получены практические навыки разработки клиент-серверного приложения, произведено знакомство с архитектурой REST, фреймворком Spring Boot.

В результате работы было разработано клиент-серверное приложение «Средства просмотра досок Agile», позволяющее пользователям просматривать задачи на доске и их описания. Для разработки клиентского приложения был применён фреймворк Vue JS, для серверного приложения – фреймворк Spring Boot, а в качестве СУБД была использована MySQL.

Была спроектирована и реализована структура БД для предметной области, разработан REST API для доступа к данным БД и система HTTP запросов к API из клиентского приложения.

В качестве вспомогательных средств были использованы: Maven, Hibernate (JPA) и др.

Список литературы

1. Java - Википедия. – [Электронный ресурс]. – URL: https://ru.wikipedia.org/wiki/Java;
2. Spring Framework - Википедия. – [Электронный ресурс]. – URL: https://ru.wikipedia.org/wiki/Spring_Framework;
3. Java Persistence_API - Википедия. – [Электронный ресурс]. – URL: https://ru.wikipedia.org/wiki/Java_Persistence_API;
4. Hibernate (библиотека) - Википедия. – [Электронный ресурс]. – URL: https://ru.wikipedia.org/wiki/Hibernate_(Библиотека);
5. Документация Spring [Электронный ресурс] – URL: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/.

Приложение А. SQL-скрипт

```
CREATE DATABASE teamlydb;

use teamlydb;

CREATE TABLE users
(
    id bigint primary key auto_increment,
    name varchar(30) not null,
    displayName varchar(75) not null,
    email varchar(50) not null,
    password varchar(50) not null
);

DROP TABLE users;

CREATE TABLE boards
(
    id bigint primary key auto_increment,
    name varchar(50) not null,
    description varchar(50) not null
);

ALTER TABLE boards
MODIFY description varchar(150) not null default '';

CREATE TABLE stages
(
    id bigint primary key auto_increment,
    name varchar(50) not null,
    description varchar(150) not null default '',
    boardId bigint REFERENCES boards(id) ON DELETE CASCADE ON UPDATE CASCADE
);

CREATE TABLE tasks
(
    id bigint primary key auto_increment,
    name varchar(50) not null,
    description varchar(150) not null default '',
    stageId bigint REFERENCES stages(id) ON DELETE CASCADE ON UPDATE CASCADE
);

SELECT b.name, s.name, t.name, t.description
FROM boards b
JOIN stages s on b.id = s.boardId
JOIN tasks t on s.id = t.stageId;
```

Приложение Б. Модели

User.java:

```
package com.cherdev.teamly.entities.users;

import com.sun.istack.NotNull;

import javax.persistence.*;

@Entity
@Table(name = "users")
public class User
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String name;
    private String displayName;
    private String email;
    private String password;

    public User()
    {
    }

    public User(String name, String displayName, String email, String password)
    {
        this.name = name;
        this.displayName = displayName;
        this.email = email;
        this.password = password;
    }

    public User(long id, String name, String displayName, String email, String
password)
    {
        this(name, displayName, email, password);

        this.id = id;
    }

    public long getId()
    {
        return id;
    }

    public String getName()
    {
        return name;
    }

    public String getDisplayName()
    {
        return displayName;
    }

    public String getEmail()
    {
        return email;
    }

    public String getPassword()
```

```
    {
        return password;
    }

    public void setId(long id)
    {
        this.id = id;
    }

    public void setName(String name)
    {
        this.name = name;
    }

    public void setDisplayName(String displayName)
    {
        this.displayName = displayName;
    }

    public void setEmail(String email)
    {
        this.email = email;
    }

    public void setPassword(String password)
    {
        this.password = password;
    }
}
```

Board.java:

```
package com.cherdev.teamly.entities.boards;

import javax.persistence.*;
import java.util.List;

@Entity
@Table(name = "boards")
public class Board
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String name;
    private String description;
    @OneToMany(mappedBy = "board")
    private List<Stage> stages;

    public Board()
    {

    }

    public Board(String name, String description)
    {
        this.name = name;
        this.description = description;
    }

    public long getId()
    {
```

```
        return id;
    }

    public String getName()
    {
        return name;
    }

    public String getDescription()
    {
        return description;
    }

    public List<Stage> getStages()
    {
        return stages;
    }
}
```

Stage.java:

```
package com.cherdev.teamly.entities.boards;

import com.fasterxml.jackson.annotation.JsonIgnore;

import javax.persistence.*;
import java.util.List;

@Entity
@Table(name = "stages")
public class Stage
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String name;
    private String description;
    @OneToMany(mappedBy = "stage")
    private List<Task> tasks;
    @ManyToOne
    @JoinColumn(name = "board_id")
    private Board board;

    public Stage()
    {

    }

    public Stage(String name, String description, Board board)
    {
        this.name = name;
        this.description = description;
        this.board = board;
    }

    public Stage(long id, String name, String description, Board board)
    {
        this(name, description, board);

        this.id = id;
    }
```

```
    public long getId()
    {
        return id;
    }

    public String getName()
    {
        return name;
    }

    public List<Task> getTasks()
    {
        return tasks;
    }

    @JsonIgnore
    public Board getBoard()
    {
        return board;
    }
}
```

Task.java:

```
package com.cherdev.teamly.entities.boards;

import com.fasterxml.jackson.annotation.JsonIgnore;

import javax.persistence.*;

@Entity
@Table(name = "tasks")
public class Task
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String name;
    private String description;
    @ManyToOne
    @JoinColumn(name = "stage_id")
    private Stage stage;
    @ManyToOne
    @JoinColumn(name = "parent_task_id")
    private Task parentTask;

    public Task()
    {
    }

    public Task(String name, String description, Stage stage, Task parentTask)
    {
        this.name = name;
        this.description = description;
        this.stage = stage;
        this.parentTask = parentTask;
    }

    public long getId()
    {
        return id;
    }
```

```
    public String getName()
    {
        return name;
    }

    public String getDescription()
    {
        return description;
    }

    @JsonIgnore
    public Stage getStage()
    {
        return stage;
    }

    public Task getParentTask()
    {
        return parentTask;
    }

    public void setId(long id)
    {
        this.id = id;
    }

    public void setName(String name)
    {
        this.name = name;
    }

    public void setDescription(String description)
    {
        this.description = description;
    }

    public void setStage(Stage stage)
    {
        this.stage = stage;
    }

    public void setParentTask(Task parentTask)
    {
        this.parentTask = parentTask;
    }
}
```

Приложение В. Репозитории

UserRepository.java:

```
package com.cherdev.teamly.repositories.users;

import com.cherdev.teamly.entities.users.User;
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long>
{
    User findByName(String name);
}
```

BoardRepository.java:

```
package com.cherdev.teamly.repositories.board;

import com.cherdev.teamly.entities.boards.Board;
import org.springframework.data.jpa.repository.JpaRepository;

public interface BoardRepository extends JpaRepository<Board, Long>
{
}
```

StageRepository.java:

```
package com.cherdev.teamly.repositories.board;


import com.cherdev.teamly.entities.boards.Stage;

import org.springframework.data.jpa.repository.JpaRepository;


public interface StageRepository extends JpaRepository<Stage, Long>

{

}
```

TaskRepository.java:

```
package com.cherdev.teamly.repositories.board;


import com.cherdev.teamly.entities.boards.Task;

import org.springframework.data.jpa.repository.JpaRepository;


public interface TaskRepository extends JpaRepository<Task, Long>

{

}
```

Приложение Г. REST контроллеры

UserCntroller.java:

```
package com.cherdev.teamly.controllers.users;

import com.cherdev.teamly.entities.users.User;
import com.cherdev.teamly.repositories.users.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import javax.persistence.EntityManager;
import java.util.List;

@RestController
@RequestMapping("/api/users")
public class UserController
{
    @Autowired
    private UserRepository userRepository;
    @Autowired
    private EntityManager entityManager;

    @GetMapping
    public List<User> list()
    {
        return userRepository.findAll();
    }

    @GetMapping("/{id}")
    public User user(@PathVariable long id)
    {
        return userRepository.findById(id).orElse(null);
    }

    @GetMapping(value = "/login")
    public ResponseEntity<String> login(@RequestParam String name,
                                        @RequestParam String password)
    {
        User user = userRepository.findByName(name);

        if (user != null)
        {
            if (!password.equals(user.getPassword()))
            {
                return new ResponseEntity<>("Invalid password",
HttpStatus.BAD_REQUEST);
            }
        }
        else
        {
            return new ResponseEntity<>("User doesn't exists",
HttpStatus.BAD_REQUEST);
        }

        return new ResponseEntity<>("Success", HttpStatus.OK);
    }

    @PostMapping(value = "/register", consumes = {
MediaType.APPLICATION_FORM_URLENCODED_VALUE })
```

```
    public ResponseEntity<String> register(User user)
    {
        User findedUser = userRepository.findByName(user.getName());

        if (findedUser != null)
        {
            return new ResponseEntity<>("User already contains",
HttpStatus.BAD_REQUEST);
        }
        else
        {
            userRepository.saveAndFlush(user);
        }

        return new ResponseEntity<>("Success", HttpStatus.CREATED);
    }
}
```

BoardCntroller.java:

```
package com.cherdev.teamly.controllers.boards;

import com.cherdev.teamly.entities.boards.Board;
import com.cherdev.teamly.entities.boards.Stage;
import com.cherdev.teamly.properties.mappings.BoardsMappings;
import com.cherdev.teamly.repositories.board.BoardRepository;
import com.cherdev.teamly.repositories.board.StageRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping(BoardsMappings.BOARDS)
public class BoardController
{
    @Autowired private BoardRepository boardRepository;

    @GetMapping
    public List<Board> allBoards()
    {
        return boardRepository.findAll();
    }

    @GetMapping(BoardsMappings.BY_ID_KEY)
    public Board board(@PathVariable long id)
    {
        return boardRepository.findById(id).get();
    }

    @PostMapping(BoardsMappings.NEW)
    public ResponseEntity<String> newBoard(@RequestParam String name,
                                           @RequestParam(required = false,
defaultValue = "") String description)
    {
        Board newBoard = new Board(name, description);

        boardRepository.saveAndFlush(newBoard);

        return new ResponseEntity<>("Success", HttpStatus.OK);
```

```
    }
}
```

StageCntroller.java:

```
package com.cherdev.teamly.controllers.boards;

import com.cherdev.teamly.entities.boards.Board;
import com.cherdev.teamly.entities.boards.Stage;
import com.cherdev.teamly.properties.mappings.BoardsMappings;
import com.cherdev.teamly.repositories.board.BoardRepository;
import com.cherdev.teamly.repositories.board.StageRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping(BoardsMappings.STAGES)
public class StageController
{
    @Autowired private BoardRepository boardRepository;
    @Autowired private StageRepository stageRepository;

    @GetMapping
    public List<Stage> allStages()
    {
        return stageRepository.findAll();
    }

    @GetMapping(BoardsMappings.BY_ID_KEY)
    public Stage stage(@PathVariable long id)
    {
        return stageRepository.findById(id).orElse(null);
    }

    @PostMapping(BoardsMappings.NEW)
    public ResponseEntity<String> newStage(@RequestParam String name,
                                           @RequestParam(required    =    false,
defaultValue = "") String description,
```

```
                                                @RequestParam(name = "board_id") long
boardId)
    {
        Board board = boardRepository.findById(boardId).orElse(null);

        if (board != null)
        {
            Stage newStage = new Stage(name, description, board);

            stageRepository.saveAndFlush(newStage);

            return new ResponseEntity<>("Success", HttpStatus.OK);
        }
        else
        {
            return     new     ResponseEntity<>("Board     doesn't     exists",
HttpStatus.BAD_REQUEST);
        }
    }
}
```

TaskController.java:

```
package com.cherdev.teamly.controllers.boards;

import com.cherdev.teamly.entities.boards.Stage;
import com.cherdev.teamly.entities.boards.Task;
import com.cherdev.teamly.properties.mappings.BoardsMappings;
import com.cherdev.teamly.repositories.board.StageRepository;
import com.cherdev.teamly.repositories.board.TaskRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.lang.Nullable;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping(BoardsMappings.TASKS)
public class TaskController
{
    @Autowired private StageRepository stageRepository;
```

```
    @Autowired private TaskRepository taskRepository;

    @GetMapping
    public List<Task> allTasks()
    {
        return taskRepository.findAll();
    }

    @GetMapping(BoardsMappings.BY_ID_KEY)
    public Task task(@PathVariable long id)
    {
        return taskRepository.findById(id).orElse(null);
    }

    @PostMapping(BoardsMappings.NEW)
    public ResponseEntity<String> newTask(@RequestParam String name,
                                          @RequestParam(required    =    false,
defaultValue = "") String description,
                                          @RequestParam(name = "stage_id") long
stageId,
                                          @RequestParam(name = "parent_task_id",
required = false) Long parentTaskId)
    {
        Stage stage = stageRepository.findById(stageId).orElse(null);

        if (stage != null)
        {
            Task newTask = new Task(name, description, stage, parentTaskId != null
                    ? taskRepository.findById(parentTaskId).get()
                    : null);

            taskRepository.saveAndFlush(newTask);

            return new ResponseEntity<>("Success", HttpStatus.OK);
        }
        else
        {
            return     new     ResponseEntity<>("Stage     doesn't     exists",
HttpStatus.BAD_REQUEST);
        }
    }
}
```

Приложение Д. Главный класс TeamlyApplication

TeamlyApplication.java:

```
package com.cherdev.teamly;


import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;


import java.sql.*;


@SpringBootApplication
public class TeamlyApplication
{
	public static void main(String[] args)
	{
		SpringApplication.run(TeamlyApplication.class, args);
	}
}
```