МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ «ВИТЕБСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ П.М. МАШЕРОВА»

Факультет математики и информационных технологий

Кафедра информатики и информационных технологий

КУРСОВАЯ РАБОТА

по дисциплине «Современные постреляционные базы данных»

**Разработка web-приложения с использованием современных фреймворков**

Ковзов Владислав Сергеевич

3 курс, 31 группа

Руководитель

Ермоченко Сергей Александрович

доцент

Витебск, 2021

**Реферат**

Курсовая работа, 36 стр., 22 рис.

WEB-ПРОГРАММИРОВАНИЕ, СОВРЕМЕННЫЕ ФРЕЙМВОРКИ, ЮНИТ ТЕСТЫ

Объект исследования - современные фреймворки стека Java технологий.

Предмет исследования - возможности фреймворков по разработке web-приложений.

Цель работы – изучить современные фреймворки стека Java технологии и продемонстрировать их возможности на примере разработки некоторого web-приложения.

Методы исследования – изучение литературы по выше указанной теме, создание web-приложения.

Результатом проделанной работы является изучение вышеуказанных технологий и создание на их основе web-приложения.

**Оглавление**

[**Реферат** 2](#_Toc91093216)

[**Введение** 4](#_Toc91093217)

[**1.** **Теоретическая часть** 5](#_Toc91093218)

[**Java** 5](#_Toc91093219)

[**Apache Maven** 6](#_Toc91093220)

[**Spring Framework** 7](#_Toc91093221)

[**Spring Core** 8](#_Toc91093222)

[**Spring MVC** 12](#_Toc91093223)

[**MyBatis** 14](#_Toc91093224)

[**2.** **Практическая часть** 14](#_Toc91093225)

[**Установка необходимых компонентов** 14](#_Toc91093226)

[**Создание проекта** 15](#_Toc91093227)

[**Models** 18](#_Toc91093228)

[**DAO** 19](#_Toc91093229)

[**Services** 24](#_Toc91093230)

[**Controllers** 26](#_Toc91093231)

[**Unit Tests** 28](#_Toc91093232)

[**Результаты проделанной работы** 32](#_Toc91093233)

[**Заключение** 34](#_Toc91093234)

[**Список использованных источников** 35](#_Toc91093235)

**Введение**

Люди всегда стремились упростить себе жизнь, изобретая новые фреймворки, позволяющие быстрее и эффективнее выполнять некоторую работу.

Фреймворк (неологизм от англ. framework - каркас, структура) - программная платформа, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.

Одной из таких технологий стал Spring. Spring Framework (или коротко Spring) — универсальный фреймворк с открытым исходным кодом. Spring обеспечивает решения многих задач, с которыми сталкиваются Java-разработчики и организации, которые хотят создать информационную систему, основанную на платформе Java.

Целью данного курсового проекта является изучить современные фреймворки стека Java технологии и продемонстрировать их возможности на примере разработки некоторого web-приложения.

Для достижения заданной цели необходимо решить следующие задачи:

1. Создать модели для временного хранения и перемещения данных из базы данных в приложение.
2. Создать слой для взаимодействия с базой данных.
3. Построить бизнес-логику в классах service.
4. Создать контроллеры для непосредственной обработки запросов от клиента и возвращения результатов.
5. Покрыть юнит тестами написанный код.
6. **Теоретическая часть**

**Java**

Java – строго типизированный объектно-орентированный язык программирования общего назначения.

Java разработала компания Sun Microsystems в начале 90-х годов XX века. Ведущую роль в создании языка сыграл канадский инженер Джеймс Гослинг (James Gosling). На ранних этапах разработки язык назывался Oak. Затем его переименовали в честь сорта кофе Java. Связь языка с напитком отражается в логотипе.

Джеймс Гослинг и его единомышленники хотели создать язык с си-подобным синтаксисом. В то же время он должен быть более простым по сравнению с C/C++. Создатели планировали использовать Java для программирования бытовой электроники. Однако практически сразу после выпуска версии 1.0 в 1995 язык стали использовать разработчики серверного и клиентского ПО[2].

Создатели реализовали принцип “Напиши один раз, запускай везде”. Благодаря чему программы написанные на java без перекомпиляции могут выполняться на различных платформах - Windows, Linux, Mac OS и т.д. Эта задача решается благодаря компиляции написанного на Java кода в байт-код. Этот формат исполняет JVM (Java Virtual Machine) или виртуальная машина Java. JVM — часть среды исполнения Java (JRE). Для каждой из платформ может быть своя реализация виртуальной машины JVM, но каждая из них может выполнять один и тот же код.

Java является языком с Си-подобным синтаксисом и близок в этом отношении к C/C++ и C#. Поэтому, если вы знакомы с одним из этих языков, то овладеть Java будет легче.

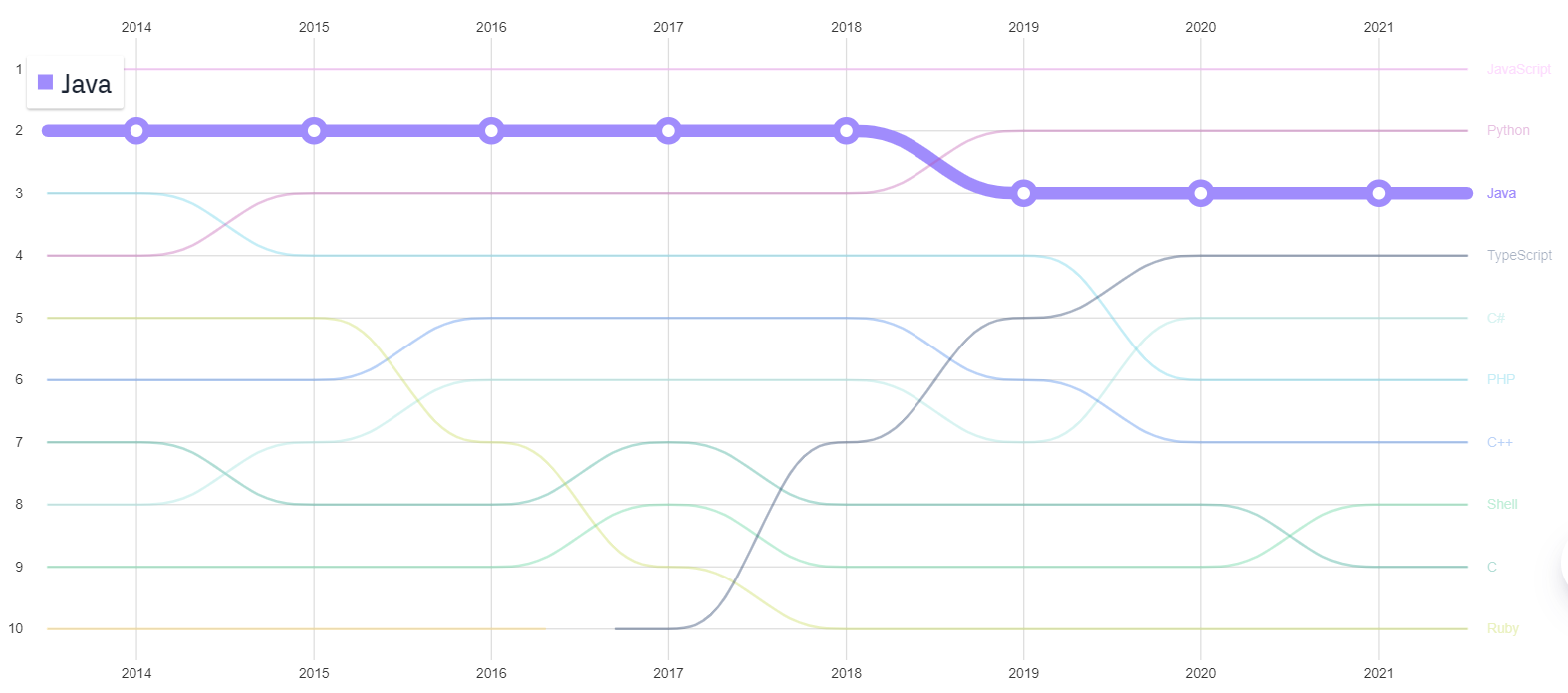
На сегодняшний момент язык Java является одним из самых распространенных и популярных языков программирования. По результатам рейтинга State of Octoverse 2021 Java входит в тройку самых популярых языков программирования. Java опережает по популярности PHP, C#, C++, TypeScript и другие востребованные языки, а уступает только JavaScript и Python. 

Рисунок 1.1 – Лучшие языки

**Apache Maven**

Apache Maven — фреймворк для автоматизации сборки проектов на основе описания их структуры в файлах на языке POM (англ. Project Object Model), являющемся подмножеством XML.

Собрать на Java проект уровня «Hello, world!» можно и с помощью командной строки. Но чем сложнее разрабатываемое ПО и чем больше оно использует сторонних библиотек и ресурсов, тем сложнее будет команда для сборки. Maven разработан для облегчения этой работы.

Одна из главных особенностей фреймворка — декларативное описание 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.

Ещё одно достоинство проекта — гибкое управление зависимостями. Maven умеет подгружать в свой локальный репозиторий сторонние библиотеки, выбирать необходимую версию пакета, обрабатывать транзитивные зависимости. Транзитивные зависимости, это цепочка A ← B ← C, где A зависит от В, а В от С. Maven позволяет не напрягаться по этому поводу, и если нужен именно А, указывать только его, об остальном он позаботится.

**Spring Framework**

Spring Framework (или коротко Spring) — универсальный фреймворк с открытым исходным кодом для Java-платформы. Считается, что Spring Framework реализует модель разработки, основанную на лучших стандартах индустрии, и делает её доступной во многих областях Java.

Первая версия была написана Родом Джонсоном, который впервые опубликовал её вместе с изданием своей книги «Expert One-on-One Java EE Design and Development» (Wrox Press, октябрь 2002 года).

Фреймворк был впервые выпущен под лицензией Apache 2.0 license в июне 2003 года. Первый стабильный релиз 1.0 был выпущен в марте 2004. Последующие стабильные релизы вышли в сентябре 2004 года и марте 2005 года.

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

Spring — это не один какой-то конкретный фреймворк. Это скорее общее названия для целого ряда небольших фреймворков, каждый из которых выполняет какую-то свою работу.

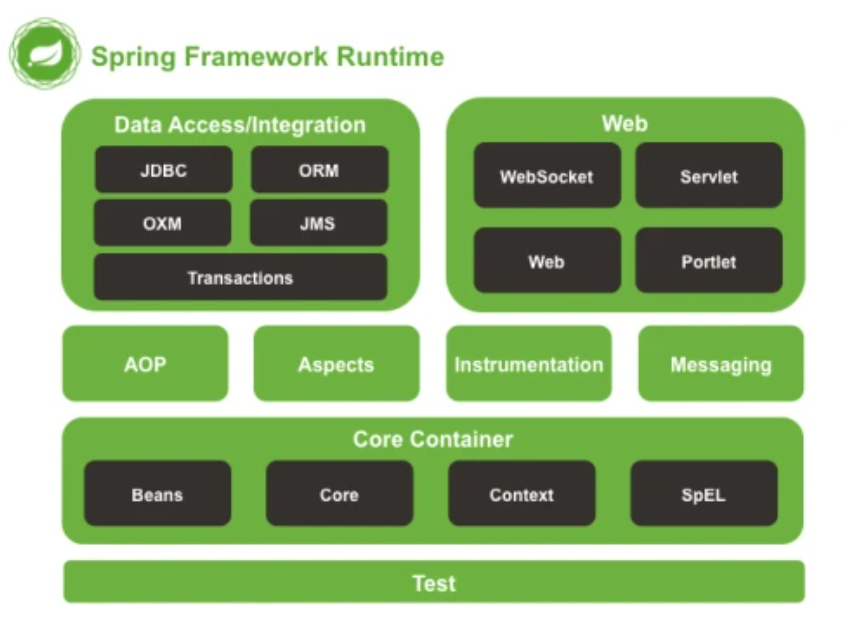


Рисунок 1.2 – Структура Spring Framework

Как видно, у spring модульная структура. Это позволяет подключать только те модули, что нам нужны для нашего приложения и не подключать те, которыми мы заведомо не будем пользоваться.

**Spring Core**

Spring Core - предназначен для конфигурирования приложения, состоящего из различных объектов различных классов. Конфигурация содержит как список объектов приложения, так и устанавливает связи между ними. Это позволяет разрабатывать классы для логики приложения, не беспокоясь о типах используемых баз данных, обеспечения безопасности и др.

Ядро Spring Framework основано на принципе инверсии управления. Инверсия управления (англ. Inversion of Control, IoC) — один из популярных принципов объектно-ориентированного программирования, при помощи которого можно снизить связанность между компонентами, а также повысить модульность и расширяемость ПО. Реализуется инверсия управления несколькими способами, среди которых есть внедрение зависимостей (Dependency Injection, DI) и поиск зависимостей (Dependency Lookup, DL).

Бины – это объекты, которые являются основой приложения и управляются Spring IoC контейнером.

В современных реалиях разработки бизнес-ориентированного программного обеспечения, когда оно разрабатывается не раз и навсегда, а разрабатывается, поддерживается и изменяется с течением времени, сильная связанность может серьёзно осложнять процесс разработки. По этой причине во многих современных фреймворках применяются те или иные реализации инверсии управления.

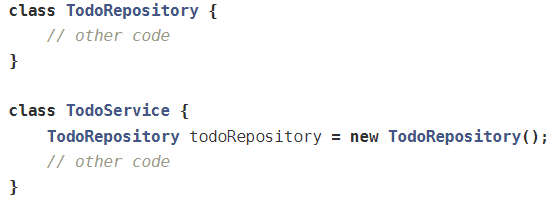


Рисунок 1.3 – Пример без применения инверсии управления

Как видно из примера, класс TodoService самостоятельно создаёт объект класса TodoRepository. Это образует сильную связь между классами TodoService и TodoRepository. Если в классе TodoService потребуется использование другого класса вместо TodoRepository, то придётся вносить соответствующие изменения в него.

Применение инверсии управления предполагает, что объект класса TodoRepository должен быть создан за пределами класса TodoService, но в дальнейшем должен быть передан объекту этого класса.

Первый способ это сделать — применить внедрение зависимостей. Внедрение зависимостей реализуется несколькими способами, среди которых можно выделить:

* Внедрение через конструктор.
* Внедрение через set-методы.
* Внедрение через интерфейс.
* Внедрение через свойства.



Рисунок 1.4 – Пример внедрения зависимостей через конструктор



Рисунок 1.5 – Пример внедрения зависимостей через set-метод

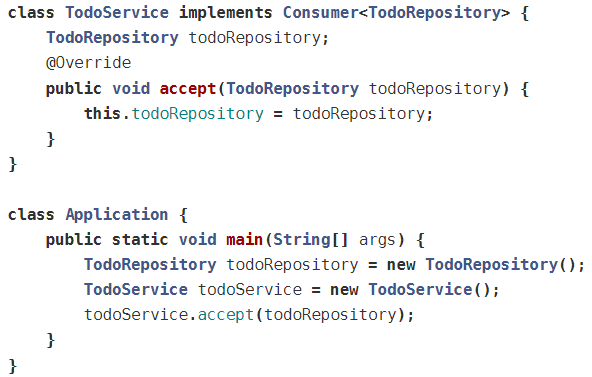


Рисунок 1.6 – Пример внедрения зависимостей через интерфейс

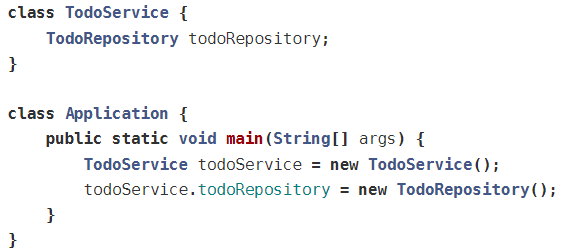


Рисунок 1.7 – Пример внедрения зависимостей через свойства

В случае с поиском зависимостей класс должен самостоятельно реализовывать логику получения зависимостей извне. Для этого он должен иметь доступ к некоему источнику зависимостей.

В приведённом ниже примере объект TodoRepository внедряется при помощи поиска зависимостей, а DependencyProvider — при помощи внедрения зависимостей через интерфейс.



Рисунок 1.8 – Пример поиска зависимостей

**Spring MVC**

Spring MVC — это веб-фреймворк Spring. Он позволяет создавать веб-сайты или RESTful сервисы и хорошо интегрируется в экосистему Spring.

Spring MVC предоставляет архитектуру модель-представление-контроллер и готовые компоненты, которые можно использовать для разработки гибких и слабо связанных веб-приложений. MVC — это шаблон программирования, который позволяет разделить логику приложения на три части:

* Model (модель). Получает данные от контроллера, выполняет необходимые операции и передаёт их в вид.
* View (вид или представление). Получает данные от модели и выводит их для пользователя.
* Controller (контроллер). Обрабатывает действия пользователя, проверяет полученные данные и передаёт их модели.

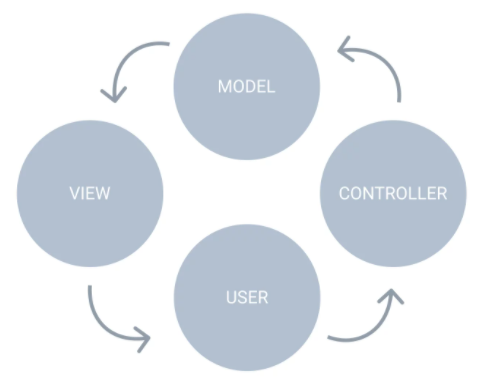


Рисунок 1.9 – MVC

Spring MVC разработана на основе DispatcherServlet, который обрабатывает все HTTP-запросы и ответы. Рабочий процесс обработки запросов показан на следующем рисунке.

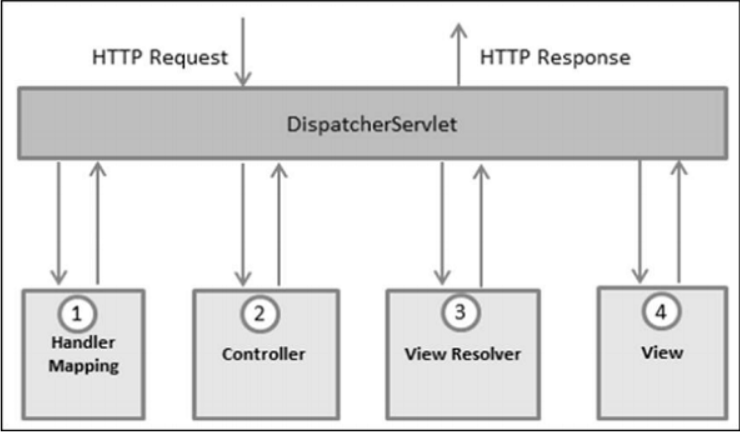


Рисунок 1.10 – Рабочий процесс обработки запросов

Spring MVC разработана на основе DispatcherServlet, который обрабатывает все HTTP-запросы и ответы. Рабочий процесс обработки запросов показан на следующем рисунке.

**MyBatis**

Фреймворк с открытым исходным кодом iBatis, первоначально разработанный Apache, был перенесен в код Google в 2010 году и переименован в MyBatis.

MyBatis - это фреймворк для работы с базой данных, он позволяет работать с простым SQL и хранимыми процедурами. Для его конфигурации может использовать XML или аннотации. Так же он без проблем интегрируется с Spring Framework и Google Guice.

В Java сообществе Hibernate framework считается стандартом для удобной работы с базой данных. Разработчику трудно выбрать другой фреймфорк, потому что порой он не знает о существовании альтернатив. Основное отличие MyBatis от Hibernate — это то как производится мапинг объектов. Hibernate мапит таблицы БД на сущности, давая нам доступ к данным. Для получения данных Hibernate генерирует SQL запросы, а генерируемые запросы хороши до поры — до времени, а потом они съедают кучу времени, становятся громоздкими и не управляемыми. MyBatis мапится не на таблицы, а на SQL запросы, за формирование запросов отвечает разработчик и только от него будет зависеть как быстро будет работать приложение.

1. **Практическая часть**

**Установка необходимых компонентов**

Первым делом, для разработки программ на языке программирования Java необходимо скачать комплект разработчика Java Development Kit (JDK). Сделать это можно на официальном сайте Java [www.oracle.com](http://www.oracle.com). Выбрав подходящую версию JDK, можно выбрать способ загрузки (windows installer, compress archive или др.). Скачав JDK, требуется прописать путь до папки его расположения в переменную среды JAVA\_HOME, а также в переменную Path путь до папки bin в каталоге JDK.

После установки JDK скачаем среду разработки, в которой будем писать весь последующий код. В качестве среды разработки была выбрана IntelliJ IDEA. IntelliJ IDEA можно скачать на сайте <https://www.jetbrains.com/idea/>. Скачать данную среду разработки можно бесплатно выбрав Community версию, однако для создания web-приложения рекомендуется выбрать Ultimate версию. IntelliJ IDEA Ultimate является платной программой, но существует много способов скачать ее бесплатно. Например, студентам IntelliJ IDEA предоставляется бесплатно на срок обучения в университете.

Далее скачаем и установим фреймворк Maven <https://maven.apache.org/download.cgi>. Также как и в случае с JDK прописываем переменную среды MAVEN\_HOME, а также в переменную Path путь до папки bin.

**Создание проекта**

Для создания проекта запускаем IntelliJ IDEA и выбираем New Project. В появившемся меню выбираем пункт Spring Initializr и заполняем поля.

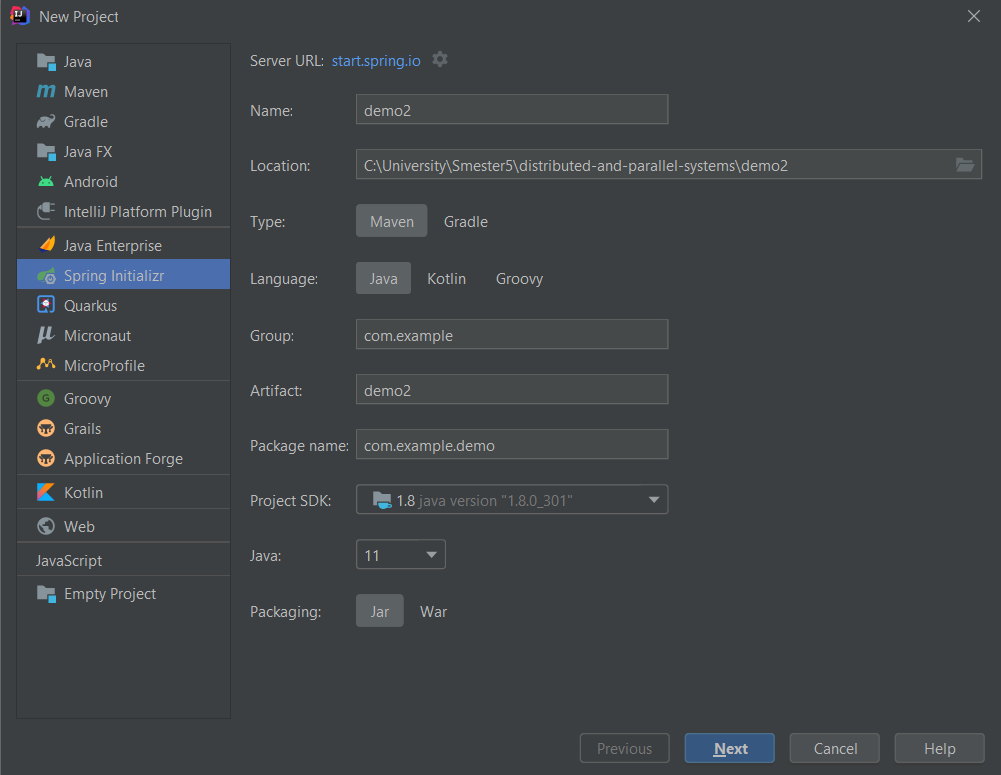


Рисунок 2.1 – Создание проекта

Затем выбираем нужные зависимости, в моем случае: Spring WEB и PostgreSQL Driver.

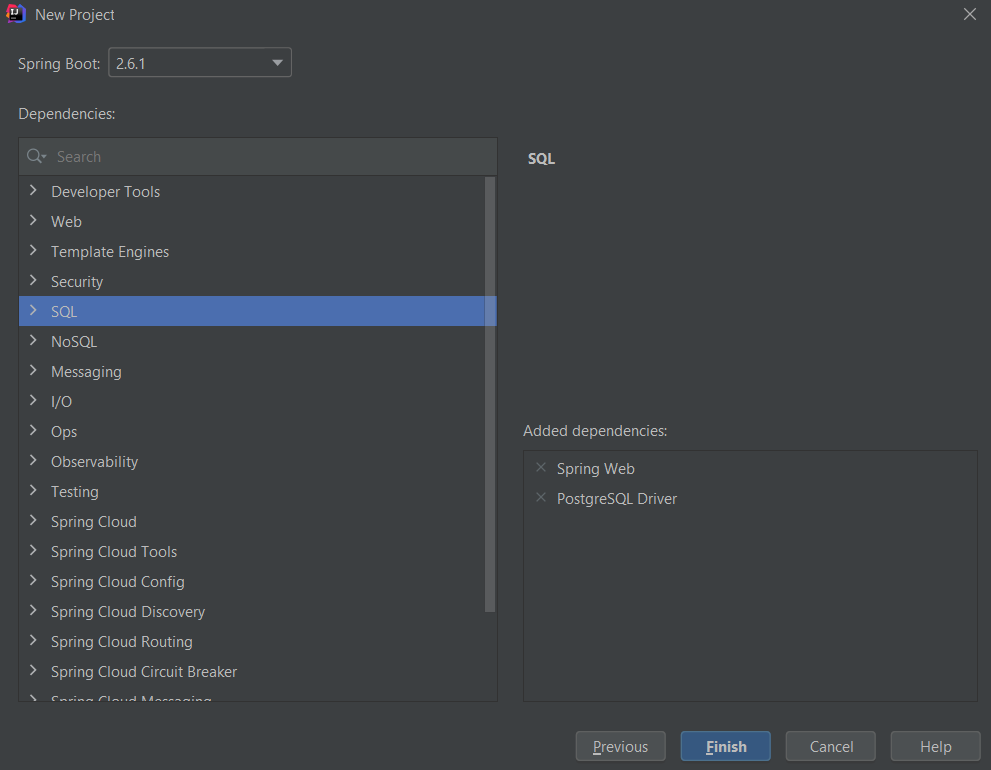


Рисунок 2.2 – Выбор зависимостей

После создания проекта мы видим сгенерированную IntelliJ IDEA проекта с нужными зависимостями.

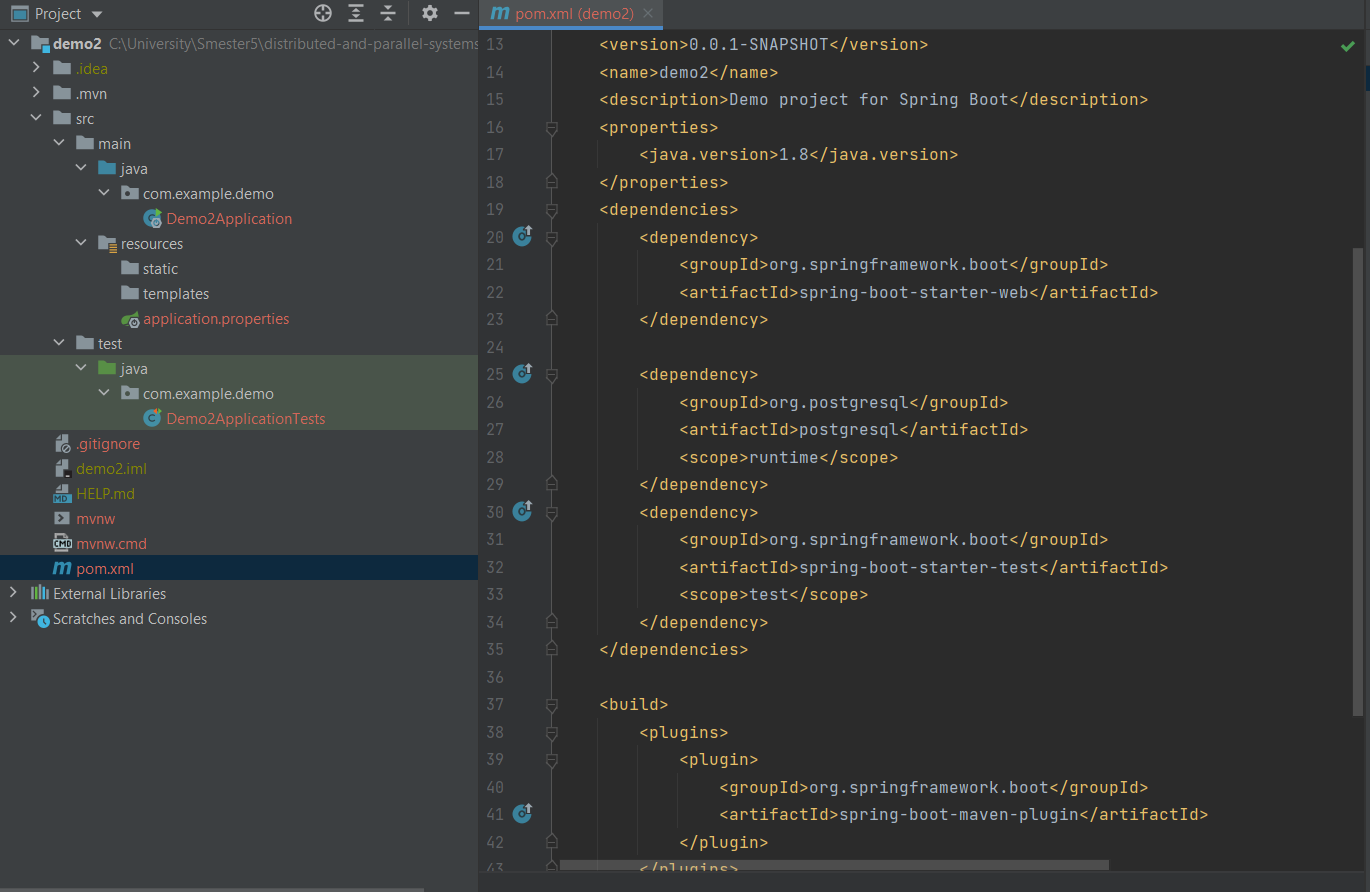


Рисунок 2.3 – Стартовый проект

Такого же результата можно добиться, если перейти на сайт <https://start.spring.io/>,заполнить поля, добавить нужные зависимости и нажать generate.

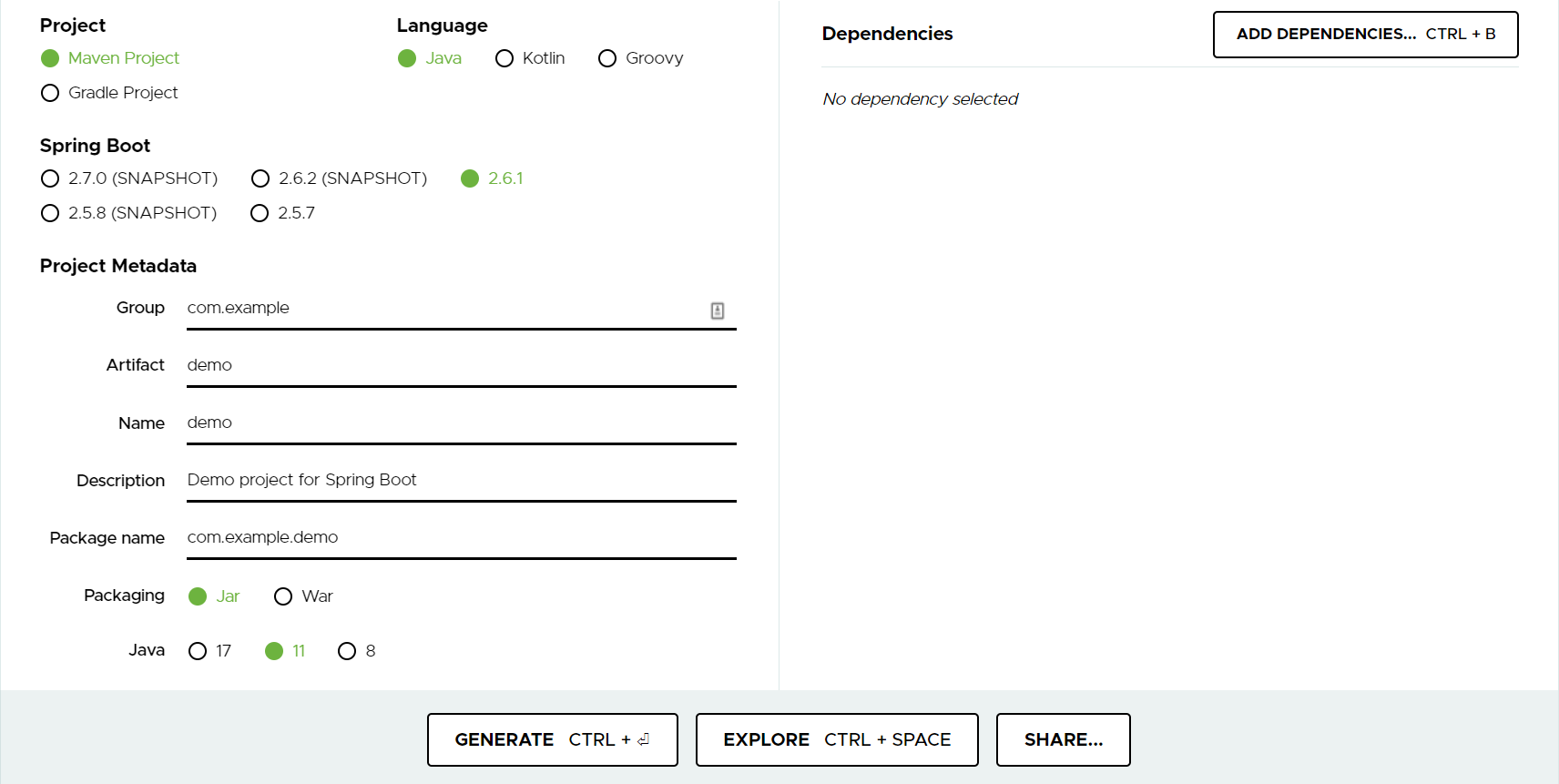


Рисунок 2.4 – Spring initializr

Далее нужно добавить недостающие зависимости. Для этого переходим на сайт <https://mvnrepository.com/>, находим там нужную зависимость рис 2.5. Затем вставляем выделенный текст в pom.xml в тег dependencies.

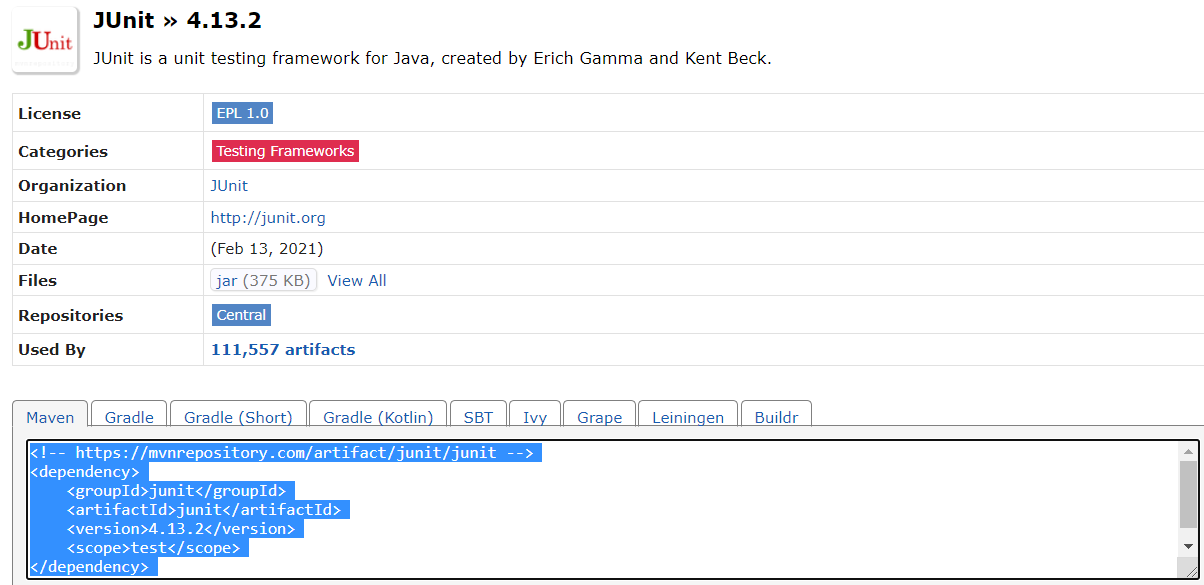


Рисунок 2.5 – Пример подключения зависимостей

**Models**

Models – это классы, которые хранят и передают данные во время работы программы. Данные классы не содержат методы обработки данных. Они могут содержать только геттеры, сеттеры, и конструкторы.

В текущем web-приложении я использовал семь основных классов, отражающие таблицы в базе данных: Restaurant, Review, Scheme, Table, TableReservation, User, UserInfo.

Листинг 2.1 – модель Review.

package by.vsu.model;  
  
import java.util.Date;  
import java.util.Objects;  
  
*/\*\*  
 \* Represents a review for a specific restaurant.  
 \*/*public class Review {  
 private Integer id;  
  
 */\*\* Number from 1 to 10 \*/* private int rate;  
  
 */\*\* Comment that leave user. \*/* private String text;  
  
 private Date date;  
  
 */\*\* User who left this review. \*/* private User user;  
  
 private Restaurant restaurant;  
  
 */\*\* constructors \*/  
 /\*\* getters and setters \*/* @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Review review1 = (Review) o;  
 return rate == review1.rate && Objects.*equals*(id, review1.id) && Objects.*equals*(text, review1.text) && Objects.*equals*(date, review1.date) && Objects.*equals*(user, review1.user) && Objects.*equals*(restaurant, review1.restaurant);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(id, rate, text, date, user, restaurant);  
 }  
  
 @Override  
 public String toString() {  
 return "Review{" +  
 "id=" + id +  
 ", rate=" + rate +  
 ", review='" + text + '\'' +  
 ", date=" + date +  
 ", user=" + user +  
 ", restaurant=" + restaurant +  
 '}';  
 }  
}

Аналогичным образом описаны остальные классы. Так же я создал вспомогательные классы:

* Position – хранит информацию о расположение стола в ресторане.
* Comparison – хранит знак < <= = >= >
* Вложенные классы: Role, Status в классе User; Status в классе TableReservation

**DAO**

Data access object (DAO) — абстрактный интерфейс к какому-либо типу базы данных или механизму хранения. Определённые возможности предоставляются независимо от того, какой механизм хранения используется и без необходимости специальным образом соответствовать этому механизму хранения. Этот шаблон проектирования применим ко множеству языков программирования, большей части программного обеспечения, нуждающемуся в хранении информации, и к большей части баз данных. Но традиционно этот шаблон связывают с приложениями на платформе Java Enterprise Edition, взаимодействующими с реляционными базами данных через интерфейс JDBC, потому что он появился в рекомендациях от фирмы Sun Microsystems[3].

Для реализации Dao я использовал MyBatis, что существенно упростило мне работу.

Рассмотрим в качестве примера UserDao:

Листинг 2.2 – модель UserDao.

package by.vsu.dao;  
  
import by.vsu.model.User;  
  
import java.util.List;  
  
public interface UserDao {  
 */\*\*  
 \* Find list of users.  
 \** ***@param*** *limit number of users to return.  
 \** ***@param*** *skip number of users to skip.  
 \** ***@return*** *list of users.  
 \*/* List<User> findAll(int limit, int skip);  
  
 */\*\*  
 \* Find user by its id.  
 \** ***@return*** *User.  
 \*/* User findById(Integer id);  
  
 */\*\*  
 \* Create new user.User id should be auto-generated.  
 \* Created User object saved in param user.  
 \** ***@param*** *user User data.  
 \*/* void create(User user);  
  
 */\*\*  
 \* Updates user using given data.  
 \** ***@param*** *user User data for update. Should have id set  
 \*/* void update(User user);  
  
 */\*\*  
 \* Deletes user by its id.  
 \** ***@param*** *id User id.  
 \** ***@return*** *number of deleted users.  
 \*/* int delete(Integer id);  
}

После описание основных методов, нужно сконфигурировать MyBatis, чтобы он реализовал этот интерфейс, для этого есть два способа:

* Через XML
* Через аннотации

Использование xml: В главный тег mapper в качестве атрибута namespace передается нужный интерфейс. Далее для того чтобы сказать MyBatisу как сопоставлять столбцы в sql с полями класса используется тет resulMap. У этого тега можно установить атрибут autoMapping="true" и тогда MyBatis будет делать это автоматически. Далее внутри тегов select, update, delete, insert пишутся sql запросы, а атрибут id совпадает с названием методов реализуемого интерфейса.

Листинг 2.3 – конфигурация MyBatis через xml

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper  
 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
  
<mapper namespace="by.vsu.dao.UserDao">  
 <resultMap id="userResultMap" type="User">  
 <id property="id" column="id"/>  
 <result property="login" column="login"/>  
 <result property="password" column="password"/>  
 <result property="role" column="role" javaType="by.vsu.model.User$Role"/>  
 <result property="status" column="status" javaType="by.vsu.model.User$Status"/>  
 </resultMap>  
  
 <select id="findAll"  
 resultMap="userResultMap"  
 parameterType="Integer"  
 >  
 SELECT id, login, password, role, status  
 FROM users  
 LIMIT #{limit} OFFSET #{skip};  
 </select>  
  
 <select id="findById"  
 resultMap="userResultMap"  
 parameterType="Integer">  
 SELECT id, login, password, role, status  
 FROM users  
 WHERE id = #{id}  
 </select>  
  
 <insert id="create"  
 parameterType="User"  
 useGeneratedKeys="true"  
 keyProperty="id">  
 INSERT INTO users (login, password, role, status)  
 VALUES (#{login}, #{password}, #{role}, #{status});  
 </insert>  
  
 <update id="update"  
 parameterType="User">  
 UPDATE users  
 SET login = #{login},  
 password = #{password},  
 role = #{role},  
 status = #{status}  
 WHERE id = #{id};  
 </update>  
  
 <delete id="delete"  
 parameterType="Integer">  
 DELETE  
 FROM users  
 WHERE id = #{id};  
 </delete>  
  
</mapper>

Использование аннотации: Для того чтобы добиться такого же результата что и с xml, нужно над каждым методом написать соответствующую аннотацию (select, update, delete, insert), а внутри этой аннотации sql запрос. Так же если этот метод возвращает объект, нужно добавить аннотацию Results. На рис 2.6 приведен пример.

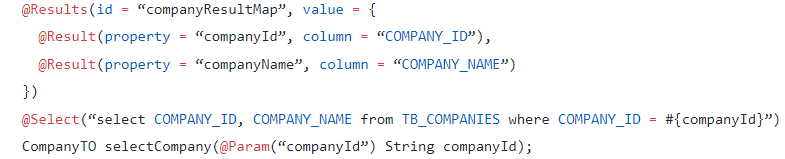


Рисунок 2.6 – Пример аннотации Results

Так же можно объединять xml и аннотации, к примеру, можно правило сопоставления полей объекта с столбцами в базе данных описать в xml, а сами запросы писать при помощи аннотаций (см. листинг 2.3, 2.4).

Листинг 2.3 – конфигурация MyBatis используя аннотации

package by.vsu.dao;  
  
import by.vsu.model.Review;  
import org.apache.ibatis.annotations.\*;  
  
import java.util.List;  
  
public interface ReviewDao {  
 @Select("SELECT id, rate, review, date, user\_id, restaurant\_id " +  
 "FROM reviews " +  
 "LIMIT #{size} OFFSET #{skip} ")  
 @ResultMap("reviewResultMap")  
 List<Review> findAll(@Param("size") int size, @Param("skip") int skip);  
  
 @Select("SELECT id, rate, review, date, user\_id, restaurant\_id " +  
 "FROM reviews " +  
 "WHERE id = #{id}")  
 @ResultMap("reviewResultMap")  
 Review findById(@Param("id") Integer id);  
  
 @Select("SELECT id, rate, review, date, user\_id, restaurant\_id " +  
 "FROM reviews " +  
 "WHERE restaurant\_id = #{restaurantId} " +  
 "LIMIT #{size} OFFSET #{skip} ")  
 @ResultMap("reviewResultMap")  
 List<Review> findByRestaurantId(@Param("restaurantId") Integer restaurantId,  
 @Param("size") int size,  
 @Param("skip") int skip);  
  
 @Select("SELECT id, rate, review, date, user\_id, restaurant\_id " +  
 "FROM reviews " +  
 "WHERE user\_id = #{userId} " +  
 "LIMIT #{size} OFFSET #{skip} ")  
 @ResultMap("reviewResultMap")  
 List<Review> findByUserId(@Param("userId") Integer userId,  
 @Param("size") int size,  
 @Param("skip") int skip);  
  
 @Insert("INSERT INTO reviews (rate, review, date, user\_id, restaurant\_id) " +  
 "VALUES(#{rate}, #{text}, #{date}, #{user.id}, #{restaurant.id}) ")  
 @Options(useGeneratedKeys = true, keyProperty = "id")  
 void create(Review review);  
  
 @Update("UPDATE reviews " +  
 "SET rate=#{rate}, review=#{text}, date=#{date}, user\_id=#{user.id}, restaurant\_id=#{restaurant.id}")  
 void update(Review review);  
  
 @Delete("DELETE FROM reviews " +  
 "WHERE id = #{id}")  
 int delete(@Param("id") Integer id);  
}

Листинг 2.4 – правило сопоставления полей объекта с столбцами в БД

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper  
 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="by.vsu.dao.ReviewDao">  
 <resultMap id="reviewResultMap" type="Review" autoMapping="true">  
 <result property="text" column="review"/>  
 <result property="user.id" column="user\_id"/>  
 <result property="restaurant.id" column="restaurant\_id"/>  
 </resultMap>  
</mapper>

А теперь хотелось бы показать, чтобы мне пришлось делать, если бы я использовал JDBC.

Листинг 2.5 – метод create используя JDBC

@Override  
public long create(User user) throws DaoException {  
 String sql = "INSERT INTO `user` (`login`, `password`, `name`, `role`) VALUES (?, ?, ?, ?)";  
 PreparedStatement statement = null;  
 ResultSet resultSet = null;  
 try {  
 statement = connection.prepareStatement(sql, Statement.*RETURN\_GENERATED\_KEYS*);  
 statement.setString(1, user.getLogin());  
 statement.setString(2, user.getPassword());  
 statement.setString(3, user.getName());  
 statement.setInt(4, user.getRole().ordinal());  
 statement.executeUpdate();  
  
 resultSet = statement.getGeneratedKeys();  
 resultSet.next();  
 return resultSet.getLong(1);  
 } catch (SQLException ex) {  
 throw new DaoException(ex);  
 } finally {  
 try {  
 resultSet.close();  
 } catch (Exception e) {  
 }  
 try {  
 statement.close();  
 } catch (Exception e) {  
 }  
 }  
}

Сравнив листинги 2.5 и 2.3 можно увидеть на сколько MyBatis упрощает написание кода.

**Services**

Слой services является буферным между DAO и контролерами. Его работа заключается в том, что он берет данные из DAO, обрабатывает их в соответствии с техническим заданием и передает контролерам, которые передают уже обработанную информацию пользователю.

Также, как и в предыдущем случае основные методы опишем в интерфейсах, что даст нам возможность не завесить от конкретной реализации. Рассмотрим UserService в качестве примера.

Листинг 2.6 – пример interface UserService

package by.vsu.service;  
  
import by.vsu.model.User;  
  
import java.util.List;  
  
public interface UserService {  
 */\*\*  
 \* Get list of users.  
 \* In case nothing was found, empty list is returned.  
 \** ***@param*** *pageSize Pagination param. Number of users to return on a page.  
 \** ***@param*** *pageNum Pagination param. Number of the page to return. Starts from 0.  
 \** ***@return*** *List of users.  
 \*/* List<User> getUsers(int pageSize, int pageNum);  
  
 */\*\*  
 \* Gets user by its id.  
 \** ***@param*** *id User id.  
 \** ***@return*** *User.  
 \*/* User getUserById(Integer id);  
  
 */\*\*  
 \* Create new user if user doesn't have id. Or update old user if it have id.  
 \** ***@param*** *user User data for save.  
 \** ***@return*** *Saved user object.  
 \*/* User save(User user);  
  
 */\*\*  
 \* Delete user by its id.  
 \** ***@param*** *id User id.  
 \** ***@return*** *Flag that shows whether user has been deleted.  
 \*/* boolean delete(Integer id);  
}

После описание интерфейсов для каждой модели, начнем писать их реализацию. Рассмотри в качестве примера UserServiceImpl.

Листинг 2.7 – пример UserServiceImpl

package by.vsu.service.logic;  
  
import by.vsu.dao.UserDao;  
import by.vsu.model.User;  
import by.vsu.service.UserService;  
import by.vsu.service.exception.NotFoundException;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import java.util.List;  
  
@Service  
public class UserServiceImpl implements UserService {  
 private UserDao userDao;  
  
 @Autowired  
 public UserServiceImpl(UserDao userDao) {  
 this.userDao = userDao;  
 }  
  
 @Override  
 public List<User> getUsers(int pageSize, int pageNum) {  
 return userDao.findAll(pageSize, pageNum \* pageSize);  
 }  
 @Override  
 public User getUserById(Integer id) {  
 return userDao.findById(id);  
 }  
 @Override  
 public User save(User user) {  
 if (user.getId() != null) {  
 User temp = userDao.findById(user.getId());  
 if (temp == null) {  
 throw new NotFoundException("there is not user with id = " + user.getId());  
 }  
 userDao.update(user);  
 } else {  
 userDao.create(user);  
 }  
 return user;  
 }

@Override  
 public boolean delete(Integer id) {  
 return userDao.delete(id) != 0;  
 }  
}

**Controllers**

Контроллер - это класс, предназначенный для непосредственной обработки запросов от клиента и возвращения результатов.

Чаще всего, в самом контроллере не описывается логика обработки данных. Создаются дополнительные, сервисные классы, которые выполняют все основные задачи по обработке данных. Задача методов в контроллере - вызов сервисных методов в нужном порядке и возвращение результатов клиенту.

Рассмотрим на примере ReviewController

Листинг 2.8 – ReviewController

package by.vsu.controller;  
  
import by.vsu.model.Review;  
import by.vsu.service.ReviewService;  
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("/reviews")  
public class ReviewController {  
 private ReviewService reviewService;  
  
 @Autowired  
 public void setReviewService(ReviewService reviewService) {  
 this.reviewService = reviewService;  
 }  
  
 @GetMapping  
 public List<Review> all(@RequestParam(value = "pageSize", defaultValue = "10") int pageSize,  
 @RequestParam(value = "pageNum", defaultValue = "0") int pageNum) {  
 return reviewService.getReviews(pageSize, pageNum);  
 }  
  
 @GetMapping("/restaurant/{id}")  
 public List<Review> allByRestaurantId(@PathVariable("id") Integer restaurantId,  
 @RequestParam(value = "pageSize", defaultValue = "10") int pageSize,  
 @RequestParam(value = "pageNum", defaultValue = "0") int pageNum) {  
 return reviewService.getReviewsByRestaurantId(restaurantId, pageSize, pageNum);  
 }  
  
 @GetMapping("user/{id}")  
 public List<Review> allByUserId(@PathVariable("id") Integer userId,  
 @RequestParam(value = "pageSize", defaultValue = "10") int pageSize,  
 @RequestParam(value = "pageNum", defaultValue = "0") int pageNum) {  
 return reviewService.getReviewsByUserId(userId, pageSize, pageNum);  
 }  
  
 @PostMapping()  
 public ResponseEntity<Review> newReview(@RequestBody Review review) {  
 ResponseEntity<Review> responseEntity = null;  
 if (review.getId() == null) {  
 Review created = reviewService.save(review);  
 responseEntity = new ResponseEntity(created, HttpStatus.*CREATED*);  
 } else {  
 responseEntity = new ResponseEntity<>(HttpStatus.*CONFLICT*);  
 }  
 return responseEntity;  
 }  
  
 @PutMapping  
 public ResponseEntity<Review> updateReview(@RequestBody Review review) {  
 ResponseEntity<Review> responseEntity;  
 if (review.getId() != null) {  
 Review updated = reviewService.save(review);  
 responseEntity = new ResponseEntity<Review>(updated, HttpStatus.*OK*);  
 } else {  
 responseEntity = new ResponseEntity<>(HttpStatus.*BAD\_REQUEST*);  
 }  
 return responseEntity;  
 }  
  
 @DeleteMapping("/{id}")  
 public ResponseEntity<Review> delete(@PathVariable("id") Integer id) {  
 ResponseEntity<Review> responseEntity = null;  
 if (reviewService.delete(id)) {  
 responseEntity = new ResponseEntity<>(HttpStatus.*NO\_CONTENT*);  
 } else {  
 responseEntity = new ResponseEntity<>(HttpStatus.*NOT\_FOUND*);  
 }  
 return responseEntity;  
 }  
  
}

Контроллеры в Spring помечаются аннотацией Controller или RestController. Это позволяет спрингу автоматически находить бины. RestController используется, если методы возвращают объекты, а не путь к отображению. Аннотация RequestMapping расположенная над классом задает базовый путь для этого контроллера. GetMapping, PutMapping и т.д. говорят спрингу, что при ходе соответсвующего http запроса вызвать этот метод.

**Unit Tests**

Модульное тестирование или юнит-тестирование (англ. unit testing) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Для тестирования Dao уровня я использовал тестовую БД с заранее известным размером и содержимым. Аннотация Transactional позволяет отменить изменения в БД после выполнения теста.

Листинг 2.9 – пример нескольких тестов из ReviewDaoTest

@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration(classes = AppConfig.class)  
public class ReviewDaoTest {  
 private final static int *REVIEWS\_SIZE* = 5;  
  
 private ReviewDao reviewDao;  
  
 @Autowired  
 public void setReviewDao(ReviewDao reviewDao) {  
 this.reviewDao = reviewDao;  
 }  
  
 @Test  
 public void findAllSizeTest() {  
 List<Review> reviews = reviewDao.findAll(*REVIEWS\_SIZE*, 0);  
 *assertEquals*(*REVIEWS\_SIZE*, reviews.size());  
 }  
  
 @Test  
 @Transactional  
 public void deleteChangeSizeTest() {  
 *assertEquals*("start size doesn't match",  
 *REVIEWS\_SIZE*,  
 reviewDao.findAll(*REVIEWS\_SIZE*, 0).size());  
  
 Integer id = 1;  
 reviewDao.delete(id);  
 int expectedSize = *REVIEWS\_SIZE* - 1;  
 *assertEquals*("size after delete review doesn't match",  
 expectedSize,  
 reviewDao.findAll(expectedSize, 0).size());  
 }  
}

Для тестирования уровень сервисов я использовал Mockito framework. Он упрощает тестирования, создавая мок объекты.

Mock-объект представляет собой конкретную фиктивную реализацию интерфейса, предназначенную исключительно для тестирования.

Для начала нужно создать мок объект и внедрить его в тестируемый класс. За это отвечают две аннотации: Mock и InjectMocks. MockitoAnnotations.openMock инициализирует поля, помеченные аннотациями.

Далее перед вызовом метода у тестируемого объекта нужно описать логику мок объекта (смотри листинг 2.10).

Листинг 2.10 – пример нескольких тестов из ReviewServiceImplTest

@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration(classes = TestConfig.class)  
public class ReviewServiceImplTest implements ApplicationContextAware {  
 @Mock  
 private ReviewDao reviewDao;  
  
 @InjectMocks  
 private ReviewServiceImpl reviewService;  
  
 private ApplicationContext context;  
  
 @Autowired  
 @Qualifier("reviews")  
 private List<Review> reviews;  
  
 @Override  
 public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {  
 this.context = applicationContext;  
 }  
  
 @Before  
 public void setUp() throws Exception {  
 MockitoAnnotations.*openMocks*(this);  
 }  
  
 @Test  
 public void getReviewsTest() {  
 *when*(reviewDao.findAll(*anyInt*(), *anyInt*())).thenReturn(reviews);  
  
 List<Review> actual = reviewService.getReviews(reviews.size(), 0);  
 *assertEquals*(reviews, actual);  
 }  
  
 @Test  
 public void saveNewObjectTest() {  
 Integer id = 1;  
 Review review = getReview();  
 *doAnswer*(i -> {  
 Review r = i.getArgument(0);  
 r.setId(id);  
 return null;  
 }).when(reviewDao).create(*any*(Review.class));  
  
 Review actual = reviewService.save(review);  
 review.setId(id);  
 *assertEquals*(review, actual);  
 }  
  
 @Test  
 public void saveExistedObjectTest() {  
 Integer id = 1;  
 Review review = getReview();  
 review.setId(id);  
 *when*(reviewDao.findById(*anyInt*())).thenReturn(review);  
  
 Review actual = reviewService.save(review);  
 *assertEquals*(review, actual);  
 }  
  
 @Test(expected = NotFoundException.class)  
 public void saveNotExistedObjectTest() {  
 *when*(reviewDao.findById(*anyInt*())).thenReturn(null);  
 reviewService.save(new Review(1));  
 }  
  
 @Test(expected = NotAllowedChangeException.class)  
 public void saveExistedObjectNotAllowedChangeTest() {  
 Review newReview = getReview();  
 newReview.setId(1);  
 Review oldReview = getReview();  
 oldReview.setRestaurant(new Restaurant(111));  
  
 *when*(reviewDao.findById(*anyInt*())).thenReturn(oldReview);  
 reviewService.save(newReview);  
 }  
}

Класс MockMvc предназначен для тестирования контроллеров. Он позволяет тестировать контроллеры без запуска http-сервера. То есть при выполнении тестов сетевое соединение не создается. Для работы mockMvc необходимо класс с тестами пометить аннотацией @WebMvcTest и передать в качестве параметра тестируемый контроллер.

Листинг 2.11 – пример нескольких тестов из ReviewControllerTest

@RunWith(SpringJUnit4ClassRunner.class)  
@WebMvcTest(ReviewController.class)  
@Import(TestConfig.class)  
public class ReviewControllerTest implements ApplicationContextAware {  
 @Autowired  
 private MockMvc mockMvc;  
  
 @MockBean  
 private ReviewService reviewService;  
  
 @Autowired  
 @Qualifier("reviews")  
 private List<Review> reviews;  
  
 @Autowired  
 private ObjectMapper mapper;  
  
 private ApplicationContext context;  
  
 @Override  
 public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {  
 context = applicationContext;  
 }  
  
 @Before  
 public void setUp() throws Exception {  
 MockitoAnnotations.*openMocks*(this);  
 }  
  
 @Test  
 public void allTest() throws Exception {  
 *when*(reviewService.getReviews(*anyInt*(), *anyInt*())).thenReturn(reviews);  
  
 mockMvc.perform(*get*("/reviews"))  
 .andExpect(*status*().isOk())  
 .andExpect(*content*().json(mapper.writeValueAsString(reviews)));  
 }  
  
   
 @Test  
 public void updateNotAllowedChangeTest() throws Exception{  
 Review review = getReview();  
 review.setId(1);  
  
 *when*(reviewService.save(*any*())).thenThrow(new NotAllowedChangeException());  
  
 mockMvc.perform(*put*("/reviews")  
 .content(mapper.writeValueAsString(review))  
 .contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect(*status*().isForbidden());  
 }  
  
 @Test  
 public void updateNotFoundTest() throws Exception{  
 Review review = getReview();  
 review.setId(1);  
  
 *when*(reviewService.save(*any*())).thenThrow(new NotFoundException());  
  
 mockMvc.perform(*put*("/reviews")  
 .content(mapper.writeValueAsString(review))  
 .contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect(*status*().isNotFound());  
 }  
  
  
 @Test  
 public void deleteNoContentTest() throws Exception {  
 *when*(reviewService.delete(*anyInt*())).thenReturn(true);  
  
 mockMvc.perform(*delete*("/reviews/1"))  
 .andExpect(*status*().isNoContent());  
 }  
  
 @Test  
 public void deleteNotFoundTest() throws Exception {  
 *when*(reviewService.delete(*anyInt*())).thenReturn(false);  
  
 mockMvc.perform(*delete*("/reviews/1"))  
 .andExpect(*status*().isNotFound());  
 }  
}



Рисунок 2.7 – Запуск всех тестов

**Результаты проделанной работы**

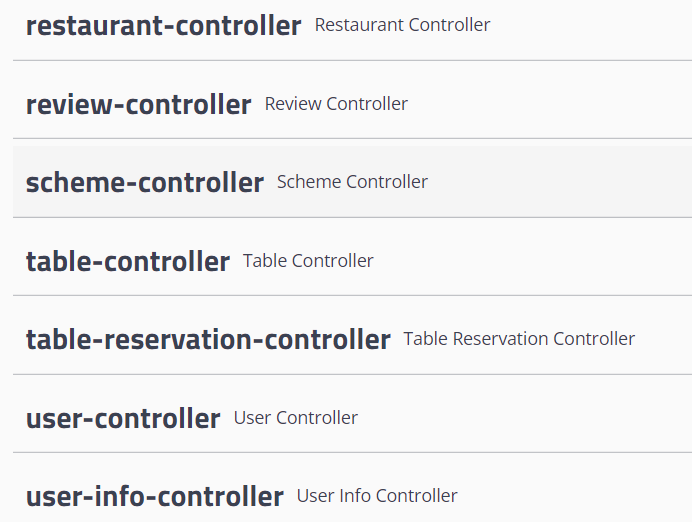


Рисунок 2.8 – Список контроллеров

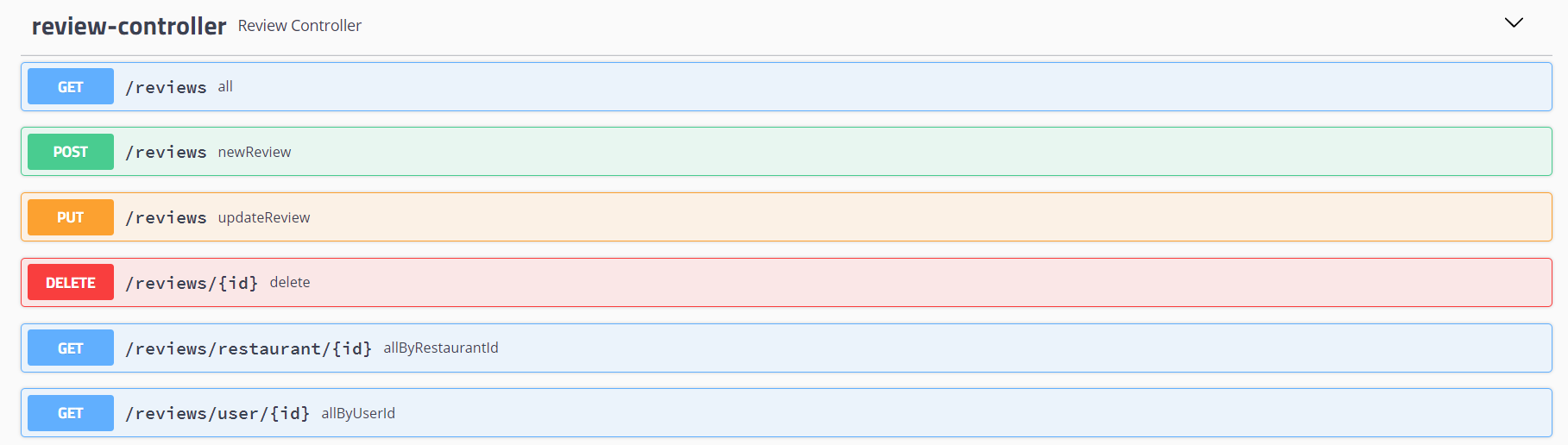


Рисунок 2.9 – API для review controller

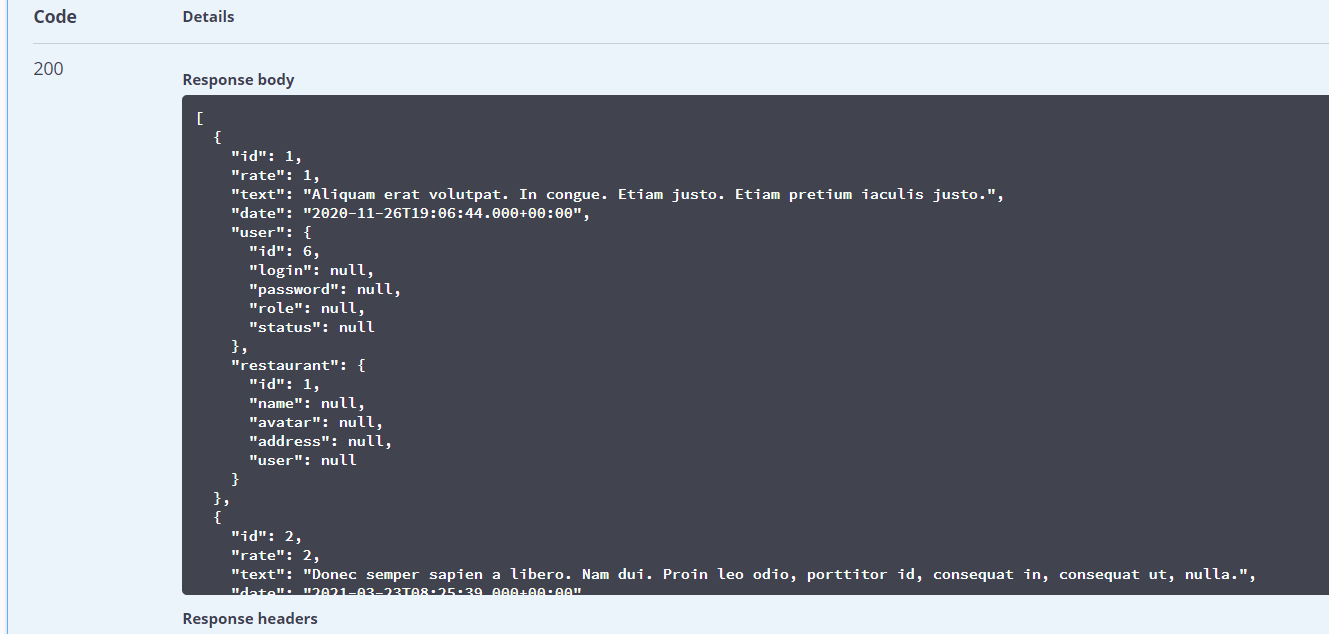


Рисунок 2.10 – результат get запроса <http://localhost/reviews>



Рисунок 2.11 – результат post запроса <http://localhost/reviews>



Рисунок 2.12 – результат delete запроса http://localhost/reviews/7

**Заключение**

В ходе проделанной работы были получены теоретические и практические знания по разработке web-приложений на языке программирования Java с использованием современных фреймворков. Полученные знания были успешно применены на практике. В процессе выполнения курсовой работы было разработано web-приложение функционал которого соответствует предметной области.

Были применены знания, полученные в процессе изучения курсов «Компьютерные сети», «Проектирование программных систем», «Методы алгоритмизации и программирования», «Современные пострелляционные базы данных».

В процессе выполнения представленной курсовой работы были решены следующие задачи:

1. Созданы модели для временного хранения и перемещения данных из базы данных в приложение.
2. Создан слой для взаимодействия с базой данных.
3. Созданы сервисы.
4. Созданы контролеры.
5. Код покрыт юнит тестами.

Считаю, что цель, поставленная в введении, была достигнута.

**Список использованных источников**

1. Mentanit ­ сайт о программировании для программистов [Электронный ресурс] – Режим доступа Дата доступа: 01.06.2021
2. Хекслет [Электронный ресурс] – Режим доступа https://ru.hexlet.io/blog/posts/yazyk-programmirovaniya-java-osobennosti-populyarnost-situatsiya-na-rynke-truda. – Дата доступа 18.12.2021..
3. Википедия- электронный ресурс [Электронный ресурс] – Режим доступа <https://ru.wikipedia.org/wiki/Data_Access_Object>. Дата доступа: 12.22.2021
4. Spring in Action, 5th Edition.