Создание программного обеспечения

Пальчевский Евгений Владимирович

Кандидат технических наук

Доцент кафедры индустриального программирования

О курсе

Группы в потоке: ЭФБО-07-24 – ЭФБО-12-24

Что ждет в зимнем семестре 2025/26 учебного года?

1. **Лекции (8 штук) и семинары-практики (32 штуки).**
2. **Выполнение индивидуальных практических заданий в рамках курсового проекта (курсовой работы).**
3. **За все получаем баллы в соответствии с балльно-рейтинговой системой (БРС).**
4. **Экзамен + курсовой проект.**

Посещения – 30 баллов. Экзамен – 30 баллов. Текущий контроль (то, что будем делать в течение семестра) – 40 баллов:

Наименование	Формат + защита	Даты проведения	Баллы
Контрольная работа №1	Очно (на занятиях)	09.02.2026 – 28.02.2026	5
Контрольная работа №2	Очно (на занятиях)	02.03.2026 – 31.03.2026	8
Контрольная работа №3	Очно (на занятиях)	01.04.2026 – 30.04.2026	5
Контрольная работа №4	Очно (на занятиях)	01.05.2026 – 31.05.2026	7
Контрольная работа №5	Очно (на занятиях)	01.05.2026 – 31.05.2026	5
Тестирование	Дистанционно (СДО)	01.05.2026 – 31.05.2026	10
Итого за семестр			**40**

Если сумма «Посещения + Текущий контроль» < 10, то студент до экзамена не допускается. Задания для КР на последнем слайде презентации.

Создание программного обеспечения

Пальчевский Евгений Владимирович

Кандидат технических наук

Доцент кафедры индустриального программирования

Оценки за экзамен

Количество баллов	Оценка
Менее 40	2 (неудовлетворительно)
От 40 включительно и до 60 включительно	3 (удовлетворительно)
От 60 включительно и до 80 включительно	4 (хорошо)
От 80 включительно и выше	5 (отлично)

Условия допуска к экзамену

Студент допускается к экзамену только при одновременном выполнении следующих условий:

1. **Успешно пройдены все мероприятия текущего контроля,** предусмотренные рабочей программой дисциплины.
2. **Защищен курсовой проект,** предусмотренный учебным планом в текущем семестре.

В ином случае студент до экзамена НЕ допускается.

Основание: *положение о текущем контроле успеваемости и промежуточной аттестации* (СМКО МИРЭА 7.5.1/03.П.08-19)**, п. 3.8**.

Создание программного обеспечения

Пальчевский Евгений Владимирович

Кандидат технических наук

Доцент кафедры индустриального программирования

О курсовом проекте

Краткие инструкции. Необходимо:

1. **Ознакомиться с основными положениями о выполнении курсового проекта.**
2. **Ознакомиться с методическими указаниями и требованиями к оформлению отчета – расчетно-пояснительной записки (РПЗ).**
3. **Разбиться на команды и выбрать тему курсового проекта (вкладка СПО МИРЭА). В таблице вписать свои ФИО и группу.**

3.1. Курсовую работу можно выполнять одному или в команде (максимум 5 человек).

Создание программного обеспечения

Пальчевский Евгений Владимирович

Кандидат технических наук

Доцент кафедры индустриального программирования

Нарушения дисциплины на занятии

При нарушениях, мешающих проведению занятия (игнорирование законных требований преподавателя, неуважительное поведение, действия, отвлекающие группу и нарушающие организацию/безопасность учебного процесса), преподаватель оформляет служебную записку (докладную) на имя руководителя подразделения.

По подтвержденному факту нарушения в порядке, предусмотренном п. 4.3.2.5 Временного положения о БРС (приказ РТУ МИРЭА от 19.01.2026 № 66):

1. Баллы за посещение/участие в данном занятии не начисляются (аннулируются) — фактически «о баллов за занятие».
2. Право на «автомат» (получение оценки по дисциплине без прохождения мероприятия семестрового контроля) не применяется к студенту.
3. При необходимости ставится вопрос о дисциплинарной ответственности (в установленном порядке).

Базы данных для индустриального программирования

Пальчевский Евгений Владимирович

Кандидат технических наук

Доцент кафедры индустриального программирования

Инструкции и материалы

Ответы на контрольные загружаем сюда (проекты либо ссылки на гит):

Максимальная сумма баллов по мероприятиям текущего контроля по дисциплине в течение учебного семестра			40
Наименование	Формат	Даты проведения	Баллы
Контрольная работа №1	Очно (на занятиях)	3 неделя	5
Контрольная работа №2	Очно (на занятиях)	6 неделя	8
Контрольная работа №3	Очно (на занятиях)	9 неделя	5
Контрольная работа №4	Очно (на занятиях)	12 неделя	7
Контрольная работа №5	Очно (на занятиях)	15 неделя	5
Тестирование	СДО	16 неделя	10

После загрузки ответов показываем работоспособность программы

Социальная сеть / мессенджер / почта	Контакт
Электронная почта	teelxp@inbox.ru
VK	https://vk.com/teelxp
WhatsApp	+7-937-485-80-48
YouTube-канал с видеолекциями	https://www.youtube.com/@teelxp
Лекции в ВК (альтернатива ютубчику)	https://vk.com/finkablog

Базы данных для индустриального программирования

Пальчевский Евгений Владимирович

Кандидат технических наук

Доцент кафедры индустриального программирования

Софт и языки программирования

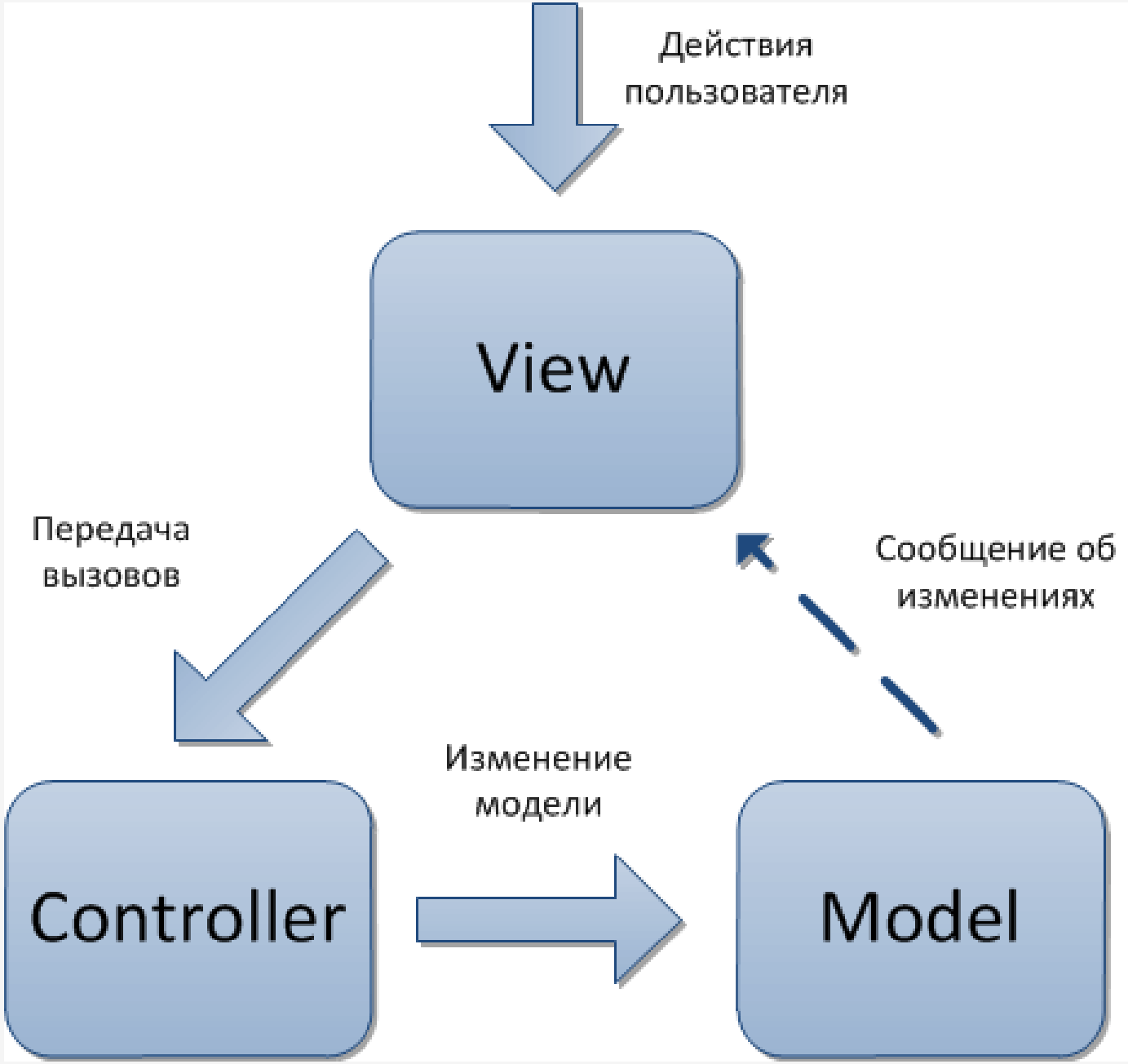
Логотип	Название	Группы
	IntelliJ IDEA Ultimate	https://palchevsky.ru/materials.php
	Visual Studio Code	https://code.visualstudio.com/
	PostgreSQL	https://www.postgresql.org/download/
	PyCharm	https://palchevsky.ru/materials.php

Язык программирования + SQL + Фреймворк	Иное
Python + SQL + Django	Либо используйте свою связку, предварительно согласованную с преподавателем.
Java + SQL + Spring	
C# + SQL + ASP.NET Core	
C++ + SQL + Drogon	

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»



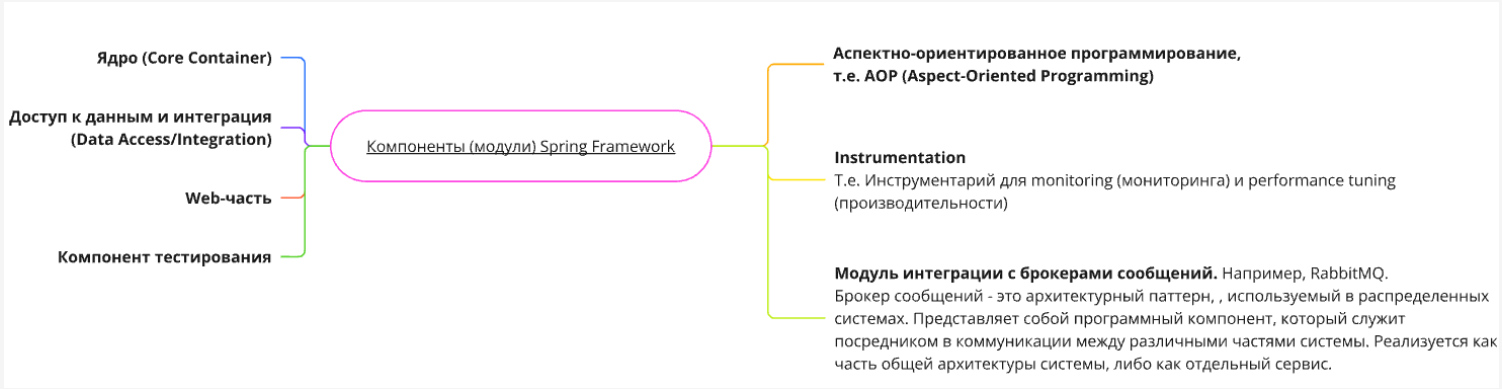
Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Немного о Spring

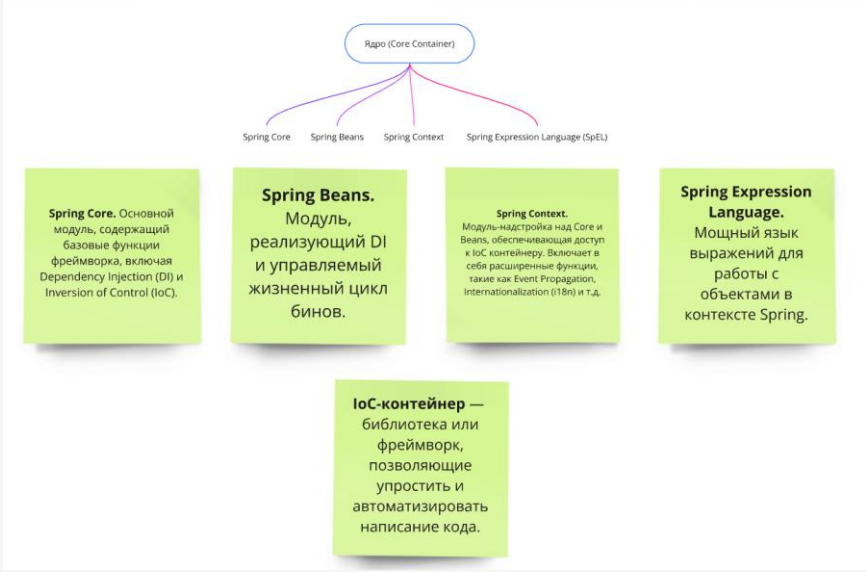
Spring – фреймворк с открытым исходным кодом для языка программирования Java. Он был создан для упрощения разработки и поддержки масштабируемых, слабосвязанных и повторно используемых приложений. Фреймворк нужен, чтобы разработчикам было легче проектировать и создавать приложения.



Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»



Spring Beans. Этот модуль отвечает за так называемые «бины» — обычные Java-объекты, которыми управляет Spring. Он определяет, как объект создать, как подставить в него зависимости (например, сервису — репозиторий), и как корректно завершить его работу. То есть Spring Beans — это управление жизненным циклом объектов и их «сборкой» внутри контейнера.

Spring Context. Это «контейнер с дополнительными возможностями» поверх Core и Beans. Он дает удобный доступ к объектам приложения через ApplicationContext, умеет автоматически находить компоненты (@Component, @Service, @Controller), поддерживает события, настройки приложения и другие полезные механизмы. Проще: Context делает контейнер Spring удобным и пригодным для реальных проектов.

Ядро (Core Container). Это основная часть Spring, которая «управляет объектами» в приложении. Вместо того чтобы создавать классы вручную через new, вы описываете компоненты, а Spring сам их создает, хранит и выдает там, где они нужны. Именно здесь реализована главная идея Spring: приложение собирается из независимых блоков, которые контейнер соединяет между собой.

Spring Core. Это «движок» контейнера: набор базовых механизмов, благодаря которым Spring вообще умеет работать. В нем находятся ключевые принципы IoC и DI: Spring решает, какие объекты создать, в каком порядке и как передать их друг другу. Проще говоря, Spring Core — это фундамент, без которого все остальные модули Spring не заработают.

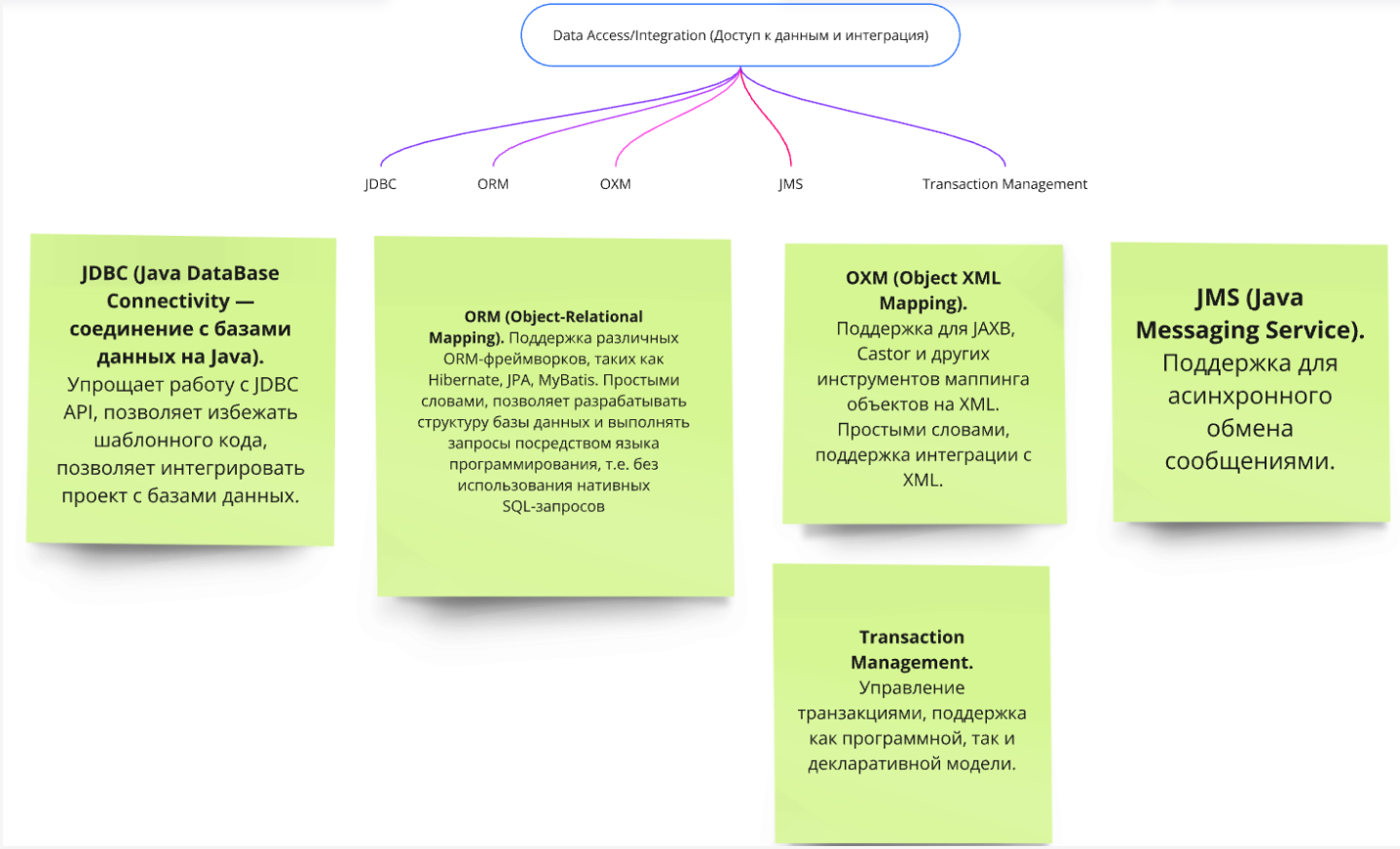
Spring Expression Language (SpEL). Это язык выражений внутри Spring, который позволяет писать условия и подстановки прямо в аннотациях и настройках. Например, можно подставлять значения из конфигурации, выбирать нужные данные по условию или задавать правила доступа. Проще: SpEL нужен, чтобы «умно» настраивать поведение приложения без лишнего кода.

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Немного о Spring. Data Access/Integration



Data Access / Integration (доступ к данным и интеграция). Это часть Spring, которая отвечает за работу приложения с внешним миром: базами данных, XML-обменом, очередями сообщений и транзакциями. Идея простая: Spring дает «единый удобный способ» подключаться к данным и не писать каждый раз одно и то же вручную (подключения, обработка ошибок, шаблонные куски кода). В реальных проектах это как раз то, без чего нормальный сервер почти не живет: нужно читать/писать в БД, гарантировать целостность данных и иногда общаться через сообщения.

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

JDBC (Java DataBase Connectivity). Это самый «низкий» и прямой способ работать с реляционной БД в Java: вы пишете SQL-запросы сами, получаете результат и руками превращаете строки таблицы в объекты. Spring упрощает JDBC тем, что убирает типовую рутину: открытие/закрытие соединений, обработку исключений, повторяющийся код. Проще говоря: JDBC — это когда вы управляете SQL напрямую, а Spring помогает делать это аккуратнее и короче.

ORM (Object-Relational Mapping). Это подход «объекты вместо таблиц»: вы описываете сущности (например, Student, Course), а ORM (чаще всего JPA/Hibernate) сам связывает их с таблицами и делает операции чтения/записи. Вам не нужно каждый раз писать SQL для типовых задач — часто достаточно вызвать методы репозитория, а ORM под капотом сформирует запрос. Проще: ORM позволяет думать в терминах Java-объектов, а не строк SQL (хотя иногда SQL все равно нужен для сложных случаев).

OXM (Object XML Mapping). Это модуль для случаев, когда данные приходят или уходят в формате XML (например, обмен данными с внешними системами). Он помогает «преобразовывать» XML в Java-объекты: то есть вы получаете не текст XML, а нормальный объект с полями, и наоборот. Проще: OXM нужен, чтобы работать с XML так же удобно, как с обычными объектами.

JMS (Java Messaging Service). Это про обмен сообщениями через очереди/брокеры (например, ActiveMQ, RabbitMQ через адаптеры и т.п.), когда сервисы общаются не напрямую HTTP-запросом, а сообщениями. Это полезно, если задача должна выполняться асинхронно: отправили сообщение -> система обработает позже, не заставляя пользователя ждать. Проще: JMS — это «письма в очередь», чтобы сервисы работали более устойчиво и не зависели от мгновенного ответа.

Transaction Management (управление транзакциями). Транзакция — это «все или ничего» для набора операций с данными: либо все изменения успешно сохраняются, либо при ошибке откатываются обратно. Spring умеет управлять транзакциями так, чтобы разработчик не писал вручную commit/rollback в каждом месте — чаще всего достаточно аннотации @Transactional. Проще: этот модуль нужен, чтобы данные не «ломались», когда в середине операции случилась ошибка (например, деньги списали, а запись о платеже не сохранилась — так быть не должно).

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

При помощи данного компонента мы можем реализовывать веб-интерфейс.

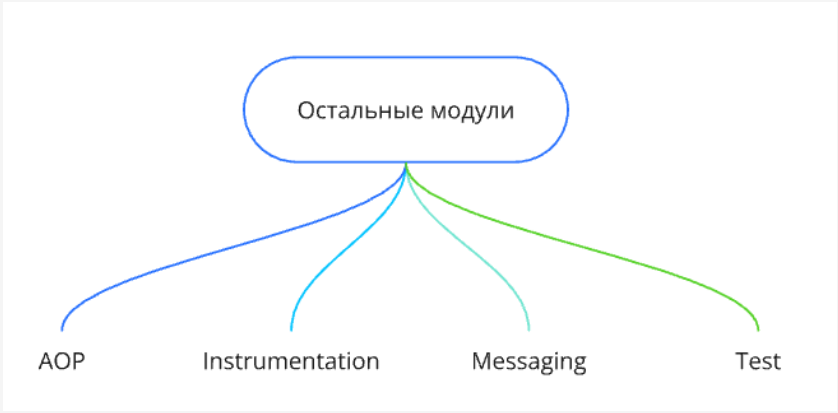
1. **Spring MVC.** Компонент (модуль) для разработки веб-приложений на основе паттерна Model-View-Controller. Поддерживает RESTful веб-сервисы и легко интегрируется с различными видами представлений, включая JSP, Thymeleaf и др.
2. **Spring WebFlux.** Асинхронный, неблокирующий веб-фреймворк, использующий реактивное программирование и поддержку сервлетов 3.1+.

Реактивное программирование — это асинхронность, соединенная с потоковой обработкой данных. То есть если в асинхронной обработке нет блокировок потоков, но данные обрабатываются все равно порциями, то реактивность добавляет возможность обрабатывать данные потоком. Грубо говоря, мы можем обрабатывать равное количество данных потоком.

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»



AOP (Aspect-Oriented Programming). AOP — это способ вынести повторяющиеся действия из бизнес-кода и подключать их «сбоку». Например, логирование, проверка прав доступа, замер времени выполнения, транзакции — все это можно не писать в каждом методе вручную. Проще: AOP позволяет сказать «перед вызовом метода сделай X» или «после вызова сделай Y», чтобы код сервиса оставался чистым и читабельным.

Instrumentation. Это про «инструментирование» приложения — то есть подключение средств, которые помогают наблюдать и управлять программой во время работы: профилирование, сбор метрик, мониторинг, иногда — модификация поведения классов на уровне загрузки (через Java-агенты). В повседневных учебных проектах это встречается редко, но в больших системах важно для диагностики: понять, где тормозит, где утечки памяти, сколько ресурсов потребляет. Проще: модуль про наблюдение и диагностику приложения «изнутри».

Messaging. Это поддержка обмена сообщениями между частями системы (или разными сервисами), когда они общаются не «в лоб» через HTTP-запрос, а через очередь/брокер сообщений. Это удобно для асинхронных задач: например, пользователь нажал «сформировать отчет» → запрос приняли сразу, а обработка пошла в фоне через сообщения. Проще: Messaging помогает строить архитектуру, где задачи обрабатываются очередями — устойчиво и без ожидания ответа «прямо сейчас».

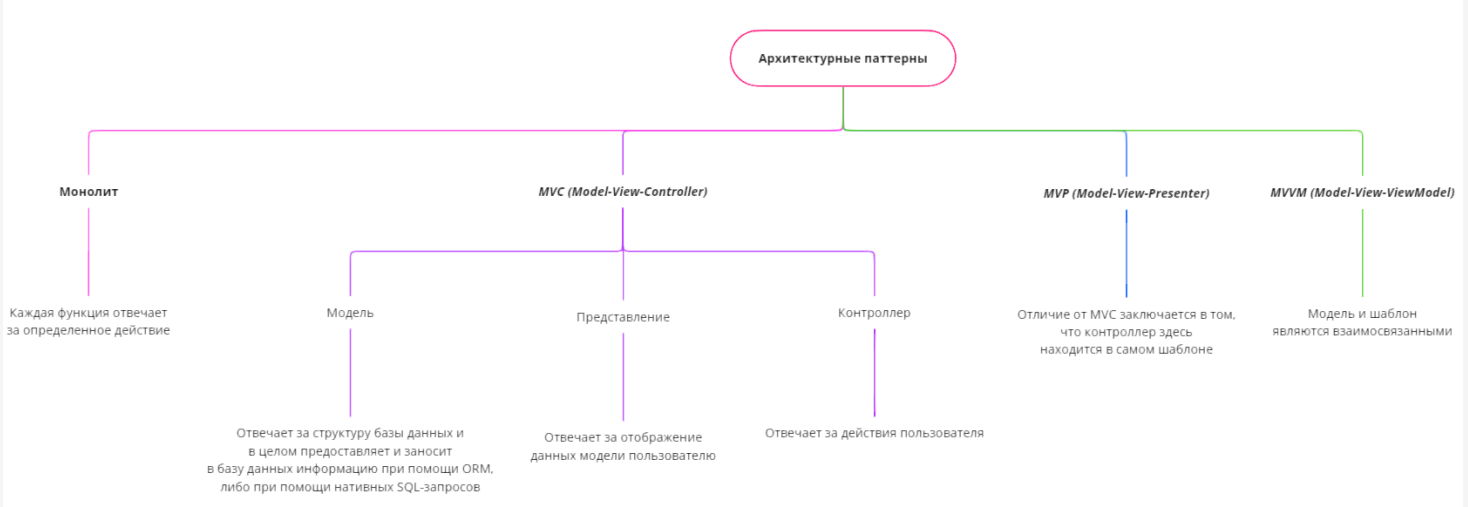
Test. Это модуль для тестирования Spring-приложений: чтобы можно было проверять контроллеры, сервисы и работу с БД без ручных запусков и “клика в браузере”. Он дает инструменты для запуска тестов с контекстом Spring, подмены зависимостей (моки), проверки HTTP-слоев (например, через MockMvc) и настройки тестовых окружений. Проще: Test нужен, чтобы быстро и уверенно проверять, что приложение работает правильно после изменений.

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Какие архитектурные паттерны существуют и какие можно применять в Spring?



Монолит. Это архитектура, где все приложение собирается и запускается как один цельный «кусок»: интерфейс, бизнес-логика и работа с данными живут в одном проекте и часто в одном процессе. Плюсы — проще стартовать и разворачивать. Минусы — со временем становится тяжело развивать, тестировать и масштабировать отдельные части независимо.

MVC (Model–View–Controller). Модель хранит данные и бизнес-логику, View отвечает за отображение, а Controller принимает действия пользователя/запросы и связывает все вместе. В веб-разработке это выглядит так: контроллер получил HTTP-запрос → вызвал сервис/модель → вернул страницу или JSON. Паттерн помогает разделить ответственность, чтобы код был понятнее и поддерживаемее.

MVP (Model–View–Presenter). Здесь View стараются сделать «тупой»: она только показывает данные и передает события, а основная логика взаимодействия переносится в Presenter. Presenter получает события от View, обращается к модели и возвращает во View готовые данные для отображения. Это удобно, когда хочется проще тестировать UI-логику и не раздувать контроллеры/код представления.

MVVM (Model–View–ViewModel). ViewModel — это прослойка между View и Model, которая хранит данные и состояние в формате, удобном для отображения, и часто поддерживает «двустороннее связывание» (binding). View автоматически обновляется, когда меняется ViewModel, и наоборот — изменения в UI могут автоматически попадать в ViewModel. Чаще всего MVVM используют в UI-фреймворках (например, WPF, Angular, некоторые подходы в Android), где binding встроен или легко реализуется.

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

1. Модульная архитектура. Можно использовать только те компоненты, которые необходимы.

2. Гибкость. Поддержка различных подходов к разработке, включая реактивное программирование и RESTful API.

3. Универсальность. Поддержка множества технологий, стандартов и фреймворков, таких как JPA, Hibernate, JMS и других.

4. Большая экосистема. Легко интегрируется с другими библиотеками и фреймворками, такими как Spring Boot, Spring Cloud, которые расширяют возможности Spring Framework.

5. Spring Framework широко используется для разработки веб-приложений, микросервисов, а также крупных корпоративных систем благодаря своей гибкости, расширяемости и поддержке передовых технологий и стандартов.

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Небольшое техническое задание.

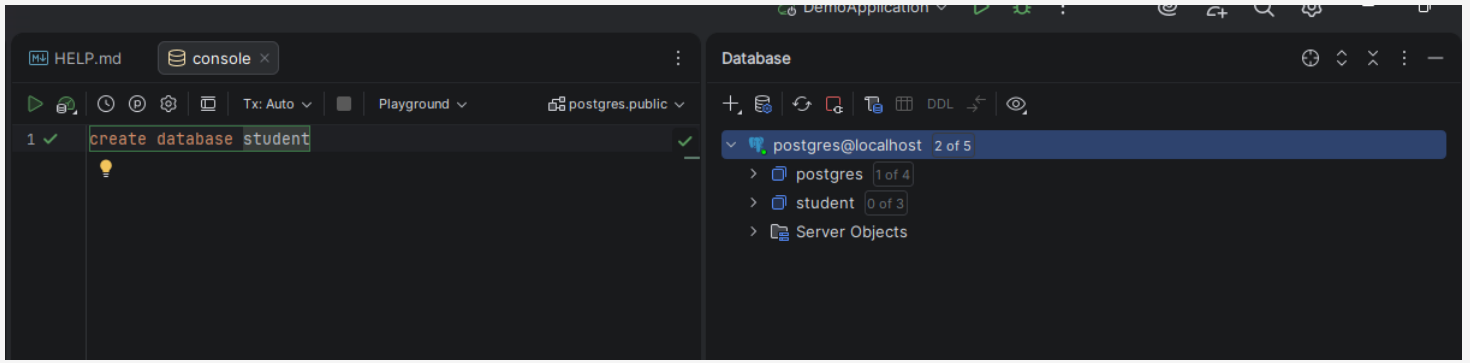
Разработать информационную систему для работы со студентами. Необходимые функции:

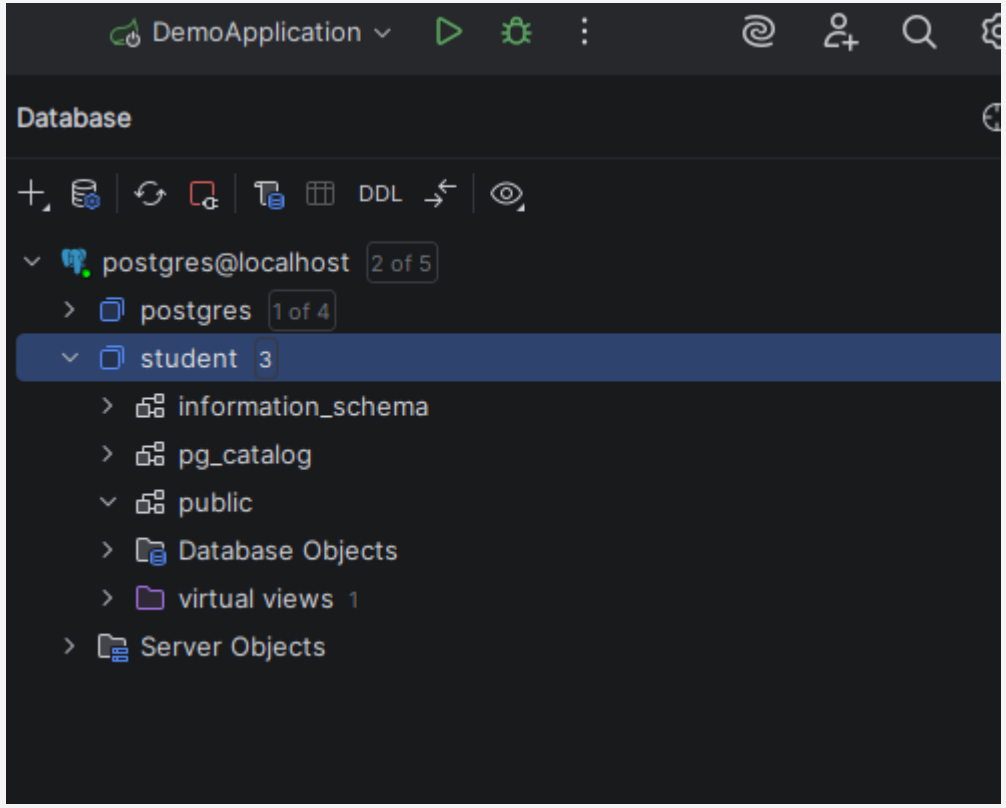
1. Добавить студента
2. Удалить студента
3. Редактировать студента
4. Поиск по любому критерию

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»



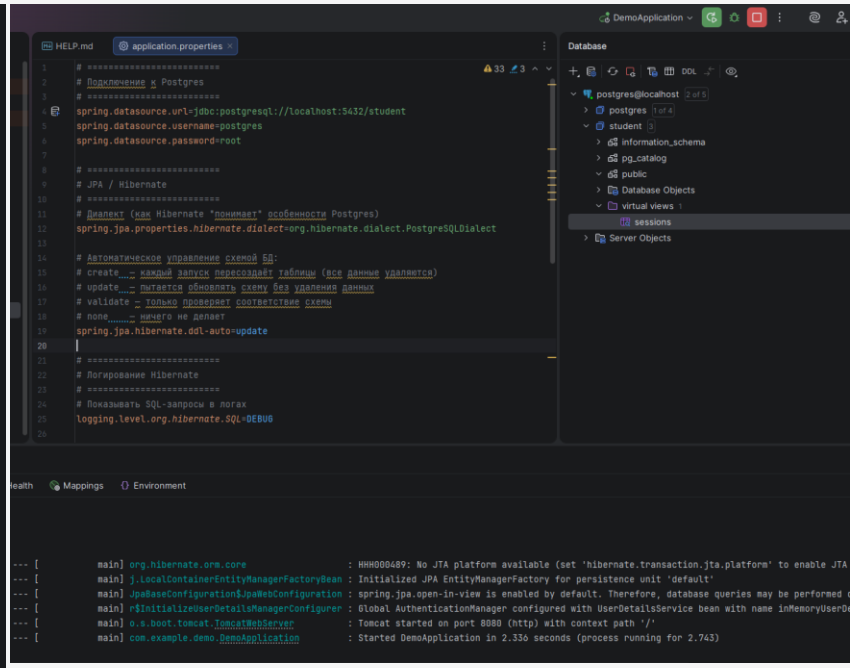


Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»


```
# ========================
# Подключение к Postgres
# ========================
spring.datasource.url=jdbc:postgresql://localhost:5432/student
spring.datasource.username=postgres
spring.datasource.password=root

# ========================
# JPA / Hibernate
# ========================
# Диалект (как Hibernate "понимает" особенности Postgres)
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

# Автоматическое управление схемой БД:
# create  — каждый запуск пересоздает таблицы (все данные удаляются)
# update  — пытается обновлять схему без удаления данных
# validate — только проверяет соответствие схемы
# none    — ничего не делает
spring.jpa.hibernate.ddl-auto=update

# ========================
# Логирование Hibernate
# ========================
# Показывать SQL-запросы в логах
logging.level.org.hibernate.SQL=DEBUG

# Подробные логи по типам данных и передаваемым значениям (очень много вывода)
logging.level.org.hibernate.type=TRACE

# ========================
# Spring Security (учетка по умолчанию)
# ========================
spring.security.user.name=root
spring.security.user.password=root
spring.security.user.roles=manager

# ========================
# Кэширование
# ========================
spring.cache.type=none

# ========================
# Статические ресурсы (CSS/JS/картинки из resources/static)
# ========================
spring.web.resources.add-mappings=true
```



JPA (Java Persistence API) - это спецификация API, которая предоставляет возможность сохранять Java-объекты в базе данных. Она поддерживает различные аспекты сохранения данных, включая API, язык запросов и метаинформацию.

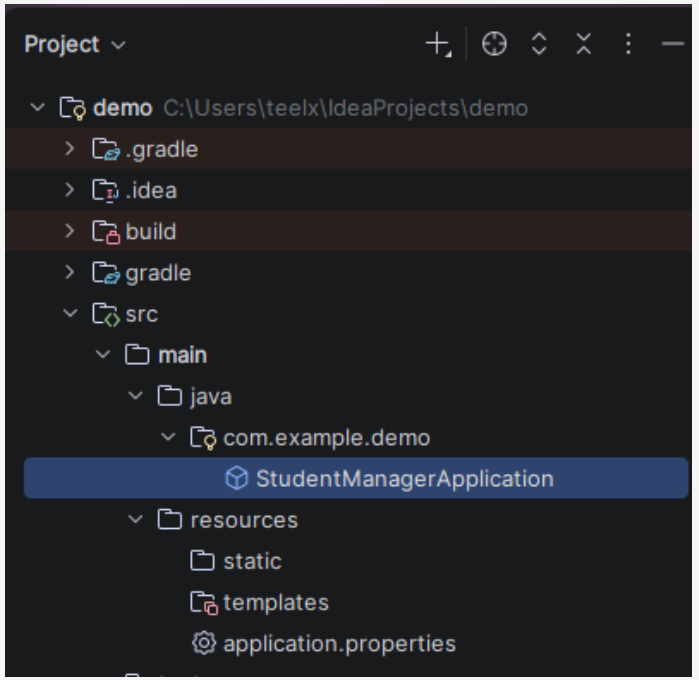
Hibernate - это библиотека для языка программирования Java, которая является одной из самых популярных реализаций JPA. Она позволяет сократить объемы низкоуровневого программирования при работе с реляционными базами данных и предоставляет средства для автоматической генерации и обновления таблиц, построения запросов и обработки данных. Hibernate также предоставляет поддержку для использования SQL-подобного языка запросов и аннотаций для проверки верности данных.

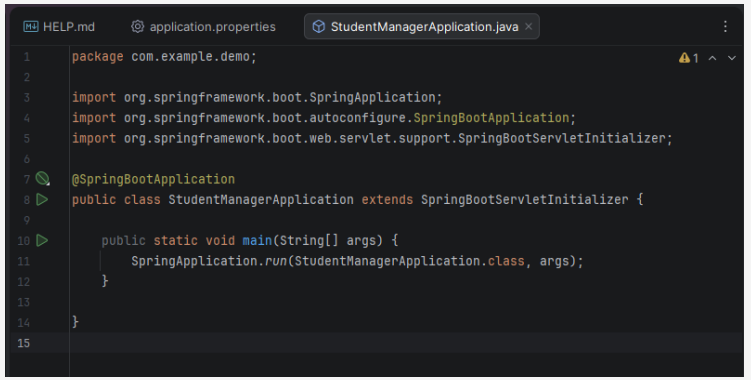
Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Класс StudentManagerApplication





Данный класс отвечает за сборку и запуск нашего веб-приложения. Класс **StudentManagerApplication** является точкой входа в приложение Spring Boot. Класс наследуется от **SpringBootServletInitializer**, что позволяет ему работать как сервлетный контейнер.

Сервлетный контейнер – это наш веб-сервер (по умолчанию Apache Tomcat).

Класс помечен аннотацией @**SpringBootApplication**, которая указывает Spring Boot, что это основной класс приложения. Это включает в себя автоматическую конфигурацию Spring, такие как автоматическое обнаружение компонентов приложения и их сборку.

Метод **main** запускает приложение, вызывая метод **run** класса **SpringApplication** с классом **StudentManagerApplication** и аргументами командной строки.

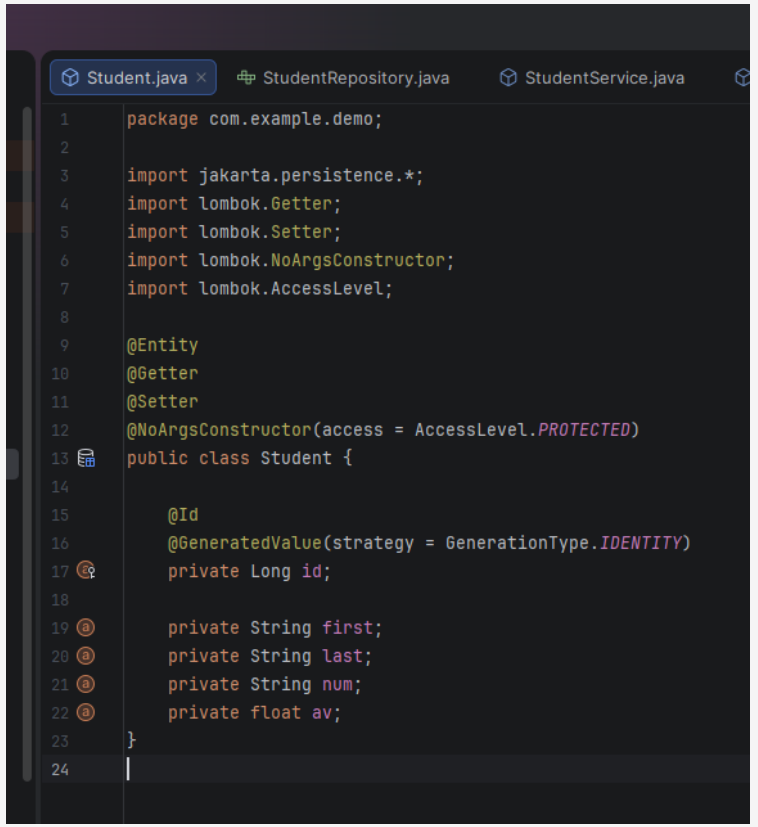
Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Класс Student. Начало

```java
package com.example.demo; // Пакет (папка) проекта, где лежит этот класс

import jakarta.persistence.*;       // JPA-аннотации: @Entity, @Id, @GeneratedValue и др.
import lombok.Getter;               // Lombok: автоматически генерирует геттеры
import lombok.Setter;               // Lombok: автоматически генерирует сеттеры
import lombok.NoArgsConstructor;    // Lombok: генерирует конструктор без параметров
import lombok.AccessLevel;          // Уровни доступа (PUBLIC/PROTECTED/PRIVATE) для конструктора

@Entity // Говорим Hibernate/JPA: этот класс — "сущность", то есть будет таблицей в базе данных
@Getter // Lombok создаст геттеры для всех полей (getId(), getFirst() и т.д.)
@Setter // Lombok создаст сеттеры для всех полей (setId(), setFirst() и т.д.)
@NoArgsConstructor(access = AccessLevel.PROTECTED)
// Создает конструктор без параметров (нужен JPA, чтобы передавать параметры класса далее). PROTECTED — чтобы случайно не создавали объект "пустым" где попало

public class Student { // Класс-сущность "Студент" (обычно = таблица student)

    @Id
    // Первичный ключ таблицы (уникальный идентификатор записи: 1, 2, 3, ...)

    @GeneratedValue(strategy = GenerationType.IDENTITY)
    // ID будет генерироваться самой базой данных автоматически (обычно автоинкремент)

    private Long id; // Поле ID (ключ). Тип Long — потому что ключи часто большие (и так принято в JPA)

    private String first; // Имя студента
    private String last;  // Фамилия студента
    private String num;   // Номер студенческого билета (или любой идентификатор)
    private float av;    // Средний балл
}
```



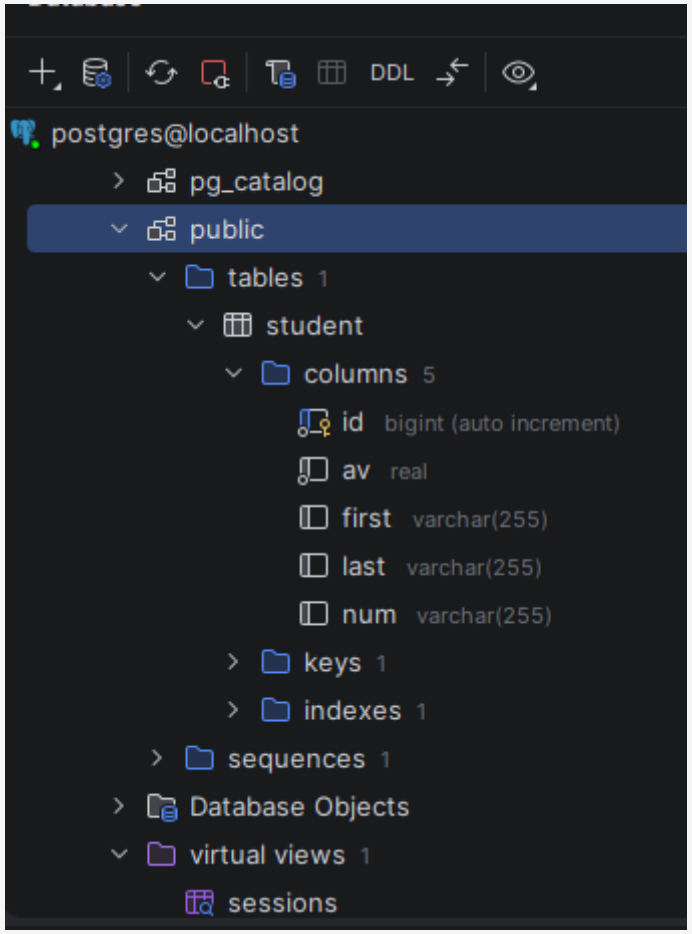
Геттер (getter) — это метод, который возвращает значение поля объекта. Нужен, чтобы безопасно «читать» данные из объекта и не лезть напрямую к его полям (это часть инкапсуляции).

Сеттер (setter) — это метод, который изменяет (устанавливает) значение поля объекта. Нужен, чтобы контролировать изменения: можно проверять данные (валидация), ограничивать доступ, запускать доп. логику при изменении.

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»



В результате у нас появилась таблица с полями, идентичными по названию с переменными. Более того, наименование таблицы = наименование класса Student.

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Интерфейс StudentRepository

Интерфейс **StudentRepository** является репозиторием в Spring Data JPA. Репозитории в Spring Data JPA предоставляют абстракции над базовыми CRUD-операциями (**C**REATE – создание, **R**EAD – чтение, **U**PDATE – обновление, **D**ELETE – удаление) для работы с данными в базе данных.

Интерфейс **StudentRepository** расширяет (наследует) интерфейс Jpa**R**epository, который предоставляет базовые операции CRUD для сущностей, идентифицируемых по Long. **JpaRepository** автоматически генерируется Spring Data JPA на основе информации, полученной из аннотаций **@Entity** и **@Id** в классе **Student**.

Интерфейс **StudentRepository** также содержит метод **search**, который использует аннотацию **@Query** для определения SQL-запроса, который будет выполняться для поиска студентов по заданному поисковому запросу. Запрос использует функцию **CONCAT** для объединения различных полей студента в одно строковое значение, которое затем сравнивается с шаблоном, включающим **%?1%**, где **?1** представляет собой параметр запроса, который будет заполнен значением **keyword**.

```
package com.example.demo; // Пакет (пространство имен) — куда относится этот файл

import java.util.List; // Импорт интерфейса List — нужен, чтобы возвращать список студентов

import org.springframework.data.jpa.repository.JpaRepository; // Базовый интерфейс Spring Data JPA: дает готовые CRUD-методы (save, findById, findAll, delete и т.д.)
import org.springframework.data.jpa.repository.Query; // Аннотация @Query — позволяет писать свой запрос (JPQL/SQL) прямо над методом репозитория

public interface StudentRepository extends JpaRepository<Student, Long> {
   // Репозиторий для сущности Student.
   // Student — класс-сущность (таблица), Long — тип первичного ключа (id).

   @Query("SELECT p FROM Student p WHERE CONCAT(p.first, ' ', p.last, ' ', p.num, ' ', p.av) LIKE %?1%")
      // JPQL-запрос (это НЕ чистый SQL):
      // SELECT p FROM Student p  -> "выбери объекты Student" (p — псевдоним/короткое имя для сущности Student)
      // CONCAT(...)          -> склеиваем поля (имя, фамилия, номер, средний балл) в одну строку
      // LIKE %?1%            -> ищем в этой строке подстроку; ?1 означает "первый параметр метода" (keyword)

   List<Student> search(String keyword);
   // Метод, который вернет список студентов, у которых keyword встречается в склеенной строке полей
}
```

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

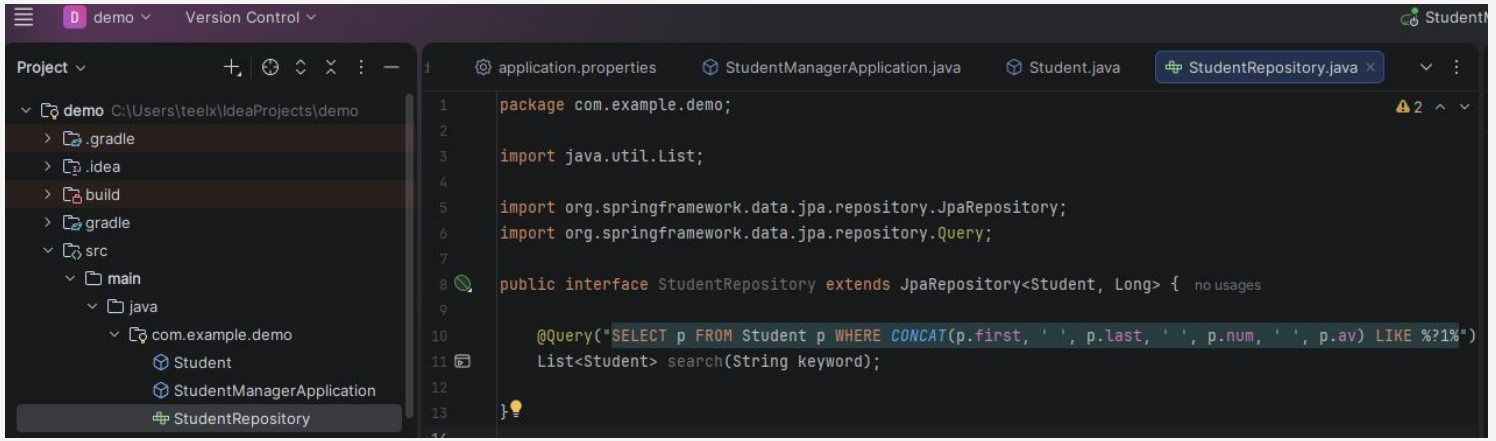
Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Интерфейс StudentRepository

Интерфейс **StudentRepository** является репозиторием в Spring Data JPA. Репозитории в Spring Data JPA предоставляют абстракции над базовыми CRUD-операциями (**C**REATE – создание, **R**EAD – чтение, **U**PDATE – обновление, **D**ELETE – удаление) для работы с данными в базе данных.

Интерфейс **StudentRepository** расширяет (наследует) интерфейс Jpa**R**epository, который предоставляет базовые операции CRUD для сущностей, идентифицируемых по Long. **JpaRepository** автоматически генерируется Spring Data JPA на основе информации, полученной из аннотаций **@Entity** и **@Id** в классе **Student**.

Интерфейс **StudentRepository** также содержит метод **search**, который использует аннотацию **@Query** для определения SQL-запроса, который будет выполняться для поиска студентов по заданному поисковому запросу. Запрос использует функцию **CONCAT** для объединения различных полей студента в одно строковое значение, которое затем сравнивается с шаблоном, включающим **%?1%**, где **?1** представляет собой параметр запроса, который будет заполнен значением **keyword**.



Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

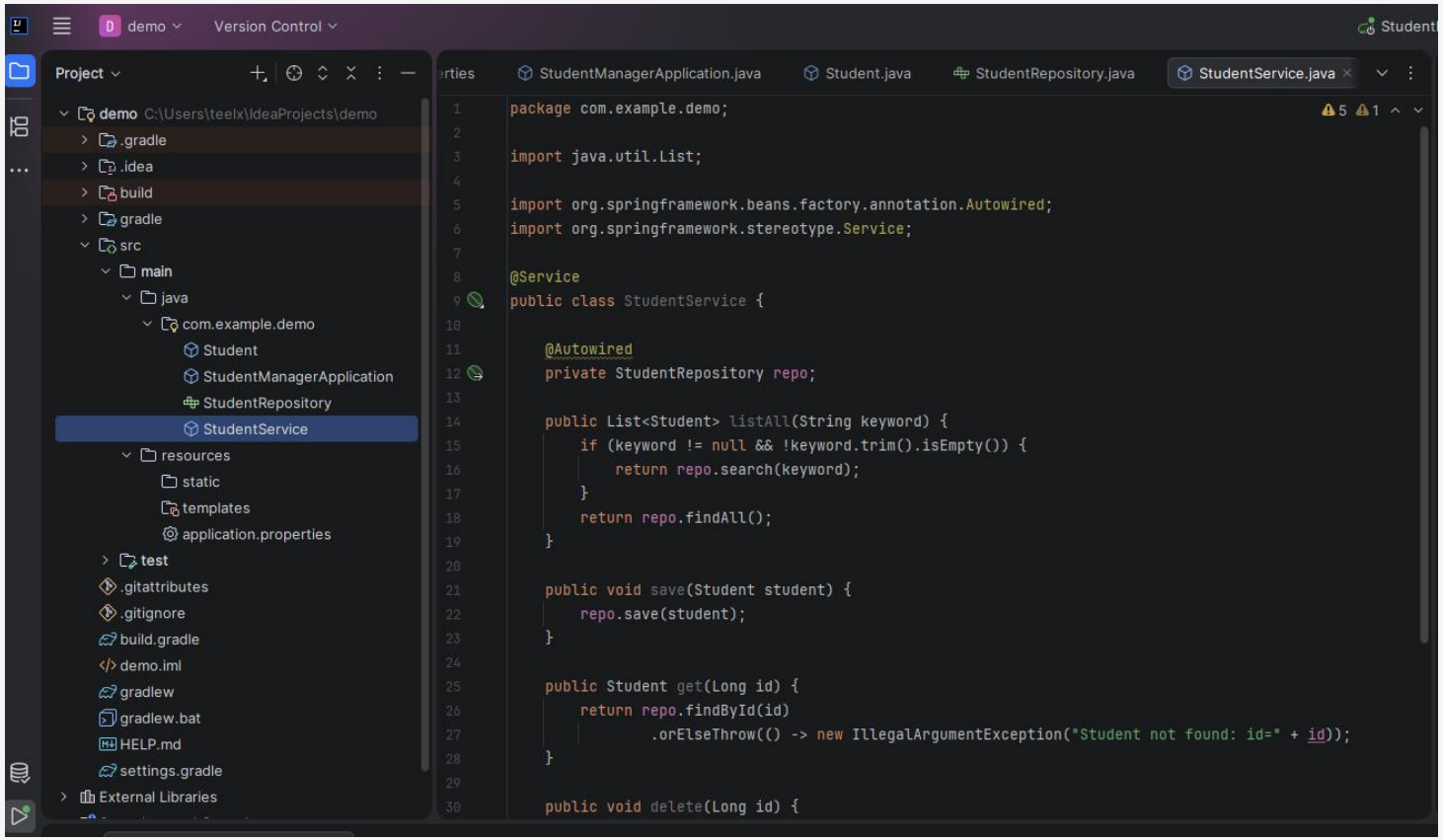
Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Класс StudentService

```
package com.example.demo; // Пакет (пространство имен), к которому относится класс

import java.util.List; // Нужен для возвращаемого типа: список студентов

import org.springframework.beans.factory.annotation.Autowired; // Аннотация для внедрения зависимостей (DI)
import org.springframework.stereotype.Service;          // Аннотация: помечает класс как сервис Spring (бизнес-логика)

@Service // Spring создаст объект (bean) этого класса и будет управлять его жизненным циклом
public class StudentService {

  @Autowired
  private StudentRepository repo;
  // Внедряем репозиторий (слой доступа к данным).
  // Spring сам создаст реализацию StudentRepository и подставит ее сюда (dependency injection).

  // Метод возвращает список студентов.
  // Если keyword задан (не null и не пустой), берем отфильтрованный список через repo.search(keyword).
  // Если keyword не задан — возвращаем всех студентов через repo.findAll().
  public List<Student> listAll(String keyword) {
    if (keyword != null && !keyword.trim().isEmpty()) { // Проверяем, что строка реально содержит текст
      return repo.search(keyword);              // Наш метод поиска из репозитория
    }
    return repo.findAll(); // Стандартный метод JpaRepository: вернуть всех студентов
  }

  // Сохраняет студента в базе данных.
  // JpaRepository.save(...) умеет и добавлять нового студента, и обновлять существующего (если id уже есть).
  public void save(Student student) {
    repo.save(student);
  }

  // Получает студента по его id.
  // findById(...) возвращает Optional<Student>.
  // .orElseThrow(...) лучше, чем .get(): так будет понятная ошибка, если студента не нашли.
  public Student get(Long id) {
    return repo.findById(id)
        .orElseThrow(() -> new IllegalArgumentException("Student not found: id=" + id));
  }

  // Удаляет студента из базы данных по его id.
  public void delete(Long id) {
    repo.deleteById(id);
  }

}
```

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»



Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

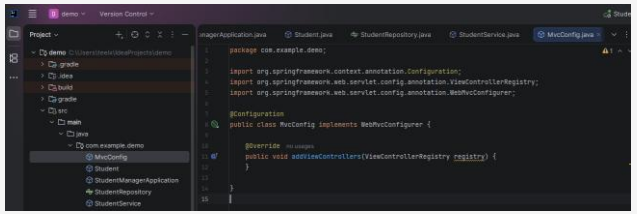
Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Класс MvcConfig

Класс **MvcConfig** реализует интерфейс **WebMvcConfigurer**. Этот интерфейс содержит методы, которые позволяют настраивать поведение архитектурного паттерна Spring **MVC** (**Model-View-Controller**), включая добавление контроллеров, обработчиков исключений, фильтров и многое другое. Интерфейс **WebMvcConfigurer** имеет метод **addViewControllers**, который позволяет добавлять контроллеры в Spring MVC. В данном случае, метод **addViewControllers** пустой, что означает, что в приложении нет необходимости добавлять дополнительные контроллеры.

Класс **MvcConfig** помечен аннотацией **@Configuration**, что указывает Spring, что этот класс содержит конфигурационные настройки для приложения. Создается на тот случай, если недостаточно класса **AppController**.

```
package com.example.demo;
// Пакет (пространство имен), в котором находится этот класс

import org.springframework.context.annotation.Configuration;
// @Configuration — говорит Spring, что это конфигурационный класс (настройки приложения)

import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
// ViewControllerRegistry — объект, через который можно добавить "простые" маршруты (URL → страница)
// без написания отдельного контроллера

import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
// WebMvcConfigurer — интерфейс, который позволяет тонко настраивать Spring MVC
// (маршруты, конвертеры, форматирование, CORS, interceptors и т.д.)

@Configuration
// Spring увидит этот класс при запуске и применит настройки из него
public class MvcConfig implements WebMvcConfigurer {

    @Override
    // Переопределяем метод из WebMvcConfigurer, чтобы добавить view-контроллеры
    public void addViewControllers(ViewControllerRegistry registry) {

        // Здесь можно "привязать" URL к шаблону/странице без написания @Controller.
        // Пример:
        // registry.addViewController("/login").setViewName("login");
        // Это означает: при запросе GET /login показать шаблон login.html

    }

}
```



Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

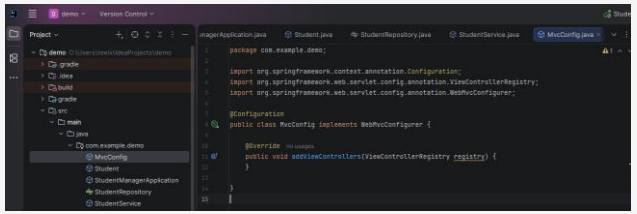
Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Класс MvcConfig

Класс **MvcConfig** реализует интерфейс **WebMvcConfigurer**. Этот интерфейс содержит методы, которые позволяют настраивать поведение архитектурного паттерна Spring **MVC** (**Model-View-Controller**), включая добавление контроллеров, обработчиков исключений, фильтров и многое другое. Интерфейс **WebMvcConfigurer** имеет метод **addViewControllers**, который позволяет добавлять контроллеры в Spring MVC. В данном случае, метод **addViewControllers** пустой, что означает, что в приложении нет необходимости добавлять дополнительные контроллеры.

Класс **MvcConfig** помечен аннотацией **@Configuration**, что указывает Spring, что этот класс содержит конфигурационные настройки для приложения. Создается на тот случай, если недостаточно класса **AppController**.

```
package com.example.demo;
// Пакет (пространство имен), в котором находится этот класс

import org.springframework.context.annotation.Configuration;
// @Configuration — говорит Spring, что это конфигурационный класс (настройки приложения)

import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
// ViewControllerRegistry — объект, через который можно добавить "простые" маршруты (URL → страница)
// без написания отдельного контроллера

import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
// WebMvcConfigurer — интерфейс, который позволяет тонко настраивать Spring MVC
// (маршруты, конвертеры, форматирование, CORS, interceptors и т.д.)

@Configuration
// Spring увидит этот класс при запуске и применит настройки из него
public class MvcConfig implements WebMvcConfigurer {

    @Override
    // Переопределяем метод из WebMvcConfigurer, чтобы добавить view-контроллеры
    public void addViewControllers(ViewControllerRegistry registry) {

        // Здесь можно "привязать" URL к шаблону/странице без написания @Controller.
        // Пример:
        // registry.addViewController("/login").setViewName("login");
        // Это означает: при запросе GET /login показать шаблон login.html

    }

}
```



Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Контроллер. Часть 1

```
package com.example.demo;

import java.util.List; // Импортируем коллекцию (класс списков)
import org.springframework.beans.factory.annotation.Autowired; // Аннотация @Autowired используется в Spring Framework для переопределения и автоматического внедрения зависимостей (dependency injection)
import org.springframework.data.repository.query.Param; // Аннотация @Param используется в Spring Data для параметризации методов репозитория. Она указывает, что параметр метода является параметром запроса, который будет использован в SQL-запросе или другом запросе к базе данных.
import org.springframework.stereotype.Controller; // Используется в Spring Framework для маркировки классов как контроллеров. Контроллеры в Spring MVC отвечают за обработку HTTP-запросов и возвращение ответов в виде представлений, JSON, XML и т.д.
import org.springframework.ui.Model; // Класс (Модель), который используется в Spring MVC для передачи данных из контроллера в представление
import org.springframework.web.bind.annotation.ModelAttribute; // Аннотация @ModelAttribute используется в Spring MVC для связывания параметров запроса с атрибутами модели. Это позволяет легко передавать данные из формы в контроллер и обратно
import org.springframework.web.bind.annotation.PathVariable; // Аннотация @PathVariable используется в Spring MVC для параметризации методов контроллера с использованием частей пути запроса. Это позволяет легко маршрутизировать запросы на основе определенных частей URL.
import org.springframework.web.bind.annotation.RequestMapping; // Задаем наш URl. Аннотация @RequestMapping используется в Spring MVC для указания, какие методы контроллера должны обрабатывать определенные HTTP-методы и пути
import org.springframework.web.bind.annotation.RequestMethod; // Какие запросы выполнять: POST или GET?
import org.springframework.web.servlet.ModelAndView; // Класс ModelAndView представляет собой абстрактный класс, который используется в Spring MVC для возвращения модели и представления из контроллера. Он объединяет модель, которая содержит данные для представления, и имя представления, которое будет отображено.
```

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»


```
@Controller // Весь наш класс будет контроллером
public class AppController { // Создаем класс с модификатором доступа Public, так как контроллер должен быит открытым полностью из-за аннотации @Controller

    @Autowired // Автоматически создаем и внедряем класс StudentService и переименовываем в его service. Вот так и выглядит внедрение зависимостей (dependency injection)
    private StudentService service;

    @RequestMapping("/") // Наша главная страница
    //1. Вызывается метод service.listAll(keyword), который, вероятно, возвращает список студентов, соответствующих заданному поисковому запросу.
    //2. Полученный список добавляется в модель под именем listStudents.
    //3. Значение keyword также добавляется в модель.
    //4. Возвращается имя представления index, которое будет отображаться в ответ на запрос.
    public String viewHomePage(Model model, @Param("keyword") String keyword) {
        //Реализация поиска на главной странице по критериям
        List<Student> listStudents = service.listAll(keyword); // Наш список студентов. Элементы в список передаются из класса StudentService
        model.addAttribute("listStudents", listStudents); //  Создаем модель и добавляем в нее атрибут. На главной странице будет выводиться список студентов
        model.addAttribute("keyword", keyword); // Создаем модель и добавляем в нее атрибут. На главной странице будет выводиться поиск студентов
        return "index"; // Выводится все то, что отображено в шаблоне index.html, модели будут туда также добавляться
    }
```

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

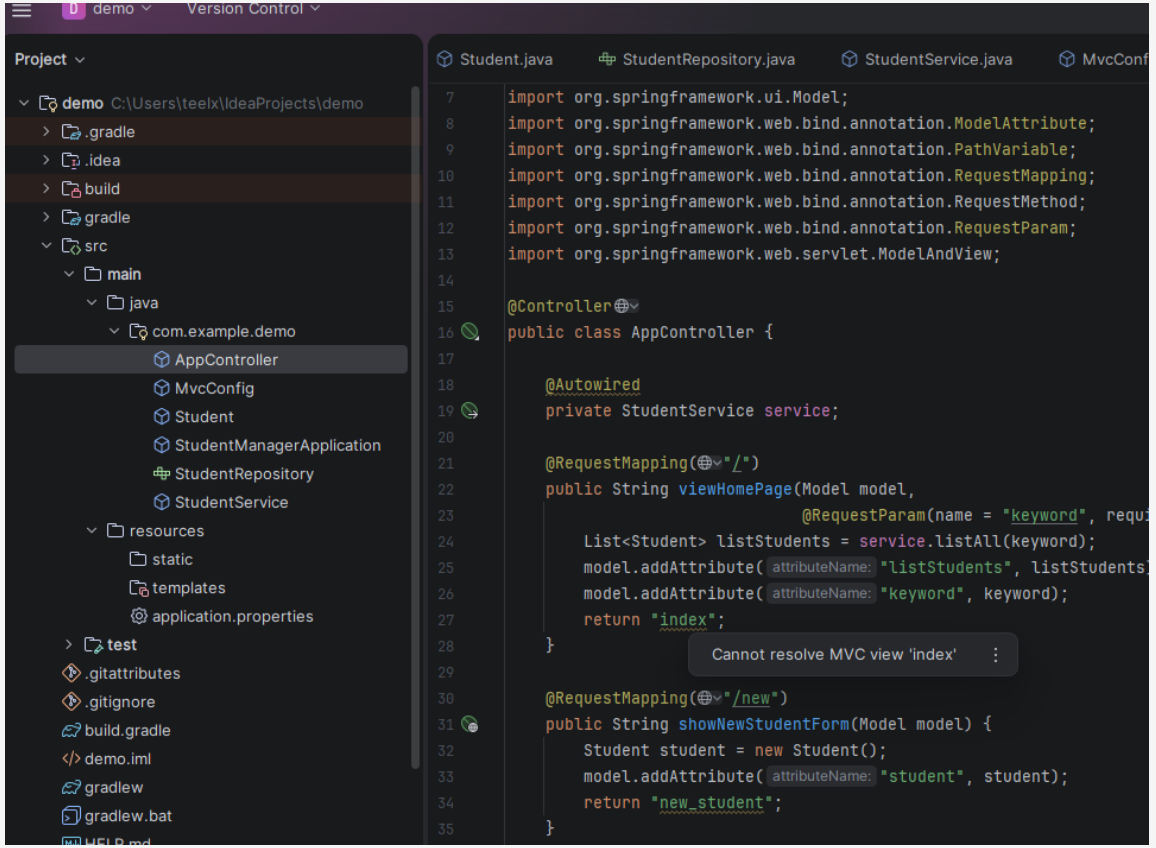
Подтема: «Разработка веб-приложений с использованием Git»


```
@RequestMapping("/new") // Добавление студента
  public String showNewStudentForm(Model model) {
    Student student = new Student(); // Создаем экземпляр класса Student, внутри
которого "лежит" наша модель базы данных
    model.addAttribute("student", student); // Добавляем в модель данные (грубо говоря,
модель данных в нашем случае - список)
    return "new_student"; // В браузере отображается все то, что есть в шаблоне
new_student.html
  }

  @RequestMapping(value = "/save", method = RequestMethod.POST) // Передаем данные
из модели (получили в методе showNewStudentForm) в базу данных
  public String saveStudent(@ModelAttribute("student") Student student) {
    service.save(student); // Сохраяем наш список
    return "redirect:/"; // После сохранения данных нас перенаправит на главную
страницу, где уже будет обновленный список с учетом внесенных данных
  }

  @RequestMapping("/edit/{id}") // Страница редактирования данных о студенте
  public ModelAndView showEditStudentForm(@PathVariable(name = "id") Long id) {
    ModelAndView mav = new ModelAndView("edit_student"); // Добавляем шаблон в
модель
    Student student = service.get(id); // Передаем ID, по которому будем редактировать
    mav.addObject("student", student);
    return mav; // Возвращаем полностью модель
  }

  @RequestMapping("/delete/{id}") // Удаялем студента
  public String deleteStudent(@PathVariable(name = "id") Long id) {
    service.delete(id);
    return "redirect:/";
  }
}
```

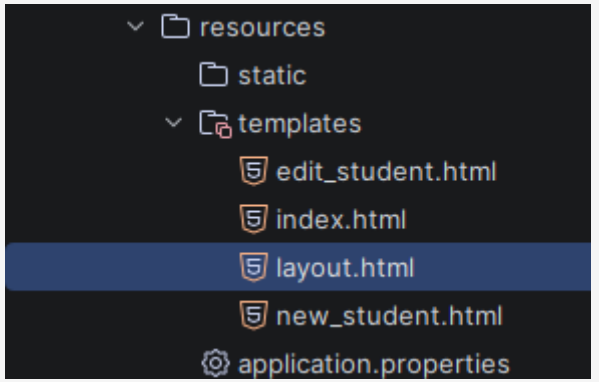
Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»



Далее создадим шаблоны, чтобы избавиться от данной ошибки.



Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Интерфейс. Часть 1. layout.html. Создаем базовый шаблон и будем его наследовать

```
<!doctype html>
<!-- Документ HTML5 -->

<html lang="ru" xmlns:th="http://www.thymeleaf.org">
<!-- lang="ru" — язык страницы (подсказки, переносы, доступность, поисковики)
    xmlns:th — подключаем пространство имен Thymeleaf, чтобы работали атрибуты th:* -->

<head>
  <meta charset="UTF-8" />
  <!-- Кодировка UTF-8 — чтобы русский текст отображался корректно -->

  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <!-- Для адаптивной верстки: на телефонах страница не будет "уменьшаться" -->

  <title th:text="${title} ?: 'Student Manager'">Student Manager</title>
  <!-- Заголовок вкладки:
      th:text="${title} ?: 'Student Manager'" — если в модель передали title, берем его,
      иначе используем значение по умолчанию 'Student Manager' -->

  <!-- Bootstrap 5: готовые стили (кнопки, таблицы, сетка, отступы и т.д.) -->
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">

  <!-- Bootstrap Icons: библиотека иконок (лупа, плюс, карандаш, корзина и т.д.) -->
  <link href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.11.3/font/bootstrap-icons.css" rel="stylesheet">
</head>

<body class="bg-light">
<!-- bg-light — светлый фон страницы -->

<!-- Верхняя навигационная панель (navbar) -->
<nav class="navbar navbar-expand-lg bg-white border-bottom">
  <!-- navbar-expand-lg — на больших экранах панель расширяется (если будут пункты меню)
      bg-white — белый фон
      border-bottom — линия снизу -->

  <div class="container">
    <!-- container — центрирование и ограничение ширины контента -->

    <a class="navbar-brand fw-semibold" th:href="@{/}">
      <!-- navbar-brand — стиль "логотипа/названия"
          fw-semibold — полужирный текст
          th:href="@{/}" — Thymeleaf: ссылка на главную страницу -->
      <i class="bi bi-mortarboard"></i> Student Manager
      <!-- bi-mortarboard — иконка (академическая шапочка) -->
    </a>

    <!-- Правая часть navbar: сюда будем подставлять кнопки (например "Добавить", "Назад") -->
    <div class="ms-auto d-flex gap-2" th:insert="${navRight} ?: ~{}"></div>
    <!-- ms-auto — прижимает блок вправо
        d-flex — элементы внутри в одну линию
        gap-2 — расстояние между кнопками
        th:insert="${navRight} ?: ~{}" — если передали navRight (фрагмент), вставляем его,
        иначе вставляем пустоту (~{}) -->
  </div>
</nav>
```

```
<!-- Основная часть страницы -->
<main class="container py-4"
th:insert="${content}"></main>
<!-- container — нормальная ширина и центрирование
    py-4 — отступы сверху и снизу
    th:insert="${content}" — сюда будет подставляться главный контент конкретной страницы -->

<!-- JS Bootstrap: нужен для интерактивных компонентов (меню, модалки, выпадающие списки и т.п.) -->
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"></script>

</body>
</html>
```

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Интерфейс. Часть 2. Index.html

```
<!doctype html>
<!-- Документ HTML5 -->
<html lang="ru" xmlns:th="http://www.thymeleaf.org"
     th:replace="~{layout :: html(
       title='Students',
       navRight=~{::navRight},
       content=~{::content}
     )}">
<!-- th:replace — Thymeleaf: вместо ЭТОГО файла будет подставлен layout.html (его фрагмент html),
     а сюда мы передаем параметры:
     title   — заголовок вкладки,
     navRight — правая часть navbar,
     content  — основной контент страницы -->

<!-- ========================= -->
<!-- Правая часть navbar -->
<!-- ========================= -->
<th:block th:fragment="navRight">
   <!-- th:fragment="navRight" — именованный фрагмент, который layout подставит в navbar справа -->
   <a class="btn btn-primary" th:href="@{/new}">
      <!-- Кнопка перехода на страницу добавления студента -->
      <i class="bi bi-plus-lg"></i> Добавить
      <!-- Иконка "плюс" -->
   </a>
</th:block>

<!-- ========================= -->
<!-- Основной контент страницы -->
<!-- ========================= -->
<th:block th:fragment="content">
   <!-- th:fragment="content" — основной блок страницы, который layout вставит внутрь <main> -->
```

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»


```
<div class="row g-3 align-items-stretch">
  <!-- row — строка Bootstrap-сетки
      g-3 — общий отступ между колонками/элементами
      align-items-stretch — колонки растягиваются по высоте одинаково -->

  <!-- Левая большая колонка: список студентов -->
  <div class="col-12 col-lg-8">
    <!-- col-12 — на маленьких экранах 100% ширины
        col-lg-8 — на больших экранах 8/12 ширины -->

    <div class="card shadow-sm">
      <!-- card — карточка Bootstrap
          shadow-sm — небольшая тень -->

      <div class="card-body">
        <!-- card-body — внутренние отступы карточки -->

        <h5 class="card-title mb-3">Список студентов</h5>
        <!-- card-title — заголовок карточки
            mb-3 — отступ снизу -->

        <!-- Форма поиска -->
        <form class="row g-2 mb-3" th:action="@{/}" method="get">
          <!-- method="get" — поиск идет через URL-параметр, например: /?keyword=иван
              th:action="@{/}" — отправляем запрос на главную страницу -->

          <div class="col-12 col-md-8">
            <!-- На телефоне 100%, на md+ 8/12 -->

            <div class="input-group">
              <!-- input-group — объединение иконки + поля ввода -->

              <span class="input-group-text">
                <i class="bi bi-search"></i>
              </span>
              <!-- input-group-text — приставка слева
                  bi-search — иконка лупы -->

              <input class="form-control"
                   type="text"
                   name="keyword"
                   placeholder="Поиск: имя, фамилия, номер, средний балл"
                   th:value="${keyword}">
              <!-- form-control — стиль Bootstrap
                  name="keyword" — имя параметра запроса (?keyword=...)
                  th:value="${keyword}" — Thymeleaf: сохраняем введенный текст в поле после поиска -->
            </div>
          </div>
```

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Интерфейс. Часть 3. Index.html

```
  <div class="col-12 col-md-4 d-flex gap-2">
    <!-- Блок кнопок: на телефоне 100%, на md+ 4/12
         d-flex — кнопки в одну линию
         gap-2 — расстояние между кнопками -->

    <button class="btn btn-outline-primary w-100" type="submit">
      Найти
    </button>
    <!-- submit — отправляет форму -->

    <a class="btn btn-outline-secondary w-100" th:href="@{/}">
      Сброс
    </a>
    <!-- Ссылка на "/" без параметров — сбрасывает поиск -->
  </div>
</form>

<!-- Таблица студентов -->
<div class="table-responsive">
  <!-- table-responsive — на узких экранах появится горизонтальная прокрутка -->

  <table class="table table-hover align-middle">
    <!-- table — таблица Bootstrap
         table-hover — подсветка строк при наведении
         align-middle — выравнивание содержимого по центру по вертикали -->

    <thead class="table-light">
    <!-- table-light — светлая шапка таблицы -->

    <tr>
      <th style="width: 90px;">ID</th>
      <!-- фиксируем ширину колонки ID -->
      <th>Имя</th>
      <th>Фамилия</th>
      <th>Студ. билет</th>
      <th style="width: 150px;">Средний балл</th>
      <th class="text-end" style="width: 180px;">Действия</th>
      <!-- text-end — выравнивание вправо -->
    </tr>
    </thead>
```

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Интерфейс. Часть 5. Index.html

```
            <tbody>
            <!-- th:each — Thymeleaf-цикл по списку listStudents из модели -->
            <tr th:each="s : ${listStudents}">
              <td class="text-muted" th:text="${s.id}">1</td>
              <!-- th:text="${s.id}" — выводим ID студента -->

              <td th:text="${s.first}">Иван</td>
              <td th:text="${s.last}">Иванов</td>

              <td>
                <span class="badge text-bg-secondary" th:text="${s.num}">A-123</span>
                <!-- badge — “плашка”
                     text-bg-secondary — серый фон -->
              </td>

              <td>
                <span class="badge text-bg-success" th:text="${s.av}">4.5</span>
                <!-- зеленая плашка со средним баллом -->
              </td>

              <td class="text-end">
                <!-- Редактирование -->
                <a class="btn btn-sm btn-outline-primary"
                  th:href="@{/edit/{id}(id=${s.id})}">
                  <!-- ссылка получится вида /edit/123 -->
                  <i class="bi bi-pencil"></i> Изменить
                </a>

                <!-- Удаление -->
                <a class="btn btn-sm btn-outline-danger"
                  th:href="@{/delete/{id}(id=${s.id})}"
                  onclick="return confirm('Удалить этого студента?');">
                  <!-- confirm(...) — подтверждение удаления.
                       Cancel → false → переход по ссылке отменится -->
                  <i class="bi bi-trash"></i> Удалить
                </a>
              </td>
            </tr>

            <!-- Если список пустой -->
            <tr th:if="${listStudents == null || #lists.isEmpty(listStudents)}">
              <td colspan="6" class="text-center text-muted py-4">
                <!-- colspan="6" — объединяем все колонки -->
                Ничего не найдено
              </td>
            </tr>

            </tbody>
          </table>
        </div>

    </div>
  </div>
</div>
```

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Интерфейс. Часть 6. Index.html

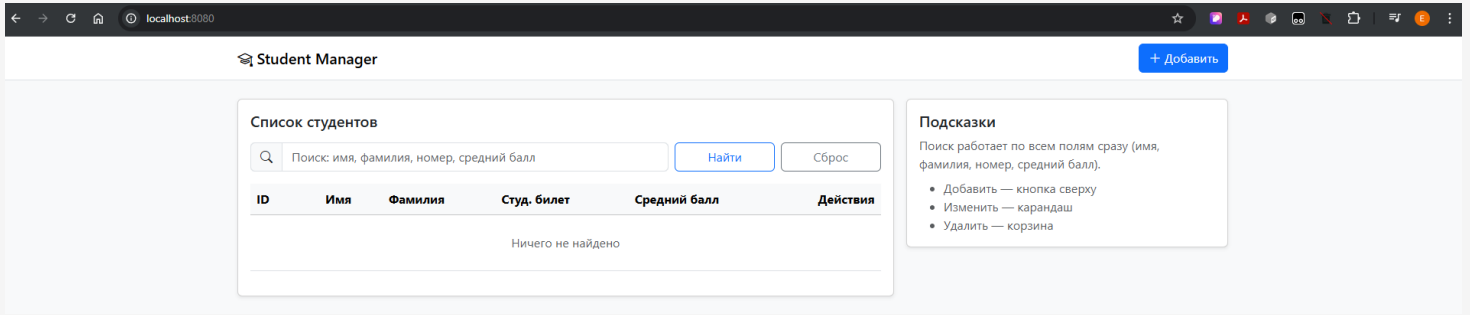
```
    <!-- Правая колонка: подсказки -->
     <div class="col-12 col-lg-4">
       <!-- col-12 — на маленьких экранах 100%
            col-lg-4 — на больших 4/12 -->

       <div class="card shadow-sm">
         <div class="card-body">
           <h5 class="card-title mb-2">Подсказки</h5>

           <p class="text-muted mb-2">
             Поиск работает по всем полям сразу (имя, фамилия, номер, средний балл).
           </p>

           <ul class="mb-0 text-muted">
             <li>Добавить — кнопка сверху</li>
             <li>Изменить — карандаш</li>
             <li>Удалить — корзина</li>
           </ul>
         </div>
       </div>
     </div>

   </div>
</th:block>

</html>
```

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Интерфейс. Часть 7. Index.html – Результат



Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Интерфейс. Часть 8. new_student.html – Добавление студента

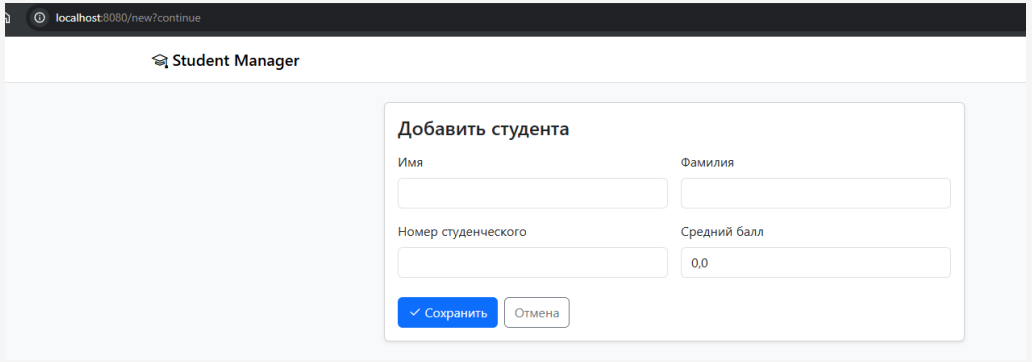
```
<!doctype html>
<!-- Документ HTML5 -->

<html lang="ru" xmlns:th="http://www.thymeleaf.org"
    th:replace="~{layout :: html(
      title='New Student',
      navRight=~{::navRight},
      content=~{::content}
    )}">
<!-- th:replace — вместо этого файла будет подставлен layout.html (его общий шаблон),
    а мы сюда передаем:
    title   — заголовок вкладки,
    navRight — правая часть navbar,
    content  — основной контент страницы -->

<!-- ========================= -->
<!-- Правая часть navbar -->
<!-- ========================= -->
<th:block th:fragment="navRight">
  <!-- Кнопка "Назад" (возврат на список) -->
  <a class="btn btn-outline-secondary" th:href="@{/}">
    <i class="bi bi-arrow-left"></i> Назад
  </a>
</th:block>

<!-- ========================= -->
<!-- Основной контент страницы -->
<!-- ========================= -->
<th:block th:fragment="content">
  <!-- layout уже дает <main class="container py-4">,
       поэтому здесь делаем просто блок по центру нужной ширины -->

  <div class="mx-auto" style="max-width: 720px;">
    <!-- mx-auto — центрирование блока
         max-width — ограничиваем ширину формы, чтобы выглядело аккуратно -->

    <div class="card shadow-sm">
      <div class="card-body">
        <h4 class="card-title mb-3">Добавить студента</h4>

        <!-- th:object="${student}" — связываем форму с объектом student из контроллера -->
        <form th:action="@{/save}" method="post" th:object="${student}">
          <!-- th:action="@{/save}" — отправка формы на /save
               method="post" — данные отправляются POST-запросом -->

          <div class="row g-3">
            <!-- row — сетка Bootstrap
                 g-3 — отступы между полями -->

            <div class="col-12 col-md-6">
              <label class="form-label">Имя</label>
              <input class="form-control"
                     type="text"
                     th:field="*{first}"
                     required
                     maxlength="100">
              <!-- th:field="*{first}" — поле first у объекта student -->
            </div>

            <div class="col-12 col-md-6">
              <label class="form-label">Фамилия</label>
              <input class="form-control"
                     type="text"
                     th:field="*{last}"
                     required
                     maxlength="100">
            </div>

            <div class="col-12 col-md-6">
              <label class="form-label">Номер студенческого</label>
              <input class="form-control"
                     type="text"
                     th:field="*{num}"
                     required
                     maxlength="50">
            </div>

            <div class="col-12 col-md-6">
              <label class="form-label">Средний балл</label>
              <input class="form-control"
                     type="number"
                     step="0.01"
                     min="0"
                     max="5"
                     th:field="*{av}"
                     required>
              <!-- step="0.01" — можно вводить дробные значения -->
            </div>
          </div>

          <div class="d-flex gap-2 mt-4">
            <!-- Кнопка отправки формы -->
            <button class="btn btn-primary" type="submit">
              <i class="bi bi-check2"></i> Сохранить
            </button>

            <!-- Отмена — просто возвращаемся на главную -->
            <a class="btn btn-outline-secondary" th:href="@{/}">Отмена</a>
          </div>
        </form>
      </div>
    </div>
  </div>
</th:block>

</html>
```

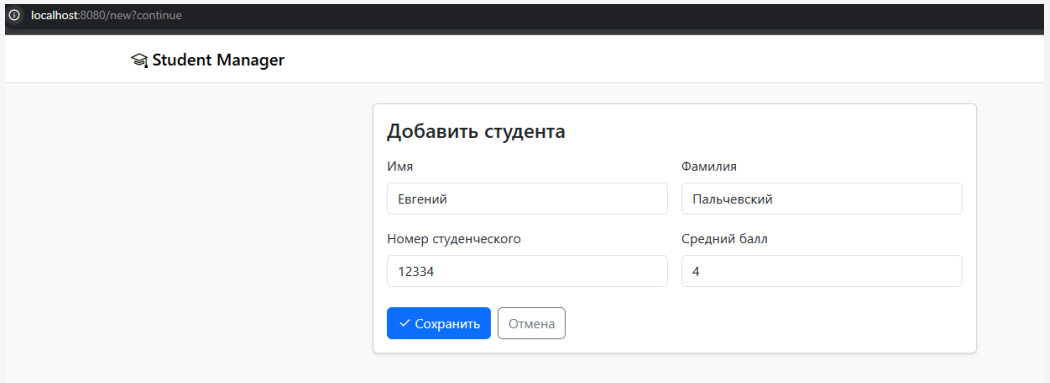
Лекция №1

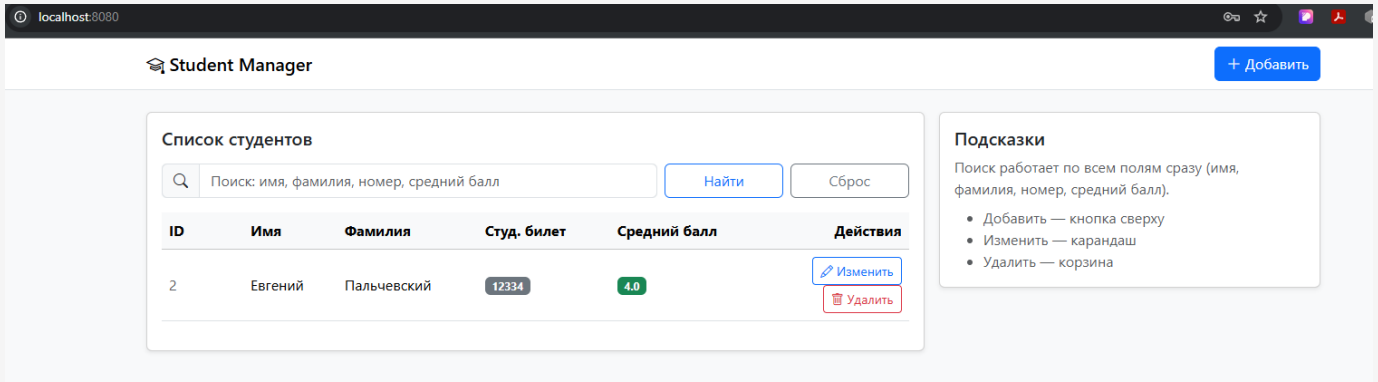
Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Интерфейс. Часть 9. new_student.html – Результат







Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»


```
<!doctype html>
<!-- Документ HTML5 -->

<html lang="ru" xmlns:th="http://www.thymeleaf.org"
    th:replace="~{layout :: html(
      title='Edit Student',
      navRight=~{::navRight},
      content=~{::content}
    )}">
<!-- th:replace — вместо этого файла будет подставлен layout.html (общий шаблон),
    а мы передаем параметры:
    title   — заголовок вкладки,
    navRight — правая часть navbar (кнопка "Назад"),
    content  — основной контент (форма редактирования) -->

<!-- ========================= -->
<!-- Правая часть navbar -->
<!-- ========================= -->
<th:block th:fragment="navRight">
  <!-- Кнопка "Назад" — возвращаемся на главную страницу со списком -->
  <a class="btn btn-outline-secondary" th:href="@{/}">
    <i class="bi bi-arrow-left"></i> Назад
  </a>
</th:block>

<!-- ========================= -->
<!-- Основной контент страницы -->
<!-- ========================= -->
<th:block th:fragment="content">
  <!-- layout уже дает <main class="container py-4">,
       здесь делаем контент по центру ограниченной ширины -->

  <div class="mx-auto" style="max-width: 720px;">
    <!-- mx-auto — центрирование блока
         max-width — чтобы форма была компактной и аккуратной -->

    <div class="card shadow-sm">
      <div class="card-body">
        <h4 class="card-title mb-3">Редактировать студента</h4>

        <!-- th:object="${student}" — связываем форму с объектом student из контроллера -->
        <form th:action="@{/save}" method="post" th:object="${student}">
          <!-- th:action="@{/save}" — отправляем на /save
               method="post" — данные отправляем POST-запросом -->

          <!-- ВАЖНО: id должен отправляться обратно, иначе Spring/JPA может создать новую запись -->
          <input type="hidden" th:field="*{id}">
          <!-- th:field="*{id}" — поле id объекта student -->

          <div class="row g-3">
            <!-- row — сетка Bootstrap
                 g-3 — отступы между полями -->

            <div class="col-12 col-md-6">
              <label class="form-label">Имя</label>
              <input class="form-control"
                     type="text"
                     th:field="*{first}"
                     required
                     maxlength="100">
              <!-- th:field="*{first}" — имя студента -->
            </div>
```

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

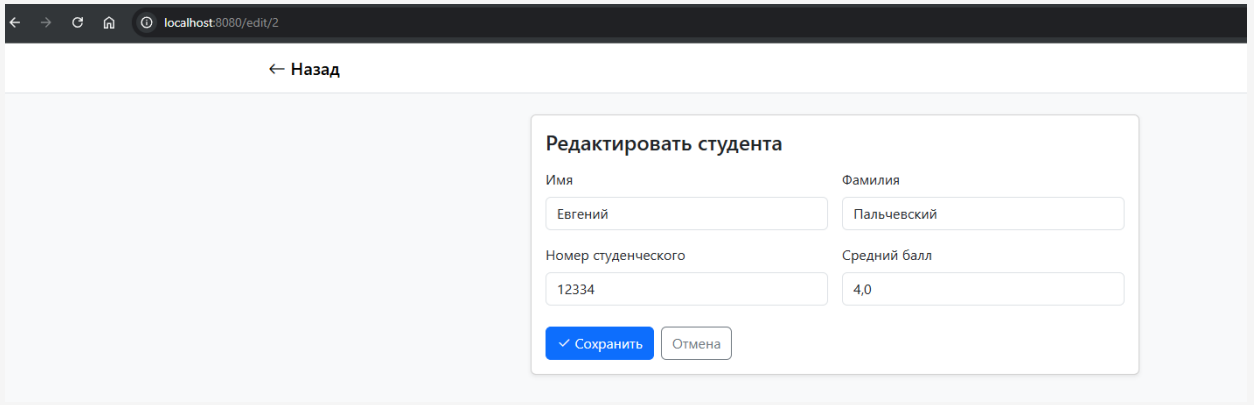

```
            <div class="col-12 col-md-6">
              <label class="form-label">Фамилия</label>
              <input class="form-control"
                     type="text"
                     th:field="*{last}"
                     required
                     maxlength="100">
              <!-- th:field="*{last}" — фамилия студента -->
            </div>

            <div class="col-12 col-md-6">
              <label class="form-label">Номер студенческого</label>
              <input class="form-control"
                     type="text"
                     th:field="*{num}"
                     required
                     maxlength="50">
              <!-- th:field="*{num}" — номер студенческого -->
            </div>

            <div class="col-12 col-md-6">
              <label class="form-label">Средний балл</label>
              <input class="form-control"
                     type="number"
                     step="0.01"
                     min="0"
                     max="5"
                     th:field="*{av}"
                     required>
              <!-- step="0.01" — можно вводить дробные значения -->
            </div>

          </div>

          <div class="d-flex gap-2 mt-4">
            <!-- Сохранить изменения -->
            <button class="btn btn-primary" type="submit">
              <i class="bi bi-check2"></i> Сохранить
            </button>

            <!-- Отмена — не сохраняем, просто возвращаемся на список -->
            <a class="btn btn-outline-secondary" th:href="@{/}">Отмена</a>
          </div>
        </form>

      </div>
    </div>
  </div>
</th:block>

</html>
```

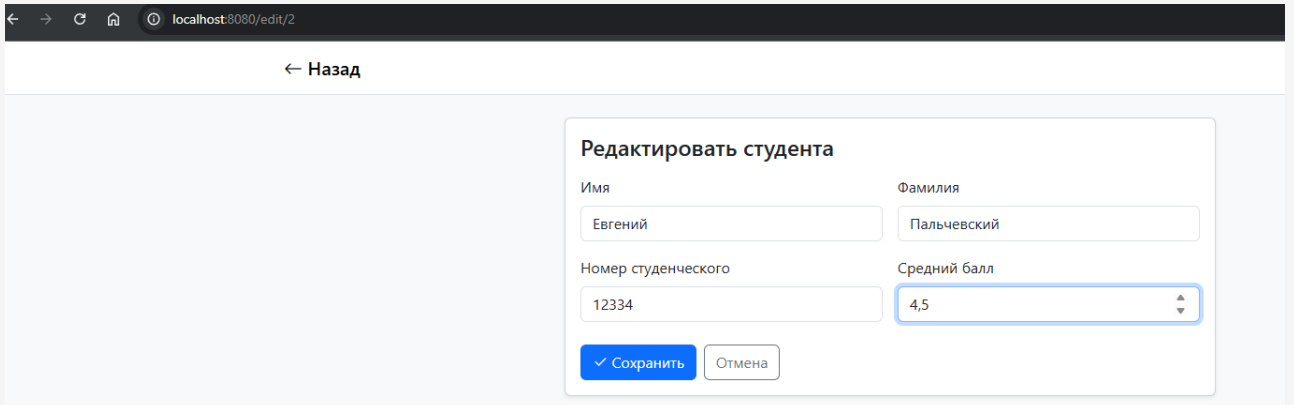
Лекция №1

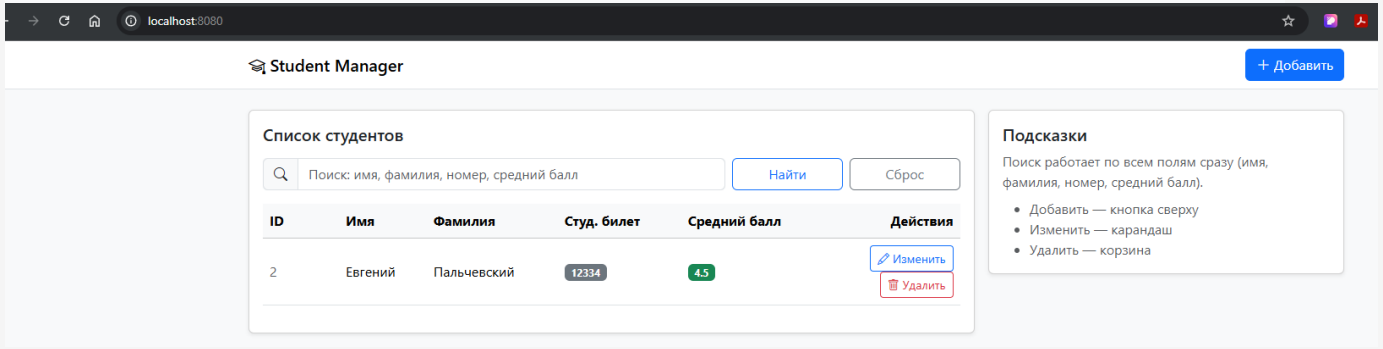
Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Интерфейс. Часть 12. edit_student.html – Результат





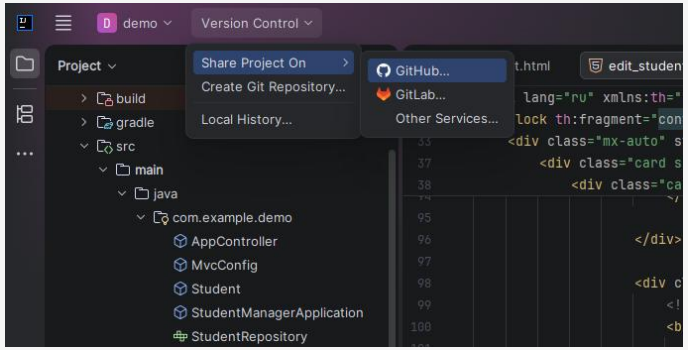


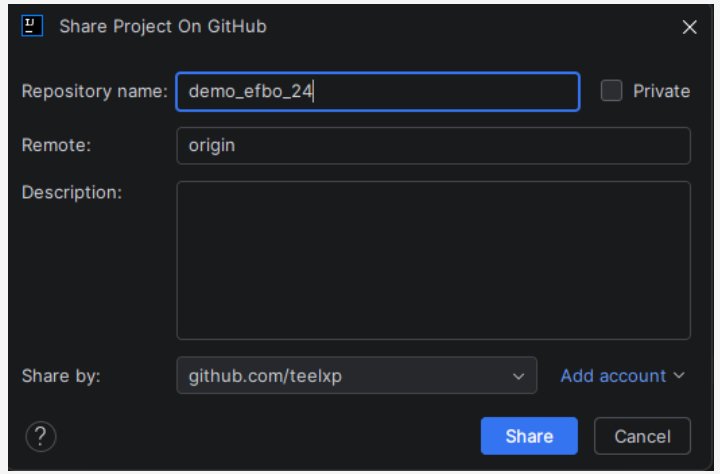
Лекция №1

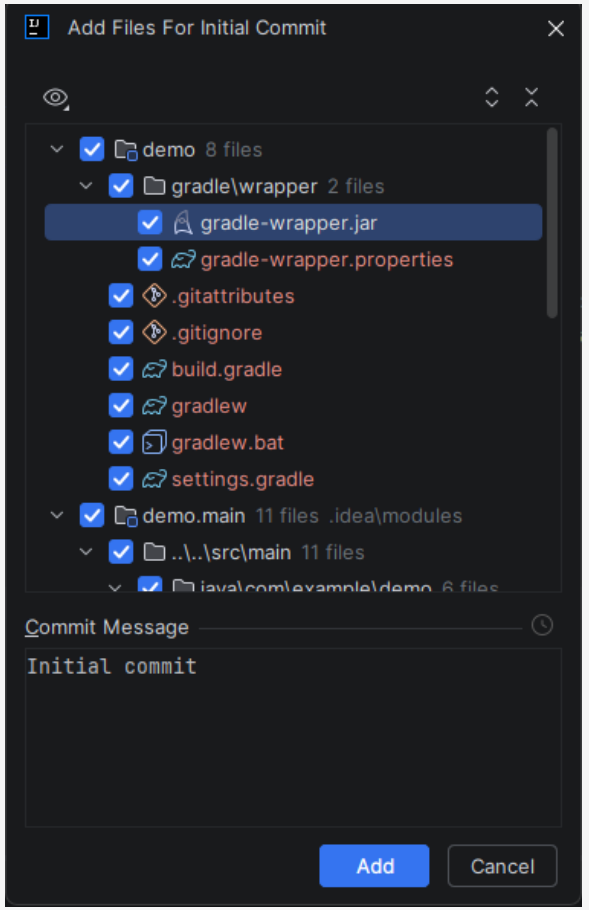
Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Java + Spring + MySQL с использованием архитектурного паттерна MVC. Добавляем проект в гит





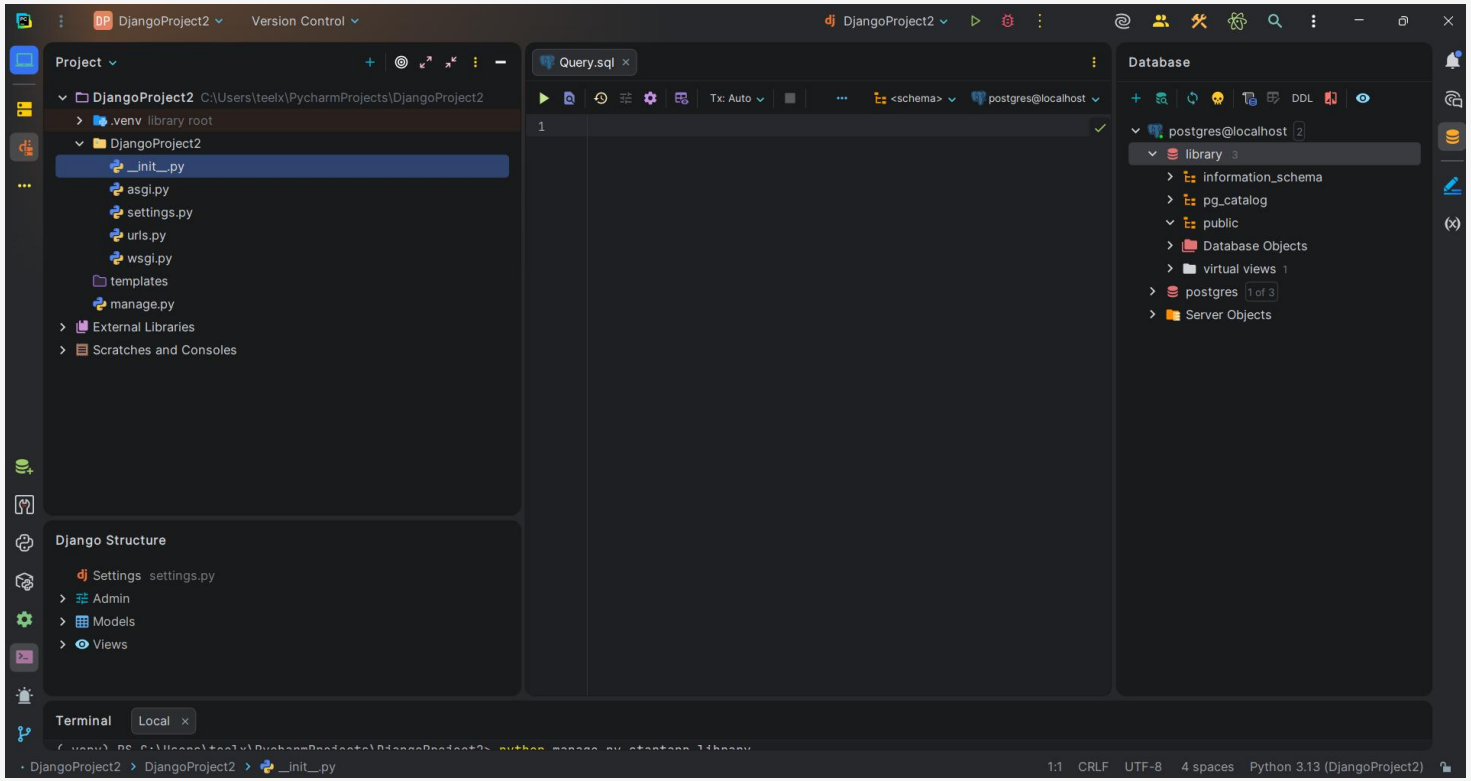


Ссылка на проект: https://github.com/teelxp/demo_efbo_24

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

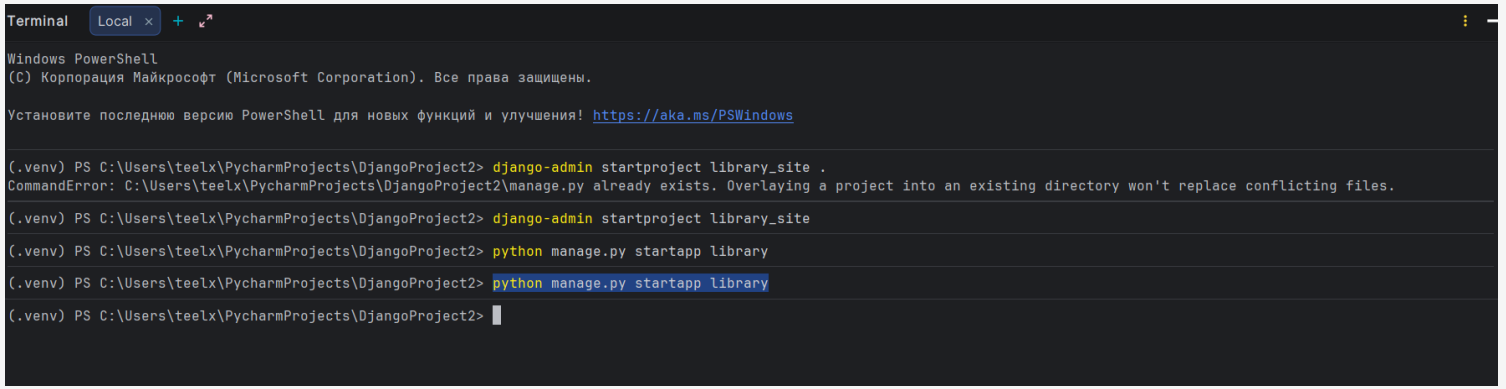


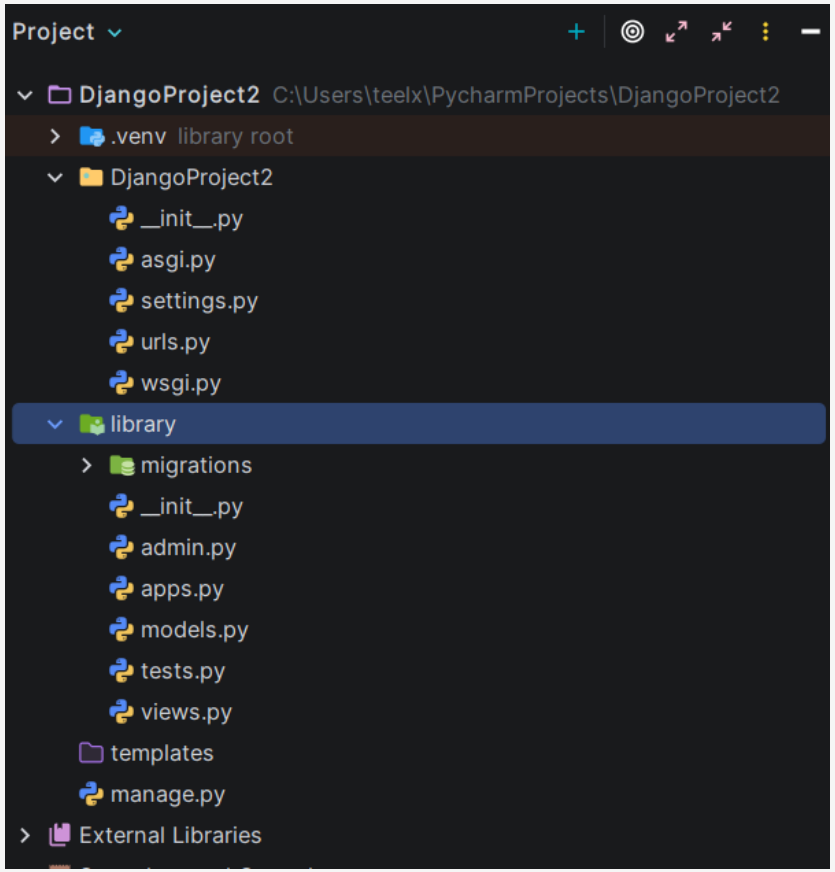
Слева отображается файловая структура проекта, включая основные файлы Django (settings.py, urls.py, manage.py и др.), структуру Django (Settings, Admin, Models, Views) и встроенный терминал. Справа – инструменты для работы с базой данных.

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»



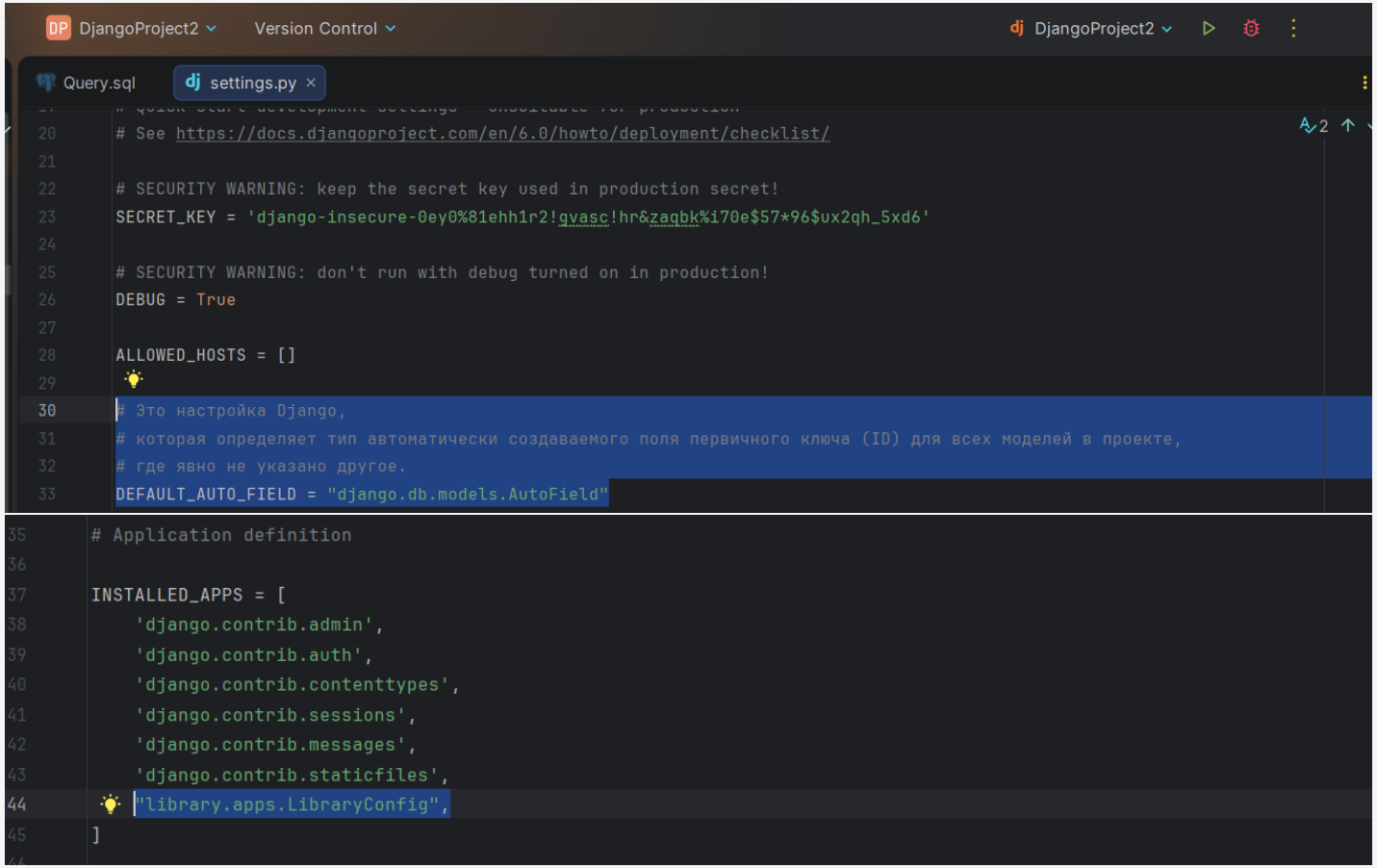


Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

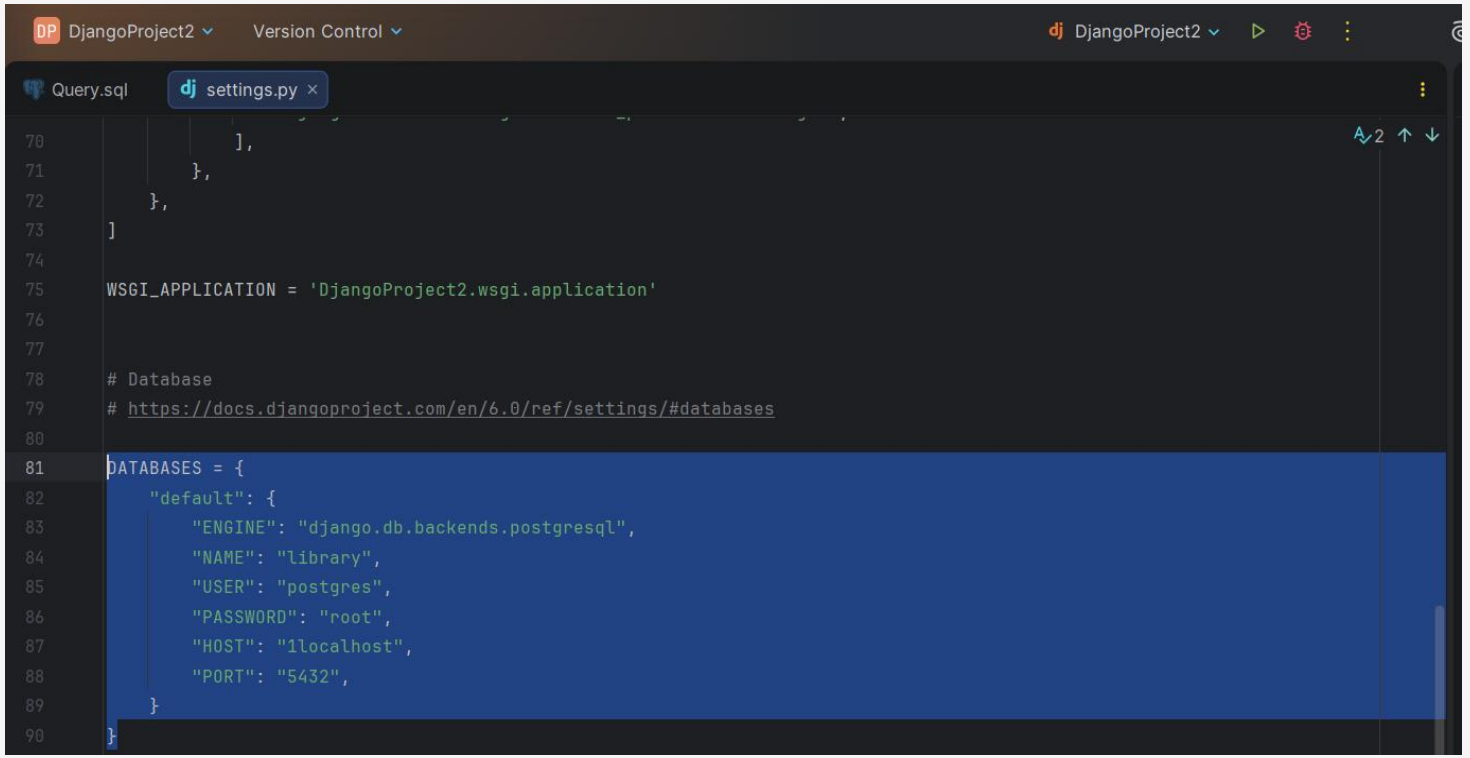
Разработка веб-приложения на примере Python + Django + PostgreSQL с использованием архитектурного паттерна MVC. Подключение раннее созданного приложения в проект



Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»



Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Python + Django + PostgreSQL с использованием архитектурного паттерна MVC. Создание модели базы данных, т.е. таблиц и столбцов

```
from django.db import models  # Импорт базового модуля для создания моделей

from django.db import models  # Повторный импорт (избыточно, можно удалить)

class Book(models.Model):  # Модель "Книга"
    book_id = models.AutoField(primary_key=True, db_column="book_id")  # Автоинкрементный первичный ключ
    title = models.CharField(max_length=200, db_column="title")  # Название книги, строка до 200 символов
    author = models.CharField(max_length=100, db_column="author")  # Автор книги, строка до 100 символов
    year_published = models.IntegerField(db_column="year_published")  # Год публикации, целое число
    available = models.BooleanField(default=True, db_column="available")  # Доступность книги, булево значение (по умолчанию True)

    class Meta:  # Мета-класс для дополнительных настроек модели
        db_table = "books"  # Имя таблицы в БД — "books"

class Reader(models.Model):  # Модель "Читатель"
    reader_id = models.AutoField(primary_key=True, db_column="reader_id")  # Автоинкрементный первичный ключ
    full_name = models.CharField(max_length=100, db_column="full_name")  # ФИО читателя, строка до 100 символов
    passport_data = models.CharField(max_length=20, db_column="passport_data")  # Паспортные данные, строка до 20 символов
    registration_date = models.DateField(db_column="registration_date")  # Дата регистрации, дата

    class Meta:
        db_table = "readers"  # Имя таблицы — "readers"
```

```
class BookLoan(models.Model):  # Модель "Выдача книги"
    loan_id = models.AutoField(primary_key=True, db_column="loan_id")  # Автоинкрементный первичный ключ

    reader = models.ForeignKey(  # Внешний ключ на модель Reader
        Reader, on_delete=models.CASCADE,  # При удалении читателя удаляются все связанные выдачи
        db_column="reader_id", related_name="loans"  # Имя столбца в БД и имя обратной связи
    )
    book = models.ForeignKey(  # Внешний ключ на модель Book
        Book, on_delete=models.CASCADE,  # При удалении книги удаляются все связанные выдачи
        db_column="book_id", related_name="loans"  # Имя столбца и обратной связи
    )

    loan_date = models.DateField(db_column="loan_date")  # Дата выдачи книги
    return_date = models.DateField(null=True, blank=True, db_column="return_date")  # Дата возврата (может быть пустой)
    returned = models.BooleanField(default=False, db_column="returned")  # Факт возврата (по умолчанию False)

    class Meta:
        db_table = "book_loans"  # Имя таблицы — "book_loans"

class AuditLog(models.Model):  # Модель "Журнал аудита"
    log_id = models.AutoField(primary_key=True, db_column="log_id")  # Автоинкрементный первичный ключ
    table_name = models.CharField(max_length=50, db_column="table_name")  # Имя таблицы, где произошли изменения
    action = models.CharField(max_length=10, db_column="action")  # Тип операции (INSERT, UPDATE, DELETE)
    old_data = models.TextField(null=True, blank=True, db_column="old_data")  # Старые данные (могут быть пустыми)
    new_data = models.TextField(null=True, blank=True, db_column="new_data")  # Новые данные (могут быть пустыми)
    change_date = models.DateTimeField(auto_now_add=True, db_column="change_date")  # Дата и время изменения (автоматически)

    class Meta:
        db_table = "audit_log"  # Имя таблицы — "audit_log"
```

Лекция №1

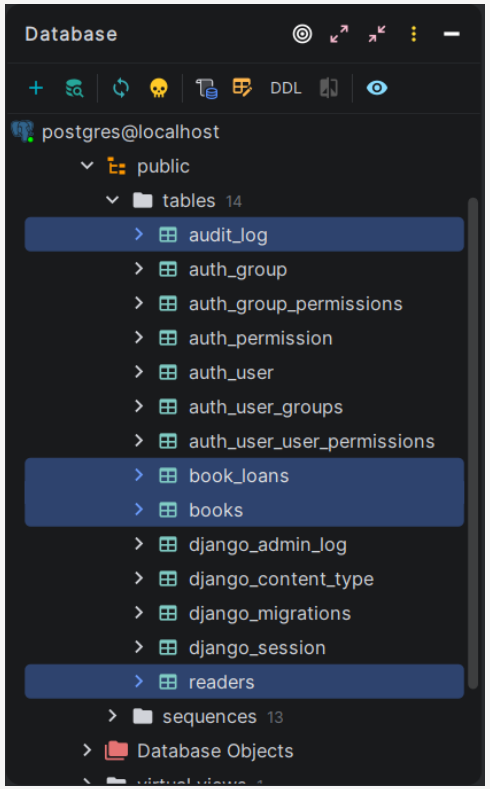
Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»


```
Terminal  Local ×  +

(.venv) PS C:\Users\teelx\PycharmProjects\DjangoProject2> python manage.py makemigrations
C:\Users\teelx\PycharmProjects\DjangoProject2\.venv\Lib\site-packages\django\core\management\commands\makemigrations.py:162: RuntimeWarning: Got an error checking a consistent migr
ation history performed for database connection 'default': could not translate host name "1localhost" to address: Name or service not known

  warnings.warn(
Migrations for 'library':
  library\migrations\0001_initial.py
    + Create model AuditLog
    + Create model Book
    + Create model Reader
    + Create model BookLoan
```

Команда ***python manage.py makemigrations*** в Django используется для создания файлов миграций, которые описывают изменения, внесенные в ваши модели (models.py), чтобы синхронизировать их с схемой базы данных. Это первый шаг, который упаковывает изменения (например, добавление поля, удаление модели) в файлы, а затем команда ***python manage.py migrate*** применяет эти изменения к самой базе данных.

```
(.venv) PS C:\Users\teelx\PycharmProjects\DjangoProject2> python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, library, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying library.0001_initial... OK
  Applying sessions.0001_initial... OK

(.venv) PS C:\Users\teelx\PycharmProjects\DjangoProject2>
```



Лекция №1

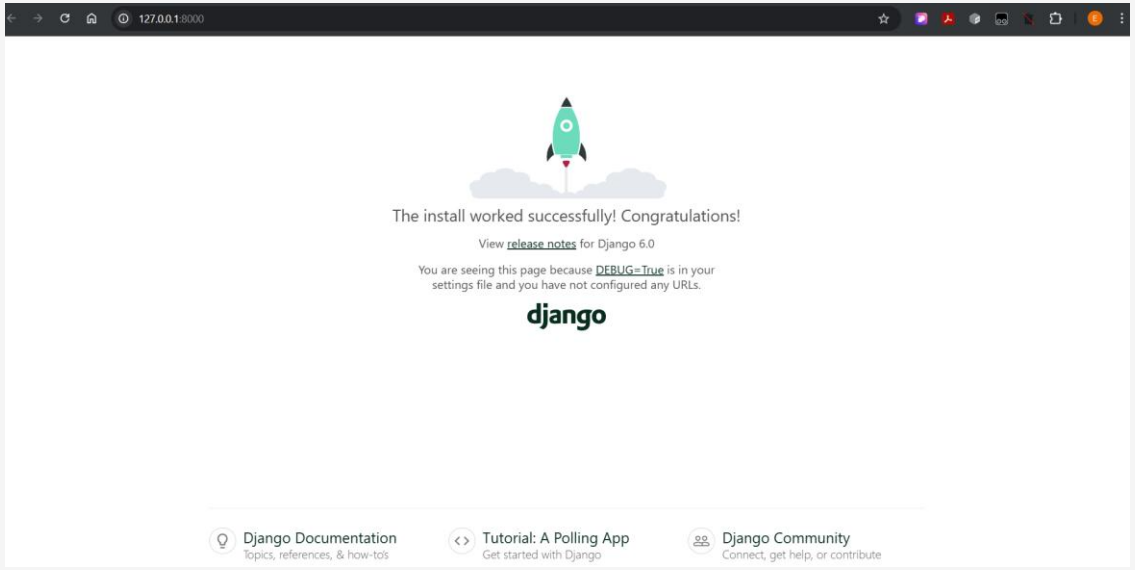
Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Python + Django + PostgreSQL с использованием архитектурного паттерна MVC. Создание суперпользователя и запуск проекта. Часть 1

```
(.venv) PS C:\Users\teelx\PycharmProjects\DjangoProject2> python manage.py createsuperuser
Username (leave blank to use 'teelx'): root
Email address: teelxp@inbox.ru
Password:
Password (again):
The password is too similar to the username.
This password is too short. It must contain at least 8 characters.
This password is too common.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

```
(.venv) PS C:\Users\teelx\PycharmProjects\DjangoProject2> python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
December 16, 2025 - 16:02:59
Django version 6.0, using settings 'DjangoProject2.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

WARNING: This is a development server. Do not use it in a production setting. Use a production WSGI or ASGI server instead.
For more information on production servers see: https://docs.djangoproject.com/en/6.0/howto/deployment/
```

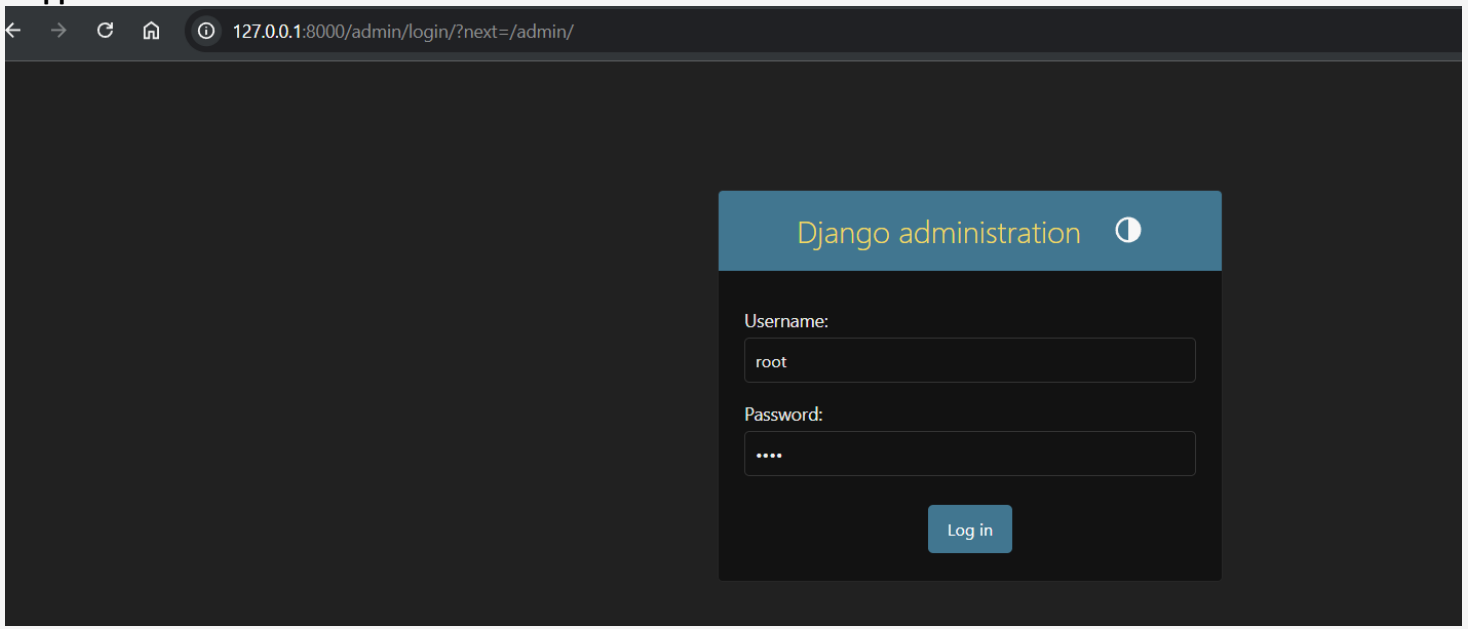


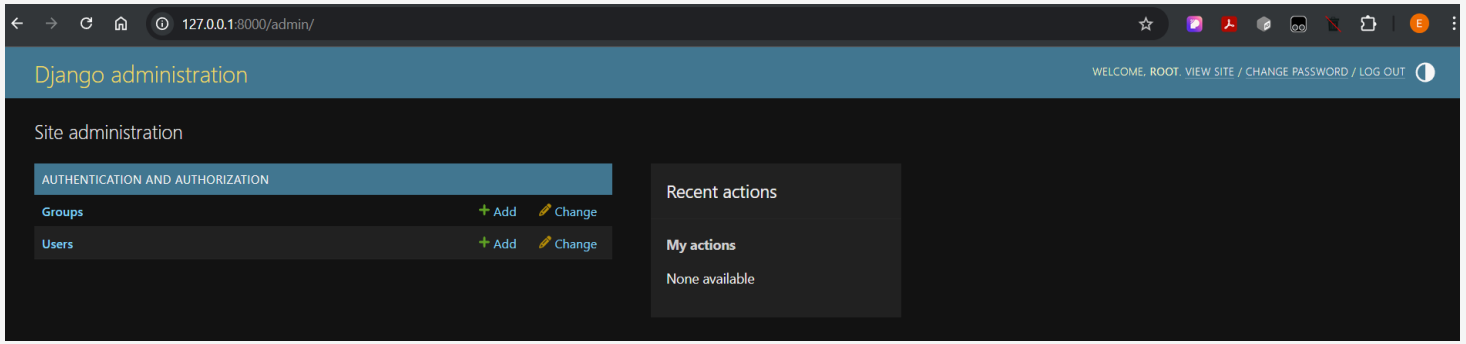
Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Python + Django + PostgreSQL с использованием архитектурного паттерна MVC. Создание суперпользователя и запуск проекта. Часть 2. Админка





Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка модуля (***signals.py***) записи в журнал аудита (AuditLog) всех изменений (***создание, обновление, удаление***) в моделях Book, Reader и BookLoan.

```python
# Импорт необходимых модулей:
import json  # Для работы с JSON
from django.db.models.signals import pre_save, post_save, pre_delete, post_delete  # Сигналы Django
from django.dispatch import receiver  # Декоратор для подключения обработчиков сигналов
from django.forms.models import model_to_dict  # Преобразование объекта модели в словарь
from .models import Book, Reader, BookLoan, AuditLog  # Импорт моделей текущего приложения

# Вспомогательная функция для преобразования объекта в JSON-строку
def _to_json(obj) -> str:
    return json.dumps(obj, ensure_ascii=False, default=str)  # ensure_ascii=False для корректного отображения кириллицы

# Вспомогательная функция для преобразования экземпляра модели в словарь
def _as_dict(instance):
    return model_to_dict(instance)  # Использует встроенный метод Django

# Обработчик сигнала pre_save (срабатывает перед сохранением объекта)
# Регистрируется для трех моделей: Book, Reader, BookLoan
@receiver(pre_save, sender=Book)
@receiver(pre_save, sender=Reader)
@receiver(pre_save, sender=BookLoan)
def capture_old_instance(sender, instance, **kwargs):
    # Если у объекта еще нет первичного ключа (новый объект) - сохраняем None
    if not instance.pk:
        instance._old_instance = None  # Создаем временный атрибут для хранения старой версии
        return
    # Если объект уже существует, пытаемся получить его старую версию из БД
    try:
        instance._old_instance = sender.objects.get(pk=instance.pk)
    except sender.DoesNotExist:
        instance._old_instance = None  # Если не найден (редкий случай) - сохраняем None

# Обработчик сигнала post_save (срабатывает после сохранения объекта)
@receiver(post_save, sender=Book)
@receiver(post_save, sender=Reader)
@receiver(post_save, sender=BookLoan)
def write_audit_on_save(sender, instance, created, **kwargs):
    # Получаем старую версию объекта из временного атрибута
    old_inst = getattr(instance, "_old_instance", None)
    # Создаем запись в журнале аудита:
    AuditLog.objects.create(
        table_name=sender._meta.db_table,  # Имя таблицы модели (например, "books")
        action="INSERT" if created else "UPDATE",  # Тип операции: вставка или обновление
        # Старые данные (None для новой записи, иначе JSON старой версии)
        old_data=None if old_inst is None else _to_json(_as_dict(old_inst)),
        new_data=_to_json(_as_dict(instance)),  # Новые данные в JSON
    )
```

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»


```python
# Обработчик сигнала pre_delete (срабатывает перед удалением объекта)
@receiver(pre_delete, sender=Book)
@receiver(pre_delete, sender=Reader)
@receiver(pre_delete, sender=BookLoan)
def capture_old_before_delete(sender, instance, **kwargs):
    # Сохраняем текущий объект как старую версию (перед удалением)
    instance._old_instance = instance

# Обработчик сигнала post_delete (срабатывает после удаления объекта)
@receiver(post_delete, sender=Book)
@receiver(post_delete, sender=Reader)
@receiver(post_delete, sender=BookLoan)
def write_audit_on_delete(sender, instance, **kwargs):
    # Получаем старую версию (сохраненную перед удалением)
    old_inst = getattr(instance, "_old_instance", None)
    # Создаем запись в журнале аудита об удалении:
    AuditLog.objects.create(
        table_name=sender._meta.db_table,
        action="DELETE",  # Тип операции - удаление
        old_data=None if old_inst is None else _to_json(_as_dict(old_inst)),  # Данные удаленного объекта
        new_data=None,  # Новых данных нет (объект удален)
    )
```

Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»


```python
from django.contrib import admin
from .models import Book, Reader, BookLoan, AuditLog

@admin.register(Book)
class BookAdmin(admin.ModelAdmin):
    list_display = ("book_id_ru", "title_ru", "author_ru", "year_ru",
"available_ru")
    list_filter = ("available", "year_published")
    search_fields = ("title", "author")

    @admin.display(description="ID")
    def book_id_ru(self, obj): return obj.book_id

    @admin.display(description="Название")
    def title_ru(self, obj): return obj.title

    @admin.display(description="Автор")
    def author_ru(self, obj): return obj.author

    @admin.display(description="Год издания")
    def year_ru(self, obj): return obj.year_published

    @admin.display(description="Доступна")
    def available_ru(self, obj): return obj.available


@admin.register(Reader)
class ReaderAdmin(admin.ModelAdmin):
    list_display = ("reader_id_ru", "full_name_ru", "passport_ru",
"reg_date_ru")
    search_fields = ("full_name", "passport_data")

    @admin.display(description="ID")
    def reader_id_ru(self, obj): return obj.reader_id

    @admin.display(description="ФИО")
    def full_name_ru(self, obj): return obj.full_name

    @admin.display(description="Паспортные данные")
    def passport_ru(self, obj): return obj.passport_data

    @admin.display(description="Дата регистрации")
    def reg_date_ru(self, obj): return obj.registration_date
```

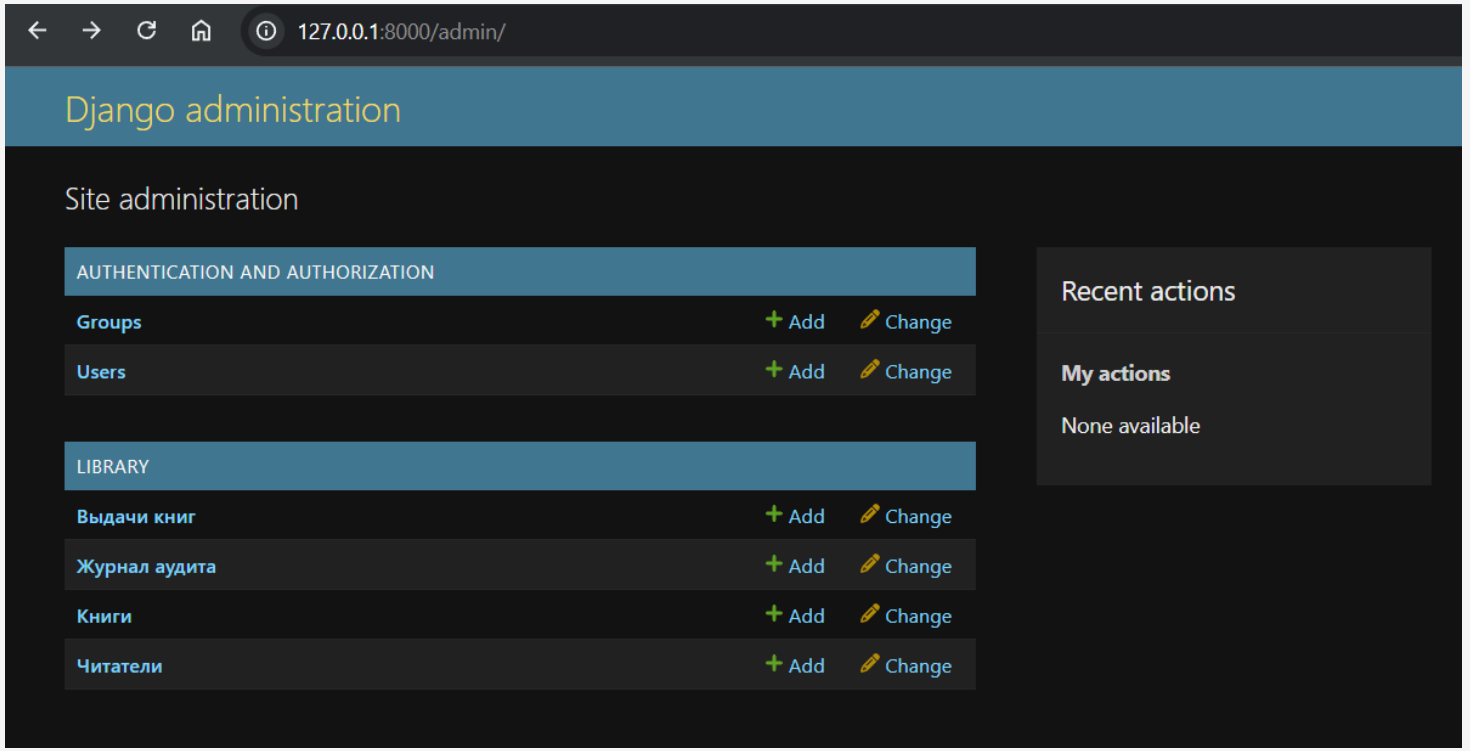
```python
@admin.register(BookLoan)
class BookLoanAdmin(admin.ModelAdmin):
    list_display = ("loan_id_ru", "reader_ru", "book_ru",
"loan_date_ru", "return_date_ru", "returned_ru")
    list_filter = ("returned", "loan_date")
    search_fields = ("reader__full_name", "book__title")

    @admin.display(description="ID")
    def loan_id_ru(self, obj): return obj.loan_id

    @admin.display(description="Читатель")
    def reader_ru(self, obj): return obj.reader

    @admin.display(description="Книга")
    def book_ru(self, obj): return obj.book

    @admin.display(description="Дата выдачи")
    def loan_date_ru(self, obj): return obj.loan_date

    @admin.display(description="Дата возврата")
    def return_date_ru(self, obj): return obj.return_date

    @admin.display(description="Возвращена")
    def returned_ru(self, obj): return obj.returned


@admin.register(AuditLog)
class AuditLogAdmin(admin.ModelAdmin):
    list_display = ("log_id_ru", "change_date_ru", "table_ru",
"action_ru")
    list_filter = ("table_name", "action")
    search_fields = ("table_name", "action", "old_data", "new_data")

    @admin.display(description="ID")
    def log_id_ru(self, obj): return obj.log_id

    @admin.display(description="Дата/время")
    def change_date_ru(self, obj): return obj.change_date

    @admin.display(description="Таблица")
    def table_ru(self, obj): return obj.table_name

    @admin.display(description="Действие")
    def action_ru(self, obj): return obj.action
```

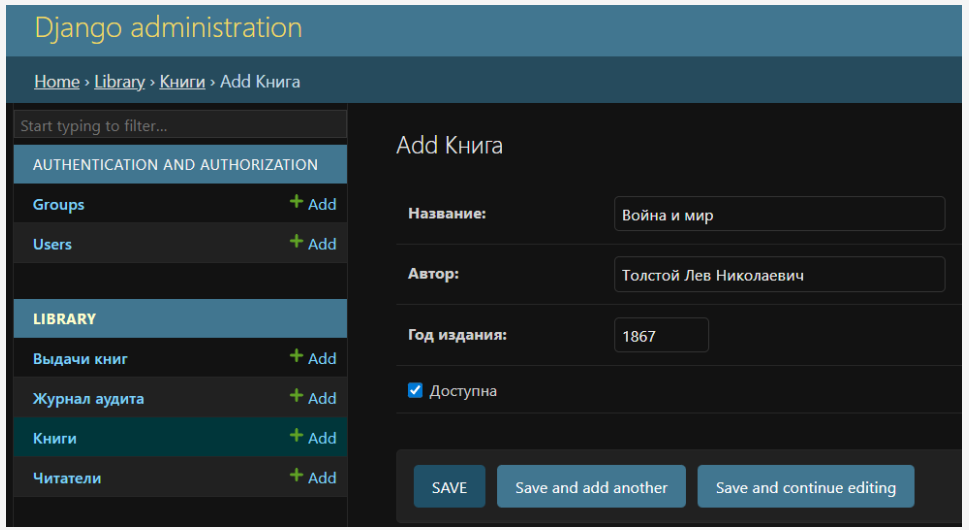
Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Python + Django + PostgreSQL с использованием архитектурного паттерна MVC. Разработка модуля записи логов. Часть 4 – результат (#1)



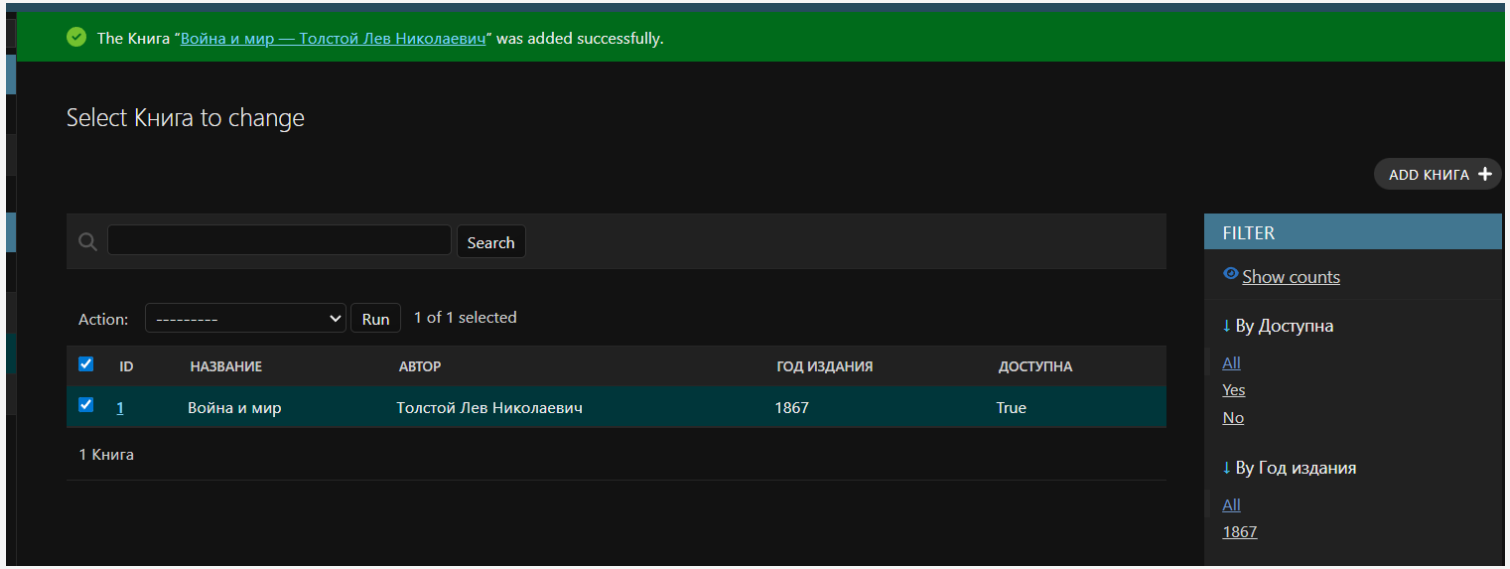


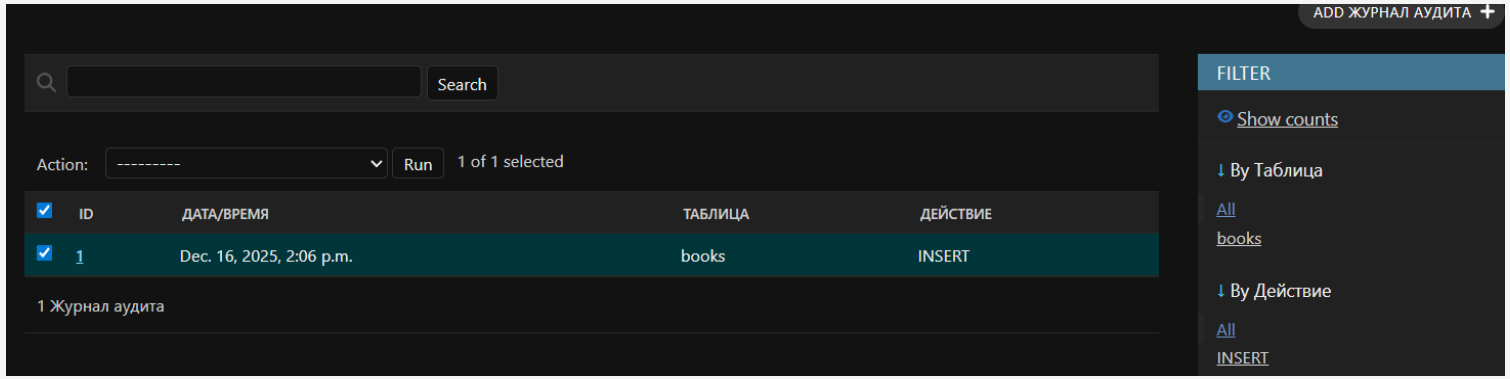
Лекция №1

Тема: «**Структура семестра. Форма отчетности. Инструменты. Git**»

Подтема: «Разработка веб-приложений с использованием Git»

Разработка веб-приложения на примере Python + Django + PostgreSQL с использованием архитектурного паттерна MVC. Разработка модуля записи логов. Часть 4 – результат (#2)





Лекция №2

Тема: «**Технологии управления проектами. Профессия тимлида и аналитика**»

Подтема: «Формирование отчета и постановка задач с использованием различных нотаций и технологий моделирования»

Формирование отчета по курсовому проекту

Инструкция по написанию отчета.

1. Открываем последний слайд данной презентации и видим следующее:

Контрольные работы

№ Контрольной работы	Что должно быть готово (результат)	Что сдаете	Минимум для баллов (как проверяется)
Контрольная работа №1: Анализ + требования (без проектирования и без кода)	Понятно, что именно автоматизируем и какие функции обязаны быть, включая AI-функцию как часть процесса	1) Описание предметной области «как есть» + проблемы/узкие места 2) Цель/задачи/назначение системы (кратко) 3) Роли + матрица прав (кто что может) 4) Функциональные требования (что умеет система) + минимум 3 сквозных бизнес-процесса (пошагово) 5) AI-функция (постановка): тип задачи, входные данные, выход, где используется в процессе, критерий качества (хотя бы простой)	Проверяю, что: а) процессы понятны и реализуемы; б) роли не противоречат процессам; в) AI встроена в процесс (а не «где-то будет модель»)
Контрольная работа №2: Готовый backend	Рабочий backend без фронта: БД + API + роли/права + журнал + AI-endpoint	1) Репозиторий/архив проекта 2) Скрипты/миграции БД + тестовые данные 3) Swagger/OpenAPI или перечень эндпоинтов 4) Auth (рег/логин), CRUD, поиск/фильтрация, бизнес-операции процессов, журнал действий 5) AI-endpoint: запрос → обработка → ответ	Проект запускается, 3 процесса выполняются через API, роли реально ограничивают доступ, AI-endpoint возвращает понятный результат
Контрольная работа №3: Полностью реализованная система (backend + frontend) + проектирование	Рабочая система «под ключ»: UI ↔ API ↔ БД, можно пройти сценарии мышкой, AI показывается пользователю. И здесь же сдается пакет проектирования, соответствующий текущей реализации	1) Все из этапа №2 + фронт (репозиторий/архив) 2) UI-экраны: вход/регистрация, основной раздел, формы/карточки, поиск/фильтры, экран/виджет AI, журнал/админ (если предусмотрено) 3) Демонстрация: минимум 2 процесса через UI + AI-результат в UI 4) акет проектирования (теперь тут): ER-схема (финальная), Use Case, 1–2 Activity, (при необходимости) Component/Deployment, + макеты ключевых экранов (как «как должно быть», но уже согласовано с тем, что реально сделано)	Проверяю «сквозняк»: UI не заглушки, роли работают на UI и API, процессы проходят, AI вызывается из UI. Диаграммы/ER/макеты не расходятся с реализацией
Контрольная работа №4: Тестирование + документация	Доказана работоспособность и понятна эксплуатация (чтобы другой человек мог запустить и пользоваться)	1) Протокол тестирования (факт): backend ≥10, frontend ≥10, сквозные ≥5 (проверка/ожидаемо/факт/дефекты) 2) Документация: руководство пользователя, руководство администратора (если есть), описание AI (данные/метрика/интерпретация), инструкция запуска	По инструкции реально поднять систему; тесты/проверки привязаны к требованиям и процессам; документация написана понятно
Контрольная работа №5: Сдача проекта вместе с отчетом КП **После КР №5 идет защита проекта перед комиссией**	Финальный комплект: рабочий проект + полный отчет КП + приложения	1) Финальная версия проекта 2) Финальная версия отчета КП (по вашей структуре) 3) Приложения: диаграммы/скрины, протокол тестирования, инструкция запуска, описание API, описание AI	Отчет соответствует тому, что в коде; все разделы и приложения на месте; нет противоречий между требованиями/диаграммами/реализацией

2. Выбираем № этапа (контрольной работы) и смотрим на столбцы «Что сдаете?» и «Минимум для баллов». И вникаем в написанное ☺

3. Далее открываем методические указания: https://palchevsky.ru/uploads/spo/Methods.pdf и смотрим пункт 2.1 – «Структура отчета». Отчет необходимо оформлять строго по указанной структуре. Объем работы не должен быть меньше установленного минимума. При необходимости допускается превышение рекомендованного максимума, но в разумных пределах (без искусственного увеличения текста). Контрольная работа № 1 выполняется по таблице из пункта 2.1 «Структура отчета» методических указаний — включительно до раздела 1.3 (т.е. все элементы структуры отчета от начала таблицы до пункта 1.3 включительно). При этом в текст пояснительной записки (отчета) курсового проекта обязательно должны быть включены все пункты, предусмотренные заданием КР № 1.

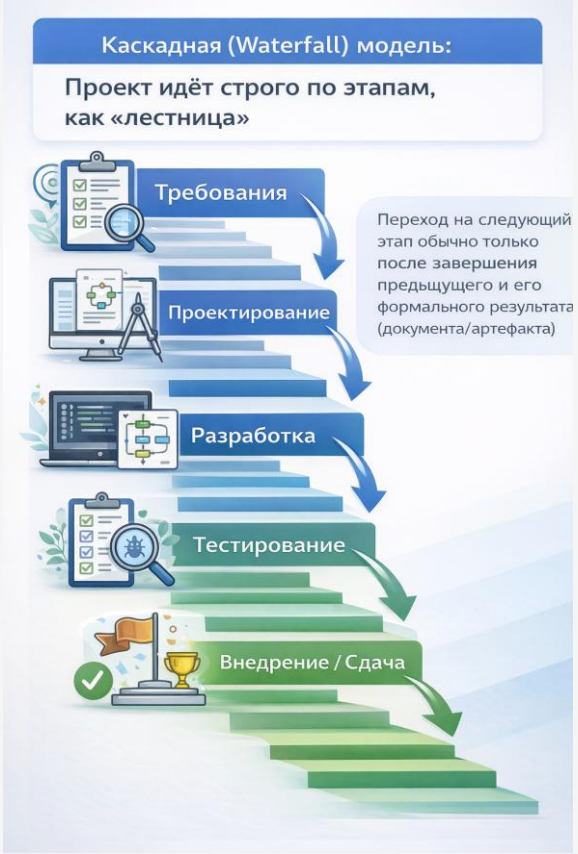
Остальные разделы добавляются в отчет по мере выполнения курсового проекта. Например, после реализации бэкенда вы дополняете отчет всеми разделами таблицы структуры до раздела 2 «Проектирование информационной системы» (раздел 2 не включается). И так далее.

Лекция №2

Тема: «**Технологии управления проектами. Профессия тимлида и аналитика**»

Подтема: «Формирование отчета и постановка задач с использованием различных технологий моделирования»

Каскадная модель жизненного цикла ПО (Waterfall)



Waterfall — это подход, где проект идет строго по этапам, как «лестница»:

1. Сначала требования.
2. Потом проектирование.
3. Затем разработка.
4. Затем тестирование.
5. Затем внедрение/сдача.

Переход на следующий этап обычно происходит после завершения предыдущего и его формального результата (документа/артефакта, т.е. программный элемент системы или система в целом).

Когда подходит?

Если:

1. Требования относительно стабильны (или их можно «заморозить»).
2. Важна документированность (учебные отчеты, ГОСТ-подобные требования).
3. Нужен понятный план и контроль этапов (в т.ч. для комиссии/заказчика).

Не очень подходит, если требования часто меняются и нужно быстро проверять новые идеи и решения: в Waterfall изменения, появившиеся после утверждения требований, обычно приводят к возврату на предыдущие этапы и переработке уже сделанных документов, дизайна и кода. Из-за этого растут сроки и стоимость, а обратная связь получается поздно — «рабочую» версию пользователь видит только ближе к концу. В таких условиях эффективнее итеративные подходы (Agile/Scrum/Kanban), прототипирование/MVP или спиральная модель, где идеи проверяются короткими циклами и требования уточняются по результатам.

Лекция №2

Тема: «**Технологии управления проектами. Профессия тимлида и аналитика**»

Подтема: «Формирование отчета и постановка задач с использованием различных технологий моделирования»

Этапы Waterfall и результаты (артефакты). Часть 1

Этап 1 — Анализ требований

Цель: понять, *что именно* делаем и *зачем*.
Главная роль: Тимлид + системный аналитик.
Артефакты:

- постановка задачи, цель, ограничения;
- список функциональных требований (что система умеет);
- нефункциональные требования (производительность, безопасность, удобство и т.д.);
- сценарии использования;
- критерии приемки (как поймем, что готово).

Этап 2 — Проектирование (архитектура и дизайн)

Цель: решить, *как именно* это будет устроено.
Главные роли: Архитектор + (частично) Backend/Frontend + системный аналитик.
Артефакты:

- архитектурная схема (компоненты, связи);
- модель данных / схема БД (таблицы, связи);
- спецификация API (эндпоинты, форматы запросов/ответов);
- прототип интерфейса (черновые макеты/экраны).

Этап 3 — Реализация (разработка)

Цель: написать код и собрать работающее приложение.
Главные роли: Backend + Frontend, Архитектор консультирует, SA контролирует соответствие требованиям.
Артефакты:

- репозиторий с кодом;
- реализованные модули;
- сборка/запуск проекта;
- инструкции по запуску.

Лекция №2

Тема: «**Технологии управления проектами. Профессия тимлида и аналитика**»

Подтема: «Формирование отчета и постановка задач с использованием различных технологий моделирования»

Этапы Waterfall и результаты (артефакты). Часть 2

Этап 4 — Тестирование

Цель: доказать, что система работает правильно и стабильно.
Главная роль: QA – тестировщик (при участии разработчиков).
Артефакты:

- написанные тесты (Unit-тесты и автотесты);
- баг-репорты;
- отчет о тестировании.

Автотесты — это *общий термин*: любые тесты, которые запускаются автоматически (вручную не «кликаем» по сценарию).
Unit-тесты (UNIT) — это *конкретный вид автотестов*: проверяют одну маленькую единицу кода (функцию/метод/класс) в изоляции.

Unit:

- «calculateDiscount(1000, VIP) должно вернуть 900»;
- «валидатор email должен отклонять abc@»;
- «метод parseDate() должен правильно обработать формат».

Автотест, но не unit (например API/интеграционный):

- «POST /users создает пользователя в базе и возвращает 201»;
- «логин выдает токен, токен дает доступ к /profile»;
- «заказ создается, статус меняется, запись появилась в таблице Orders».

Этап 5 — Сдача проекта / внедрение

Цель: передать результат «заказчику» (преподавателю/комиссии).
Роли: все, но координация у Тимлида+SA.
Артефакты:

- финальная версия проекта;
- презентация;
- пояснительная записка/отчет;
- демонстрация.

Лекция №2

Тема: «**Технологии управления проектами. Профессия тимлида и аналитика**»

Подтема: «Формирование отчета и постановка задач с использованием различных технологий моделирования»

Диаграмма Ганта

Диаграмма Ганта — это календарный план проекта, где:

- по вертикали — задачи;
- по горизонтали — время (дни/недели);
- каждая задача — полоска с датой начала и окончания;
- можно видеть параллельность, зависимости, вехи.

Зачем нужна:

- понятно, *кто что делает и когда*;
- видно, что можно делать параллельно, а что нельзя;
- легко контролировать отставания и «узкие места».

Из чего состоит диаграмма Ганта (минимальный набор)

1. Список задач (крупно → потом дробим).
2. Длительность каждой задачи.
3. Ответственный (роль/ФИО).
4. Зависимости («это можно начать только после того»).
5. Вехи (milestones) — контрольные точки без длительности (например: «Требования утверждены»).

Лекция №2

Тема: «**Технологии управления проектами. Профессия тимлида и аналитика**»

Подтема: «Формирование отчета и постановка задач с использованием различных технологий моделирования»

Диаграмма Ганта

№	Задача	Роль (ответственный)	Длительность	Зависит от № Задачи	Результат
1	Анализ предметной области, сбор требований, границы проекта	Тимлид+SA	4 дня	—	Документ требований
2	Сценарии использования + критерии приемки проекта	Тимлид+SA	3 дня	1	Use Cases / критерии
3	Архитектура решения (компоненты)	Архитектор	3 дня	2	Схема архитектуры
4	Проектирование БД (ER + таблицы)	Архитектор	4 дня	2	Схема БД
5	Спецификация API	Архитектор + Backend	3 дня	3,4	Список эндпоинтов
6	Прототип интерфейса (экраны)	Frontend	4 дня	2,5	Макеты/прототип
7	Реализация каркаса backend (проект, авторизация, БД)	Backend	5 дней	4,5	Запуск + база
8	Реализация основных эндпоинтов	Backend	8 дней	7	CRUD, бизнес-логика
9	Каркас frontend + маршрутизация	Frontend	5 дней	6	Страницы/навигация
10	Интеграция frontend с API	Frontend	8 дней	8,9	Рабочие формы
11	Подготовка тест-плана и тест-кейсов	QA	5 дней	2,5	Тест-документация
12	Тестирование функционала + баг-репорты	QA	8 дней	10	Список дефектов
13	Исправление багов + полировка	Backend+Frontend	6 дней	12	Стабильная сборка
14	Регрессия (повторная проверка)	QA	4 дня	13	«Готово к сдаче»
15	Подготовка отчета и презентации	Тимлид+SA (все помогают)	4 дня	14	Отчет + слайды

Лекция №2

Тема: «**Технологии управления проектами. Профессия тимлида и аналитика**»

Подтема: «Формирование отчета и постановка задач с использованием различных технологий моделирования»

Диаграмма Ганта

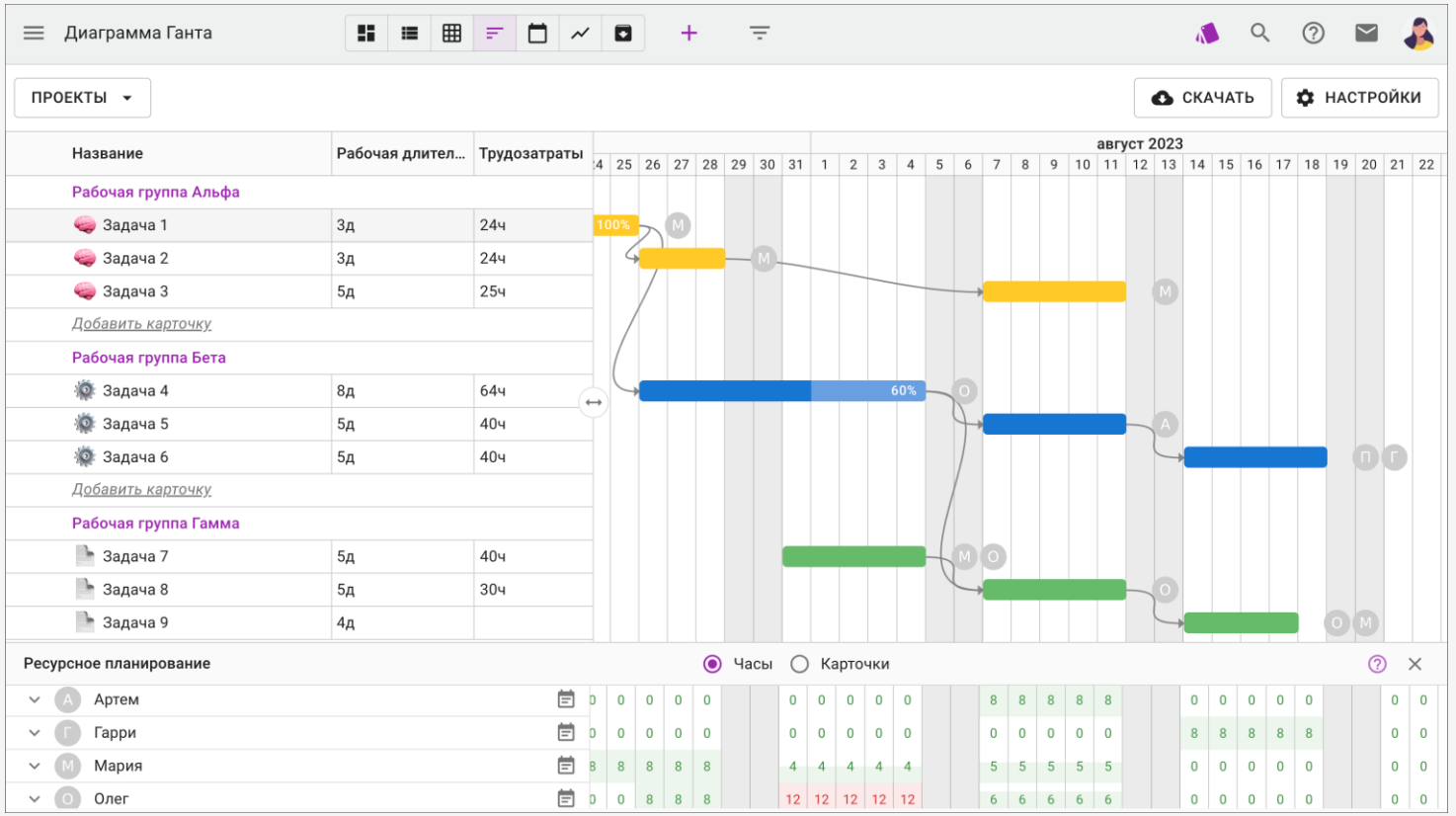


Рисунок 1 – Диаграмма Ганта на примере платформы «Kaiten» https://kaiten.ru/

Но можно использовать даже Microsoft Excel.

Лекция №2

Тема: «**Технологии управления проектами. Профессия тимлида и аналитика**»

Подтема: «Формирование отчета и постановка задач с использованием различных технологий моделирования»

Методология функционального моделирования IDEF0

IDEFo — это методология, позволяющая **описать функцию/работу** в виде блоков «что делаем» и «за счет чего». Каждый блок — это **функция (действие)**, у которой есть 4 вида стрелок:

•**I (Input, Вход)** — *что перерабатываем* (данные на входе, которые превращаются в результат).

•**C (Control, Управление)** — *по каким правилам делаем* (требования, регламенты, ТЗ, ГОСТ, политика, критерии).

•**O (Output, Выход)** — *что получаем* (результаты, документы, продукт).

•**M (Mechanism, Механизм)** — *кто/чем делает* (люди, роли, ПО, оборудование).

Классическое расположение (важно запомнить как «картинку в голове»):

- **Вход** — слева;
- **Выход** — справа;
- **Управление** — сверху;
- **Механизм** — снизу.

Лекция №2

Тема: «**Технологии управления проектами. Профессия тимлида и аналитика**»

Подтема: «Формирование отчета и постановка задач с использованием различных технологий моделирования»

Методология функционального моделирования IDEF0 – Контекстная диаграмма

Контекстная диаграмма IDEFo верхнего уровня описывает функцию Ao «Разработать программный продукт»: на вход (I) поступают исходный запрос/идея заказчика, сведения предметной области и исходные материалы, сверху как управление (C) задаются ТЗ и требования, стандарты оформления и критерии приемки, снизу как механизмы (M) выступают команда (тимлид+SA, архитектор, backend, frontend, QA) и инструменты разработки (IDE, Git, трекер задач, CI/CD), а на выходе (O) формируется готовое ПО вместе с документацией, результатами тестирования и материалами для сдачи (презентация/демо).

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

REST и RESTful API

REST (Representational State Transfer, передача состояния представления) – набор принципов, по которым строят взаимодействие между клиентом и сервером по протоколу HTTP (HyperText Transfer Protocol — протокол передачи гипертекста). На практике чаще всего используется HTTPS — защищенная версия HTTP.

RESTful API — это API (Application Programming Interface), которое разработано в соответствии с принципами REST.
То есть это конкретный программный интерфейс, в котором взаимодействие между клиентом и сервером организовано по правилам REST: через ресурсы, стандартные методы HTTP/HTTPS (GET, POST, PUT, PATCH, DELETE) и единообразные адреса запросов.

А есть REST API и RESTful API. По сути, это одно и тоже.
REST API — общее, разговорное название.
RESTful API — более точное название API, которое действительно следует принципам REST.

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

REST и RESTful API. HTTP-методы

Основные HTTP-методы в RESTful API:

1. **GET.** Назначение: получить данные. Примеры: GET /students – получить список студентов, GET /students/5 – получить студента с id=5. Связь с **CRUD:** Read.
2. **POST.** Назначение: создать новый ресурс или создать новую запись. Пример: POST /students. Связь с **CRUD**: Create.
3. **PUT.** Назначение: полностью обновит ресурс или запись. Пример: PUT /students/5 — полностью заменить данные студента с id=5. Связь с **CRUD**: UPDATE. Важно. **PUT** обычно понимают как **полное обновление** объекта. То есть если у объекта было 10 полей, обычно передают весь объект целиком.
4. **PATCH.** Назначение: частично изменить ресурс. Пример: PATCH /students/5 — изменить только часть данных студента, например только фамилию. Связь с **CRUD**: UPDATE. Разница между **PUT** и **PATCH: PUT** – полная изменение объекта или строки, PATCH – частичное изменение объекта или строки.
5. **DELETE.** Назначение: удалить ресурс. Пример: DELETE /students/5 — удалить студента с id=5. Связь с **CRUD**: DELETE.

Дополнительные HTTP-методы в RESTful API:

6. **HEAD.** Назначение: получить только заголовки ответа, без тела.
7. **OPTIONS.** Назначение: узнать, какие методы поддерживает ресурс. Пример: сервер может ответить, что для /students/5 доступны методы GET, PUT, PATCH, DELETE.
8. **TRACE.** Назначение: диагностический метод для проверки прохождения запроса. На практике фактически не используется.
9. **CONNECT.** Назначение: установить туннель к серверу. На практике: нужен в основном для прокси-соединений и к обычным методам RESTful API почти не относится.

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

Ресурс — это объект, с которым работает система и к которому можно обратиться через API.
Примеры ресурсов: студент, преподаватель, книга, заказ, товар.
Обычно ресурс соответствует некоторой **сущности системы** и часто связан с:

1. Классом в программе.
2. Таблицей в базе данных.
3. Объектом в предметной области.

Например, ресурсу **student** могут соответствовать:

1. Класс Student в программе.
2. Таблица students в базе данных.
3. Адрес /students в API.

В RESTful API адрес **URL** обычно обозначает именно **ресурс**, а **HTTP-метод** показывает действие над ним.

Примеры:
1. GET /students — получить список студентов.
2. GET /students/5 — получить студента с id = 5.
3. POST /students — создать нового студента.
4. DELETE /students/5 — удалить студента с id = 5.

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

Важно различать два вида параметров.

1. Path parameter — параметр в пути адреса. Он указывает на конкретный ресурс. Пример: /students/5, где 5 — идентификатор студента.

2. Query parameter — параметр запроса после символа ?. Он используется для фильтрации, поиска, сортировки и постраничного вывода.

Примеры:

1. /students?group=ЭФБО-01-24
2. /students?page=2
3. /students?sort=name

Заключение. В RESTful API адрес должен описывать сущность, а не команду. Поэтому вариант /students/5 считается более корректным, чем /getStudentById.

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

После получения запроса сервер возвращает **HTTP-ответ**.
Он обычно включает:

1. Код состояния.
2. Заголовки.
3. Тело ответа с данными.

Основные коды ответа:

1. 200 OK — запрос выполнен успешно.
2. 201 Created — ресурс успешно создан.
3. 204 No Content — запрос выполнен успешно, но тело ответа отсутствует.
4. 400 Bad Request — запрос составлен неверно.
5. 401 Unauthorized — требуется аутентификация.
6. 403 Forbidden — доступ запрещен.
7. 404 Not Found — ресурс не найден.
8. 500 Internal Server Error — внутренняя ошибка сервера.

В RESTful API данные чаще всего передаются в формате **JSON (JavaScript Object Notation)**.
Пример ответа сервера:

```
{
  "id": 5,
    "name": "Иван Петров",
    "group": "ЭФБО-01-24"
}
```

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

Также важно различать

Request body — тело запроса, которое клиент отправляет серверу. **Response body** — тело ответа, которое сервер возвращает клиенту.

Например, при создании студента через POST /students клиент может отправить в теле запроса JSON-объект с данными нового студента.

Заключение. При работе с RESTful API важно учитывать не только метод и адрес запроса, но и код ответа сервера, а также формат передаваемых данных.

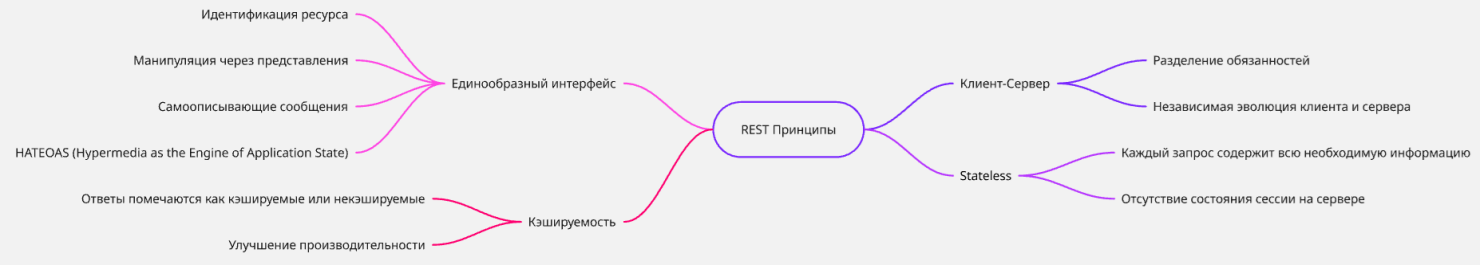
Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

REST и RESTful API. Основные принципы REST

REST — это архитектурный стиль, основанный на нескольких важных принципах.



1. Client–Server

Клиент и сервер разделены по ролям. Клиент отправляет запросы и получает ответы, а сервер обрабатывает запросы и управляет данными.
Примеры клиентов: браузер, мобильное приложение, frontend.
Примеры сервера: backend-приложение, веб-сервер с API.

2. Stateless

Stateless — это принцип REST, согласно которому каждый HTTP-запрос обрабатывается независимо от предыдущих и должен содержать всю информацию, необходимую серверу для выполнения этого запроса.
Stateless = сервер не хранит состояние клиента между запросами.
Это означает, что каждый новый запрос рассматривается независимо от предыдущего.

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

3. Uniform Interface (Единообразный интерфейс)

Интерфейс взаимодействия клиента и сервера должен быть единым, понятным и предсказуемым.
Это означает, что для всех ресурсов применяются одинаковые правила обращения.

Основная идея такая:

1. URL указывает, с каким ресурсом идет работа.
2. HTTP-метод показывает, какое действие нужно выполнить.
3. Данные передаются в едином и понятном формате (чаще всего JSON).
4. Ответы сервера оформляются по единым правилам.

Например:

- GET /students — получить список студентов;
- POST /students — создать нового студента;
- DELETE /students/5 — удалить студента с id=5.

За счет этого API становится проще изучать, использовать, тестировать, поддерживать и расширять.

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

4. Cacheable (Возможность кэширования)

Ответы сервера в некоторых случаях могут сохраняться во временной памяти (кэше), чтобы не выполнять один и тот же запрос заново много раз.

Это дает несколько преимуществ:

1. Снижает нагрузку на сервер.
2. Ускоряет получение данных.
3. Снижает количество повторных сообщений к базе данных.
4. Делает работу приложения быстрее для пользователей.

Например, если список факультетов или справочник дисциплин меняется редко, его можно кэшировать, чтобы каждый раз не запрашивать заново.

При этом важно правильно определять:

1. Какие ответы можно кэшировать.
2. Как долго они считаются актуальными.
3. Когда кэш нужно обновить или очистить.

5. Layered System (Многоуровневая система)

REST допускает, что система может состоять из нескольких уровней, и клиент не обязан знать внутреннюю структуру всей системы.

Например, между клиентом и базой данных могут находиться: веб-сервер, сервер приложений, прокси-сервер, шлюз, балансировщик нагрузки, кэш, микросервисные компоненты.

Типичная схема многоуровневой системы может выглядеть так:

Или сложнее: Клиент -> прокси/шлюз -> API-сервер -> внутренние сервисы -> база данных.

Такой подход позволяет: упростить архитектуру, повысить безопасность, легче масштабировать систему, заменять или дорабатывать отдельные уровни независимо друг от друга.

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

Заключение. REST важен не только как набор HTTP-методов, но и как архитектурный подход, который позволяет логично, единообразно и удобно организовать взаимодействие клиента и сервера.
Благодаря этому API становится более понятным, предсказуемым, расширяемым и удобным для практического использования.

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

Практика работы с RESTful API

При разработке, тестировании и сопровождении RESTful API используют специальные инструменты, которые помогают отправлять запросы, анализировать ответы сервера, проверять корректность работы методов и документировать API.

1. **Postman**

Postman — один из самых популярных инструментов для ручного тестирования API.

Он позволяет:

1. Отправлять HTTP-запросы разных типов (GET, POST, PUT, PATCH, DELETE).
2. Проверять коды ответа сервера.
3. Просматривать тело ответа, заголовки и служебную информацию.
4. Передавать параметры запроса, JSON-данные и файлы.
5. Работать с токенами, cookies и заголовками.
6. Сохранять наборы запросов в коллекции для повторного использования.

Postman особенно удобен на этапе разработки backend-части, когда нужно быстро проверить, как работает API, еще до подключения полноценного frontend-приложения.

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

2. Swagger / OpenAPI

Swagger — это набор инструментов с открытым исходным кодом для проектирования, документирования, тестирования и генерации кода RESTful API на базе спецификации OpenAPI.

Swagger / OpenAPI используются для описания и документирования RESTful API.

С их помощью можно: перечислить доступные методы API, показать адреса запросов, описать параметры, заголовки и тело запроса, указать возможные коды ответа, зафиксировать формат данных, например JSON, автоматически сформировать удобную интерактивную документацию.

Преимущество Swagger состоит в том, что разработчик может не только читать описание API, но и сразу отправлять тестовые запросы через браузер. Это делает Swagger полезным одновременно и для программистов, и для тестировщиков, и для тех, кто будет интегрировать API в другое приложение.

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

3. Curl

curl — это консольный инструмент для отправки HTTP-запросов из командной строки.

Он удобен, когда нужно: быстро проверить доступность API, протестировать запрос без графического интерфейса, использовать API на сервере или в скриптах, автоматизировать обращения к API.

Пример простого запроса:

```
curl -X GET http://localhost:8080/students
```

С помощью curl можно передавать: заголовки, JSON в теле запроса, параметры аутентификации, данные формы и файлы.

Пример запроса с ответом:

```
curl -X POST http://localhost:8080/students \
     -H "Content-Type: application/json" \
     -d "{\"name\":\"Иван Петров\",\"group\":\"ЭФБО-01-24\"}"
```

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

4. Аутентификация и авторизация

В реальных RESTful API часто требуется контроль доступа, потому что не все пользователи должны иметь одинаковые права.

Аутентификация — это проверка, кто именно выполняет запрос. Например, система должна убедиться, что перед ней действительно зарегистрированный пользователь.

Авторизация — это проверка, что именно разрешено этому пользователю. Например, один пользователь может только читать данные, а другой — создавать, изменять и удалять записи.

Для реализации контроля доступа часто используют: логин и пароль, токены доступа, заголовок Authorization, JWT (JSON Web Token), API-ключи.

Пример:

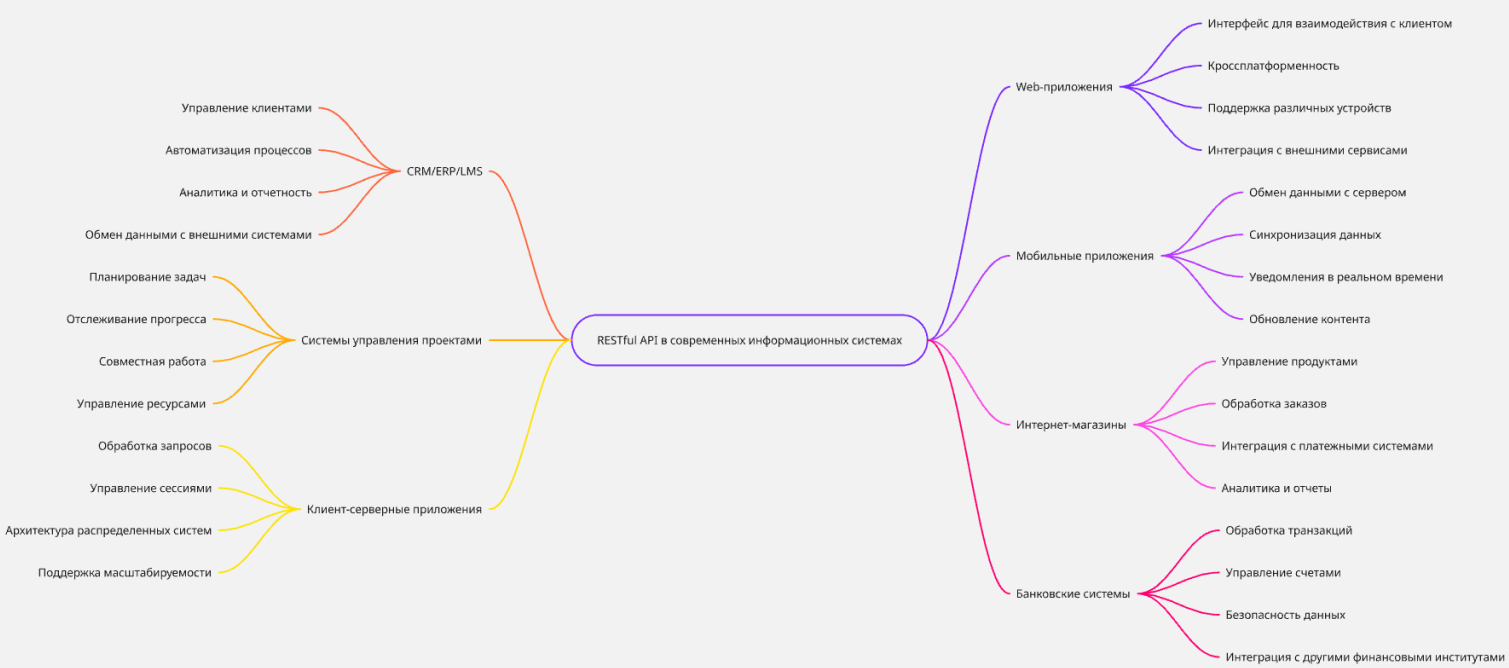
1. Студент может просматривать только свои данные.
2. Преподаватель может просматривать и изменять оценки.
3. Администратор может управлять всеми ресурсами системы.

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

Практика работы с RESTful API. Где применяется?



Практическое значение для разработчика. Знание инструментов работы с RESTful API позволяет:

1. Проверять API без пользовательского интерфейса.
2. Быстрее находить ошибки в запросах и ответах.
3. Тестировать backend до подключения frontend.
4. Документировать API для других разработчиков.
5. Организовывать безопасный доступ к данным.

Таким образом, RESTful API — это не только теоретическая модель взаимодействия клиента и сервера, но и практический инструмент, с которым ежедневно работают разработчики, тестировщики и интеграторы. Для этого используют Postman, Swagger/OpenAPI, curl и механизмы аутентификации и авторизации, которые позволяют тестировать, документировать и безопасно использовать API в реальных проектах.

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

Практика работы с RESTful API. Разработка

1. Исправление предыдущей ошибки, когда при редактировании студента создавался новый. Причина заключалась в том, что я не передал ID в шаблонизатор в файле edit_students.html.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Редактирование данных</title>
</head>
<body>
<h1>
  Редактирование данных студента
</h1>
<form th:action="@{/save}" th:object="${student}" method="post">
  <!--Добавил ID для того, чтобы редактировать текущий экземпляр, а не создавать новый-->
  <input type="hidden" th:field="*{id}">
  <label class="form-label">Имя студента</label>
  <input type="text" th:field="*{firstName}" required>
  <label class="form-label">Фамилия студента</label>
  <input type="text" th:field="*{lastName}" required>
  <label class="form-label">Возраст студента</label>
  <input type="number" th:field="*{age}" required>
  <button type="submit">Сохранить</button>
  <a href = "/">На главную</a>

</form>
</body>
</html>
```

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

Практика работы с RESTful API. Разработка

2. Переименуем класс «RestController» в «StudentApiController». Причина: RestController — это еще и название аннотации Spring.
3. Меняем аннотация с @Controller на @RestController

```
package com.example.demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.repository.query.Param;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.ModelAndView;

import java.util.List;

/**
 * @RestController говорит Spring: возвращать данные как JSON
 *
 * @RequestMapping("/api/students") задает общий путь для всех методов
 */
@RestController
@RequestMapping("/api/students")
public class StudentApiController {
```

4. Также необходимо заменить @RequestMapping на специальные аннотации: @GetMapping, @PostMapping, @PutMapping, @DeleteMapping. Это более правильно, чище и понятно.
5. Меняем метод получения всех студентов.

```
/**
 * Получение списка студентов по API (главная страница)
 * @param keyword - Ключевое слово для поиска студентов
 * @return - Возвращаем список студентов в формате JSON
 */
@GetMapping
public List<Student> getAllStudents(@RequestParam(required = false) String keyword) {
    return service.ListAll(keyword);
}
```

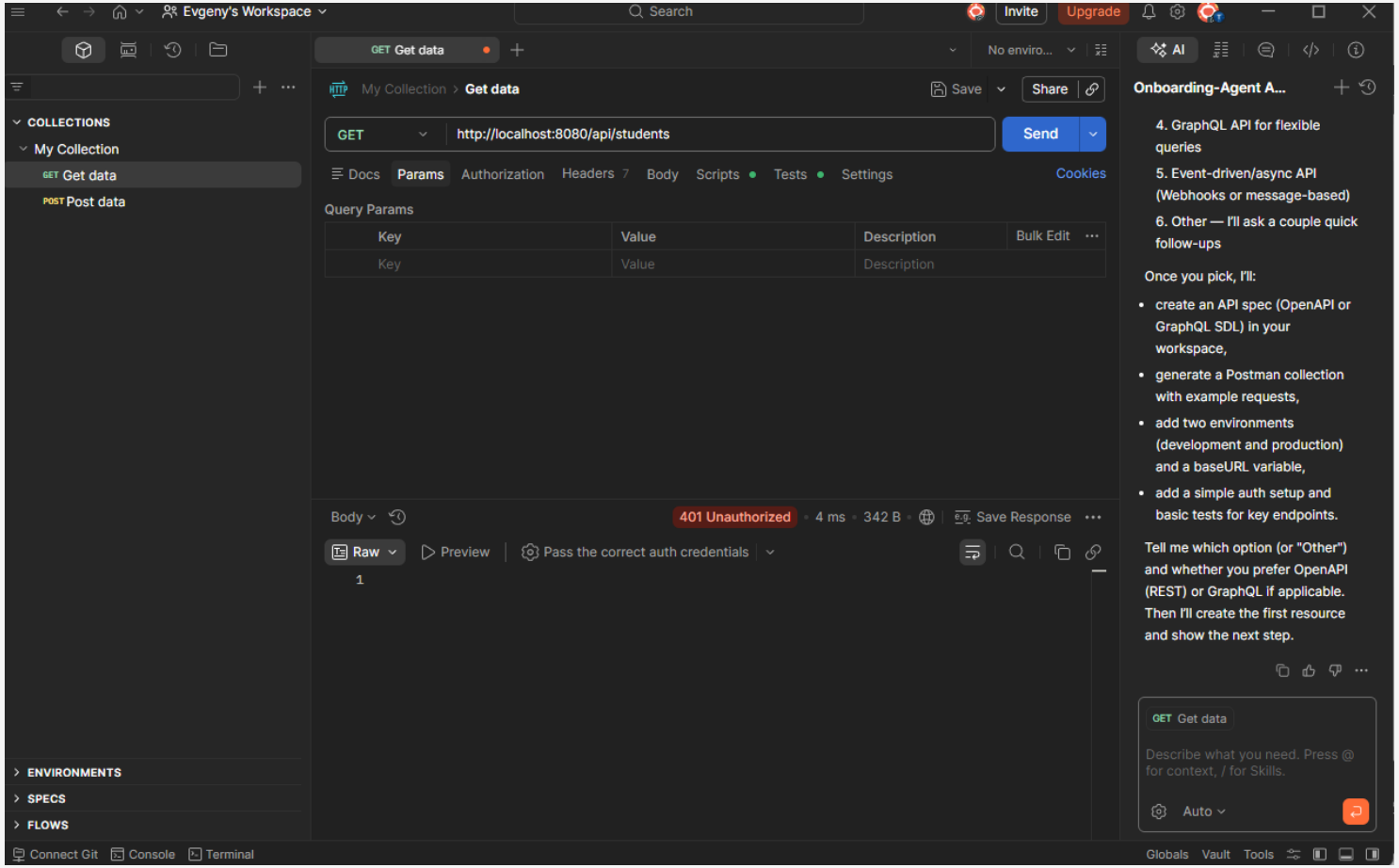
Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

Практика работы с RESTful API. Разработка

По результатам выполнения GET-запроса выйдет 401-я ошибка, означающая ограничение доступа из-за авторизации.



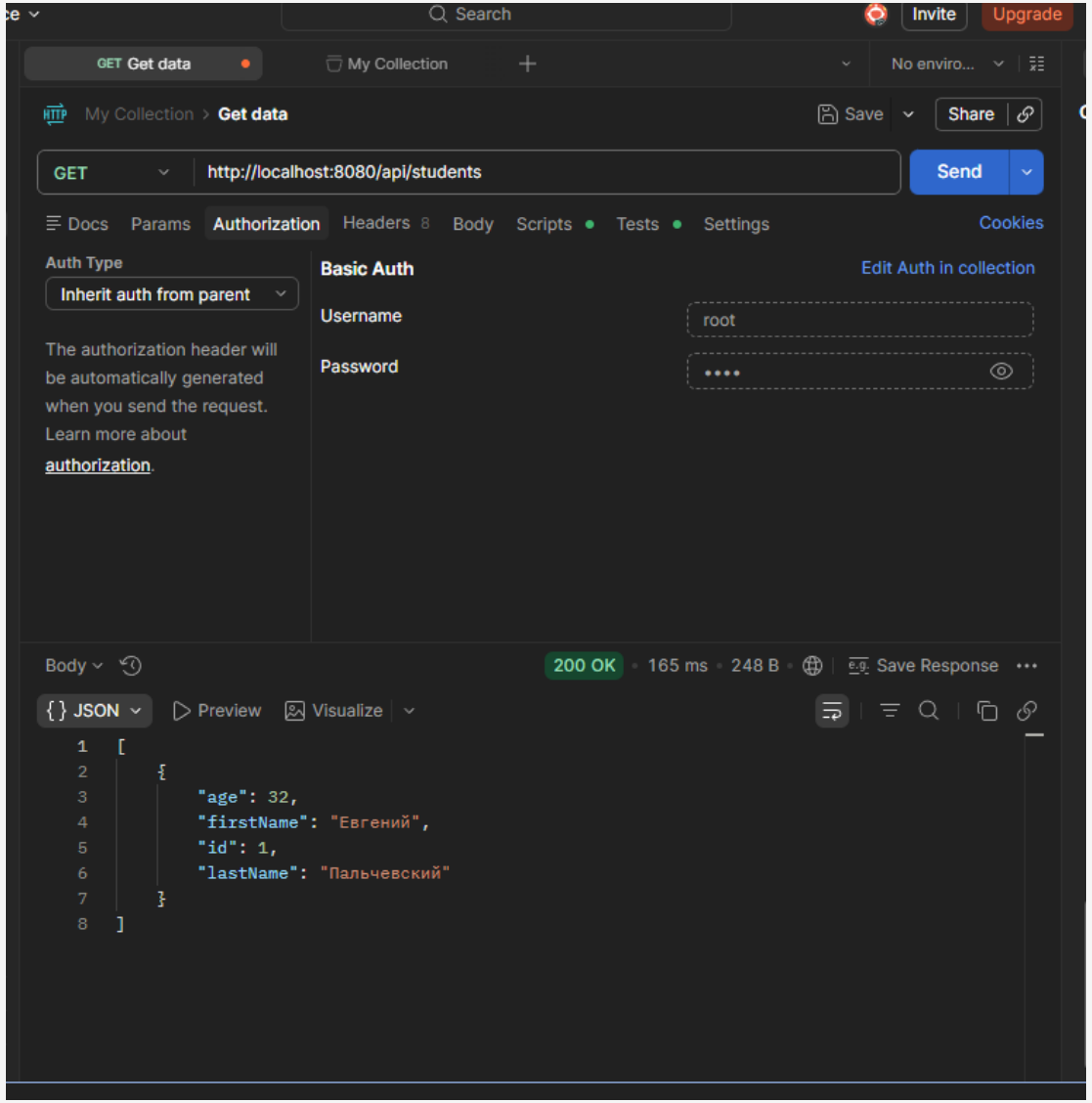
Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

Практика работы с RESTful API. Разработка

Необходимо авторизоваться и тогда будет выведен весь список студентов. В веб-интерфейсе он пока что отображаться не будет из-за изменения метода.



Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

Практика работы с RESTful API. Разработка

6. Меняем метод создания студента

```
/**
 * Метод добавления студентов
 *
 * @return - Возврат рендера шаблона
 */
@PostMapping
public ResponseEntity<Student> createStudent(@RequestBody Student student) {
   student.setId(null);
   Student savedStudent = service.save(student);
   return ResponseEntity.status(HttpStatus.CREATED).body(savedStudent);
}
```

7. Одновременно с этим меняем у ID тип данных int на Integer в классе Student. Причина – Integer работает со ссылочными типами данных

```
@Entity
@Getter @Setter
public class Student {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY) // Первичный ключ
   private Integer id;
   private String firstName;
   private String lastName;
   private int age;
}
```

8. Одновременно с этим меняем метод save в классе StudentService, чтобы возвращал данные.

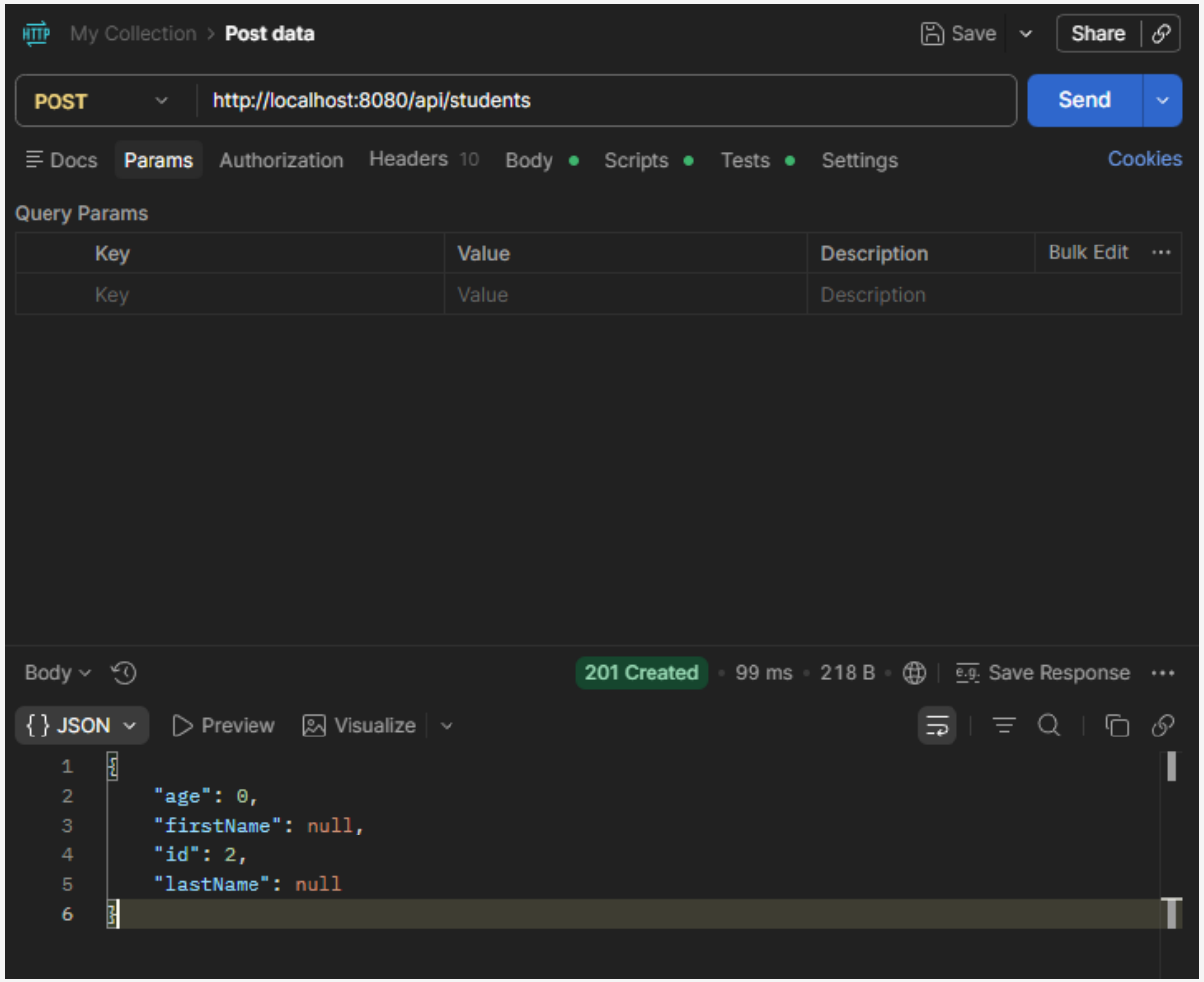
```
/**
 * Метод сохранения студентов
 * @param student - Наш класс Student
 */
public Student save(Student student) {
   return repo.save(student);
}
```

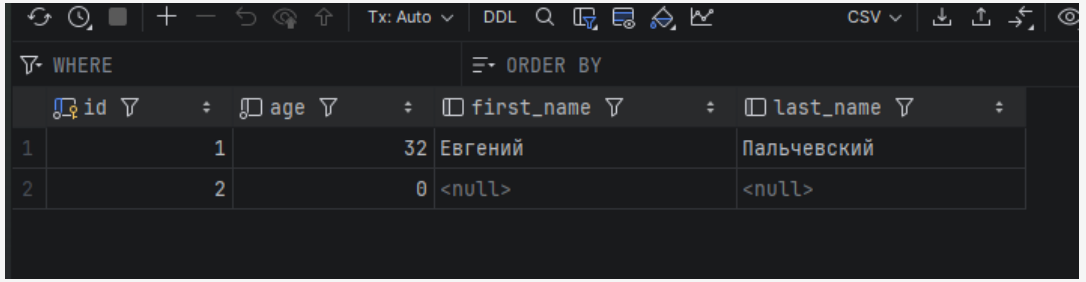
Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

9. Тестируем с пустыми именем и фамилией. Все работает





Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

Практика работы с RESTful API. Разработка

10. Переделываем получение информации об одном студенте по ID и в целом метод редактирования

```
/**
 * Получение студента по ID
 * @param id - ID студента
 */
@GetMapping("/{id}")
public ResponseEntity<Student> getStudentById(@PathVariable Integer id) {
    try {
        Student student = service.get(id);
        return ResponseEntity.ok(student);
    } catch (Exception e) {
        return ResponseEntity.notFound().build();
    }
}
```

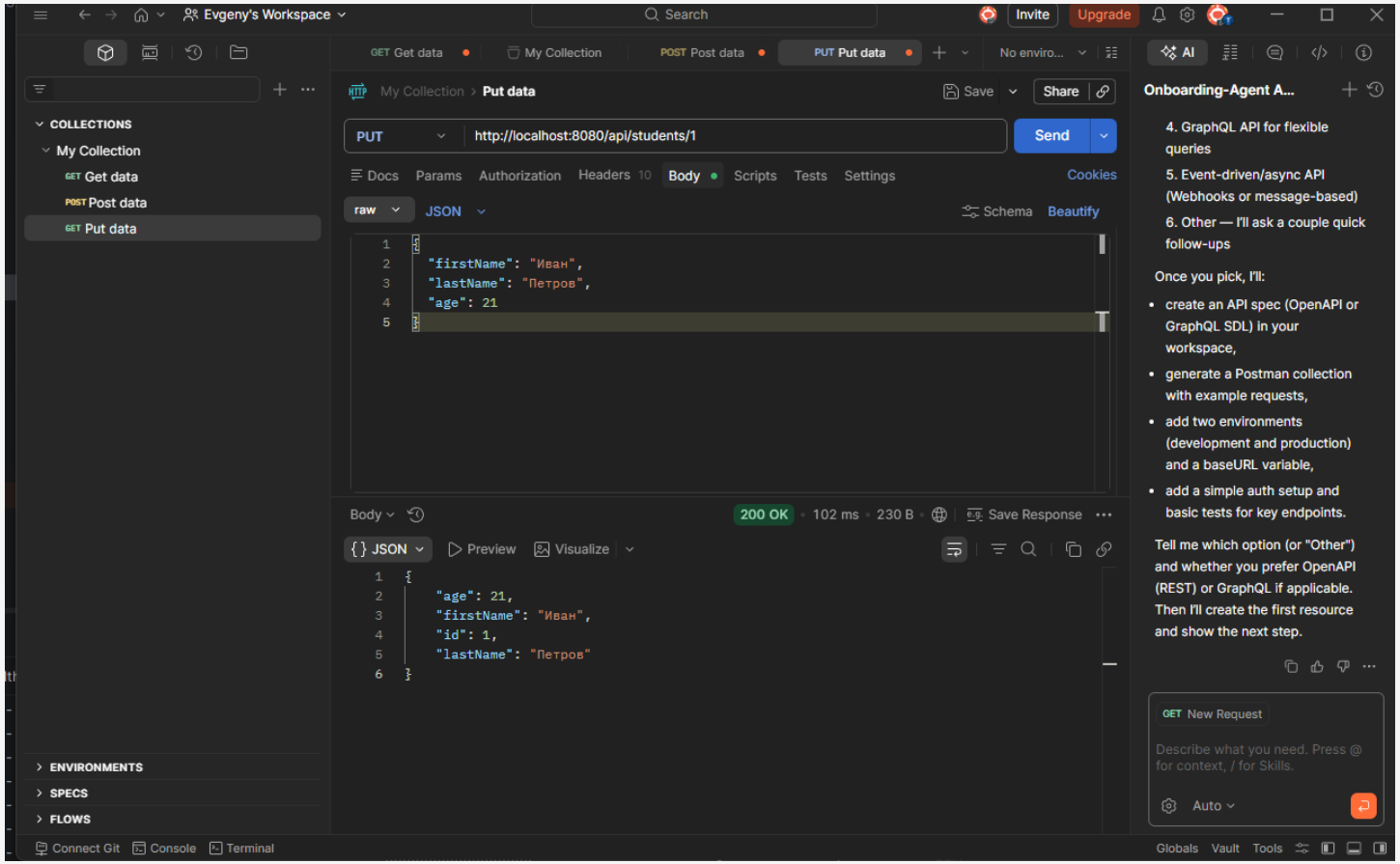
```
/**
 * Редактирование студента по его идентификатору
 * @param id - ID студента, которого нужно изменить
 * @param studentDetails - новые данные студента, пришедшие в теле запроса
 * @return - возвращаем обновленного студента или статус 404, если студент не найден
 */
@PutMapping("/{id}") // Обрабатывает HTTP PUT-запросы вида /api/students/{id}
public ResponseEntity<Student> updateStudent(@PathVariable Integer id, // Берем id из адресной строки
                                             @RequestBody Student studentDetails) { // Получаем новые данные студента из тела запроса в формате
JSON
    try { // Пробуем выполнить обновление; если что-то пойдет не так, управление перейдет в catch

        Student existingStudent = service.get(id); // Получаем из базы существующего студента по его id

        existingStudent.setFirstName(studentDetails.getFirstName()); // Обновляем имя студента новым значением
        existingStudent.setLastName(studentDetails.getLastName()); // Обновляем фамилию студента новым значением
        existingStudent.setAge(studentDetails.getAge()); // Обновляем возраст студента новым значением

        Student updatedStudent = service.save(existingStudent); // Сохраняем уже измененный объект обратно в базу данных

        return ResponseEntity.ok(updatedStudent); // Возвращаем HTTP 200 OK и обновленного студента в ответе

    } catch (Exception e) { // Если студент не найден или произошла другая ошибка при получении/сохранении
        return ResponseEntity.notFound().build(); // Возвращаем HTTP 404 Not Found без тела ответа
    }
}
```

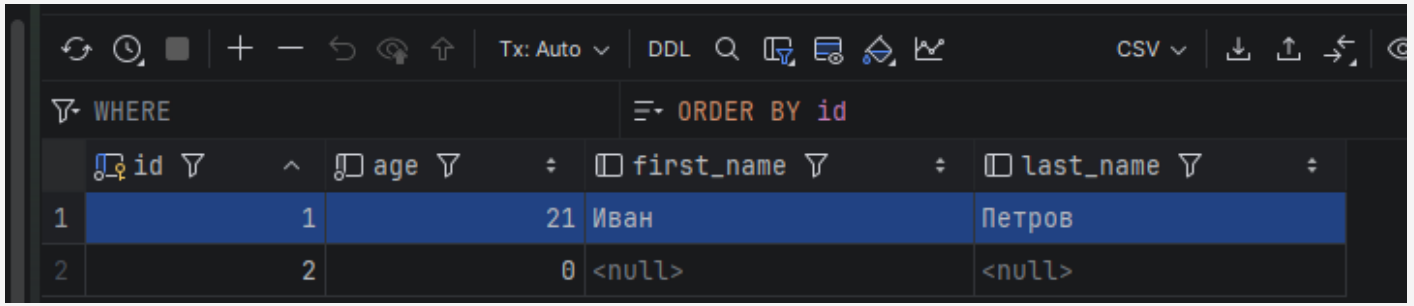
Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

11. Тестируем метод получения по ID и редактирования





Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

Практика работы с RESTful API. Разработка

12. Меняем метод удаления

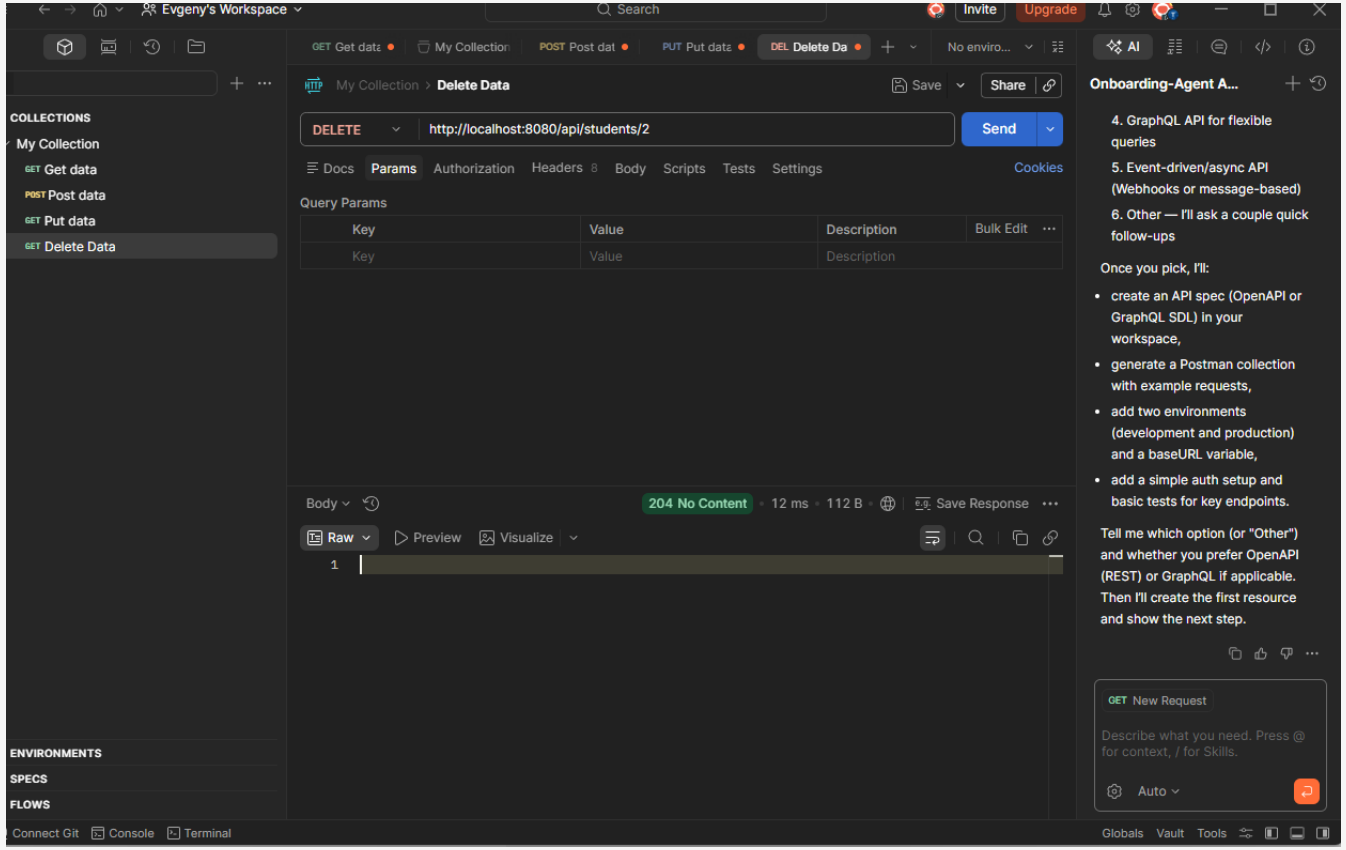
```
/**
 * Метод удаления студента по его идентификатору.
 * Используется для обработки HTTP DELETE-запроса.
 *
 * @param id идентификатор студента, которого нужно удалить
 * @return HTTP-ответ без тела:
 *       204 No Content — если удаление прошло успешно,
 *       404 Not Found — если возникла ошибка
 */
@DeleteMapping("/{id}") // Обрабатывает DELETE-запрос по адресу /api/students/{id}
public ResponseEntity<Void> deleteStudent(@PathVariable("id") Integer id) { // Получаем id из URL и возвращаем HTTP-ответ без тела
    try { // Начало блока, в котором пробуем выполнить удаление
        service.delete(id); // Вызываем сервисный метод удаления студента по его id
        return ResponseEntity.noContent().build(); // Возвращаем статус 204 No Content: удаление выполнено успешно
    } catch (Exception e) { // Если при удалении возникла любая ошибка, переходим сюда
        return ResponseEntity.notFound().build(); // Возвращаем статус 404 Not Found
    } // Конец блока catch
} // Конец метода deleteStudent
```

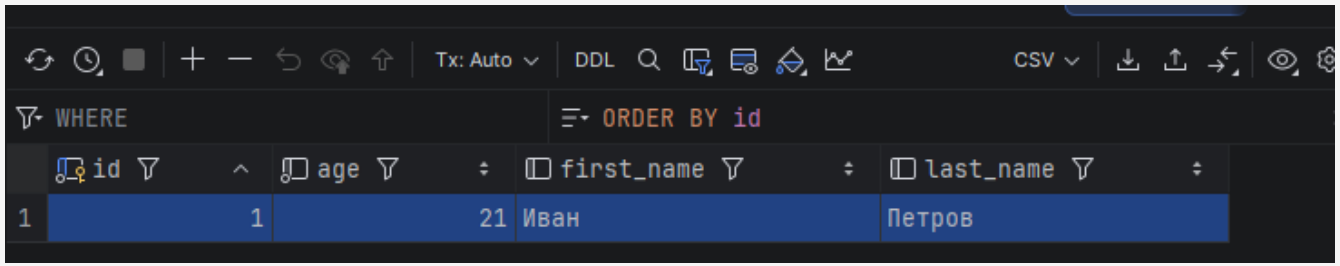
Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

13. Тестируем метод удаления студентов





Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

14. Итоговый класс StudentApiController – Часть 1

```
package com.example.demo;

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("/api/students")
public class StudentApiController {

    @Autowired
    private StudentService service;

    /**
     * Получение списка студентов по API
     * @param keyword - Ключевое слово для поиска студентов
     * @return - Возвращаем список студентов в формате JSON
     */
    @GetMapping
    public List<Student> getAllStudents(@RequestParam(value = "keyword", required = false) String
keyword) {
        return service.ListAll(keyword);
    }

    /**
     * Метод добавления студентов
     * @param student - Данные нового студента
     * @return - Возвращаем сохраненного студента
     */
    @PostMapping
    public ResponseEntity<Student> createStudent(@RequestBody Student student) {
        student.setId(null);
        Student savedStudent = service.save(student);
        return ResponseEntity.status(HttpStatus.CREATED).body(savedStudent);
    }
```

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

Практика работы с RESTful API. Разработка

14. Итоговый класс StudentApiController – Часть 2

```
/**
 * Получение студента по ID
 * @param id - ID студента
 * @return - Возвращаем найденного студента или 404
 */
@GetMapping("/{id}")
public ResponseEntity<Student> getStudentById(@PathVariable("id") Integer id) {
    try {
        Student student = service.get(id);
        return ResponseEntity.ok(student);
    } catch (Exception e) {
        return ResponseEntity.notFound().build();
    }
}

/**
 * Редактирование студента по его идентификатору
 * @param id - ID студента, которого нужно изменить
 * @param studentDetails - новые данные студента
 * @return - возвращаем обновленного студента или 404
 */
@PutMapping("/{id}")
public ResponseEntity<Student> updateStudent(@PathVariable("id") Integer id,
                                             @RequestBody Student studentDetails) {
    try {
        Student existingStudent = service.get(id);

        existingStudent.setFirstName(studentDetails.getFirstName());
        existingStudent.setLastName(studentDetails.getLastName());
        existingStudent.setAge(studentDetails.getAge());

        Student updatedStudent = service.save(existingStudent);
        return ResponseEntity.ok(updatedStudent);

    } catch (Exception e) {
        return ResponseEntity.notFound().build();
    }
}
```

Лекция №3

Тема: «**Технологии управления проектами**»

Подтема: «RESTful API с последующим применением при реализации проекта»

Практика работы с RESTful API. Разработка

14. Итоговый класс StudentApiController – Часть 3

```
    /**
     * Метод удаления студента по его идентификатору.
     * Используется для обработки HTTP DELETE-запроса.
     *
     * @param id идентификатор студента, которого нужно удалить
     * @return HTTP-ответ без тела:
     *       204 No Content — если удаление прошло успешно,
     *       404 Not Found — если возникла ошибка
     */
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteStudent(@PathVariable("id") Integer id) {
        try {
            service.delete(id);
            return ResponseEntity.noContent().build();
        } catch (Exception e) {
            return ResponseEntity.notFound().build();
        }
    }
}
```

Контрольные работы

№ Контрольной работы	Что должно быть готово (результат)	Что сдаете	Минимум для баллов (как проверяется)
Контрольная работа №1: Анализ + требования (без проектирования и без кода)	Понятно, что именно автоматизируем и какие функции обязаны быть, включая AI-функцию как часть процесса	1) Описание предметной области «как есть» + проблемы/узкие места 2) Цель/задачи/назначение системы (кратко) 3) Роли + матрица прав (кто что может) 4) Функциональные требования (что умеет система) + минимум 3 сквозных бизнес-процесса (пошагово) 5) AI-функция (постановка): тип задачи, входные данные, выход, где используется в процессе, критерий качества (хотя бы простой)	Проверяю, что: а) процессы понятны и реализуемы; б) роли не противоречат процессам; в) AI встроена в процесс (а не «где-то будет модель»)
Контрольная работа №2: Готовый backend	Рабочий backend без фронта: БД + API + роли/права + журнал + AI-endpoint	1) Репозиторий/архив проекта 2) Скрипты/миграции БД + тестовые данные 3) Swagger/OpenAPI или перечень эндпоинтов 4) Auth (рег/логин), CRUD, поиск/фильтрация, бизнес-операции процессов, журнал действий 5) AI-endpoint: запрос → обработка → ответ	Проект запускается, 3 процесса выполняются через API, роли реально ограничивают доступ, AI-endpoint возвращает понятный результат
Контрольная работа №3: Полностью реализованная система (backend + frontend) + проектирование	Рабочая система «под ключ»: UI ⇿ API ⇿ БД, можно пройти сценарии мышкой, AI показывается пользователю. И здесь же сдается пакет проектирования, соответствующий текущей реализации	1) Все из этапа №2 + фронт (репозиторий/архив) 2) UI-экраны: вход/регистрация, основной раздел, формы/карточки, поиск/фильтры, экран/виджет AI, журнал/админ (если предусмотрено) 3) Демонстрация: минимум 2 процесса через UI + AI-результат в UI 4) акет проектирования (теперь тут): ER-схема (финальная), Use Case, 1–2 Activity, (при необходимости) Component/Deployment, + макеты ключевых экранов (как «как должно быть», но уже согласовано с тем, что реально сделано)	Проверяю «сквозняк»: UI не заглушки, роли работают на UI и API, процессы проходят, AI вызывается из UI. Диаграммы/ER/макеты не расходятся с реализацией
Контрольная работа №4: Тестирование + документация	Доказана работоспособность и понятна эксплуатация (чтобы другой человек мог запустить и пользоваться)	1) Протокол тестирования (факт): backend ≥10, frontend ≥10, сквозные ≥5 (проверка/ожидаемо/факт/дефекты) 2) Документация: руководство пользователя, руководство администратора (если есть), описание AI (данные/метрика/интерпретация), инструкция запуска	По инструкции реально поднять систему; тесты/проверки привязаны к требованиям и процессам; документация написана понятно
Контрольная работа №5: Сдача проекта вместе с отчетом КП **После КР №5 идет защита проекта перед комиссией**	Финальный комплект: рабочий проект + полный отчет КП + приложения	1) Финальная версия проекта 2) Финальная версия отчета КП (по вашей структуре) 3) Приложения: диаграммы/скрины, протокол тестирования, инструкция запуска, описание API, описание AI	Отчет соответствует тому, что в коде; все разделы и приложения на месте; нет противоречий между требованиями/диаграммами/реализацией