Лабораторная работа 4. Взаимодействие Spring и PostgreSQL

Взаимодействие между Spring Boot и PostgreSQL строится на нескольких ключевых компонентах, которые обеспечивают плавную интеграцию между приложением и базой данных.

1. Конфигурация подключения

Spring Boot использует автоматическую настройку (auto-configuration), что означает, что вам достаточно указать базовые параметры подключения к базе данных в файле конфигурации application.properties или application.yml. Эти параметры включают URL базы данных, имя пользователя, пароль и драйвер базы данных.

Пример конфигурации:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/mydatabase
spring.datasource.username=postgres
spring.datasource.password=secret
spring.datasource.driver-class-name=org.postgresql.Driver
```

Spring Boot автоматически настраивает DataSource — компонент, который управляет пулом соединений с базой данных.

2. JPA (Java Persistence API) и ORM

Обычно в Spring Boot для взаимодействия с реляционными базами данных используется JPA (Java Persistence API) с библиотекой Hibernate, которая является реализацией JPA. ORM (Object-Relational Mapping) позволяет преобразовывать объекты Java в записи таблиц базы данных и наоборот, абстрагируя работу с SQL.

Для этого разработчик создает сущности (Java-классы), которые аннотируются специальными JPA-аннотациями, такими как @Entity, @Table, @Id и т.д. Эти сущности позволяют организовать взаимодействие с таблицами базы данных.

3. Spring Data JPA

Spring Data JPA — мощный модуль, который упрощает взаимодействие с базой данных через JPA. Основное преимущество Spring Data JPA — это создание репозиториев (интерфейсов), которые позволяют совершать CRUD-операции (Create, Read, Update, Delete) без необходимости писать SQL-запросы.

Пример:

```
java
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByLastName(String lastName);
}
```

Этот интерфейс автоматически предоставляет методы для работы с базой данных, такие как save(), findById(), findAll(), и может быть расширен для более сложных запросов.

4. Автоматическое создание таблиц

Если в настройках приложения включена опция spring.jpa.hibernate.ddl-auto=update, Hibernate автоматически создает или обновляет структуру таблиц в базе данных на основе определений сущностей в коде. Это удобно на этапе разработки, так как избавляет от необходимости вручную писать SQL для создания схемы.

5. Транзакции

Spring Boot использует Spring Transaction Management для управления транзакциями. Аннотация методов с помощью @Transactional Spring гарантирует, что все операции внутри метода будут выполнены в рамках одной транзакции. Если одна из операций завершается неудачно, все изменения откатываются.

6. SQL-запросы и кастомизация

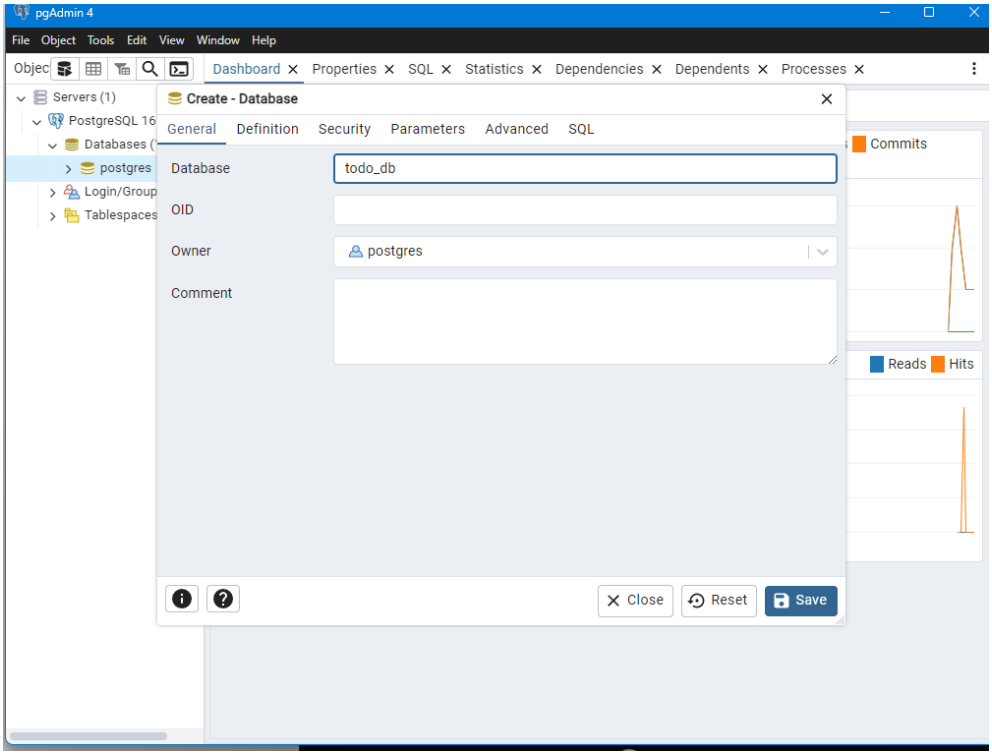
Хотя Spring Data JPA поддерживает автоматическое создание SQL-запросов, вы также можете писать собственные запросы с помощью аннотации @Query или использовать нативные SQL-запросы. Это полезно, когда стандартные механизмы JPA не покрывают нужды приложени.

Цель работы: получить практические навыки работы с Spring Framework.

Пример выполнения работы.

В качестве примера возьмем приложение, разработанное в лабораторной работе 5.

Для начала создадим базу данных (СУБД PostgreSQL), в которой позже будет создана таблица, содержащая записи из to-do списка.



Создание базы данных

Таблицу для базы данных можно пока что не создавать, она сформируется автоматически при запуске приложения.

Настройка конфигурации базы данных:

В файле `application.properties` настройте соединение с базой данных:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/todo_db
spring.datasource.username=postgres
spring.datasource.password=ВАШ_ПАРОЛЬ
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

1. **`spring.datasource.url=jdbc:postgresql://localhost:5432/todo_db`**
   Определяет URL для подключения к базе данных PostgreSQL.
   - `jdbc:postgresql://localhost:5432/todo_db` указывает, что приложение будет подключаться к базе данных с именем `todo_db`, работающей на локальном компьютере (`localhost`) на порту 5432.
2. **`spring.datasource.username=postgres`**
   Указывает имя пользователя для подключения к базе данных. В данном случае используется пользователь `postgres`.

3. **`spring.datasource.password=ВАШ_ПАРОЛЬ`**
   Здесь указывается пароль для пользователя базы данных. Вместо `ВАШ_ПАРОЛЬ` нужно указать реальный пароль.
4. **`spring.datasource.driver-class-name=org.postgresql.Driver`**
   Указывает драйвер для подключения к базе данных PostgreSQL. В данном случае это `org.postgresql.Driver`.
5. **`spring.jpa.hibernate.ddl-auto=update`**
   Эта настройка управляет тем, как Hibernate (реализация JPA по умолчанию в Spring Boot) будет обрабатывать структуру базы данных. Опция `update` означает, что Hibernate автоматически обновит схему базы данных при каждом запуске приложения, создавая недостающие таблицы и изменяя существующие.
6. **`spring.jpa.show-sql=true`**
   Эта опция включает вывод SQL-запросов в логах. Это удобно для отладки, поскольку вы сможете видеть все SQL-запросы, которые генерирует Hibernate.

Вместе эти настройки позволяют вашему приложению подключаться к базе данных PostgreSQL, автоматически управлять схемой базы данных и отображать SQL-запросы для отладки.

Не забудьте изменить пароль на тот, который вы установили в строке:
spring.datasource.password=ВАШ_ПАРОЛЬ

(Если вы меняли имя и порт, то исправьте и их)

Добавление зависимостей в `pom.xml`:

Добавьте необходимые зависимости для работы с базой данных PostgreSQL и JPA/Hibernate:

```
    <dependency>
       <groupId>org.postgresql</groupId>
       <artifactId>postgresql</artifactId>
       <scope>runtime</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
       <plugin>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-maven-plugin</artifactId>
       </plugin>
```

Зависимость (<dependency>) добавляет драйвер PostgreSQL в проект. Она нужна для того, чтобы ваше приложение могло подключаться к базе данных PostgreSQL. Разберем по частям:

- **`<groupId>org.postgresql</groupId>`** — указывает на организацию, которая предоставляет этот пакет, в данном случае это PostgreSQL.
- **`<artifactId>postgresql</artifactId>`** — указывает конкретный артефакт, то есть библиотеку драйвера для PostgreSQL.
- **`<scope>runtime</scope>`** — указывает, что драйвер будет использоваться только во время выполнения (runtime), а не во время компиляции. Это оптимизирует

процесс сборки, исключая драйвер из компиляции, поскольку он нужен только во время запуска приложения.

Плагин <plugin> необходим для работы с Spring Boot и выполнения задач, связанных с Maven. Вот его роль:

- **<groupId>org.springframework.boot</groupId>** — указывает на группу плагинов Spring Boot.
- **<artifactId>spring-boot-maven-plugin</artifactId>** — добавляет Maven-плагин для Spring Boot. Этот плагин позволяет упрощать такие операции, как запуск приложения, сборка JAR-файлов с зависимостями и выполнение задач Spring Boot через командную строку (`mvn spring-boot:run`).

Создание сущности Task:

Замените класс Task (который использовался для хранения записей в памяти приложения) на сущность, которая будет храниться в базе данных:

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Task {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String description;
    private boolean completed;

    // Getter and Setter for id
    public Long getId() {
        return id;
    }

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

    // Getter and Setter for description
    public String getDescription() {
        return description;
    }

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

    // Getter and Setter for completed
    public boolean isCompleted() {
        return completed;
    }

    public void setCompleted(boolean completed) {
        this.completed = completed;
    }
}
```

Аннотация `@Entity`

- Аннотация `@Entity` указывает, что этот класс представляет собой сущность и соответствует таблице в базе данных. При запуске приложения, фреймворк (например, Hibernate) создаст или сопоставит этот класс с таблицей в базе данных.

2. Поле `id` и аннотации `@Id` и `@GeneratedValue`

- Поле `id` представляет собой уникальный идентификатор каждой задачи.
- Аннотация `@Id` указывает, что это поле является первичным ключом таблицы.
- Аннотация `@GeneratedValue(strategy = GenerationType.IDENTITY)` указывает, что значение поля `id` будет генерироваться автоматически базой данных при вставке новой записи. Стратегия `GenerationType.IDENTITY` обычно означает, что база данных будет увеличивать значение автоматически (например, автоинкремент в SQL).

3. Поле `description`

- Это поле хранит текстовое описание задачи. Оно представляет собой строку и не содержит дополнительных аннотаций, что значит, что оно просто будет сопоставлено с соответствующим столбцом в таблице.

4. Поле `completed`

- Это булево поле, которое указывает, завершена ли задача или нет (true/false). Тоже соответствует полю таблицы базы данных.

5. Геттеры и сеттеры

- Геттеры и сеттеры позволяют получить и установить значения полей `id`, `description`, и `completed`. Эти методы необходимы для того, чтобы взаимодействовать с полями класса при использовании ORM (Object-Relational Mapping), например, при сохранении и извлечении данных из базы данных.

Создание репозитория `TaskRepository` для работы с задачами:

Создайте интерфейс для взаимодействия с базой данных:

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

public interface TaskRepository extends JpaRepository<Task, Long> {
}
```

- **`TaskRepository`** — это репозиторий для сущности `Tasks`. Репозитории используются для работы с данными (CRUD-операции: создание, чтение, обновление и удаление).
- **`extends JpaRepository<Tasks, Long>`** — здесь `JpaRepository` является интерфейсом, предоставляемым Spring Data JPA, который обеспечивает набор стандартных методов для взаимодействия с базой данных.
    - **`Task`** — это тип сущности, с которой будет работать репозиторий.

- **`Long`** — это тип идентификатора (ID) сущности `Task`. В данном случае предполагается, что у сущности `Task` есть поле `id` типа `Long`.

Модификация контроллера TaskController:

Обновите контроллер для работы с базой данных через репозиторий:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

@Controller
public class TaskController {

    @Autowired
    private TaskRepository taskRepository;

    @GetMapping("/")
    public String index(Model model) {
        model.addAttribute("tasks", taskRepository.findAll());
        return "index";
    }

    @PostMapping("/addTask")
    public String addTask(@ModelAttribute Task task) {
        taskRepository.save(task);
        return "redirect:/";
    }

    @PreAuthorize("hasRole('MODERATOR')")
    @GetMapping("/deleteTask/{id}")
    public String deleteTask(@PathVariable Long id) {
        taskRepository.deleteById(id);
        return "redirect:/";
    }

    @PreAuthorize("hasRole('MODERATOR')")
    @PostMapping("/updateTask/{id}")
    public String updateTask(@PathVariable Long id, @RequestParam boolean
completed) {
        Tasks task = taskRepository.findById(id).orElseThrow();
        task.setCompleted(completed);
        taskRepository.save(task);
        return "redirect:/";
    }
}
```

Основные компоненты:

1. **Аннотация `@Controller`**:

- Этот класс помечен как контроллер Spring MVC с помощью аннотации `@Controller`. Это указывает, что он будет обрабатывать HTTP-запросы и возвращать представления (HTML-страницы или другие типы данных).

2. **Внедрение репозитория `TaskRepository`**:

- Поле `taskRepository` внедряется с помощью аннотации `@Autowired`. Это значит, что Spring автоматически создаст экземпляр этого репозитория и передаст его в контроллер для взаимодействия с базой данных.

3. **Методы контроллера**:

Метод для отображения задач (GET-запрос на `/`):

```
@GetMapping("/")
public String index(Model model) {
    model.addAttribute("tasks", taskRepository.findAll());
    return "index";
}
```

- Этот метод обрабатывает GET-запрос на корневом URL (`/`).
- С помощью метода `taskRepository.findAll()` он получает все задачи из базы данных и добавляет их в модель как атрибут `"tasks"`.
- Возвращает строку `"index"`, которая соответствует имени HTML-шаблона, который будет отображать список задач.

Метод для добавления новой задачи (POST-запрос на `/addTask`):

```
@PostMapping("/addTask")
public String addTask(@ModelAttribute Tasks task) {
    taskRepository.save(task);
    return "redirect:/";
}
```

- Этот метод обрабатывает POST-запрос на URL `/addTask`.
- Используя аннотацию `@ModelAttribute`, задача (`Tasks task`) автоматически связывается с данными, переданными в форме.
- Затем задача сохраняется в базе данных с помощью метода `taskRepository.save(task)`.
- После этого происходит перенаправление на корневой URL, чтобы снова отобразить список задач.

Метод для удаления задачи (GET-запрос на `/deleteTask/{id}`, с авторизацией):

```
@PreAuthorize("hasRole('MODERATOR')")
@GetMapping("/deleteTask/{id}")
public String deleteTask(@PathVariable Long id) {
    taskRepository.deleteById(id);
    return "redirect:/";
```

```
}
```

- Этот метод обрабатывает GET-запрос на URL `/deleteTask/{id}` и позволяет удалять задачу по её идентификатору (`id`).
- Аннотация `@PreAuthorize("hasRole('MODERATOR')")` ограничивает доступ к этому методу, разрешая его выполнение только пользователям с ролью `MODERATOR`.
- Задача удаляется из базы данных через `taskRepository.deleteById(id)`.
- По завершении происходит перенаправление на главную страницу для отображения обновленного списка задач.

Метод для обновления задачи (POST-запрос на `/updateTask/{id}`, с авторизацией):

```
@PreAuthorize("hasRole('MODERATOR')")
@PostMapping("/updateTask/{id}")
public String updateTask(@PathVariable Long id, @RequestParam boolean
completed) {
    Tasks task = taskRepository.findById(id).orElseThrow();
    task.setCompleted(completed);
    taskRepository.save(task);
    return "redirect:/";
}
```

- Этот метод обрабатывает POST-запрос на URL `/updateTask/{id}` и позволяет обновлять статус задачи (поле `completed`).
- Доступ к методу также ограничен аннотацией `@PreAuthorize("hasRole('MODERATOR')")`, что означает, что только пользователи с ролью `MODERATOR` могут изменять задачи.
- Он находит задачу по её идентификатору с помощью `taskRepository.findById(id)`, обновляет её состояние (поле `completed`), сохраняет изменения и перенаправляет на главную страницу.

Безопасность:

Аннотации `@PreAuthorize("hasRole('MODERATOR')")` добавляют уровень безопасности в приложение, обеспечивая выполнение операций удаления и обновления задач только пользователями, у которых есть роль `MODERATOR`. Это помогает предотвратить несанкционированные действия над задачами.

index.html

Немного модернизируем файл разметки.

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="UTF-8">
    <title>ToDo Tracker</title>
</head>
<body>
<h1>ToDo List</h1>
<ul>
    <!-- Отображение задач -->
    <li th:each="task : ${tasks}">
        <span th:text="${task.description}"></span> <!-- Описание задачи -->
        <span th:text="${task.completed ? 'Completed' : 'Pending'}"></span>
<!-- Статус задачи -->

        <!-- Ссылка для удаления задачи -->
        <a th:href="@{/deleteTask/{id}(id=${task.id})}">Delete</a>

        <!-- Форма для обновления статуса задачи -->
        <form th:action="@{/updateTask/{id}(id=${task.id})}" method="post"
style="display:inline;">
            <input type="hidden" name="completed" th:value="true"
th:if="${!task.completed}">
            <input type="hidden" name="completed" th:value="false"
th:if="${task.completed}">
            <button type="submit" th:text="${task.completed ? 'Mark as
Pending' : 'Mark as Completed'}"></button>
        </form>
    </li>
</ul>

<h2>Add New Task</h2>
<!-- Форма для добавления новой задачи -->
<form th:action="@{/addTask}" method="post">
    <label for="description">Description:</label>
    <input type="text" id="description" name="description" required>
    <button type="submit">Add Task</button>
</form>
</body>
</html>
```

- **`th:each="task : ${tasks}"`** — цикл Thymeleaf, который проходит по списку задач (`tasks`) и создает элемент списка (`<li>`) для каждой задачи.
- **`<span th:text="${task.description}"></span>`** — отображает описание задачи. Thymeleaf заменяет содержимое тега на значение поля `description` задачи.

- **`<span th:text="${task.completed ? 'Completed' : 'Pending'}"></span>`** — отображает статус задачи (выполнена/не выполнена). Если задача выполнена, показывается текст "Completed", иначе "Pending".

2. **Удаление задачи**:

```
<a th:href="@{/deleteTask/{id}(id=${task.id})}">Delete</a>
```

   - Генерируется ссылка для удаления задачи по её идентификатору. Thymeleaf заменяет переменную `id` на реальное значение ID задачи, и по нажатию на ссылку вызывается метод контроллера для удаления задачи.

3. **Форма для обновления статуса задачи**:

```
<form th:action="@{/updateTask/{id}(id=${task.id})}" method="post"
style="display:inline;">
    <input type="hidden" name="completed" th:value="true"
th:if="${!task.completed}">
    <input type="hidden" name="completed" th:value="false"
th:if="${task.completed}">
    <button type="submit" th:text="${task.completed ? 'Mark as Pending'
: 'Mark as Completed'}"></button>
</form>
```

- Форма отправляется на URL `/updateTask/{id}` с методом POST, чтобы обновить статус задачи.
- Используются скрытые поля `<input type="hidden">` для передачи значения `true` или `false` в зависимости от текущего статуса задачи:
  - Если задача не выполнена, передается `true`, чтобы отметить её как выполненную.
  - Если задача выполнена, передается `false`, чтобы отметить её как невыполненную.
- Текст кнопки также меняется в зависимости от текущего статуса задачи.

4. **Добавление новой задачи**:

```
<h2>Add New Task</h2>
<form th:action="@{/addTask}" method="post">
    <label for="description">Description:</label>
    <input type="text" id="description" name="description" required>
    <button type="submit">Add Task</button>
</form>
```

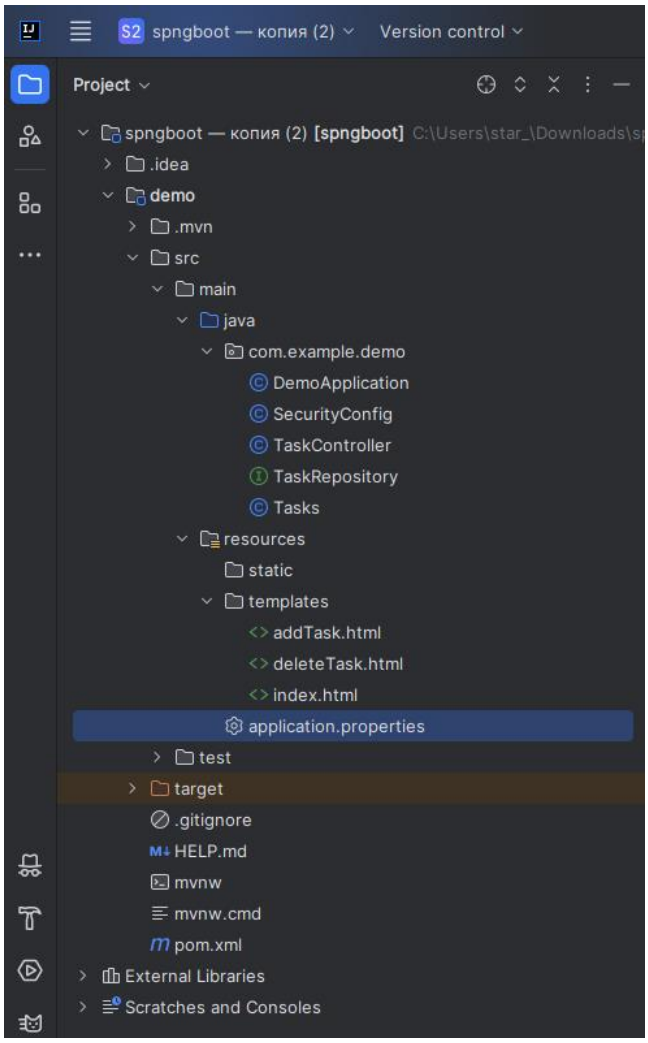
- Заголовок для добавления новой задачи.
- Форма отправляет POST-запрос на `/addTask` для добавления новой задачи.
- Поле для ввода описания задачи (`<input type="text" name="description">`) с обязательной валидацией (`required`).

Как это работает:

Thymeleaf: Этот шаблон Thymeleaf интегрируется с серверной логикой Spring Boot. Thymeleaf автоматически заменяет выражения, заключенные в `${}`, на реальные данные, переданные из контроллера (например, список задач или ID).

CRUD-операции: Этот шаблон позволяет пользователям просматривать, добавлять, обновлять и удалять задачи. Методы контроллера в `TaskController` принимают и обрабатывают запросы, отправляемые с этой страницы.

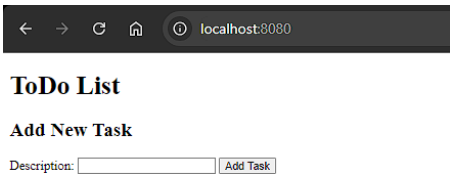
Структура проекта:



Проверим, как работает программа.

Запускаем pgAdmin 4.

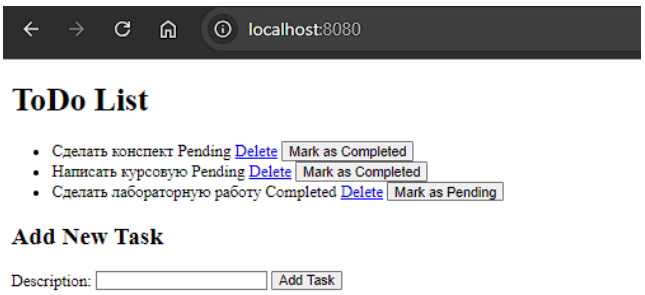
Запускаем Приложение.



Пустой список

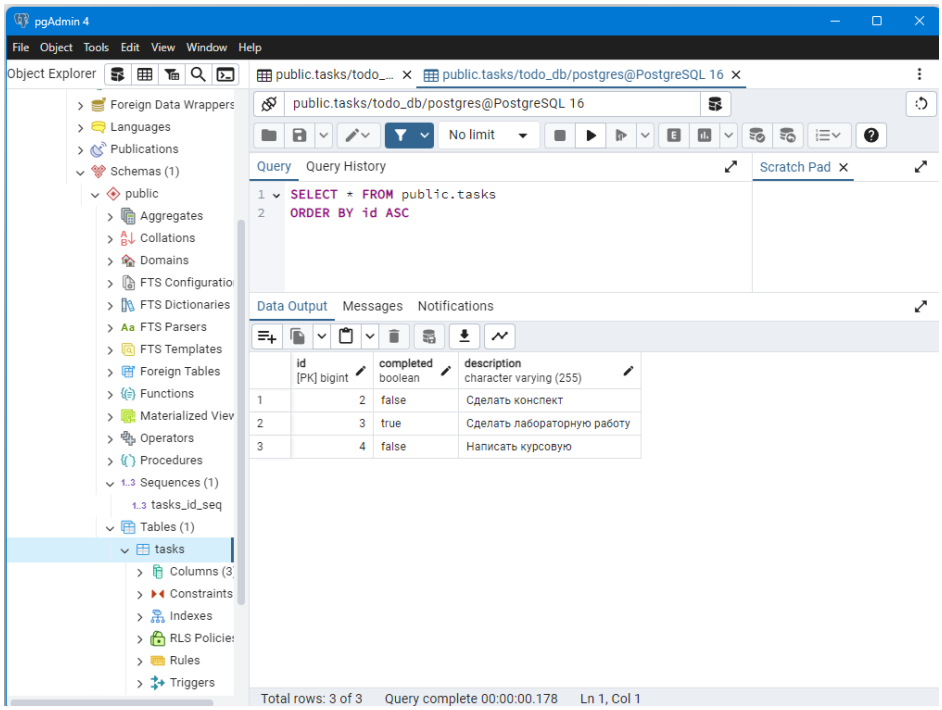
Открывается пустой список.

Добавляем несколько записей.



Заполненный список

Проверяем записи в базе данных на локальном сервере.



Данные в базе данных

Теперь все записи хранятся в базе данных.

Задание:

1. Изучить теоретический материал;

2. Написать приложение следуя примеру;

3. Выполнить дополнительное задание;

4. Сделать отчет.

Дополнительное задание.

- Модернизируйте таблицу tasks так, чтобы она хранила дополнительные сведенья о задаче время создания/приоритет/категория задачи
- Создайте таблицу `SubTask` для подзадач (ID, описание, статус, ID родительской задачи). Одна задача может иметь несколько подзадач (отношение "один ко многим").
- Добавьте таблицу `User` для хранения данных о пользователях (ID, имя, email, пароль), которые создают и управляют задачами. Свяжите таблицы `Task` и `User`.
- Создайте таблицу `LoginHistory`, которая будет хранить информацию о входах пользователей (ID, дата входа, ID пользователя, IP-адрес).
- Создайте таблицу `TaskHistory`, которая будет хранить историю изменений каждой задачи (ID, дата изменения, старое и новое состояние задачи, ID задачи).

Приложение

Полный код файла pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-parent</artifactId>
       <version>3.2.3</version>
       <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>demo</name>
    <description>Demo project for Spring Boot</description>
    <properties>
       <java.version>21</java.version>
    </properties>
    <dependencies>
       <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter</artifactId>
       </dependency>
       <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-test</artifactId>
          <scope>test</scope>
       </dependency>
       <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-web</artifactId>
       </dependency>
       <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-thymeleaf</artifactId>
       </dependency>
       <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-security</artifactId>
       </dependency>
       <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-data-jpa</artifactId>
       </dependency>
       <dependency>
          <groupId>org.postgresql</groupId>
          <artifactId>postgresql</artifactId>
          <scope>runtime</scope>
       </dependency>
    </dependencies>
    <build>
       <plugins>
          <plugin>
             <groupId>org.springframework.boot</groupId>
             <artifactId>spring-boot-maven-plugin</artifactId>
          </plugin>
       </plugins>
```

```
    </build>
</project>
```