






Spring Data

Урок 5



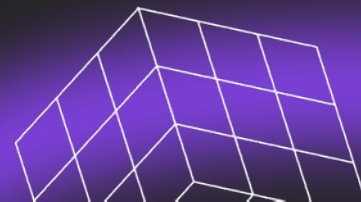
Что будет на уроке сегодня

-  Базы данных
-  Spring Data: Обобщение и упрощение работы с базами данных
-  Конфигурации Spring Data
-  Техническое задание на разработку Spring приложения
-  Проверка работы нашего приложения



Правильное хранение данных

Нам нужно думать о том, как организована наша база данных, какие типы данных мы храним, как они связаны друг с другом и как они будут использоваться.





Хранение данных в Java коллекциях

ArrayList

HashMap



Когда хранение данных в коллекциях эффективно

Если у нас небольшое приложение для управления задачами, где одновременно работает только один пользователь, и все данные сбрасываются после завершения сессии, Java коллекции могут быть более быстрыми и эффективными в использовании, чем полноценная база данных.





Базы данных

Базы данных были специально созданы для эффективного хранения, поиска и изменения больших объемов данных.

Они предлагают функции, такие как транзакции, которые гарантируют целостность данных, и индексы, которые ускоряют поиск данных.





SQL базы данных

SQL базы данных, такие как PostgreSQL, MySQL и Oracle, используют структурированный язык запросов (SQL) для управления данными.

SQL (Structured Query Language) – это стандартный язык для работы с базами данных, который используется для создания, изменения, управления и извлечения данных из реляционных баз данных.





NoSQL базы данных

NoSQL базы данных, такие как MongoDB, Cassandra и Redis.

NoSQL (Not Only SQL) базы данных – это тип баз данных, который был разработан для обработки больших объемов данных, которые не могут быть эффективно обработаны с помощью традиционных реляционных баз данных.





Взаимодействие с базами данных: JDBC, JPA и Spring Data. Использование JDBC.

JDBC, или Java Database Connectivity, – это API, которое позволяет нам взаимодействовать с базами данных напрямую из Java кода.

С его помощью мы можем выполнять SQL-запросы и обрабатывать результаты.





Взаимодействие с базами данных: JDBC, JPA и Spring Data. Использование JDBC.

JPA представляет собой спецификацию, которая описывает, как взаимодействовать с базами данных на уровне объектов.

Она позволяет нам использовать базы данных как хранилище объектов, минуя необходимость прямого написания SQL-кода.





Взаимодействие с базами данных: JDBC, JPA и Spring Data. Использование JDBC.

Spring Data – это слой абстракции, который построен поверх JPA (или других технологий доступа к данным), и который автоматизирует много общего кода, который нам пришлось бы написать самим.

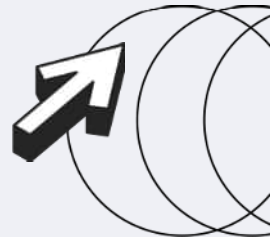




SPRING DATA. Spring Data Commons

Основой всего Spring Data является модуль Spring Data Commons.

Он предоставляет общие интерфейсы и классы, которые используются всеми остальными модулями Spring Data.

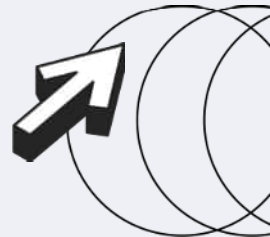




SPRING DATA. Spring Data JPA

Spring Data JPA – это модуль, который предоставляет поддержку для работы с SQL базами данных через JPA.

Он расширяет базовые интерфейсы Spring Data Commons и добавляет специфические для JPA функции, такие как поддержка JPQL (Java Persistence Query Language) и Specifications для динамического построения запросов.



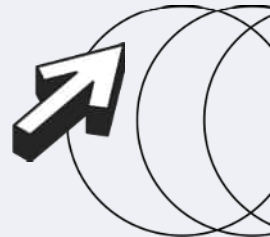


SPRING DATA.

Spring Data MongoDB, Spring Data Redis

Spring Data MongoDB позволяет нам работать с документами MongoDB, как если бы это были обычные Java объекты.

А Spring Data Redis предоставляет поддержку для структур данных Redis, таких как списки, множества и отсортированные множества.





Репозитории

Репозитории в Spring Data – это как магазины данных для ваших Java объектов.

Они обеспечивают доступ к базе данных и помогают вам выполнять различные операции над данными без необходимости писать SQL или JPQL код.





Как создать репозиторий в Spring Data

Все, что вам нужно сделать, – это определить интерфейс, который наследуется от одного из базовых интерфейсов репозитория Spring Data, таких как `CrudRepository` или `JpaRepository`.

Так, если у нас есть сущность `User`, мы можем создать репозиторий для него следующим образом:

```
1 public interface UserRepository extends CrudRepository<User, Long> {  
2 }  
3
```




Создание пользовательских запросов

Если мы хотим найти всех пользователей с определенным именем, мы можем просто добавить метод с подходящим именем в наш репозиторий:

```
1 public interface UserRepository extends CrudRepository<User, Long> {  
2     List<User> findByName(String name);  
3 }
```



Создание пользовательских запросов

Для сложных или специфических запросов, которые не поддаются генерации из названий методов, можно использовать аннотацию `@Query`, чтобы задать свой собственный запрос на языке, поддерживаемом вашей базой данных.

```
1 public interface UserRepository extends CrudRepository<User, Long> {  
2     @Query("SELECT u FROM User u WHERE u.email = ?1")  
3     User findByEmail(String email);  
4 }  
5
```



Создание JPA классов

Вот пример простого JPA класса:

```
1 @Entity
2 public class User {
3     @Id
4     @GeneratedValue(strategy=GenerationType.AUTO)
5     private Long id;
6
7     private String name;
8     private String email;
9
10    // геттеры и сеттеры
11 }
```



Создание JPA классов

В этом примере у нас есть класс User, который представляет пользователя нашего приложения.

- Аннотация @Entity на классе указывает, что этот класс является JPA сущностью и должен быть отображен на таблицу базы данных.
- Поле id отмечено аннотациями @Id и @GeneratedValue.
- Остальные поля класса будут отображены на колонки таблицы с теми же именами.



Аннотации для полей JPA классов

1

@Column

Позволяет задать имя колонки и другие параметры, такие как nullable и length.

2

@Temporal

Используется для указания типа даты/времени для поля java.util.Date или java.util.Calendar

3

@Enumerated

Указывает, что поле является перечислением.

4

@OneToMany.

@ManyToOne.

@ManyToMany

эти аннотации используются для отображения отношений между сущностями.



Конфигурация Spring Data с помощью Java кода

Вот базовый пример конфигурации Spring Data на Java:

```
1 @Configuration
2 @EnableJpaRepositories(basePackages =
   "com.example.myapp.repository")
3 @EnableTransactionManagement
4 public class JpaConfig {
5
6     @Bean
7     public DataSource dataSource() {
8         // создание и настройка источника данных
9     }
```



Конфигурация Spring Data с помощью Java кода

Вот базовый пример конфигурации Spring Data на Java (продолжение):

```
1 @Bean
2     public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
3         // создание и настройка фабрики EntityManager
4     }
5
6     @Bean
7     public PlatformTransactionManager transactionManager() {
8         // создание и настройка менеджера транзакций
9     }
10 }
```



Конфигурация Spring Data с помощью Java кода

- `@EnableJpaRepositories`: Эта аннотация активирует создание репозитория Spring Data.
- `@EnableTransactionManagement`: Эта аннотация включает поддержку управления транзакциями Spring.
- Методы, аннотированные `@Bean`, создают различные компоненты, необходимые для работы JPA.





Конфигурация Spring Data с помощью application.yaml

В файле application.yaml мы можем указать следующее:

```
1 spring:
2   datasource:
3     url: jdbc:mysql://localhost:3306/mydb
4     username: user
5     password: secret
6   jpa:
7     hibernate:
8       ddl-auto: update
9     show-sql: true|
```



Конфигурация Spring Data с помощью application.yaml

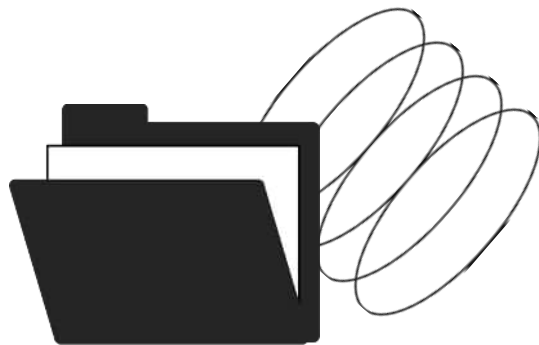
- `spring.datasource`: Здесь мы указываем параметры подключения к базе данных, такие как URL, имя пользователя и пароль.
- `spring.jpa.hibernate.ddl-auto`: Этот параметр определяет, как Hibernate должен управлять схемой базы данных.
- `spring.jpa.show-sql`: Если этот параметр установлен в `true`, Hibernate будет показывать SQL запросы, которые он выполняет.



Создание Spring проекта

Перейдите на сайт Spring Initializr и выберите следующие параметры:

- **Project:** Maven Project
- **Language:** Java
- **Spring Boot:** Выберите последнюю стабильную версию
- **Project Metadata:** Введите информацию о своем проекте. Например:
 - **Group:** com.example
 - **Artifact:** mynotes
 - **Name:** MyNotes
 - **Description:** A simple note management application
 - **Package Name:** com.example.mynotes





Создание Spring проекта

Выбор зависимостей.

Теперь нам нужно выбрать зависимости, которые нам понадобятся для нашего проекта:

- **Spring Web:** Для создания веб-приложения с использованием Spring MVC.
- **Spring Data JPA:** Для работы с базой данных через JPA.
- **Thymeleaf:** Для создания веб-страниц нашего приложения.
- **Spring Boot DevTools:** Для автоматической перезагрузки приложения при изменении кода.
- **PostgreSQL:** Драйвер для нашей базы данных. Мы будем использовать postgres.



Поднятие PostgreSQL в Docker

Шаг 1: Создание файла Dockerfile

Вначале нам нужно создать файл Dockerfile, который опишет наш контейнер. Создайте новый файл с именем Dockerfile и добавьте в него следующий код:

```
1 FROM postgres:13.2-alpine
2 ENV POSTGRES_DB mynotes
3 ENV POSTGRES_USER mynotes
4 ENV POSTGRES_PASSWORD secret
5
```



Поднятие PostgreSQL в Docker

Шаг 2: Создание и запуск контейнера Docker

Теперь мы можем создать и запустить наш контейнер Docker. Откройте терминал, перейдите в директорию, где находится ваш Dockerfile, и выполните следующую команду:

```
1 docker build -t mynotes-db |
```

Эта команда создаст новый образ Docker с именем “mynotes-db”. После того, как образ будет создан, вы можете запустить контейнер с помощью следующей команды:

```
1 docker run --name mynotes-db -p 5432:5432 -d mynotes-db|
```



Подключение к базе данных из Spring проекта

Теперь, когда у нас есть работающий контейнер с PostgreSQL, мы можем подключиться к нему из нашего Spring проекта. Для этого нам нужно добавить несколько строк в наш файл конфигурации application.yaml:

```
1 spring:
2   datasource:
3     url: jdbc:postgresql://localhost:5432/mynotes
4     username: mynotes
5     password: secret
6   jpa:
7     hibernate:
8       ddl-auto: update
9     show-sql: true
```



Создание классов моделей и JPA

Класс модели Note

Начнем с создания класса Note, который будет представлять наши заметки.

Этот класс будет простым POJO (Plain Old Java Object) с четырьмя полями: id, author, title, и content.

Добавим в него геттеры и сеттеры для этих полей:

```
1 public class Note {  
2     private Long id;  
3     private String author;  
4     private String title;  
5     private String content;  
6  
7     // геттеры и сеттеры  
8 }
```




Создание классов моделей и JPA

JPA класс NoteEntity

Теперь перейдем к созданию JPA класса NoteEntity, который будет отображать наши заметки на таблицу базы данных. Он будет очень похож на класс Note, но с добавлением аннотаций JPA:

```
1 @Entity
2 @Table(name = "notes")
3 public class NoteEntity {
4     @Id
5     @GeneratedValue(strategy=GenerationType.IDENTITY)
6     private Long id;
7
8     @Column(nullable = false)
9     private String author;
10
```



Создание классов моделей и JPA

JPA класс NoteEntity

```
11     @Column(nullable = false)
12     private String title;
13
14     @Column(nullable = false, length = 2000)
15     private String content;
16
17     // геттеры и сеттеры
18 }
```



Создание классов моделей и JPA

JPA класс NoteEntity

В этом классе мы видим несколько важных аннотаций:

- **@Entity:** эта аннотация указывает, что класс является JPA сущностью.
- **@Table:** эта аннотация позволяет нам указать имя таблицы, на которую будет отображаться наш класс.
- **@Id и @GeneratedValue:** эти аннотации указывают, что поле id является идентификатором и его значение должно быть сгенерировано автоматически.
- **@Column:** эта аннотация позволяет нам указать параметры для колонки, соответствующей данному полю.



Создание интерфейса репозитория

Для нашего приложения нам понадобится репозиторий `NoteRepository`, который будет обеспечивать операции CRUD для нашего класса `NoteEntity`:

```
1 public interface NoteRepository extends JpaRepository<NoteEntity, Long> {  
2 }
```

В этом интерфейсе мы указываем два параметра типа для `JpaRepository`: тип нашей сущности (`NoteEntity`) и тип идентификатора (`Long`). `JpaRepository` предоставляет нам множество полезных методов, таких как `findAll()`, `findById()`, `save()`, `delete()`, и т.д., без необходимости их реализовывать.



Добавление пользовательских методов в репозиторий

В дополнение к методам, предоставляемым JpaRepository, мы можем добавить свои собственные методы в репозиторий. Например, давайте добавим метод для поиска заметок по автору:

```
1 public interface NoteRepository extends JpaRepository<NoteEntity, Long> {  
2     List<NoteEntity> findByAuthor(String author);  
3 }
```



Создание интерфейса сервиса

Вначале, создадим интерфейс для нашего сервиса. Это хорошая практика, поскольку она делает наш код более гибким и тестируемым:

```
1 public interface NoteService {  
2     List<Note> getAllNotes();  
3     Note getNoteById(Long id);  
4     Note createNote(Note note);  
5     Note updateNote(Long id, Note note);  
6     void deleteNote(Long id);  
7 }
```



Реализация интерфейса сервиса

Теперь создадим класс `NoteServiceImpl`, который будет реализовывать наш интерфейс `NoteService`:

```
1 @Service
2 public class NoteServiceImpl implements NoteService {
3     private final NoteRepository repository;
4
5     @Autowired
6     public NoteServiceImpl(NoteRepository repository) {
7         this.repository = repository;
8     }
9
10    @Override
11    public List<Note> getAllNotes() {
12        return repository.findAll();
13    }
```



Реализация интерфейса сервиса

```
1 @Override
2     public Note getNoteById(Long id) {
3         return repository.findById(id)
4             .orElseThrow(() → new RuntimeException("Note not found"));
5     }
6
7     @Override
8     public Note createNote(Note note) {
9         return repository.save(note);
10    }
```




Реализация интерфейса сервиса

```
1 @Override
2     public Note updateNote(Long id, Note note) {
3         // мы должны сначала проверить, существует ли заметка с данным ID
4         Note existingNote = getNoteById(id);
5         // обновляем поля существующей заметки
6         existingNote.setTitle(note.getTitle());
7         existingNote.setContent(note.getContent());
8         // сохраняем и возвращаем обновленную заметку
9         return repository.save(existingNote);
10    }
11
12    @Override
13    public void deleteNote(Long id) {
14        // проверяем, существует ли заметка с данным ID
15        getNoteById(id);
16        // если да, то удаляем ее
17        repository.deleteById(id);
18    }
19 }
```



Как это работает в Spring

Для этого нам нужно добавить новый тег в наш HTML-файл – `<a>`.

Этот тег создает ссылку на другую страницу. Вот как это выглядит:

```
1 <body>
2     <h1 id="greeting">Привет, мир!</h1>
3     <p>Добро пожаловать в мое первое Spring приложение!</p>
4     <a href="/other-page">Посмотреть другую страницу</a>
5
6     <script>
7         document.getElementById('greeting').textContent = 'Привет, Spring!';
8     </script>
9 </body>
```



Создание класса контроллера

Для создания контроллера создадим новый класс `NoteController` и аннотируем его как `@RestController`. Это указывает Spring, что этот класс будет обрабатывать веб-запросы:

```
1 @RestController
2 @RequestMapping("/api/notes")
3 public class NoteController {
4     private final NoteService service;
5
6     @Autowired
7     public NoteController(NoteService service) {
8         this.service = service;
9     }
10
11     // методы контроллера
12 }
```



Добавление методов контроллера

Теперь добавим методы в наш контроллер для каждой из операций, которые мы хотим поддерживать:

```
1 @GetMapping
2 public List<Note> getAllNotes() {
3     return service.getAllNotes();
4 }
5
6 @GetMapping("/{id}")
7 public Note getNoteById(@PathVariable Long id) {
8     return service.getNoteById(id);
9 }
10
11 @PostMapping
12 public Note createNote(@RequestBody Note note) {
13     return service.createNote(note);
14 }
15
16 @GetMapping("/{id}")
```



Добавление методов контроллера

```
16 @PutMapping("/{id}")
17 public Note updateNote(@PathVariable Long id, @RequestBody Note note) {
18     return service.updateNote(id, note);
19 }
20
21 @DeleteMapping("/{id}")
22 public void deleteNote(@PathVariable Long id) {
23     service.deleteNote(id);
24 }
```



Добавление методов контроллера

Каждый из этих методов соответствует определенному типу HTTP-запроса (GET, POST, PUT, DELETE) и обрабатывает определенный тип операции.

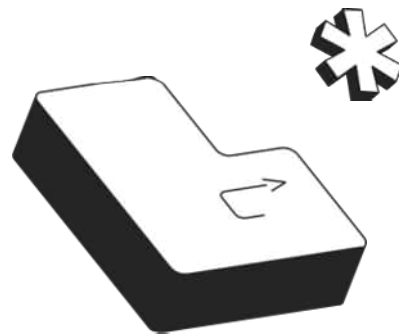
- **@GetMapping**, **@PostMapping**, **@PutMapping** и **@DeleteMapping** – это специализированные версии аннотации **@RequestMapping**, которые указывают тип HTTP-запроса.
- **@PathVariable** используется для привязки части URL к параметру метода.
- **@RequestBody** используется для привязки тела запроса к параметру метода



Проверка работы приложения

Шаг 1: Запуск приложения

Сначала убедимся, что наше приложение запускается без ошибок. Если вы видите сообщение, что приложение успешно запущено и доступно по адресу <http://localhost:8080>, это хороший знак.





Проверка работы приложения

Шаг 2. Получение списка всех заметок: Отправьте GET-запрос к `http://localhost:8080/api/notes`.

Вы должны получить список всех заметок, включая только что созданную.

Шаг 3. Обновление заметки: Отправьте PUT-запрос к `http://localhost:8080/api/notes/{id}`, где `{id}` – это ID заметки, которую вы хотите обновить. В теле запроса укажите новые данные для заметки, например:

```
1 {  
2   "author": "New author",  
3   "title": "New title",  
4   "content": "New content"  
5 }  
6 |
```

В ответ вы должны получить обновленную заметку.

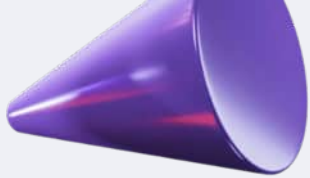


Проверка работы приложения

Шаг 4. Удаление заметки: Отправьте DELETE-запрос к `http://localhost:8080/api/notes/{id}`, где `{id}` – это ID заметки, которую вы хотите удалить. В ответ вы не должны получить тело сообщения, и при повторном запросе списка заметок удаленной заметки в нем быть не должно.

После выполнения этих шагов мы можем быть уверены, что наше приложение корректно работает с базой данных и выполняет все необходимые операции.





Спасибо за внимание

