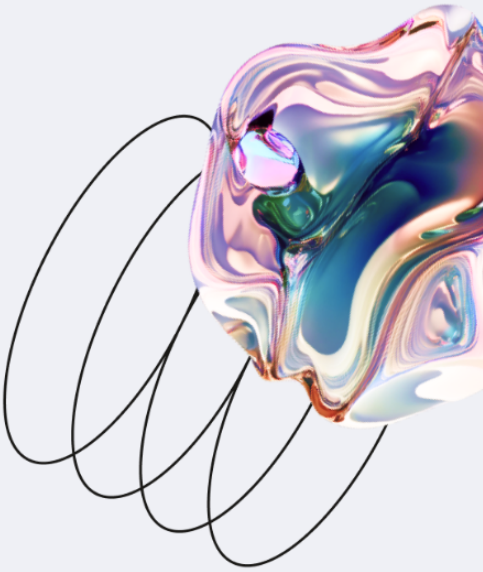
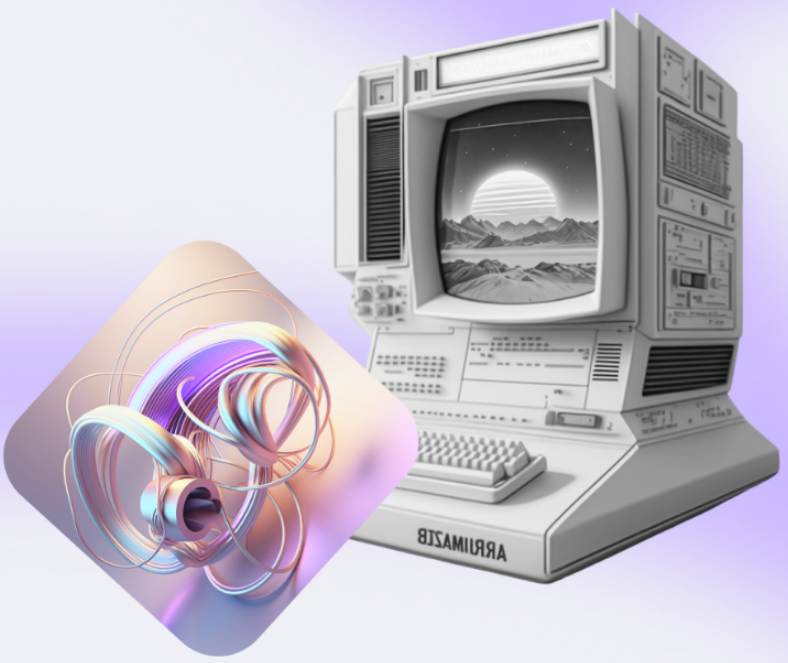
GeekBrains



Spring Data для работы с базами данных

Фреймворк Spring



Оглавление

Введение

Привет, друзья! Встречайте новый урок из нашего курса по фреймворку Spring. Сегодня мы с вами окунемся в увлекательный мир Spring Data. Возможно, ты уже слышал о нем или, быть может, впервые слышишь. Но поверь мне, это тема, которая откроет тебе новые горизонты в разработке серверных приложений.

Но что же такое Spring Data? Это подпроект Spring, предназначенный для упрощения работы с базами данных. С его помощью ты можешь организовывать взаимодействие с базой данных на уровне Java-кода, а сам Spring Data заботится о всей “тяжелой работе” за тебя. По-настоящему мощная штука, не правда ли?

Но почему мы так увлечены этой темой? Данные – это сердце любого приложения, и быть способным эффективно работать с ними – это ключ к созданию успешных проектов.

Помнишь, как мы говорили, что Spring MVC позволяет организовать обмен данными между клиентом и сервером? И как Thymeleaf помогает нам отображать эти данные в виде великолепных веб-страниц?

Вот здесь-то и вступает в игру Spring Data. Ведь откуда эти данные берутся? Да, именно из базы данных. Spring Data обеспечивает нам простой и удобный способ извлечения этих данных, обработки их и сохранения обратно в базу. Так что на самом деле, Spring Data – это как магический мост, связывающий наши темы воедино.

Сегодня мы погрузимся в эту тему и узнаем, как Spring Data может сделать нашу жизнь проще и наши проекты эффективнее. Готовы? Тогда поехали!

Многие говорят, что данные – это новое золото, и на самом деле, они абсолютно правы. В 21 веке данные стали невероятно ценным активом. Но почему? Ведь, это же просто... данные, не так ли?

Спойлер: нет. Данные – это нечто большее. Они являются основой для принятия решений, а также стимулом для инноваций в различных отраслях от здравоохранения до финтеха. Более того, данные позволяют нам понять сложные паттерны, предсказывать поведение и оптимизировать процессы.

Термины, используемые в лекции

Spring Data Repository – Это интерфейс, который предоставляет базовые операции CRUD (создание, чтение, обновление, удаление) для сущности. Repository реализует шаблон проектирования Repository, описанный в Domain-Driven Design (DDD).

CRUD - Сокращение от Create, Read, Update, Delete – это базовые операции, которые могут быть выполнены в любой базе данных.

JPA (Java Persistence API) – это спецификация для управления, доступа и сохранения Java объектов в базу данных. Это способ взаимодействия с базой данных с помощью объектно-ориентированного подхода.

Spring Data JPA – Это подпроект Spring Data, который предоставляет реализацию Repository, используя JPA. Он упрощает написание кода для работы с базами данных, предоставляя реализацию шаблонов CRUD и позволяя вам фокусироваться на написании бизнес-логики.

Hibernate – Это популярная реализация JPA. Она предоставляет собственные API поверх стандартного JPA для расширенного управления и взаимодействия с базами данных.

@Entity – Аннотация, используемая для обозначения класса в JPA как сущности, что означает, что этот класс может быть отображен на таблицу в базе данных.

@Table – Аннотация, используемая для указания имени таблицы, к которой относится сущность.

@Id – Аннотация, используемая для обозначения первичного ключа сущности.

@GeneratedValue – Аннотация, используемая для указания стратегии генерации первичного ключа.

Query methods – Методы, определенные в интерфейсе Repository, которые автоматически создают SQL-запросы на основе своего названия.

@Query – Аннотация, используемая для задания произвольного SQL или JPQL запроса.

Transaction – Транзакция описывает последовательность действий, которые приводят базу данных из одного согласованного состояния в другое.

Spring Data REST – Это подпроект Spring Data, который предоставляет возможность быстрого создания RESTful сервисов на основе репозиториев Spring Data

Хранение данных

Давай рассмотрим несколько примеров. Вы когда-нибудь задумывались, почему Amazon умеет рекомендовать вам именно те товары, которые вам скорее всего понравятся? Или почему Netflix всегда знает, какие фильмы или сериалы

предложить вам следующими? Ответ прост: они используют данные о вашем поведении, чтобы улучшить свои продукты и услуги.

Компания Google анализирует миллиарды поисковых запросов каждый день, чтобы сделать результаты поиска еще более релевантными. А Facebook использует данные о ваших предпочтениях, чтобы показывать вам наиболее подходящую рекламу.

Но все эти суперсилы приходят с ответственностью. Данные становятся бесценными только тогда, когда они правильно хранятся, обрабатываются и анализируются. Если мы храним данные неструктурированно или без должного учета, они могут стать бесполезными, или еще хуже, вредными.

Вот почему важно внимательно подходить к вопросу хранения 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. И вот здесь на помощь приходит Spring Data.

В следующих разделах мы разберемся, как Spring Data может помочь нам эффективно организовать хранение данных и использовать все преимущества “золота 21 века”.

Давай представим, что у нас есть веб-приложение, которое позволяет пользователям записывать свои заметки. Мы могли бы хранить все эти заметки прямо в Java-приложении, используя коллекции, такие как ArrayList или HashMap.

Список (ArrayList) был бы идеальным решением, если бы мы хотели сохранять заметки в порядке их добавления. А если бы нам нужно было быстро находить заметку по определенному ключу, например, по ID пользователя, мы могли бы использовать словарь (HashMap).

В некоторых случаях, например, когда у нас мало данных или когда данные не нужно сохранять между сессиями, использование Java коллекций может быть вполне эффективным.

Например, 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

Однако для большинства реальных приложений подход с Java коллекциями не подойдет. А теперь я расскажу почему.

Во-первых, если наше приложение должно поддерживать нескольких пользователей одновременно, возникнут проблемы с синхронизацией данных. Во-вторых, данные в коллекциях не сохраняются после перезапуска приложения, что является критическим недостатком для большинства приложений.

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.

Также базы данных обеспечивают постоянное хранение данных. Это значит, что данные не пропадут при перезапуске приложения и будут доступны для многих приложений одновременно.

Так что, несмотря на то, что Java коллекции могут быть полезны в определенных ситуациях, для сложных приложений, работающих с большими объемами данных, нам понадобится что-то более мощное. Именно здесь на сцену выходят базы данных и Spring Data.

Мир баз данных: SQL и NoSQL

По мере того, как компьютерные технологии развивались, развивались и базы данных. Сегодня существует огромное количество различных типов баз данных, но все они можно разделить на две большие категории: SQL и NoSQL.

SQL базы данных

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

Считайте, что SQL базы данных - это как шкаф с большим количеством отсеков. У каждого отсека есть свои метки (названия колонок в таблице), и каждый отсек строго определен. Если вы хотите сохранить новый предмет (запись), вы должны убедиться, что у вас есть отсек, подходящий для каждой его части (поля данных).

SQL базы данных очень хороши в управлении структурированными данными, где все поля известны заранее и редко меняются. Они также предлагают мощные средства для выполнения сложных запросов и гарантируют целостность данных с помощью транзакций.

NoSQL базы данных

С другой стороны, у нас есть NoSQL базы данных, такие как MongoDB, Cassandra и Redis. NoSQL означает “не только SQL”, и это базы данных, которые не следуют строгой табличной структуре SQL баз данных.

Вернемся к нашей аналогии. NoSQL базы данных можно представить как комнату с мешками для хранения вещей. У вас есть мешки разных форм и размеров, и вы можете положить в них что угодно, без необходимости следовать строгим правилам размещения, как в случае с отсеками шкафа.

NoSQL базы данных хороши для работы с неструктурированными или полуструктурированными данными, где схема может часто меняться. Они также могут быть очень полезными для работы с большими объемами данных, так как они обычно легко масштабируются.

Как видите, обе эти категории имеют свои преимущества и недостатки, и важно выбрать правильный тип базы данных в зависимости от специфики вашего проекта.

Теперь, когда мы знаем, что такое базы данных и какие они бывают, давайте поговорим о том, как мы можем взаимодействовать с ними на уровне кода. Существует несколько способов, и первый из них – это использование JDBC.

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.

Например, мы можем отправить SQL-запрос, чтобы получить все заметки пользователя из нашей базы данных. Затем мы можем проитерироваться по результатам и создать объекты Java для каждой заметки. Но в этом подходе есть много “ручной” работы, и он может стать сложным и трудоемким при больших проектах.

Использование JPA

Здесь на помощь приходит JPA, или Java Persistence API. 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.

С JPA мы можем определить классы-сущности, которые отображаются на таблицы базы данных, и выполнять операции над этими объектами, как если бы они были обычными Java объектами.

Использование Spring Data

Но даже с JPA у нас все равно есть некоторый объем “ручной” работы. Нам нужно определять репозитории для каждой сущности и написать запросы для сложных операций. И здесь мы подходим к Spring Data.

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.

С помощью Spring Data, большинство базовых операций над данными становятся настолько простыми, что нам даже не нужно писать код для них. Вместо этого мы можем просто определить интерфейс, и Spring Data сгенерирует все необходимые методы за нас.

В следующем разделе мы погрузимся в конкретные примеры использования Spring Data, и вы увидите, как это может упростить вашу работу с базами данных.

Spring Data

Как мы обсуждали в предыдущем разделе, работа с базами данных может быть сложной и трудоемкой задачей. Именно здесь Spring Data выходит на сцену. Он обобщает базы данных и упрощает их использование, сокращая количество рутинного кода, который нам нужно написать.

Но каким образом Spring Data это делает? Внутри этого подпроекта Spring скрыто много различных технологий, каждая из которых служит определенной цели. Давайте познакомимся с некоторыми из них.

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. Например, он определяет интерфейс Repository, который является базой для всех репозиториев Spring Data.

Спринг Data Commons можно представить как фундамент здания. Он не предоставляет никаких конкретных возможностей сам по себе, но он необходим для поддержки остальной структуры.

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 JPA можно представить как рабочий инструмент, который преобразует общие принципы Spring Data Commons в конкретные действия, специфичные для работы с JPA.

Spring Data MongoDB, Spring Data Redis и другие

Кроме JPA, Spring Data предоставляет поддержку для многих других типов баз данных, включая NoSQL базы данных, такие как MongoDB и Redis. Каждый из этих

модулей аналогичен Spring Data JPA, но они предоставляют функции, специфичные для своих баз данных.

Например, 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.

Каждый из этих модулей можно представить как дополнительный инструмент в нашем наборе инструментов Spring Data. Они позволяют нам использовать те же общие принципы для работы с различными базами данных.

Теперь, когда мы знакомы с основными технологиями, которые лежат в основе Spring Data, давайте посмотрим, как мы можем использовать их на практике.

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.

Думайте об этом как о книжном магазине. У вас есть много книг (данных), которые нужно упорядочить, и вы хотите легко и быстро найти нужную вам книгу. Вместо того чтобы искать каждую книгу вручную, вы обращаетесь к продавцу (репозиторию), который знает, где находится каждая книга и может найти ее для вас.

Создание репозитория в Spring Data – это процесс, который на удивление прост. Все, что вам нужно сделать, – это определить интерфейс, который наследуется от одного из базовых интерфейсов репозитория Spring Data, таких как CrudRepository или JpaRepository.

Эти базовые интерфейсы уже содержат множество полезных методов для работы с данными, таких как save(), delete(), findAll() и findById().

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

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

В этом случае User – это тип сущности, а Long – это тип идентификатора этой сущности.

Создание пользовательских запросов

Но что, если нам нужно что-то более сложное, чем базовые операции CRUD? Spring Data предлагает удивительно мощный механизм для этого – возможность создания запросов прямо из названий методов!

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

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

Spring Data автоматически сгенерирует запрос для этого метода на основе его имени. Очень удобно, не правда ли?

Более того, 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:

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

В этом примере мы определяем запрос JPQL прямо в нашем репозитории, который найдет пользователя по его адресу электронной почты.

Вместе, эти функции делают репозитории Spring Data мощным и гибким инструментом для работы с базами данных.

JPA классы

В работе с базами данных через JPA нам часто приходится сталкиваться с так называемыми JPA классами или сущностями. Сущности JPA – это обычные Java классы, которые представляют таблицы в базе данных и строку в этих таблицах.

Можно сказать, что JPA классы это как дипломаты в мире Java и баз данных. Они говорят на “языке” Java, но представляют и защищают интересы баз данных.

Создание JPA классов довольно просто. Вот пример простого JPA класса:

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    private String name;
    private String email;

    // геттеры и сеттеры
}
```

В этом примере у нас есть класс User, который представляет пользователя нашего приложения. Мы видим здесь несколько важных моментов:

- 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. Это говорит о том, что это поле является идентификатором (ключом) и его значение генерируется автоматически.
- Остальные поля класса будут отображены на колонки таблицы с теми же именами.

JPA предоставляет множество других аннотаций, которые можно использовать для более детального управления отображением полей класса на колонки таблицы:

- @Column: позволяет задать имя колонки и другие параметры, такие как nullable и length.

- @Temporal: используется для указания типа даты/времени для поля java.util.Date или java.util.Calendar.
- @Enumerated: указывает, что поле является перечислением.
- @OneToMany, @ManyToOne, @OneToOne, @ManyToMany: эти аннотации используются для отображения отношений между сущностями.

Здесь только некоторые из многих аннотаций JPA, но они уже позволяют нам создавать богатые и сложные модели данных.

Конфигурация Spring Data

Как и большинство компонентов Spring, Spring Data нуждается в правильной конфигурации для своей работы. Это важно для того, чтобы указать Spring Data, как подключиться к базе данных, как работать с транзакциями и как отобразить наши JPA классы на таблицы базы данных.

Конфигурация Spring Data с помощью Java кода

Вот базовый пример конфигурации Spring Data на Java:

```
@Configuration
@EnableJpaRepositories(basePackages = "com.example.myapp.repository")
@EnableTransactionManagement
public class JpaConfig {

    @Bean
    public DataSource dataSource() {
        // создание и настройка источника данных
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean
entityManagerFactory() {
        // создание и настройка фабрики EntityManager
```

```
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        // создание и настройка менеджера транзакций
    }
}
```

- @EnableJpaRepositories: Эта аннотация активирует создание репозиториев Spring Data. В параметре basePackages указывается пакет, где Spring должен искать интерфейсы репозиториев.
- @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

Вместо или в дополнение к Java-конфигурации, мы можем использовать файлы properties или yaml для настройки Spring Data. Например, в файле application.yaml мы можем указать следующее:

```yaml
spring:
 datasource:
  url: jdbc:mysql://localhost:3306/mydb
  username: user
  password: secret
 jpa:
  hibernate:
   ddl-auto: update
  show-sql: true
```

- 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. Значение update означает, что Hibernate

будет автоматически обновлять схему базы данных в соответствии с нашими JPA классами.

- spring.jpa.show-sql: Если этот параметр установлен в true, Hibernate будет показывать SQL запросы, которые он выполняет.

Техническое задание

Отлично, теперь мы знакомы с основами Spring Data, давайте перейдем к практической части. Представим, что мы получили следующее техническое задание от нашего заказчика.

Заказчик хочет, чтобы мы разработали простое веб-приложение для управления заметками. У каждой заметки есть автор, заголовок и текст. Пользователи должны иметь возможность создавать новые заметки, просматривать существующие, редактировать и удалять их.

В рамках данного задания нам нужно разработать следующие компоненты:

- Модель Note, которая будет представлять заметку. У заметки должны быть поля id, author, title и content.
- Репозиторий NoteRepository, который будет предоставлять операции CRUD над заметками.
- Сервис NoteService, который будет использовать NoteRepository для выполнения бизнес-логики приложения.
- Контроллер NoteController, который будет обрабатывать веб-запросы от пользователей и использовать NoteService для выполнения операций.

В следующих разделах мы будем постепенно реализовывать каждый из этих компонентов, начиная с модели Note и заканчивая контроллером NoteController.

Создание нового Spring проекта

Чтобы начать разработку нашего приложения, нам нужно сначала создать новый Spring проект. Самый простой способ сделать это - использовать Spring Initializr.

Шаг 1: Настройка проекта

Перейдите на сайт Spring Initializr и выберите следующие параметры:

- **Project**: Maven Project (если вы предпочитаете Gradle, можете выбрать Gradle 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

Шаг 2: Выбор зависимостей

Теперь нам нужно выбрать зависимости, которые нам понадобятся для нашего проекта. В случае с нашим приложением для управления заметками, нам понадобятся следующие зависимости:

- **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 (это не обязательно, если мы планируем создать REST API).
- **Spring Boot DevTools**: Для автоматической перезагрузки приложения при изменении кода.
- **PostgreSQL**: Драйвер для нашей базы данных. Мы будем использовать postgres.

Шаг 3: Генерация проекта

После того, как вы выбрали все параметры, нажмите кнопку “Generate”, чтобы сгенерировать и скачать архив с вашим новым проектом. Распакуйте архив в удобное для вас место и откройте проект в вашей любимой среде разработки.

Теперь у вас есть новый Spring проект, готовый к разработке!

Поднятие PostgreSQL в Docker

Docker - это удобный инструмент для запуска различных сервисов, таких как базы данных, в изолированных контейнерах. Давайте создадим контейнер Docker для PostgreSQL.

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:

```
FROM postgres:13.2-alpine
ENV POSTGRES_DB mynotes
ENV POSTGRES_USER mynotes
ENV POSTGRES_PASSWORD secret
```

В этом файле мы указываем, что хотим использовать образ postgres версии 13.2 на базе alpine. Мы также устанавливаем несколько переменных среды для задания имени базы данных, пользователя и пароля.

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:

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:

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

Эта команда создаст и запустит новый контейнер с именем “mynotes-db”, привязав порт 5432 на вашем компьютере к порту 5432 в контейнере.

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:

```yaml
spring:
 datasource:
  url: jdbc:postgresql://localhost:5432/mynotes
  username: mynotes
  password: secret
 jpa:
  hibernate:
   ddl-auto: update
  show-sql: true
```

Здесь мы указываем, что хотим подключиться к базе данных PostgreSQL на localhost на порту 5432, и указываем имя базы данных, имя пользователя и пароль, которые мы задали при создании контейнера Docker.

Теперь у нас есть все, что нужно для работы с базой данных PostgreSQL в нашем Spring проекте.

Создание классов моделей и JPA

Для нашего приложения для управления заметками, нам понадобится одна модель – Note, и соответствующий ей JPA класс – NoteEntity.

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:

```java
public class Note {
    private Long id;
```

```
    private String author;
    private String title;
    private String content;

    // геттеры и сеттеры
}
```

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:

```
@Entity
@Table(name = "notes")
public class NoteEntity {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String author;

    @Column(nullable = false)
    private String title;

    @Column(nullable = false, length = 2000)
    private String content;

    // геттеры и сеттеры
}
```

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. В данном случае, мы указываем, что все поля являются обязательными (nullable = false), а поле content ограничиваем 2000 символами в длину.

Теперь у нас есть все необходимые классы для представления наших заметок в приложении и в базе данных.

Создание слоя репозитория

В нашем приложении нам потребуется слой репозитория для взаимодействия с базой данных. В Spring Data это очень просто сделать с помощью интерфейсов репозитория.

Создание интерфейса репозитория

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

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

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:

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

Мы просто определили метод с именем findByAuthor, и Spring Data автоматически предоставит его реализацию для нас. Теперь, когда мы вызываем этот метод, Spring

Data выполнит запрос к базе данных, чтобы найти все заметки с указанным автором.

Таким образом, слой репозитория позволяет нам легко и быстро взаимодействовать с базой данных, без необходимости писать сложный код для этого.

Создание сервис-слоя

Слой сервиса – это место, где выполняется основная бизнес-логика нашего приложения. В нашем случае нам нужно создать сервис NoteService, который будет использовать наш NoteRepository для выполнения операций над заметками.

Создание интерфейса сервиса

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

```
public interface NoteService {
    List<Note> getAllNotes();
    Note getNoteById(Long id);
    Note createNote(Note note);
    Note updateNote(Long id, Note note);
    void deleteNote(Long id);
}
```

Здесь мы определяем пять основных операций, которые мы хотим выполнять над нашими заметками: получить все заметки, получить заметку по ID, создать новую заметку, обновить существующую заметку и удалить заметку.

Реализация интерфейса сервиса

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

```
@Service
public class NoteServiceImpl implements NoteService {
    private final NoteRepository repository;

    @Autowired
    public NoteServiceImpl(NoteRepository repository) {
```

```
        this.repository = repository;
    }

    @Override
    public List<Note> getAllNotes() {
        return repository.findAll();
    }

    @Override
    public Note getNoteById(Long id) {
        return repository.findById(id)
                .orElseThrow(() -> new RuntimeException("Note not found"));
    }

    @Override
    public Note createNote(Note note) {
        return repository.save(note);
    }

    @Override
    public Note updateNote(Long id, Note note) {
        // мы должны сначала проверить, существует ли заметка с данным ID
        Note existingNote = getNoteById(id);
        // обновляем поля существующей заметки
        existingNote.setTitle(note.getTitle());
        existingNote.setContent(note.getContent());
        // сохраняем и возвращаем обновленную заметку
        return repository.save(existingNote);
    }

    @Override
    public void deleteNote(Long id) {
        // проверяем, существует ли заметка с данным ID
        getNoteById(id);
        // если да, то удаляем ее
        repository.deleteById(id);
    }
}
```

Здесь мы используем NoteRepository для выполнения операций над нашими заметками. Благодаря Spring Data, нам не нужно писать много кода для этого – мы просто вызываем методы, предоставляемые NoteRepository.

Создание контроллера

В нашем приложении контроллер будет обрабатывать HTTP-запросы от клиентов и вызывать соответствующие методы нашего сервиса NoteService.

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:

```
@RestController
@RequestMapping("/api/notes")
public class NoteController {
    private final NoteService service;

    @Autowired
    public NoteController(NoteService service) {
        this.service = service;
    }

    // методы контроллера
}
```

В этом классе мы используем NoteService для выполнения операций над заметками. Аннотация @RequestMapping("/api/notes") указывает, что все методы в этом контроллере будут обрабатывать запросы, начинающиеся с /api/notes.

Добавление методов контроллера

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

```
@GetMapping
public List<Note> getAllNotes() {
    return service.getAllNotes();
}
```

```
@GetMapping("/{id}")
public Note getNoteById(@PathVariable Long id) {
    return service.getNoteById(id);
}

@PostMapping
public Note createNote(@RequestBody Note note) {
    return service.createNote(note);
}

@PutMapping("/{id}")
public Note updateNote(@PathVariable Long id, @RequestBody Note note) {
    return service.updateNote(id, note);
}

@DeleteMapping("/{id}")
public void deleteNote(@PathVariable Long id) {
    service.deleteNote(id);
}
```

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.

Таким образом, контроллер предоставляет “входную точку” в наше приложение для клиентов, переводя HTTP-запросы в вызовы методов нашего сервиса.

Завершение настройки нашего приложения

На данный момент наше приложение уже имеет все необходимые компоненты: модель, репозиторий, сервис и контроллер. Также мы подняли базу данных в Docker

и настроили подключение к ней. Но, чтобы все заработало, нам еще потребуется главный класс, который запустит наше приложение.

Создание главного класса

В Spring Boot приложении обычно есть один главный класс, который используется для запуска приложения. Создайте новый класс с именем Application и добавьте в него следующий код:

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

@SpringBootApplication – это удобная аннотация, которая включает в себя несколько других аннотаций Spring:

- @Configuration делает этот класс источником определений бинов для контекста приложения.
- @EnableAutoConfiguration говорит Spring Boot начать добавлять бины в зависимости от настроек класса, других бинов и различных настроек свойств.
- @ComponentScan говорит Spring искать другие компоненты, конфигурации и сервисы в пакете приложения, позволяя ему находить контроллеры.

Метод main() запускает наше приложение с помощью SpringApplication.run(), который начинает выполнение Spring и, следовательно, веб-сервер Tomcat.

Запуск приложения

Теперь мы готовы запустить наше приложение. Для этого просто запустите главный класс Application из вашей среды разработки, как обычное Java приложение. Если все настроено правильно, вы должны увидеть в консоли логи запуска Spring и Tomcat, и ваше приложение будет доступно по адресу http://localhost:8080.

Проверка работы нашего приложения

Чтобы проверить, что наше приложение работает корректно и соответствует техническому заданию, мы можем выполнить следующие шаги.

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.

Шаг 2: Проверка работы с базой данных

Чтобы проверить, что наше приложение корректно работает с базой данных, мы можем использовать инструменты, такие как Postman или curl для отправки HTTP-запросов к нашему приложению.

1. **Создание новой заметки**: Отправьте POST-запрос к http://localhost:8080/api/notes со следующим телом запроса:

   ```
   {
       "author": "Author",
       "title": "Title",
       "content": "Content"
   }
   ```

   В ответ вы должны получить созданную заметку с ID.

2. **Получение списка всех заметок**: Отправьте GET-запрос к http://localhost:8080/api/notes. Вы должны получить список всех заметок, включая только что созданную.

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:

   ```
   {
       "author": "New author",
       "title": "New title",
       "content": "New content"
   }
   ```

   В ответ вы должны получить обновленную заметку.

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.

Заключение

Ваш путь по полям баз данных и фреймворка Spring Data подходит к концу. Сквозь путь, наполненный кодом, конфигурациями и мощными инструментами, мы прошлись вместе, и, надеюсь, он был по-настоящему увлекательным для вас.

Каждый из этапов урока был важен для общего понимания того, как работать с базами данных в серверных приложениях на Spring. Все это в совокупности дает нам мощный инструментарий для создания надежных, масштабируемых и эффективных приложений.

Но на этом наше обучение не заканчивается! Всегда помните, что самое главное в обучении - это практика. Попробуйте применить полученные знания на практике, создайте свое собственное приложение, экспериментируйте, исследуйте и продолжайте обучение.

Итак, мы закончили наш путь по Spring Data и пришло время перейти к следующему уровню нашего обучения - “Проектирование и реализация API для серверного приложения”. Эта тема - это следующий важный шаг на пути становления профессиональным разработчиком на Spring.

Проектирование и реализация API - это то, что позволяет вашему серверному приложению взаимодействовать с внешним миром. Без хорошо спроектированного API ваше замечательное приложение, наполненное великолепно работающими функциями и взаимодействующее с базой данных, может оказаться недоступным для внешних клиентов.

На следующем уроке мы изучим, как проектировать и создавать эффективные и простые в использовании API с помощью Spring. Мы углубимся в такие темы, как RESTful API, HTTP статус-коды, тестирование API и многое другое.

Уроки, которые мы прошли, и уроки, которые нас ждут, важны не по отдельности, а вместе. Они образуют единое целое, предоставляя вам все необходимые знания и

навыки для создания высококачественных серверных приложений на Spring. Поэтому продолжайте обучение, не упускайте деталей и всегда стремитесь к практическому применению полученных знаний. Это позволит вам стать действительно компетентным и уверенным разработчиком на Spring.

Впереди нас ждут новые открытия и интересные темы. Удачи вам в этом пути!

Что можно почитать еще?

1. Изучение Spring Boot 2.0. Грег Тернквист
2. Освоение Spring Boot 2.0. Динеш Раджпут

Используемая литература

1. Spring Boot в действии. Крейг Уоллс.
2. Spring Microservices в действии. Джон Карнелл
3. Cloud Native Java. Джош Лонг и Кенни Бастани