

Транзакции. Spring Boot



ПРЕПОДАВАТЕЛЬ



Юрий Костяной

Java/Kotlin backend-разработчик

- 3+ года опыта в коммерческой разработке
- 2+ года опыта в преподавании
- Проекты по интеграции сторонних платформ, CRM
- Проблемно-ориентированный подход в преподавании



ВАЖНО:

- Камера должна быть включена на протяжении всего занятия.
- Если у Вас возник вопрос в процессе занятия, пожалуйста, поднимите руку и дождитесь, пока преподаватель закончит мысль и спросит Вас, также можно задать вопрос в чате или когда преподаватель скажет, что начался блок вопросов.
- Организационные вопросы по обучению решаются с кураторами, а не на тематических занятиях.
- Вести себя уважительно и этично по отношению к остальным участникам занятия.
- Во время занятия будут интерактивные задания, будьте готовы включить камеру или демонстрацию экрана по просьбе преподавателя.

ПЛАН ЗАНЯТИЯ

1. Повторение
2. Основной блок
3. Вопросы по основному блоку
4. Домашняя работа

1

ОСНОВНОЙ БЛОК

Введение

- В трансе
- Ботинком с полпинка



Проблема

Ранее мы уже упоминали транзакции. Рассмотрим их получше.

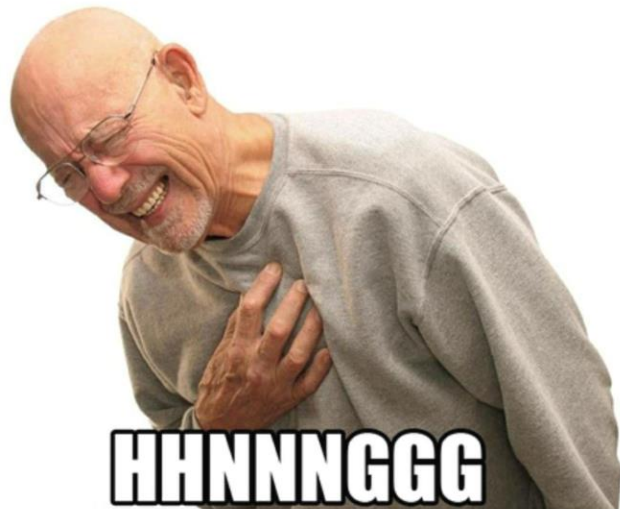
Представьте, что мы создали банковское приложение. Клиент Джим переводит своему другу Майку 50 евро. Для выполнения этой транзакции нам потребуется три действия:

- Списать деньги с баланса Джима;
- Записать операцию перевода от Джима к Майку;
- Добавить денег на баланс Майка.

Допустим, мы успешно списали деньги с баланса Джима, но потом произошла ошибка. В ходе транзакции у Джима списались деньги, но и до Майка деньги не дошли. Как раз такие проблемы решают транзакции.

Как настроить транзакции в нашем приложении?

When all those pending transactions hit and you see your avalabe balance..



В трансе

Транзакция

Транзакцией называется набор связанных операций, все из которых должны быть выполнены корректно без ошибок. Если при выполнении одной из операций возникла ошибка, все остальные должны быть отменены. Такой механизм распространён при работе с БД. Транзакция позволит нам объединить определенные операции таким

образом, что по итогу мы либо запишем все изменения, либо ничего. То есть объединить несколько различных действий в атомарную операцию.

Trans Action



Transaction



В трансе

Управление транзакциями

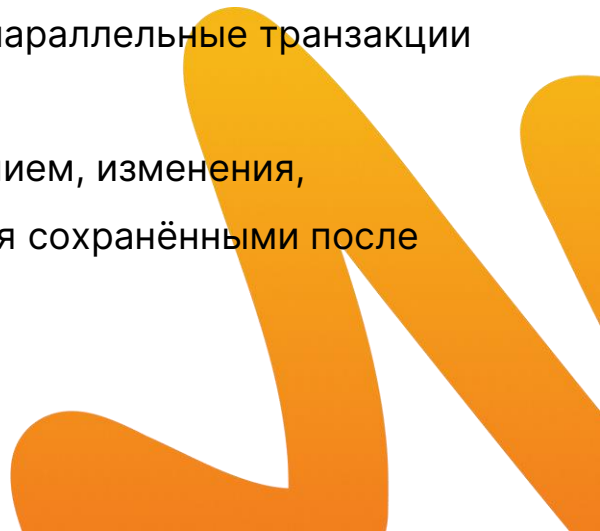
Транзакции обладают свойствами, которые называют **ACID**:

Атомарность (*Atomicity*) гарантирует, что никакая транзакция не будет зафиксирована в системе частично. Будут либо выполнены все операции, либо ни одной.

Согласованность (*Consistency*). Выполненная транзакция, сохраняет согласованность базы данных.

Изолированность (*Isolation*). Во время выполнения транзакции параллельные транзакции не должны оказывать влияние на её результат.

Устойчивость (*Durability*). Независимо от проблем с оборудованием, изменения, сделанные успешно завершённой транзакцией, должны остаться сохранёнными после возвращения системы в работу.



В трансе

Управление транзакциями

Транзакции могут закончиться позитивно, т.е. командой *commit*, или откатом выполненных действий, т.е. командой *rollback*. При откате все команды, выполненные с начала транзакции будут отменены. Транзакция – одноразовая сущность. Вызов *commit* или *rollback* возможен только при открытой транзакции.

Ранее мы уже говорили о том, что *EntityManager* делегирует управление транзакциями контейнеру бинов (*EJB*-контейнер или *Spring*-контекст) при использовании чистого *JPA*. В свою очередь *Hibernate* может управлять транзакциями на уровне объектов *Session*.



В трансе

Управление транзакциями

Для начала работы с транзакциями в Spring-приложении нужно выполнить ряд действий:

1 Указать в конфигурации аннотацию **@EnableTransactionManagement**.

2 Настроить бин **PlatformTransactionManager**. Пример для JPA:

```
@Configuration
```

```
...
```

```
@EnableTransactionManagement // включает работу транзакций
```

```
@EnableJpaRepositories(basePackages = "org.example.repository")
```

```
public class PersistenceConfig {
```

```
...
```

```
@Bean // Бин для автоматического управления транзакциями
```

```
public PlatformTransactionManager transactionManager(DataSource dataSource) {
```

```
    JpaTransactionManager transactionManager = new JpaTransactionManager();
```

```
    transactionManager.setEntityManagerFactory(entityManagerFactory(dataSource).getObject());
```

```
    return transactionManager;
```

```
}
```


```
}
```

В трансе

Управление транзакциями

Пример для Hibernate:

```
@Bean // Bean для управления транзакциями Hibernate
public HibernateTransactionManager transactionManager(SessionFactory sessionFactory) {
    HibernateTransactionManager transactionManager = new HibernateTransactionManager();
    transactionManager.setSessionFactory(sessionFactory);
    return transactionManager;
}
```



В трансе

Управление транзакциями

3.1 При использовании *DAO* с *EntityManager* можно управлять транзакциями вручную.

3.2 В Spring-приложении для управления транзакцией достаточно указать **@Transactional** (*org.springframework.transaction.annotation.Transactional*) над методом *DAO*, чтобы в рамках вызова метода была создана начата транзакция.

```
@Transactional
public void delete(long id) {
    Optional.ofNullable(read(id))
        .ifPresent(u -> entityManager.remove(u));
}
```

```
@PersistenceContext
private EntityManager entityManager;

public void addUser(User user) {
    entityManager.getTransaction().begin();
    entityManager.persist(user);
    entityManager.getTransaction().commit();
}
```



В трансе

Управление транзакциями

4.1 *@Transactional* помечаются все методы, изменяющие данные в БД (*create, update, delete*). Для методов чтения тоже создаются транзакции, но они являются *read-only*, а такие транзакции значительно безопасней. *@Transactional(readOnly = true)* можно указать, например, над методом репозитория, который выполняет SELECT-запрос с помощью аннотации [@Query](#).

```
@Repository
public interface PostRepository extends JpaRepository<Post, Long> {
    @Transactional(readOnly = true)
    @Query("select p from Post p where p.user.id=:id and p.title like :title")
    List<Post> findMySuperPosts(String title, long id);
}
```

4.2 Если метод чтения данных кроме основной части сущности загружает её ленивую часть (например, с помощью вызова *Hibernate.initialize()*) или делает несколько запросов на чтение, то его нужно помечать *@Transactional*.



В трансе

Управление транзакциями

5 Важный момент, что если метод из *DAO* помечен *@Transactional*, то и метод в сервисе, вызывающий этот транзакционный метод *DAO* должен быть помечен *@Transactional*.



В трансе

Настройки @Transactional

1 Веб-приложение создает отдельный поток (*thread*) для каждого REST-запроса, таким образом обращение к методу, который помечен *@Transactional* может происходить параллельно из нескольких потоков. При этом одна транзакция будет ожидать завершения *create/update/delete*-метода другой транзакции в зависимости от настройки **isolation**.

- *Isolation.READ_COMMITTED* – используется по умолчанию. *Update*-запрос блокирует модифицируемую строку от параллельных *update*-запросов до конца транзакции. От параллельных *select* не блокирует. Все происходит последовательно.
- *Isolation.REPEATABLE_READ* – требует повторяемого чтения, то есть теперь нашу транзакцию не устроит тот случай, когда *update* пересчитал значение и обнаружил, что оно не такое, как в начале транзакции (оптимистичная блокировка). В этом случае будет откат и исключение.

В трансе

Настройки @Transactional

- для синхронизации всех параллельных запросов (update и select) можно отказаться от оптимистичной блокировки и использовать явную блокировку *select .. for update*
Она блокирует как параллельные *select for update*, так и *update*, то есть заставляет их ждать до конца своей транзакции.

```
@Transactional(propagation = Propagation.REQUIRED)
@Repository
public interface HitRepository extends CrudRepository<Hits, Long> {
    @Query("select id, count from hits where id=:id for update")
    Hits getCount(long id);
    @Modifying // указывается совместно с @Transactional для запросов, изменяющих данные
    @Query("update hits set count=count+1 where id=:id")
    void updateCount(long id);
}
```

Синтаксис select
for update

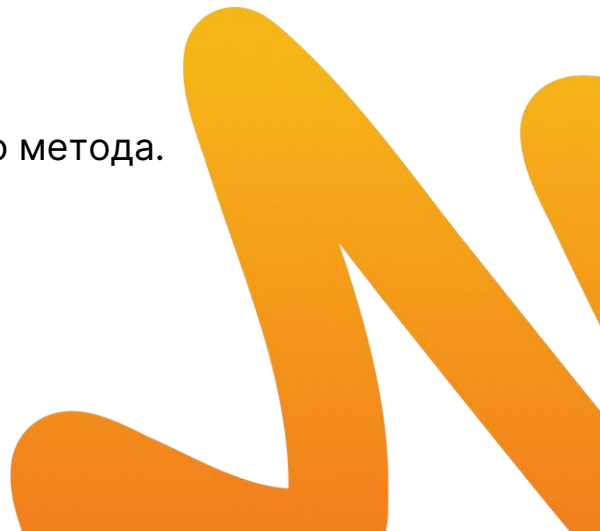
В трансе

Настройки @Transactional

2 Аннотация *@Transactional* (без настроек) заставляет метод выполняться в рамках транзакции. За это отвечает параметр **propagation**, имеющий по умолчанию значение *REQUIRED*, т.е.

- Если метод вызывается вне транзакции, для него создается отдельная транзакция.
- Если метод вызывается из метода сервиса, в котором уже есть транзакция, то метод вызывается в рамках этой транзакции.

REQUIRED_NEW создаёт отдельную аннотацию для вызываемого метода.



В трансе

Настройки @Transactional

SUPPORTS использует транзакцию во внешнем методе, если она есть. Но если нет, своя транзакция для внутреннего метода создаваться не будет. А без транзакции все команды внутреннего метода будут выполнены в режиме автофиксации (*AUTOCOMMIT*): каждая команда автоматически подтверждается (как бы обрамляется своей отдельной транзакцией - commit-ом, происходит это на уровне базы данных). То есть если в методе выполняется несколько SQL-операторов *update*, то часть из них может выполняться, а часть - нет.

NOT_SUPPORTED - режим, при котором всегда выполняется *AUTOCOMMIT*.

NEVER не терпит транзакции снаружи и выбрасывает исключение, если транзакция обнаружена.

MANDATORY требует внешнюю транзакцию, а иначе выбрасывается исключение.

В транс

Пример настройки транзакций



SpringRelationsExample.zip



Задание

Доработайте приложение о меню так, чтобы заработали транзакции.

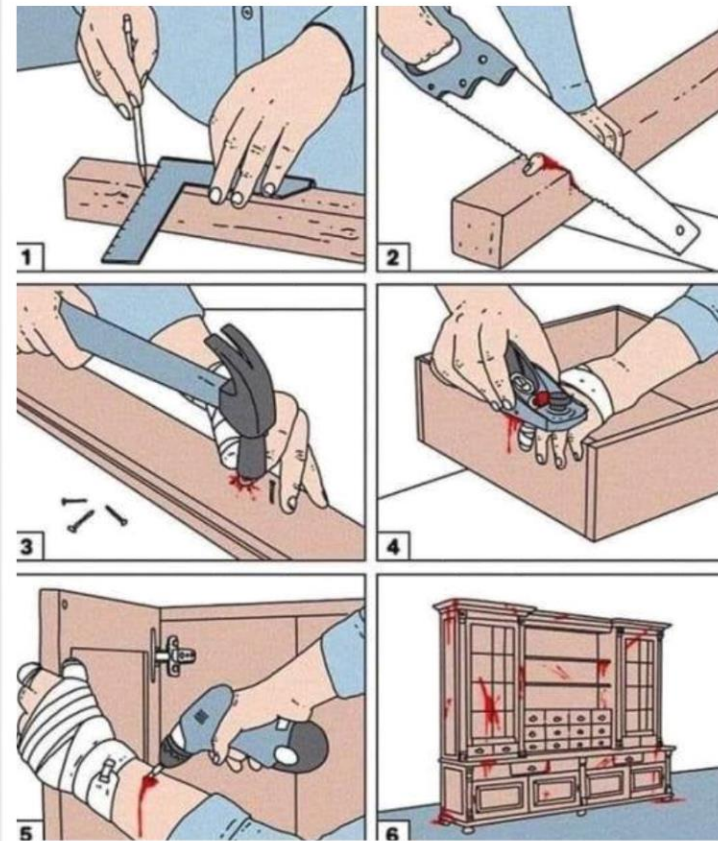


Проблема

Spring Framework – это зонтичный продукт, т.е. он содержит множество технологий и направлений в своём составе.

Из-за большого объёма зависимостей, настроек и сложных конфигураций каждого из направлений при сборке *Spring*-приложения легко можно сделать ошибку или что-то забыть настроить. Непрозрачность *Spring'a* этому способствует.

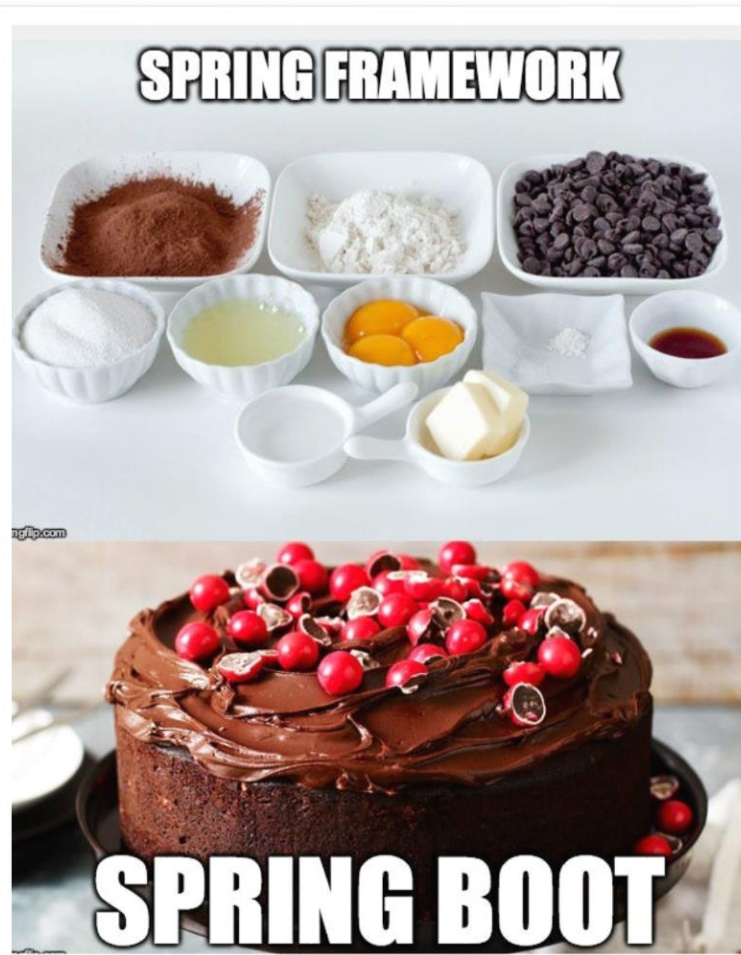
*Каким образом можно упростить создание и работу проекта Spring, чтобы всё работало «из коробки» (принцип **just run**)?*



Ботинком с полпинка

Spring Boot

Spring Boot – это технология в составе *Spring Framework*, которая упрощает настройку фреймворка под распространённые случаи создания web-приложений.



Ботинком с полпинка

Преимущества Spring Boot

1 Нет конфликта версий разных частей *Spring*.

2 Встроенные и настроенные Web-серверы (*Tomcat*, *Jetty* или *Undertow*). Не надо деплоить экземпляр сервера каждый раз.

3 Вместо множество зависимостей достаточно подключить несколько стартеров (starter).

4 Автоматическая настройка Spring-приложения для работы со сторонними библиотеками и технологиями (например, *Jackson*, *Hibernate* и др.)

5 Предоставляет возможность получать метрики приложения, его состояние, выполнять внешнюю настройку.

6 Не требует написания xml-конфигурации или конфигурации в Java-коде.

WHAT IF I TOLD YOU



You can use Spring
without any configuration

Ботинком с полпинка

Шаги перехода на Spring Boot

1 Заменить зависимости в *pom.xml*: удалить все существующие spring-зависимости и добавить вместо них стартеры ([spring-boot-starters](#)).

2 Указать в *pom.xml* spring-boot-starter-parent в качестве родительского POM для управления версиями зависимостей и конфигурацией по умолчанию.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.2.3</version>
</parent>
```



Ботинком с полпинка

Шаги перехода на Spring Boot

3 Создать точку входа приложения

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

@SpringBootApplication добавляет следующие аннотации:

@Configuration – помечает класс как источник определений *bean*-компонентов.

@EnableAutoConfiguration – указывает платформе автоматически добавлять *bean*-компоненты на основе зависимостей пути к классам.

@ComponentScan – сканирует другие конфигурации и *bean*-компоненты в том же пакете, что и класс *Application*, или ниже.



Ботинком с полпинка

Шаги перехода на Spring Boot

Для исключения классов или пакетов из области сканирования бинов используют фильтры:

```
@SpringBootApplication
@ComponentScan(excludeFilters = {
    @ComponentScan.Filter(type = FilterType.REGEX,
        pattern = "com.foreach.config.*"))
public class Application { //... }
```

Чтобы явно добавить классы или пакеты можно использовать *@ComponentScan* или **@Import** для основного класса:

```
@SpringBootApplication
@ComponentScan(basePackages="com.foreach.config")
@Import(UserRepository.class)
public class Application { //... }
```

Для импорта xml-кофигурации используется **@ImportResource**

```
@SpringBootApplication
@ImportResource("applicationContext.xml")
public class Application { //... }
```

БОТИНКОМ С ПОЛПИНКА

Шаги перехода на Spring Boot

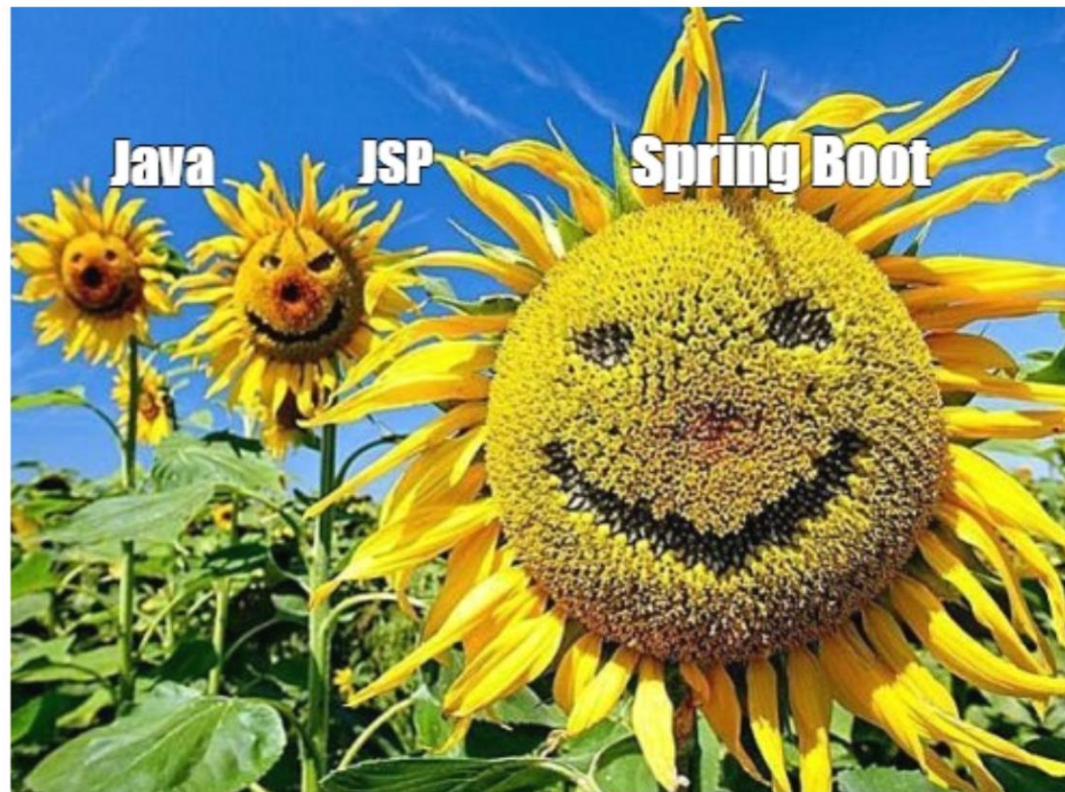
4 Сделать загрузку ресурсов приложения. *Spring Boot* ищет файлы ресурсов в следующих каталогах:

- */resources*
- */public*
- */static*
- */META-INF/resources*

Достаточно перенести все ресурсы в один из этих каталогов.

Другой вариант – добавить путь к ресурсом в файле *application.yml* или *application.properties* :

```
spring.resources.static-locations=  
classpath:/images/,classpath:/jsp/
```



Ботинком с полпинка

Шаги перехода на Spring Boot

5 Фреймворк автоматически загрузит любые свойства, определенные в файлах с именами *application.properties* или *application.yml*, размещенных в одном из следующих мест:

- *подкаталог /config текущей директории*
- *текущая директория*
- *подкаталог /config в classpath*
- *корневой каталог classpath root.*

При необходимости мы можем создавать разные файлы настроек для разных профилей запуска приложения (для отладки, для тестирования, для стендов, для прода). Для этого используются property-файлы с разными именами по схеме:

application-{profile}.properties

Существует множество predefined настроек для тонкой настройки Spring Boot приложения.



БОТИНКОМ С ПОЛПИНКА

Шаги перехода на Spring Boot

6 Удалить класс-Initializer, который настраивает *DispatcherServlet*. В нём более нет необходимости, как и в аннотации *@EnableWebMvc*.

Подключение *spring-boot-starter-web* добавляет

- поддержку загрузки статического контента из каталогов с именем */static*, */public*, */resources* или */META-INF/resources* в пути к классам.
- Компоненты *HttpMessageConverter* для распространенных случаев использования (JSON и XML).
- Маппинг исключений в ответы с кодами ошибок.

7 Если был добавлен стартер шаблонизатора. Например, *spring-boot-starter-thymeleaf*, то шаблоны следует перенести в каталог */resources/templates*.

8 Зависимость *spring-boot-starter-web* добавляет также зависимость *spring-boot-starter-tomcat*, которая добавляет в приложение встроенный web-сервер. Вместо *Tomcat* может быть использован *Jetty* или *Undertow*.

Ботинком с полпинка

Шаги перехода на Spring Boot

Вы уже могли заметить, что работа *Spring Boot* сильно зависит от именования и расположения файлов и каталогов. Чтобы *Spring Boot* приложение «поднялось» и автоматизация заработала приходится всё разложить по полочкам.



Ботинком с полпинка

Шаги перехода на Spring Boot

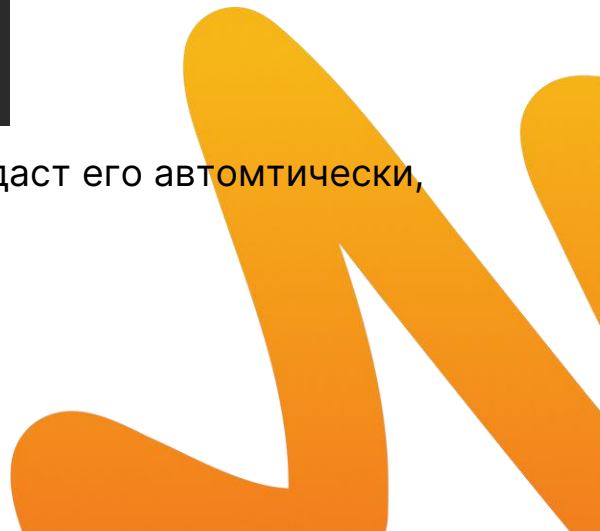
9 В зависимости от используемого подхода работы с БД (JDBC, JPA, noSQL и т.д.) нужно

добавить подходящий стартер. Например, для JPA: *spring-boot-starter-data-jpa*

Для начала работы с in-memory БД (*H2, Derby, and HSQLDB*) достаточно добавить зависимость этой БД. Например,

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
</dependency>
```

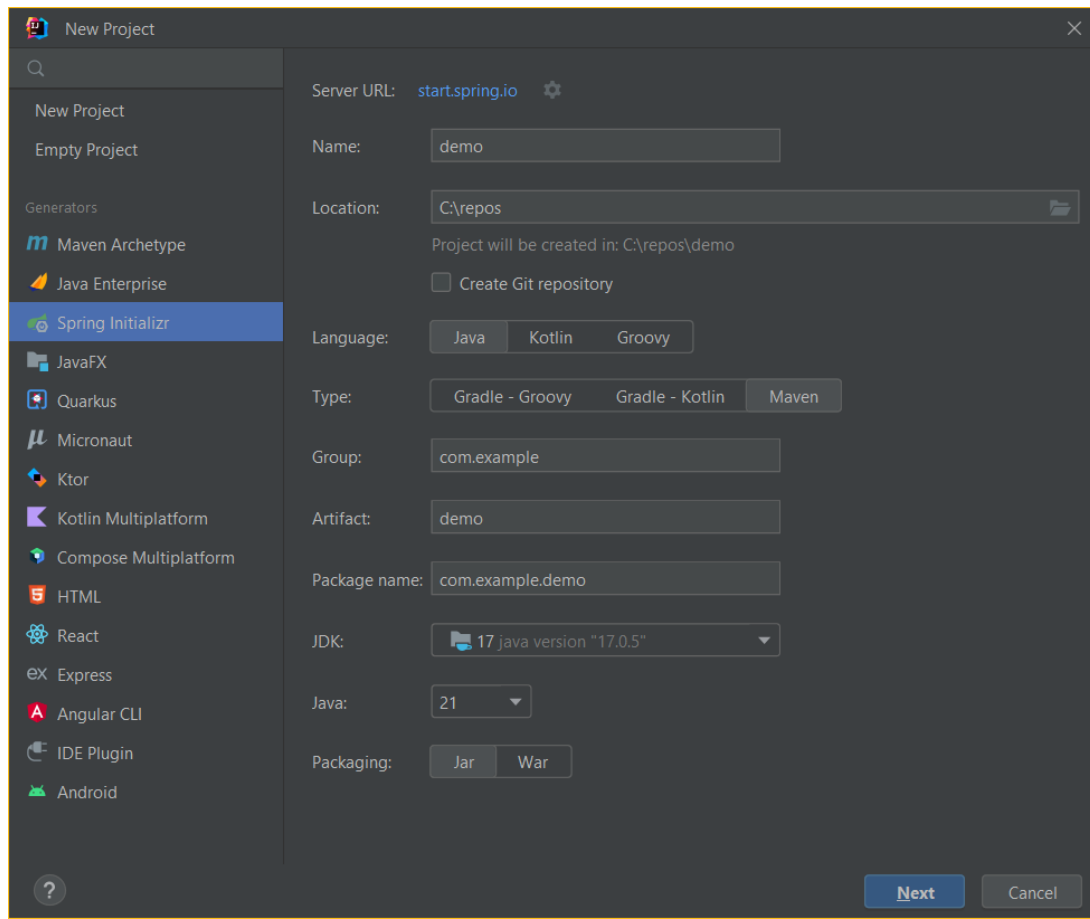
Для обычных БД нужно создать бин *DataSource*. *Spring Boot* создаст его автоматически, если в настройках указать нужные поля:



Ботинком с полпинка Spring initializr

Есть онлайн-сервис, который позволяет через web-интерфейс настроить будущий проект и получить архив с подготовленным проектом, в котором будут указаны нужные стартеры и другие настройки: <https://start.spring.io/>

IntelliJ IDEA Ultimate тоже поддерживает взаимодействие со *Spring initializr*.



В транс

Пример Spring Boot приложения



SpringBootExample.zip



Задание

Переведите приложение о меню для использования Spring Boot.



2

Домашнее задание

Домашнее задание

Создайте Spring Boot приложение с RESTful API для бронирования номеров в отеле (Hotel Booking Application), которое позволяет пользователям просматривать доступные номера, бронировать их на определенные даты, а также отменять бронирование.

Дополнительно (по желанию) можно реализовать систему отзывов о номерах и отелях, а также различные способы фильтрации номеров (по цене, количеству звезд, расположению и т.д.).



Полезные ссылки

- Spring позволяет управлять http/WebSocket/webFlux-сессиями (открытое соединение, при котором пользователю не требуется проходить авторизацию для каждого следующего запроса). Для того, чтобы сессия не терялась между запусками программы, её тоже можно хранить в БД. К сожалению, управление http-сессиями не входит в данный курс. Почитать об этом подробнее можно здесь

<https://docs.spring.io/spring-session/docs/2.2.x/reference/html/httpsession.html>

<https://habr.com/ru/articles/513668/>

- Подробно про использование транзакций

<https://habr.com/ru/companies/rosbank/articles/707378/>

- Вопросы про JPA на собеседовании

<https://habr.com/ru/articles/265061/>

ЗАКЛЮЧЕНИЕ

