

Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Институт космических и информационных технологий
Кафедра «Информатика»

Практический пример №4
Работа с базами данных в Spring Framework

Составил: Старший преподаватель кафедры «Информатика»
Черниговский Алексей Сергеевич

Красноярск 2020

Содержание

1 Создание проекта.....	3
2 Добавление зависимостей.....	3
3 Добавление файлов исходного кода.....	3
4 Создание модели данных.....	3
5 Создание базы данных.....	4
6 Конфигурация Spring приложения для работы с JdbcTemplate.....	4
6.1 DataSource.....	4
6.2 DAO.....	4
6.3 Запуск.....	5
7 Конфигурация Spring приложения для работы с JPA.....	5

1 Создание проекта

Создаем Maven проект из архетипа webapp. Artifact Id в данном приложении будет – jdbctest.

2 Добавление зависимостей

Добавляем зависимости в pom.xml: Spring JDBC, Spring Data JPA. Я буду использовать базу данных PostgreSQL, поэтому подключаю PostgreSQL JDBC Driver.

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.2.8.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-jpa</artifactId>
        <version>2.3.3.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.2.15</version>
    </dependency>
</dependencies>
```

3 Добавление файлов исходного кода

В папку main добавляем папку java и resources. В resources помещаем файл application.properties со следующим содержимым:

```
#--- Postgres ---
dataSource.driverClassName =org.postgresql.Driver
jpa.database=POSTGRESQL
dataSource.url=jdbc:postgresql://localhost:5432/postgres

dataSource.username=postgres
dataSource.password=postgres
```

Естественно, если вы используете другую базу или другие данные для подключения (url или username/pass), тогда необходимо написать их.

4 Создание модели данных

Создадим класс Student в пакете entity со следующим содержимым:

```
public class Student {  
    private int id;  
    private String firstName;  
    private String lastName;  
    private String patronymic;  
    private double avgMark;  
}
```

Создаем конструкторы, сеттеры геттеры и метод `toString()`.

5 Создание базы данных

Создадим таблицу в PostgreSQL с именем Student и соответствующими столбцами как в классе Student.

Надеюсь с этой задачей вы справитесь самостоятельно.

6 Конфигурация Spring приложения для работы с JdbcTemplate

Конфигурировать приложение будем через аннотации и Java-код.

Создадим класс SpringConfig.

6.1 DataSource

В конфигурацию поместим следующим код:

```
@Configuration  
@ComponentScan("ru.jdbctest")  
@PropertySource("classpath:application.properties")  
public class SpringConfig {  
  
    @Autowired  
    private Environment env;  
  
    @Bean  
    DataSource dataSource() {  
        DriverManagerDataSource dataSource = new DriverManagerDataSource();  
  
        dataSource.setDriverClassName(env.getProperty("dataSource.driverClassName"));  
        dataSource.setUrl(env.getProperty("dataSource.url"));  
        dataSource.setUsername(env.getProperty("dataSource.username"));  
        dataSource.setPassword(env.getProperty("dataSource.password"));  
        return dataSource;  
    }  
}
```

Дополнительно о DriverManagerDataSource вы можете прочитать в конспекте лекции.

6.2 DAO

Создадим класс StudentJdbcDao

```
@Component  
public class StudentJdbcDao {  
  
    JdbcTemplate jdbcTemplate;
```

```

@Autowired
public void setDataSource(DataSource dataSource){
    this.jdbcTemplate = new JdbcTemplate(dataSource);
}

public List<Student> findAll(){
    List<Student> students = jdbcTemplate.query("select * from student", new
BeanPropertyRowMapper<>(Student.class));

    return students;
}

public int insert(Student student){
    return jdbcTemplate.update("insert into student " + "(id, first_name,
last_name, patronymic, avg_mark) "
        + "values (?, ?, ?, ?, ?)",
        new Object[] {
            student.getId(), student.getFirstName(),
student.getLastName(), student.getPatronymic(), student.getAvgMark()
        });
}

```

6.3 Запуск

Теперь содержимое нашего запускаемого класса будет выглядеть следующим образом:

```

AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(SpringConfig.class);
StudentJdbcDao studentJdbcDao = context.getBean("studentJdbcDao",
StudentJdbcDao.class);
Student vasya = new Student(1, "Petya", "Ivanov", "Ivanovich", 4.5);
studentJdbcDao.insert(vasya);
context.close();

```

Таким образом мы выполнили вставку новой записи в таблицу базы данных при помощи JdbcTemplate.

7 Конфигурация Spring приложения для работы с JPA

Теперь продемонстрируем пример с использованием Spring JPA

Класс Student аннотируем как @Entity.

Поле id аннотируем @Id и @GeneratedValue(генерируется автоматически).

Создадим интерфейс StudentRepository унаследованный от JpaRepository.

```

@Repository
public interface StudentRepository extends JpaRepository<Student, Integer> {
}

```

В конфигурации у нас уже прописан DataSource из предыдущего примера с JDBCTemplate, поэтому используем его и дописываем необходимые для работы с JPA компоненты.

```

@Bean
public EntityManagerFactory entityManagerFactory() {
    HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    vendorAdapter.setGenerateDdl(true);

    LocalContainerEntityManagerFactoryBean factory = new
LocalContainerEntityManagerFactoryBean();
    factory.setJpaVendorAdapter(vendorAdapter);
    factory.setPackagesToScan("entity");
    factory.setDataSource(dataSource());
    factory.afterPropertiesSet();

    return factory.getObject();
}

@Bean
public PlatformTransactionManager transactionManager() {

    JpaTransactionManager txManager = new JpaTransactionManager();
    txManager.setEntityManagerFactory(entityManagerFactory());
    return txManager;
}

```

Продемонстрируем пример запускаемого класса:

```

public class JdbcTestRunner {

    @Autowired
    static StudentRepository studentRepository;

    public static void main(String[] args) {

        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(SpringConfig.class);

        Student st = new Student(3, "Ilya", "Sidorov", "Petrovish", 3);
        ArrayList<Student> students = new ArrayList<Student>();
        students.add(st);

        StudentRepository rep = context.getBean("studentRepository",
StudentRepository.class);
        Optional<Student> s = rep.findById(1);
        List<Student> lst = rep.findByAvgMarkGreater Than(4.0);
        rep.saveAll(students);

    }
}

```

Здесь происходит:

- 1) Создание списка из одного элемента типа Student.
- 2) Получение компонента studentRepository из контекста.
- 3) Получение объекта типа Optional (о нем можно прочитать самостоятельно. Простыми словами — это объект, который представляет результат запроса. Пустой, если элемента с таким id нет, или ссылка на объект, если поиск дал результат.)
- 4) В StudentRepository можем создать метод

```
List<Student> findByAvgMarkGreater Than(double avgMark);
```

Это механизм создания запросов с помощью кода, о нем можно
дополнительно почитать здесь:

<https://docs.spring.io/spring-data/jpa/docs/1.5.0.RELEASE/reference/html/repositories.html#repositories.query-methods>

В коде вызываем данный метод, он возвращает все элементы, которые
имеют значение поля avgMark > 4,0.

5) В конце сохраняем список объектов в базу данных.

Дополнительно о Spring JPA можно прочитать по следующей ссылке:

<https://docs.spring.io/spring-data/jpa/docs/1.5.0.RELEASE/reference/html/jpa.repositories.html>