7616

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

РЯЗАНСКИЙ ГОСУДАРСТВЕННЫЙ РАДИОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
им. В.Ф. УТКИНА

ОСНОВЫ ПРОМЫШЛЕННОЙ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Часть 3

Методические указания
к лабораторным и практическим занятиям

Рязань 2023

УДК 004.43

Основы промышленной разработки программного обеспечения. Часть 3: методические указания к лабораторным и практическим занятиям / Рязан. гос. радиотехн. ун-т; сост.: Б.В. Костров, А.С. Бастрычкин, Е.А. Трушина. Рязань, 2023. 32 с.

Содержат методические материалы для подготовки к выполнению лабораторных и практических работ.

Предназначены для бакалавров, обучающихся по направлениям: 02.03.03 «Математическое обеспечение и администрирование информационных систем», 09.03.01 «Информатика и вычислительная техника» и 38.03.05 «Бизнес-информатика», а также для специалистов, обучающихся по направлению 27.05.01 «Специальные организационно-технические системы»

Ил. 2. Библиогр.: 4 назв.

Язык программирования Java, объектно-ориентированное программирование, программное обеспечение, промышленная разработка

Печатается по решению редакционно-издательского совета Рязанского государственного радиотехнического университета.

Рецензент: кафедра электронных вычислительных машин Рязанского государственного радиотехнического университета (зав. кафедрой д-р техн. наук, проф. Б.В. Костров)

Занятие № 11
Spring Framework

Цель работы: изучение Spring Framework, приобретение навыков использования внедрения зависимостей и Web MVC при работе с базами данных в Java-приложениях.

1. Теоретическая часть

Spring Framework представляет собой контейнер внедрения зависимостей с несколькими удобными слоями (например, доступ к базе данных, прокси, аспектно-ориентированное программирование, RPC, веб-инфраструктура MVC), что позволяет быстрее и удобнее создавать Java-приложения.

1.1. Основы внедрения зависимостей

Допустим, необходимо написать Java-класс, который позволяет получить доступ к таблице пользователей в базе данных, то есть класс DAO (объект доступа к данным) или репозиторий.

Рассмотрим класс UserDAO:

```
public class UserDao {

    public User findById(Integer id) {
        // execute a sql query to find the user
    }
}
```

Класс UserDAO имеет только один метод, который позволяет находить пользователей в таблице базы данных по их идентификаторам.

Чтобы выполнить соответствующий SQL запрос, классу UserDAO требуется соединение с базой данных. Получить соединение с базой данных можно из другого класса, называемого DataSource.

Дополним код:

```
import javax.sql.DataSource;

public class UserDao {

 public User findById(Integer id) throws SQLException {
  try (Connection connection = dataSource
                                  .getConnection()) { //(1)

   PreparedStatement selectStatement = connection
     .prepareStatement("select * from users where id = ?");

               // use the connection etc.
  }
 }
}
```

UserDao зависит от действительного источника данных для запуска SQL-запросов.

1.1.1. Внедрение зависимостей с помощью new()

Для подключения к базе данных MySQL UserDAO может выглядеть следующим образом:

```
import com.mysql.cj.jdbc.MysqlDataSource;

public class UserDao {

 public User findById(Integer id) {
  MysqlDataSource dataSource = new MysqlDataSource(); // (1)

  dataSource
    .setURL("jdbc:mysql://localhost:3306/myDatabase");
  dataSource.setUser("root");
  dataSource.setPassword("s3cr3t");


  try (Connection connection = dataSource
                               .getConnection()) { // (2)

   PreparedStatement selectStatement = connection
     .prepareStatement("select * from users where id =  ?");

   // execute the statement..
   // convert the raw jdbc resultset   to a user

   return user;
  }
 }
}
```

Для подключения к базе данных MySQL используется MysqlDataSource. Для облегчения чтения кода непосредственно в коде заданы url/username/password. Для запроса используется недавно созданный источник данных.

Это работает, но можно использовать другой подход, расширив класс UserDao другим методом − findByFirstName. Этому методу также нужен источник данных для работы. Добавим этот метод к UserDAO и применим некоторые рефакторинги, введя метод newDataSource:

```
import com.mysql.cj.jdbc.MysqlDataSource;

public class UserDao {

 public User findById(Integer id) {
  try (Connection connection = newDataSource()
                                   .getConnection()){ // (1)
   PreparedStatement selectStatement = connection
     .prepareStatement("select * from users where id =  ?");

   // TODO execute the select ,
```

```
    //handle exceptions,
    //return the user
   }
  }
  public User findByFirstName(String firstName) {
   try (Connection connection = newDataSource()
                                .getConnection()) { // (2)

    PreparedStatement selectStatement =connection
      .prepareStatement(
       "select * from users where first_name =  ?"
     );

     // TODO execute the select,
     // handle exceptions,
     // return the user
    }
  }
  public DataSource newDataSource() {
   MysqlDataSource dataSource = new MysqlDataSource(); // (3)
   dataSource.setUser("root");
   dataSource.setPassword("s3cr3t");
   dataSource
     .setURL("jdbc:mysql://localhost:3306/myDatabase");

   return dataSource;
  }
}
```

Для использования нового метода newDataSource() был переписан findById и добавлен findByFirstName. Недавно извлеченный метод способен создавать новые источники данных.

Этот подход работает, но имеет два недостатка.

1. Допустим, необходимо создать новый класс ProductDAO, который выполняют операторы SQL. ProductDAO также будет иметь зависимость DataSource, которая доступна только в классе UserDAO. Затем будет другой подобный метод или же будет извлечен вспомогательный класс, содержащий DataSource.

2. Допустим, нужно создать совершенно новый источник данных для каждого запроса SQL. Необходимо учитывать, что DataSource открывает реальное сокет-соединение от Java-программы к базе данных. Это довольно дорого и занимает много времени. Было бы намного лучше открыть только один источник данных и использовать его повторно. Одним из способов сделать это может быть сохранение источника данных в закрытом поле в UserDao, чтобы его можно было повторно использовать между методами, но это не помогает при дублировании между несколькими DAO.

1.1.2. Зависимости в глобальном классе приложения

Чтобы решить перечисленные проблемы, можно написать глобальный класс Application, который выглядит следующим образом:

```
import com.mysql.cj.jdbc.MysqlDataSource;

public enum Application {

 INSTANCE;

 private DataSource dataSource;

 public DataSource dataSource() {
  if (dataSource == null) {
   MysqlDataSource dataSource = new MysqlDataSource();
   dataSource.setUser("root");
   dataSource.setPassword("s3cr3t");

   dataSource
     .setURL("jdbc:mysql://localhost:3306/myDatabase");
   this.dataSource = dataSource;
  }
  return dataSource;
 }
}
```

Класс UserDAO теперь может выглядеть так:

```
import com.yourpackage.Application;

public class UserDao {
 public User findById(Integer id) {
  try (Connection connection = Application.INSTANCE
                      .dataSource().getConnection()) { // (1)
   PreparedStatement selectStatement = connection
     .prepareStatement("select * from users where id =  ?");
   // TODO execute the select etc.
  }
 }

 public User findByFirstName(String firstName) {
  try (Connection connection = Application.INSTANCE
                      .dataSource().getConnection()) { // (2)
   PreparedStatement selectStatement = connection
     .prepareStatement(
       "select * from users where first_name = ?"
     );
     // TODO execute the select etc.
  }
 }
}
```

Это способствует следующим улучшениям:

1) UserDAO и другим DAO больше не нужно создавать собственную зависимость DataSource, вместо этого будет использоваться полнофункциональная, предоставленная классом Application;

2) класс приложения является одноэлементным (будет создан только один INSTANCE), и этот одноэлементный компонент приложения содержит ссылку на одноэлементный объект DataSource.

Однако у этого решения есть еще несколько недостатков.

1. UserDAO должен знать, где получить зависимости, должен быть вызван класс приложения Application.INSTANCE.dataSource().

2. Если программа становится больше, должно быть получено большое количество зависимостей, тогда все эти зависимости будет обрабатывать один большой класс Application.java. Может понадобиться разделение на несколько классов и т.д.

1.1.3. Инверсия управления (IoC, Inversion of Control)

Инверсия управления (Inversion of Control) возникает в том случае, когда не нужно беспокоиться о поиске зависимостей, т.е. класс UserDAO вместо того, чтобы активно вызывать Application.INSTANCE.dataSource(), больше не контролирует, когда, как и откуда он его получает.

Класс UserDAO с новым конструктором будет выглядеть так:

```
import javax.sql.DataSource;
public class UserDao {
 private DataSource dataSource;
 private UserDao(DataSource dataSource) { // (1)
  this.dataSource = dataSource;
 }
 public User findById(Integer id) {
  try (Connection connection = dataSource
                               .getConnection()) { // (2)
   PreparedStatement selectStatement = connection
     .prepareStatement("select * from users where id =  ?");

   // TODO execute the select etc.
  }
 }
 public User findByFirstName(String firstName) {
  try (Connection connection = dataSource
                               .getConnection()) { // (2)
   PreparedStatement selectStatement = connection
     .prepareStatement(
       "select * from users where first_name = ?"
     );
   // TODO execute the select etc.
  }
 }
}
```

1) при создании нового UserDao через конструктор будет передаваться действительный источник данных;

2) этот источник данных будет использоваться методами findByX.

Соответственно, если необходимо создать (т.е. использовать) UserDao, нужно дать ему источник данных.

Но если необходимо запустить приложение, то теперь нужно обязательно вызвать новый UserDao(dataSource), тогда как раньше можно было вызывать «new UserService()»:

```
public class MyApplication {
 public static void main(String[] args) {
  UserDao userDao = new UserDao(Application.INSTANCE
                    .dataSource());

  User user1 = userDao.findById(1);
  User user2 = userDao.findById(2);
  // etc ...
 }
}
```

1.1.4. Контейнеры для внедрения зависимостей

Следовательно, проблема в том, что UserDAO создается с помощью конструктора и, таким образом, зависимость DataSource устанавливается вручную.

Для того, чтобы сконструировать оба работающих объекта (DataSource и зависимый от него UserDao), можно использовать контейнер внедрения зависимостей, который является именно тем, что представляет собой среда Spring.

1.2. Контейнер Spring IOC / Dependency Injection

Spring Framework (контейнер Spring IOC) по своей сути является контейнером внедрения зависимостей (Dependency Injection), который управляет написанными классами и их зависимостями.

1.2.1. ApplicationContext

ApplicationContext способен контролировать все классы и управлять ими соответствующим образом (создавать их с необходимыми зависимостями).

Применение ApplicationContext выглядит следующим образом:

```
import org.springframework.context.ApplicationContext;
import org.springframework.context
  .annotation.AnnotationConfigApplicationContext;
import javax.sql.DataSource;

public class MyApplication {

 public static void main(String[] args) {
  ApplicationContext ctx = new
                           AnnotationConfigApplicationContext(
                            someConfigClass); // (1)

  UserDao userDao = ctx.getBean(UserDao.class); // (2)
```

```
  User user1 = userDao.findById(1);
  User user2 = userDao.findById(2);
  DataSource dataSource = ctx.getBean(DataSource.class);//(3)
  // etc ...
 }
}
```

1) создан Spring ApplicationContext;

2) ApplicationContext может дать полностью сконфигурированный UserDao (с его набором зависимостей DataSource);

3) ApplicationContext может напрямую предоставить источник данных, являющийся тем же источником данных, который он устанавливает внутри UserDao.

1.2.2. ApplicationContextConfiguration

В приведенном выше коде переменная с именем someConfigClass помещается в конструктор AnnotationConfigApplicationContext:

```
import
org.springframework.context.annotation.AnnotationConfigApplica
tionContext;
public class MyApplication {
 public static void main(String[] args) {
  ApplicationContext ctx= new
                          AnnotationConfigApplicationContext(
                           someConfigClass); // (1)
  // ...
 }
}
```

В конструктор ApplicationContext должна быть передана ссылка на класс конфигурации, который должен выглядеть следующим образом:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class MyApplicationContextConfiguration {  // (1)
 @Bean
 public DataSource dataSource() {  // (2)
  MysqlDataSource dataSource = new MysqlDataSource();
  dataSource.setUser("root");
  dataSource.setPassword("s3cr3t");
  dataSource.setURL(
    "jdbc:mysql://localhost:3306/myDatabase");
  return dataSource;
 }
 @Bean
 public UserDao userDao() { // (3)
  return new UserDao(dataSource());
 }
}
```

1) существует выделенный класс конфигурации ApplicationContext, помеченный аннотацией @Configuration, который немного похож на класс Application.java из раздела 1.1.2 «Зависимости в глобальном классе приложения»;

2) есть метод, который возвращает DataSource и аннотируется @Bean для Spring;

3) есть другой метод, который возвращает UserDao и создает указанный UserDao, вызывая метод bean-компонента dataSource.

Этого класса конфигурации уже достаточно для запуска самого первого приложения Spring:

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplica
tionContext;

public class MyApplication {

 public static void main(String[] args) {
  ApplicationContext ctx = new
                          AnnotationConfigApplicationContext(
                     MyApplicationContextConfiguration.class);
  UserDao userDao = ctx.getBean(UserDao.class);
  // User user1 = userDao.findById(1);
  // User user2 = userDao.findById(1);
  DataSource dataSource = ctx.getBean(DataSource.class);
 }
}
```

1.2.3. AnnotationConfigApplicationContext и другие классы ApplicationContext

Существует много способов создания Spring ApplicationContext, например с помощью файлов XML, аннотированных классов конфигурации Java или даже программно. Для внешнего мира это представлено через единый интерфейс ApplicationContext.

MyApplicationContextConfiguration – это класс Java, который содержит аннотации Spring. Поэтому необходимо создать аннотацию AnnotationConfigApplicationContext.

Если вместо этого требуется создать ApplicationContext из файлов XML, то необходимо создать ClassPathXmlApplicationContext.

Есть и много других, но в современном приложении Spring обычно начинают с контекста приложения, основанного на аннотациях.

1.2.4. Аннотация @Bean. Spring Bean

На данный момент есть один метод, который может создавать экземпляры UserDao, и один метод, который создает экземпляры DataSource.

Экземпляры, которые создаются этими методами, называются bean-компонентами. Они создаюся Spring контейнером и находятся под его управлением.

Необходимо узнать, сколько экземпляров определенного компонента должно быть создано Spring.

1.2.5. Spring bean scope

Чтобы узнать, сколько экземпляров DAO следует создать в Spring, нужно узнать о bean scope (области применения бина):

– Spring должен создать *singleton*, если все DAO используют один и тот же источник данных;

– Spring должен создать *prototype*, если все DAO получают свой собственный источник данных;

– компоненты должны иметь еще более сложные области действия, например новый источник данных для HttpRequest / HttpSession / WebSocket.

На область действия можно повлиять с помощью еще одной аннотации:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Scope;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MyApplicationContextConfiguration {

 @Bean
 @Scope("singleton")
 // @Scope("prototype") etc.
 public DataSource dataSource() {
  MysqlDataSource dataSource = new MysqlDataSource();
  dataSource.setUser("root");
  dataSource.setPassword("s3cr3t");

  dataSource.setURL(
    "jdbc:mysql://localhost:3306/myDatabase");

  return dataSource;
 }
}
```

Аннотация области (scope annotation) определяет, сколько экземпляров создаст Spring:

– Scope("singleton") → может быть создан только один экземпляр бина (тип по умолчанию);

– Scope("prototype") → создается новый экземпляр при каждом запросе;

– Scope("session") → новый бин создается в контейнере при каждой новой HTTP сессии и т.п.

Большинство приложений Spring почти полностью состоят из одноэлементных bean-компонентов, в которые время от времени добавляются другие области действия bean-компонента (прототип, запрос, сессия, websocket и т.д.).

1.2.6. Spring Java Config

В примере выше bean-компоненты были настроены в конфигурации ApplicationContext с помощью аннотированных @Bean методов Java.

Это именно то, что можно назвать Spring *Java Config* в отличие от указания всего в XML, который исторически был подходом для Spring. Например:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MyApplicationContextConfiguration {

 @Bean
 public DataSource dataSource() {
  MysqlDataSource dataSource = new MysqlDataSource();
  dataSource.setUser("root");
  dataSource.setPassword("s3cr3t");

  dataSource.setURL(
    "jdbc:mysql://localhost:3306/myDatabase");
  return dataSource;
 }


 @Bean
 public UserDao userDao() { // (1)
  return new UserDao(dataSource());
 }
}
```

Новый UserDao() явно вызывается с ручным вызовом dataSource().

1.2.7. Применение @ComponentScan

Добавим к существующей конфигурации дополнительную аннотацию @ComponentScan:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan  // (1)
public class MyApplicationContextConfiguration {

 @Bean
 public DataSource dataSource() {
  MysqlDataSource dataSource = new MysqlDataSource();
  dataSource.setUser("root");
  dataSource.setPassword("s3cr3t");
```

```
  dataSource.setURL(
    "jdbc:mysql://localhost:3306/myDatabase");

  return dataSource;
 }
 // (2)
 // no more UserDao @Bean method!
}
```

1) добавлена аннотации @ComponentScan;

2) определение UserDAO отсутствует в конфигурации контекста.

Аннотация @ComponentScan позволяет всем классам Java находиться в том же пакете, что и конфигурация контекста, если они выглядят как Spring Bean.

Это означает, что если MyApplicationContextConfiguration находится в пакете com.marcobehler, Spring будет сканировать каждый пакет, включая подпакеты, начиная с com.marcobehler для поиска потенциальных компонентов Spring.

Для того, чтобы Spring мог отличать бины Spring, классы должны быть помечены аннотацией маркера, называемой @Component.

1.2.8. Аннотации @Component и @Autowired

Например, добавим аннотацию @Component к UserDAO:

```
import javax.sql.DataSource;
import org.springframework.stereotype.Component;

@Component
public class UserDao {

 private DataSource dataSource;

 private UserDao(DataSource dataSource) { // (1)
  this.dataSource = dataSource;
 }
}
```

Аналогично ранее написанному методу @Bean при аннотации с помощью @Component через @ComponentScan будет получен бин Spring, управляемый контейнером внедрения зависимостей.

Если взглянуть на исходный код таких аннотаций, как @Controller, @Service или @Repository, можно обнаружить, что все они состоят из нескольких дополнительных аннотаций, всегда включая @Component.

Для того, чтобы узнать, что необходимо взять DataSource, указанный как метод @Bean, а затем создать новые UserDAO с этим конкретным DataSource, Spring использует аннотацию @Autowired следующим образом:

```
import javax.sql.DataSource;
import org.springframework.stereotype.Component;
import
org.springframework.beans.factory.annotation.Autowired;

@Component
public class UserDao {

 private DataSource dataSource;

 private UserDao(@Autowired DataSource dataSource) {
  this.dataSource = dataSource;
 }
}
```

Теперь Spring имеет всю информацию, необходимую для создания bean-компонентов UserDAO:

– UserDAO аннотируется @Component → Spring создаст его;

– UserDAO имеет аргумент конструктора @Autowired → Spring автоматически внедрит источник данных, настроенный с помощью вашего метода @Bean;

– если в конфигурациях Spring не было настроено ни одного источника данных, во время выполнения будет получено исключение NoSuchBeanDefinition.

1.2.9. Внедрение зависимости через конструктор и Autowired

В более ранних версиях Spring (до 4.2) необходимо было указывать @Autowired, чтобы внедрение зависимости работало. В более новых версиях Spring внедрение зависимостей происходит без явной аннотации @Autowired в конструкторе:

```
@Component
public class UserDao {

 private DataSource dataSource;

 private UserDao(DataSource dataSource) {
  this.dataSource = dataSource;
 }
}
```

Таким образом, @Autowired делает вещи более явными и его можно использовать во многих других местах, кроме конструкторов.

1.2.10. Field Injection и Setter Injection

Spring не должен обязательно использовать конструктор для внедрения зависимостей, можно внедрять поля напрямую:

```
import javax.sql.DataSource;
import org.springframework.stereotype.Component;
import
org.springframework.beans.factory.annotation.Autowired;

@Component
public class UserDao {

 @Autowired
 private DataSource dataSource;
}
```

Кроме того, Spring также может внедрять сеттеры:

```
import javax.sql.DataSource;
import org.springframework.stereotype.Component;
import
org.springframework.beans.factory.annotation.Autowired;

@Component
public class UserDao {

 private DataSource dataSource;

 @Autowired
 public void setDataSource(DataSource dataSource) {
  this.dataSource = dataSource;
 }
}
```

Рассмотренные стили внедрения (поля, сеттеры) имеют тот же результат, что и внедрение конструктора: получается работающий Spring Bean. Есть еще один метод, называемый внедрением метода.

1.2.11. Внедрение зависимости через конструктор или поле

При выборе стиля внедрения необходимо помнить, что важна согласованность: не стоит использовать в 80 % ваших бинов внедрение зависимости через конструктор, в 10 % – инъекцию поля и для оставшихся 10 % – внедрение метода.

Подход Spring из официальной документации кажется разумным: использовать внедрение зависимости через конструктор для обязательных зависимостей и внедрение с помощью метода установки / через поле для необязательных зависимостей.

1.3. Spring AOP (аспектно-ориентированное программирование) и прокси

Внедрение зависимостей может помочь с созданием более структурированных программ, но суть экосистемы Spring заключается не только во внедрении зависимостей.

Рассмотрим простую ApplicationContextConfiguration:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MyApplicationContextConfiguration {

 @Bean
 public UserService userService() { // (1)
  return new UserService();
 }
}
```

Предположим, что UserService – это класс, который позволяет находить пользователей из таблицы базы данных или сохранять пользователей в этой таблице.

В этом контексте Spring читает конфигурацию, содержащую метод @Bean, и поэтому знает, как создавать и внедрять компоненты UserService. Однако Spring может создавать не только класс UserService.

1.3.1. Создание прокси

Любой метод Spring @Bean может вернуть то, что выглядит и ощущается как UserService, но на самом деле является прокси.

Прокси-сервер в какой-то момент делегирует службу UserService, но сначала он выполнит свою собственную функциональность, схема взаимодействия представлена на рис. 1.

В частности, Spring по умолчанию создаст динамические прокси-серверы Cglib, которым не нужен интерфейс для работы прокси-серверов (например, внутренний механизм прокси-сервера JDK).

Создание прокси позволяет Spring дать компонентам дополнительные функции без изменения кода. В сущности, это то, что является аспектно-ориентированным (AOP) программированием.

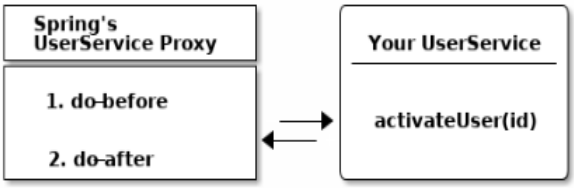

Рис. 1. Взаимодействие прокси-сервера и службы UserService

1.3.2. Аннотация @Transactional

Реализация UserService может выглядеть примерно так:

```
import org.springframework.stereotype.Component;
import
org.springframework.transaction.annotation.Transactional;

@Component
public class UserService {

 @Transactional           // (2)
 public User activateUser(Integer id) {  // (1)

  // execute some sql
  // send an event
  // send an email
 }
}
```

Метод activUser при вызове должен выполнить некоторый SQL-запрос, чтобы обновить состояние пользователя в базе данных, отправить сообщение электронной почты или событие обмена сообщениями.

@Transactional означает, что для работы этого метода необходимо открытое соединение с базой данных / транзакция и что указанная транзакция также должна быть зафиксирована в конце.

Spring может создать компонент UserService через конфигурацию applicationContext, но он не может переписать его, просто вставив код, который открывает соединение с базой данных и фиксирует транзакцию с базой данных.

Зато Spring может создать прокси вокруг UserService, который будет транзакционным. Таким образом, только прокси-сервер будет знать, как открывать и закрывать соединение с базой данных, а затем может просто делегировать его UserService.

Рассмотрим ContextConfiguration:

```
@Configuration
@EnableTransactionManagement // (1)
public class MyApplicationContextConfiguration {

 @Bean
 public UserService userService() { // (2)
  return new UserService();
 }
}
```

1) добавлена аннотация, сигнализирующая Spring о необходимости поддержки @Transactional, автоматически включающей прокси Cglib;

2) с вышеуказанным набором аннотаций Spring не просто создает и возвращает UserService. Он создает Cglib-прокси компонента, который выглядит и делегирует UserService, но фактически оборачивает UserService и предоставляет свои функции управления транзакциями.

1.4. Управление ресурсами Spring

В среде Spring существуют важные утилиты удобства, одной из которых является поддержка ресурсов Spring.

Допустим, необходимо получить доступ к файлу в Java через HTTP или FTP. Можно использовать класс URL в Java и написать код.

Или, предположим, нужно организовать чтение файлов из classpath или из контекста сервлета (из корневого каталога веб-приложений). Опять же нужно написать довольно много стандартного кода, чтобы он работал, и, к сожалению, код будет отличаться для каждого варианта использования (URL-адреса, classpath, контексты сервлетов).

Решением подобных проблем является абстракция ресурсов Spring:

```
import org.springframework.core.io.Resource;

public class MyApplication {

 public static void main(String[] args) {
  ApplicationContext ctx = new
   AnnotationConfigApplicationContext(someConfigClass);// (1)


   Resource aClasspathTemplate = ctx.getResource(
      "classpath:somePackage/application.properties"); // (2)


   Resource aFileTemplate = ctx.getResource(
      "file:///someDirectory/application.properties"); // (3)


   Resource anHttpTemplate = ctx.getResource(
    "https://marcobehler.com/application.properties"); // (4)


   Resource depends = ctx.getResource(
           "myhost.com/resource/path/myTemplate.txt"); // (5)


   Resource s3Resources = ctx.getResource(
                           "s3://myBucket/myFile.txt"); // (6)
 }
}
```

1) для начала нужен ApplicationContext;

2) при вызове getResource() для applicationContext со строкой, которая начинается с «classpath:», Spring будет искать ресурс в application classpath;

3) при вызове getResource() со строкой, начинающейся с «file:», Spring будет искать файл на жестком диске;

4) при вызове getResource() со строкой, которая начинается с «https:» (или «http:»), Spring будет искать файл в Интернете;

5) если не указан префикс, то вызов getResource() зависит от настроенного типа контекста приложения;

6) с дополнительными библиотеками, такими как Spring Cloud, можно напрямую обращаться к путям s3://.

Spring дает возможность доступа к ресурсам с помощью небольшого синтаксиса. Интерфейс ресурса имеет несколько интересных методов:

```
public interface Resource extends InputStreamSource {
 boolean exists();
 String getFilename();
 File getFile() throws IOException;
 InputStream getInputStream() throws IOException;
 // ... other methods commented out
}
```

Он позволяет выполнять самые распространенные операции с ресурсом:

- проверить существование (логический тип);
- получить имя файла;
- получить ссылку на фактический объект File;
- получить прямую ссылку на необработанные данные (InputStream).

Spring позволяет работать с ресурсом независимо от того, находится он в Интернете или в classpath на жестком диске.

1.4.1. Spring Environment

Большая часть любого приложения – это чтение свойств (properties), таких как имя пользователя и пароли базы данных, конфигурация почтового сервера, детализированная конфигурация оплаты и т.д.

В простейшем виде эти свойства находятся в файлах .properties, и их может быть много:

- некоторые из них в classpath, что дает доступ к некоторым паролям, связанным с разработкой;
- другие в файловой системе или на сетевом диске, поэтому рабочий сервер может иметь свои собственные, защищенные свойства;
- некоторые могут быть даже представлены в форме переменных среды операционной системы.

Spring может упростить регистрацию и автоматический поиск свойств во всех этих различных источниках с помощью абстракции среды (environment):

```
import org.springframework.core.env.Environment;
public class MyApplication {

 public static void main(String[] args) {
  ApplicationContext ctx = new
    AnnotationConfigApplicationContext(someConfigClass);

  Environment env = ctx.getEnvironment(); // (1)
  String databaseUrl = env.getProperty("database.url");// (2)
  boolean containsPassword = env.containsProperty(
    "database.password");

  // etc
 }
}
```

1) через ApplicationContext можно получить доступ к текущей среде выполняемого Spring-приложения;

2) с другой стороны, среда, среди прочего, позволяет получать доступ к свойствам.

1.4.2. Spring @PropertySources

Среда состоит из одного или многих источников свойств выполняемого Spring-приложения. Например:

- /mydir/application.properties;
- classpath:/application-default.properties.

Примечание: среда также состоит из профилей «dev» или «production».

По умолчанию среда веб-приложения Spring MVC состоит из параметра ServletConfig / Context, источников системных свойств JNDI и JVM. Они также являются иерархическими, т.е. они имеют порядок важности и переопределяют друг друга.

Определить новые @PropertySources довольно легко:

```
import org.springframework.context.annotation.PropertySources;
import org.springframework.context.annotation.PropertySource;

@Configuration
@PropertySources({
 @PropertySource(
"classpath:/com/${my.placeholder:default/path}/app.properties"
 ),
 @PropertySource("file://myFolder/app-production.properties")
})
public class MyApplicationContextConfiguration {
 // your beans
}
```

Аннотация @PropertySource работает с любым допустимым классом конфигурации Spring и позволяет определять новые

дополнительные источники с помощью абстракции ресурсов Spring, с помощью префиксов: http://, file://, classpath: и т.д.

1.4.3. Аннотация @Value и внедрение значений свойств

Внедрение значений свойств в bean-компоненты аналогично добавлению зависимости с аннотацией @Autowired, только для свойств необходимо использовать аннотацию @Value:

```
import org.springframework.stereotype.Component;
import org.springframework.beans.factory.annotation.Value;

@Component
public class PaymentService {
 @Value("${paypal.password}")  // (1)
 private String paypalPassword;


 public PaymentService(
           @Value("${paypal.url}") String paypalUrl) { // (2)
  this.paypalUrl = paypalUrl;
 }
}
```

1) аннотация @Value работает непосредственно с полями;

2) или по аргументам конструктора.

При использовании аннотации @Value Spring будет проходить через (иерархическую) среду в поисках соответствующего свойства или выдавать сообщение об ошибке, если такого свойства не существует.

1.5. Spring Web MVC

Spring Web MVC (Spring MVC) является веб-средой Spring и позволяет создавать все, что связано с сетью, от небольших веб-сайтов до сложных веб-сервисов. Он также поддерживает фреймворки, такие как Spring Boot.

1.5.1. Spring MVC

Рассмотрим Spring MVC в контексте рендеринга HTML-страниц, например страницы учетной записи пользователя:

– Model (Модель) содержит данные, которые необходимо отобразить на веб-странице. Однако данные полностью независимы от HTML, это простые объекты Java (например, объекты пользователя), из которых состоит приложение;

– View (Представление) будет HTML-шаблоном, который является каркасом для HTML-страницы, написанной с определенной библиотекой шаблонов. Эти библиотеки позволяют включать заполнители в шаблоны, которые позволяют получить доступ к данным модели, например к имени пользователя;

– Controller (Контроллер) будет аннотированным методом @Controller, который отвечает на HTTP-запрос / account и знает, как преобразовать HTTP-запрос в объекты Java, а также объекты Java в ответ HTML.

Следовательно, конечной целью является написание контроллеров, которые сопоставляются с конкретными HTTP-запросами, такими как (/account), и предоставляют пользователю соответствующую HTML-страницу, которая визуализируется путем объединения представления и данных, необходимых для этой страницы. Та же логика действует для написания веб-сервисов JSON или XML.

1.5.2. HttpServlet

Для того, чтобы обрабатывать HTTP-запросы и возвращать браузеру или клиенту соответствующие HTTP-ответы, используются сервлеты, в частности HttpServlet.

В спецификации HTTP определены следующие методы: GET, HEAD, POST, PUT, DELETE, OPTIONS и TRACE. Наиболее часто употребляются методы GET и POST, с помощью которых на сервер передаются запросы, а также параметры для их выполнения.

В задачу метода service() класса HttpServlet входит анализ полученного через запрос метода доступа к ресурсам и вызов метода, имя которого сходно с названием метода доступа к ресурсам, но перед именем добавляется префикс do: doGet(), doPost(), doHead(), doPut(), doDelete(), doOptions() и doTrace(). Разработчик переопределяет нужный метод, разместив в нем функциональную логику.

После написания сервлета необходимо зарегистрировать его в контейнере сервлетов, таком как Tomcat или Jetty. Регистрация сервлета всегда включает путь, чтобы указать, за какие URL в веб-приложении отвечает этот сервлет.

Предположим путь "/*", поэтому каждый входящий HTTP-запрос к приложению обрабатывается одним сервлетом, который может выглядеть следующим образом:

```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class MyServlet extends HttpServlet { // (1)

 @Override
 protected void doGet(HttpServletRequest request,
                      HttpServletResponse response) { // (2)

  if (request.getRequestURI().startsWith("/account")) {
    String userId = request.getParameter("userId");
    // return <html> or {json} or <xml>
    // for an account get request
```

```
  } else if (request.getRequestURI().startsWith("/status")) {
    // return <html> or {json} or <xml>
    // for a health status get request
  } // etc
 }
 @Override
 protected void doPost(HttpServletRequest request,
                       HttpServletResponse response) { // (3)
  // return <html> or {json} or <xml>
  // for a post request, like a form submission
 }
}
```

1) MyServlet должен расширить Java HttpServlet;

2) метод doGet() переопределен для обработки запросов. С отображением сервлета "/" это означает для всех запросов GET. Таким образом, запрос «/status», «/info», «/account» в конечном итоге будет выполнен в одном и том же методе doGet;

3) метод doPost() переопределен для обработки запросов. С отображением сервлета "/" это означает для всех запросов POST. Таким образом, отправка форм в «/register», «/submit-form», «/password-recovery» в конечном итоге будет осуществляться одним и тем же методом doPost().

Обрабатывать каждый запрос к приложению всего двумя способами – это немного громоздко, и контроллеру MyServlet требуется выполнить довольно много работы.

С этой проблемой может помочь DispatcherServlet.

1.5.3. DispatcherServlet

DispatcherServlet обрабатывает каждый входящий HTTP-запрос, анализирует его содержимое и пересылает данные в виде объектов Java в класс @Controller для дальнейшей обработки.

Кроме того, DispatcherServlet берет выходные данные из @Controller и преобразует их в HTML/JSON/XML.

Весь процесс выглядит следующим образом (рис. 2), с пренебрежением большим количеством промежуточных классов, так как DispatcherServlet не выполняет всю работу сам.

Рис.2. Процесс обработки HTTP-запросов

В данном случае Spring заботится обо всей HTTP механике самостоятельно. Разработчику остается лишь написать контроллеры, представления (шаблоны) и модели (Java-объекты).

1.5.4. Создание классов типа Controller

Допустим, необходимо написать класс типа Controller, который обрабатывает запросы /account.

Страница /account будет HTML-страницей с несколькими динамическими переменными (имя пользователя, адрес, информация о подписке и т.д.):

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

/**
*  A class that responds to /account requests.
*  Think of Netflix's account page, where you want to see
your username/password/subscription info
*/
@Controller // (1)
public class AccountController {

 @GetMapping("/account/{userId}")
 public String account(@PathVariable Integer userId) { // (2)
  // TODO retrieve name, address, subscription information
  return "templates/account"; // (3)
 }
}
```

1) класс AccountController аннотируется @Controller и должен реагировать на HTTP-запросы и ответы;

2) Java-метод account() аннотируется @GetMapping. Теперь все запросы, похожие на /account/{userId}, должны обрабатываться именно этим методом контроллера. Более того, DispatcherServlet возьмет {userId}, преобразует его в целое число и использует его в качестве параметра метода;

3) Java-метод возвращает строку с именем account, причем не просто строку, а ссылку на представление (HTML-шаблон).

1.5.5. Генерация HTML-представления с помощью Spring Web MVC

Spring MVC по умолчанию предполагает, что нужно визуализировать HTML. И, конечно же, нет смысла отображать HTML-строки с помощью конкатенации строк, можно использовать шаблонную среду, такую как Velocity или Freemarker, так как Spring интегрируется со всеми этими технологиями.

Допустим, имеется следующее представление (шаблон):

```
classpath:/templates/account.vm
<html>
  <body>
    Hello $user.name, this is your account!
    <!-- list subscriptions etc -->
  </body>
</html>
```

Это очень простая HTML-страница, содержащая одну переменную, $user.name.

Внесем изменения в код контроллера аккаунта:

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

@Controller
public class AccountController {

 @GetMapping("/account/{userId}")
 public String account(Model model,
                       @PathVariable Integer userId) { // (1)
  // TODO validate user id

  model.addAttribute("user", userDao.findById(userId));// (2)
  return "templates/account"; // (3)
 }
}
```

1) Spring автоматически вводит параметр «Model» в методы контроллера. Модель содержит любые данные, которые нужно представить, т.е. шаблон;

2) модель Spring ведет себя почти как map (отображение), достаточно добавить в нее необходимые данные, и можно ссылаться на ключи отображения из шаблона;

3) в результате возвращается ссылка на view, шаблон аккаунта, написанный выше.

1.5.6. Генерация JSON/XML-представления с помощью Spring Web MVC

Для того, чтобы сгенерировать JSON/XML-представление, нужна соответствующая библиотека, такая как Jackson, добавленная в проект. Тогда достаточно просто аннотировать Controller дополнительной аннотацией, чтобы объекты Java преобразовывались напрямую в XML/JSON:

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class HealthController {

 @GetMapping("/health")
 @ResponseBody // (1)
 public HealthStatus health() {
  return new HealthStatus(); // (2)
 }
}
```

1) добавлена дополнительная аннотация @ResponseBody в метод контроллера, сигнализирующая Spring о том, что Java-объект HealthStatus необходимо записать непосредственно в HttpResponse (например, в виде XML или JSON);

2) в результате возвращается простой Java-объект внутри метода, а не строковая ссылка на представление.

1.5.7. Согласование контента (content negotiation) Spring MVC

Существует множество способов, с помощью которых можно указать Spring MVC, какой формат ответа должен быть при выполнении запроса к приложению Spring MVC:

− указать заголовок Accept, такой как «Accept: application/json» или «Accept: application/xml». Это требует определенных библиотек в classpath для поддержки сортировки XML или JSON;

− добавить расширение URL к пути запроса, например /health.json или /health.xml. Это требует настройки на стороне Spring MVC;

− добавить параметр запроса в путь запроса, например /health?Format = json. Это требует настройки на стороне Spring MVC.

В результате Spring знает, что метод аннотирован с помощью @ResponseBody, и клиент хочет «JSON».

Если библиотека на classpath (как Jackson), которая может визуализировать JSON, в наличии, то HealthStatus будет преобразован в JSON. В противном случае будет выведено исключение.

1.5.8. HTTP-запросы и HTTP-ответы Spring MVC

Spring MVC может понимать в основном все типы ввода, которые предлагает HTTP, с помощью сторонних библиотек. Если он получает тела запросов JSON, XML или HTTP (Multipart) Fileupload, то преобразует их в объекты Java.

Spring MVC может записывать в HttpServletResponse с помощью сторонних библиотек все что угодно, будь то HTML, JSON, XML или даже тела ответов WebSocket. Более того, он берет объекты Java и генерирует тела ответов.

Таким образом, Spring MVC – это фреймворк MVC, который позволяет довольно легко писать HTML / JSON / XML веб-сайты или веб-сервисы. Он прекрасно интегрируется с контейнером внедрения зависимостей Spring, включая все его вспомогательные утилиты.

Он позволяет сосредоточиться на написании Java-классов вместо того, чтобы иметь дело с кодом сервлета, то есть работать с HTTP-запросами и ответами, и дает хорошее разделение проблем между моделью и представлениями.

2. Порядок выполнения работы

1. Изучите теоретическую часть лабораторной работы.
2. Выполните задания практической части лабораторной работы по варианту.

3. Практическая часть

3.1. Задание 1

В каждом из вариантов необходимо выполнить следующие действия:

• создать БД (не менее 3-х таблиц). Привести таблицы к одной из нормальных форм;

• организацию соединения с базой данных вынести в отдельный класс, метод которого возвращает соединение;

• с помощью фреймворков Spring / Spring Boot создать веб-приложение, которое будет отображать, модифицировать и удалять данные из таблиц;

• серверная часть должна включать в себя БД и Java-приложение;

• клиентская часть должна включать веб-страницы, оформленные с помощью HTML, JavaScript, CSS и любых других известных веб-технологий;

• для взаимодействия с БД Java-приложение может использовать JDBC, Spring Data, Hibernate.

1. **Файловая система**. В БД хранится информация о дереве каталогов файловой системы — каталоги, подкаталоги, файлы.

Для каталогов необходимо хранить: родительский каталог, название.

Для файлов необходимо хранить: родительский каталог, название, место, занимаемое на диске.

• Определить полный путь заданного файла (каталога).

• Подсчитать количество файлов в заданном каталоге, включая вложенные файлы и подкаталоги.

• Подсчитать место, занимаемое на диске содержимым заданного каталога.

• Найти в базе файлы по заданной маске с выдачей полного пути.

• Переместить файлы и подкаталоги из одного каталога в другой.

• Удалить файлы и подкаталоги заданного каталога.

2. **Видеотека**. В БД хранится информация о домашней видеотеке: фильмы, актеры, режиссеры.

Для фильмов необходимо хранить: название, имена актеров, дату выхода, страну, в которой выпущен фильм.

Для актеров и режиссеров необходимо хранить: ФИО, дату рождения.

• Найти все фильмы, вышедшие на экран в текущем и прошлом году.

• Вывести информацию об актерах, снимавшихся в заданном фильме.

• Вывести информацию об актерах, снимавшихся как минимум в N фильмах.

• Вывести информацию об актерах, которые были режиссерами хотя бы одного из фильмов.

• Удалить все фильмы, дата выхода которых была более заданного числа лет назад.

3. **Расписание занятий**. В БД хранится информация о преподавателях и проводимых ими занятиях.

Для предметов необходимо хранить: название, время проведения (день недели), аудитории, в которых проводятся занятия.

Для преподавателей необходимо хранить: ФИО; предметы, которые он ведет; количество пар в неделю по каждому предмету; количество студентов, занимающихся на каждой паре.

• Вывести информацию о преподавателях, работающих в заданный день недели в заданной аудитории.

• Вывести информацию о преподавателях, которые не ведут занятия в заданный день недели.

• Вывести дни недели, в которых проводится заданное количество занятий.

• Вывести дни недели, в которых занято заданное количество аудиторий.

• Перенести первые занятия заданных дней недели на последнее место.

4. **Письма**. В БД хранится информация о письмах и отправителях.

Для пользователей необходимо хранить: ФИО, дату рождения.

Для писем необходимо хранить: отправителя, получателя, тему письма, текст письма, дату отправки.

• Найти пользователя, длина писем которого наименьшая.

• Вывести информацию о пользователях, а также количестве полученных и отправленных ими письмах.

• Вывести информацию о пользователях, которые получили хотя бы одно сообщение с заданной темой.

• Вывести информацию о пользователях, которые не получали сообщения с заданной темой.

• Направить письмо заданного человека с заданной темой всем адресатам.

5. **Сувениры**. В БД хранится информация о сувенирах и производителях.

Для сувениров необходимо хранить: название, реквизиты производителя, дату выпуска, цену.

Для производителей необходимо хранить: название, страну.

• Вывести информацию о сувенирах заданного производителя.

• Вывести информацию о сувенирах, произведенных в заданной стране.

• Вывести информацию о производителях, чьи цены на сувениры меньше заданной.

• Вывести информацию о производителях заданного сувенира, произведенного в заданном году.

• Удалить заданного производителя и его сувениры.

6. **Заказ**. В БД хранится информация о заказах магазина и товарах в них.

Для заказа необходимо хранить: номер заказа, товары в заказе, дату поступления.

Для товаров в заказе необходимо хранить: товар, количество.

Для товара необходимо хранить: название, описание, цену.

• Вывести полную информацию о заданном заказе.

• Вывести номера заказов, сумма которых не превосходит заданную и количество различных товаров в которых равно заданному.

• Вывести номера заказов, содержащих заданный товар.

• Вывести номера заказов, не содержащих заданный товар и поступивших в течение текущего дня.

• Сформировать новый заказ, состоящий из товаров, заказанных в текущий день.

• Удалить все заказы, в которых присутствует заданное количество заданного товара.

7. **Продукция**. В БД хранится информация о продукции компании.

Для продукции необходимо хранить: название, группу продукции (телефоны, телевизоры и др.), описание, дату выпуска, значения параметров.

Для групп продукции необходимо хранить: название, перечень групп параметров (размеры и др.).

Для групп параметров необходимо хранить: название, перечень параметров.

Для параметров необходимо хранить: название, единицу измерения.

• Вывести перечень параметров для заданной группы продукции.

• Вывести перечень продукции, не содержащий заданного параметра.

• Вывести информацию о продукции для заданной группы.

• Вывести информацию о продукции и всех ее параметрах со значениями.

• Удалить из базы продукцию, содержащую заданные параметры.

• Переместить группу параметров из одной группы товаров в другую.

8. **Погода**. В БД хранится информация о погоде в различных регионах.

Для погоды необходимо хранить: регион, дату, температуру, осадки.

Для регионов необходимо хранить: название, площадь, тип жителей.

Для типов жителей необходимо хранить: название, язык общения.

• Вывести сведения о погоде в заданном регионе.

• Вывести даты, когда в заданном регионе шел снег и температура была ниже заданной отрицательной.

• Вывести информацию о погоде за прошедшую неделю в регионах, жители которых общаются на заданном языке.

• Вывести среднюю температуру за прошедшую неделю в регионах с площадью больше заданной.

9. **Магазин часов**. В БД хранится информация о часах.

Для часов необходимо хранить: марку, тип (кварцевые или механические), стоимость, количество, реквизиты производителя.

Для производителей необходимо хранить: название; страну.

• Вывести марки заданного типа часов.

• Вывести информацию о механических часах, стоимость которых не превышает заданную.

• Вывести марки часов, изготовленных в заданной стране.

• Вывести производителей, общая сумма часов которых в магазине не превышает заданную.

10. **Города**. В БД хранится информация о городах и их жителях.

Для городов необходимо хранить: название, год основания, площадь, количество населения для каждого типа жителей.

Для типов жителей необходимо хранить: город проживания, название; язык общения.

• Вывести информацию обо всех жителях заданного города, разговаривающих на заданном языке.

• Вывести информацию обо всех городах, в которых проживают жители выбранного типа.

• Вывести информацию о городе с заданным количеством населения и всех типах жителей, в нем проживающих.

• Вывести информацию о самом древнем типе жителей.

11. **Планеты**. В БД хранится информация о планетах, их спутниках и галактиках.

Для планет необходимо хранить: название, радиус, температуру ядра, наличие атмосферы, наличие жизни, спутники.

Для спутников необходимо хранить: название, радиус, расстояние до планеты.

Для галактик необходимо хранить: название, планеты.

• Вывести информацию обо всех планетах, на которых присутствует жизнь, и их спутниках в заданной галактике.

• Вывести информацию о планетах и их спутниках, имеющих наименьший радиус и наибольшее количество спутников.

• Вывести информацию о планете, имеющей максимальное количество спутников, но с наименьшим общим объемом этих спутников, информацию о ее спутниках и галактике, в которой она находится.

• Найти галактику, сумма ядерных температур планет которой наибольшая.

12. **Точки**. В БД хранится некоторое конечное множество точек с их координатами.

• Вывести точку из множества, наиболее приближенную к заданной точке.

• Вывести точку из множества, наиболее удаленную от заданной точки.

• Вывести точки из множества, лежащие на одной прямой с заданной прямой.

13. **Треугольники**. В БД хранятся треугольники и координаты их точек на плоскости.

• Вывести треугольник, площадь которого наиболее приближена к заданной.

• Вывести треугольники, сумма площадей которых наиболее приближена к заданной.

• Вывести треугольники, которые помещаются в окружность заданного радиуса.

• Вывести все равнобедренные треугольники.

• Вывести все равносторонние треугольники.

• Вывести все прямоугольные треугольники.

• Вывести все тупоугольные треугольники с площадью больше заданной.

14. **Стихотворения**. В БД хранятся несколько стихотворений с указанием автора и года создания. Для хранения использовать объекты типа Clob. Клиент выбирает автора и критерий поиска.

• Вывести стихотворение, в котором больше всего восклицательных предложений.

• Вывести стихотворение, в котором меньше всего повествовательных предложений.

• Вывести информацию о наличии среди стихотворений сонетов и их количестве.

15. **Четырехугольники**. В БД хранятся координаты вершин выпуклых четырехугольников на плоскости.

• Вывести координаты вершин параллелограммов.

• Вывести координаты вершин трапеций.

4. Содержание отчета

1. Краткие теоретические сведения о Spring Framework.
2. Код программ.
3. Результаты выполнения программ.
4. Выводы по работе.

5. Контрольные вопросы

1. Что представляет собой Spring Framework? Из каких основных компонентов он состоит?
2. Каковы основные особенности и преимущества Spring Framework?
3. Что такое внедрение зависимостей? Какие типы внедрения зависимостей существуют, чем они отличаются и в каких случаях используются?
4. Что представляет собой контейнер Spring IoC?
5. Как создать ApplicationContext в программе Java?

6. Что такое Spring Bean? Перечислите различные bean scope.
7. Для чего применяется аннотация @Autowired?
8. Что представляет собой Spring MVC?
9. Для чего используется HttpServlet?
10. Каковы особенности DispatcherServlet?

Библиографический список

1. Java. Промышленное программирование : практ. пособие / И.Н. Блинов, В.С. Романчик. – Минск : УниверсалПресс, 2007. – 704 с.
2. Java. Методы программирования : уч.-мет. пособие / И.Н. Блинов, В.С. Романчик. — Минск : издательство «Четыре четверти», 2013. — 896 с.
3. Spring Framework Documentation. [Электронный ресурс] – URL: https://docs.spring.io/spring-framework/docs/current/reference/html/index.html. – Режим доступа: свободный доступ.
4. Что такое Spring Framework? От внедрения зависимостей до Web MVC. [Электронный ресурс] – URL: https://habr.com/ru/post/490586/?ysclid=l95oqt5ib8864818737. – Режим доступа: свободный доступ.

Основы промышленной разработки
программного обеспечения. Часть 3

Составители: Костров Борис Васильевич
Бастрычкин Александр Сергеевич
Трушина Евгения Александровна

Редактор М.Е. Цветкова
Корректор С.В. Макушина
Подписано в печать 20.04.23. Формат бумаги 60х84 1/16.
Бумага писчая. Печать трафаретная. Усл. печ. л. 2,0.
Тираж 20 экз. Заказ
Рязанский государственный радиотехнический университет им. В.Ф. Уткина.
390005, Рязань, ул. Гагарина, 59/1.
Редакционно-издательский центр РГРТУ.