Министерство науки и высшего образования
Российской Федерации
Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Тверской государственный университет»
Факультет прикладной математики и кибернетики

В.Л. Волушкова

Архитектурные решения Java для доступа к данным

Учебное пособие

ТВЕРЬ 2019

УДК 004.438
ББК 3973.2-018.1
Б68

Волушкова В.Л.

Б68 Архитектурные решения java для доступа к данным: учеб. пособие. – Тверь: Твер. гос. ун-т, 2019. – 137 с.

ISBN 978-5-7609-0884-1

Рассматриваются технологии программирования на примере языка java. Многие программисты решают идентичные задачи и находят похожие решения. Для того, чтобы не создавать проблем при проектировании, можно воспользоваться уже готовыми решениями – фреймворками и шаблонами проектирования. В книге приведены описание и примеры использования фреймворка Spring для реализации доступа к данным.

Книга предназначена для магистров направления «01.04.02 Прикладная математика и информатика», изучающих курс «Технологии разработки программного обеспечения». Книга может быть полезна для тех, кто впервые столкнулся с трудностями проектирования корпоративных приложений.

УДК 004.438
ББК 3973.2-018.1

ISBN 978-5-7609-0884-1

Оглавление

Введение

Данное учебное пособие не претендует на полное освещение технологий java для доступа к данным. Под технологиями в данном случае имеется в виду комбинация паттернов проектирования и фреймворков. Целью работы является демонстрация применения технологий для тех, кто впервые столкнулся с проблемами доступа к данным из java-приложений. В книге описаны паттерны доступа к данным, на которых построены фреймворки, применяемые для создания корпоративных приложений.

Паттерн - это архитектурная конструкция, помогающая описать и решить некую задачу проектирования. Ко времени представления паттернов разработка программного обеспечения (ПО) стала индустриальной задачей. Многие программисты понимали, что не стоит изобретать велосипед при создании нового ПО. Использование паттернов часто помогает решить эту задачу и бывает полезным, как отдельному разработчику, так и целой команде.

Кэнт Бэк и Вард Каннингем представили первые шаблоны проектирования для языка Smalltalk в 1987 г. Э. Гамма, Р. Хелм, Р. Джонсон и Дж. Влиссидес в 1995 г опубликовали книгу «Приемы объектно–ориентированного проектирования. Паттерны проектирования» [1], в которой были представлены 23 шаблона, которые стали сейчас основными. Эта книга и многие другие издания, посвященные шаблонам проектирования, рассчитаны на разработчиков программ, имеющих опыт программирования. Существует необходимость в изданиях, рассматривающих паттрны проектирования, которые будут по силам неопытным пользователям. Одной из таких книг является книга Э. Фримен, Э. Фримен, К. Сьерра, Б. Бейтс «Паттерны проектирования» [2]. Можно рассмотреть также предыдущую книгу автора «Технологии программирования. Введение в паттерны проектирования java» [3], в которой автор попытался соединить простое изложение шаблонов проектирования, присущее книге [2] с описанием конкретного применения этих шаблонов на практическом примере.

Что такое фреймворк (framework)? Словарное определение фреймворка — «необходимая несущая конструкция». Фреймворки регулируют абстрактные вещи, такие как организация кода и уменьшение сложности кода.

Корпоративные приложения создаются командами программистов, в которых могут быть как хорошие, так и плохие программисты. Нужна некоторая парадигма, которая не позволит плохим программистам нанести слишком большой ущерб. Обезопасить разработку от такого ущерба не могут ни объектно-ориентированное программирование, ни шаблоны программирования. Существует мнение, что хороший код может быть написан только с использованием сильного фреймворка [4].

В пособии рассматриваются фреймворки Spring и Hibernate, как наиболее часто используемые. Эти фреймвоки работают с сервером приложений, который уменьшает сложность разработки. С этим сервером программисту нет необходимости знать, какая БД используется, по какому протоколу к серверному приложению обращаются пользователи, думать, где продакшен сервер, а где сервер разработчика. Программист просто занят бизнес логикой. Но, это только если все заранее настроено, и работает как часы. В команде разработчиков, более опытные программисты настраивают сервер приложений, а менее опытные вообще не знают, что там происходит.

В данном пособии как раз и рассматривается то, что происходит на сервере приложений при использовании Spring framework и Hibernate. Сначала мы рассмотрим паттерны проектирования, использующиеся в этих фреймворках, а затем сами фреймворки. Вы увидите, что Spring Framework это не боль и мучительное преодоление web.xml, persistence.xml, beans.xml, а сборка приложения как карточного домика по кусочкам, что является достаточно быстрым и комфортным процессом.

1. Паттерн стратегия

Для знакомства с паттернами, выберем для обсуждения наиболее употребляемый паттерн, например, паттерн «стратегия». Это паттерн поведения объектов, инкапсулирующий отдельные алгоритмы.

Попробуем разобраться в старом вопросе - что лучше отношение «содержит» или отношение «является». Иными словами - что применять наследование или композицию. Разберемся в терминологии.

Если между объектами существует отношение «является», то каждый объект подкласса является объектом суперкласса. В языке java объектные переменные являются полиморфными, т.е. объект подкласса можно присвоить переменной суперкласса. Автоматический замещение (перегрузка) метода суперкласса методом подкласса во время выполнения программы называется динамическим связыванием (аналог – виртуальный метод с++).

Рассмотрим пример с использованием полиморфизма. Допустим, имеется класс список. Каждый узел есть экземпляр вложенного класса Node. Необходимо создать класс стек и очередь, используя класс список.

Первое решение – использовать наследование, т.е. отношение «является». Классы «стек» и «очередь» отличаются только методом «добавить элемент» при условии, что извлекается элемент всегда из головы списка. Правила построения UML-диаграмм приведены в приложение 4.

UML-диагрмма такого решения показана на рис.1.

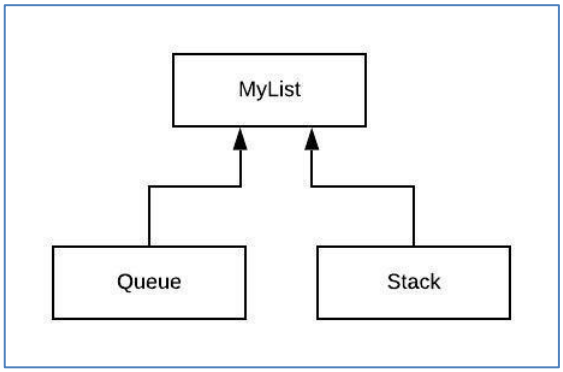


Рисунок 1. UML-диаграмма, реализующая отношение "является"

```
      MyList<E>
public class MyList<E> {
    class Node<E> {
        E item;
```

```
        Node<E> next;

        public Node() {
        }

        public Node(E item, Node<E> next) {
            this.item = item;
            this.next = next;
        }
    }

    Node<E> head;
    Node<E> tail;

    public MyList() {
        head = null;
        tail = null;
    }

    public void addBegin(E item) {
        Node<E> h = new Node<>(item, null);
        if (head == null) tail = head=h;
        else {
            h.next =  head;
            head = h;
        }
    }

    public void addEnd(E item) {
        Node<E> h = new Node<>(item, null);
        if (head == null) tail = head = h;
        else {
            tail.next = h;
            tail = h;
        }
    }

    public E getItemBegin(){
        if(head==null)return null;
        else   return head.item;
    }
    public E pop(){
        if(head==null)return null;
        else  {
            E  h = head.item;
            head = head.next;
            return h;
        }
    }
    public void push(E item){
        addEnd(item);
    }
    public boolean isEmpty(){
        return (head==null)?  true: false;
    }
    @Override
    public String toString() {
        String s = "";
        for(Node<E> h = head; h!=null; h = h.next){
            s+=" "+ h.item;
        }
        return s;
    }
```

```
}
```

В приведенном примере класс MyList<E> работает как «очередь», т.к. метод push добавляет элемент списка в конец. В классе «стек» элемент добавляется в начало списка.

Queue<E> и Stack<E>

```
public class Queue<E> extends MyList<E> {

}
public class Stack<E> extends MyList<E> {
    public void push(E item) {
        super.addBegin(item);
    }
}
```

В методе main создадим экземпляр класса «стек» и проверим его работу. Добавим целые числа от 0 до 9 в стек, а затем извлечем их.

Фрагмент main

```
MyList<Integer> list = new Stack<>();
for (int i = 0; i < 10; i++)  list.push(i);
System.out.println(list);
while(!list.isEmpty()){
    list.pop();
    System.out.println(list);
}
```

Результат работы получится следующим.

Результат

```
9 8 7 6 5 4 3 2 1 0
 8 7 6 5 4 3 2 1 0
 7 6 5 4 3 2 1 0
 6 5 4 3 2 1 0
 5 4 3 2 1 0
 4 3 2 1 0
 3 2 1 0
 2 1 0
 1 0
 0
```

В первой строчке выводятся, начиная с головы списка, все элементы, попавшие в стек. Далее из стека извлекается по одному элементу и выводится содержимое стека, до тех пор, пока содержимое стека не закончится. Результат работы соответствует принципу LIFO (last in — first out, «последним пришёл — первым вышел»).

Выполним тоже самое с очередью, предварительно создав экземпляр класса Queue.

```
MyList<Integer> list = new Queue<>();
```

В силу полиморфизма переменной типа `MyList` можно присваивать объекты типа `Stack` и `Queue`.

Результаты работы выше приведенной программы будет соответствовать принципу работы FIFO (first in— first out, «первым пришёл — первым вышел»).

Результат
0 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 2 3 4 5 6 7 8 9 3 4 5 6 7 8 9 4 5 6 7 8 9 5 6 7 8 9 6 7 8 9 7 8 9 8 9 9

В приведенном примере нельзя динамически изменить объект `list` с очереди на стек и наоборот. Обязательно надо создавать новый объект.

Рассмотрим пример, в котором отношение между списком, очередью и стеком будет отношением «содержит». UML- диаграмма такого отношения показана на рис 2.

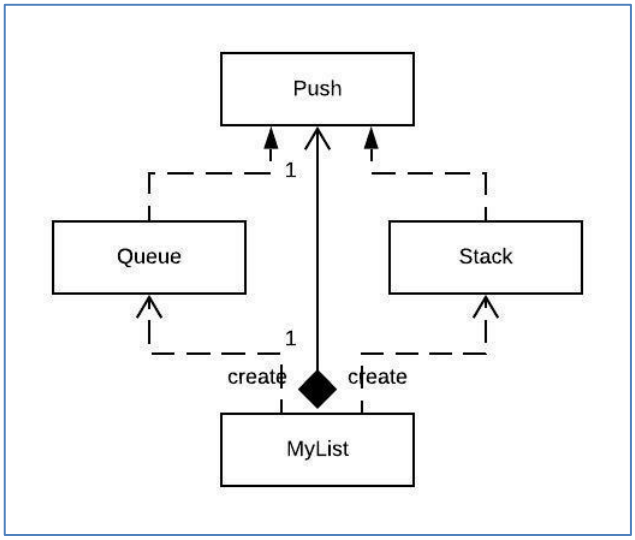


Рисунок 2. UML- диаграмма, реализующая отношения "содержит"

Изменим класс список. Добавим данное Push<E> **sQ** класса MyList, которое может содержать как экземпляр класса «стек», так и экземпляр класса «очередь». Для этого создадим интерфейс Push.

Push<E>

```
public interface Push<E> {
    void push(E item);
    void setList(MyList<E> list);
}
```

Действительно, классы `Stack` и `Queue` имеют разные методы push. Метод setList устанавливает ссылку на экземпляр класса `MyList` – основу `Stack` или `Queue`.

Queue<E> и Stack<E>

```
public class Queue<E> implements Push<E> {
    MyList<E> list;

    public Queue(MyList<E> list) {
        this.list = list;
    }

    public Queue() {
    }

    @Override
    public void push(E item) {
        list.addEnd(item);
    }

    @Override
    public void setList(MyList<E> list) {
        this.list = list;
    }
}
public class Stack<E> implements Push<E> {
    MyList<E> list;

    public Stack(MyList<E> list) {
        this.list = list;
    }

    public Stack() {
    }

    @Override
    public void push(E item) {
        list.addBegin(item);
    }
    @Override
    public void setList(MyList<E> list) {
        this.list = list;
    }
}
```

В классе `MyList` создадим метод setSq, который устанавливает способ работы списка – либо стек, либо очередь.

setSq

```
public void setSq(Push<E> sq) {
    sQ = sq;
    sQ.setList(this);
}
```

Делегируем добавление элемента в список объекту sQ.

push

```
public void push(E item){
    sQ.push(item);
}
```

Таким образом, метод push класса `MyList` делегирует способ добавления элемента в список объекту sQ, который может быть стеком или очередью, или еще чем-нибудь, имплементирующим интерфейс Push.

В методе main создадим экземпляр класса «список», установим сначала объект sQ как очередь и проверим его работу. Затем установим sQ как стек и опять добавим элементы. Получили «хитрый» объект «список», который сначала работал как очередь, а потом как стек.

Фрагмент main

```
composition.MyList<Integer> list = new composition.MyList<>();
list.setSq(new composition.Queue<Integer>());
for (int i = 0; i < 10; i++)
    list.push(i);
System.out.println(list);
list.setSq(new composition.Stack<Integer>());
for (int i = 0; i < 10; i++)
    list.push(i);
System.out.println(list);
while(!list.isEmpty()){
    list.pop();
    System.out.println(list);
}
```

Результаты работы такого списка приведен ниже.

```
Результат
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
3 2 1 0 0 1 2 3 4 5 6 7 8 9
2 1 0 0 1 2 3 4 5 6 7 8 9
1 0 0 1 2 3 4 5 6 7 8 9
0 0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 9
3 4 5 6 7 8 9
4 5 6 7 8 9
5 6 7 8 9
6 7 8 9
7 8 9
8 9
9
```

Из приведенных примеров видно, что отношение «содержит» дает больше возможностей для гибкого кода. Объект list может быть стеком, очередью и любым хранилищем, объект sQ которого, поддерживает интерфейс Push. Описанный выше прием проектирования, в котором поведение делегируется объекту внутри класса, называется паттерн «стратегия».

Задания к гл.1

1. Создайте иерархию классов, описывающих создателей книги. В качестве создателей выступает писатель и художник. Пусть писатель пишет книгу ручкой, а художник рисует картинки кистью. Ваша программа должна в одной и той же переменной сохранять одного из двух создателей книги. Продемонстрируйте метод создания книги различными создателями (писатель –“write pen the book” художник “Brush paint the book”.
2. Создайте иерархию классов, описывающую книгу. У книги есть автор, название, количество страниц и цена. Ваша программа должна выводить информацию о книге, причем цена должна быть в долларах или в евро. При изображении цены в долларах перед

числом должен стоять значок $ ($34) и описание (34 доллара). При изображении цены в евро после числа должен быть текст “eur” (34eur)и описание (34 евро).

2. Паттерн IoC (Inversion of Control)

Inversion of Control (инверсия управления) — это набор правил для написания слабо связанного кода. При соблюдении принципов IoC, каждый компонент системы будет изолирован от других, и не будет полагаться на конкретную реализацию других компонентов. Внедрение зависимостей (DI – Dependency Injection) — это одна из реализаций IoC. Терминологию, используемую в данной книге, можно найти в приложении 1.

Что такое инверсия контроля. Рассмотрим простую схему (рис.3). Менеджер создает интерфейсы: связь с базой данных, вводит login пароль и устанавливает связь с сервером. Менеджер создает зависимости и их контролирует. Это плохо, т.к. мы не можем легко тестировать и нельзя заменять зависимости без перекомпиляции.

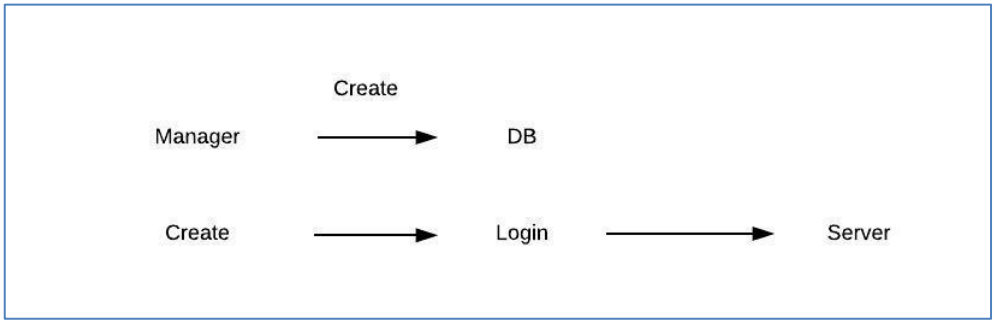


Рисунок 3 Прямая схема по созданию связей.

Рассмотрим схему инверсии контроля (рис.4). Менеджер не создает зависимости, а получает. К таксисту подсаживается пассажир, а не таксист ловит пассажира. Почему это важно? Во-первых, для тестирования, во-вторых, можно заменять зависимости (менять БД). Остается открытым вопрос– от куда получать зависимости?

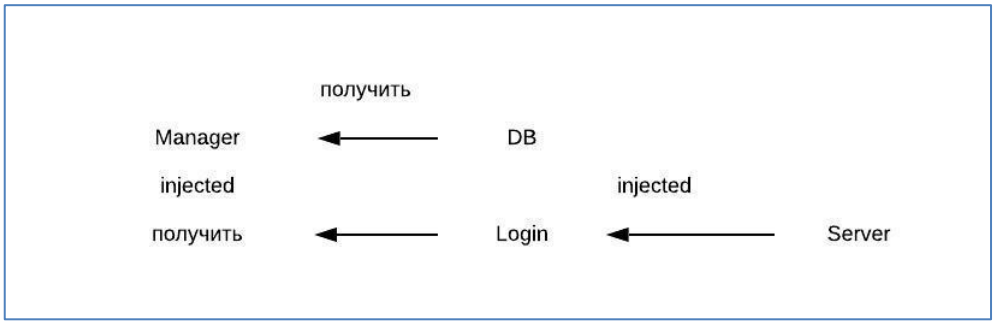


Рисунок 4 Внедрение зависимостей (Dependency Injection).

Для автоматизации соблюдения принципов IoC существуют библиотеки, фреймворки и прочие программы, которые позволяют упростить написание кода. Для java популярным фреймворком поддерживающим IoC является Spring.

Spring – это библиотека, представленная jar файлом. Этот фремворк позволяет создавать зависимости между описанными объектами при помощи XML файлов. За счет этого реализуется возможность создания связи между объектами в уже откомпилированных модулях. Spring характеризуется следующим.

- управляет жизненным циклом объектов, зависимость между объектами, т.е. – это оболочка (контейнер), в которой живет приложение.
- в основе лежит паттерн проектирования IoC и DI(dependency Injection)
- использует AOP аспектно-ориентированное программирование (выделение сквозной функциональности в отдельные модули – аспекты). В spring AOP урезанное.

Структура фремворка Spring представлен на рис.5.

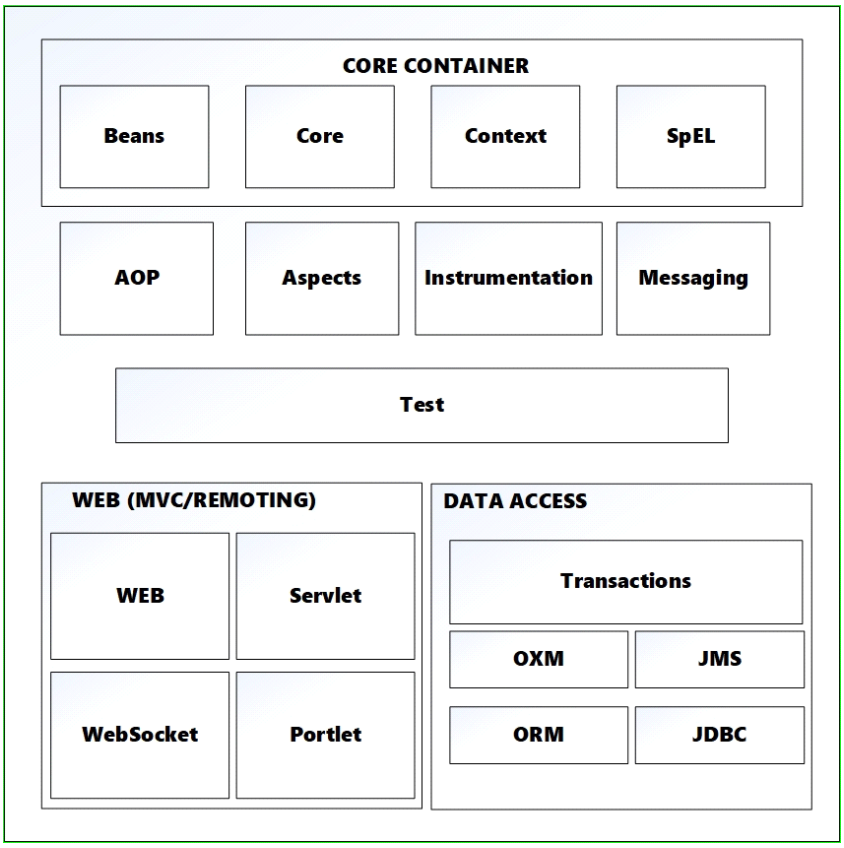


Рисунок 5 Схема Spring.

Ядро платформы, предоставляет базовые средства для создания приложений — управление компонентами (бинами, beans), внедрение зависимостей, MVC фреймворк, транзакции, базовый доступ к БД. В основном это низкоуровневые компоненты и абстракции. По сути, неявно используется всеми другими компонентами.

Рассмотрим пример. Пусть имеется интерфейс Hello. Этот интерфейс имеет один метод sayHi.

Hello

```
public interface Hello {
    void sayHi();
}
```

Создадим класс HelloImpl, который реализует интерфейс Hello. Метод sayHi выводит строку s.

HelloImpl

```
public class HelloImpl implements Hello{
    HelloImpl (String hi) {        s=hi;    }
    @Override
    public void sayHi() {         System.out.println(s);    }
    private String s;
    public String getS() {        return s;    }
```

```
        public void setS(String s) {        this.s = s;    }
    }
```

Создадим объект Hello h, реализующий класс HelloImpl и вызовем метод sayHi.

```
    main
    public static void main(String[] args) {
            Hello h = new HelloImpl ("hi");
            h.sayHi();
        }
```

Мы напрямую создаем реализацию интерфейса Hello с переменной “hi”. Для того, чтобы изменить выводимую строку надо перекомпилировать класс, содержащий main. Это неудобно. Как сделать зависимость, не требующую перекомпиляции? Выполним этот простой пример с помощью Spring.

Создадим конфигурационный файл Spring.

Сначала опишем xml схему, которая берется из готовых решений.

```
    XML схема
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns ="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd ">
```

Рабочая часть Xml начинается с тега `beans`. Все объекты проекта называются bean и находятся внутри контейнера Spring. Ниже приведен конфигурационный файл newSpringXMLConfig.xml.

```
    newSpringXMLConfig.xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd ">

<bean name="hello" class= "hellospring.HelloImpl">
    <constructor-arg  value = "hi!"/>
    <property name="s" value = "hi111"/>
</bean>
</beans
```

В теге **<bean>** создается бин hello ссылающийся на класс HelloImpl (имя класса пишется полностью, т.е. с именем пакета). В теге constructor-arg параметром value задается значение строки в конструкторе класса HelloImpl.

В теге <property> задается значение строки s, как параметра метода setS (String s) класса HelloImpl.

Приведем класс HelloSpring с методом main, в котором создается связь с файлом newSpringXMLConfig.xml.

HelloSpring

```
package hellospring;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class HelloSpring {
    public static void main(String[] args) {
      ApplicationContext context =
new ClassPathXmlApplicationContext("newSpringXMLConfig.xml");
      Hello h = (Hello)context.getBean("hello");
      h.sayHi();
    }
 }
```

При создании объекта Context используется класс ClassPathXmlApplicationContext, конструктор которого ищет файл newSpringXMLConfig.xml в текущем каталоге, заданном в проекте с помощью переменной ClassPath в config.xml. Создать контекст – значит поднять все объекты, описанные в бинах. Обычно при совместной работе это достаточно длительный процесс, т.к. создаются объекты связи с БД, сетевые объекты и т.д. Создание контекста происходит один раз для всех разработчиков, т.е. все программисты проекта имеют доступ к контексту. Чтобы вручную в коде не создавать объекты, всю операцию по созданию объектов возлагают на Spring.

Объект Hello h есть экземпляр класса HelloImpl. Далее для объекта h вызывается метод sayHi. Изменить строку вывода можно задав в файле конфигурации другое значение строки, например

```
<property name="s" value = "Hello!"/>.
```

Рассмотрим еще один пример.

Пусть необходимо создать программу, формирующую детские подарки. В подарок можно включать различные конфеты, шоколад, фрукты, игрушки.

В нашей программе в подарок(gift) может входить два вида шоколада: белый и темный, два вида игрушек: машина и медведь. У всех подарков есть один общий параметр – цена. Шоколад характеризуется названием, долей какао. Игрушка характеризуется размером (примерный объем) и типом (машина, мягкая игрушка). Формирование подарка осуществляется в xml файле. Файл конфигурации будет выглядеть следующим образом.

newSpringXMLConfig.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
">
    <bean id="Car" class="sweets.Car">
        <constructor-arg value  = "50"/>
        <constructor-arg value  = " Car Volga"/>
        <constructor-arg value  = "150.50"/>
        <property name ="price" value = "150.5"/>
    </bean>
    <bean id="TeddyBear" class="sweets.TeddyBear">
        <constructor-arg value  = "100"/>
        <constructor-arg value  = "gray"/>
        <constructor-arg value  = "250.50"/>
        <property name ="price" value = "250.5"/>
    </bean>
    <bean id="WhiteChocolate" class="sweets.WhiteChocolate">
        <constructor-arg value  = "60"/>
        <constructor-arg value  = "Alenka"/>
        <constructor-arg value  = "90.50"/>
        <property name ="price" value = "90.5"/>
    </bean>
    <bean id="BlackChocolate" class="sweets.BlackChocolate">
        <constructor-arg value  = "80"/>
        <constructor-arg value  = "Russian"/>
        <constructor-arg value  = "70"/>
    </bean>
        <bean id="gift1" class="sweets.NewYearGift">
        <property name ="list" >
            <list>
                <ref bean="WhiteChocolate"/>
                <ref bean="TeddyBear"/>
            </list>
         </property>
      </bean>
</beans>
```

Первые четыре бина описывают возможные составляющие подарка: игрушка-машина, игрушка-медвежонок, шоколад - белый, шоколад – черный. В последнем бине формируется сам подарок, который задается списком типа gift. В список включены бины WhiteChocolate и TeddyBear.

Приведем тексты классов программы.

```
package sweets;
public interface Gift {
    public double price();
}
```

```
package sweets;
public interface Toy extends
Gift{
    public int size();
    public String type();
}
```

```
package sweets;
public interface Chocolate
extends Gift{
    public double cacao();
    public String name();
}
```

```
package sweets;
public class TeddyBear implements
Toy{
    private int size;
    private String type;

    private double price;
    public TeddyBear(int size,
String type
, double price) {
        this.size = size;
        this.type = type;
        this.price = price;
    }
    public void setPrice(double
price) {
        this.price = price+50;
    }
    public void setSize(int size) {
        this.size = size;
    }
    public void setType(String type)
{       this.type = type;    }
    @Override
    public int size() {
        return size;
    }
    @Override
    public String type() {
        return type;
    }
    @Override
    public double price() {
        return price;
    }
    @Override
```

```
package sweets;
public class TeddyBear implements
Toy{
    private int size;
    private String type;
    private double price;
    public TeddyBear(int size,
String type, double price) {
        this.size = size;
        this.type = type;
        this.price = price;
    }
    public void setPrice(double
price) {
        this.price = price+50;
    }
    public void setSize(int size) {
        this.size = size;
    }
    public void setType(String type)
{       this.type = type;    }
    @Override
    public int size() {
       return size;
    }
    @Override
    public String type() {
        return type;
    }
    @Override
    public double price() {
        return price;
    }
    @Override
    public String toString() {
        return "TeddyBear{" +
```

```
    public String toString() {
        return     "TeddyBear{"     +
"size="
+ size + ", type="
+ type + ", price=" + price + '}';
    }
}
```

```
"size=" + size + ", type=" + type +
", price=" + price + '}';
    }

}
```

```
package sweets;

public class BlackChocolate
implements Chocolate{
    private double cacao;
    private String name;
    private double price;
    public      BlackChocolate(double
cacao, String name, double price) {
        this.cacao = cacao;
        this.name = name;
        this.price = price;
    }
    public    void    setCacao(double
cacao) { this.cacao = cacao;    }
    public void setName(String name)
{   this.name = name;    }
    public    void    setPrice(double
price) {this.price = price+50;    }
    @Override
    public double cacao() {
        return cacao;
    }
    @Override
    public String name() {
        return name;
    }
    @Override
    public double price() {
        return price;
    }
    @Override
    public String toString() {
        return "BlackChocolate{"
                + "cacao=" + cacao
                + ", name=" + name +
                ", price=" + price +
'}';
    }
}
```

```
package sweets;

public class WhiteChocolate
implements Chocolate{
    private double cacao;
    private String name;
    private double price;
    public      WhiteChocolate(double
cacao,String name, double price) {
        this.cacao = cacao;
        this.name = name;
        this.price = price;
    }
    public    void    setCacao(double
cacao) {this.cacao = cacao; }
    public void setName(String name)
{  this.name = name;    }
    public    void    setPrice(double
price) {this.price = price+100;   }
    @Override
    public double cacao() {
        return cacao;
    }
    @Override
    public String name() {
        return name;
    }
    @Override
    public double price() {
        return price;
    }
    @Override
    public String toString() {
        return "WhiteChocolate{"
                + "cacao="
                + cacao + ","
                + " name=" + name
                + ", price=" +
                price + '}';
    }
}
```

В классе NewYearGift данным является список подарков. Этот список может быть сформирован методом setList. В конфигурационном файле он формируется из бинов WhiteChocolate и TeddyBear как показано ниже.

```
      bean gift1
        <bean id="gift1" class="sweets.NewYearGift">
        <property name ="list" >
            <list>
                <ref bean="WhiteChocolate"/>
                <ref bean="TeddyBear"/>
```

```
            </list>
        </property>
    </bean>
```

Приведем код класса NewYearGift.

NewYearGift

```
import java.util.ArrayList;
public class NewYearGift {
    private ArrayList<Gift> list = new ArrayList<Gift>();
    public void setList(ArrayList<Gift> list) {
        this.list = list;
    }
    public NewYearGift() {    }
    public void print()
    {
        double sum = 0;
        String s="";
        for (Gift g: list)
        {
            s+=g+"\n";
            sum+=g.price();
        }
        System.out.println(s+"Total price: "+sum);
    }
}
```

В результате работы программы получится следующее.

Результат работы программы

```
First present:
WhiteChocolate{cacao=60.0, name=Alenka, price=190.5}
TeddyBear{size=100, type=gray, price=300.5}
Total price: 491.0
```

Далее мы рассмотрим применение фреймворка Spring для построения корпоративных приложений.

Задания к гл.2

1. Создать программу, описывающую битвы войн с Наполеоном (Аустерлиц, Трафальгарское сражение, Бородинская битва, Взятие Парижа, Ватерлоо. В описание битвы включить: полководца - победителя, дату, место. Формирование списка битв осуществить в xml файле.
2. Создать программу, комплектующую одежду олимпийского волонтера. В

комплект можно включать различные куртки, шапки, шарфы и т.д.

Формирование комплекта осуществить в xml файле.

3. Создайте иерархию классов, в которой метод execute (int a,int b) одного из

классов выполнял одно из арифметических действий (+,-,*,/). Определение действия задается в xml файле.

3. Многоуровневая архитектура приложения

Используя многоуровневую архитектуру приложения, можно создавать гибкие и повторно-используемые приложения. Разделяя приложение на уровни абстракции, разработчики приобретают возможность внесения изменений в какой-то определённый слой, вместо того, чтобы перерабатывать всё приложение целиком.

Многоуровневую архитектуру приложеня описывает архитектурный паттерн Layers (Слои). Он помогает структурировать приложения разложением на группы подзадач, находящихся на определенных уровнях абстракции [5].

В логически разделённых на слои архитектурах информационных систем наиболее часто встречаются следующие четыре слоя (см. рис.6):

- Слой представления, с которым взаимодействует пользователь или клиентский сервис. Реализацией слоя представления может быть, например, графический пользовательский интерфейс или веб-страница.
- Сервисный слой, реализующий взаимодействие между слоями представления и бизнес-логики. Примерами реализаций сервисного слоя являются контроллеры, веб-сервисы и слушатели очередей сообщений.
- Слой бизнес-логики, в котором реализуется основная логика проекта. Компоненты, реализующие бизнес-логику, обрабатывают запросы, поступающие от компонентов сервисного слоя, а также используют компоненты слоя доступа к данным для обращения к источникам данных.
- Слой интеграции — набор компонентов для доступа к хранимым данным. В качестве источников данных могут выступать различного рода базы данных, SOAP и REST-сервисы, файлы на файловой системе и т.д.

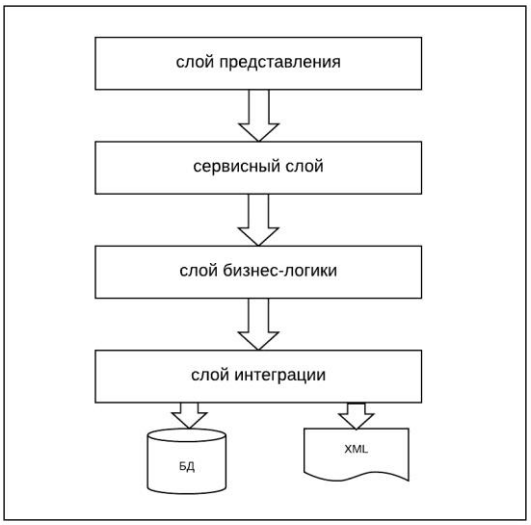


Рисунок 6. Слои сервисов в архитектуре приложения

Направление зависимостей между слоями идёт от слоя представления к слою доступа к данным (интеграции). В идеальной ситуации каждый слой зависит только от следующего слоя: слой представления зависит от сервисного слоя (например, представление зависит от контроллера), сервисный слой зависит от слоя бизнес-логики (например, контроллер зависит от бизнес-сервиса), а слой бизнес-логики — от слоя доступа к данным (например, бизнес-сервис зависит от репозитория). При этом компоненты бизнес-слоя могут зависеть от других компонентов бизнес-слоя, тогда как в других слоях аналогичные зависимости нежелательны (например, зависимость одного репозитория от другого). Так же нежелательны зависимости в обратном направлении (бизнес-слой не должен зависеть от сервисного слоя) и зависимости между слоями, не являющимися соседними (сервисный слой не должен зависеть от слоя доступа к данным, например).

Каждый слой зависит только от нижележащего слоя и может существовать без вышерасположенных слоёв. Ещё одна распространённая точка зрения заключается в том, что слои не всегда строго зависят от слоя, расположенного непосредственно под ними. Например, в нестрогой

многослойной системе (англ. a relaxed layered system) какой-то слой может зависеть от всех расположенных ниже слоёв [2].

В реальных приложениях иногда разделение на слои представлено не совсем четко. Бывает слой бизнес-логики частично или полностью находится в контроллере, а компоненты слоя представления обращаются к слою доступа к данным.

Несоблюдение разделения кода между слоями непременно приводит к путанице, замедляет процесс разработки и развития проекта и делает процесс поддержки проекта трудоёмким и дорогим.

Слой бизнес-логики является самой важной составляющей проекта. И именно с проработки компонентов слоя бизнес-логики должна начинаться разработка проекта.

Давайте создадим свое приложение, в котором будут присутствовать все четые слоя сервисов. Тестовое приложение будет получать доступ к БД sample DB derby, читать таблицу customer и выводить всех customer’ов, имена которых начинаются с заданной буквы (префикса).

Разделим наше приложение на слои.

Слой бизнес-логики – получение всех customer’ов, имена которых начинаются с заданной буквы.

Слой доступа к данным – релизация соединения с DB derby (sample), чтение данных из таблицы customer.

Сервисный слой – web-севис, реализующий получение всех customer’ов, имена которых начинаются с заданной буквы

Слой представления – jsp страница, отображающая всех customer’ов, имена которых начинаются с заданной буквы.

В разрабатываемом приложении может отсутствовать слой представления или слой сервисов.

Рассмотрим простую структуру приложения, в которой отсутствуют слои представления и сервисов. Структура проекта показана на рис.7.

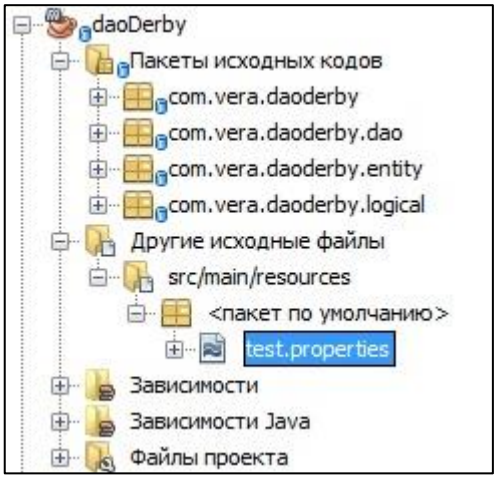


Рисунок 7. Структура тестового приложения

Пакет logical реализует бизнес-логику проекта.

Пакеты entity и dao реализуют слой доступа к данным.

Файл test. property описывает способ соединения с DB derby.

Приведенная структура приложения с некоторми дополнениями и незначительными изменениями будет использована во всех дальнейших примерах данной кнги.

Задания к гл. 3

1. Создайте в БД Postgres 3 таблицы, реализующие связь многие ко многим.
2. Создайте структуру серверного приложения для работы с БД Postgres.Приложение должно выполнять CRUD операции со всеми таблицами.

4. Паттерн DAO (data access object)

Data Access Object (DAO) – Объект Доступа к данным. Шаблон DAO используется для того, чтобы отделить логику данных в отдельном слое. Таким образом, обслуживание данных, относящееся к верхнему уровню, остается в полном неведении о том, как сделаны операции низкого уровня, чтобы получить доступ к базе данных. Это известно, как принцип разделения логики.

Этот шаблон проектирования применим к множеству языков программирования, большинству программного обеспечения, нуждающемуся в хранении информации и к большей части баз данных. Этот шаблон появился в рекомендациях от фирмы Sun Microsystems для приложений на платформе Java Enterprise Edition, взаимодействующими с реляционными базами данных через интерфейс JDBC. Диаграмма классов DAO приведена на рис 8.

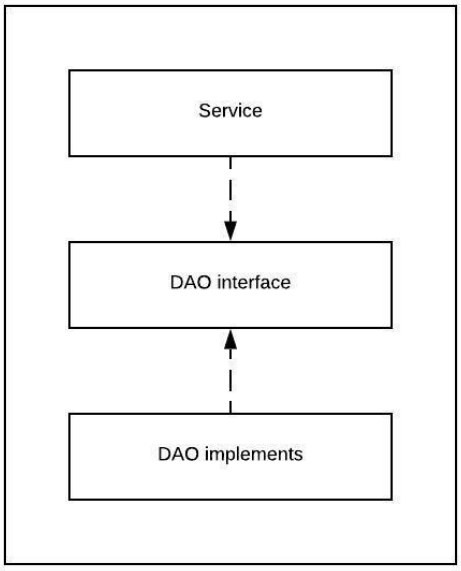


Рисунок 8. UML- диаграмма классов DAO

Для реализаций доступа к различным БД создали шаблон. Данные могут быть разбросаны по разным источникам. На момент создания шаблона не было ORM технологий, т.е. еще не придумали инкапсуляцию данных в виде объектов. Создание DAO interface, позволяет создавать различную

реализацию доступа к данным, а именно доступ к БД, доступ к файлам и т.д. Заметим, что при появлении ORM эта технология вобрала в себя DAO.

В шаблоне DAO есть следующие компоненты, от которых зависит дизайн:

- Модель, которая передает данные от одного слоя к другому.
- Интерфейс, который обеспечивает гибкий дизайн.
- Реализация интерфейса, которая является конкретной реализацией доступа к данным.

Рассмотрим пример с DAO без использования Hibernate (см. рис. 9).

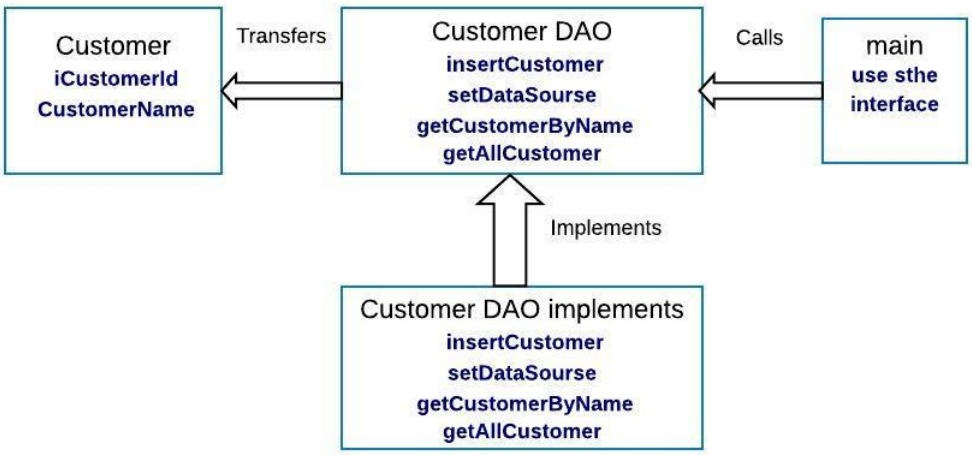


Рисунок 9. Диаграмма DAO без использования Hibernate

Напишем программу, которая читает данные из таблицы customer БД derby.

Разобьем нашу программу на логические части, соответствующие схеме на рис.7. Примеры можно скачать с https://bitbucket.org/w2lvera/daosimplederby/src/master/. Описание примеров приведено в приложении 3.

Пакеты com.vera.daoderby.dao и com.vera.daoderby.entity соответствуют слою интеграции, а com.vera.daoderby.logical слою бизнес логики.

Создадим класс сущности customer (DAO pattern model class)

Customer
`package com.vera.daoderby.entity;` `public class Customer {`

```
        private int id;
        private String name;
        public Customer(int id, String name) {
            this.id = id;
            this.name = name;
        }
        public Customer() {    }
        public int getId() {
            return id;    }
        public void setId(int id) {
            this.id = id;    }
        public String getName() {
            return name;    }
        public void setName(String name) {
            this.name = name;    }
        @Override
        public String toString() {
            return "Customer{" + "id=" + id + ", name=" + name + '}';
        }
    }
```

Создадим интерфейс DAO (DAO pattern interface).

CustomerDao

```
public interface CustomerDao {
    void insertCustomer(Customer t);
    void setDataSourse(Connection c);
    Customer getCustomerByName(String name);
    List<Customer> getAllCustomer();
}
```

и его реализацию (DAO Pattern Implementation)

CustomerDaoImpl

```
public class CustomerDaoImpl implements CustomerDao {

    private Connection dataSource;

    @Override
    public void setDataSourse(Connection c) {
        this.dataSource = c;
    }

    @Override
    public Customer getCustomerByName(String name) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    public void insertCustomer(Customer t) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    public List<Customer> getAllCustomer() {
        String sql = "SELECT * FROM Customer";
        Connection conn = null;
        try {
            conn = dataSource;
            PreparedStatement ps = conn.prepareStatement(sql);
            List<Customer> customers = new ArrayList<Customer>();
```

```
            ResultSet rs = ps.executeQuery();
            while (rs.next()) {
                Customer town = new Customer(
                        rs.getInt("CUSTOMER_ID"),
                        rs.getString("NAME")
                );
                customers.add(town);
            }
            rs.close();
            ps.close();
            return customers;
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                }
            }
        }
    }
}
```

В интерфайс CustomerDao вошел метод передачи соединения с БД – setDataSourse. Метод getAllCustomer возвращает список всех клиентов, извлеченных методом executeQuery из объекта типа PreparedStatement(служит для выполнения SQL запросов). Метод getICostomer (String s) отличается от getAllCustomer предыдущего только тем, что выполняется sql запрос, переданный в строке s.

Бизнес логика поддерживается интерфейсом CustomerListProcessor.

CustomerListProcessor

```
public interface CustomerListProcessor {
    List<Customer> getCustomerList(String prefix);
}
```

Реализация интерфейса заключается в том, что мы хотим возвратить список клиентов, начинающихся с префикса, переданного в строке параметров метода getCustomerList.

CustomerListProcessorImpl

```
package com.vera.daoderby.logical;
import com.vera.daoderby.dao.CustomerDao;
import com.vera.daoderby.entity.Customer;
import java.util.ArrayList;
import java.util.List;

public class CustomerListProcessorImpl implements CustomerListProcessor {

    @Override
    public List<Customer> getCustomerList(String prefix) {
```

```
        final List<Customer> allCustomers = customerDao.getAllCustomer();
        final List<Customer> result = new ArrayList<Customer>();
        for (Customer customer : allCustomers) {
            if (customer.getName().startsWith(prefix)) {
                result.add(customer);
            }
        }
        return result;
    }
    private CustomerDao customerDao;

    public void setCustomerDao(CustomerDao customerDao) {
        this.customerDao = customerDao;
    }
}
```

Осталось описать класс **Controller**, который создает все объекты и устанавливает соединение с БД.

Controller

```
public class Controller {
    Connector connection;
    CustomerListProcessorImpl customerListProcessorImpl;
    CustomerDao customerDao;

    public Controller() {
        connection = new Connector();
        try {
            connection.initialize();
        } catch (IOException ex) {
            System.out.println("no connection");
            Logger.getLogger(Controller.class.getName()).log(Level.SEVERE,
null, ex);
        }
        customerDao  = new CustomerDaoImpl();
        customerDao.setDataSourse(connection.getConnection());
        customerListProcessorImpl = new CustomerListProcessorImpl();
        customerListProcessorImpl.setCustomerDao(customerDao);
    }
    public List<Customer> getListCustomer(String s){
         return customerListProcessorImpl.getCustomerList(s);
    }
    public void printList(String prefix){
        String result = "";
        ArrayList<Customer> list = (ArrayList)getListCustomer(prefix);
        for(Customer x:list)result+= x+"\n";
        System.out.println(result);
    }
}
```

Обратим внимание на класс Connector. Обычно все проблемы начинаются в методе initialize().

Connector

```
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;

public class Connector {
```

```
    protected Connection connection = null;

    public Connector() {
    }

    public Connection getConnection() {
        return connection;
    }

    public void initialize() throws IOException{

    Properties defaultProps = new Properties();

defaultProps.load(Connector.class.getResourceAsStream("/test.properties"));

    String url=defaultProps.getProperty("url");
    String pasport =defaultProps.getProperty("pasvord") ;
    String driver =defaultProps.getProperty("driver") ;
    String parol = defaultProps.getProperty("parol");

        try{
              System.out.println("Start");
              Class.forName(driver);
          }
        catch(Exception e) {
              System.out.println("Class def not found: " + e);
          }

        try {
              connection = DriverManager.getConnection(url,pasport,parol);
          }
        catch (Throwable theException){
            theException.printStackTrace();
          }
  }
}
```

В файле test.properties хранятся характеристики БД (url, passport, driver, parol).

test.properties

```
driver = org.apache.derby.jdbc.ClientDriver
url = jdbc:derby://localhost:1527/sample
pasvord = app
parol = app
```

Файл test.properties хранится в \src\main\resources (path="/ daoDerby "/). Чтобы присоединить нужный драйвер создайте зависимость в pom.xml с требуемой версией derby.

pom.xml

```
<dependency>
      <groupId>org.apache.derby</groupId>
      <artifactId>derbyclient</artifactId>
      <version>10.8.1.2 </version>
      <type>jar</type>
</dependency>
```

Создадим объект **controller.**

main

```
public static void main(String[] args) {
        Controller controller = new Controller();
        controller.printList("J");
    }
```

В результате выполнения метода controller.printList("J") выведутся все клиенты с префиксом "J".

результат

```
Customer{id=1, name=Jumbo Eagle Corp}
Customer{id=149, name=John Valley Computers}
```

Есть много преимуществ при использовании паттерна DAO. Рассмотрим некоторые из них.

При изменении механизм доступа к данным, сервисный слой не должен знать, откуда появляются данные. Например, если перейти от использования derby к MongoDB, все изменения должны быть сделаны только в слое DAO.

DAO обеспечивает малую связь между различными компонентами приложения. Так, слой View (представления) не имеет никакой зависимости от слоя DAO, и только слой сервисов как-то от него зависит. Но и это связано с интерфейсами, а не с их конкретной реализацией.

Поскольку логика доступа к данным отделена DAO, намного легче написать модульные тесты на отдельные компоненты. Например, если при использовании JUnit для тестирования, будет легко заменять отдельные компоненты приложения.

Поскольку мы работаем с интерфейсами в DAO, это подчеркивает стиль “работы с интерфейсами вместо внедрения”, которое является превосходным стилем ООП программирования.

Как можно избежать создания класса Controller?

Создадим spring проект – daoDerbySpring.

По сравнению с предыдущим проектом в текущем проекте отсутствуют классы Connector и Controller, но появился файл SpringXMLConfig, лежащий в каталоге \src\main\resources (path="/ daoDerbyXML "/).

SpringXMLConfig

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

       xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
">
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
            <property name="driverClassName"
value="org.apache.derby.jdbc.ClientDriver" />
      <property name="url" value="jdbc:derby://localhost:1527/sample" />
      <property name="username" value="app" />
      <property name="password" value="app" />
</bean>
<bean id="customerDao" class="com.vera.daoderbyxml.dao.CustomerDaoImpl">
            <property name="dataSource" ref="dataSource" />

</bean>
<bean id="customerProcessor"
class="com.vera.daoderbyxml.logical.CustomerListProcessorImpl">
       <property name="customerDao"  ref="customerDao"/>
   </bean>
</beans>
```

В бине dataSource описывается соединение с БД. При этом используется DataSource из org. springframework.jdbc.datasource.DriverManagerDataSource.

В бине customerDao описывается реализация **CustomerDaoImpl** и свойство dataSource ссылается на бин dataSource.

В классе CustomerDaoImpl появится объект private DataSource dataSource вместо private Connection dataSource и соответствующий метод setDataSource.

DataSource

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

Класс Main данного проекта будет выглядеть следующим образом.

Класс Main

```
import com.vera.daoderbyxml.entity.Customer;
import com.vera.daoderbyxml.logical.CustomerListProcessor;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Properties;
import java.sql.DriverManager;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Main {

    public static void main(String[] args) throws IOException {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("SpringXMLConfig.xml");
        CustomerListProcessor processor =
```

```
            (CustomerListProcessor) context.getBean("customerProcessor");
        String result = "";
        ArrayList<Customer> list =
        (ArrayList) processor.getCustomerList("J");
        for (Customer x : list) {
            result += x + "\n";
        }
        System.out.println(result);
    }
}
```

Примеры расположены в https://bitbucket.org/w2lvera/daosimplederby/src/master/.

В методе main данного класса при создании объекта ApplicationContext создадутся все объекты, описанные в SpringXMLConfig. Объекты создаются как singletone.

Таким образом, мы избежали рутинной работы по созданию соединения с БД и созданию объектов приложения.

Задания к гл. 4

Создайте серверное приложение для работы с БД Postgres. В БД должны быть 3 таблицы, реализующие связь многие ко многим. Приложение должно выполнять CRUD операции со всеми таблицами. Используйте JDBC, DAO. Параметры соединения с БД задать в файле DBproperties.properties.

Например, создать CRUD-приложение (Create,Read,Update,Delete), которое будет уметь:

1. Создавать пользователей (owner), а также искать их в базе данных по ID, обновлять их данные в базе, а также удалять из базы.
2. Присваивать пользователям объекты автомобилей (Auto). Создавать, редактировать, находить и удалять автомобили из базы данных.
3. Кроме того, приложение должно автоматически удалять "бесхозные" автомобили из БД. Т.е. при удалении пользователя, все принадлежащии ему автомобили также должны быть удалены из БД.

5. Паттерны доступа к данным (Data Source Architectural patterns)

Рассматривая предыдущие примеры, можно обратить внимание на то, что отображение содержимого таблицы customer в объект customer подчинялся некоторым правилам.

Можно выделить четыре шаблона доступа к данным – Table Data Gateway, Row Data Gateway, Active Record, Data Maper.

Рассмотрим эти паттерны.

5.1 Table Data Gateway

Объект шлюза к таблице содержит все запросы SQL для доступа к отдельной таблице (для получения recordSet). Остальной код, для взаимодействия с БД, обращается к методам объекта шлюза. Это самое простое решение.

Характеристики Table Data Gateway

- Содержит методы поиска и CRUD операции
- Не имеет состояний – передает данные из/в БД.
- Методы поиска возвращают множества
- Идеально для вызова хранимых процедур
- Объект не работает с одной записью

Практически этот паттерн реализован в примере предыдущей главы.

5.2 Row Data Gateway

Объект выполняет роль шлюза к отдельной записи БД – каждой соответствует свой экземпляр.

Характеристики

- Каждый объект полностью повторяет одну запись в БД.
- Включает только логику доступа к БД.
- Методы поиска в отдельном классе.
- Может выполнять преобразования типов
- Идеально подходит для «сценария транзакции»

- Должен иметь методы insert, delete, update и select

Рассмотрим пример программы, которая читает данные из таблицы customer БД derby. Это измененный пример из главы DAO.

Опишем класс Customer.

```
      Customer
public class Customer  {
    private Integer customerId;
    private String name;
    private String addressline1;
    private String addressline2;
    private String city;
    private String state;
    private String phone;
    private String fax;
    private String email;
    private Integer creditLimit;
    private  static Connection dataSource=null;
      и т.д. с set и get
```

Опишем метод поиска клиента по id.

```
      метод поиска клиента по id
private final static String findCustomerString = "SELECT * FROM Customer
WHERE CUSTOMER_ID = ?";
    public static Customer findId(long id){
        Customer result=null;
        ResultSet rs= null;
        PreparedStatement ps = null;

        try {
            ps = dataSource.prepareStatement(findCustomerString);
            ps.setLong(1, id);
            rs= ps.executeQuery();
            rs.next();
            result = load(rs);
            return result;
        } catch (SQLException ex) {
            Logger.getLogger(Customer.class.getName()).log(Level.SEVERE,
null, ex);
        }
        finally{
            if (dataSource != null) {
                try {
                    dataSource.close();
                } catch (SQLException e) {
                }

            }
        }
        return result;
    }
    private static Customer load(ResultSet rs) throws SQLException {
        Customer result=null;
        result = new Customer(
                rs.getInt("CUSTOMER_ID"),
                rs.getString("NAME"),
                rs.getString("CITY"),
```

```
                rs.getString("PHONE"),
                rs.getString("EMAIL")
        );
        return result;
    }

    public Customer(Integer customerId, String name, String city, String
phone, String email) {
        this.customerId = customerId;
        this.name = name;
        this.city = city;
        this.phone = phone;
        this.email = email;
    }
```

результат

```
db.Customer[ customerId=1 ]Jumbo Eagle Corp Fort Lauderdale 305-555-0188
jumboeagle@example.com
```

Опишем метод вставки. В SQL строке есть непонятные символы 'N','48128'– это ссылки на другие таблицы БД «многие к одному». Мы взяли ссылки на существующие записи, чтобы не нарушать целостности.

insert

```
private final static String insertCustomerString = "INSERT INTO Customer
VALUES (101,'N','48128',?,?,?,?,?,?,?,?,?)";
public long insert(){
        PreparedStatement ps = null;
        try {
            ps = dataSource.prepareStatement(insertCustomerString);
            ps.setString(1, name);
            ps.setString(2, addressline1);
            ps.setString(3, addressline2);
            ps.setString(4, city);
            ps.setString(5, state);
            ps.setString(6, phone);
            ps.setString(7, fax);
            ps.setString(8, email);
            ps.setInt(9,creditLimit);
            ps.execute();
            return 0;
        } catch (SQLException ex) {
            Logger.getLogger(Customer.class.getName()).log(Level.SEVERE,
null, ex);

        }
         finally{
            if (dataSource != null) {
                try {
                    dataSource.close();
                } catch (SQLException e) {
                }

            }
        }
        return 1;
    }
```

5.3 Active Recorg

Отличается от предыдущего паттерна тем, что может включать бизнес локику.

5.4 Data Maper

Объектные и реляционные БД используют разные способы структурирования данных. Множество составляющих объектов, например коллекции и наследование, не представлены в реляционных БД. Когда проектируется объектная модель с большим количеством бизнес-логики, полезно применять такие механизмы для улучшения организации хранения данных и логики, которая работает с ними. Это приводит к различиям в организации. Так что объектная и реляционная схемы не идентичны.

Тем не менее, необходимость в обмене данными между двумя схемами не отпадает, и этот обмен становится, в свою очередь, сложным. Если же объект знает о реляционной структуре — изменения в одной из структур приведёт к проблемам в другой.

Data Mapper — это программная прослойка, разделяющая объект и БД. Его обязанность — пересылать данные между ними и изолировать их друг от друга. При использовании Data Mapper'a объекты не нуждаются в знании о существовании БД. Они не нуждаются в SQL-коде, и (естественно) в информации о структуре БД. Так как Data Mapper - это разновидность паттерна Mapper, сам объект-Mapper неизвестен объекту.

Характеристики

- Полная независимость бизнес-логики от БД.
- Объекты бизнес-логики и БД «не знают» о преобразователе
- Возможна поздняя загрузка объектов в память.
- Идеальна для сложной логики
- Трудно реализовать.

5.5 Фреймворки для доступа к данным

В явном виде реализация выше перечисленных паттернов встречается редко. В основном используют различные реализации JPA(Java Persistence API) или JDBC.

JPA – это спецификация, которая установила правила, как должно реализовываться ORM.

Как можно избежать рутинной работы при создании приложений доступа к БД? Это можно сделать с использованием фреймворков. Приведем таблицу, в которой показана функциональность фреймворков и их названия. Обратим внимание на то, что функциональность возрастает с каждой строкой таблицы.

Функциональность	Фреймворк	Особенности кода
• Открытие/закрытие связи с базой данных, обработку sql исключений. • Создание потоко-безопасных класов • Работа с транзакциями	Spring jdbcTemplate	Использование классов и интерфейсов **AnnotationConfigApplicationContext, HibernateSessionFactoryUtil, SessionFactory, RowMapper.** Создание точки входа в main **AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext (AppConfig.class);** Создание запросов с использованием интерфейса RowMapper **jdbcTemplate.query** **jdbcTemplate.update и т.д.**
• Отображениеобъектовв таблицы БД или наоборот. (ORM) • Hibernate Query Language (HQL), который позволяет выполнять SQL-подобные запросы, записанные рядом с объектами данных Hibernate.	Hibernate	Использование классов **HibernateSessionFactoryUtil и SessionFactory** Получение информации из базы **HibernateSessionFactoryUtil.getSessionFactory().openSession().createQuery ("From Customer").list()**
	Hibernate +Spring	Использование классов и интерфейсов **AnnotationConfigApplicationContext, HibernateSessionFactoryUtil и SessionFactory** Создание точки входа в main **AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext**

		`(AppConfig.class);`

• Создание и поддержание репозитория, созданного SpringandJPA.
• Поддержка QueryDSLandJPAзапросов.
• Аудит доменных классов.
• Поддержка пакетной загрузки, сортировки и динамических запросов.
• Поддержка XMLотображения для сущностей.
• Сокращение размера кода для CRUDопераций с помощью CrudRepository.
	Spring Data JPA	Использование классов и интерфейсов и аннотаций
		`AnnotationConfigApplicationContext, CrudRepository,@Repository`
		Создание точки входа в main
		`AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext (AppConfig.class);`
		Получение информации из базы
		`findByName(),findAll(),findOne()и т.д.`

Также приведенную таблицу можно расширить, добавив в нее фреймворки, связанные с web интерфейсом и с доступом к данным.

• Создание и поддержание репозитория, созданного SpringandJPA.
• Поддержка QueryDSLandJPAзапросов.
	Spring MVC обеспечивает архитектуру паттерна Model — View — Controller	Использование классов и интерфейсов и аннотаций
		`WebApplicationContext, HandlerMapping, Controller, ViewResolver, ModelMap, @Controller, @Repository, @RequestMapping`
		Создание точки входа в main
		`@Controller`
		`public class HelloController {`
		`@RequestMapping(value = "/",method = RequestMethod.GET)`
		`public String printWelcome(ModelMap model) {`
		`model.addAttribute("message", "Spring MVC - Hello World");`

• Аудит доменных классов.
• Поддержка пакетной загрузки, сортировки и динамических запросов.
• Поддержка XML отображения для сущностей.
• Сокращение размера кода для CRUD операций с помощью CrudRepository.
• Предоставление модели.
• Создание представления, контроллера.
		`return "hello";`
		`}`
		`}`

Объединяет вместе набор компонентов в готовое приложение. • Аннотации из SpringDataJPA • Создание и поддержание репозитория, созданного SpringJPA. • Embedded Tomcat • maven проект создается на start.spring.io	Spring boot	Использование классов и интерфейсов и аннотаций теже, что **`MVC.`** Создание точки входа в main **`SpringApplication.run`** – проще, чем в mvc

Рассмотрим приведенные выше фреймворки подробно.

Задания к гл. 5

1. Дополните проект из раздела 5.2 Row Data Gateway бизнес логикой, чтобы получить Active Recorg.

2. Создайте проект, реализующий шаблон Data Mapper , для доступа к одной таблице из DB Derby sumple.

6. Доступ к данным с использованием Hibernate и Spring

В явном виде реализация выше перечисленных паттернов встречается редко. В основном используют различные реализации JPA или JDBC. Все проекты данного раздела можно скачать с https://bitbucket.org/w2lvera/springdataaccessderby/src/master/.

6.1 Spring JDBC Template

JDBC требует много рутинного кода, такого как открытие/закрытие связи с базой данных, обработку sql исключений и т.д. Это делает программу чрезвычайно тяжелой и трудно читаемой.

Spring Framework берет эту работу на себя. Благодаря этому, работая с базой данных в Spring Framework, мы только должны определить параметры связи с БД и зарегистрировать вопрос SQL, остальная часть работы для нас выполнена Spring Framework.

У JDBC в Spring есть несколько классов для взаимодействия с базой данных. Наиболее распространенный из них использует класс JdbcTemplate. Это - базовый класс, который справляется с обработкой всех событий и соединениями с базой данных.

Класс JdbcTemplate выполняет SQL запросы, выводит результат в ResultSet, или выполняет запросы типа «update, delete, insert» или вызывает хранимые процедуры. «Ловит» исключения и переводит их в исключения, определенные в org.springframwork.dao пакете.

JdbcTemplate классы потоко-безопасны. Это означает, что, настраивая единственный случай класса JdbcTemplate, мы можем тогда использовать его для нескольких объектов DAO.

Чаще всего при использовании JdbcTemplate, его настройки пишутся в конфигурационном файле Spring. В силу этого очевидно использовать бины в DAO классах.

Шаблонный метод представляет основу алгоритма доступа к данным. Основа доступа к данным может быть представлена следующим набором действий.

- Создать коннект
- Открыть транзакцию
- Проверить данные
- Сохранить
- Закрыть транзакцию / Откат транзакции
- Закрыть коннект

Схема доступа к данным представлена на рис.10.

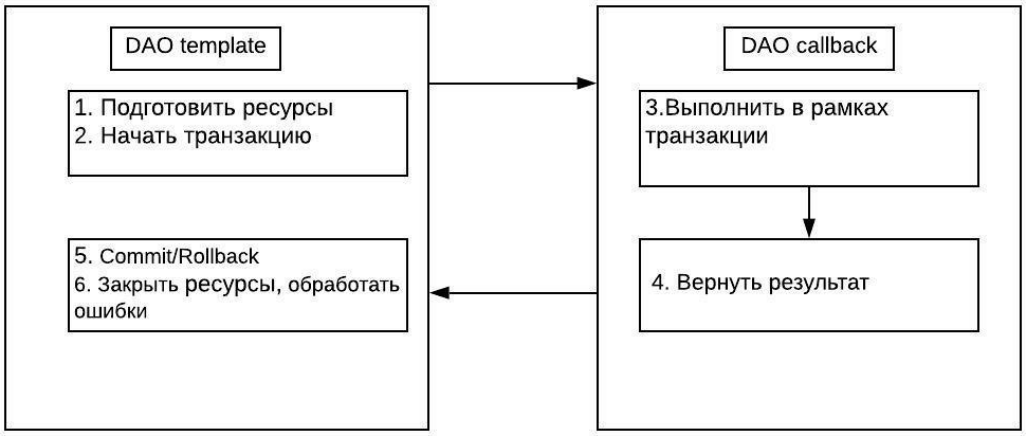


Рисунок 10. Схема доступа к данным JdbcTemplate

Рассмотрим пример.

Повторим все действия примера из главы DAO. Будем читать данные из таблицы customer (БД derby sample [app на APP]). Не будем добавлять записи и изменять записи в таблице Customer, т.к. не хочется изменять DB sumple. Рассмотрим действия insert и update позже на примере таблиц БД Postges.

Структура проекта в NetBeans показана на рис. 11.

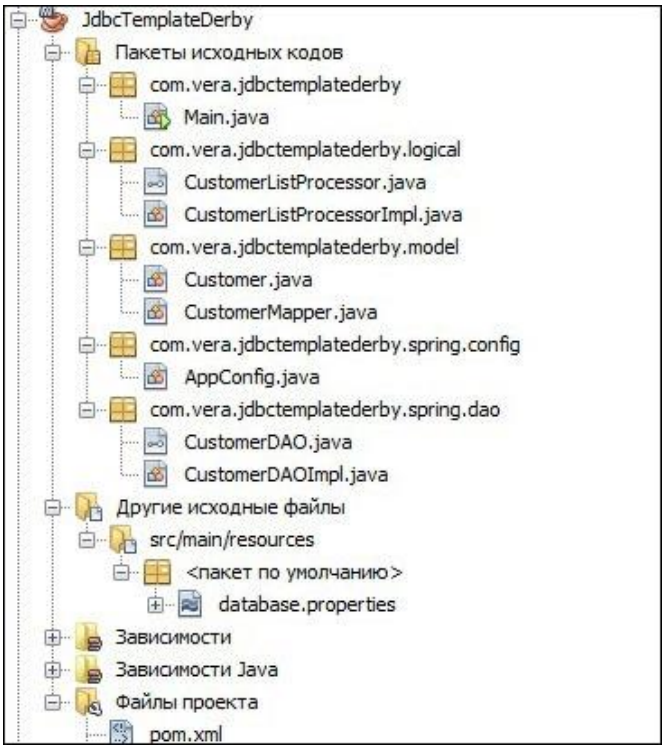


Рисунок 11. Структура проекта с использованием JdbcTemplate

Spring JDBC Maven Dependencies можно посмотреть в репозитории на bitbucket.org.

Конфигурация Spring записана с помощью аннотаций. Конфигурационный файл AppConfig представлен ниже.

AppConfig

```
import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

@Configuration
@ComponentScan(basePackages = {"com.vera.jdbctemplatederby.spring",
        "com.vera.jdbctemplatederby.logical"})

@PropertySource("classpath:database.properties")
public class AppConfig {

      @Autowired
      Environment environment;

      private final String URL = "url";
      private final String USER = "parol";
      private final String DRIVER = "driver";
      private final String PASSWORD = "password";

      @Bean
```

```
      DataSource dataSource() {

            DriverManagerDataSource driverManagerDataSource = new
            DriverManagerDataSource();
            driverManagerDataSource.setUrl(environment.getProperty(URL));

      driverManagerDataSource.setUsername(environment.getProperty(USER));

driverManagerDataSource.setPassword(environment.getProperty(PASSWORD));

      driverManagerDataSource.setDriverClassName(environment.getProperty(DRIVE
R));
            return driverManagerDataSource;
      }
}
```

@Configuration — аннотация которая используется для того чтобы сказать фреймворку Spring данный класс является конфигурационным. @ComponentScan — аннотация, которая используется вместе с аннотацией @Configuration и используем ее для определения пакета, в котором нужно искать классы компоненты.

@Bean — аннотация которая используется для того чтобы сказать фреймворку Spring что данный метод нужно использовать как бин для инжекта в классы компоненты.

@Autowired используется для injected объекта, т.е реализует IoC.

AppConfig использует файл свойств database.properties.

```
      database.properties
      driver = org.apache.derby.jdbc.ClientDriver
      url = jdbc:derby://localhost:1527/sample
      password = app
      parol = app
```

Итак, все готово для соединения с базой данных. Теперь необходимо создать модель данных, т.е. entity. Ниже представлен этот класс без set/get методов.

```
      Customer
public class Customer {
    private int id;
    private String name;
    public Customer(int id, String name) {
        this.id = id;
        this.name = name;
    }
    public Customer() {
    }
    @Override
    public String toString() {
```

```
        return "Customer{" + "id=" + id + ", name=" + name + '}';
    }
}
```

В модели представлен также класс CustomerMapper, наследующий интерфейс RowMapper, предоставленный Spring JDBC. Метод mapRow возвращает одну сущность модели, т.е. одного клиента. За счет этого не надо создавать ResultSet, ходить по записям с помощью next() и т.д.

CustomerMapper

```
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

public class CustomerMapper implements RowMapper<Customer> {

      public Customer mapRow(ResultSet rs, int i) throws SQLException {

            Customer person = new Customer();
                Customer customer = new Customer(
                        rs.getInt("CUSTOMER_ID"),
                        rs.getString("NAME")
                );

            return customer;
      }
}
```

В конце создадим DAO классы. Интерфейс CustomerDAO представлен ниже.

CustomerDAO

```
import java.util.List;
import com.vera.jdbctemplatederby.model.Customer;
public interface CustomerDAO {
    Customer getCustomerById(int id);
    List<Customer> getAllCustomers();
    boolean deleteCustomer(Customer customer);
    boolean updateCustomer(Customer customer);
    boolean createCustomer(Customer customer);
}
```

Класс CustomerDAOImpl является бином за счет аннотации @Component. Поле DataSource вставляется(injected) с использованием @Autowired аннотации.

CustomerDAOImpl

```
import java.util.List;
import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;
```

```
import com.vera.jdbctemplatederby.model.Customer;
import com.vera.jdbctemplatederby.model.CustomerMapper;

@Component
public class CustomerDAOImpl implements CustomerDAO {

    JdbcTemplate jdbcTemplate;

    private final String SQL_FIND_CUSTOMER =
    "select * from customer where customer_id = ?";

    private final String SQL_DELETE_CUSTOMER =
    "delete from customer where id = ?";

    private final String SQL_UPDATE_CUSTOMER =
    "update customer set  name = ? where customer_id = ?";

    private final String SQL_GET_ALL = "select * from customer";
    private final String SQL_INSERT_CUSTOMER =
    "insert into customer(customer_id, name) values(?,?)";

    @Autowired
    public CustomerDAOImpl(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public Customer getCustomerById(int id) {
        return jdbcTemplate.queryForObject(SQL_FIND_CUSTOMER,
        new Object[]{id}, new CustomerMapper());
    }

    public List<Customer> getAllCustomers() {
        return jdbcTemplate.query(SQL_GET_ALL, new CustomerMapper());
    }

    public boolean deleteCustomer(Customer customer) {
        return
        jdbcTemplate.update(SQL_DELETE_CUSTOMER, customer.getId()) > 0;
    }

    public boolean updateCustomer(Customer customer) {
        return jdbcTemplate.update(SQL_UPDATE_CUSTOMER, customer.getName(),
                customer.getId()) > 0;
    }

    public boolean createCustomer(Customer customer) {
        return jdbcTemplate.update
           (SQL_INSERT_CUSTOMER, customer.getId(), customer.getName()) > 0;
    }

}
```

С помощью объекта jdbcTemplate, созданного в конструкторе и настроенного на нашу БД, выполняются обычные SQL запросы, заданные строками вначале класса. Магию запросов jdbcTemplate мы не будем рассматривать подробно. В примере их логика очевидна.

Логика приложения представлена в пакете jdbctemplatederby.logical и мало отличается от логики в разделе DAO. Приведем только класс CustomerListProcessorImpl для иллюстрации определения бина.

CustomerListProcessorImpl

```
@Component
public class CustomerListProcessorImpl implements CustomerListProcessor {
     private CustomerDAO customerDAO;
     @Override
    public List<Customer> getCustomerList(String prefix) {
        final List<Customer> allCustomers = customerDAO.getAllCustomers();
        final List<Customer> result = new ArrayList<Customer>();
        for (Customer customer : allCustomers) {
            if (customer.getName().startsWith(prefix)) {
                result.add(customer);
            }
        }
        return result;
    }
    @Autowired
    public void setCustomerDAO(CustomerDAO customerDAO) {
        this.customerDAO = customerDAO;
    }
}
```

Класс Main для данного приложения

main

```
import com.vera.jdbctemplatederby.logical.CustomerListProcessor;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.vera.jdbctemplatederby.model.Customer;
import com.vera.jdbctemplatederby.spring.config.AppConfig;
import com.vera.jdbctemplatederby.spring.dao.CustomerDAO;
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context
                = new AnnotationConfigApplicationContext(AppConfig.class);

        CustomerDAO customerDAO = context.getBean(CustomerDAO.class);
        CustomerListProcessor customerListProcessor
                = context.getBean(CustomerListProcessor.class);
        ArrayList<Customer> list
                = (ArrayList) customerListProcessor.getCustomerList("J");
        String result = "";
        for (Customer x : list) {
            result += x + "\n";
        }
        System.out.println(result);
        System.out.println("List of person is:");
        for (Customer c : customerDAO.getAllCustomers()) {
            System.out.println(c);
        }
        System.out.println("\nGet person with ID 100");
        Customer customerById = customerDAO.getCustomerById(100);
```

```
        System.out.println(customerById);

        Customer cc = customerDAO.getCustomerById(100);
        cc.setName("CHANGED");
        customerDAO.updateCustomer(cc);

        System.out.println("\nList of customers is:");
        for (Customer c : customerDAO.getAllCustomers()) {
            System.out.println(c);
        }
        //customerDAO.deleteCustomer(cc);
        context.close();
  }
}
```

Результатом работы приложения будет следующее.

Результат работы

```
List of person with prefix J is:
Customer{id=1, name=Jumbo Eagle Corp}
Customer{id=149, name=John Valley Computers}

List of person is:
Customer{id=1, name=Jumbo Eagle Corp}
Customer{id=2, name=New Enterprises}
Customer{id=25, name=Wren Computers}
Customer{id=3, name=Small Bill Company}
Customer{id=36, name=Bob Hosting Corp.}
Customer{id=106, name=Early CentralComp}
Customer{id=149, name=John Valley Computers}
Customer{id=863, name=Big Network Systems}
Customer{id=777, name=West Valley Inc.}
Customer{id=753, name=Zed Motor Co}
Customer{id=722, name=Big Car Parts}
Customer{id=409, name=Old Media Productions}
Customer{id=410, name=Yankee Computer Repair Ltd}
Customer{id=100, name=CHANGED}
Customer{id=101, name=student}
Customer{id=102, name=student}

Get person with ID 100
Customer{id=100, name=CHANGED}

List of customers after change is:
Customer{id=1, name=Jumbo Eagle Corp}
Customer{id=2, name=New Enterprises}
Customer{id=25, name=Wren Computers}
Customer{id=3, name=Small Bill Company}
Customer{id=36, name=Bob Hosting Corp.}
Customer{id=106, name=Early CentralComp}
Customer{id=149, name=John Valley Computers}
Customer{id=863, name=Big Network Systems}
Customer{id=777, name=West Valley Inc.}
Customer{id=753, name=Zed Motor Co}
Customer{id=722, name=Big Car Parts}
Customer{id=409, name=Old Media Productions}
Customer{id=410, name=Yankee Computer Repair Ltd}
Customer{id=100, name=CHANGED}
Customer{id=101, name=student}
Customer{id=102, name=student}
```

Мы рассмотрели один из способов доступа к данным, который представлен в структуре Spring. В этой структуре есть способ доступа обозначенный ORM. Рассмотрим одну из реализаций ORM – Hibernate, как самую популярную.

6.2 Доступ к данным с Hibernate

Hibernate — самая популярная реализация спецификации JPA, предназначенная для решения задач объектно-реляционного отображения (ORM). Главная трудность при создании приложений с доступом к БД – как отобразить объекты в таблицы БД или наоборот. На этот вопрос отвечает ORM.

JPA – (Java Persistence API) спецификация API Java EE (Enterprise Edition),

JPA – это спецификация, которая установила правила, как должно реализовываться ORM. Более подробно о JPA можно посмотреть в [6].

Hibernate обеспечивает прозрачную поддержку сохранности данных (persistence) для «POJO» (то есть для стандартных Java-объектов); единственное строгое требование для сохраняемого класса — наличие конструктора по умолчанию (без параметров). Для корректного поведения в некоторых приложениях требуется также уделить внимание методам equals() и hashCode().

Mapping (сопоставление, проецирование) Java-классов с таблицами базы данных осуществляется с помощью конфигурационных XML-файлов или Java-аннотаций.

Обеспечиваются возможности по организации отношения между классами «один-ко-многим» и «многие-ко-многим». В дополнение к управлению связями между объектами Hibernate также может управлять рефлексивными отношениями, где объект имеет связь «один-ко-многим» с другими экземплярами своего собственного типа данных.

Hibernate обеспечивает использование SQL-подобного языка Hibernate Query Language (HQL), который позволяет выполнять SQL-подобные запросы, записанные рядом с объектами данных Hibernate.

Рассмотрим наш пример с доступом к таблице Customer.

Структура проекта в NetBeans показана на рис.12.

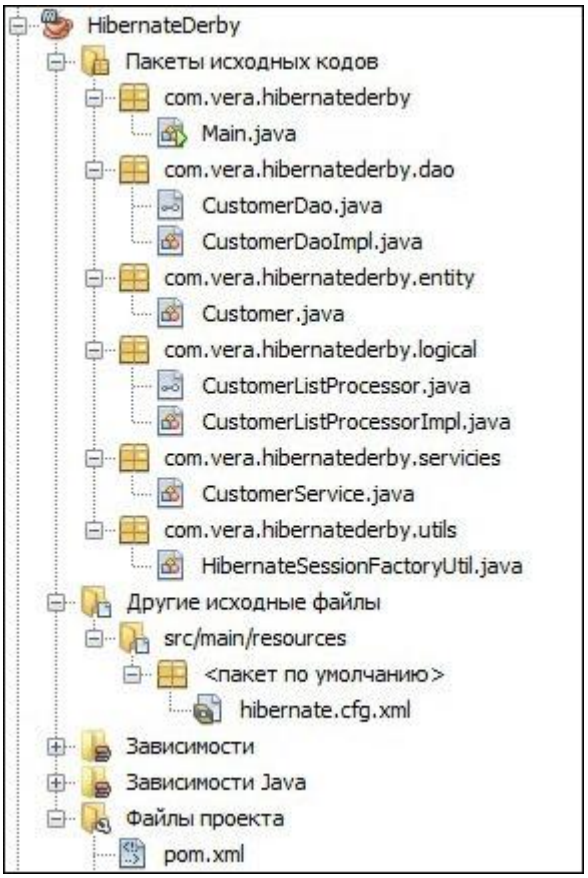


Рисунок 12. Структура проекта доступа к данным с использованием Hibernate

Maven Dependencies можно посмотреть в репозитории на bitbucket.org.

ORM Model Class представлен как сущность Customer. В примере не приведены set/get методы для того, что бы не загромождать текст.

Customer

```
import java.io.Serializable;
import javax.persistence.*;
@Entity
@Table(name = "CUSTOMER")
public class Customer implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @Column(name = "CUSTOMER_ID")
    private Integer customerId;
    @Column(name = "NAME")
```

```
    private String name;
    @Column(name = "ADDRESSLINE1")
    private String addressline1;
    @Column(name = "ADDRESSLINE2")
    private String addressline2;
    @Column(name = "CITY")
    private String city;
    @Column(name = "STATE")
    private String state;
    @Column(name = "PHONE")
    private String phone;
    @Column(name = "FAX")
    private String fax;
    @Column(name = "EMAIL")
    private String email;
    @Column(name = "CREDIT_LIMIT")
    private Integer creditLimit;

    @Override
    public String toString() {
        return  customerId + " "+name+" "+city;
    }

}
```

В классе Customer мы видим аннотации. Рассмотрим их.

- @Entity — говорит о том, что данный класс представляет собой сущность, которую надо сохранять в базе данных
- @Table — с параметром name, показывает, какая таблица используется для хранения
- @Id — говорит о том, что данное поле является идентификатором
- @GeneratedValue — поле с автонумерацией. Как задается нумерация говорит параметр GenerationType.IDENTITY. В данном случае значение будет создано с помощью базы данных. Если не задан параметр GenerationType , то по умолчанию параметр @GeneratedValue равен GenerationType.IDENTITY. Можно создавать сиквенсы прямо в БД. Но об этом позже.
- @Column — достаточно очевидная аннотация для указания имени столбца, с которым связано поле класса.
- Следующие аннотации мы рассмотрим позже.
- @ManyToOne — аннотация указывает, что поле указывает на другой класс, который связан с текущим классом связью многие-к-одному.

- @JoinColumn — данная анотация по сути похожа на @Column. Разве что она указывает, что колонка к тому же является связующей
- @OneToMany — такая запись говорит о том, что поле служить для связи один-ко-многим (потому и список).

Чтобы класс мог быть сущностью, к нему предъявляются следующие требования:

- Должен иметь пустой конструктор (public или protected);
- Не может быть вложенным, интерфейсом или enum;
- Не может быть final и не может содержать final-полей/свойств;
- Должен содержать хотя бы одно @Id-поле.

При этом entity может:

- Содержать непустые конструкторы;
- Наследоваться и быть наследованным;
- Содержать другие методы и реализовывать интерфейсы.

Если учесть, что сущность соответствует таблице из БД, то можно создать класс – сущность по готовой таблице. В NetBeans есть такая возможность.

После создания модели данных пора научить нашу программу выполнять с этими данными операции в БД. Начнем с утилитного класса HibernateSessionFactoryUtil. У него всего одна задача — создавать для нашего приложения фабрику сессий для работы с БД.

HibernateSessionFactoryUtil

```
public class HibernateSessionFactoryUtil {
    private static SessionFactory sessionFactory;

    private HibernateSessionFactoryUtil() {}

    public static SessionFactory getSessionFactory() {
        if (sessionFactory == null) {
            try {
                Configuration configuration = new
Configuration().configure();
                configuration.addAnnotatedClass(Customer.class);
                StandardServiceRegistryBuilder builder =
                        new
StandardServiceRegistryBuilder().applySettings(configuration.getProperties())
;
```

```
                sessionFactory =
configuration.buildSessionFactory(builder.build());

            } catch (Exception e) {
                System.out.println("Исключение!" + e);
            }
        }
        return sessionFactory;
    }
}
```

В этом классе мы создаем новый объект конфигураций Configuration, и передаем ему те классы, которые он должен воспринимать как сущности — Customer. Обратите внимание на метод configuration.getProperties(). Properties — это параметры для работы hibernate, указанные в специальном файле hibernate.cfg.xml.(см схему проекта).

В нем, как видите, нет ничего особенного — параметры соединения с БД, и специальный параметр show_sql. Он нужен для того, чтобы все sql-запросы, которые hibernate будет выполнять к нашей БД, выводились в консоль. Таким образом, вы будете видеть что конкретно делает Hibernate в каждый момент времени и избавитесь от эффекта "магии".

Создадим DAO, т.е. interface CustomerDao и class CustomerDaoImpl.

CustomerDao

```
public interface CustomerDao {
    Customer findById(int id);
    void update(Customer customer);
    void save(Customer customer);
    void delete(Customer customer);
    List<Customer> findAll();

}
```

Что же умеет наш класс UserDao? Собственно, как и все DAO, он умеет только работать с данными. Найти клиента по id, обновить его данные, удалить его, вытащить из БД список всех клиентов или сохранить в БД нового клиента — вот весь его функционал. Мы реализовали только один метод findAll() для поиска всех клиентов. Измениие клиентов не входит в нашу задачу.

Обратите внимание, что обращение к фабрике сессий и получение сессии прописано внутри метода findAll().

CustomerDaoImpl

```
public class CustomerDaoImpl implements CustomerDao {
    @Override
    public Customer findById(int id) {
       return
HibernateSessionFactoryUtil.getSessionFactory().openSession().get(Customer.cl
ass, id);
    }
    @Override
    public void update(Customer customer) {
        throw new UnsupportedOperationException("Not supported yet.");
    }
    @Override
    public void save(Customer customer) {
        throw new UnsupportedOperationException("Not supported yet.");
    }
    @Override
    public void delete(Customer customer) {
        throw new UnsupportedOperationException("Not supported yet.");
    }
    @Override
    public List<Customer> findAll() {
        List<Customer> customers = (List<Customer>)

HibernateSessionFactoryUtil.getSessionFactory().openSession().createQuery("Fr
om Customer").list();
        return customers;
    }
}
```

Именно DAO — "сердце" нашего приложения. Однако, мы не будем создавать DAO напрямую и вызывать его методы в нашем методе main().

Вся логика будет перемещена в класс CustomerService.

CustomerService

```
public class CustomerService {
private CustomerDao customerDao ;
    public CustomerService() {
        customerDao = new CustomerDaoImpl();
    }
    public Customer findCustomer(int id) {
        return customerDao.findById(id);
    }
    public void saveCustomer(Customer customer) {
        customerDao.save(customer);
    }
    public void deleteCustomer(Customer customer) {
       customerDao.delete(customer);
    }
    public void updateCustomer(Customer customer) {
        customerDao.update(customer);
    }
    public List<Customer> findAll() {
        return customerDao.findAll();
    }
}
```

Мы добавили еще один слой logic для имитации слоя бизнес-логики. В классе CustomerListProcessorImpl будем пользоваться CustomerService.

CustomerListProcessorImpl

```
public class CustomerListProcessorImpl implements CustomerListProcessor {

     private CustomerService customerService;
     private CustomerDao customerDao;
     @Override
    public List<Customer> getCustomerList(String prefix) {

        final List<Customer> allCustomers = customerService.findAll();
        final List<Customer> result = new ArrayList<Customer>();
        for (Customer customer : allCustomers) {
            if (customer.getName().startsWith(prefix)) {
                result.add(customer);
            }
        }
        return result;
    }
    @Override
    public void setCustomerService(CustomerService customerService) {
        this.customerService = customerService;
    }
    public Customer getCustomerId(int id) {
        return customerService.findUser(id);
    }
    public void setCustomerDao(CustomerDao customerDao) {
        this.customerDao = customerDao;
    }
}
```

В main создадим объект бизнес-логики CustomerListProcessorImpl. Установим CustomerService. Выполним поиск в БД всех клиентов с префиксом “J”.

Main

```
public class Main {

    public static void main(String[] args) {
        CustomerListProcessor processor
                = new CustomerListProcessorImpl();
        processor.setCustomerService(new CustomerService());
        ArrayList<Customer> list = (ArrayList<Customer>)
                            processor.getCustomerList("J");
        System.out.println();
        System.out.println();
        System.out.println("Customer with prefix J");
        for (Customer x : list) {
            System.out.println(x);
        }
        System.out.println("Customer 149");
        System.out.println(processor.getCustomerId(149));
    }
}
```

Получили результат.

```
Результат
Customer with prefix J
1 Jumbo Eagle Corp Fort Lauderdale
149 John Valley Computers Santa Clara
Customer 149
149 John Valley Computers Santa Clara
```

Конечно, мы увидели лишь небольшую часть функциональности Hibernate. Его возможности очень широки, и он давно является одним из промышленных стандартов Java-разработки. Если вы хотите изучить его во всех подробностях, обратитесь к книге[7].

6.3 Комбинация Spring Hibernate

Spring наиболее часто используемая Java EE Framework и Hibernate наиболее популярная реализация ORM. Поэтому комбинация Spring Hibernate часто используется в корпоративных приложениях.

В предыдущем проекте все используемые объекты создавались непосредственно в программе. Давайте создадим их с помощью Spring. Проект лежит на битв\букете.

Структура проекта в NetBeans показана на рис 13.

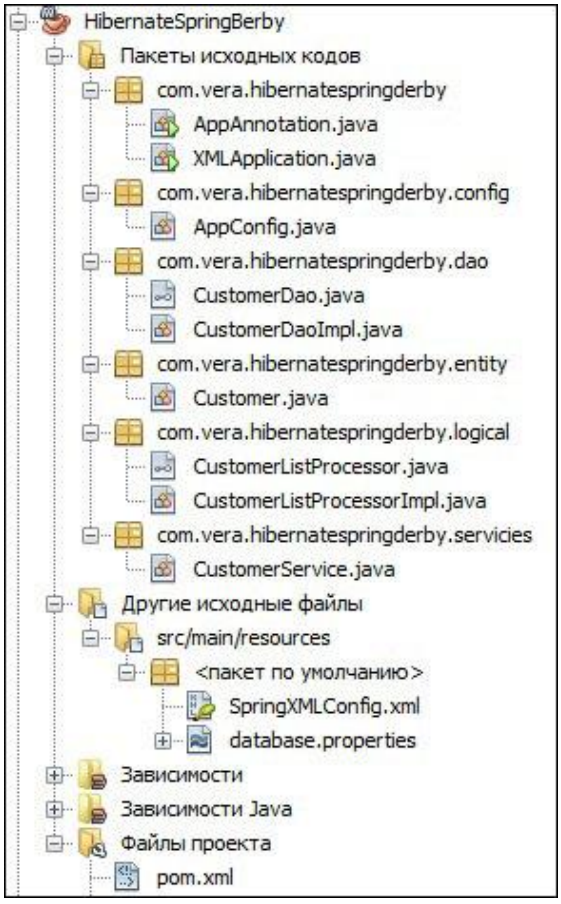


Рисунок 13. Структура проекта с использованием комбинации spring и hibernate.

Нам надо соединить описание spring бинов и создание фабрики сессий для работы с БД.

Создадим два способа описания Spring конфигурации:

- с помощью XML
- с помощью аннотаций.

Приведем XML конфигурацию – файл SpringXMLConfig.

SpringXMLConfig

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
">
    <bean id="dataSource" class=
        "org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value=
                 "org.apache.derby.jdbc.ClientDriver" />
        <property name="url" value="jdbc:derby://localhost:1527/sample" />
        <property name="username" value="app" />
        <property name="password" value="app" />
    </bean>
    <!-- Hibernate 4 Annotation SessionFactory Bean definition-->
    <bean id="sessionFactory"               class=
        "org.springframework.orm.hibernate4.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="annotatedClasses">
            <list>
              <value>com.vera.hibernatespringderby.entity.Customer</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
            <prop key="hibernate.dialect">
                  org.hibernate.dialect.DerbyDialect</prop>
            <prop key="hibernate.current_session_context_class">
                  thread</prop>
            <prop key="hibernate.show_sql">false</prop>
            </props>
        </property>
    </bean>

    <bean id="customerDAO" class=
          "com.vera.hibernatespringderby.dao.CustomerDaoImpl">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
    <bean id="customerService" class=
          "com.vera.hibernatespringderby.servicies.CustomerService">
        <property name="customerDao"  ref="customerDAO"/>
    </bean>
    <bean id="customerProcessor" class=
         "com.vera.hibernatespringderby.logical.CustomerListProcessorImpl">
        <property name="customerService"  ref="customerService"/>
    </bean>

</beans>
```

Приведем файл database.property. В него добавили поля для hibernate hibernate.dialect и hibernate.show_sql.

database.property

```
driver = org.apache.derby.jdbc.ClientDriver
url = jdbc:derby://localhost:1527/sample
password = app
parol = app
hibernate.show_sql = false
hibernate.dialect = org.hibernate.dialect.DerbyDialect
```

Фабрика сессий поддерживается бином sessionFactory класса org.springframework.jdbc.datasource.DriverManagerDataSource(для Hibernate 4).

Метод main для такой конфигурации выглядит следующим образом.

XMLApplication

```
public class XMLApplication {

    public static void main(String[] args) {
        ClassPathXmlApplicationContext context =
                new ClassPathXmlApplicationContext("SpringXMLConfig.xml");
        CustomerListProcessor processor =
                (CustomerListProcessor)
context.getBean("customerProcessor");
        ArrayList<Customer> list =
                (ArrayList<Customer>)processor.getCustomerList("J");
        System.out.println();
        System.out.println();
        System.out.println("Customer whith prefix J");
        for(Customer x:list)
            System.out.println(x);
        System.out.println("Customer 149");
        System.out.println(processor.getCustomerId(149));
    }
}
```

Теперь рассмотрим тоже, но с помощью аннотаций.

Тогда в проекте нужно создать класс **AppConfig**.

AppConfig

```
import com.vera.hibernatespringderby.entity.Customer;
import java.util.Properties;
import javax.sql.DataSource;
import org.hibernate.SessionFactory;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.hibernate4.LocalSessionFactoryBean;

@Configuration
@ComponentScan(
         basePackages = {"com.vera.hibernatespringderby.dao",
```

```
        "com.vera.hibernatespringderby.logical",
        "com.vera.hibernatespringderby.servicies"})
@PropertySource("classpath:database.properties")
public class AppConfig {

    @Autowired
    Environment environment;
    private final String URL = "url";
    private final String USER = "parol";
    private final String DRIVER = "driver";
    private final String PASSWORD = "password";
    private final String PROPERTY_SHOW_SQL = "hibernate.show_sql";
    private final String PROPERTY_DIALECT = "hibernate.dialect";

    @Bean
    DataSource dataSource() {
        DriverManagerDataSource driverManagerDataSource =
                                    new  DriverManagerDataSource();
        driverManagerDataSource.setUrl(environment.getProperty(URL));
        driverManagerDataSource.setUsername(environment.getProperty(USER));
        driverManagerDataSource.setPassword
                      (environment.getProperty(PASSWORD));
        driverManagerDataSource.setDriverClassName
                       (environment.getProperty(DRIVER));
        return driverManagerDataSource;
    }

    @Bean
    org.springframework.orm.hibernate4.LocalSessionFactoryBean
    sessionFactory() {
        org.springframework.orm.hibernate4.LocalSessionFactoryBean
        factoryBean = new
             org.springframework.orm.hibernate4.LocalSessionFactoryBean();
        factoryBean.setDataSource(dataSource());
        factoryBean.setPackagesToScan
                  ("com.vera.hibernatespringberby.entity");
        factoryBean.setAnnotatedClasses(new Class<?>[]{Customer.class});
        factoryBean.setHibernateProperties(hibernateProps());
        return factoryBean;
    }

    @Bean
    Properties hibernateProps() {
        Properties properties = new Properties();
        properties.setProperty
           (PROPERTY_DIALECT, environment.getProperty(PROPERTY_DIALECT));
        properties.setProperty
           (PROPERTY_SHOW_SQL, environment.getProperty(PROPERTY_SHOW_SQL));
        return properties;
    }
}
```

Можно заметить, что при описании бинов, реализующих системные классы, в классе AppConfig наблюдается полное соответствие файлу SpringXMLConfig. Откуда берутся описание бинов нашего приложения customerDAO, customerService и customerProcessor. Создание бинов инициализируется аннотацией @Component, внедрение объекта

реализуется аннотацией @Autowired. Ниже приведен фрагмент класса CustomerService с такой аннотацией.

CustomerService

```
@Component
public class CustomerService {

    private CustomerDao customerDao;
    @Autowired
    public void setCustomerDao(CustomerDao customerDao) {
        this.customerDao = customerDao;
    }
}
```

Метод main для такой конфигурации выглядит следующим образом. В нем используется AnnotationConfigApplicationContext для поднятия контекста приложения.

AppAnnotation

```
public class AppAnnotation {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
        CustomerListProcessor customerListProcessor =
context.getBean(CustomerListProcessorImpl.class);

        ArrayList<Customer> list =
(ArrayList<Customer>)customerListProcessor.getCustomerList("J");
        System.out.println();
        System.out.println();
        System.out.println("Customer whith prefix J");
        for(Customer x:list)
            System.out.println(x);
        System.out.println("Customer 149");
        System.out.println(customerListProcessor.getCustomerId(149));
    }
}
```

Получили результат.

результат

```
Customer with prefix J
1 Jumbo Eagle Corp Fort Lauderdale
149 John Valley Computers Santa Clara
Customer 149
149 John Valley Computers Santa Clara
```

6.4 Spring Data JPA

Spring Data делает проще создание приложений поддерживаемых Spring с доступом к данным, таким как нереляционные базы данных, map-reduction frameworks, облачные сервисы, а также хорошо как и к реляционным базам данных. В Spring Data вы можете использовать Hibernate,

Eclipse Link или любого другого поставщика JPA. Очень интересным преимуществом является то, что вы можете декларативно управлять границами транзакции, используя аннотацию @Transactional .

В Spring Data JPA мы используем следующее.

JPA: API персистентности Java, который предоставляет спецификацию для сохранения, чтения, управления данными из вашего Java-объекта и отношений в базе данных.

Hibernate: Есть различные провайдеры, которые реализуют jpa. Hibernate является одним из них. Так что у нас есть и другой провайдер. Но если использовать jpa с пружиной, это позволит вам в будущем переключаться на разных провайдеров.

Spring Data JPA: это еще один слой поверх jpa, который предоставляет Spring для облегчения вашей жизни.

Мы будем рассматривать Spring Data JPA, которая обеспечивает некоторые замечательные функции.

- Создание и поддержание репозитория, созданного Spring and JPA.
- Поддержка QueryDSL and JPA запросов.
- Аудит доменных классов.
- Поддержка пакетной загрузки, сортировки и динамических запросов.
- Поддержка XML отображения для сущностей.
- Сокращение размера кода для CRUD операций с помощью CrudRepository.

Рассмотрим наш традиционный пример.

Структура проекта в NetBeans показана на рис.14.

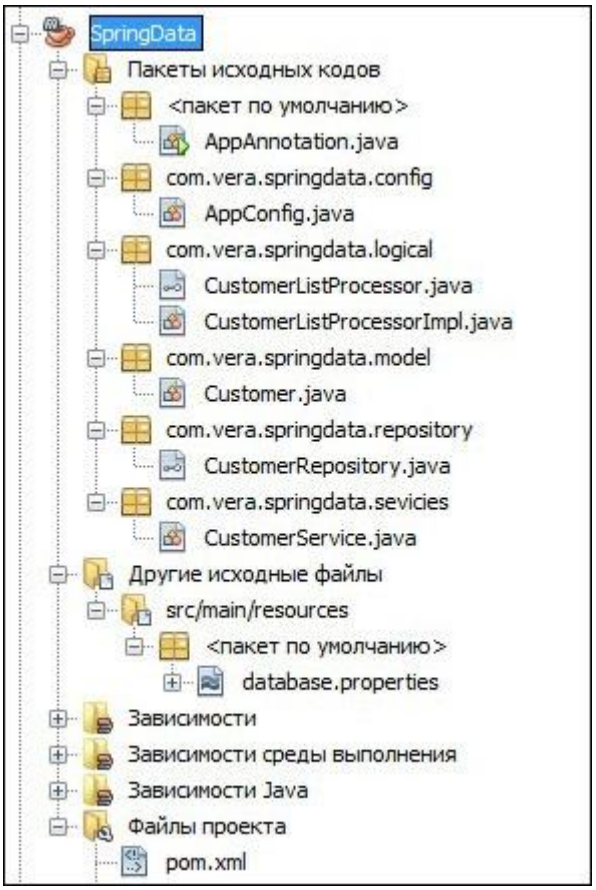


Рисунок 14. Структура проекта с использованием Spring Data

Spring Data JPA Maven Dependencies можно увидеть в репозитории битбакет.

Spring Configuration Classes приведен ниже.

Spring Configuration Classes

```
import java.util.Properties;
import javax.sql.DataSource;
import org.hibernate.ejb.HibernatePersistence;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;
import
org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import
org.springframework.transaction.annotation.EnableTransactionManagement;

@Configuration
@ComponentScan(basePackages = {
    "com.vera.springdata.logical", "com.vera.springdata.sevicies"})
@EnableJpaRepositories("com.vera.springdata.repository")
@EnableTransactionManagement
@PropertySource("classpath:database.properties")
```

```
public class AppConfig {

    @Autowired
    Environment environment;

    private final String URL = "url";
    private final String USER = "parol";
    private final String DRIVER = "driver";
    private final String PASSWORD = "password";
    private final String PROPERTY_SHOW_SQL = "hibernate.show_sql";
    private final String PROPERTY_DIALECT = "hibernate.dialect";

    @Bean
    DataSource dataSource() {
        DriverManagerDataSource driverManagerDataSource =
                new DriverManagerDataSource();
        driverManagerDataSource.setUrl(environment.getProperty(URL));
        driverManagerDataSource.setUsername(
               environment.getProperty(USER));
        driverManagerDataSource.setPassword(
               environment.getProperty(PASSWORD));
        driverManagerDataSource.setDriverClassName(
               environment.getProperty(DRIVER));
        return driverManagerDataSource;
    }

    @Bean
    LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        LocalContainerEntityManagerFactoryBean lfb =
                new LocalContainerEntityManagerFactoryBean();
        lfb.setDataSource(dataSource());
        lfb.setPersistenceProviderClass(HibernatePersistence.class);
        lfb.setPackagesToScan("com.vera.springdata.model");
        lfb.setJpaProperties(hibernateProps());
        return lfb;
    }

    @Bean
    Properties hibernateProps() {
        Properties properties = new Properties();
        properties.setProperty
        (PROPERTY_DIALECT, environment.getProperty(PROPERTY_DIALECT));
        properties.setProperty
        (PROPERTY_SHOW_SQL, environment.getProperty(PROPERTY_SHOW_SQL));
        return properties;
    }

    @Bean
    JpaTransactionManager transactionManager() {
        JpaTransactionManager transactionManager =
                new JpaTransactionManager();
        transactionManager.setEntityManagerFactory(
                entityManagerFactory().getObject());
        return transactionManager;
    }
}
```

- @Configuration – spring annotation определяет класс конфигурации.
- @EnableTransactionManagement–позволяет пользователю использоватьуправление транзакциями в приложении.

- @EnableJpaRepositories("**com.vera.springdata.repository** ")– указывает каталог, где представлен репозиторий.
- @PropertySource("classpath:database.properties")–указывает, что вы храните свойства БД в classpath. Значения из этого файла будут внедрены(injected) в преременную окружения.

Файл database.properties такой же, как в предыдущем проекте.

Класс Customer также не изменился. Только переместился в пакет model, что соответствует терминологии Spring Data.

На следующем шаге создадим репозиторий.

CustomerRepository

```
import com.vera.springdata.model.Customer;
import java.util.List;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface CustomerRepository<C> extends
CrudRepository<Customer, Integer>{
    List<Customer> findByName(String name);
}
```

Как работает CrudRepository .

1. Spring Data ищет всех наследников CrudRepository и автоматически генерирует для них дефолтные реализации, которые включают базовые методы репозитория, типа findOne, findAll, save и т.д.
2. Spring автоматически конфигурирует слой для доступа к данным — JPA (точнее, его реализацию Hibernate).
3. Благодаря аннотации @Repository этот компонент становится доступным в приложении .

В CrudRepository мы можем вызвать много методов без их реализации:

- save
- findOne
- exists
- findAll
- count
- delete
- deleteAll

Мы можем также определить наши собственные методы. В названии метода должны использовать специальные ключевые слова префиксы “find”, “order” перед именем переменной. В нашем примере findByName(String name);

Возможность создавать собственные методы доступа к данным – одна из самых важных особенностей Spring Data JPA, потому что это уменьшает количество рутинного кода. Также уменьшается количество ошибок, т.к. Spring методы уже проверены многими проектами.

Создадим класс сервисов и определим в нем методы работы с репозиторием.

CustomerService

```
@Service

public class CustomerService {
    @Autowired
      CustomerRepository<Customer> customerRepository;

      @Transactional
      public List<Customer> getAllCustomers() {
            return (List<Customer>) customerRepository.findAll();
      }

      @Transactional
      public List<Customer> findByName(String name) {
            return customerRepository.findByName(name);
      }

      @Transactional
      public Customer getById(Integer id) {
            return customerRepository.findOne(id);
      }

      @Transactional
      public void deletePerson(Integer customerId) {
            customerRepository.delete(customerId);
      }

      @Transactional
      public boolean addCustomer(Customer customer) {
            return customerRepository.save(customer) != null;
      }

      @Transactional
      public boolean updateCustomer(Customer customer) {
            return customerRepository.save(customer) != null;
      }
}
```

@Transactional аннотация позволяет методу выполняться внутри транцакции. Spring заботиться об управлении транзакциями.

@Service, @Repository аннотации аналогичны @Component аннотации, т.е. поднимется бин CustomerService.

Слой логики остался прежним.

Метод main не изменился в этом проекте и результат тоже.

6.5 Spring Data JPA vs Spring JDBC.

Spring JDBC намного легче и предназначен для собственных запросов, и если вы собираетесь использовать только JDBC, то лучше использовать Spring JDBC для обработки многословности JDBC.

Если вы хотите создать репозиторий, основанный на JPA для осуществления главным образом CRUD операций и не хотите маяться с созданием абстрактного ДАО и реализацией его интерфейсов, то Spring Data JPA правильный выбор.

Когда не следует использовать Hibernate или Spring Data JPA? Кроме того, когда шаблон Spring JDBC может работать лучше, чем JPA Hibernate/Spring Data?

Используйте Spring JDBC только тогда, когда вам нужно использовать много объединений или когда вам нужно использовать Spring с несколькими подключениями к источникам данных. Как правило, избегайте JPA для Joins.

Задания к гл. 6

1. Создайте серверное приложение для работы с БД Postgres. В БД должны быть 2 таблицы, реализующие связь один ко многим. Приложение должно выполнять CRUD операции со всеми таблицами. Используйте Spring JDBC Template, конфигурационный файл AppConfig, аннотации для поддержания IoC, интерфейс RowMapper.
2. Создайте серверное приложение аналогичное, созданному в п.1, но с использованием hibernate+spring.
3. Создайте серверное приложение аналогичное, созданному в п.1, но с использованием Spring Data JPA.

7. Сервлеты и JSP

7.1 Web server

WEB сервер - программа, которая держит сокет сервера, принимает и передаёт данные. Чаще всего, для ускорения работы, сервер бывает написан не на Java, а на каком-либо другом языке программирования (например на C++).

Приведем примеры WEB серверов.

- NCSA – один из первых веб-серверов. Разработан Роббом Маккулом из NCSA (национальный центр суперкомпьютерных технологий).
- Apache Новая версия NCSA 1995 год.(A PAtCHy Web Serer)

ASF(Apache Software Faundation) была создана в 1999 году из Apache Group в Делавэре, США. Отличительные черты проектов Apache — это совместная разработка кода и открытая, прагматичная лицензия — Apache Software License(ASL).

Рассмотрим лицензии свободного программного обеспечения.

Проекты ASF оладают лицензией ASL.

Проект GNU **(**GNU является рекурсивным акронимом, расшифровывающимся, как *«**GNU is Not Unix**»)* имеют несколько лицензий.

- Общая Публичная Лицензия GNU(GPL, General Public License).
- Менее общая публичная лицензия GNU(LGPL, Lesser General Public License).

Коммерческое ПО не может содержать продукты GPL, но коммерческое ПО может содержать продукты LGPL, что характерно для программных библиотек.

Tomcat имеет лицензию ASL. Если бы Tomcat был с лицензией LGPL, то его можно было бы встроить в коммерческую программу, но сам Tomcat не мог бы быть модифицирован и выпущен как платный продукт.

7.2 Контейнер сервлетов

Контейнер сервлетов – это среда для выполнения сервлетов. Выполняет сервлет, написанный в соответствии со спецификациями.

Примером контейнера сервлетов служит web-сервер Tomcat, разрабатываемый и поддериваемый Apache Software Faundation.

Tomcat – программный продукт с открытым исходным кодом. Sun предоставила ASF в 1999 году. Последующие версии имеют кодовое имя Ctalina. Tomcat – эталонная реализация сервлетов и jsp. О работе с Tomcat написано в приложении 2.

7.3 Сервер приложений

Место сервера приложений в архитектуре клиент-сервер показано на рис.15.

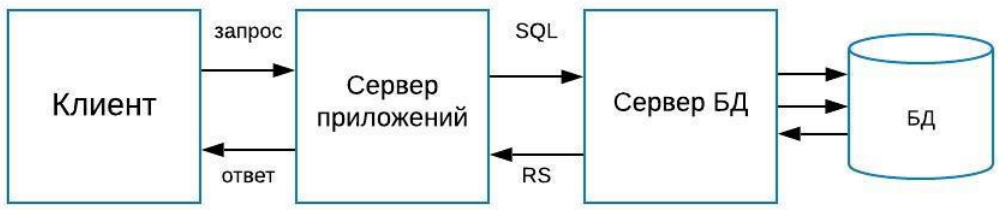


Рисунок 15. Сервер приложений и архитектура клиент.сервер

Серверы приложений дают разработчику следующие возможности.

- Целостность данных и кода. Выделяя бизнес-логику на отдельный сервер или на небольшое количество серверов, можно гарантировать обновления и улучшения приложений для всех пользователей. Отсутствует риск, что старая версия приложения получит доступ к данным или сможет их изменить старым несовместимым образом.
- Централизованная настройка и управление. Изменения в настройках приложения, таких, как изменение сервера базы данных или системных настроек, могут производиться централизованно.
- Безопасность. Сервер приложений действует как центральная точка, используя которую, поставщики сервисов могут управлять доступом к

данным и частям самих приложений, что считается преимуществом защиты. Её наличие позволяет переместить ответственность за аутентификацию с потенциально небезопасного уровня клиента на уровень сервера приложений, при этом дополнительно скрывая уровень базы данных.

- Поддержка транзакций. Конечные пользователи при этом могут выиграть от стандартизованного поведения системы, от уменьшения времени на разработку и от снижения стоимости. В то время как сервер приложений выполняет массу нужного генерирования кода, разработчики могут сфокусироваться на бизнес-логике.

  Примерами серверов приложений служат:

  - Sun GlassFish,
  - IBM WebSphere,
  - RedHat JBoss Application Server, WildFly
  - Apple WebObjects (англ.),
  - Oracle Weblogic Server

7.4 Сервлеты

Сервлеты – это Java программы, которые вызываются для динамического генерирования контента и предоставляют способ для развертывания приложения в интернет. Сервлеты являются специализированным механизмом Java для создания WEB ресурсов. Взаимодействие клиента с контейнером сервлетов показано на рис. 16.

Сервлеты характеризуются тем, что:

- Созданы Sun Microsystems.
- Все сервлеты реализуют интерфейс Servlet, который определяет стандартный жизненный цикл приложения.
- Каждый сервлет может обработать много запросов от многих клиентов.

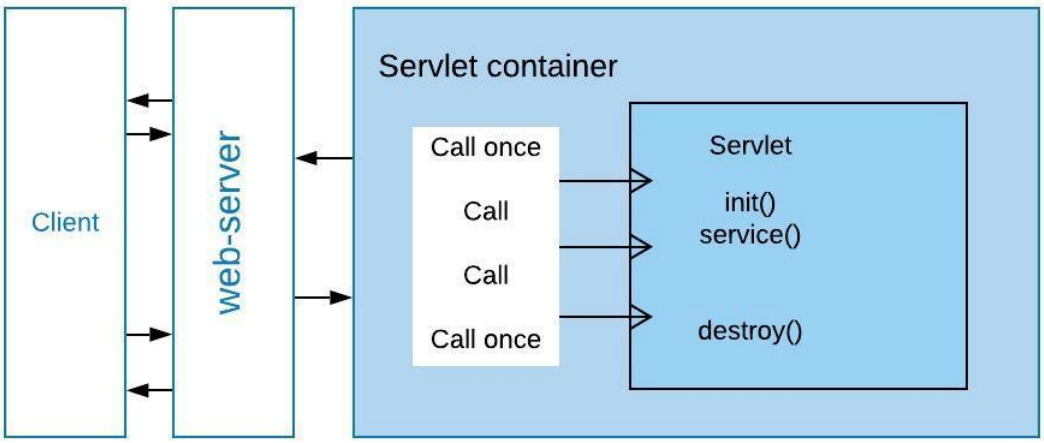


Рисунок 16. .Взаимодействие с сервлетом

На начальном этапе web-программирования использовалась технология Common Gatewey Interface (CGI). Рассмотрим CGI-технологию в сравнении с сервлетами.

Недостатки CGI(Common Gateway Interface)

- Нестабильность CGI программы может привести к краху всей машины.
- Для каждого запроса необходимо создание нового экземпляра приложения (новый поток)
- Не предоставляет никаких служб, помогающих реализовать системы с поддержкой персонализации (поддержка опознавания клиента, предоставление доступа к средствам поддержки пользовательской информации, запрет доступа к приложению авторизованным пользователям, хранение в приложении информации периода выполнения).

В связке с web-сервером работает базовый сервлет. Именно ему отправляет сервер данные и от него же получает ответ, отправляемый клиенту. Фактически, базовый сервлет является "мозгом" сервера. Основная функция этого сервлета - прочитать запрос клиента, расшифровать его и, в соответствиии с расшифровкой, передать работу сервлету, отвечающему за этот тип запрашиваемой информации. Зачастую, для достижения скорости,

роль базового сервлета играет сам сервер. Именно по такой схеме работает Tomcat.

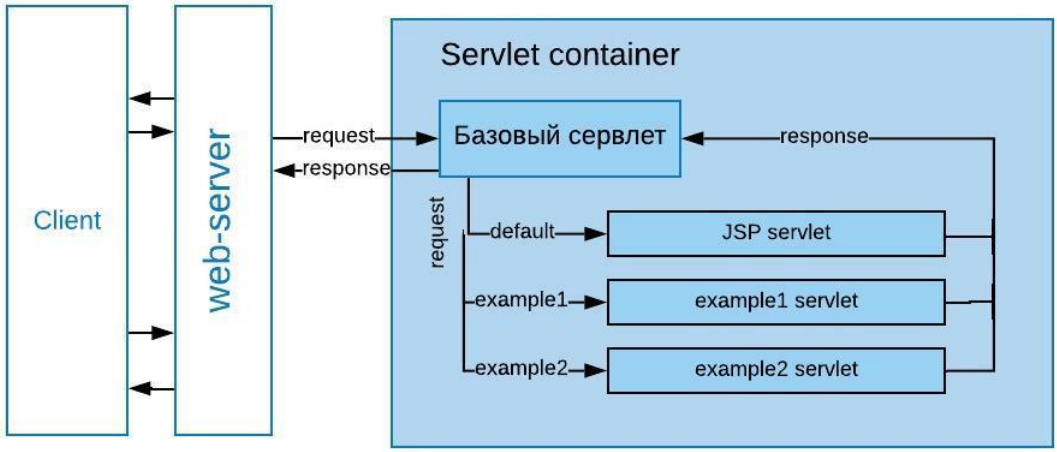


Рисунок 17. Схема работы сервлетов и сервера

На рисунке 17 изображена схема передачи вызовов (request) и ответов (response) между сервером и сервлетами. Данная схема изображает работу HTTP сервера, который имеет несколько JSP страниц и два ресурса "/example1" и "/example2", за обработку которых отвечает два сервлета - "example1 servlet" и "example2 servlet" соответственно.

Разберём пошагово то, что изображено на рисунке:

1. клиент подсоединяется к серверу
2. сервер передаёт запрос (request) базовому сервлету.
3. базовый сервлет вычеленяет из запроса URI ресурса
   - если URI указывает на "/example1", то запрос передаётся сервлету " example1 servlet ", который, в дальнейшем, и обрабатывает этот запрос.
   - если URI указывает на "/ example1", сервер передаёт запрос сервлету "example2 servlet"/
   - во всех остальных случаях запрос передаётся модулю "JSP servlet".
4. сервлет, которому было передано управление, обрабатывает данные, создаёт ответ (response), после чего ответ отсылается обратно базовому сервлету.

5. базовый сервлет, не обрабатывая полученные данные, тут же пересылает их обратно серверу

6. сервер выдаёт данные клиенту

Таким образом, достигается разбиение задачи обработки запроса на логические части, за каждую из которых отвечает свой модуль. На самом деле, ступеней в обработке запроса может быть гораздо больше. К примеру, за методы "GET" и "POST" могут отвечать разные модули.

7.5 Java server page JSP.

JSP (Java server pages) — это технология, которая позволяет внедрять Java-код, а также EL (expression language), в статичное содержимое страницы. Позволяющая также веб-разработчикам динамически генерировать HTML, XML и другие веб-страницы. JSP является составной частью единой технологии создания бизнес-приложений J2EE.

Основная часть JSP страницы это HTML, в которую вставляются теги с динамическим содержимым.

7.6 Простые примеры с использованием сервлетов и JSP

Рассмотрим простейший пример с использованием сервлетов. В этом примере не используется сборщик maven, т.к. незачем.

Структура проекта приведена на рис. 18.

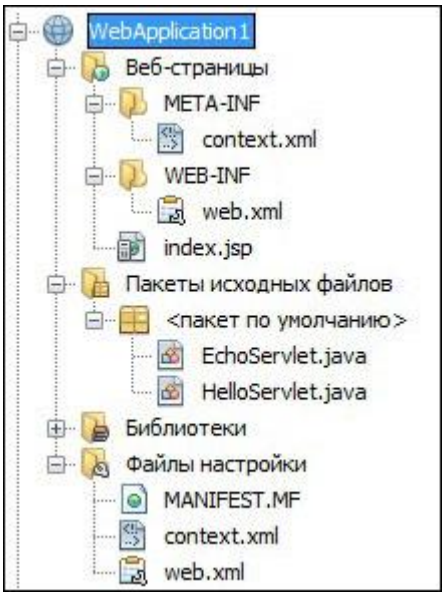


Рисунок 18. Структура простого примера с ипользованием сервлетов и JSP

В структуре web-проектов появляется папка «Веб-страницы», но каталог на диске называется «web», привет русской версии NetBeans. Папки

META-INF и WEB-INF – содержание и описание web приложения. META-INF содержит файл context.xml с содержанием

context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Context antiJARLocking="true" path="/WebApplication1"/>
```

Путь «WebApplication1» – доступ к вашему web-приложению. Результат показан на рис.19.

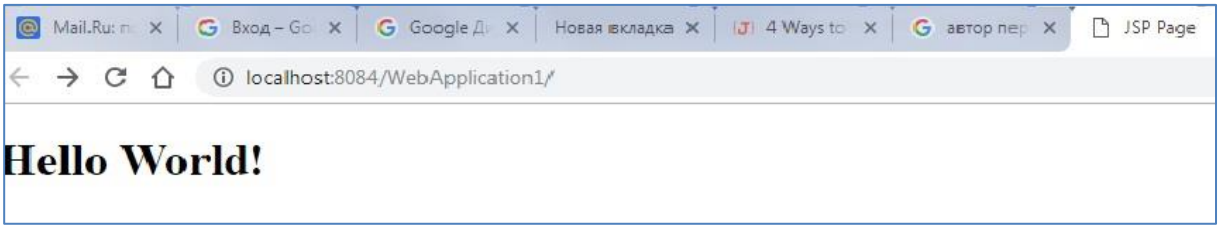


Рисунок 19. Результат работы примера

Папка WEB-INF содержит файл входа в web – приложение – web.xml, если ваш проект не настроен на использование конфигурационных классов и аннотаций.

Рассмотрим web.xml.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <servlet>
        <servlet-name>HelloServlet</servlet-name>
        <servlet-class>HelloServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>HelloServlet</servlet-name>
        <url-pattern>/HelloServlet</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
</web-app>
```

Все, что написано до тега <servlet>, пишется с помощью среды netbeans или другой. Эта схема, возможно, устарела и в других примерах появится другой код или будет отсутствовать web.xml.

В теге <servlet> пишется имя сервлета и полное имя класса сервлета. В теге <servlet-mapping> пишется <url-pattern>/HelloServlet</url-pattern>.

/HelloServlet – url - путь к сервлету с именем HelloServlet. Результат работы HelloServlet показан на рис.20.

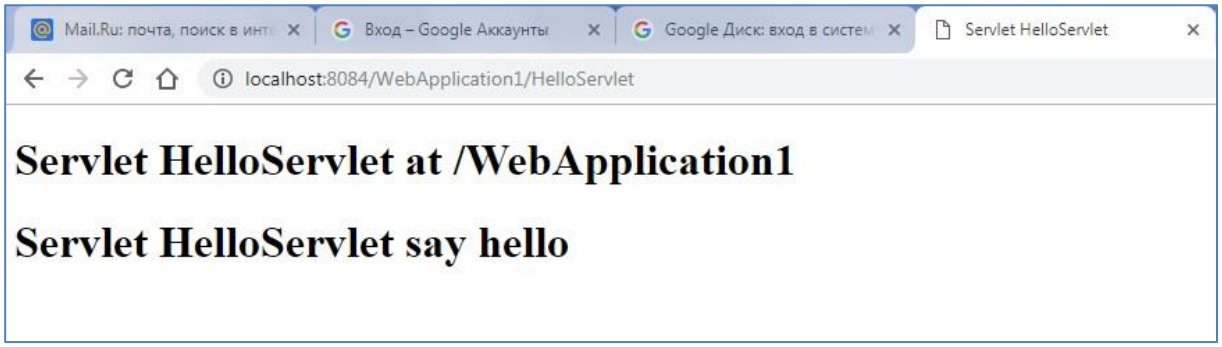


Рисунок 20. Результат работы HelloSevlet

Рассмотрим классы EchoServlet и HelloServlet.

EchoServlet, HelloServlet

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name = "EchoServlet", urlPatterns = {"/EchoServlet"})

public class EchoServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet EchoServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet EchoServlet at " +
                        request.getQueryString() + "</h1>");
            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        processRequest(request, response);
    }
```

```
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        processRequest(request, response);
    }


    @Override
    public String getServletInfo() {
        return "Short description";
    }
}
```

Класс EchoServlet наследует HttpServlet и может считаться сервлетом. Вместо записи в файл web.xml используется аннотация `@WebServlet(name = "EchoServlet", urlPatterns = {"/EchoServlet"})`.

Метод Processes requests используется для обоих методов посылки общений HTTP – GET и POST. Переменная request содержит запрос, переменная response – ответ. Мы не будем детально описывать методы классов HttpServletRequest и HttpServletResponse. Скажем, только, что если способ посылки сообщения Get, то строку сообщения можно считать методом request.getQueryString(), а если Post, то методом request.getParameter("name").

Пример посылки сообщения методом “get”. Результат посылки сообщения методом “get” показан на рис.21.

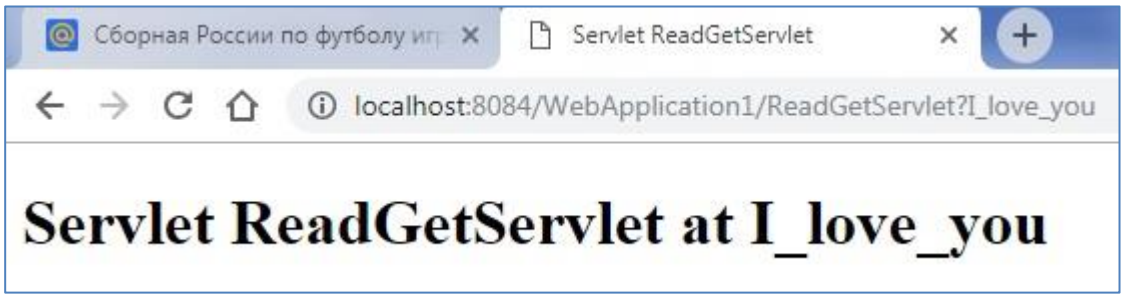


Рисунок 21. Результат посылки сообщения методом “get”

Приведем описание сервлета HelloServlet, определение которого задано в файле web.xml.

```
      HelloServlet
public class HelloServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
```

```
                                    HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet HelloServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet HelloServlet at " +
                        request.getContextPath() + "</h1>");
            out.println("<h1>Servlet HelloServlet say hello " +"</h1>");
            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }

  //определения методов doGet и   doPost не приводятся для краткости

}
```

Формировать html – страницы в коде программы неудобно. Для упрощения создания динамических web-страниц служат страницы jsp (Java server pages**).**

Продолжим наш пример с элементарным сервлетом.

Добавим в проект echojsp.jsp, который будет находиться в проекте там же, где index.jsp.

echojsp.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>Echo</h1>
        <div>
    <h3>Echo: ${Echo}</h3>
    </div>
    </body>
</html>
```

Единственное отличие данного файла от файла html – ${Echo}. Это и есть динамическая часть. Echo будет передано из сервлета ReadEchoJsp.

ReadEchoJsp

```
@WebServlet(urlPatterns = {"/ReadEchoJsp"})
public class ReadEchoJsp extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
```

```
            HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        request.setAttribute("Echo", request.getQueryString());
        String address = "echojsp.jsp";
        RequestDispatcher dispatcher
                = request.getRequestDispatcher(address);
        dispatcher.forward(request, response);
    }
…}
```

В этом сервлете формируется request с помощью метода setAttribute. Первый параметр в этом методе должен совпадать с ${Echo}. В ${Echo} мы помещаем строчку запроса. Далее устанавливаем адрес страницы jsp и методом dispatcher.forward пересылаем все из request'a в response, т.е. echojsp.jsp.

Результат представлен на рис 22.

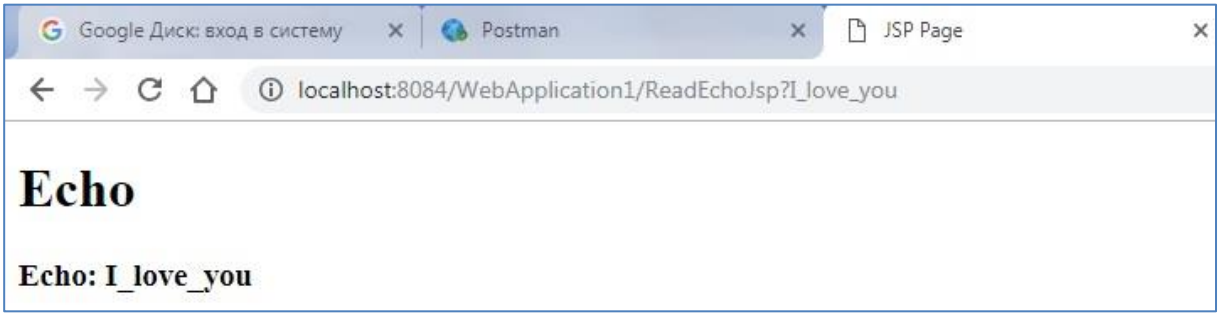


Рисунок 22. Результат работы ReadEchoJsp

Приведем пример с посылкой сообщения из jsp страницы методом post и реализацией «эхо». Файл echojsp_post.jsp приведен ниже.

echojsp_post.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>Hello World!</h1>
        <form name="inputName"
action="http://localhost:8084/WebApplication1/ReadEchoJsp" method="POST">
            <input type="text" name="name" value="" />
            <input type="submit" value="submit" name="submit" />
        </form>
        <h1>Echo</h1>
        <div>
            <h3>Echo: ${Echo}</h3>
        </div>
    </body>
```

```
</html>
```

Приведем пример запроса (см. рис. 23).

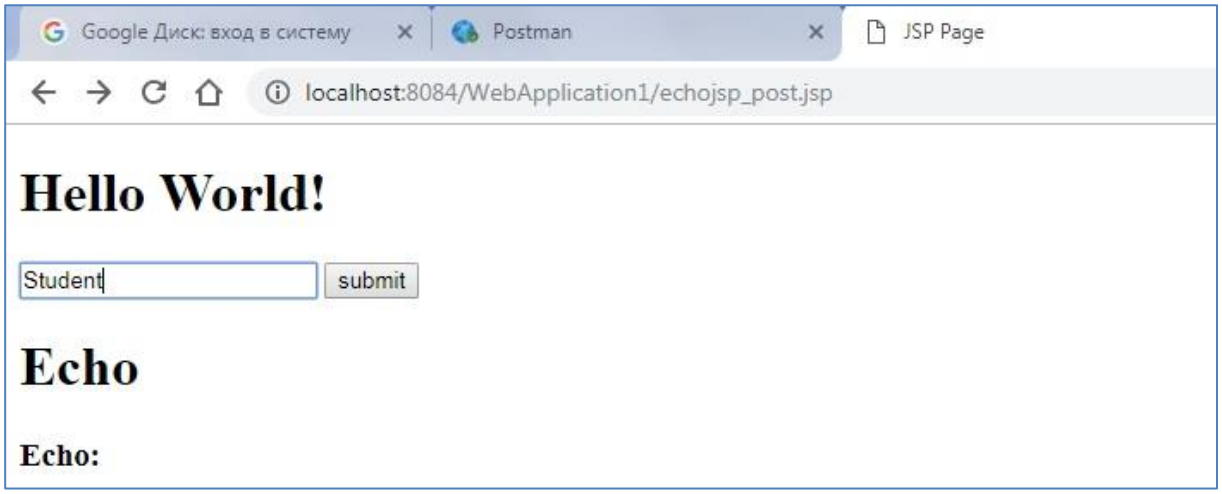


Рисунок 23. Пример запроса echojsp_post.jsp

И результат ответа показан на рис. 24.

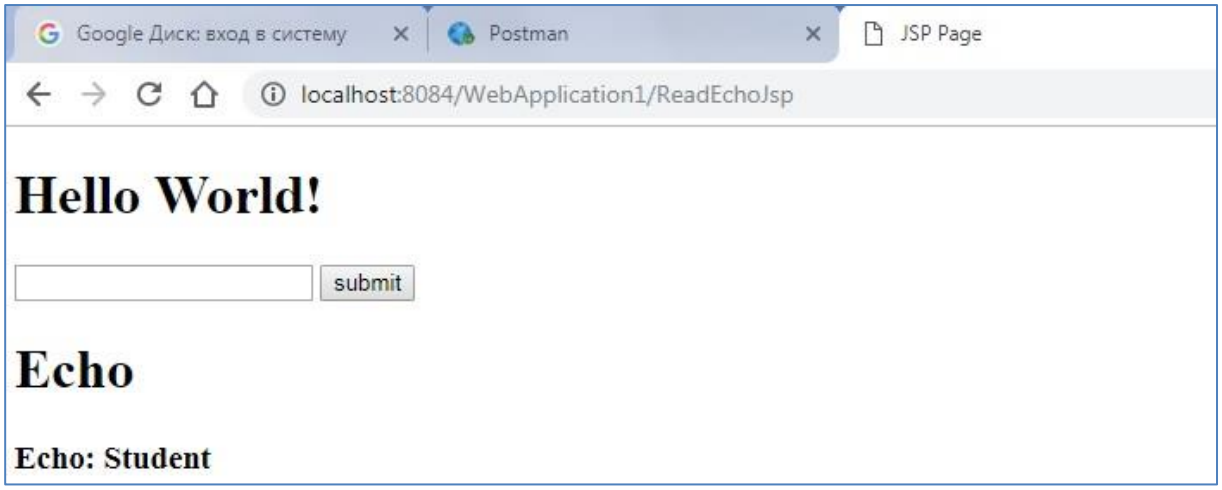


Рисунок 24. Результат ответа echojsp_post.jsp

На этом мы закончим рассмотрение элементарных понятий web-приложений на java.

Обычно, как мы видели в предыдущих разделах, для упрощения создания приложений используются библиотеки или фреймворки. Аналогично поступают и с web-приложениями. Рассмотрим фраймворк Spring для web-приложений.

Задания к гл. 7

1. Напишите сервлет, который выводит заголовок запроса.

2. Напишите сервлет, который возвращает количество байт в отправленной строке.
3. Напишите JSP страницу, которая повтояет себя введенное число раз.

8. Spring MVC

Фреймворк Spring MVC обеспечивает архитектуру паттерна Model — View — Controller (Представление — Модель — Контроллер) при помощи слабо связанных готовых компонентов. Паттерн MVC разделяет аспекты приложения (логику ввода, бизнес-логику и логику UI), обеспечивая при этом свободную связь между ними.

Model (Модель) инкапсулирует (объединяет) данные приложения, в целом они будут состоять из POJO (Plain Old Java Object –Старые добрые Java-объекты) или бинов.

View (Представление, Вид) отвечает за отображение данных модели, — как правило, генерируя HTML, которые мы видим в своём браузере.

Controller (Контроллер) обрабатывает запрос пользователя, создаёт соответствующую модель и передаёт её для отображения в View (Представление). Логика работы Spring MVC представлена на рис.25.

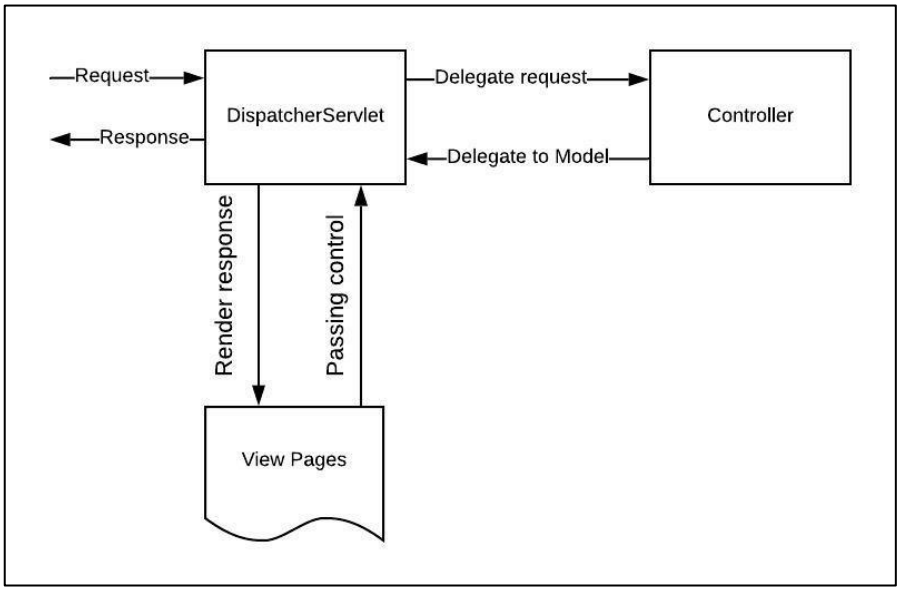


Рисунок 25. Логика работы Spring MVC

Вся логика работы Spring MVC построена вокруг DispatcherServlet, который принимает и обрабатывает все HTTP-запросы (из UI) и ответы на них. DispatcherServlet работает со следующими компонентами: HandlerMapping, контроллер, ViewResolver**,** представление.

Ниже приведена последовательность событий, соответствующая входящему HTTP-запросу:

- После получения HTTP-запроса DispatcherServlet обращается к интерфейсу HandlerMapping, который определяет, какой контроллер должен быть вызван, после чего, отправляет запрос в нужный контроллер.
- Контроллер принимает запрос и вызывает соответствующий служебный метод, основанный на GET или POST. Вызванный метод определяет данные модели, основанные на определённой бизнес-логике и возвращает в DispatcherServlet имя представления (View).
- При помощи интерфейса ViewResolver DispatcherServlet определяет, какое представление нужно использовать на основании полученного имени.
- После того, как отображение (View) создан, DispatcherServlet отправляет данные Модели в виде атрибутов в представление, который в конечном итоге отображается в браузере.

Все вышеупомянутые компоненты, а именно, HandlerMapping, Controller и ViewResolver, являются частями интерфейса WebApplicationContext extends ApplicationContext, с некоторыми дополнительными особенностями, необходимыми для создания web-приложений. Все проекты данного раздела можно скачать с https://bitbucket.org/w2lvera/springmvcsimple/src/master/.

8.1 Spring MVC Hello Word

Рассмотрим простейшее приложение Spring MVC.

Связь приложения с web можно выполнить 2 способами– с помощью файла xml или с помощью class Config.

Приведем пример с файлом Xml. Структура проекта представлена на рис.26.

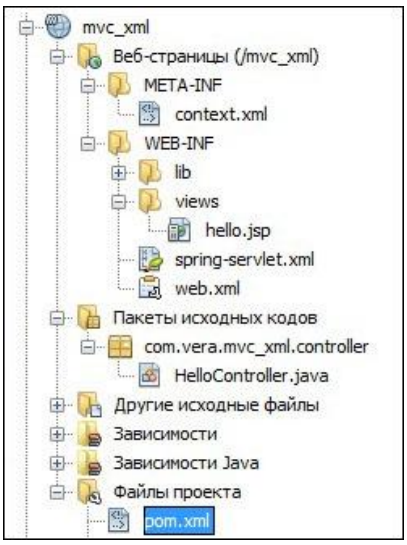


Рисунок 26. Структура проекта Spring MVC с файлом Xml

Файл web.xml, который осуществляет отображение web, представлен ниже.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://java.sun.com/xml/ns/javaee"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         id="WebApp_ID" version="3.0">
  <display-name>mvc_xml</display-name>

      <!-- Add Spring MVC DispatcherServlet as front controller -->
      <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>
                org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <init-param>
                 <param-name>contextConfigLocation</param-name>
                 <param-value>/WEB-INF/spring-servlet.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

 </web-app>
```

Фреймворк попытается загрузить контекст приложения из файла с именем [servlet-name]-servlet.xml, находящегося, в нашем случае, в каталоге

mvc_xml/WEB-INF. У нас этот файл будет называться spring-servlet.xml. Название каталога веб страницы(/mvc_xml) берется из следующих тегов файла pom.xml.

pom.xml

```
<plugin>
    <artifactId>maven-war-plugin</artifactId>
    <version>3.0.0</version>
    <configuration>
         <warSourceDirectory>mvc_xml</warSourceDirectory>
    </configuration>
</plugin>
```

Далее тэг <servlet-mapping> указывает, какие веб-адреса обрабатываются каким DispatcherServlet'ом. В нашем случае, все HTTP-запросы, не требуют дополнительных адресов. Достаточно url – localhost:8080/mvc_xml.

Теперь давайте проверим конфигурацию для spring-servlet.xml, размещённую в каталоге Веб страницы(/mvc_xml) /WEB-INF:

spring-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns:beans="http://www.springframework.org/schema/beans"
             xmlns:context="http://www.springframework.org/schema/context"
             xsi:schemaLocation="http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <annotation-driven />
    <context:component-scan base-package="com.vera.mvc_xml.controller" />

    <beans:bean
 class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <beans:property name="prefix" value="/WEB-INF/views/" />
        <beans:property name="suffix" value=".jsp" />
    </beans:bean>

</beans:beans>
```

Отметим следующие важные свойства конфигурации.

1. `annotation-driven` указывает DispatcherServlet, что Controller classes надо искать с использованием аннотации @Controller.

2. `context:component-scan` указывает DispatcherServlet где искать классы контроллеров.

3. Бин **InternalResourceViewResolver** показывает, где расположены страницы view и их расширение. Методы контроллера возвращают имя страницы view, а затем к этому имени добавляется suffix для формирования view page, используемой для ответа.

DispatcherServlet отправляет запрос контроллерам для выполнения определённых функций. Аннотация @Controllerannotation указывает, что конкретный класс является контроллером. Аннотация @RequestMapping используется для мапинга (связывания) с URL для всего класса или для конкретного метода обработчика.

HelloController

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class HelloController {

    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String printWelcome(ModelMap model) {

        model.addAttribute("message", "Spring  MVC - Hello World");
        return "hello";
    }
}
```

Отметим, что класс контроллер выполняет следующие действия.

- Вызывает методы. Внутри методов можно вызвать другие методы, определяющие бизнес-логику приложения.
- Модель (Model) создается на основании заданной бизнес-логики. Можно добавлять аттрибуты Модели, которые будут добавлены в Вид (View). В примере выше создаётся Модель с атрибутом «message».
- В данном примере метод **printWelcome** возвращает имя View в виде строки String – «hello».
- Запрос к методу **printWelcome** будет осуществляться методом Get с URL совпадающим с именем war файла.

Осталось создать отображение –view.

Понятно, что оно будет иметь имя /WEB-INF/views/hello.jsp.

hello.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
         pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ page session="false"%>
<html>
    <head>
        <title>Home</title>
    </head>
    <body>
        <h1>Message : ${message}</h1>
    </body>
</html>
```

`${message}` означает, что будет выведена срока из модели с атрибутом message.

Spring MVC поддерживает множество типов View для различных технологий отображения страницы. В том числе — JSP, HTML, PDF, Excel, XML, Velocity templates, XSLT, JSON, каналы Atom и RSS, JasperReports и проч. Но чаще всего используются шаблоны JSP, написанные при помощи JSTL.

На рис.27 предствлен результат работы web приложения. Вывод осуществляет hello.jsp. Доступ к контроллеру по url определенному в аннотации `@RequestMapping(value = "/")`.

Рисунок 27. Результат работы web приложения

Приведем пример с config class.

Структура проекта представлена на рис.28.

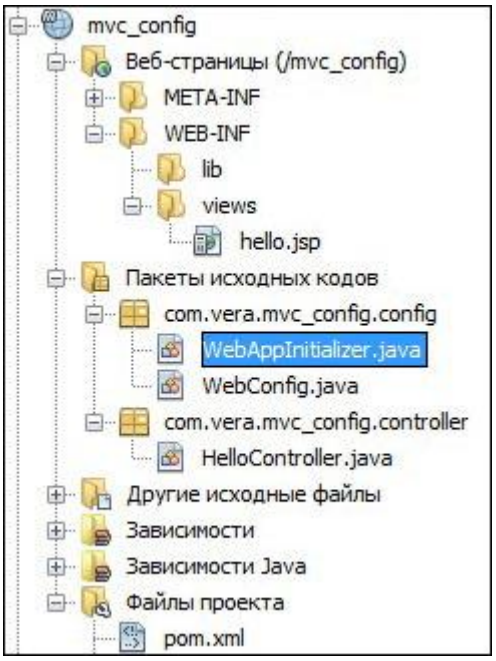


Рисунок 28. Структура проекта с классом config

В проекте в части «Веб страницы» отсутствуют xml файлы. Вместо них появился каталог mvc_config.config. В этом каталоге главным классом является WebAppInitializer, т.к. он наследует AbstractAnnotationConfigDispatcherServletInitializer. Этот класс можно сформировать различными способами. Покажем простейший.

WebAppInitializer

```
import org.springframework.web.servlet.support.
        AbstractAnnotationConfigDispatcherServletInitializer;

public class WebAppInitializer extends
        AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class[] getRootConfigClasses() {
        return new Class[] { WebConfig.class };
    }

    @Override
    protected Class[] getServletConfigClasses() {
        return new Class[0] ;
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

При создании **RootConfigClasses** использовался класс конфигурации **WebConfig.**

WebConfig

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.ViewResolver;
import
org.springframework.web.servlet.config.annotation.DefaultServletHandlerConfig
urer;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.InternalResourceViewResolver;
import org.springframework.web.servlet.view.JstlView;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.vera.mvc_config.controller")
public class WebConfig extends WebMvcConfigurerAdapter {

    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver viewResolver =
                                        new InternalResourceViewResolver();
        viewResolver.setViewClass(JstlView.class);
        viewResolver.setPrefix("/WEB-INF/views/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }

    @Override
    public void
configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
        configurer.enable();
    }
}
```

Отметим следующие важные свойства представленной конфигурации.

- **@Configuration** аннотация говорит о том,что этот класс является конфигурационным.
- **@EnableWebMvc** аннотация говорит о том, что используется фреймворк mvc.
- **@ComponentScan** указывает DispatcherServlet где искать бины проекта. В данном случае контроллера.

  Остальный файлы не изменились и остались как в проекте mvc_xml. Результат работы проекта также не изменился (см. рис.29).

Рисунок 29. Результат работы проекта с config class

8.2 RestServicies

Web Service предоставляет необработанную информацию и используется приложениями. Эти приложения отображают данные в соответствии с требованиями конкретного web сайта. Данные могут быть проанализированы и переработаны перед тем, как вернуть их конечному пользователю. Данные обычно возвращаются в формате XML или JSON.

RESTful Web Service это Web Service, написанный на основании структуры REST.

REST (REpresentational State Transfer)(2000 год) определяет правила архитектуры для дизайна веб сервисов. Приведем четыре основных правила дизайна.

- Использовать явные методы HTTP
- Не имеет состояния
- Отображает структуру папок как URls
- Передача JavaScript Object Notation (JSON), XML или обоих.

RESTful Web Service широко используется и заменяет веб сервисы, которые основывались на SOAP и WSDL. RESTful занимает мало места и его легко расширять и поддерживать.

В REST архитектуре определены правила для методов HTTP.

- Чтобы создать ресурс на сервере, вам нужно использовать метод POST.
- Для получения ресурса, используйте GET.

- Чтобы поменять состояние ресурса или обновить его, используйте PUT.
- Чтобы отменить или удалить ресурс, используйте DELETE.

Заметьте, что правила выше необязательны, на самом деле вы можете использовать метод GET, чтобы получить данные, вставлять, менять или удалять данные на сервере.

Стандарт JSON – текстовый формат обмена данными. JSON-текст представляет собой (в закодированном виде) одну из двух структур:

- Набор пар ключ: значение. В различных языках это реализовано как запись, структура, словарь, хеш-таблица, список с ключом или ассоциативный массив. Ключом может быть только строка, значением — любая форма.
- Упорядоченный набор значений. Во многих языках это реализовано как массив, вектор, список или последовательность.

Рассмотрим пример простого Rest сервиса, который поддерживает посылку сообщений типа «ключ – значение».

Структура проекта представлена на рис.30.

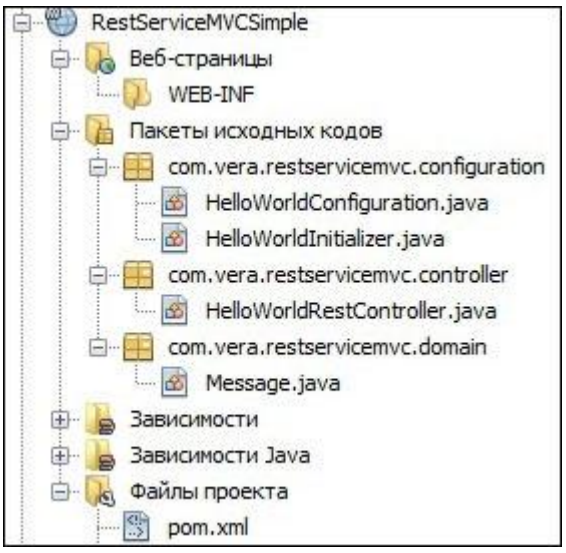


Рисунок 30. Структура проекта простого Rest сервиса

Классы конфигураций не отличаются от предыдущих примеров. Отсутствует класс WebConfig. В каталоге WEB-INF также нет файлов.

Класс контроллер содержит аннотацию @RestController, что говорит о создании контроллера, который может формировать json или xml сообщения. Аннотация @PathVariable используется для работы с параметрами, передаваемыми через адрес запроса в SpringWebMVC.

HelloWorldRestController

```
package com.vera.restservicemvc.controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.vera.restservicemvc.domain.Message;

@RestController
public class HelloWorldRestController {

    @RequestMapping("/")
    public String welcome() {//Welcome page, non-rest
        return "Welcome to RestTemplate Example.";
    }

    @RequestMapping("/hello/{player}")
    public Message message(@PathVariable String player) {//REST Endpoint.

        Message msg = new Message(player, "Hello " + player);
        return msg;
    }

}
```

Приведем класс Message, который содержит два поля name(ключ)и text(значение).

Message

```
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
//@XmlRootElement(name = "player")
public class Message {
      String name;
      String text;
      public Message(){
      }
      public Message(String name, String text) {
            this.name = name;
            this.text = text;
      }

      //@XmlElement
      public String getName() {
            return name;
      }

      //@XmlElement
      public String getText() {
```

```
            return text;
        }
}
```

В результате работы метода message класса контроллер получим либо json сообщение, либо, если раскоментируем xml аннотации в классе Message, xml сообщение.

Приведем json и xml результат (рис. 31 и 32).

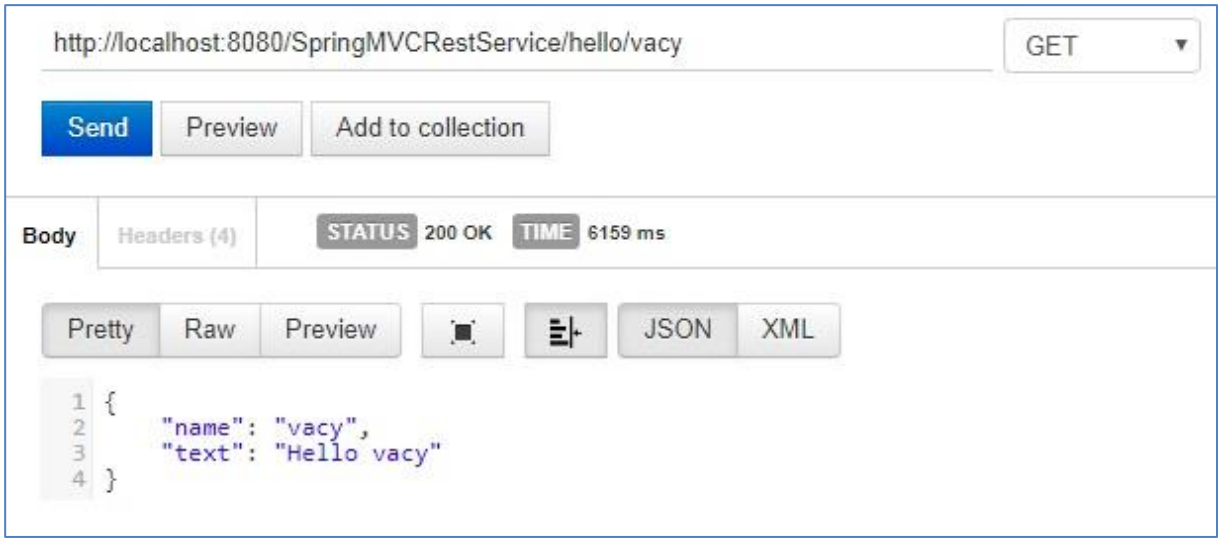


Рисунок 31. Рзультат работы сервиса в формате json

Рисунок 32. Рзультат работы сервиса в формате xml

Для иммитании посылки сообщений и получения ответов от Rest сервиса воспользуемся программой rest-клиент – Postman (устанавливается как «дополнительные инструменты – расширения» к браузеру).

8.3 Spring MVC с доступом к БД derby

Рассмотрим наш пример с доступом к БД derby sumple. Будем использовать проект из раздела Spring Data JPA. Выполним вывод данных из нашего примера на страницу jsp и в Rest service.

RestService в нашем проекте будет формировать в JSON формате список заказчиков (customers), имена которых начинаются с заданного префикса (буквы). Такой список – это результат работы функции getCustomerList (String prefix) класса CustomerListProcessorImpl. Также мы создадим RestService, который выводит в JSON формате список всех заказчикови заказчика с заданным id. Структура проекта представлена на рис.33.

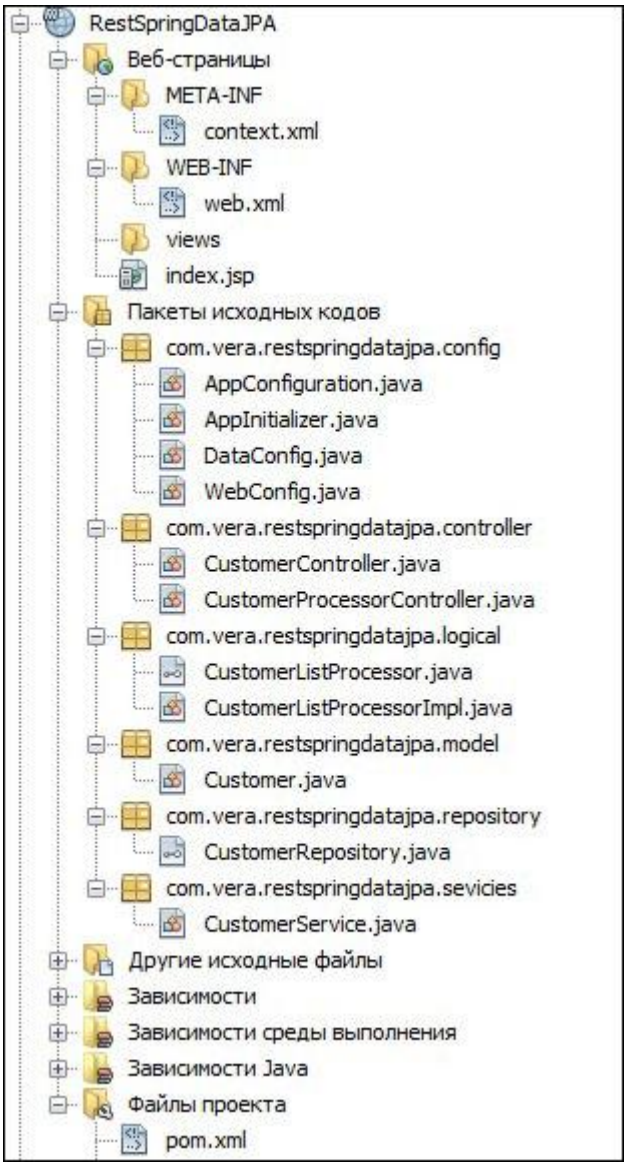


Рисунок 33. Структура проекта RestService

В файле проекта pom.xml заданы зависимости для поддержки rest. Jackson стандартная библиотека JSON для Java.

Зависимости rest

```
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.5</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
            <version>${serializer.version}</version>
        </dependency>
```

Класс AppInitializer представлен ниже. Класс WebConfig не содержит кода, т.к. не создаются web-страницы. Главное – аннотации @EnableWebMvc и ComponentScan("com.vera.restspringdatajpa"). Последняя аннотация прикрепляет к проекту DataConfig и создает все бины.

AppInitializer и WebConfig

```
package com.vera.restspringdatajpa.config;
import
org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherSer
vletInitializer;

public class AppInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {
    protected Class<?>[] getRootConfigClasses() {
              return  new Class[]{WebConfig.class};
    }

    protected Class<?>[] getServletConfigClasses() {
              return  new Class[0];
    }

    protected String[] getServletMappings() {
        return new String[]{"/"};
    }
}

@Configuration
@EnableWebMvc
@ComponentScan("com.vera.restspringdatajpa")
public class WebConfig extends WebMvcConfigurerAdapter {
}
```

Рассмотрим классы контроллеров. Класс CustomerController использует внедряемый объект customerService, т.к. есть аннотация @Autowired. Аннотация @RequestMapping(value = "/customer", method = RequestMethod.GET)позволяет вызвать метод getAll(),возвращающий список клиентов. Список возвращается в json формате, т.к. есть аннотация @RestController класса CustomerController. Аннотация @GetMapping ("/list/{prefix}") заменяет @RequestMapping(value = "/customer", method =

RequestMethod.GET). Аннотация @PathVariable позволяет использовать параметр {prefix} из url как переменную.

CustomerController и CustomerProcessorController

```
package com.vera.restspringdatajpa.controller;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;
import com.vera.restspringdatajpa.model.Customer;
import com.vera.restspringdatajpa.sevicies.CustomerService;
import org.springframework.web.bind.annotation.GetMapping;

@RestController
public class CustomerController {

    @Autowired
    CustomerService customerService;

    @RequestMapping("/")
    public String welcome() {//Welcome page, non-rest
        return "Welcome to RestTemplate Example.";
    }

    @RequestMapping(value = "/customer/{id}", method = RequestMethod.GET)
    public @ResponseBody
    Customer getAllCustomer(@PathVariable Integer id) {
        return customerService.getById(id);
    }

    @RequestMapping(value = "/customerByName/{name}", method =
                                                        RequestMethod.GET)
    public List<Customer> getCustomerByName(@PathVariable String name) {
        return customerService.findByName(name);
    }

    @RequestMapping(value = "/customer", method = RequestMethod.GET)
    public List<Customer> getAll() {
        return customerService.getAllCustomers();
    }

    @RequestMapping(value = "/customer/{id}", method =
                                                    RequestMethod.DELETE)
    public HttpStatus deleteCustomer(@PathVariable Integer id) {
        customerService.deleteCustomer(id);
        return HttpStatus.NO_CONTENT;
    }

    @RequestMapping(value = "/customer", method = RequestMethod.POST)
    public HttpStatus insertCustomer(@RequestBody Customer customer) {
        return customerService.addCustomer(customer) ?
                        HttpStatus.CREATED : HttpStatus.BAD_REQUEST;
    }

    @RequestMapping(value = "/customer", method = RequestMethod.PUT)
    public HttpStatus updateCustomer(@RequestBody Customer customer) {
```

```
        return customerService.updateCustomer(customer) ?
                            HttpStatus.ACCEPTED : HttpStatus.BAD_REQUEST;
    }
}
@RestController
@RequestMapping("/customerProcessor")
public class CustomerProcessorController {
    @Autowired
    CustomerListProcessor listProcessor;
    @GetMapping ("/list/{prefix}")
    public List<Customer> listCustomer(@PathVariable String prefix){
        List<Customer> customers = listProcessor.getCustomerList(prefix);
        return customers;
    }
}
```

Приведем результаты работы созданных web-сервисов (рис. 34). Для получения результатов воспользуемся программой rest-клиент – Postman.

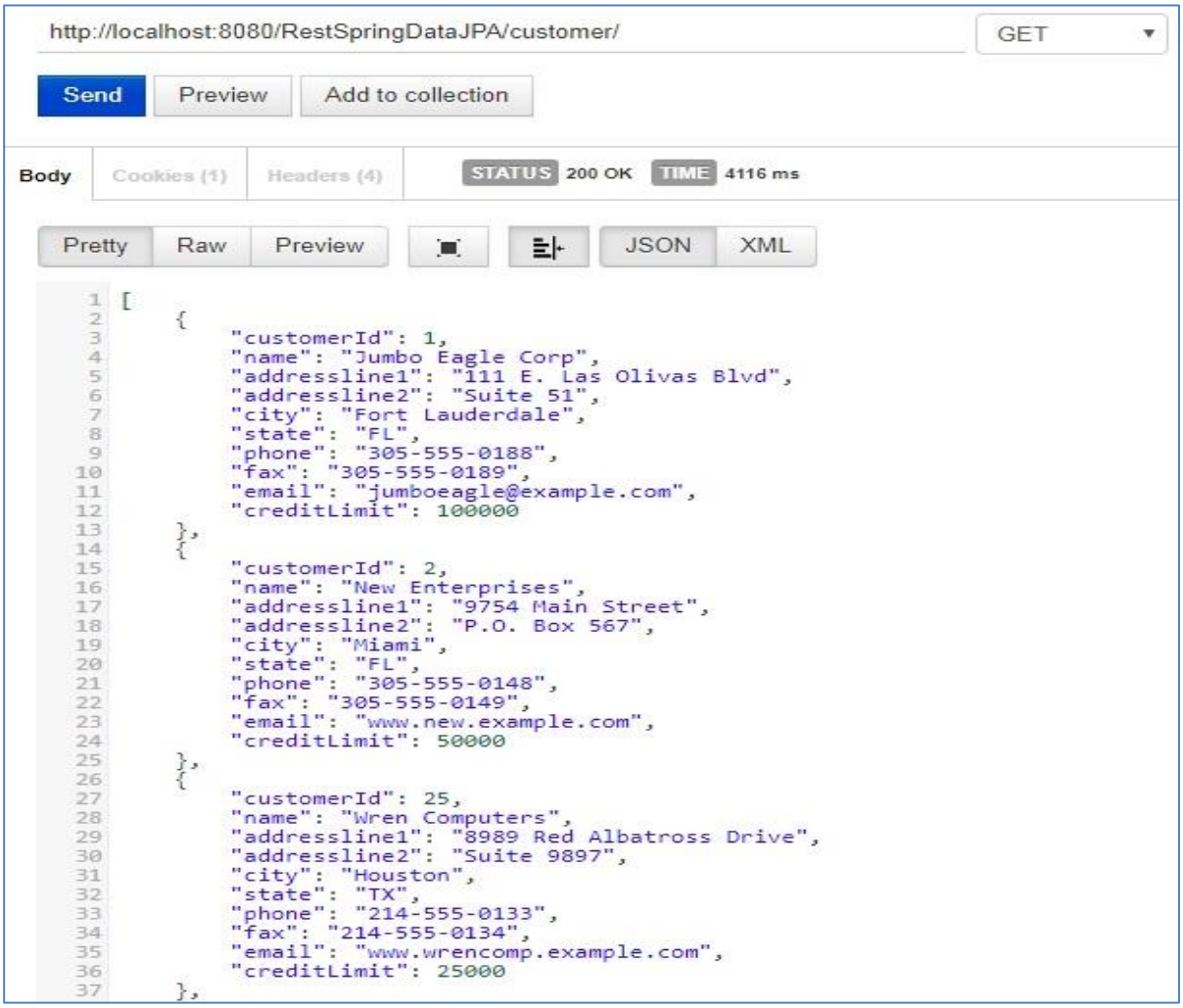


Рисунок 34. Результат работы web-сервисов

Обратите внимание на строку запроса http://localhost:8080/RestSpringDataJPA/customer/. War файл проекта называется

RestSpringDataJPA . Это название задано в pom.xml в теге <configuration>. Путь /customer задан в аннотации @RequestMapping(value = "/customer", method = RequestMethod.GET) к методу **`getAll()`** сервиса. Пример работы остальных методов рассмотрим позже.

Pom.xml <configuration>

```
<configuration>
   <warSourceDirectory>src/main/webapp</warSourceDirectory>
   <warName>RestSpringDataJPA</warName>
   <failOnMissingWebXml>false</failOnMissingWebXml>
</configuration>
```

Результат работы web сервиса имитирующего бизнес-логику (processor) приведен на рис.35. В выводимых заказчиках оказались лишь те, имена которых начинаются с буквы ‘J’.

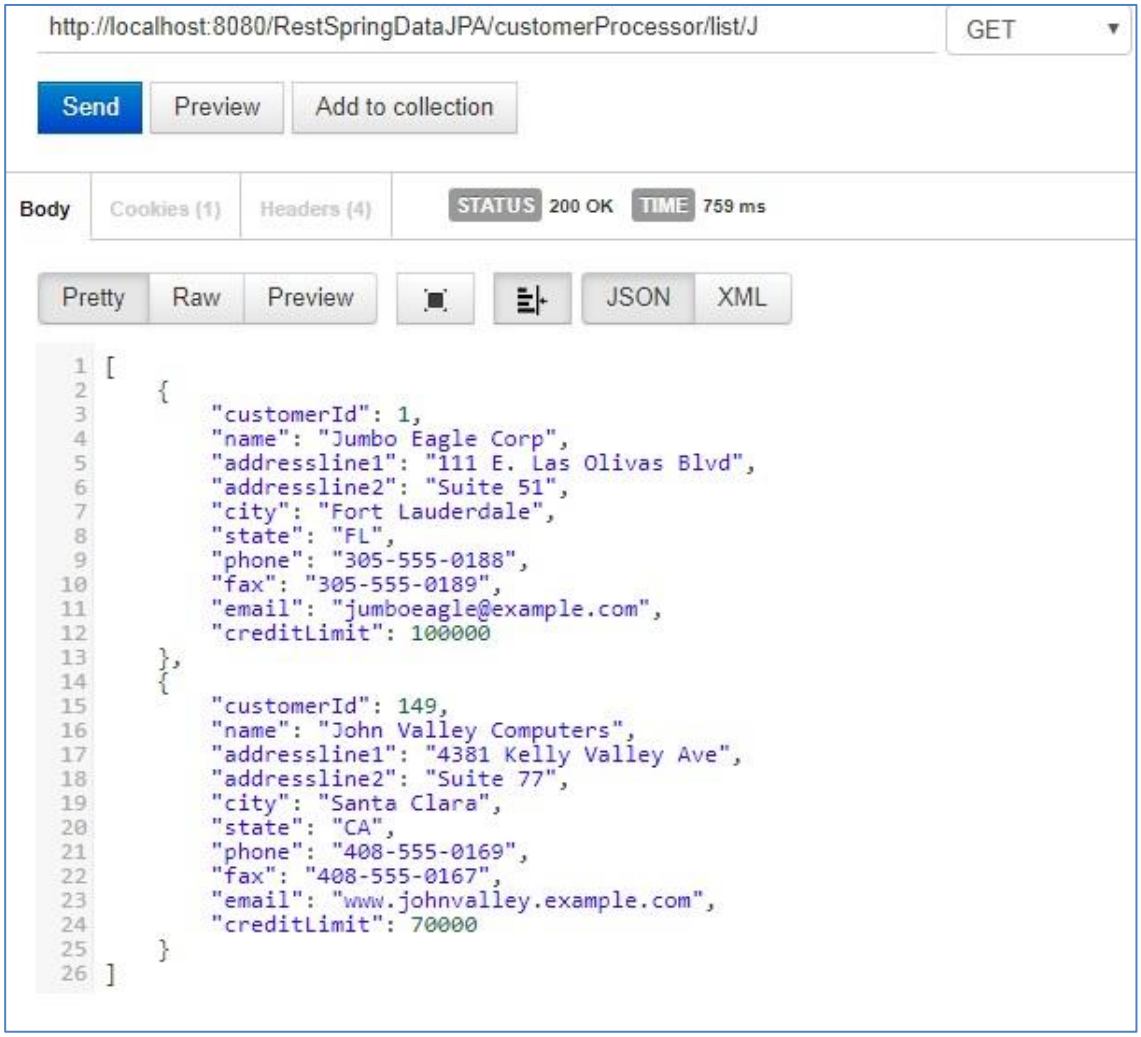


Рисунок 35. Результат работы web сервиса имитирующего бизнес-логику

Доступ к данным из БД с использованием RESTful Web Service широко распространен. Использование jsp для аналогичных целей вызывает много

трудностей, т.к. web страницы создают чаще всего с использованием JavaScript. Но мы приведем пример вывода результатов работы контроллера на страницу jsp для наглядности результатов. С этой целью в страницах jsp будем использовать таблицу стилей bootstrap.min.css и библиотеку JavaScript jquery-1.11.1.min.js с bootstrap.min.js.

Структура проекта показана на рис.36.

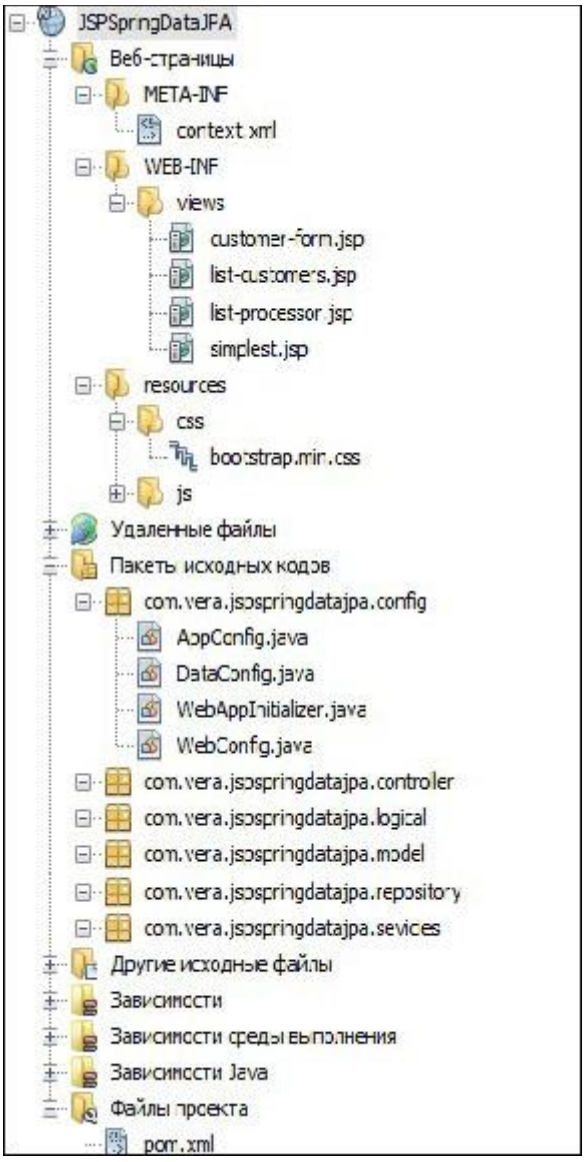


Рисунок 36. Структура проекта с выводом результатов на страницу jsp

В классе инициализации WebAppInitializer используется класс WebConfig, в котором определяются jsp страницы.

WebAppInitializer и WebConfig

```
import org.springframework.web.servlet.support.
AbstractAnnotationConfigDispatcherServletInitializer;
```

```
public class WebAppInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class[] getRootConfigClasses() {
        return new Class[] {WebConfig.class};
    }

    @Override
    protected Class[] getServletConfigClasses() {
       // return new Class[] { WebConfig.class };
        return new Class[0];
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }

}

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.vera.jspspringdatajpa")
public class WebConfig extends WebMvcConfigurerAdapter {

    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver viewResolver =
                    new InternalResourceViewResolver();
        viewResolver.setViewClass(JstlView.class);
        viewResolver.setPrefix("/WEB-INF/views/");
        viewResolver.setSuffix(".jsp");

        return viewResolver;
    }

    @Override
    public void configureDefaultServletHandling
                        (DefaultServletHandlerConfigurer configurer) {
        configurer.enable();
    }

}
```

Класс CustomerVeiwController формирует функции для работы с GRUD операциями. При этом используются методы класса CustomerService.

CustomerVeiwController

```
package com.vera.jspspringdatajpa.controller;

import com.vera.jspspringdatajpa.model.Customer;
import com.vera.jspspringdatajpa.sevicies.CustomerService;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
```

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
@RequestMapping("/customerView")
public class CustomerVeiwController {

    @Autowired
    CustomerService customerService;

    @GetMapping("/list")
    public String listCustomer(Model model) {
        List<Customer> customers = customerService.getAllCustomers();
        model.addAttribute("customers", customers);
        return "list-customers";
    }

    @GetMapping("/showForm")
    public String showForm(Model model) {
        Customer customer = new Customer();
        model.addAttribute("customer", customer);
        return "customer-form";
    }

    @PostMapping("/saveCustomer")
    public String saveCustomer(@ModelAttribute("customer") Customer
theCustomer) {
        customerService.saveCustomer(theCustomer);
        return "redirect:/customerView/list";
    }

    @GetMapping("/updateForm")
    public String showFormForUpdate(@RequestParam("customerId") int theId,
            Model theModel) {
        Customer theCustomer = customerService.getById(theId);
        theModel.addAttribute("customer", theCustomer);
        return "customer-form";
    }

    @GetMapping("/delete")
    public String deleteCustomer(@RequestParam("customerId") int theId) {
        customerService.deleteCustomer(theId);
        return "redirect:/customerView/list";
    }
}
```

В этом примере мы наруши наши правила о неизменности табицы customer БД sumple. Для вставки данных в таблицу будем имитировать связи с таблицами Discount и Zip и вводить не все поля записи таблицы customer. Для этого в метод saveCustomer класса CustomerService добавим установки полей класса Customer.

CustomerService метод saveCustomer

```
public boolean saveCustomer(Customer customer) {
            customer.setCreditLimit(9000);
            customer.setEmail("@gmail.com");
            customer.setFax("fax");
```

```
            customer.setState("FL");
            customer.setPhone("8900");
            customer.setDiscountCode("N");
            customer.setZip("48128");
            return customerRepository.save(customer) != null;
        }
```

Приведем страницу list-customers.jsp. Не будем подробно разбирать содержание страницы. Читателю знакомому с html и JQuary она понятна. В 37 строке осуществляется связь с методом контроллера, возвращающего список заказчиков.

list-customers.jsp

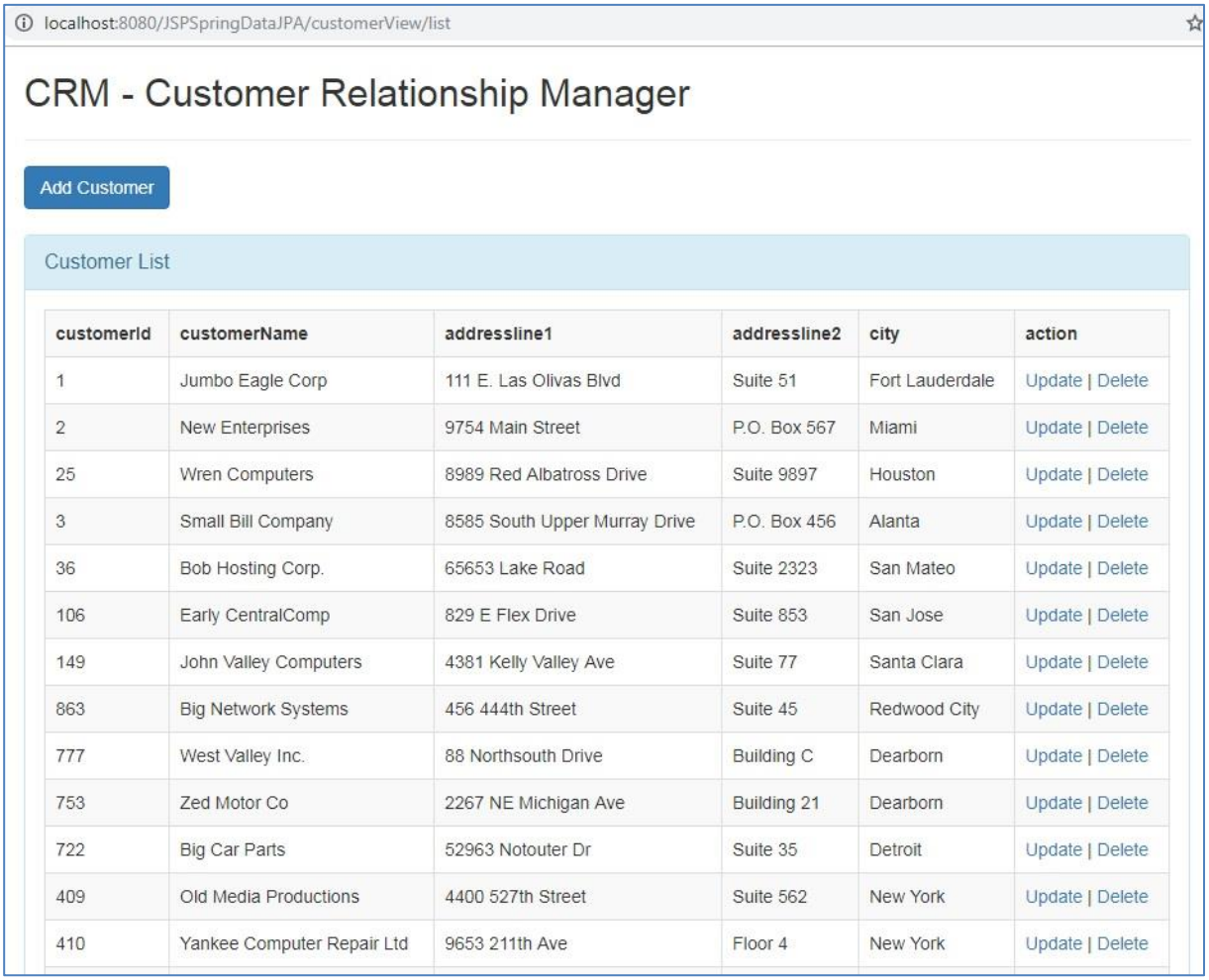
```
1.  <%@ page language="java" contentType="text/html; charset=windows-1251"
2.        pageEncoding="ISO-8859-1"%>
3.  <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

4.  <!DOCTYPE html>
5.  <html>
6.    <head>
7.      <meta http-equiv="Content-Type" content="text/html; charset=windows-1251">
8.      <title>javaguides.net</title>
9.      <link href="<c:url value="/resources/css/bootstrap.min.css" />" rel="stylesheet">
10.     <script src="<c:url value="/resources/js/jquery-1.11.1.min.js"/>"> </script>
11.     <script src="<c:url value="/resources/js/bootstrap.min.js"/>"></script>
12.   </head>
13.   <body>
14.     <div class="container">
15.       <div class="col-md-offset-1 col-md-10">
16.         <h2>CRM - Customer Relationship Manager</h2>
17.         <hr />
18.         <input type="button" value="Add Customer"
19.             onclick="window.location.href = 'showForm'; return false;"
20.             class="btn btn-primary" />
21.         <br/><br/>
22.         <div class="panel panel-info">
23.           <div class="panel-heading">
24.             <div class="panel-title">Customer List</div>
25.           </div>
26.           <div class="panel-body">
```

```
27.                 <table class="table table-striped table-bordered">
28.                     <tr>
29.                         <th>customerId</th>
30.                         <th>customerName</th>
31.                         <th>addressline1</th>
32.                         <th>addressline2</th>
33.                         <th>city</th>
34.                         <th>action</th>
35.                     </tr>
36.                     <!-- loop over and print our customers -->
37.                     <c:forEach var="tempCustomer" items="${customers}">
38.                     <c:url var="updateLink" value="/customerView/updateForm">
39.                             <c:param name="customerId"
40.                              value="${tempCustomer.customerId}" />
41.                     </c:url>
42.                     <c:url var="deleteLink" value="/customerView/delete">
43.                         <c:param name="customerId" value="${tempCustomer.customerId}" />
44.                      </c:url>
45.                         <tr>
46.                             <td>${tempCustomer.customerId}</td>
47.                             <td>${tempCustomer.name}</td>
48.                             <td>${tempCustomer.addressline1}</td>
49.                             <td>${tempCustomer.addressline2}</td>
50.                             <td>${tempCustomer.city}</td>
51.                             <td>
52.                                 <a href="${updateLink}">Update</a>
53.                                 | <a href="${deleteLink}"
54.                               onclick="if !(confirm('Are you sure you want to delete this customer?')))
55.                                 return false">Delete</a>
56.                             </td>
57.                         </tr>
58.                     </c:forEach>
59.                 </table>
60.             </div>
61.         </div>
62.     </div>
63. </div>
```

```
64.     </body>
        </html>
```

Результат работы данной страницы приведен на рис. 37.



customerId	customerName	addressline1	addressline2	city	action
1	Jumbo Eagle Corp	111 E. Las Olivas Blvd	Suite 51	Fort Lauderdale	Update | Delete
2	New Enterprises	9754 Main Street	P.O. Box 567	Miami	Update | Delete
25	Wren Computers	8989 Red Albatross Drive	Suite 9897	Houston	Update | Delete
3	Small Bill Company	8585 South Upper Murray Drive	P.O. Box 456	Alanta	Update | Delete
36	Bob Hosting Corp.	65653 Lake Road	Suite 2323	San Mateo	Update | Delete
106	Early CentralComp	829 E Flex Drive	Suite 853	San Jose	Update | Delete
149	John Valley Computers	4381 Kelly Valley Ave	Suite 77	Santa Clara	Update | Delete
863	Big Network Systems	456 444th Street	Suite 45	Redwood City	Update | Delete
777	West Valley Inc.	88 Northsouth Drive	Building C	Dearborn	Update | Delete
753	Zed Motor Co	2267 NE Michigan Ave	Building 21	Dearborn	Update | Delete
722	Big Car Parts	52963 Notouter Dr	Suite 35	Detroit	Update | Delete
409	Old Media Productions	4400 527th Street	Suite 562	New York	Update | Delete
410	Yankee Computer Repair Ltd	9653 211th Ave	Floor 4	New York	Update | Delete

Рисунок 37. Результат работы list-customers.jsp

При нажатии на кнопку Add Customer появляется форма (рис. 38) для ввода нового заказчика. Связь со страницей jsp поисходит в 19 строке onclick="window.location.href = 'showForm', где showForm аннотация метода showForm(Model model) класса CustomerVeiwController.

```
1.  <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2.  pageEncoding="ISO-8859-1"%>
3.  <%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
4.  <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
5.  <!DOCTYPE html>
6.  <html>
7.  <head>
```

```
8.  <meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
9.  <title>Spring MVC  - form </title>
10. <link href="<c:url value="/resources/css/bootstrap.min.css" />" rel="stylesheet">
11. <script src="<c:url value="/resources/js/jquery-1.11.1.min.js" />"></script>
12. <script src="<c:url value="/resources/js/bootstrap.min.js" />"></script>

13. </head>
14. <body>
15. <div class="container">
16.    <div class="col-md-offset-2 col-md-7">
17.        <h2 class="text-center">Spring MVC +  Spring Data Hibernate + JSP + Derby Example
18.        </h2>
19.     <div class="panel panel-info">
20.          <div class="panel-heading">
21.          <div class="panel-title">Add Customer</div>
22.     </div>
23.    <div class="panel-body">
24.        <form:form action="saveCustomer" cssClass="form-horizontal"
25.                                    method="post" modelAttribute="customer">
26.        <div class="form-group">
27.           <label for="customerId" class="col-md-3 control-label">customerId</label>
28.           <div class="col-md-9">
29.           <form:input path="customerId" cssClass="form-control" />
30.       </div>
31.    </div>
32.  <div class="form-group">
33.          <label for="name" class="col-md-3 control-label">      Name</label>
34.    <div class="col-md-9">
35.       <form:input path="name" cssClass="form-control" />
36.    </div>
37. </div>
38. <div class="form-group">
39. <label for="addressline1" class="col-md-3 control-label">addressline1</label>
40. <div class="col-md-9">
41. <form:input path="addressline1" cssClass="form-control" />
42. </div>
43. </div>
```

```
44. <div class="form-group">
45. <label for="addressline2" class="col-md-3 control-label">addressline2</label>
46. <div class="col-md-9">
47. <form:input path="addressline2" cssClass="form-control" />
48. </div>
49. </div>
50. <div class="form-group">
51. <label for="city" class="col-md-3 control-label">city</label>
52. <div class="col-md-9">
53. <form:input path="city" cssClass="form-control" />
54. </div>
55. </div>
56. <div class="form-group">
57. <!-- Button -->
58. <div class="col-md-offset-3 col-md-9">
59. <form:button cssClass="btn btn-primary">Submit</form:button>
60. </div>
61. </div>
62. </form:form>
63. </div>
64. </div>
65. </div>
66. </div>
67. </body>
    </html>
```

В 24 строке действие action="saveCustomer" указыает на метод saveCustomer класса CustomerVeiwController с аннотацией @PostMapping("/saveCustomer"). @ModelAttribute("customer") указывает на заказчика как атрибут модели.

Результат работы формы показан на рис. 38.

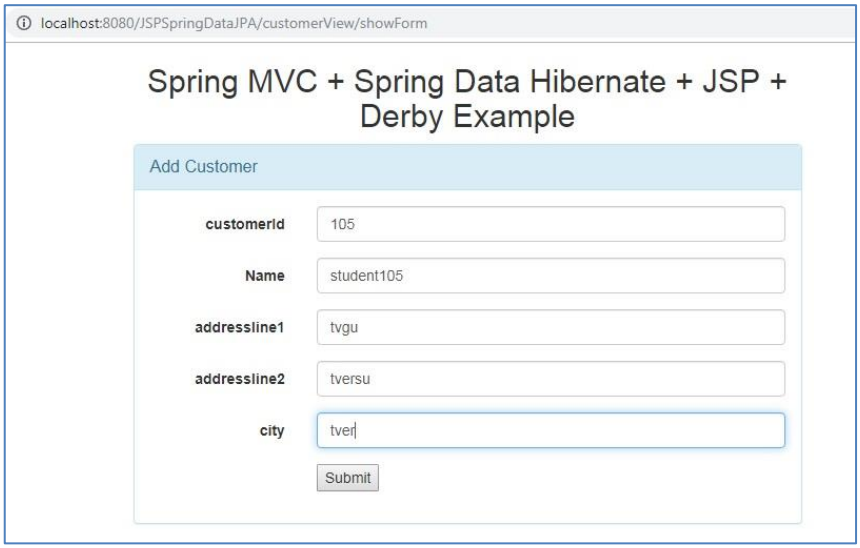


Рисунок 38. Форма ввода customer

Для имитации бизнес-логики (processor) создадим контроллер CustomerProcessorController. В результате работы метода listCustomer этого класса вызовется list-processor.jsp.

```
@Controller
@RequestMapping("/customerProcessorView")
public class CustomerProcessorController {
    @Autowired
    CustomerListProcessor listProcessor;
    @GetMapping ("/list/{prefix}")
    public String listCustomer(@PathVariable String prefix, Model model){
        List<Customer> customers = listProcessor.getCustomerList(prefix);
        model.addAttribute("customers",customers);
        return "list-processor";
    }
}
```

Результат работы list-processor.jsp приведен на рис.39. В выводимых заказчиках оказались лишь те, имена которых начинаются с буквы ‘J’.

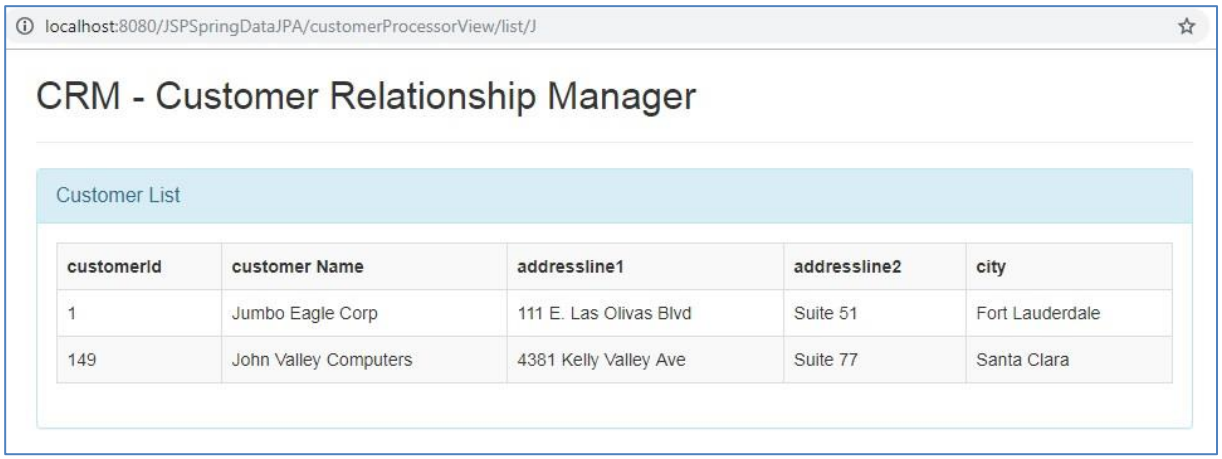


Рисунок 39. Результат работы форы ввода заказчика

На этом мы закончим рассмотрение примеров с испоьзованием Spring MVC. Надо заметить, что со Spring MVC интегрированы нормальные шаблонные движки, типа Thymeleaf, Freemaker, Mustache, плюс есть сторонние интеграции с кучей других. Так что никакого ужаса типа JSP или JSF писать не нужно.

Задания к гл. 8

1. Создайте серверное приложение для работы с БД Postgres. В БД должны быть 2 таблицы, реализующие связь один ко многим. Воспользуйтесь проектом из задания 3 гл.6. Реализуйте Rest сервисы для CRUD операции со всеми таблицами. Используйте Spring MVC.
2. Воспользуйтесь приложением, созданным в п.1. Создайте jsp-страницы для вывода и ввода данных таблиц.

9. Spring boot

Boot позволяет быстро создать и сконфигурить приложение, упаковать его в исполняемый jar файл. Исполняемость достигается за счет возможности встраивания http сервера (например, tomcat) в jar файл. Spring Boot характеризуется тем, что:

- Он не использует кодогенерацию. Из кода, который генерится, присутствует только метод main.
- Вся кофигурация осуществляется через аннотации (XML не используется) .
- Используется convention over configuration. Для большинства конфигураций не нужно ничего настраивать.

Начнём с “hello world”. Пример можно скачать с https://bitbucket.org/w2lvera/boothello_world/src/master/.

9.1 Spring MVC Hello Word

Структура проета показана на рис.40.

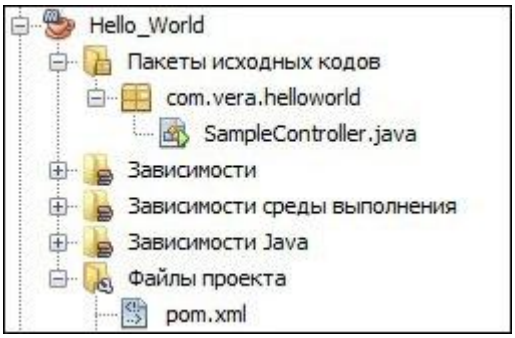


Рисунок 40. Структура проекта “hello world”

В проекте всего один исходный файл java – SampleController.

SampleController

```
package com.vera.helloworld;

import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;
@Controller
@EnableAutoConfiguration
public class SampleController {

    @RequestMapping("/")
  @ResponseBody
    String home() {
        return "Hello World!";
    }

    public static void main(String[] args) throws Exception {
        SpringApplication.run(SampleController.class, args);
    }
}
```

Приложение запускается методом SpringApplication.run.

@EnableAutoConfiguration – аннотация автоматической настройки. Автоматически настраивает бины, которые могут понадобиться. В данном случае мы используем встроенный tomcat (tomcat-embedd.jar), поэтому будет использоваться TomcatServletWebServerFactory bean.

@ ResponseBody - это аннотация Spring, которая связывает возвращаемое значение метода с телом веб-отклика. Аннотация использует конвертеры сообщений HTTP для преобразования возвращаемого значения в тело ответа HTTP на основе типа содержимого в заголовке HTTP запроса.

На основании этих двух аннотаций получается результат, показанный на рис.41.

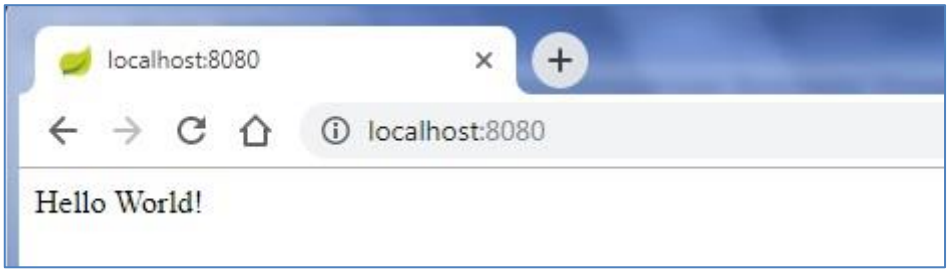


Рисунок 41. Результат “hello world” spring boot

Приведем содержимое pom.xml.

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns=http://maven.apache.org/POM/4.0.0
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>
    <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.2.RELEASE</version>
      <relativePath/> <!-- lookup parent from repository -->
    </parent>
  <groupId>com.vera</groupId>
    <artifactId>Hello_World</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
      <java.version>1.8</java.version>
    </properties>

    <dependencies>
      <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
      </dependency>

    </dependencies>


  <build>
      <plugins>
            <plugin>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
      </plugins>
    </build>

 </project>
```

Для быстрого построения проекта Spring Boot служит проект spring-boot-starter-parent. Это специальный стартовый проект, который предоставляет конфигурации по умолчанию для нашего приложения и полное дерево зависимостей.

spring-boot-starter-web - основа для создания веб, в том числе RESTful, приложений, использующих Spring MVC. Позволяет использовать Tomcat в качестве встроенного контейнера по умолчанию.

Важно отметить, что файл pom.xml может создаваться автоматически с помощью сайта https://start.spring.io/. Полученный в downloads zip архив можно импортировать в netbeans.

9.2 Создание CRUD приожения для БД Postgres

Для работы с БД postgres желательно иметь прогамму pgAdmin. Если таковой не имеется, то можно обойтись вкладкой «Службы» среды Netbeans, если пользоваться sql командами.

Будем писать rest сервис для выполнения CRUD операций с двумя таблицами «Parent» и «Childs», между которыми установлена связь один ко многим. Заметим, что родитель будет только мужского пола, т.е. мы будем иметь классическую зависимость «папа – дети». Таблицы нахоятся в БД boot.

DATABASE boot

```
CREATE DATABASE boot
  WITH OWNER = "user"
       ENCODING = 'UTF8'
       TABLESPACE = pg_default
       LC_COLLATE = 'Russian_Russia.1251'
       LC_CTYPE = 'Russian_Russia.1251'
       CONNECTION LIMIT = -1;
```

Для создания таблиц можно воспользоваться следующими командами SQL.

TABLE parent и TABLE childs

```
CREATE TABLE parent
(
  id bigint NOT NULL,
  created_at timestamp without time zone NOT NULL,
  updated_at timestamp without time zone NOT NULL,
  name character varying(100),
  CONSTRAINT parent_pkey PRIMARY KEY (id )
)
WITH (
  OIDS=FALSE
 );
 ALTER TABLE parent
 OWNER TO "user";
```

```
CREATE TABLE childs
(
  id bigint NOT NULL,
  created_at timestamp without time zone NOT NULL,
```

```
  updated_at timestamp without time zone NOT NULL,
  name character varying(100),
  parent_id bigint NOT NULL,
  CONSTRAINT childs_pkey PRIMARY KEY (id ),
  CONSTRAINT parent_fkey FOREIGN KEY (parent_id)
      REFERENCES parent (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE CASCADE
)
WITH (
  OIDS=FALSE
);ALTER TABLE childs
  OWNER TO "user";
```

Поля created_at и updated_at будут заполняться соответствующими датами. Для полей id каждой таблицы будут построены sequence. Поля «name» будут заполняться именами с использованием Rest сервисов. Владельцем таблиц является пользователь «user» с паролем «user». Поэтому необходимо создать такого пользователя.

```
      ROLE "user"
CREATE ROLE "user" LOGIN
  ENCRYPTED PASSWORD 'md55cc32e366c87c4cb49e4309b75f57d64'
  NOSUPERUSER INHERIT CREATEDB NOCREATEROLE NOREPLICATION;
COMMENT ON ROLE "user" IS 'parol user';
```

Для создания соединения с БД служит файл application.properties.

```
      application.properties
spring.datasource.url=jdbc:postgresql://localhost:5432/boot
spring.datasource.username = user
spring.datasource.password = user
# The SQL dialect makes Hibernate generate better SQL for the chosen
database
spring.jpa.properties.hibernate.dialect                                    =
org.hibernate.dialect.PostgreSQLDialect
# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto = update
```

В файле application.properties описан доступ к БД boot со схемой public для пользователя с именем user и аналогичным паролем. Строка `spring.jpa.hibernate.ddl-auto = update` означает, что в БД можно писать, стирать и изменять таблицы. Если таблицы в базе существуют, то они не создаются заново, а используются и могут изменяться. Если `create`, то таблицы создаются заново. Если `auto`, то тоже создаются заново.

Структура проекта показана на рис.42.

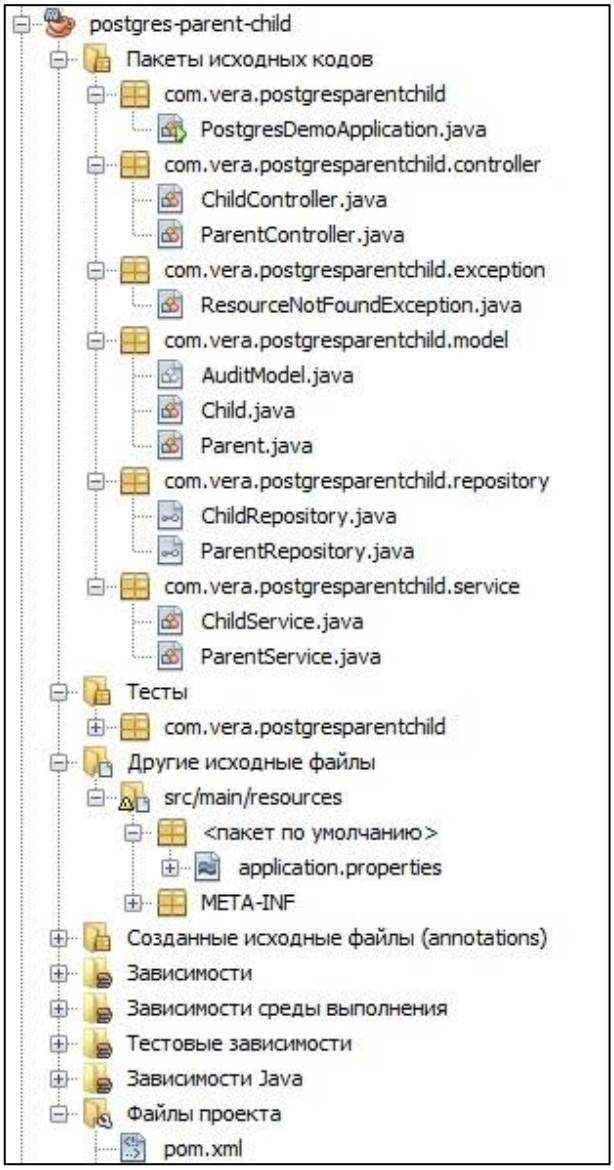


Рисунок 42. Структура проекта CRUD операций к таблицам Postgres

В структуре проекта присутствуют 4 слоя (пакета): контроллер, модель , репозиторий и слой сервисов. Проект можно скачать с https:// https://bitbucket.org/w2lvera/bootpostgres-parent-child/src/src/master/.

Такой проект практически не отличается от проектов MVC. Только запуск проекта осуществляется с помощью метода run объекта SpringApplication и отсутствуют конфигурационные файлы.

PostgresDemoApplication

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.jpa.repository.config.EnableJpaAuditing;

@SpringBootApplication
@EnableJpaAuditing
```

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

Файл pom.xml был сформирован на сайте https://start.spring.io/.

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>

	<groupId>com.example</groupId>
	<artifactId>postgres-demo</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<packaging>jar</packaging>

	<name>postgres-parent-child</name>
	<description>Demo project for Spring Boot</description>

	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>2.1.2.RELEASE</version>
		<relativePath/> <!-- lookup parent from repository -->
	</parent>

	<properties>
	<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
	<project.reporting.outputEncoding>UTF-
</project.reporting.outputEncoding>
		<java.version>1.8</java.version>
	</properties>

	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-data-jpa</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>

		<dependency>
			<groupId>org.postgresql</groupId>
			<artifactId>postgresql</artifactId>
			<scope>runtime</scope>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
		</dependency>
	 <dependency>
	  <groupId>org.eclipse.persistence</groupId>
	  <artifactId>eclipselink</artifactId>
```

```
        <version>2.5.2</version>
       </dependency>
       <dependency>
        <groupId>org.eclipse.persistence</groupId>

<artifactId>org.eclipse.persistence.jpa.modelgen.processor</artifactId>
        <version>2.5.2</version>
        <scope>provided</scope>
       </dependency>
      </dependencies>

      <build>
            <plugins>
                  <plugin>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-maven-plugin</artifactId>
                  </plugin>
            </plugins>
      </build>


</project>
```

Классы сущьностей описаны с помощью javax.persistence. Аннотация @SequenceGenerator служит для создания последовательности для автоматической генерации id. При создании экземпляра класса Parent создается последовательность с именем parent_sequence. Это можно увидеть в программе pgAdmin либо воспользоваться sql в netbeans.

class Parent

```
package com.vera.postgresparentchild.model;

import javax.persistence.*;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Size;

@Entity
@Table(name = "parent")
public class Parent extends AuditModel {

    @Id
    @GeneratedValue(generator = "parent_generator")
    @SequenceGenerator(
            name = "parent_generator",
            sequenceName = "parent_sequence"
           , initialValue = 1
    )
    private Long id;
    @NotBlank
    @Size(min = 3, max = 100)
    private String name;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
```

```
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

При создании экземпляра класса Child создается последовательность с именем childs_sequence. В классе присутствует ссылка на объект Parent. Это необходимо для реализации связи «многие к одному», которая задается аннотацией `@ManyToOne`. Когда JPA загружает объект, он также загружает все связи "join fetch". Стратегии загрузки (fetch) бывает две: EAGER и LAZY. В первом случае объекты коллекции сразу загружаются в память, во втором случае только при обращении к ним.

class Child

```
package com.vera.postgresparentchild.model;

import com.fasterxml.jackson.annotation.JsonIgnore;
import org.hibernate.annotations.OnDelete;
import org.hibernate.annotations.OnDeleteAction;
import javax.persistence.*;
@Entity
@Table(name = "childs")

public class Child extends AuditModel{

    @Id
    @Basic(optional = false)
    @Column(name = "id")
    @GeneratedValue(generator = "childs_generator")
    @SequenceGenerator(
            name = "childs_generator",
            sequenceName = "childs_sequence",
            initialValue = 1000
    )
    private Long id;

    @Column(name = "name")
    private String name;

    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "parent_id", nullable = false)
    @OnDelete(action = OnDeleteAction.CASCADE)
    @JsonIgnore
    private Parent parent;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }
    public Parent getParent() {
        return parent;
    }

    public void setParent(Parent parent) {
        this.parent = parent;
    }

}
```

CRUD операции выполняются в слое севисов. Рассмотрим сервис родителя.

class ParentService

```
package com.vera.postgresparentchild.service;

import com.vera.postgresparentchild.exception.ResourceNotFoundException;
import com.vera.postgresparentchild.model.Parent;
import com.vera.postgresparentchild.repository.ParentRepository;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class ParentService {

    @Autowired
    ParentRepository parentRepository;

    @Transactional
    public List<Parent> getAllParents() {
        return parentRepository.findAll();
    }

    @Transactional
    public Parent saveParent(Parent parent) {
        return parentRepository.save(parent);
    }

    @Transactional
    public Parent updateParent(Long parentId,
            Parent parentRequest) {
        return parentRepository.findById(parentId)
                .map(parent -> {
                    parent.setName(parentRequest.getName());
                    return parentRepository.save(parent);
                }).orElseThrow(()
                        -> new ResourceNotFoundException(
                                "Parent not found with id " + parentId));
    }

    @Transactional
    public ResponseEntity<?> deleteQuestion(Long parentId) {
        return parentRepository.findById(parentId)
                .map(parent -> {
```

```
                    parentRepository.delete(parent);
                    return ResponseEntity.ok().build();
                }).orElseThrow(()
                        -> new ResourceNotFoundException(
                                "Parent not found with id " + parentId));
    }

}
```

Методы **getAllParents** и **saveParent** осуществляют вывод всех родителей из таблицы и вставку одного родителя в таблицу. Метод **updateParent** изменяет родителя с введенным id. Метод **deleteQuestion** стирает родителя с введенным id.

Сервис ребенка выглядит сложнее, т.к. в нем надо учитывать помимо ребенка еще и родителя.

```
      class ChildService
import com.vera.postgresparentchild.exception.ResourceNotFoundException;
import com.vera.postgresparentchild.model.Child;
import com.vera.postgresparentchild.repository.ChildRepository;
import com.vera.postgresparentchild.repository.ParentRepository;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Service;

@Service
public class ChildService {
    @Autowired
    private ChildRepository childRepository;
    @Autowired
    private ParentRepository parentRepository;

    public List<Child> getAllChilds() {
        return childRepository.findAll();
    }

    public List<Child> getChildsByParentId(Long parentId) {
        return childRepository.findByParentId(parentId);
    }

    public Child addChild( Long parentId, Child child) {
        return parentRepository.findById(parentId)
                .map(parent -> {
                    child.setParent(parent);
                    return childRepository.save(child);
                }).orElseThrow(() -> new ResourceNotFoundException(
                        "Parent not found with id " + parentId));
    }

    public    Child    updateChild(    Long    parentId,Long    childId,Child
childRequest) {
        if(!parentRepository.existsById(parentId)) {
            throw new ResourceNotFoundException(
```

```
                    "Parent not found with id " + parentId);
        }
        return childRepository.findById(childId)
                .map(child -> {
                    child.setName(childRequest.getName());
                    return childRepository.save(child);
                }).orElseThrow(() -> new ResourceNotFoundException(
                        "Child not found with id " + childId));
    }

    public ResponseEntity<?> deleteChild( Long parentId,Long childId) {
        if(!parentRepository.existsById(parentId)) {
            throw new ResourceNotFoundException(
                    "Parent not found with id " + parentId);
        }
        return childRepository.findById(childId)
                .map(child -> {
                    childRepository.delete(child);
                    return ResponseEntity.ok().build();
                }).orElseThrow(() -> new ResourceNotFoundException(
                        "Child not found with id " + childId));

    }
}
```

Метод **getAllChilds** возвращает всех детей из таблицы childs.

Метод **getChildsByParentId** возвращает всех детей отца, id которого задано параметром функции.

Метод **addChild** добавляет ребенка в таблицу. При этом отец, заданный парметром **parentid** и найденный **parentRepository.findById(parentId),** прописывается в объект **child** методом **setParent.** И только после этого объект **child** сохраняется в **childRepository** с помощью метода **save.**

Метод **updateChild** изменяет имя ребенка, но перед этим проверяет, есть ли отец, с **id** заданным в параметре **(parentRepository.existsById(parented)).** Если такой отец существует, то изменяется имя у найденного по **id** ребенка и ребенок сохраняется в репозитории **(childRepository.save(child)).**

Метод **deleteChild** удаляет из таблицы ребенка, заданного с помощью двух **id** – отца и ребенка**.** Как и в предыдущей функции проверяется наличие отца.

Rest контроллеры для работы с двумя сервисами представлены двумя классами **ParentController** и **ChildController.** В этих контроллерах происходит обращение к рассмотренным выше методам соответствующих сервисов.

class ParentController и class ChildController

```
package com.vera.postgresparentchild.controller;
import com.vera.postgresparentchild.exception.ResourceNotFoundException;
import com.vera.postgresparentchild.model.Parent;
import com.vera.postgresparentchild.repository.ParentRepository;
import com.vera.postgresparentchild.service.ParentService;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import javax.validation.Valid;

@RestController
public class ParentController {

    @Autowired
    private ParentService parentService;
    @GetMapping("/parents")
    public List<Parent> getParents() {
        return parentService.getAllParents();
    }

    @PostMapping("/parents")
    public Parent createParent(@Valid @RequestBody Parent parent) {
        return parentService.saveParent(parent);
    }

    @PutMapping("/parents/{parentId}")
    public Parent updateParent(@PathVariable Long parentId,
                               @Valid @RequestBody Parent parentRequest)
{
        return parentService.updateParent(parentId, parentRequest);
    }

    @DeleteMapping("/parents/{parentId}")
    public ResponseEntity<?> deleteQuestion(@PathVariable Long parentId) {
        return parentService.deleteQuestion(parentId);
    }
}

package com.vera.postgresparentchild.controller;
import com.vera.postgresparentchild.model.Child;
import com.vera.postgresparentchild.service.ChildService;
import com.vera.postgresparentchild.service.ParentService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import javax.validation.Valid;
import java.util.List;
```

```
@RestController
public class ChildController {

    @Autowired
    private ChildService  childService;

    @Autowired
    private ParentService parentService;

     @GetMapping("/childs")
    public List<Child> getAllChilds() {
        return childService.getAllChilds();
    }

    @GetMapping("/childs/{parentId}/childs")
    public List<Child> getChildsByParentId(@PathVariable Long parentId) {
        return childService.getChildsByParentId(parentId);
    }

    @PostMapping("/childs/{parentId}/childs")
    public Child addChild(@PathVariable Long parentId,
                          @Valid @RequestBody Child child) {
           return childService.addChild(parentId, child);
    }

    @PutMapping("/childs/{parentId}/childs/{childId}")
    public Child updateAnswer(@PathVariable Long parentId,
                              @PathVariable Long childId,
                              @Valid @RequestBody Child childRequest) {
        return childService.updateChild(parentId, childId, childRequest);
    }

    @DeleteMapping("/childs/{parentId}/childs/{childId}")
    public ResponseEntity<?> deleteChild(@PathVariable Long parentId,
                                         @PathVariable Long childId) {
        return childService.deleteChild(parentId, childId);
    }

}
```

Рассмотрим метод, который создает родителя. Для вызова метода используется аннотация `@PostMapping("/parents")`.Объект `parent` не передается в командной строке, а передается парой имя поля – значение. В данном случае у родителя и у ребенка есть только одно поле – name, т.к. `id` генерируется автомаически с помощью sequence. Вызов Rest сервиса parents с помощью программы Postman показан на рис.43.

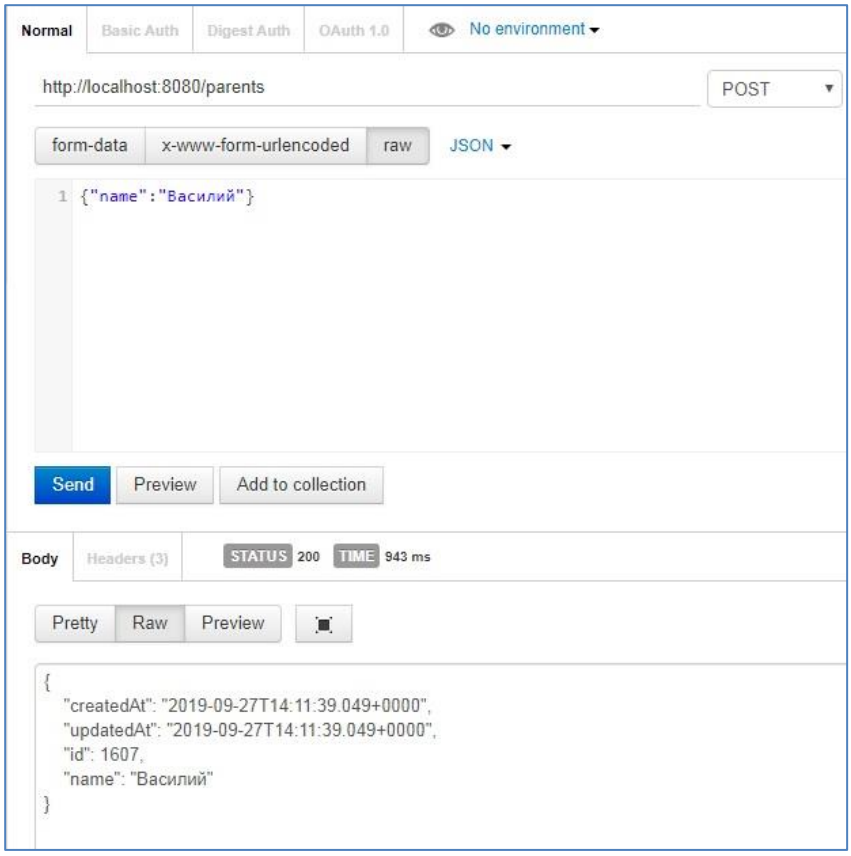


Рисунок 43. Вызов rest сервиса по созданию parent

Мы добавили отца с именем Василий. С помощью rest контроллера класса child добавим детей этому родителю как показано на рис.44.

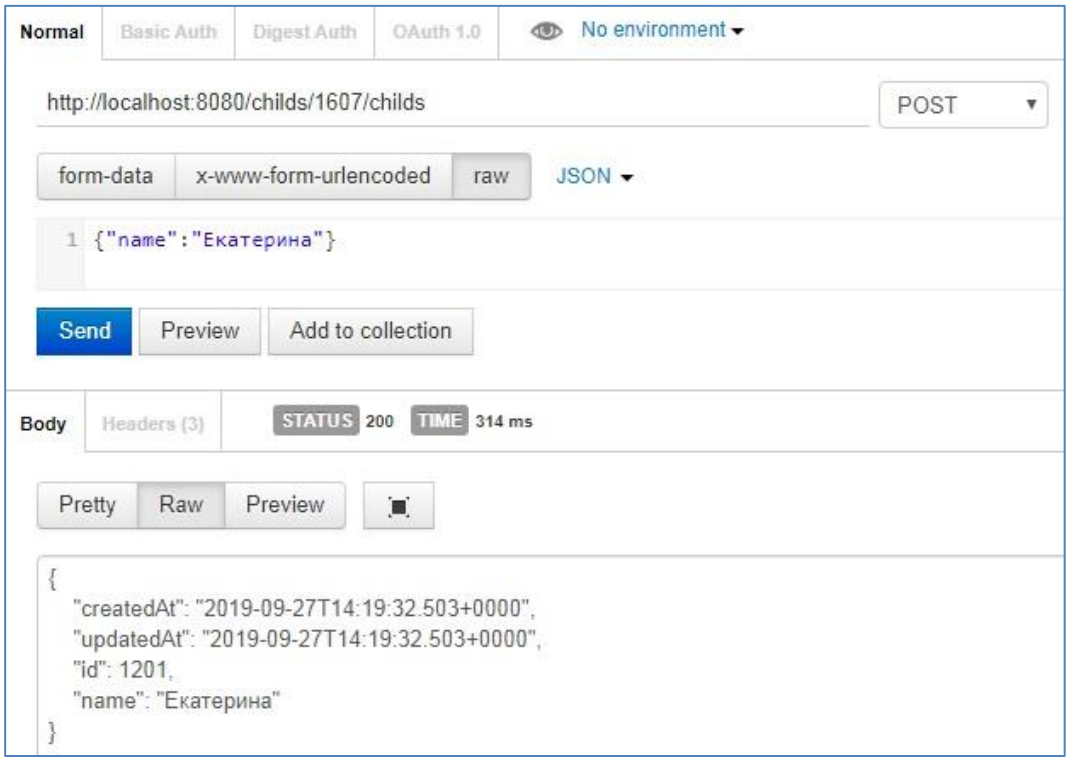


Рисунок 44. Добавление child

На рис. 45 показан вызов метода по удалению записи родителя с id -36.

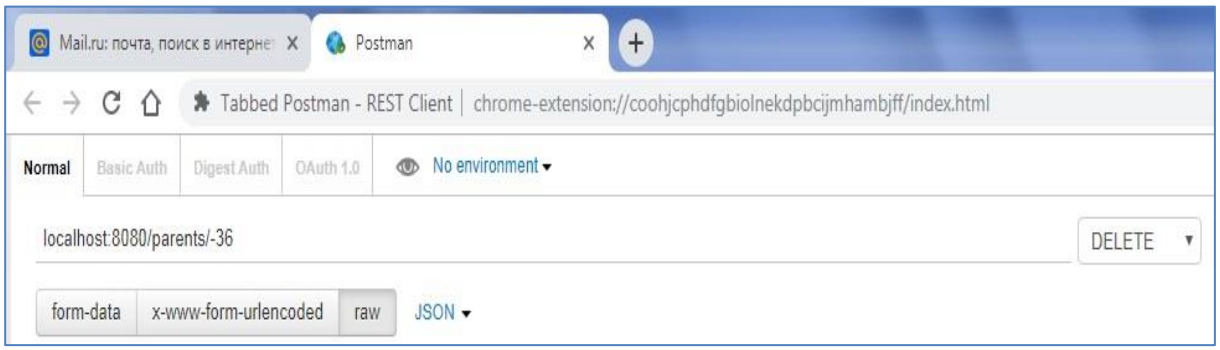


Рисунок 45. Удаление записи из таблицы parent

Отметим одну особенность при посылке post и put сообщений серверу. Помимо информации, которая показана на рис. 43 и 44, необходимо задать заголовок (слева кнопка “headers”) как показано на рис.46.

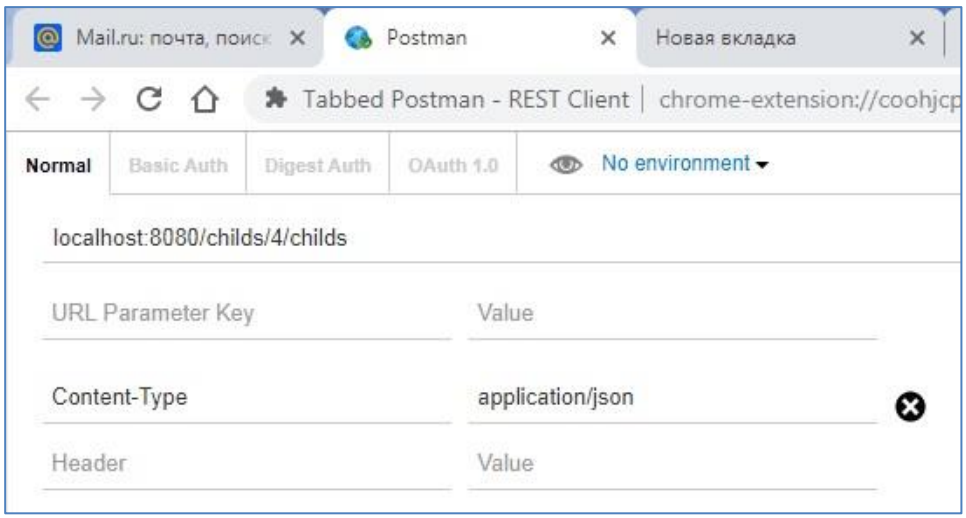


Рисунок 46. Особенности посылки post, put методов из Postman

И так мы рассмотрели простейшие примеры с spring boot. Надеемся, что этот фреймворк, позволит вам думать, что *Spring Framework* вполне себе быстрый и комфортный процесс по созданию web приложений с использованием сервера приложений.

Задания к гл. 9

1. Создайте серверное приложение для работы с БД Postgres. В БД должны быть 2 таблицы, реализующие связь один ко многим.

Воспользуйтесь проектом из задания 1 гл.8. Реализуйте Rest сервисы для CRUD операции со всеми таблицами. Используйте Spring boot.

2. Воспользуйтесь приложением, созданным в п.1. Создайте jsp-страницы для вывода и ввода данных таблиц. Используйте Spring boot.

Заключение

Рассмотренные архитектурные решения Java, включающие в себя паттерны DAO, IoC и фреймворки Spring и Hibernate, позволяют упростить процесс проектирования корпоративного приложения. Кроме того фреймворки избавляют такое приложение от множества ошибок. Представленные в пособии примеры помогают сделать первые шаги в освоении Spring и Hibernate.

Литература

1. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объекто-ориентированного программирования. Паттерны проектирования. – СПб.: Питер, 2016. – 366с.
2. Э. Фримен, Э. Фримен, К. Сьерра, Б. Бейтс Head First. Паттерны проектирования. Обновленное юбилейное издание. – Питер, 2018. – 656с.
3. Волушкова В.Л. Технологии программирования. Введение в паттерны проектирования java: учеб. пособие. – Тверь: Твер.гос.ун-т, 2013 – 100с.
4. Object-Oriented Programming — The Trillion Dollar Disaster [Электронный ресурс] — Режим доступа: https://medium.com/better-programming/object-oriented-programming-the-trillion-dollar-disaster-92a4b666c7c7.
5. Мартин Фаулер Шаблоны корпоративных приложений. – Диалектика, 2018. – 544с.
6. Java Persistence API[Электронный ресурс] — Режим доступа: https://easyjava.ru/data/jpa/.
7. Грегори Гэри, Бауэр Кристиан, Кинг Гэвин Java Persistence API и Hibernate .– ДМК Пресс, 2018. – 632с.
8. Шаблон проектирования [Электронный ресурс] — Режим доступа: https://ru.wikipedia.org/wiki/pattern

Приложение 1. Глоссарий

Apache сервер (A PAtCHy Web Server –исправленный Web Server) – новая версия NCSA 1995 год.

ASF (Apache Software Faundation) – программный фонд. Был создан в 1999 году из Apache Group в Делавэре, США. Отличительные черты проектов Apache — это совместная разработка кода и открытая, прагматичная лицензия — Apache Software License.

ASL лицензия – лицензия проектов ASF.

CRUD – обозначение SQL операции (create, read, update, delete).

Hibernate – одна из реализаций ORM.

DOM (Document Object Model) – объектная модель HTML страницы.

Java2EE (Java 2 Enterprise Edition) – набор спецификаций Java, описывающий архитектуру серверной платформы для задач средних и крупных предприятий.

javax.persistence – библиотека, которая дает возможность сформировать классы, описывающие уже существующие таблицы в БД.

JPA (Java Persistence API) – спецификация API Java EE. Установливает правила, как должно реализовываться ORM.

JQuery – библиотека JavaScript, фокусирующаяся на взаимодействии JavaScript и HTML. Библиотека jQuery помогает легко получать доступ к любому элементу DOM, обращаться к атрибутам и содержимому элементов DOM, манипулировать ими.

JSON (JavaScript Object Notation) – текстовый формат обмена данными, основанный на JavaScript. Json - текст представляет собой набор ключ: значение.

Mapping – сопоставление (проецирование) Java-классов с таблицами базы данных. Осуществляется с помощью конфигурационных XML-файлов или Java-аннотаций.

NCSA – один из первых веб-серверов. Разработан Роббом Маккулом из NCSA (национальный центр суперкомпьютерных технологий).

ORM (Object-Relational Mapping) – объектно-реляционное отображение. Отвечает за отображение объектов в БД.

POLO (Plain Old Java Object –Старые добрые Java-объекты) класс – обычный класс java.

REST service – преобладающая модель проектирования Web-сервисов. Следует четырем принципам: явное использование HTTP-методов, несохранение состояния, предоставление URI, аналогичных структуре каталогов, передача данных в XML, JSON или в обоих форматах.

UML-диаграмма – графическое описание объектной модели.

Веб-сервис (web service) — программная система, отображаемая браузером. Идентифицируется уникальным URL-адресом. Взаимодействует с другими системами посредстом сообщений, основанных на SOAP протоколах или на соглашениях REST.

Контейнер сервлетов – программа, представляющая собой сервер, который занимается системной поддержкой сервлетов. Может работать как полноценный самостоятельный веб-сервер, быть поставщиком страниц для другого веб-сервера, например, Apache, или интегрироваться в Java EE сервер приложений.

Паттерн – шаблон проектирования. Повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста [8].

Сервлет – это Java программа, которая выполняется в контейнере сервлетов.

Фреймворк – каркас (framework), не меняющийся от конфигурации к конфигурации и несущий в себе гнезда, в которых размещаются сменные модули.

Приложение 2. Информация о стандартном ПО проектов.

2.1 Некоторые сведения о Tomcat

- Tomcat можно зарегистрировать как службу, а можно и не регистрировать.
- Tomcat может встраиваться в среду (NetBeans), а может не встраиваться из-за конфликта версий.
- Все web проекты с hibernate и с webConig (но не с xml) идут только под jre 8.
- Как запустить tomcat? Запустить файл startup.bat в каталоге tomcat/bin.

Как изменить порт, на которм стартует tomcat? Поменять порт в строке, которая находится в файле ..\ApacheTomcat 8.0.15\conf\server.xml:

```
<Connector executor="tomcatThreadPool"
      port="8080" protocol="HTTP/1.1"
      connectionTimeout="20000"
      redirectPort="8443" />
```

- Как поменять или добавить пароль для deploy?

В файл..\ApacheTomcat 8.0.15\conf\tomcat-users.xml добавить пользователя с ролью "manager-gui":

```
<role rolename="manager-gui"/>
<user username="admin" password="admin" roles="manager-
gui"/>
```

- Как поменять версию java?

В каталог..\ApacheTomcat 8.0.15\bin поместите файл setenv.bat со следующим содержимым:

```
set "JRE_HOME=C:\Program
Files\Java\jre1.8.0_201"
exit /b 0
```

При развертывании сервера версия jre поменяется. Это можно увидеть в окне tomcat http://localhost:8080/manager/status. столбец JVM Version.

- Как остановить tomcat?

Это можно сделать как минимум тремя способами.

1. Через кнопку «настроить» в панели в правом нижнем углу (не всегда).

2. Запустив файл shutdown.bat из каталога ..\ ApacheTomcat 8.0.15\bin.

3. Остановив службу.

- Как задеплоить war файл?

Зайти на tomcat http://localhost:8080/manager/html и в разделе, показанном на рис.47, выбрать war-файл.

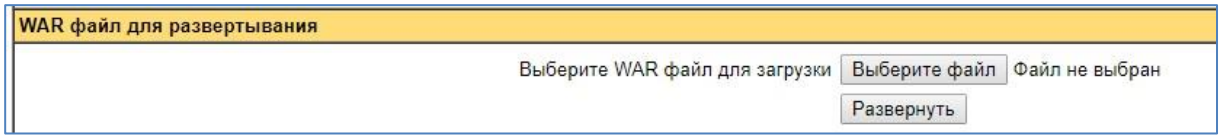


Рисунок 47. Размещение war-файла

2.2 О запуске приложений с доступом к derby.

При работе с NetBeans необходимо в «службах - базы данных» выполнить коннект с jdbc:derby://localhost:1527/sample [app на APP]. Без этого программный коннект не устанавливается.

Приложение 3. Проекты на bitbucket.org

Проекты 4 главы

Все проекты этой главы можно скачать с https://bitbucket.org/w2lvera/daosimplederby/.

- Проект с классом Controller находится в каталоге daoDerby.
- Проект с использованием Spring находится в каталоге daoDerbySpring.

Проекты 6 главы

Все проекты этой главы можно скачать с https://bitbucket.org/w2lvera/springdataaccessderby/src/master/. У проектов общий pom.xml.

- Проект раздела Spring JDBC Template находится в каталоге JDBCTemplateDerby.
- Проект раздела «Доступ к данным с Hibernate» находится в каталоге HibernateDerby.
- Проект раздела «Комбинация Spring Hibernate» находится в каталоге HibernateSpringBerby.
- Проект раздела Spring Data JPA находится в каталоге SpringData.
- Проект раздела Spring JDBC Template находится в каталоге JDBCTemplateDerby.

Проекты 8 главы

1. Проекты разделов 8.1 и 8.2 можно скачать с https://bitbucket.org/w2lvera/springmvcsimple/src/master/. У проектов общий файл pom.xml.

Проекты раздела Spring MVC Hello Word.

- Пример с xml файлом находится в каталоге mvc_xml.

- Пример с class Config находится в каталоге mvc_config.

Проект раздела RestServicies находится в каталоге RestServiceMVC.

2. Проекты раздела 8.3 Spring MVC с доступом к БД derby можно скачать с https://bitbucket.org/w2lvera/mvc_derby/src/master/. У проектов общий файл pom.xml.

- Проект с генерацией Rest service находится в каталоге RestSpringDataJPA.
- Проект с генерацией страниц jsp находится в каталоге JSPSpringDataJPA.

Проекты 9 главы

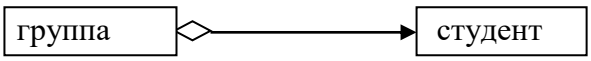
1. Проект раздела 9.1 можно скачать с https://bitbucket.org/w2lvera/boothello_world/src/master/
2. Проект раздела 9.2 можно скачать https://bitbucket.org/w2lvera/bootpostgres-parent-child/src/master/

Приложение 4. UML диаграммы классов

Для описания взаимодействия между классами используется UML диаграмма. Ниже приведены схемы, которые показывают возможные отношения между классами.

Классы связаны между собой различным образом:

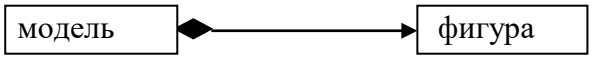
Агрегация



Агрегация (является частью) встречается, когда один класс является коллекцией или контейнером других. Причём по умолчанию, агрегацией называют *агрегацию по ссылке*, то есть когда время существования содержащихся классов не зависит от времени существования содержащего их класса. Если контейнер будет уничтожен, то его содержимое — нет.

Графически агрегация представляется пустым ромбиком на блоке класса и линией, идущей от этого ромбика к содержащемуся классу.

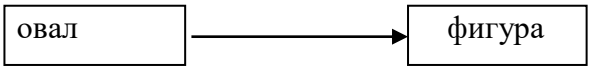
Композиция



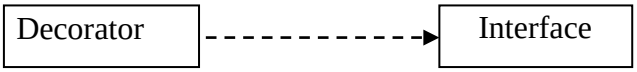
Композиция имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.

Графически представляется, как и агрегация, но с закрашенным ромбиком.

Наследование

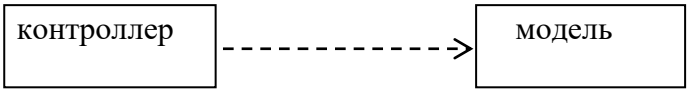


Для обозначения наследования используют треугольник, направленный от подкласса к родительскому классу (extends).



Для обозначения наследования интерфейса (implements).

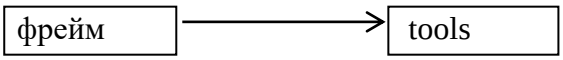
Реализация



Отношение между классами, в котором один класс создает экземпляр другого класса.

Реализация — отношение целое-часть. Графически реализация представляется так же, как и наследование, но с пунктирной линией.

Зависимость



Отношение осведомленности. Фрейм знает о классе tools.

Зависимость (dependency) — это слабая форма отношения использования, при котором изменение в спецификации одного влечёт за собой изменение другого, причем обратное не обязательно. Возникает, когда объект выступает, например, в форме параметра или локальной переменной.

Графически представляется пунктирной стрелкой, идущей от зависимого элемента к тому, от которого он зависит.

Существует несколько именованных вариантов.

Зависимость может быть между экземплярами, классами или экземпляром и классом.

Мощность отношений (Кратность).

Мощность отношения (мультипликатор) означает число связей между каждым экземпляром класса (объектом) в начале линии с экземпляром класса в её конце. Различают следующие типичные случаи:

нотация	*объяснение*	*пример*
0..1	Ноль или один экземпляр	кошка имеет или не имеет хозяина

1	Обязательно один экземпляр	у кошки одна мать
0..* или *****	Ноль или более экземпляров	у кошки может быть, а может и не быть котят
1..*	Один или более экземпляров	у кошки есть хотя бы одно место, где она спит

В.Л. Волушкова

Архитектурные решения Java для доступа к данным

Учебное пособие

Отпечатано с оригинала автора

Подписано в печать 03.10.2019. Формат 60x84 $^{1}/_{16}$.
Усл. печ. л. 7,96. Тираж 100. Заказ № 394.
Редакционно-издательское управление
Тверского государственного университета.
Адрес: 170100, г. Тверь, Студенческий пер. 12, корпус Б.
Тел. РИУ (4822) 35-60-63.