Министерство науки и высшего образования Российской Федерации
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

И.А. ВАСЮТКИНА

# РАЗРАБОТКА СЕРВЕРНОЙ ЧАСТИ WEB-ПРИЛОЖЕНИЙ НА JAVA

Утверждено
Редакционно-издательским советом университета
в качестве учебного пособия



НОВОСИБИРСК
2021

УДК 004.738.52:004.655.3(075.8)
В 206

Рецензенты:
канд. техн. наук, доцент *А.В. Гунько*,
канд. техн. наук, доцент *А.А. Малявко*

Работа подготовлена на кафедре вычислительной техники

**Васюткина И.А.**

В 206 Разработка серверной части web-приложений на Java : учебное пособие / И.А. Васюткина – Новосибирск : Изд-во НГТУ, 2021. – 83 с.

ISBN 978-5-7782-4394-1

Рассмотрены технологии и фреймворки, применяемые для разработки серверной части web-приложений на языке Java, а также основные приемы работы с базами данных через JDBC API и технологию Hibernate. Учебное пособие предназначено для студентов бакалавриата и магистратуры, обучающихся по направлениям 09.03.01, 09.04.01 – «Информатика и вычислительная техника».

**УДК 004.738.52:004.655.3(075.8)**

**ISBN 978-5-7782-4394-1**

# ПРЕДИСЛОВИЕ

Учебное пособие содержит изложение основных вопросов по программированию серверной части web-приложений на языке Java с использованием ряда Java технологий и фреймворков, применяемых в настоящее время для динамического отображения данных, в основном извлекаемых из базы данных на сервере и отправляемых клиенту.

Пособие рассчитано на студентов всех форм обучения направления «Информатика и вычислительная техника», знакомых с языком запросов SQL и основами языка Java.

Цель написания данного учебного пособия – дать единую теоретическую базу, необходимую для выполнения курсовых и лабораторных работ по курсам «Распределенные информационные системы и базы данных», «Базы данных», «Технологии и методы программирования».

Предлагаемое учебное пособие позволяет получить необходимые теоретические знания и опыт практической разработки web-приложений.

Каждый из восьми разделов учебного пособия включает в себя примеры программного кода и контрольные вопросы, необходимые для более глубокого понимания излагаемого материала и получения практических навыков его использования.

Основное внимание уделено вопросам разработки приложений, использующих паттерн проектирования MVC, а также ORM-технологию Hibernate для организации доступа к реляционным базам данных.

Рассматриваемые в учебном пособии средства разработки позволяют создавать современные полнофункциональные динамические web-приложения и корпоративные информационные системы.

# 1. ТЕХНОЛОГИЯ JAVA SERVLET

Создание Java-приложений на стороне сервера с самого начала появления клиент-серверных технологий основывалось на сервлетах – программах, которые так же, как и апплеты, могут выполняться в контейнере. Контейнер реализует настройку среды выполнения и запускает сервлет. В качестве контейнера для сервлетов может использоваться любой сервер, имеющий контейнер сервлетов. Часто для этих целей используется Apache Tomcat. Как апплеты, так и сервлеты должны писаться по определенным правилам, при этом сервлеты не должны иметь графический интерфейс. Они могут извлекать данные из форм страниц HTML, создавать страницы HTML и посылать их клиенту для отображения.

С появлением Java 2 Enterprise Edition (J2EE) сервлеты стали просто одной из многих Java технологий, доступных программистам. Enterprise JavaBeans (EJB) стали использоваться для создания сложных приложений, а сервлеты остались для использования в тех же целях, как и раньше.

## 1.1. ВОЗМОЖНОСТИ СЕРВЛЕТОВ

Сервлеты – это маленькие программы, которые выполняются на серверной стороне в контейнере. Являясь приложениями на Java, они импортируют различные пакеты библиотеки Java, могут писать на жесткий диск, обмениваться данными с базами данных. Через браузер можно запустить на сервере сервлет в контейнере для выполнения каких-либо действий.

Другое преимущество сервлетов – их хорошая масштабируемость. Контейнер отвечает за создание объекта класса сервлета, когда он потребуется, и запускает сервлет в потоках, которые создаются контейнером (рис.1.1).

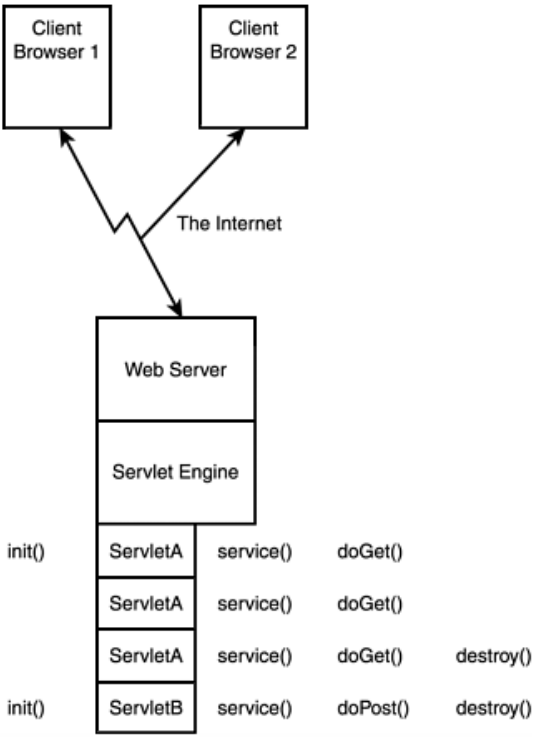



*Рис. 1.1.* Запуск сервлета в контейнере

Методы:

**init()** – служит для инициализации объекта сервлета, выполняется один раз, если запросов пользователей несколько;

**dectroy()** – служит для удаления объекта сервлета при отсутствии запросов к сервлету;

**service()** – основной метод сервлета, выполняющий его задачу. Запускается в потоках, если несколько пользователей обратились к данному сервлету. По умолчанию, сервлет считается поточно-безопасным, если он не реализует интерфейс SingleThreadModel. Метод service() вызывает методы doGet() или doPost(), в зависимости от типа запроса, переданного по HTTP протоколу.

При завершении работы потока ответ посылается Web-серверу, а тот посылает его браузеру в приложение.

Каждое HTTP-приложение выполняется в своем процессе. Процессы намного дороже создавать, чем потоки, поэтому подход, реализуемый сервлетами, считается более эффективным. Контейнеры также поддерживают управление соединениями, так что освободившиеся потоки используются для новых соединений.

## 1.2. ПРОГРАММИРОВАНИЕ СЕРВЛЕТОВ

Сервлеты – это классы, расширяющие классы Java GenericServlet или HttpServlet. Эти классы содержат всю необходимую функциональность для работы с контейнером. Наиболее значимыми являются следующие классы и интерфейсы:

- **Servlet** – это интерфейс, который определяет методы init(), service() и destroy(). Он также определяет еще два метода – getServletConfig() и getServletInfo(), позволяющих получить информацию о самом сервлете (автор, версия, дата создания и т. д.). Контейнер использует эти пять методов при работе с сервлетами;
- **HttpServlet** – самый используемый класс. Он расширяет GenericServlet, чтобы обеспечить HTTP-обработку. Обычно в сервлете надо перекрыть методы doGet(), doPost() или оба. В классе реализован метод service(), который вызывает необходимый метод doXXX, в зависимости от типа запроса;
- **ServletRequest** – это интерфейс, определенный для объекта, который содержит данные от клиента для запускаемого сервлета;
- **HttpServletRequestWrapper** – это расширение класса Servlet RequestWrapper, содержащее запрос в формате HTTP. В большинстве сервлетов этот класс доступен через ссылку на интерфейс Servlet Request;
- **ServletResponse** – этот интерфейс определяет удобный способ взаимодействия с данными, которые сервлет посылает клиенту через контейнер;
- **HttpServletResponseWrapper** – это расширение класса Servlet ResponseWrapper, полагается, что ответ будет также в формате HTTP. В большинстве сервлетов именно этот класс мы получаем по ссылке на интерфейс ServletResponse [1].

## 1.3. ПРИМЕР СЕРВЛЕТА

Напишем сервлет, который приветствует пользователя по введенному имени.

Страница index.html. Ввод имени пользователя и отправка его сервлету HelloServlet

```
<!DOCTYPE html>
<html>
  <head>
    <title>Пример Java Servlets</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <BODY BGCOLOR="#FDF5E6">
    <H1 align="CENTER">  Введите имя пользователя  </H1>
    <FORM action='HelloServlet'>
      Name:     <INPUT TYPE="TEXT" NAME="UserName"><BR><BR>
      <CENTER>  <INPUT TYPE="SUBMIT">    </CENTER>
    </FORM>
  </BODY>
</html>
```

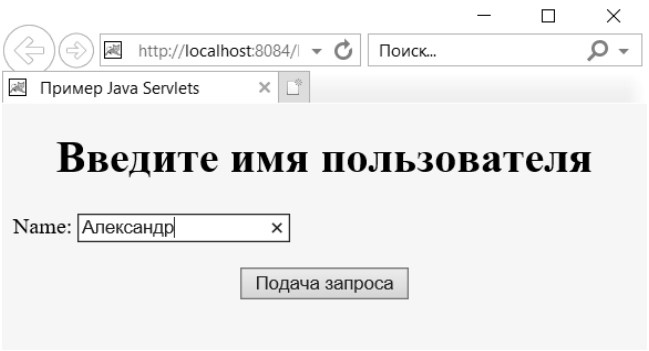



*Рис. 1.2.* Вид страницы в браузере

Сервлет HelloServlet получает запрос пользователя, извлекает из него имя и отправляет обратно в браузер клиента (рис. 1.3). Используемый метод посылки запроса по умолчанию GET(). Для размещения сервлетов в проекте создается отдельная папка /servlets.

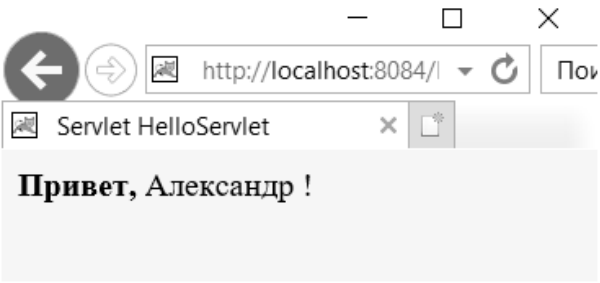



*Рис.1.3.* Вид страницы в браузере

```
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(urlPatterns = {"/HelloServlet"})
public class HelloServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)          throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
          // формирование HTML страницы ответа на запрос
        try (PrintWriter out = response.getWriter()) {
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet HelloServlet</title>");
            out.println("</head>");
            out.println("<body BGCOLOR=\"#FDF5E6\"\n>");
            out.println("<B>Привет,  </B>");
            out.println(request.getParameter("UserName"));
            out.println(" !");
            out.println("</body>");
            out.println("</html>");
```

```
        }
    }
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }
    @Override
    public String getServletInfo() {
        return "Short description";
    }
}
```

## 1.4. ИСПОЛЬЗОВАНИЕ COOKIES

Протокол HTTP, используемый в web-приложениях, не сохраняет состояние, а позволяет реализовать простую и быструю доставку документов клиентам по требованию. Протокол работает по принципу запрос-ответ (рис. 1.4.)

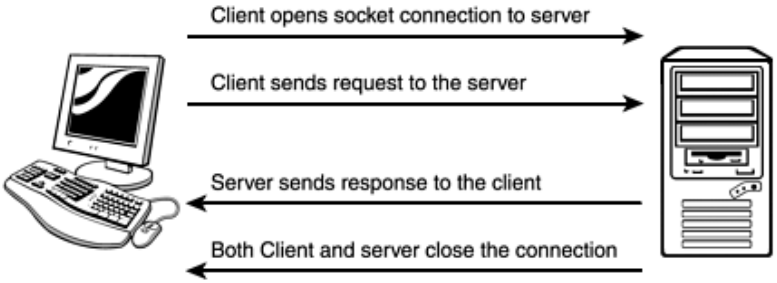



*Рис. 1.4.* Принцип работы протокола HTTP

Как же быть, если необходимо связать передачу нескольких страниц между собой или идентифицировать пользователя и вернуться к определенной странице, которая предназначена для него?

В этом случае можно использовать cookies. Cookies – это маленькие файлы, которые располагаются на жестком диске клиентской машины. Сервер создает и посылает cookie браузеру в ответ на первый запрос. При последовательных запросах браузер посылает серверу те

же самые cookies, что позволяет серверу определять, от кого пришел запрос.

Java поддерживает манипулирование cookies с помощью класса javax.servlet.http.Cookie. Наиболее часто используются методы этого класса: конструктор, setValue() и getValue(). Перепишем метод сервлета processRequest(), добавив создание cookies.

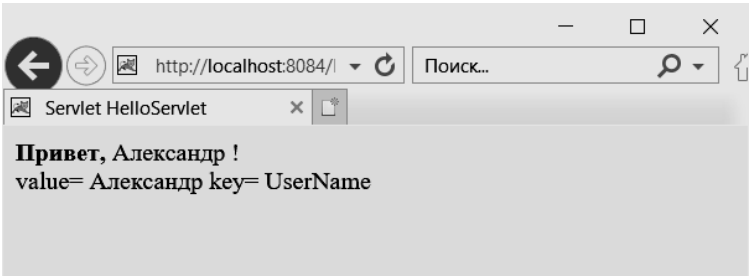



*Рис. 1.5.* Пересылка cookies клиенту

```
protected void processRequest(HttpServletRequest request,
HttpServletResponse response)          throws ServletException, IOException {
      response.setContentType("text/html;charset=UTF-8");
      try (PrintWriter out = response.getWriter()) {
         out.println("<!DOCTYPE html>");
         out.println("<html>");
         out.println("<head>");
         out.println("<title>Servlet HelloServlet</title>");
         out.println("</head>");
         out.println("<body BGCOLOR=\"#FDF5E6\"\n>");
         out.println("<B>Привет,  </B>");
         out.println(request.getParameter("UserName"));
         out.println(" !");
              // извлечение переменных и их значений из запроса
         String key, value;
         Enumeration keys = request.getParameterNames();
         while (keys.hasMoreElements()) {
            key = (String) keys.nextElement();
            value = request.getParameter(key);
            if (!key.equals("btnSubmit")) {
               out.println("<br>" +"value= " + value + " key= " + key);
               myCookie = new Cookie(value, key);   // создаем cookies
```

```
            response.addCookie(myCookie);       // отправляем в браузер
        }
      }
      out.println("</body>");
      out.println("</html>");
   }
```

Для извлечения cookies из запроса пользователя вызывается метод getCookies() класса HttpServletRequest для получения массива cookies.

```
Cookie cookies[];
cookies = request.getCookies();
for (int i=0; i< cookies.length; i++)       {
  out.println(cookies[i].getName()+"  " + cookies[i].getValue() + "<BR>");
}
```

## 1.5. ИСПОЛЬЗОВАНИЕ СЕССИЙ

Поддержка сервлетов в Java позволяет использовать интерфейс HttpSession. Наиболее важные методы интерфейса – конструктор, getAttribute() и setAttribute(). Сервлет создает сессию клиента на сервере и помещает данные в нее. При последующих обращениях к серверу браузер имеет доступ к сохраненным данным. Сессии используют cookie только для хранения идентификатора сессии. Вся остальная информация хранится на сервере, а это значит, что при перезапуске сервера все данные пропадут.

Перепишем метод сервлета processRequest(), добавив создание session для получения данных из запроса и сохранения их в сессии.

```
protected void processRequest(HttpServletRequest request, HttpServletResponse
response)          throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    try (PrintWriter out = response.getWriter()) {
      out.println("<!DOCTYPE html>");
      out.println("<html>");
      out.println("<head>");
      out.println("<title>Servlet HelloServlet</title>");
      out.println("</head>");
      out.println("<body BGCOLOR=\"#FDF5E6\"\n>");
      out.println("<B>Привет,  </B>");
      out.println(request.getParameter("UserName"));
```

```
        out.println(" !");

        HttpSession httpSess = request.getSession(true);      // создание сессии
            // извлечение переменных и их значений из запроса
        String key, value;
        Enumeration keys = request.getParameterNames();
        while (keys.hasMoreElements()) {
           key = (String) keys.nextElement();
           value = request.getParameter(key);
           if (!key.equals("btnSubmit")) {
              httpSess.setAttribute(key, value);  // запись в сессию переменных
              out.println("<br>" + "value= " + value + " key= " + key);
           }
        }
        out.println("</body>");
        out.println("</html>");
    }
```

getSession(true) указывает, что надо создать новый объект сессии, если его нет для этого клиента. Если же сессия уже открыта, то новая не создается.

```
Чтение данных из сесии:
HttpSession httpSess = request.getSession(true);
Enumeration keys = httpSess.getAttributeNames();
while (keys.hasMoreElements()) {
       String name = (String)e.nextElement();
       String value = session.getAttribute(name).toString();
  }
```

Метод invalidate удаляет данные в сессии. В противном случае она накапливает данные при каждом запуске сервлета.

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое сервлет? Основные методы сервлета?
2. В чем разница между приложением Java и сервлетом?
3. Что такое контейнер сервлетов? Опишите жизненный цикл сервлетов.
4. Как выполняется сервлет при обращении к нему одновременно нескольких клиентов?

5. Когда вызывается метод doGET(), а когда doPost()? В чем отличие?
6. Когда удаляется сервлет?
7. В чем разница между классами HTTPServlet и GenericServlet?
8. Что такое сессия?
9. Что такое cookies?
10. В чем разница между cookies и сессией?

# 2. ТЕХНОЛОГИЯ JAVA SERVER PAGES

Java Server Pages – это технология, позволяющая создавать динамические веб-страницы, обрабатываемые на сервере.

JavaServer Pages – это текстовые файлы с обычным HTML-кодом, в которые могут быть вставлены фрагменты Java-кода с помощью специальных тегов JSP [1]. Страница JSP транслируется в сервлет при первом обращении. И далее вызывается и работает как сервлет в контейнере сервлетов до перезапуска сервера. В результате выполнения сервлета создается HTML код.

JSP может использовать классы и библиотеки классов Java на серверной стороне. А также имеется возможность создавать свои теги для расширения функциональности JSP.

**Достоинства JSP**

- Использует обычный HTML.
- Можно написать код сервлета на Java.
- Можно легко соединиться в JSP с любой базой данных.
- Производительность и масштабируемость JSP обеспечиваются тем, что JSP позволяет встраивать динамические элементы в HTML-страницы.
- JSP построен на технологии Java, поэтому он не зависит от платформы и не зависит от каких-либо операционных систем.
- JSP включает в себя функцию многопоточности Java.
- Можно использовать обработку исключений Java в JSP.
- Позволяет отделить уровень представления от уровня бизнес-логики, реализуемый в JavaBeans.
- Разработчикам легко отображать и обрабатывать информацию [1].

## 2.1. КАК РАБОТАЮТ JAVASERVER PAGES

Когда запрос поступает на сервер, сервер переводит JSP-страницу в сервлет и запускает его (рис. 2.1). С этого момента и до тех пор, пока сервер не завершил работу, все запросы к этому JSP приводят к выполнению указанного сервлета.

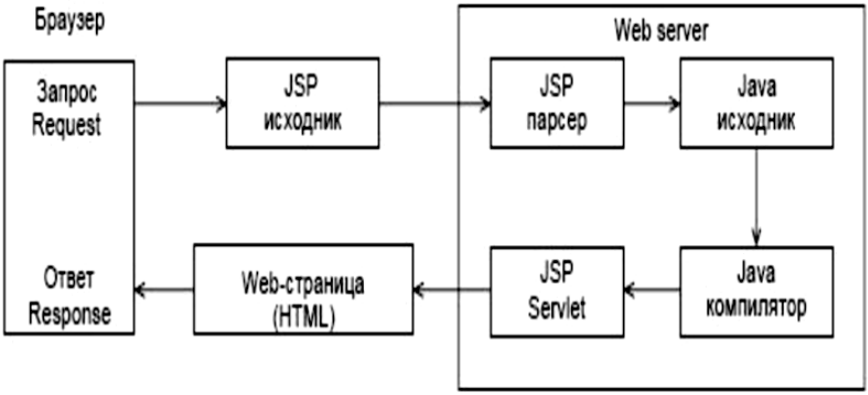



*Рис. 2.1.* Рабочий цикл JSP-страницы

Если JSP код страницы изменялся разработчиками или сервер был остановлен и запущен снова, трансляция JSP в сервлет повторяется.

## 2.2. СИНТАКСИЧЕСКИЕ КОНСТРУКЦИИ JSP

Каждый язык программирования имеет свой специальный синтаксис. JSP использует синтаксис Java, помещенный в специальные теги, чтобы можно было отделить код HTML от того кода, который относится к JSP.

Рассмотрим основные конструкции JSP.

**Выражения**

Выражения – это предложения, которые возвращают значение, включаемое в HTML-документ при его создании. Теги для выражения имеют вид: <%= и %>. Пример выражения:

Hello, <%= request.getParameter("UserName") %>

**Скриплеты в JSP-страницах**

Скриплеты – это фрагменты Java-кода, включаемые в страницы. Результат их выполнения – HTML-код. Синтаксис скриплета: <% Java код %> .

Пример использования скриплетов:

```
<% if (Math. random () < 0.5) { %>
<B>Удачного</B> Вам дня!
<% } else { %>
<B>Хорошего</B> Вам дня!
<% } %>
```

**Объявления**

Объявления применяются для создания переменных уровня сервлета вне всяких методов. Синтаксис объявления: <%! Код определения %> . Пример:

```
<% ! private int accessCount = 0; %>
```

**Комментарии**

В JSP есть два типа комментариев.

Первый тип – комментарии в стиле HTML: <!-- Это HTML-комментарий -->. Они расставляются вне кода скриплетов и передаются браузеру.

Второй тип – это JSP-комментарии. Они не передаются браузеру и имеют форму: <%-- Это JSP-комментарий --%>

**Директивы**

Директивы – это команды, которые изменяют результирующий сервлет. Существуют три типа директив: page, include и taglib.

Синтаксис директив: <%@ директива атрибут="значение" %>

**Директива page** позволяет импортировать классы, указывать базовый класс сервлета, указывать тип содержимого и т. д.

Пример директивы:

```
<%@ page import="mypackage.myclass" %>
```

**Директива include** позволяет вставлять внешний файл в страницу JSP в определенном месте. Это позволяет использовать модульный принцип построения JSP-страниц. Синтаксис директивы: <%@ include file="относительный url" %>

Пример директивы:

<% @ include file = "header.html" %>

**Директива taglib** используется для подключения тегов пользователя.

Пример директивы:

<%@taglib uri="http://java.sun.com/jsp.jstl.core" prefix="c" %>

**Неявные объекты**

В JSP имеется восемь неявных объектов, которые можно использовать в страницах. Это следующие объекты:

- **request** – это объект типа HttpServletRequest, из него можно извлечь параметры и данные браузера;
- **response** – это объект типа HttpServletResponse, через него пересылаются данные в браузер;
- **out** – это буферизованный объект типа PrintWriter, его можно использовать для вывода в сервлеты. В выражениях JSP используется этот объект без явного указания, а скриплеты явно ссылаются на него;
- **session** – это объект типа HttpSession, в нем сохраняются данные состояния для разных сервлетов, вызываемых одним браузером;
- **application** – это объект типа ServletContext, он содержит данные, доступные всем сервлетам, вне зависимости от того, какой браузер их вызывал;
- **config** – это объект типа ServletConfig для этой страницы. Контейнер сервлетов помещает туда информацию во время инициализации сервлета;
- **pageContext** – это объект типа PageContext, уникальный для JSP. В нем сохраняются параметры страницы. Он может использоваться для сохранения разделяемых данных;
- **page** – объект самой страницы, используется редко.

Эти объекты разрешены для использования в страницах JSP, чтобы иметь доступ к различной информации.

## 2.3. РАЗМЕЩЕНИЕ И ЗАПУСК JSP

Для использования JSP необходимо иметь контейнер сервлетов для их выполнения. Самый простой способ – это установить Apache Tomcat сервер. Для размещения файлов JSP создается в проекте отдельная папка /jsp в папке Веб-страницы.

```
<%@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>
    JSP Request Method: <%= request.getMethod()%> <br>
    Server: <%= request.getServerName()%> <br>
    port: <%= request.getServerPort()%> <br>
    address: <%= request.getRemoteAddr()%> <br>
    host: <%= request.getRemoteHost()%> <br>
    Locale: <%= request.getLocale()%> <br>
    Request URI: <%= request.getRequestURI()%> <br>
    Request Protocol: <%= request.getProtocol()%> <br>
    Servlet path: <%= request.getServletPath()%> <br>
    Browser version <%= request.getHeader("User-Agent")%> <br>
  </body>
</html>
```

Как видно, в странице размещены обычные теги HTML и некоторые дополнительные теги синтаксиса JSP. Также в страницу вставлены выражения, использующие методы JSP, выводящие данные о методе посылки запроса к серверу, имя машины, порт, адрес сервера, протокол и т. д. Результат выполнения сервлета показан на рис. 2.2.


*Рис. 2.2.* Результат вызова страницы jsp

## 2.4. ВЫПОЛНЕНИЕ МЕТОДОВ В JSP

Конечно, желательно использовать модульность в JSP. Один из лучших способов сделать это – использовать методы для выполнения каких-то действий. Методы могут вызываться в выражениях, в скриплетах. Возвращаемые значения можно выводить. Например:

```
<%!
   private java.util.Date sinceDate = new java.util.Date();
   private java.util.Date getSinceDate()   // определение метода на странице
   {
      return sinceDate;
   }
%>
<html>
        <head>
            <title>Вызов метода</title>
        </head>
        <body>
             <h2>Текущая дата:</h2>
             Страница первый раз запущена <%= getSinceDate() %>.
         </body>
</html>
```

В этом примере в объявлении устанавливается дата первого запуска и трансляции JSP через утилиту Date(). Затем создается метод, который просто возвращает значение этой переменной. Метод вызывается в выражении:

Страница первый раз запущена <%= getSinceDate() %>.

## 2.5. ДОБАВЛЕНИЯ БИЗНЕС-ЛОГИКИ В JSP-СТРАНИЦЫ

Имеется несколько способов для добавления бизнес-логики в страницу.

- Использовать директиву include, чтобы вставлять другую страницу JSP.
- Использовать действие include, чтобы добавить другие сервлеты, JSP и HTML-файлы в наш JSP.
- Создать JavaBean, который содержит бизнес-логику, и встроить его в JSP.

Директивы JSP были рассмотрены ранее (п. 2.2). В отличие от действий они выполняются один раз на этапе компиляции страницы в сервлет.

Действия JSP – это специальные теги, которые выполняют особые функции, такие как передача управления в другой JSP, генерация вывода, управление JavaBeans.

Действия JSP используют конструкции с синтаксисом XML для управления работой движка сервлета. Действия выполняются каждый раз при обращении к странице, позволяя делать страницу динамической.

- jsp:declaration – объявление, аналогичен тегу <%! … %>;
- jsp:scriptlet – скриплет, аналогичен тегу <% … %>;
- jsp:expression – выражение, аналогичен тегу <%= … %>;
- jsp:text – вывод текста;
- jsp:useBean – поиск или создание нового экземпляра JavaBean;
- jsp:setProperty – установка свойств JavaBean;
- jsp:getProperty – вставить свойство JavaBean в поток вывода;
- jsp:include – подключает файл в момент запроса страницы;
- jsp:forward – перенаправляет запрос на другую страницу;
- jsp:param – добавляет параметры в объект запроса, например в элементах forward, include, plugin.

Пример использования действия ***jsp:include***

```
<body>
      <h1>Новости на JspNews.com</h1>
   <p> Фрагменты четырех самых популярных статей: </p>
   <ol>
      <li><jsp: include  page="news/iteml.html" flush="true"/>
      <li><jsp: include  page="news/item2.html" flush="true"/>
      <li><jsp: include  page="news/item3.html" flush="true"/>
   </ol>
</body>
```

Пример реализации выбора подключаемого файла на странице:

```
<b> Выбор страницы <br>
      <%   String num = request.getParameter(“PageNum");
           if (num.equals("1"))    { %>
            <jsp:include page='includes/IncludedPage1.jsp' />
      <%   }    if (num.equals("2"))    { %>
            <jsp:include page='includes/IncludedPage2.jsp' />
      <%   }    if (num.equals("3"))    { %>
            <jsp:include page='includes/IncludedPage3.jsp' />
      <%   } %>
</b>
```

## 2.6. JAVABEANS

Спецификация Sun Microsystems определяет JavaBeans как повторно используемые программные компоненты, которыми можно управлять, используя графические конструкторы и средства IDE.

JavaBeans – это обычные классы Java. JavaBeans могут наследовать от любого класса и реализовывать любой интерфейс. Это значит, что Bean может быть многопоточным, иметь доступ к базе данных и т. д.

Правила создания серверных JavaBeans совсем простые:

- JavaBean должен иметь конструктор без аргументов;
- все поля класса должны быть объявленны как private. Это является хорошим стилем программирования;
- JavaBean может иметь свойства. Эти свойства – приватные переменные, доступные через методы get и set. Если переменная имеет только метод get, то она рассматривается как свойство, которое доступно только по чтению.

Создадим JavaBean для идентификации пользователя:

```
public class User {
   private String username;
   private String pass;
   private String mail;

   public User() {}
   public void setUsername(String value){
      username=value;
   }
   public void setMail(String value){
      mail=value;
   }
   public void setPass(String value){
      pass=value;
   }
   public String getUsername(){
      return username;
   }
   public String getMail() {
      return mail;
   }
```

```
    public String getPass() {
        return pass;
    }
}
```

Данные авторизации передаются через запрос в JSP-страницу, где создается JavaBean.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<jsp:useBean id="user" scope="session" class="paket.User" />
<jsp:setProperty name="user" property="*" />
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Пример JavaBean</title>
    </head>
    <body>
        <%! private String username; %>
        <% username=user.getUsername();%>
        <h1>Hello  <%=username%> !</h1>
        <a href="next.jsp"> Продолжить </a>
    </body>
</html>
```

Параметр scope задает область видимости бина (page – страница, request – запрос, session – сессия, application – все приложение).

Установка значений свойств JavaBeans описана в таблице.

**Установка свойств компонента JavaBeans**

| | |
|---|---|
| Строковая константа | <jsp:setProperty name="beanName" property="propName" value="string constant"/> |
| Параметр запроса | <jsp:setProperty name="beanName" property="propName" param="paramName"/> |
| Имя параметра запроса соответствует свойству компонента | <jsp:setProperty name="beanName" property="propName"/><br><jsp:setProperty name="beanName" property="*"/> |
| Выражение | <jsp:setProperty name="beanName" property="propName" value="<%= expression %>"/> |

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем разница между сервлетами и JSP-страницами?
2. Во что компилируется JSP-страница при первом обращении?
3. В чем преимущества JSP перед сервлетами?
4. Опишите рабочий цикл JSP-страницы.
5. Какие существуют синтаксические конструкции JSP?
6. Где может располагаться бизнес-логика серверных страниц?
7. В чем разница между выражениями и скриплетами?
8. Что такое предопределенные объекты в JSP? И какие вы знаете?
9. Что такое JavaBean? Какие требования предъявляются к его описанию?
10. Как установить значения свойств в JavaBean?
11. В чем разница между директивами и действиями?

# 3. АРХИТЕКТУРА JAVA SERVER PAGES MODEL 2

Модель 2 JSP – это шаблон проектирования, используемый при разработке web-приложений Java, который отделяет отображение содержимого от логики, используемой для получения и управления содержимым. Поскольку Модель 2 разделяет логику и отображение, то ее связывают с парадигмой модель – представление – контроллер (MVC), где модель – это JavaBeans, представление – страницы JSP, контроллеры – сервлеты Java (рис. 3.1). Запросы от клиентского браузера передаются контроллеру. Контроллер выполняет логику, необходимую для получения правильного содержимого для отображения. Для этого он обращается к классам JavaBeans, которые в свою очередь могут использовать классы, реализующие бизнес-логику, обращающие к базам данных и выполняющие обработку данных. Затем он помещает контент

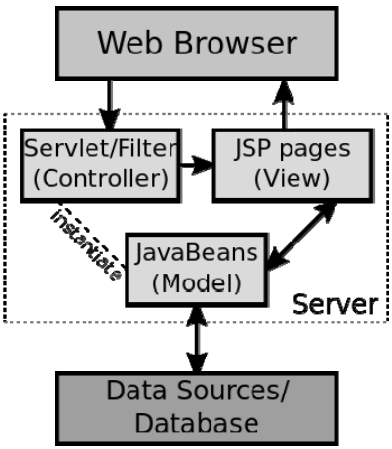



*Рис. 3.1.* Архитектура JSP модель 2

в запрос и решает, в какое представление будет передаваться запрос. Выбранное представление отображает содержимое, переданное контроллером. Модель 2 рекомендуется использовать для средних и крупных приложений.

В качестве примера реализуем авторизацию пользователя в web-приложении, использующем этот шаблон проектирования. Для модели будем использовать класс User, код которого был представлен в главе 2.

**Страница авторизации пользователя index.html**

```
<html>
  <head>
    <title>Пример Java Servlets</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body bgcolor="#fdf5e6">
    <h1 align="center">  Введите имя пользователя  </h1>
    <form action='RegController' method="post">
      Login:     <input type="text" name="login"><br><br>
      Password:     <input type="password" name="pass"><br><br>
      E-mail:     <input type="email" name="mail"><br><br>
      <center>  <input type="submit" value="Авторизовать">     </center>
    </form>
  </body>
</html>
```

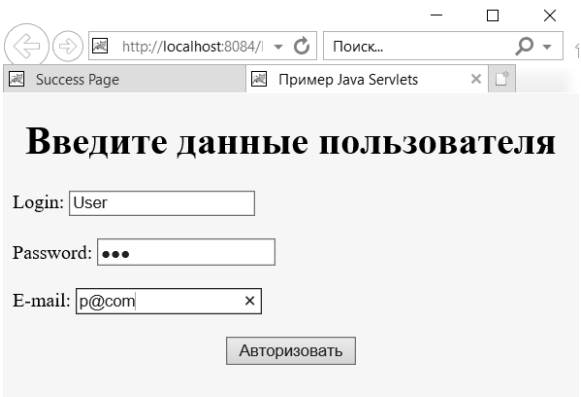



*Рис. 3.2.* Страница авторизации

**Контролер RegController**

```
import dao.UserDao;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet(name = "RegController", urlPatterns = {"/RegController"})
public class RegController extends HttpServlet {

   protected void processRequest(HttpServletRequest request, HttpServletResponse
response)          throws ServletException, IOException {
      response.setContentType("text/html;charset=UTF-8");
// получение параметров запроса
      String login = request.getParameter("login");
      String password = request.getParameter("pass");
      String email = request.getParameter("mail");

      UserDao userDao = new UserDao();
      boolean answer = userDao.ValidationUser(login, password, email);
      if (answer) {
         request.setAttribute("login", login);
         RequestDispatcher requestDispatcher =
request.getRequestDispatcher("./jsp/success.jsp");
         requestDispatcher.forward(request, response);
      } else {
         String error = "Введен неверный логин или пароль!";
         request.setAttribute("err1", error);
         RequestDispatcher requestDispatcher =
request.getRequestDispatcher("./jsp/err.jsp");
         requestDispatcher.forward(request, response);
      }
   }
   @Override
   protected void doPost(HttpServletRequest request, HttpServletResponse re-
sponse)
         throws ServletException, IOException {
      processRequest(request, response);
   }
}
```

**Страница отображения ошибки авторизации err.jsp**

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Page error</title>
   </head>
   <body>
      <h1>Ошибка авторизации!</h1>
      <%=request.getAttribute("err1") %>
   </body>
</html>
```

**Страница приветствия пользователя success.jsp**

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Success Page</title>
   </head>
   <body>
      <h1>Привет,  <%=request.getAttribute("login") %> </h1>
   </body>
</html>
```

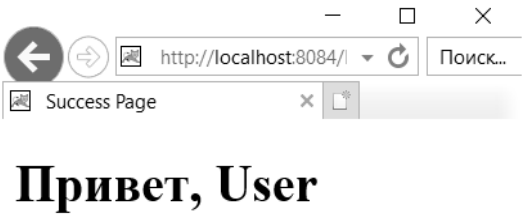



*Рис. 3.3.* Страница приветствия

Класс UserDao, содержащий метод проверки данных авторизации.

```
package dao;
public class UserDao {
  public static boolean ValidationUser(String login, String password, String email){
      return true;
   }
}
```

### КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое MVC?
2. Как реализован паттерн проектирования MVC в модель 2 JSP?
3. Какие используются средства для реализации Представления, Контроллера, Модели в модель 2 JSP?
4. Что такое RequestDispatcher? Для чего он используется в контроллере?

## 4. БИБЛИОТЕКА ПОЛЬЗОВАТЕЛЬСКИХ ТЕГОВ JSTL

Библиотека пользовательских тегов javax.servlet.jsp.jstl состоит из 5 блоков:

- core – теги, создающие подобие языка программирования, призваны заменить скриплеты java;
- xml – теги, выбирающие, интерпретирующие и преобразующие документ xml;
- fmt – теги, помогающие в интернализации страниц JSP;
- sql – теги связи и работы с базами данных;
- fn – теги для обработки строк.

Использование этих тегов в значительной мере расширяет возможности JSP. Чаще всего используемой является бибиотека core. Теги core позволяют исключить из страницы скриплеты. Подключается библиотека на странице директивой taglib:

<%@taglib uri="http://java.sun.com/jsp.jstl.core" prefix="c" %>

Рассмотрим основные теги:

<c:out> – используется для вывода данных на экран;

<c:set> – используется для определения переменной;

<c:remove> – используется для удаления переменной;

<c:if> – проверяет некоторое условие. Имеет два атрибута :

- var – имя переменной, в которой будет сохранен результат сравнения test;
- scope область видимости хранения результата сравнения;

<c:choose> – является аналогом оператора switch;

<c:forEach> – позволяет задать цикл;

<c:import> – служит для вложения одной страницы в другую;

<c:catch> – для обработки исключений в теле JSP.

Пример использования тегов core. На странице JSP выводятся данные коллекции, содержащей объекты класса User. Для этого в UserDao добавим метод:

```
public List<User> AllUsers(){
      list = new ArrayList<User>();
      list.add(new User("nick","111","q@gmail.com"));
      list.add(new User("mick","222","w@gmail.com"));
      list.add(new User("tick","333","e@gmail.com"));
      return list;
   }
```

Также в проект добавим контролер SelectController, выполняющий запрос на получение всех зарегистрированных в базе данных пользователей.

```
import dao.UserDao;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.List;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;
import model.User;

@WebServlet(name = "SelectController", urlPatterns = {"/SelectController"})
public class SelectController extends HttpServlet {
```

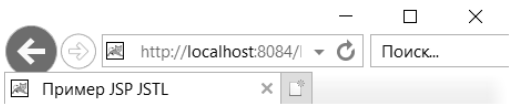
```
    List<User> list;
    protected void processRequest(HttpServletRequest request, HttpServletResponse
response)          throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        UserDao userDao = new UserDao();
        list = userDao.AllUsers();

        if (list!=null) {
            request.setAttribute("users", list);
            RequestDispatcher requestDispatcher =
request.getRequestDispatcher("./jsp/SpisokUser.jsp");
            requestDispatcher.forward(request, response);
        } else {
            String error = "Список пуст!";
            request.setAttribute("err2", error);

            RequestDispatcher requestDispatcher =
request.getRequestDispatcher("./jsp/err1.jsp");
            requestDispatcher.forward(request, response);
        }
    }
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }
}
```

В index.html добавим еще одну форму для запроса списка пользователей:

```
    <body>
……
        <h3> Получить всех пользователей</h3>
        <form action="SelectController">
            <input type="submit" value="Список">
        </form>
    </body>
```

**Страница вывода таблицы пользователей SpisokUser.jsp**

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
      <title>Пример JSP JSTL</title>
   </head>
   <body>
      <h1>Список пользователей</h1>
      <table border="1" cellspacing="0" cellpadding="2">
        <tr>
            <td>login</td>
            <td>password</td>
            <td>e-mail</td>
        </tr>
        <c:forEach items="${users}" var="user">
        <tr>
            <td>${user.username}</td>
            <td>${user.pass}</td>
            <td>${user.mail}</td>
        </tr>
        </c:forEach>
    </table>
   </body>
</html>
```




# Список пользователей

| login | password | e-mail |
|---|---|---|
| nick | 111 | q@gmail.com |
| mick | 222 | w@gmail.com |
| tick | 333 | e@gmail.com |

*Рис. 4.1.* Страница вывода списка пользователей

### КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое JSTL? Для чего применяется JSTL на страницах JSP?
2. Какие группы тегов входят в состав JSTL? Перечислите.
3. Что заменяет библиотека core на странице JSP?
4. Как подключить библиотеку JSTL?

## 5. JDBC – СТАНДАРТ ВЗАИМОДЕЙСТВИЯ JAVA С БД

**JDBC** (Java DataBase Connectivity – соединение с базами данных на Java) – платформенно-независимый стандарт взаимодействия Java-приложений с различными СУБД, реализованный в виде пакета java.sql, входящего в состав Java SE.

JDBC является крайне гибким и позволяет писать приложения, которые не зависят от конкретной платформы и могут взаимодействовать с различными СУБД без каких-либо изменений в программном коде.

Плюсы JDBC:

- простая и понятная обработка sql-запросов;
- крайне удобен для небольших приложений;
- простой и понятный синтаксис.

Минусы JDBC:

- сложно использовать и поддерживать в больших проектах;
- большое количество спагетти кода (повторяющегося);
- сложно реализовывать концепцию MVC.

**JDBC API** содержит два основных типа интерфейсов: первый – для разработчиков приложений и второй (более низкого уровня) – для разработчиков драйверов.

### 5.1. ТИПЫ JDBC-ДРАЙВЕРОВ

JDBC драйверы разрабатывают либо сами поставщики баз данных, либо сторонние фирмы, специализирующиеся на данных программных средствах. Различают несколько типов драйверов.

1. Драйверы, реализующие методы JDBC вызовами функций ODBC (входит в состав операционной системы Windows). Это так

называемый мост (bridge) JDBC-ODBC. Непосредственную связь с базой осуществляет драйвер ODBC.

2. Драйверы, написанные на языке Java и содержащие весь код в классах. Это самые легкие и удобные драйверы. Для них не требуется никакой инсталляции ни на клиенте, ни на сервере.

Любой из указанных типов драйверов должен быть скачан с сайта разработчика.

Принцип взаимодействия Java приложения с реляционными базами данных представлен на рис. 5.1.

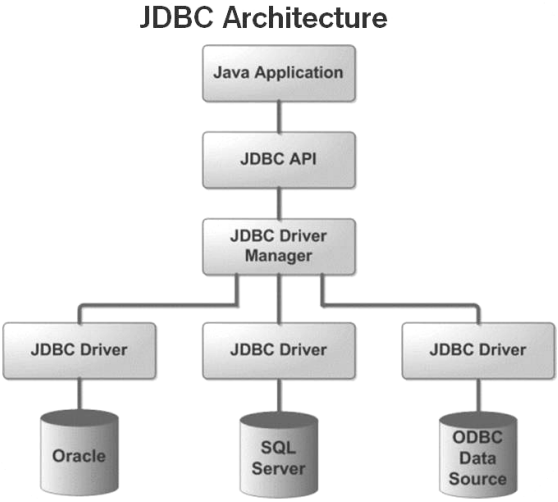



*Рис. 5.1.* Взаимодействие Java приложения с реляционной базой данных

**Преимущества использования JDBC**

– простота разработки: разработчик может не знать специфики базы данных, с которой работает;

– код практически не меняется, если будет переход на другую базу данных (изменения зависят только от различий между диалектами SQL);

– к любой базе можно подсоединиться через задание нового URL. Автоматически происходит создание соединения по протоколу TCP/IP.

## 5.2. СОЗДАНИЕ БАЗЫ ДАННЫХ И ТАБЛИЦ В POSTGRESQL

Для создания приложения, работающего с СУБД PostgreSQL, необходимо скачать с сайта разработчика https://www.enterprisedb.com/downloads/postgres-postgresql-downloads актуальную версию пакета PostgreSQL и установить ее, а также драйвер JDBC с сайта https://jdbc.postgresql.org/download.html), который можно добавить в библиотеку проекта в виде jar-архива. В процессе установки СУБД необходимо будет указать логин и пароль, которые необходимо запомнить.

В состав PostgreSQL входит бесплатный инструмент pgAdmin4 для управления СУБД PostgreSQL. pgAdmin **4** используется для создания базы данных и таблиц в PostgreSQL, написания SQL запросов, разработки процедур, функций, а также для администрирования PostgreSQL.

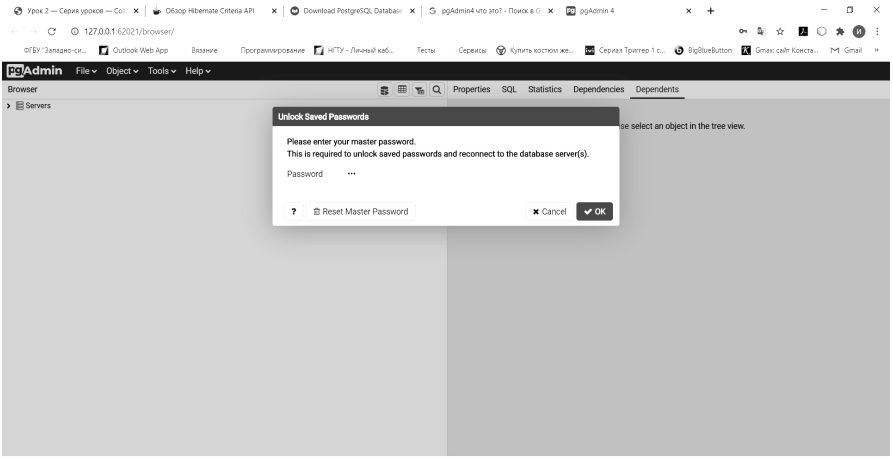



*Рис. 5.2.* Вход в pgAdmin4

Создадим для выполнения примеров базу данных example и в ней таблицу registration. Это можно сделать вручную (рис. 5.3) или через выполнение SQL-запроса.

SQL запрос для создания таблицы registration:

```
CREATE TABLE registration (
user_id serial NOT NULL,
```

```
user_login character(10),
user_pass character(8),
user_mail character(15),
PRIMARY KEY (user_id));
```

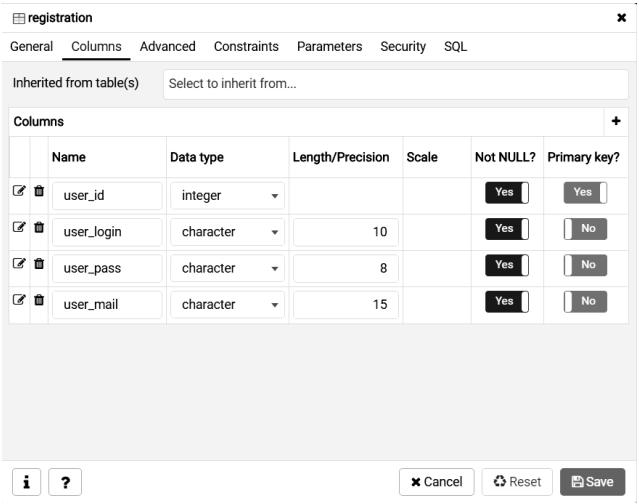



*Рис. 5.3.* Создание таблицы в pgAdmin4

## 5.3. ПОРЯДОК РАБОТЫ С БАЗОЙ ДАННЫХ

Принцип работы с базой данных для выполнения любых действий одинаков:

1) подключиться к базе данных;
2) выполнить SQL-запрос;
3) обработать данные;
4) отключиться от базы данных.

Рассмотрим все пункты подробнее.

### Подключение к базе данных

```
import java.sql.*;

public class UserDao {
// задаем параметры для подключения: uri базы данных, логин и пароль
   private static final String url = "jdbc:postgresql://localhost/example";
```

```
private static final String username = "postgres";
private static final String password = "123";
static Connection con=null;

public static Connection getConnection() {
   try{
            //загрузка класса драйвера выбранной СУБД
      Class.forName("org.postgresql.Driver");
      try {
            // подключение к базе данных
        con = DriverManager.getConnection(url, username, password);
      } catch(SQLException e){
        System.out.println(e.getMessage());
      }
   }catch(ClassNotFoundException e){
      System.out.println(e.getMessage());
   }
   return con;
}
```

Для создания подключения используется класс DriverManager, в котором есть статический метод getConnection(). При вызове нужно указать параметры: uri базы данных, логин и пароль, задаваемые при установке СУБД.

**Создание и выполнение SQL- запроса**

SQL запрос формируется в виде строки String, например:

```
String sql = "SELECT * FROM registration WHERE user_login = '" + log-
in+"'and user_pass = '"+password + "' and user_mail = '"+ email +"'";
```

Для взаимодействия с БД можно использовать один из трех интерфейсов:

– **Statement.** Интерфейс используется для доступа к БД для общих целей. Он применяется, когда используются статические SQL-выражения во время работы программы, не принимающие никаких параметров;

– **PreparedStatement.** Этот интерфейс используется в случае, когда планируется использовать SQL-выражения множество раз. Принимает параметры во время работы программы.

– **CallableStatement.** Этот интерфейс становится полезным вслучае, когда необходимо получить досутп к различным процедурам БД. Он также может прнимать параметры во время работы программы.

Пример создания объекта Statement и выполнения запроса:

```
Statement st1 = con.createStatement();
ResultSet results = st1.executeQuery(sql);
```

Пример создания объекта PreparedStatement и выполнения запроса:

```
String sql = "SELECT * FROM registration WHERE user_login = ? and user_pass = ? and user_mail = ?";
PreparedStatement pst1 = con.pst(sql);
```

Экземпляры PreparedStatement "помнят" скомпилированные SQL-выражения. Именно поэтому они называются "prepared" ("подготовленные").

SQL-выражения в PreparedStatement могут иметь один (или более) входной параметр, не указанный при создании SQL-выражения. Вместо него в выражении на месте каждого входного параметра ставится знак ("?"). Все параметры, отмеченные символом **?**, называются маркерами параметра. Параметры запроса устанавливаются методами setXXX(), например:

```
pst1.setString(1, login);
pst1. setString(2, password);
pst1. setString(3, email);
ResultSet results = pst1.executeQuery(sql);
```

Для выполнения запроса интерфейс Statement имеет три метода, которые реализуются каждой конкретной реализацией JDBC драйвера:

– **boolean execute (String SQL).** Этот метод возвращает логическое значение **true**, если объект ResultSet может быть получен. В противном случае он вовращает **false**. Он используется в случаях, когда используется динамический SQL;

– **int executeUpdate (String SQL).** Этот метод возвращает количество столбцов в таблице, на которое повлиял наш SQL-запрос. Используется для выполнения SQL-запросов update, delete, insert;

– **ResultSet executeQuery (String SQL).** Этот метод возвращает экземпляр ResultSet, представляющий множество строк, полученных

из таблицы в результате выполнения SQL-запроса. Например, при получении списка элементов, которые удовлетворяют определенным условиям.

**Обработка данных**

Обработка данных требуется в основном при выполнении запроса select. При выполнении запроса получаемый результат находится в объекте ResultSet. Извлечь полученные данные можно при выполнении цикла while (results.next()) и сохранить созданные объекты в коллекции для отображения на странице jsp в браузере.

```
while (results.next()) {
            user = new User();
            user.setUsername(results.getString("user_login"));
            user.setPass(results.getString("user_pass"));
            user.setMail(results.getString("user_mail"));
            list.add(user);
}
```

**Отключение от базы данных**

Для закрытия используется метод close(), который обычно вызывается в блоке finally для каждого созданного объекта:

```
finally   {
    try  {
        if (st1 != null)
            st1.close();
        if (con != null)
            con.close();
    } catch (SQLException e)        {
        System.out.println("SQLException during close(): " +  e.getMessage());
    }
}
```

Добавим в UserDao еще один метод, возвращающий всех зарегистрированных пользователей из таблицы registration.

```
public List<User> AllUsers(){
     list = new ArrayList<User>(); // коллекция пользователей
     Statement st1=null;
    try {
        con=getConnection();                    // получение соединения с БД
```

```
            st1 = con.createStatement();              // создание запроса
            String sql = "SELECT * FROM registration";
            ResultSet results = st1.executeQuery(sql);      // выполнение запроса
            while(results.next()) {                   // обработка запроса
               User user = new User();
               user.setUsername(results.getString("user_login"));
               user.setPass(results.getString("user_pass"));
               user.setMail(results.getString("user_mail"));
               list.add(user);                // добавление объектов User в коллекцию
            }
         } catch(SQLException e){
               System.out.println(e.getMessage());
         }finally {
            try {
               if (st1 != null) {
                  st1.close();
               }
               if (con != null) {
                  con.close();
               }
            } catch (SQLException sqle) {
               System.out.println("SQLException during close(): " + sqle.getMessage());
            }
         }
         return list;
      }
```

**Практические задания**

Реализуйте архитектуру web-приложения JSP Модель 2 по одной из предложенных тем.

- Web-сайт турфирмы.
- Web-сайт косметического салона.
- Web-сайт аптечной сети.
- Web-сайт книжного магазина.
- Web-сайт интернет магазина оргтехники.
- Web-сайт салона красоты.

1. Для ввода данных пользователем и отображения создайте JSP страницы.

2. Используйте библиотеку JSTL при разработке интерфейса страниц и расширения возможностей JSP.

3. В качестве контроллеров в модели разработайте Java-классы Servlet.

4. Для хранения данных используйте базу данных по выбору (MySQL, PostgreSQL). Для чего предварительно выполните проектирование структуры базы данных.

5. Для таблиц базы данных разработайте компоненты useBean.

6. Для доступа к данным используйте JDBC API.

7. Задачи:

– отображение таблиц данных;

– добавление данных в таблицу;

– поиск данных по полному критерию или его части;

– фильтрация данных по заданному критерию, диапазону, дате;

– удаление данных по условию.

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое JDBC API и когда его используют?
2. Что такое JDBC Driver и какие различные типы драйверов JDBC вы знаете?
3. Как JDBC API помогает достичь слабой связи между Java программой и JDBC Drivers API?
4. Что такое JDBC Connection? Укажите шаги для подключения программы к базе данных.
5. Как используется JDBC DriverManager class?
6. Что такое JDBC Statement?
7. Какие различия между execute, executeQuery, executeUpdate?
8. Что такое JDBC PreparedStatement?
9. Что такое SQL?
10. Что такое JDBC ResultSet?
11. Какие существуют различные типы JDBC ResultSet?
12. Как обработать данные, получаемые в результате выполнения запроса?

# 6. АРХИТЕКТУРА HIBERNATE FRAMEWORK

## 6.1. ORM – ОБЪЕКТНО-РЕЛЯЦИОННОЕ СВЯЗЫВАНИЕ

ORM (Object-Relational Mapping – Объектно-Реляционное Связывание) – это техника программирования, которая обеспечивает преобразование данных при их обмене между реляционной базой данных и в данном случае с Java.

Преимущества ORM в сравнение с JDBC[3]:

– позволяет бизнес-методам обращаться не к БД, а к Java-классам;
– ускоряет разработку приложения;
– основан на JDBC;
– отделяет SQL-запросы от объектно-ориентированной модели;
– позволяет не думать о реализации БД;
– сущности основаны на бизнес-задачах, а не на структуре БД;
– управление транзакциями.

ORM состоит:

– из **API**, который реализует базовые операции (создание, чтение, изменение, удаление) объектов-моделей;
– средства настройки метаданных связывания;
– техники взаимодействия с транзакциями, которая позволяет реализовать такие функции, как dirty checking (механизм «грязной» проверки), lazy association fetching (ленивая загрузка ассоциаций между объектами) и т. д. [3].

Эта связь схематично изображена на рис. 6.1.

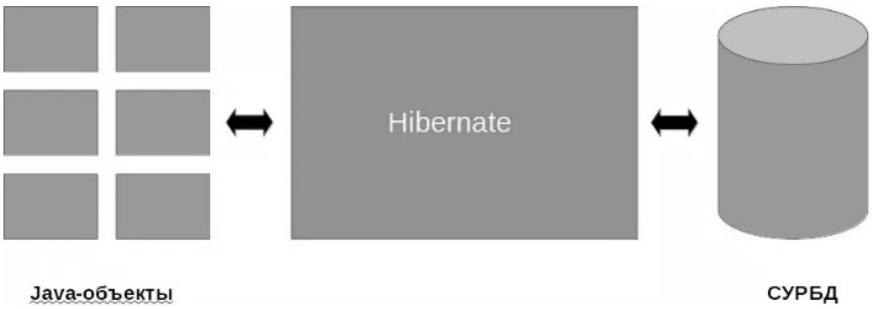



*Рис. 6.1*. Схема связи между БД и Java-объектами

Hibernate – это ORM фреймворк для Java с открытым исходным кодом. Эта технология является крайне мощной и имеет высокие показатели производительности.

Hibernate создает связь между таблицами в базе данных (далее – БД) и Java-классами и наоборот. Это избавляет разработчиков от огромного количества лишней рутинной работы, в которой крайне легко допустить ошибку и крайне трудно потом ее найти.

**Преимущества Hibernate**

1. Обеспечивает простой API для записи и получения Java-объектов в/из БД.

2. Минимизирует доступ к БД, используя стратегии fetching.

3. Не требует сервера приложения.

4. Позволяет работать не с типами данных языка SQL, а иметь дело с привычными типами данных Java.

5. Заботится о создании связей между Java-классами и таблицами БД с помощью XML-файлов, не внося изменения в программный код.

6. Если необходимо изменить БД, то достаточно лишь внести изменения в конфигурационные XML-файлы.

Hibernate поддерживает все основные СУБД: MySQL, Oracle, PostgreSQL, Microsoft SQL Server Database, HSQL, DB2.

Hibernate также может работать в связке с такими технологиями, как Maven и J2EE.

Приложение, которое использует Hibernate (в приближенном представлении), имеет архитектуру, как показано на рис. 6.2.

Hibernate поддерживает такие API, как JDBC, JNDI (**Java Naming** and **Directory Interface** – API для доступа к службам имен и каталогов), JTA (**Java Transaction API**).

JDBC обеспечивает простейший уровень абстракции функциональности для реляционных БД. JTA и JNDI, в свою очередь, позволяют Hibernate использовать серверы приложений J2EE.

Рассмотрим элементы Hibernate, которые показаны на рис. 6.2.

***Transaction***

Этот объект представляет собой рабочую единицу сеанса с БД. В Hibernate транзакции обрабатываются менеджером транзакций.

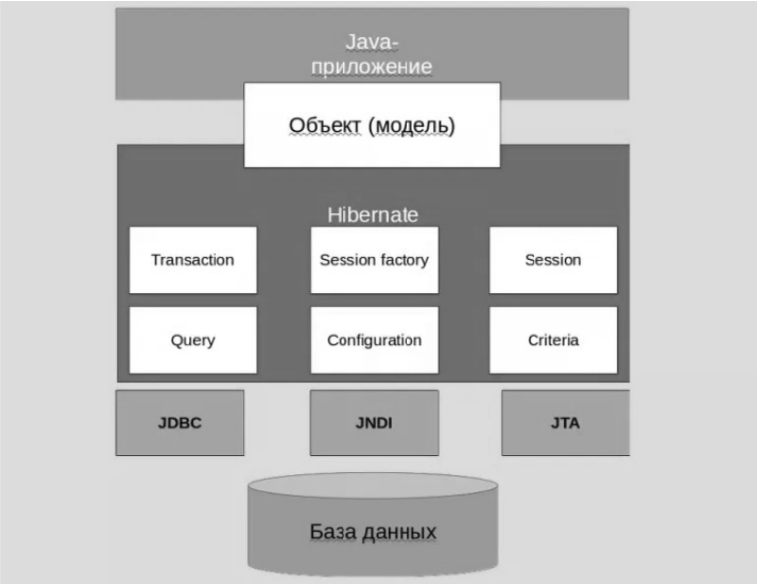



*Рис. 6.2*. Архитектура приложения с Hibernate

***SessionFactory***

Самый важный и самый тяжелый объект (обычно создается в единственном экземпляре при запуске приложения). Нам необходима как минимум одна SessionFactory для каждой БД, каждая из которых конфигурируется отдельным конфигурационным файлом.

Внутреннее состояние SessionFactory неизменно (immutable), включает в себя все метаданные об Object/ Relational Mapping и задается при создании SessionFactory.

SessionFactory также предоставляет методы для получения метаданных класса и статистики, например, данных о втором уровне кэша, выполняемых запросах и т. д. Так как объект SessionFactory неизменяемый, то он потокобезопасный. Множество потоков может обращаться к одному объекту одновременно.

Метод getCurrentSession объекта SessionFactory возвращает сессию, связанную с контекстом. Но для того, чтобы метод вернул не NULL, необходимо настроить параметр current_session_context_class в конфигурационном файле Hibernate. Поскольку полученный объект Session

связан с контекстом Hibernate, то отпадает необходимость в его закрытии, он закрывается вместе с закрытием SessionFactory.

```
<property name="hibernate.current_session_context_class">
thread
</property>
```

Метод openSession объекта SessionFactory всегда создает новую сессию. В этом случае необходимо обязательно контролировать закрытие объекта сессии по завершению всех операций с базой данных. Для многопоточной среды необходимо создавать новый объект Session для каждого запроса.

При загрузке больших объемов данных без удержания большого количества информации в кэше можно использовать метод openStatelessSession(), который возвращает Session без поддержки состояния. Полученный объект не реализует первый уровень кэширования и не взаимодействует со вторым уровнем. Сюда же можно отнести игнорирование коллекций и некоторых обработчиков событий.

***Session***

Сессия используется для получения физического соединения с БД. Обычно сессия создается при необходимости выполнения запроса к БД, а после этого закрывается. Это связано с тем, что эти объекты крайне легковесны. Чтобы понять, что это такое, можно сказать, что создание, чтение, изменение и удаление объектов происходит через объект Session. Другими словами, объект предоставляет методы для CRUD (create, read, update, delete) операций для объекта персистентности. С помощью этого экземпляра можно выполнять HQL-, SQL-запросы и задавать критерии выборки.

Объект Hibernate Session не является потокобезопасным. Каждый поток должен иметь свой собственный объект Session и закрывать его по окончанию.

***Query***

Этот объект использует HQL или SQL для чтения/записи данных из/в БД. Экземпляр запроса используется для связывания параметров запроса, ограничения количества результатов, которые будут возвращены, и для выполнения запроса.

***Configuration***

Этот объект используется для создания объекта SessionFactory и конфигурирует сам Hibernate с помощью конфигурационного XML-файла, который объясняет, как обрабатывать объект Session.

***Criteria***

Используется для создания и выполнения объекто-ориентированных запросов для получения объектов.

## 6.2. КОНФИГУРИРОВАНИЕ HIBERNATE

Для корректной работы необходимо передать Hibernate подробную информацию, которая свяжет Java-классы с таблицами в базе данных (далее – БД). Также необходимо указать значения определенных свойств Hibernate.

Обычно вся эта информация помещена в отдельный файл либо XML-файл – hibernate.cfg.xml, либо – hibernate.properties.

Для начала рассмотрим ключевые свойства, которые должны быть настроены в типичном приложении.

1) *hibernate.dialect*

Указывает Hibernate диалект БД. Hibernate, в свою очередь, генерирует необходимые SQL-запросы (например, org.hibernate.dialect. MySQLDialect, если в приложении используется СУБД MySQL).

2) *hibernate.connection-driver_class*

Указывает класс JDBC драйвера.

3) *hibernate.connection.url*

Указывает URL (ссылку) необходимой БД (например, jdbc:mysql://localhost:3306/database).

4) *hibernate.connection.username*

Указывает имя пользователя БД (например, root).

5) *hibernate.connection.password*

Указывает пароль к БД (например, password).

6) *hibernate.connection.pool_size*

Ограничивает количество соединений, которые находятся в пуле соединений Hibernate.

7) *hibernate.connection.autocommit*

Указывает режим autocommit для JDBC-соединения.

Пример конфигурационного XML-файла представлен ниже.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
    </property>
    <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
    </property>
  <!-- Assume test is the database name -->
  <property name="hibernate.connection.url">
    jdbc:mysql://localhost/database
  </property>
  <property name="hibernate.connection.username">
    root
  </property>
  <property name="hibernate.connection.password">
    password
  </property>

</session-factory>
</hibernate-configuration>
```

Таким образом, файл конфигурации Hibernate содержит данные о базе данных и необходим для инициализации SessionFactory. В .xml файле необходимо указать вендора базы данных или JNDI ресурсы, а также информацию об используемом диалекте, что поможет hibernate выбрать режим работы с конкретной базой данных.

## 6.3. СЕССИИ

Сессия используется для получения физического соединения с базой данных. Мы создаем, читаем, редактируем и удаляем объекты с помощью сессий. Сессии создаются при необходимости, а затем уни-

чтожаются из-за того, что не являются потокозащищенными и не должны быть открыты в течение длительного времени.

Экземпляр класса может находиться в одном из трех состояний:

– **transient**. Это устойчивый экземпляр класса, который не привязан к сессии и еще не представлен в БД. Он не имеет значения, по которому может быть идентифицирован. Объект в этом состоянии может стать персистентным при вызове метода save(), persist() или saveOrUpdate(). Объект персистентности может перейти в transient состоянии после вызова метода delete();

– **persistent**. Устойчивый экземпляр класса представлен в БД, а значение идентификатора связано с сессией. Когда объект связан с уникальной сессией, он находится в состоянии persistent (персистентности). Любой экземпляр, возвращаемый методами get() или load(), находится в состоянии persistent;

– **detached**. После того как сессия закрыта, экземпляр класса становится отдельным независимым экземпляром класса. Такой объект можно сделать персистентным, используя методы update(), saveOrUpdate(), lock() или replicate(). Состояния transient или detached так же могут перейти в состояние persistent, как новый объект персистентности после вызова метода merge().

В примитивном виде транзакция представлена ниже.

```
Session session = sessionFactory.openSession();
Transaction transaction = null;

   try{
      transaction = session.beginTransaction();
      /**
       * Here we make some work.
       * */
      transaction.commit();
   }catch(Exception e){
      if(transaction !=null){
         transaction.rollback();
         e.printStackTrace();
      }
      e.printStackTrace();
   }finally{
      session.close();
   }
```

Основные методы интерфейса Session представлены в табл. 6.1.

Таблица 6.1

**Методы интерфейса Session**

| Методы | Описание |
|---|---|
| Transaction beginTransaction() | Начинает транзакцию и возвращает объект Transaction |
| void cancelQuery() | Отменяет выполнение текущего запроса |
| void clear() | Полностью очищает сессию |
| Connection close() | Заканчивает сессию, освобождает JDBC-соединение и выполняет очистку |
| Criteria createCriteria(String entityName) | Создание нового экземпляра Criteria для объекта с указанным именем |
| Criteria createCriteria(Class persistentClass) | Создание нового экземпляра Criteria для указанного класса |
| Serializable getIdentifier(Object object) | Возвращает идентификатор данной сущности как сущности, связанной с данной сессией |
| void update(String entityName, Object object) | Обновляет экземпляр с идентификатором, указанном в аргументе |
| void update(Object object) | Обновляет экземпляр с идентификатором, указанном в аргументе |
| void saveOrUpdate(Object object) | Сохраняет или обновляет указанный экземпляр |
| Serializable save(Object object) | Сохраняет экземпляр, предварительно назначив сгенерированный идентификатор |
| boolean isOpen() | Проверяет, открыта ли сессия |
| boolean isDirty() | Проверяет, есть ли в данной сессии какие-либо изменения, которые должны быть синхронизованы с базой данных |
| boolean isConnected() | Проверяет, подключена ли сессия в данный момент |
| Transaction getTransaction() | Получает связанную с этой сессией транзакцию |
| void refresh(Object object) | Обновляет состояние экземпляра из БД |
| SessionFactory getSessionFactory() | Возвращает фабрику сессий (Session Factory), которая создала данную сессию |

Окончание табл. 6.1

| Методы | Описание |
|---|---|
| Session get(String entityName, Serializable id) | Возвращает сохраненный экземпляр с указанными именем сущности и идентификатором. Если таких сохраненных экземпляров нет – возвращает null |
| void delete(String entityName, Object object) | Удаляет сохраненный экземпляр из БД |
| void delete(Object object) | Удаляет сохраненный экземпляр из БД |
| SQLQuery createSQLQuery (String queryString) | Создает новый экземпляр SQL-запроса (SQLQuery) для данной SQL-строки |
| Query createQuery (String queryString) | Создает новый экземпляр запроса (Query) для данной HQL-строки |
| Query createFilter(Object collection, String queryString) | Создает новый экземпляр запроса (Query) для данной коллекции и фильтра-строки |

***Разница между методами Hibernate Session get() и load()***

1) get() загружает данные сразу при вызове, в то время как load() использует прокси объект и загружает данные только тогда, когда это требуется на самом деле. В плане ленивой загрузки данных load() имеет преимущество;

2) load() бросает исключение, когда данные не найдены. Поэтому его нужно использовать только при уверенности в существовании данных.

Нужно использовать метод get(), если необходимо удостовериться в наличии данных в БД.

У класса Sission есть метод merge(). Он может быть использован для обновления существующих значений, однако этот метод создает копию из переданного объекта сущности и возвращает его. Возвращаемый объект является частью контекста персистентности и отслеживает любые изменения, а переданный объект не отслеживается.

***Разница между Hibernate save(), saveOrUpdate() и persist().***

Hibernate save() используется для сохранения сущности в базу данных. Проблема с использованием метода save() заключается в том, что он может быть вызван без транзакции. А, следовательно, если у нас

имеется отображение нескольких объектов, то только первичный объект будет сохранен и мы получим несогласованные данные. Также save() немедленно возвращает сгенерированный идентификатор.

Hibernate persist() аналогичен save() с транзакцией. persist() не возвращает сгенерированный идентификатор сразу.

Hibernate saveOrUpdate() использует запрос для вставки или обновления, основываясь на предоставленных данных. Если данные уже присутствуют в базе данных, то будет выполнен запрос обновления. Метод saveOrUpdate() можно применять без транзакции, но это может привести к аналогичным проблемам, как и в случае с методом save().

## 6.4. СОХРАНЯЕМЫЕ КЛАССЫ

Ключевая функция Hibernate заключается в том, что можно взять значения из экземпляра Java-класса (persistent class) и сохранить их в таблице базы данных. С помощью конфигурационных файлов указывается Hibernate, как извлечь данные из класса и соединить с определенными столбцами в таблице БД.

Для того чтобы сделать работу с Hibernate максимально удобной и эффективной используется программная модель Простых Старых Java Объектов (Plain Old Java Object – POJO). Существуют определенные требования к POJO-классам. Вот самые главные из них:

– все классы должны иметь ID для простой идентификации наших объектов в БД и в Hibernate. Это поле класса соединяется с первичным ключом (primary key) таблицы БД;

– должны иметь конструктор по умолчанию (пустой);

– все поля должны иметь модификатор доступа private и иметь набор getter-ов и setter-ов в стиле JavaBean;

– POJO-классы не должны содержать бизнес-логику.

Другими словами, чтобы класс мог быть сущностью, к нему предъявляются следующие требования:

1) должен иметь пустой конструктор (public или protected);

2) не может быть вложенным интерфейсом или enum;

3) не может быть final и не может содержать final-полей/свойств.

Hibernate использует прокси классы для ленивой загрузки данных (т. е. по необходимости, а не сразу). Это достигается с помощью

расширения entity bean и, следовательно, если бы он был final, то это было бы невозможно. Ленивая загрузка данных во многих случаях повышает производительность, и, следовательно, важна;

4) должен содержать хотя бы одно @Id-поле.

## 6.5. ВИДЫ СВЯЗЕЙ В ORM

Связи в ORM делятся на три группы:

– связывание коллекций;

– ассоциативное связывание;

– связывание компонентов.

***Связывание коллекций***

Если среди значений класса есть коллекции (collections) каких-либо значений, мы можем связать их с помощью любого интерфейса коллекций, доступных в Java.

В Hibernate мы можем оперировать следующими коллекциями:

– java.util.List. Связывается с помощью элемента <list> и инициализируется с помощью java.util.ArrayList;

– java.util.Collection. Связывается с помощью элементов <bag> или <ibag> и инициализируется с помощью java.util.ArrayList;

– java.util.Set. Связывается с помощью элемента <set> и инициализируется с помощью java.util.HashSet;

– java.util.SortedSet. Связывается с помощью элемента <set> и инициализируется с помощью java.util.TreeSet. В качестве параметра для сравнения может выбрать либо компаратор, либо естественный порядок;

– java.util.Map. Связывается с помощью элемента <map> и инициализируется с помощью java.util.HashMap;

– java.util.SortedMap. Связывается с помощью элемента <map> и инициализируется с помощью java.util.TreeMap. В качестве параметра для сравнения может выбрать либо компаратор, либо естественный порядок.

***Ассоциативное связывание***

Связывание ассоциаций – это связывание (mapping) классов и отношений между таблицами в БД.

Существует четыре типа таких зависимостей.

– **Many-to-One**. Связывание отношений many-to-one с использованием Hibernate.

– **One-to-One**. Связывание отношений one-to-one с использованием Hibernate.

– **One-to-Many**. Связывание отношений one-to-many с использованием Hibernate.

– **Many-to-Many**. Связывание отношений many-to-many с использованием Hibernate.

Файл отображения (mapping file) используется для связи entity бинов и колонок в таблице базы данных. В случаях, когда не используются аннотации JPA, файл отображения .xml может быть полезен (например, при использовании сторонних библиотек).

***Связывание компонентов***

Возможна ситуация, при которой наш Java класс имеет ссылку на другой класс, как одну из переменных. Если класс, на который ссылаются, не имеет своего собственного жизненного цикла и полностью зависит от жизненного цикла класса, который на него ссылается, то этот класс называется классом Компонентом (Component Class).

Необходимо отметить, что существует несколько способов реализовать связи в Hibernate:

1) использовать ассоциативное связывание (one-to-one, one-to-many, many-to-many);

2) использовать в HQL-запросе команду JOIN. Существует другая форма «join fetch», позволяющая загружать данные немедленно (не lazy).

3) использовать чистый SQL-запрос с командой join.

## 6.6. АННОТАЦИИ

Аннотации являются мощным инструментом для предоставления метаданных, а также намного нагляднее при чтении кода другим разработчиком.

Таблица 6.2

**Обязательные аннотации**

| Аннотация | Описание |
|---|---|
| @Entity | Указывает, что данный класс является сущностью (entity bean). Такой класс должен иметь конструктор по умолчанию. Hibernate использует рефлексию для создания экземпляров Entity бинов при вызове методов get() или load() |
| @Table | Указывает, с какой таблицей необходимо связать данный класс. Аннотация имеет различные атрибуты, с помощью которых можно указать имя таблицы, каталог, БД и уникальность столбцов в таблиц БД |
| @Id | Указывает первичный ключ (Primary Key) данного класса |
| @GeneratedValue | Используется вместе с аннотацией @Id и определяет такие параметры, как strategy и generator |
| @Column | Определяет, к какому столбцу в таблице БД относится конкретное поле класса (атрибут класса). Наиболее часто используемые атрибуты:<br>– name. Задает имя столбца в таблице;<br>– unique. Значения столбца уникальны;<br>– nullable. Задает, что данное поле быть NULL или нет;<br>– length. Указывает размер столбца (например, количество символов при использовании String) |

Также можно добавить:

– javax.persistence.Access – определяет тип доступа, поле или свойство. Поле является значением по умолчанию и, если нужно чтобы hibernate использовал методы getter/setter, то их необходимо задать для нужного свойства;

– javax.persistence.EmbeddedId – используется для определения составного ключа в бине;

– javax.persistence.OneToOne – задает связь один-к-одному между двумя сущностными бинами. Соответственно есть другие аннотации OneToMany, ManyToOne и ManyToMany;

– org.hibernate.annotations.Cascade – определяет каскадную связь между двумя entity бинами. Используется в связке с org.hibernate. annotations.CascadeType;

– javax.persistence.PrimaryKeyJoinColumn – определяет внешний ключ для свойства. Используется вместе с org.hibernate. annotations.GenericGenerator и org.hibernate.annotations.Parameter.

# 7. ЯЗЫКИ ЗАПРОСОВ В HIBERNATE

## 7.1. ЯЗЫК ЗАПРОСОВ HIBERNATE (HQL)

HQL (Hibernate Query Language) – это объекто-ориентированный язык запросов, который очень похож на SQL. Отличие между HQL и SQL состоит в том, что SQL работает с таблицами в базе данных и их столбцами, а HQL – с сохраняемыми объектами (Persistent Objects) и их полями (атрибутами класса). Hibernate транслирует HQL-запросы в понятные для БД SQL-запросы, которые и выполняют необходимые нам действия в БД.

Также имеется возможность использовать обычные SQL-запросы в Hibernate, используя Native SQL, но использование HQL является более предпочтительным.

**Основные ключевые слова языка HQL**

| Ключевые слова | Описание | Пример |
|---|---|---|
| FROM | Для загрузки в память объектов из таблицы БД | Query query = session.createQuery ("FROM Developer");<br>List developers = query.list(); |
| INSERT | Для добавления записи в таблицу БД | Query query = session.createQuery ("INSERT INTO Developer (firstName, lastName, specialty, experience)"); |
| UPDATE | Для обновления одного или нескольких полей объекта | Query query = session.createQuery ("UPDATE Developer SET experience =:experience WHERE id =:developerId");<br>query.setParameter("expericence", " ");<br>query.setParameter("developerId", 3); |
| DELETE | Для удаления одного или нескольких объектов | Query query = session.createQuery ("DELETE FROM Developer WHERE id = :developerId");<br>query.setParameter("developerId", 1); |

Окончание таблицы

| Ключевые слова | Описание | Пример |
|---|---|---|
| SELECT | Получить запись из таблицы БД | Query query = session.createQuery ("SELECT D.lastName FROM Developer D"); List developers = query.list(); |
| AS | Использование опционального ключевого слова | Query query = session.createQuery ("FROM Developer AS D"); List developers = query.list(); |
| WHERE | Для выборки объектов, соответствующих определенным параметрам | Query query = session.createQuery ("FROM Developer D WHERE D.id = 1"); List developer = query.list(); |
| ORDER BY | Для сортировки списка объектов, полученных по запросу. Тип сортировки – по возрастанию (ASC) или по убыванию (DESC) | Query query = session.createQuery ("FROM Developer D WHERE experience > 3 ORDER BY D.experience DESC"); |
| GROUP BY | Для группировки данных, полученных из БД | Query query = session.createQuery ("SELECT MAX(D.experience), D.lastName, D.specialty FROM Developer D GROUP BY D.lastName"); List developers = query.list(); |

***Методы агрегации***

Язык запросов Hibernate (HQL) поддерживает различные методы агрегации, которые доступны и в SQL. HQL поддерживает следующие методы:

– min(имя свойства). Минимальное значение данного свойства;

– max(имя свойства). Максимальное значение данного свойства;

– sum(имя свойства). Сумма всех значений данного свойства;

– avg(имя свойства). Среднее арифметическое всех значений данного свойства;

– count(имя свойства). Какое количество раз данное свойство встречается в результате.

## 7.2. ЗАПРОСЫ С ИСПОЛЬЗОВАНИЕМ CRITERIA

Hibernate поддерживает различные способы манипулирования объектами и транслирования их в таблицы баз данных. Одним из таких способов является Criteria API, который позволяет нам создавать запросы с критериями, программным методом.

Для создания Criteria используется метод createCriteria() интерфейса Session. Этот метод возвращает экземпляр сохраняемого класса (persistent class) в результате его выполнения.

```
Criteria criteria = session.createCriteria(Developer.class);
List developers = criteria.list();
```

Criteria имеет два важных метода:

– public Criteria setFirstResult(int firstResult). Этот метод указывает первый ряд нашего результата, который начинается с 0;

– public Criteria setMaxResults(int maxResults). Этот метод ограничивает максимальное количество объектов, которое Hibernate сможет получить в результате запроса.

***Разница между sorted collection и ordered collection***

При использовании алгоритмов сортировки из Collection API для сортировки коллекции используется сортированный список (sorted list). Для маленьких коллекций это не приводит к излишнему расходу ресурсов, но на больших коллекциях это может привести к потере производительности и ошибкам OutOfMemory. Также entity бины должны реализовывать интерфейс Comparable или Comparator для работы с сортированными коллекциями.

При использовании фреймворка Hibernate для загрузки данных из базы данных мы можем применить Criteria API и команду order by для получения отсортированного списка (ordered list). Ordered list является лучшим выбором к sorted list, так как он использует сортировку на уровне базы данных. Она быстрее и не может привести к утечке памяти.

```
List<Employee> empList = session.createCriteria(Employee.class)
                .addOrder(Order.desc("id")).list();
```

***Преимущества Hibernate Criteria API***

Hibernate Criteria API является более объектно-ориентированным для запросов, которые получают результат из базы данных. Для опера-

ций update, delete или других DDL-манипуляций использовать Criteria API нельзя. Критерии используются только для выборки из базы данных в более объектно-ориентированном стиле.

Вот некоторые области применения Criteria API:

– поддерживает проекцию, которую можно использовать для агрегатных функций вроде sum(), min(), max() и т. д.;

– может использовать ProjectionList для извлечения данных только из выбранных колонок;

– может быть использован для join запросов с помощью соединения нескольких таблиц, используя методы createAlias(), setFetchMode() и setProjection();

– поддерживает выборку результатов согласно условиям (ограничениям). Для этого используется метод add(), с помощью которого добавляются ограничения (Restrictions);

– позволяет добавлять порядок (сортировку) к результату с помощью метода addOrder().

## 7.3. НАТИВНЫЙ SQL

Также можно создавать запросы на родном для выбранной базы данных диалекте SQL. Это полезно, если вы хотите использовать специфичные возможности СУБД. Это также предусматривает простой путь переноса приложений, написанных с использованием SQL/JDBC на Hibernate.

SQL-запросы представляются через тот же Query интерфейс, который используется для HQL-запросов. Единственное различие заключается в том, что для SQL используется метод Session.createSQLQuery().

Однако в общем случае не рекомендуется их использование, так как теряются все преимущества HQL (ассоциации, кэширование).

Использование нативного SQL может быть необходимо при выполнении запросов к некоторым базам данных, которые могут не поддерживаться в Hibernate. Примером могут служить некоторые специфичные запросы и «фишки» при работе с БД от Oracle.

```
Transaction tx = session.beginTransaction();
SQLQuery query = session.createSQLQuery("select emp_id, emp_name, emp_salary from Employee");
List<Object[]> rows = query.list();
for(Object[] row : rows){
```

```
        Employee emp = new Employee();
        emp.setId(Long.parseLong(row[0].toString()));
        emp.setName(row[1].toString());
        emp.setSalary(Double.parseDouble(row[2].toString()));
        System.out.println(emp);
}
```

Hibernate поддерживает именованный запрос, который можно задать в каком-либо центральном месте и потом использовать его в любом месте кода. Именованные запросы поддерживают как HQL, так и Native SQL. Создать именованный запрос можно с помощью JPA аннотаций @NamedQuery, @NamedNativeQuery или в конфигурационном файле отображения (mapping files).

Синтаксис NamedQuery проверяется при создании SessionFactory, что позволяет заметить ошибку на раннем этапе, а не при запущенном приложении и выполнении запроса.

Однако одним из основных недостатков именованного запроса является то, что его очень трудно отлаживать (могут быть сложности с поиском места определения запроса).

```
    @NamedNativeQuery(name="night&area", query="select night.id nid, " +
"night.night_duration, night.night_date, area.id aid, night.area_id, area.name "
        + "from Night night, Area area where night.area_id = area.id",
                resultSetMapping="joinMapping")
    @SqlResultSetMapping(name="joinMapping", entities={
        @EntityResult(entityClass=Night.class, fields = {
            @FieldResult(name="id", column="nid"),
            @FieldResult(name="duration", column="night_duration"),
            @FieldResult(name="date", column="night_date"),
            @FieldResult(name="area", column="area_id"),
            discriminatorColumn="disc"
        }),
@EntityResult(entityClass=org.hibernate.test.annotations.query.Area.class, fields =
{
            @FieldResult(name="id", column="aid"),
            @FieldResult(name="name", column="name")
        })
        }
)
```

# 8. СОЗДАНИЕ ПРИЛОЖЕНИЯ С HIBERNATE

## 8.1. ПОДГОТОВКА К РАЗРАБОТКЕ

Для выполнения задания понадобятся следующие программные средства.

1. Среда разработки. В данном примере используется NetBeans IDE 8.2.

2. СУБД PostgreSQL и драйвер JDBC для данной СУБД.

*Задание*

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

1) создавать владельцев авто (User) и сохранять их в базе данных, а также искать их по ID, обновлять их данные в базе, а также удалять из базы;

2) присваивать владельцам авто объекты автомобилей (Auto). Создавать, редактировать, находить и удалять автомобили из базы данных;

3) приложение должно автоматически удалять ненужные автомобили из БД, т. е. при удалении владельца авто все принадлежащие ему автомобили также должны быть удалены из БД.

В рассматриваемом примере реализуется лишь часть функционала: создание списка владельцев авто и их автомобилей, получение из БД и вывод на экран списка, удаление владельцев авто и их автомобилей.

## 8.2. РАЗРАБОТКА ПРИЛОЖЕНИЯ ПО ШАГАМ

**Шаг № 1 – создаем и настраиваем проект**

На рис. 8.1 показано окно для создания проекта. Создаем проект через главное меню Файл – Создать проект – Web-приложение нажимаем кнопку Далее.

Задаем имя проекта Example1 и выбираем место расположения. Нажимаем на кнопку Далее.

В окне на рис. 8.2 выбираем установленный сервер и версию JavaEE.

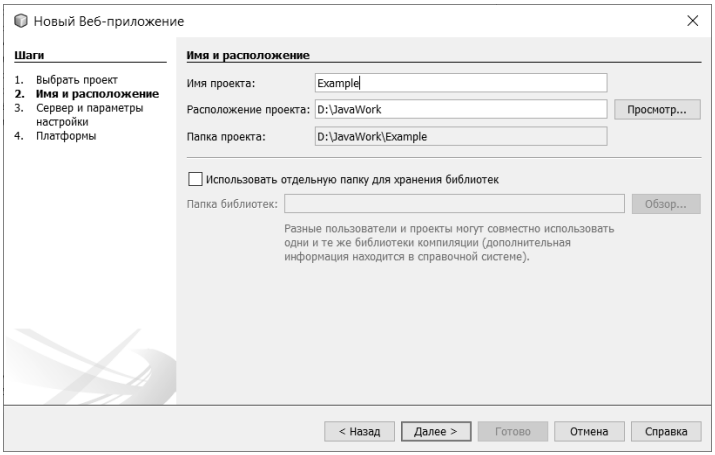



*Рис. 8.1*. Окно для создания проекта

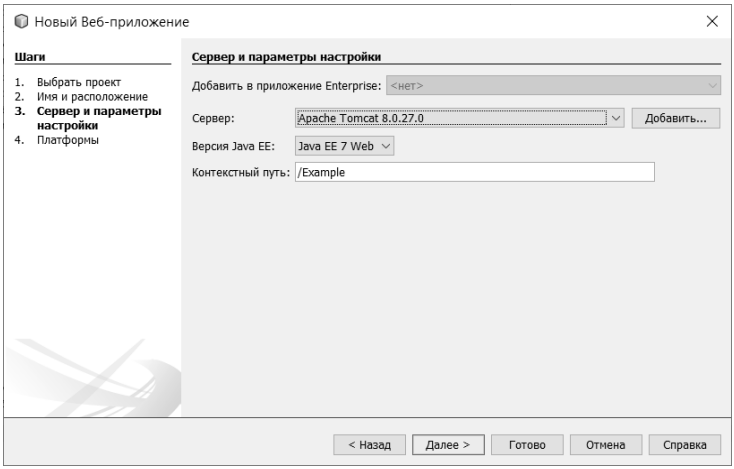



*Рис. 8.2.* Окно выбора сервера

В следующем окне на рис. 8.3 необходимо указать используемые фреймворки и настроить соединение с базой данных (если подключение уже было выполнено ранее или через вкладку Службы (описание ниже)). В данном примере будем использовать СУБД PostgreSQL. Далее нажимаем Готово. Создание и настройка проекта завершены.

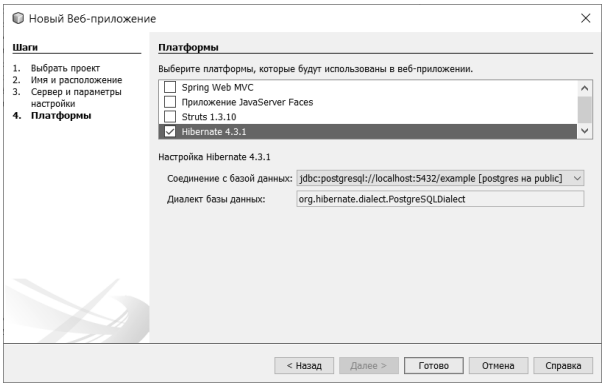



*Рис. 8.3.* Окно настройки проекта

**Шаг № 2 – создаем структуру проекта**

В папке Пакеты исходных кодов создать пакеты dao, models, services, utils, controllers. В папке Веб-страницы – папку jsp и стартовую страницу index.html.

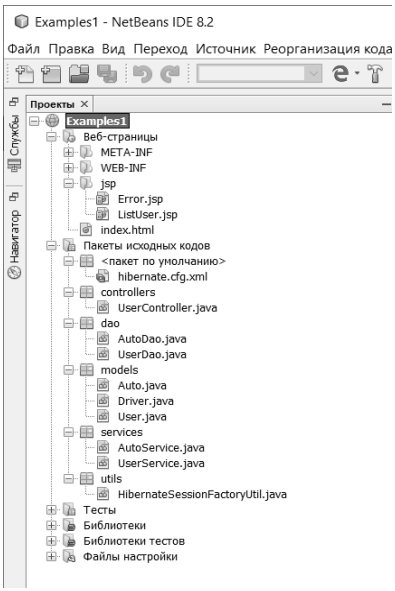



*Рис. 8.4.* Структура проекта

**Шаг № 3 – создаем таблицы в PostgreSQL**

Через pgAdmin4 создать базу данных example и в ней две таблицы: Users и Autos (рис. 8.5).

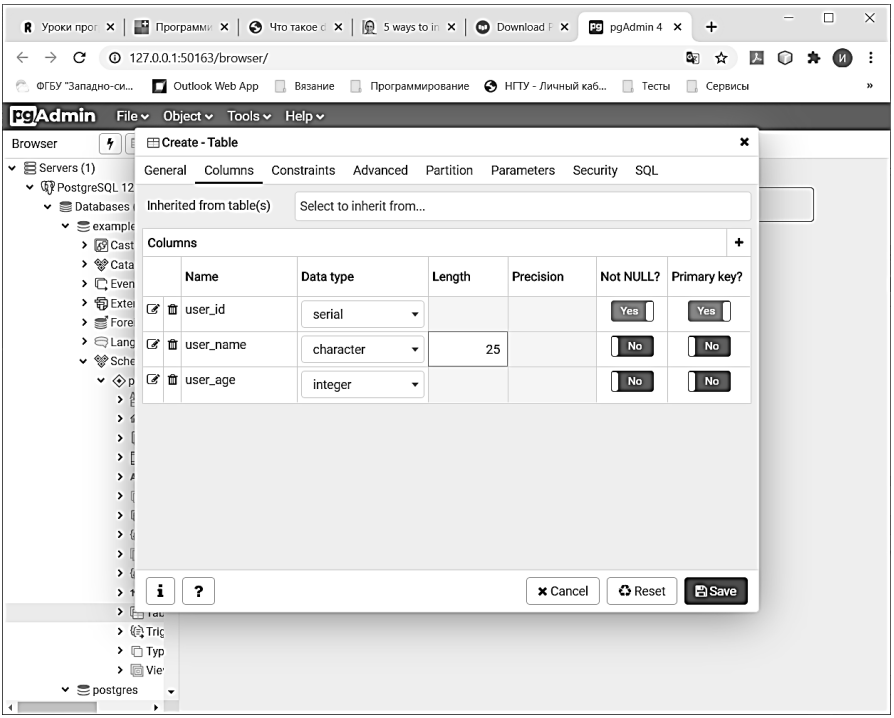



*Рис. 8.5*. Создание базы данных и таблиц в pgAdmin4

Ниже представлены SQL-запросы для создания таблиц users и autos.

```
CREATE TABLE users (
user_id serial NOT NULL,
user_name character(25),
user_age int,
PRIMARY KEY (user_id));

CREATE TABLE autos (
auto_id serial NOT NULL,
```

```
auto_model char(15),
auto_color char(15),
user_id int,
PRIMARY KEY (auto_id),
FOREIGN KEY (user_id) REFERENCES users (user_id));
```

**Шаг № 4 – добавление необходимых библиотек**

Щелчком правой кнопки мыши по папке проекта Библиотеки добавляем в проект драйвер для PostgreSQL и библиотеку тегов JSTL (рис. 8.6).

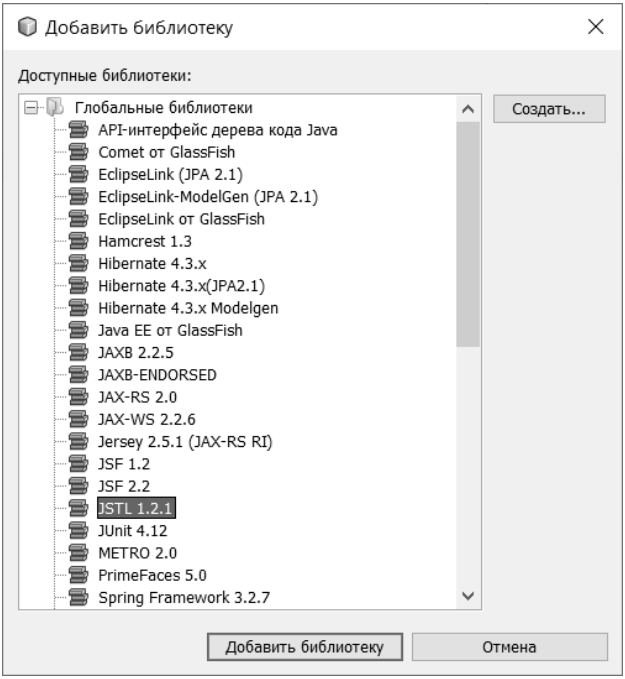



*Рис. 8.6.* Добавление библиотеки

**Шаг № 5 – подключаем PostgreSQL к проекту**

Настроить новое подключение к базе данных можно в окне Службы (рис. 8.7), кликнув правой кнопкой мыши на узел Базы данных. В контекстном меню выбрать пункт Новое подключение…

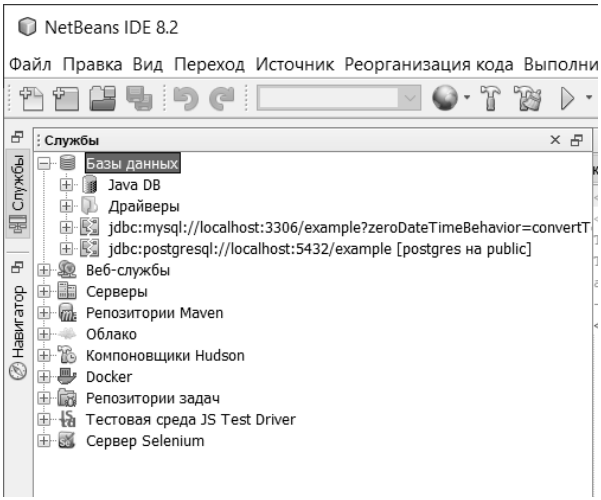



*Рис. 8.7.* Пошаговые действия для добавления БД

После выполненных действий появится окно, которое представлено на рис. 8.8.

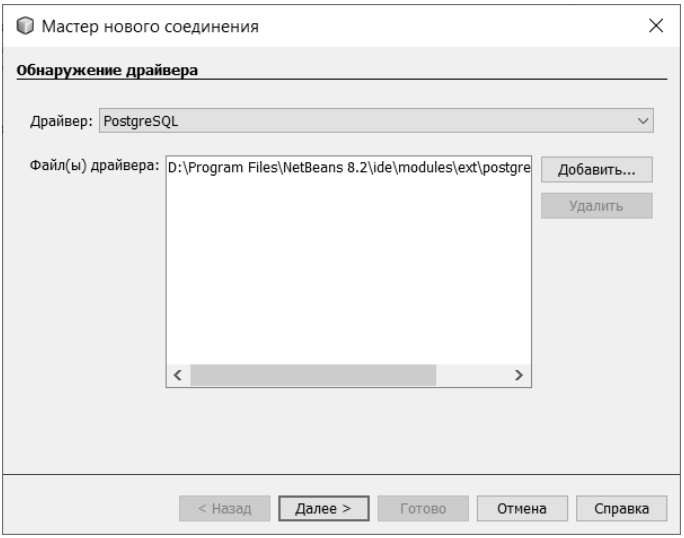



*Рис. 8.8.* Настройка подключения БД

Выбираем драйвер для PostgreSQL и нажимаем на Далее.

На рис. 8.9 представлены поля, которые нужно заполнить. По умолчанию уже будет всё введено, кроме названия базы данных, пользователя и пароля. В данном случае пользователь postgres, а пароль был задан при установке СУБД. Далее необходимо нажать на кнопку Проверить соединение, чтобы проверить соединение с БД. Должно прийти сообщение «Соединение установлено успешно». Далее нажимаем кнопку Готово.

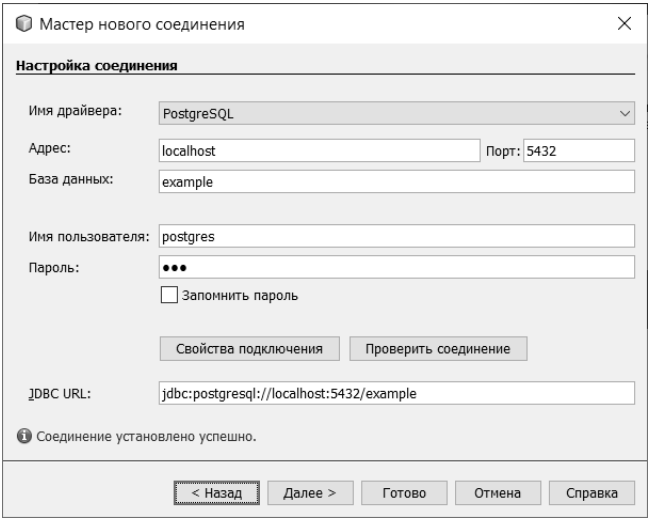



*Рис. 8.9.* Результат подключения

Создание таблиц можно выполнить через SQL-запросы. Для этого щелчком правой кнопки мыши по созданной строке соединения (рис. 8.7) вызовите контекстное меню и выберите пункт Выполнить команду… В левой части среды разработки появится новая вкладка, где можно будет ввести SQL-запрос и создать по очереди две таблицы.

**Шаг № 6 – создание сущностей**

Создадим класс сущности User в пакете models.

```
package models;

import javax.persistence.*;
import java.util.ArrayList;
```

```
import java.util.List;

@Entity
@Table (name = "users")
public class User {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "user_id")
   private int id;
   @Column(name = "user_name")
   private String name;
   //можно не указывать Column name, если оно совпадает с названием столбца в таблице
   @Column(name = "user_age")
   private int age;
// описание связи между таблицами
   @OneToMany(mappedBy = "user", cascade = CascadeType.ALL, orphanRemoval = true)
   private List<Auto> autos;

   public User() {    }
   public User(String name, int age) {
      this.name = name;
      this.age = age;
      autos = new ArrayList<Auto>();
   }
   public void addAuto(Auto auto) {
      auto.setUser(this);
      autos.add(auto);
   }
   public void removeAuto(Auto auto) {
      autos.remove(auto);
   }
   public int getId() {
      return id;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
```

```
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public List<Auto> getAutos() {
        return autos;
    }
    public void setAutos(List<Auto> autos) {
        this.autos = autos;
    }
    @Override
    public String toString() {
        return "models.User{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

Аннотация OneToMany означает, что одному объекту класса user может соответствовать несколько машин. Настройка "mappedBY" указывает на поле user класса Auto. Настройка orphanRemoval. Если мы удалим юзера из БД — все связанные с ним автомобили также будут удалены.

Класс сущности Auto в пакете models.

```
package models;

import javax.persistence.*;

@Entity
@Table(name = "autos")
public class Auto {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "auto_id")
    private int id;
    @Column (name = "auto_model")
```

```
    private String model;
    @Column (name = "auto_color")
    private String color;
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "user_id")
    private User user;
    public Auto() {    }
    public Auto(String model, String color) {
        this.model = model;
        this.color = color;
    }
    public int getId() {
        return id;
    }
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public User getUser() {
        return user;
    }
    public void setUser(User user) {
        this.user = user;
    }
    @Override
    public String toString() {
        return color + " " + model;
    }
}
```

Аннотация @ManyToOne (многим Auto может соответствовать один User). Аннотация @JoinColumn указывает, через какой столбец в таблице autos происходит связь с таблицей users (внешний ключ).

Дополнительный класс Driver не связан ни с какой таблицей, но потребуется для передачи данных в браузер пользователю.

```
public class Driver {
    private String name;
    private int age;
    private String model;
    private String color;
    public Driver() {}
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
}
```

Далее добавим в папку utils класс, в котором будет создаваться объект sessionFactory для получения сессии. Класс называется HibernateSessionFactoryUtil.

```
package utils;

import models.Auto;
import models.User;
import org.hibernate.SessionFactory;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;

public class HibernateSessionFactoryUtil {
   private static SessionFactory sessionFactory;

   private HibernateSessionFactoryUtil() {}
   public static SessionFactory getSessionFactory() {
      if (sessionFactory == null) {
         try {
            Configuration configuration = new Configuration().configure();
            configuration.addAnnotatedClass(User.class);
            configuration.addAnnotatedClass(Auto.class);
            StandardServiceRegistryBuilder builder = new StandardServiceRegis-
tryBuilder().applySettings(configuration.getProperties());
            sessionFactory = configuration.buildSessionFactory(builder.build());
         } catch (Exception e) {
            System.out.println("Исключение!" + e);
         }
      }
      return sessionFactory;
   }
}
```

В этом классе создаем новый объект конфигураций Configuration и передаем ему те классы, которые он должен воспринимать как сущности – User и Auto.

В коде используется метод configuration.getProperties(). Properties – это параметры для работы hibernate, указанные в специальном файле hibernate.cfg.xml. Этот файл создался автоматически в папке по умолчанию.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
      "http://hibernate.org/dtd/hibernate-configuration-3.0.dtd">
```

```
<hibernate-configuration>
   <session-factory>
<property
name="hibernate.connection.driver_class">org.postgresql.Driver</property>
<property
name="hibernate.connection.url">jdbc:postgresql://localhost:5432/example</property>
<property name="hibernate.connection.username">postgres</property>
<property name="hibernate.connection.password">xxx</property>
<property
name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</property>
   </session-factory>
</hibernate-configuration>
```

Hibernate.cfg.xml считывается при создании объекта: configuration = new Configuration().configure(). В файле содержаться параметры соединения с БД.

Далее необходим класс UserDAO. DAO (data access object) – один из наиболее распространенных паттернов проектирования – "Доступ к данным". Его смысл прост – создать в приложении слой, который отвечает только за доступ к данным. Достать данные из БД, обновить, удалить и т. д.

```
package dao;

import models.Auto;
import models.User;
import org.hibernate.Session;
import org.hibernate.Transaction;
import utils.HibernateSessionFactoryUtil;
import java.util.List;

public class UserDao {

   public User findById(int id) {          // поиск владельца по id
      return HibernateSessionFactoryUtil.getSessionFactory().openSession()
.get(User.class, id);
   }
   public void save(User user) {          // запись владельца в таблицу
      Session session = HibernateSessionFactoryUtil.getSessionFactory()
```

```
.openSession();
        Transaction tx1 = session.beginTransaction();
        session.save(user);
        tx1.commit();
        session.close();
    }
    public void update(User user) {        // обновление данных владельца
        Session session = HibernateSessionFactoryUtil.getSessionFactory()
.openSession();
        Transaction tx1 = session.beginTransaction();
        session.update(user);
        tx1.commit();
        session.close();
    }
    public void delete(User user) {    // удаления из таблиц данных владельца
        Session session = HibernateSessionFactoryUtil.getSessionFactory()
.openSession();
        Transaction tx1 = session.beginTransaction();
        session.delete(user);
        tx1.commit();
        session.close();
    }
    public Auto findAutoById(int id) {          // поиск авто по id владельца
        return HibernateSessionFactoryUtil.getSessionFactory()
.openSession().get(Auto.class, id);
    }
    public List<User> findAll() {            // получение списка владельцев авто
        List<User> users = (List<User>)  HibernateSessionFactoryUtil.
getSessionFactory().openSession().createQuery("From User").list();
        return users;
    }
}
```

В большинстве этих методов получаем объект Session (сессия соединения с нашей БД) с помощью класса HibernateSessionFactoryUtil, создаем в рамках этой сессии одиночную транзакцию, выполняем необходимые преобразования данных, сохраняем результат транзакции в БД и закрываем сессию.

Логика помещена в класс UserService:

```
package services;

import dao.UserDao;
import models.Auto;
import models.User;
import org.hibernate.Query;
import java.util.List;

public class UserService {
   private UserDao usersDao = new UserDao();
   public UserService() {    }
   public User findUser(int id) {
      return usersDao.findById(id);
   }
   public void saveUser(User user) {
      usersDao.save(user);
   }
   public void delete(User user) {
      Session session = HibernateSessionFactoryUtil.
getSessionFactory().openSession();
      Transaction tx1 = session.beginTransaction();
      Query q = session.createSQLQuery("delete From Autos where user_id = "
+user.getId());
      q.executeUpdate();
      Query q1 = session.createSQLQuery("delete From Users where user_id = "
+user.getId());
      q1.executeUpdate();
      tx1.commit();
      session.close();
   }
   public void updateUser(User user) {
      usersDao.update(user);
   }
   public List<User> findAllUsers() {
      return usersDao.findAll();
   }
   public Auto findAutoById(int id) {
      return usersDao.findAutoById(id);
   }
```

```
public User findByName(String name) {
      List<User> users = (List<User>)  HibernateSessionFactoryUtil.
getSessionFactory().openSession().
createQuery("From User where user_name = '" +name+ "'").list();
      if (!users.isEmpty())
         return (users.get(0));
      else return new User("NoName");
   }
}
```

Service – слой данных в приложении, отвечающий за выполнение бизнес-логики. Если ваша программа должна выполнить какую-то бизнес-логику – она делает это через сервисы. Сервис содержит внутри себя UserDao и в своих методах вызывает методы DAO. При большом количестве объектов и сложной логике разбиение приложения на слои дает огромные преимущества.

Для проверки работы реализованных методов создадим web-страницы.

Главная страница index.html (рис. 8 10):

```
<html>
   <head>
      <title>Пример Hibernate</title>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
   </head>
   <BODY BGCOLOR="#FDF5E6">
      <H1 align="CENTER"> Введите данные водителя и машины  </H1>
      <FORM action='UserController' method="post">
         <table>
            <tr>
               <td> Имя:</td>
               <td><INPUT TYPE="text" NAME="name"></td>
            </tr>
            <tr>
               <td> Возраст: </td>
               <td><INPUT TYPE="text" NAME="age"></td>
            </tr>
            <tr>
               <td> Марка: </td>
               <td><INPUT TYPE="text" NAME="model"></td>
```

```
            </tr>
            <tr>
               <td>Цвет:   </td>
               <td><INPUT TYPE="text" NAME="color"></td>
            </tr>
         </table>
         <INPUT TYPE="SUBMIT" name = "save" value="Сохранить">
         <br><br>
      </FORM>
Удалить водителя
      <form action="UserController">
         Имя: <INPUT TYPE="text" NAME="name">
         <INPUT TYPE="SUBMIT" name= "delete" value="Удалить">
      </form>
      <br><br>
      Получить список водителей
      <form action="UserController">
         <INPUT TYPE="SUBMIT" name= "list" value="Список">
      </form>
   </BODY>
</html>
```

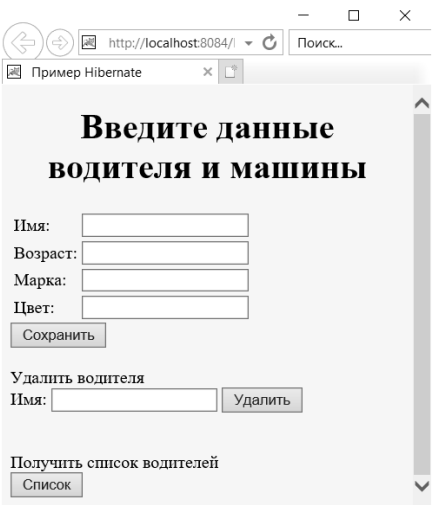



*Рис. 8.10.* Стартовая тестовая страница приложения

Для отображения списка водителей и автомобилей ListUser.jsp:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
       <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
      <title>Пример JSP JSTL</title>
   </head>
   <body>
      <h1>Список водителей</h1>
      <table border="1" cellspacing="0" cellpadding="2">
         <tr>
            <td>Имя</td>
            <td>Возраст</td>
            <td>Марка</td>
            <td>Цвет</td>
         </tr>
            <c:forEach items="${driver}" var="dr">
            <tr>
               <td>${dr.name}</td>
               <td>${dr.age}</td>
               <td>${dr.model}</td>
               <td>${dr.color}</td>
            </tr>
         </c:forEach>
      </table>
   </body>
</html>
```

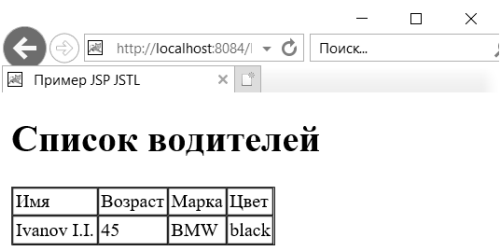



*Рис. 8.11.* Вывод списка водителей
и автомобилей

Вывод ошибки при выполнении запросов Error.jsp:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Page error</title>
   </head>
   <body>
      <h1>Ошибка запроса!</h1>
      <%=request.getAttribute("err") %>
   </body>
</html>
```

Все запросы со страницы index.html направляются контроллеру UserController, который создает объект UserService для выполнения запросов:

```
package controllers;

import dao.UserDao;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.List;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;
import models.Auto;
import models.Driver;
import models.User;
import services.UserService;

@WebServlet(name = "UserController", urlPatterns = {"/UserController"})
public class UserController extends HttpServlet {
```

```
   protected void processRequest(HttpServletRequest request, HttpServletResponse
response)          throws ServletException, IOException {
      response.setContentType("text/html; charset=UTF-8");
      response.setCharacterEncoding("utf-8");          //Кодировка данных
      UserService userService = new UserService();
      ArrayList<Driver> list1=new ArrayList<Driver>();

      if (request.getParameter("save")!=null) {          //запрос на запись в БД
         String name = request.getParameter("name");
         int age = (new Integer(request.getParameter("age"))).intValue();
         User user = new User(name, age);
         userService.saveUser(user); // запись в таблицу users
         String model = request.getParameter("model");
         String color = request.getParameter("color");
         Auto auto = new Auto(model, color);
         auto.setUser(user);
         user.addAuto(auto);
         userService.updateUser(user);                    //запись  в таблицу autos
         RequestDispatcher requestDispatcher =
request.getRequestDispatcher("index.html");
         requestDispatcher.forward(request, response); // переход на страницу
      }                                                   // index.html
      else if (request.getParameter("delete") != null) {
         String name = request.getParameter("name");
         User user = userService.findUserbyName(name);
         userService.deleteUser(user);
         RequestDispatcher requestDispatcher =
request.getRequestDispatcher("index.html");
         requestDispatcher.forward(request, response);
      }
      else if (request.getParameter("list") != null) {     // получение списка
         List<User> list;
         list = userService.findAllUsers();
         if (list!=null) {
            for(int i=0; i<list.size(); i++){
               int t = list.get(i).getId();
               Auto auto = userService.findAutoById(t);
```

```
                Driver driver= new Driver();
                driver.setName(list.get(i).getName());
                driver.setAge(list.get(i).getAge());
                driver.setModel(auto.getModel());
                driver.setColor(auto.getColor());
                list1.add(driver);
            }
            request.setAttribute("driver", list1);
            RequestDispatcher requestDispatcher =
request.getRequestDispatcher("./jsp/ListUser.jsp");
            requestDispatcher.forward(request, response);     // переход на страницу
        } else {                                               // ListUser.jsp
            String error = "Список пуст!";
            request.setAttribute("err1", error);
            RequestDispatcher requestDispatcher =
 request.getRequestDispatcher("./jsp/Error.jsp");
            requestDispatcher.forward(request, response);   // переход на страницу
        }                                                      // Error.jsp
    }
  }
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse re-
sponse)
        throws ServletException, IOException {
    processRequest(request, response);
  }
  @Override
  protected void doPost(HttpServletRequest request, HttpServletResponse re-
sponse)
        throws ServletException, IOException {
    processRequest(request, response);
  }
}
```

Удаление водителя и его автомобилей по фамилии показано на рис. 8.12.

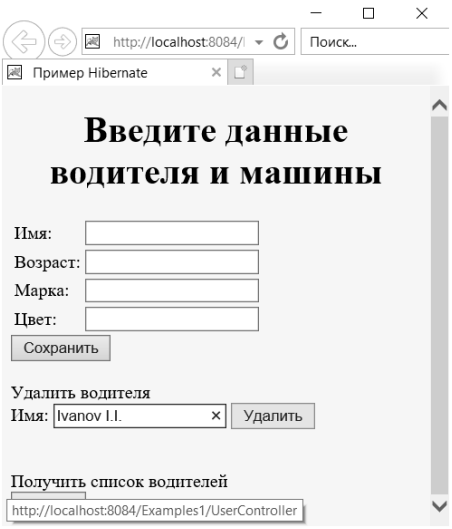



*Рис. 8.12.* Удаление данных водителя из таблиц

Полученный в результате список приведен на рис. 8.13.

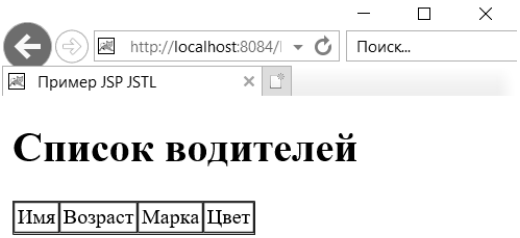



*Рис. 8.13.* Список водителей после удаления

## ПРАКТИЧЕСКИЕ ЗАДАНИЯ

1. Реализуйте для web-приложения Example1 все остальные перечисленные в задаче CRUD операции.
2. Для доступа к данным используйте технологию Hibernate.

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое Hibernate Framework?
2. Какие важные преимущества дает использование Hibernate Framework?
3. Каковы преимущества Hibernate над JDBC?
4. Назовите некоторые важные интерфейсы Hibernate.
5. Что такое конфигурационный файл Hibernate?
6. Что такое Hibernate mapping file?
7. Назовите некоторые важные аннотации, используемые для отображения в Hibernate.
8. Что вы знаете о Hibernate SessionFactory и как его сконфигурировать?
9. Является ли Hibernate SessionFactory потокобезопасным?
10. Как получить Hibernate Session и что это такое?
11. Является ли Hibernate Session потокобезопасным?
12. В чем разница между openSession и getCurrentSession?
13. Какая разница между методами Hibernate Session get() и load()?
14. Каковы различные состояния у entity bean?
15. Как используется вызов метода Hibernate Session merge()?
16. В чем разница между Hibernate save(), saveOrUpdate() и persist()?
17. Что произойдет, если будет отсутствовать конструктор без аргументов у Entity Bean?
18. В чем разница между sorted collection и ordered collection? Какая из них лучше?
19. Какие типы коллекций в Hibernate вы знаете?
20. Как реализованы Join'ы Hibernate?
21. Почему мы не должны делать Entity class как final?
22. Что вы знаете о HQL и каковы его преимущества?
23. Что такое Query Cache в Hibernate?
24. Можем ли мы выполнить нативный запрос SQL (sql native) в Hibernate?
25. Назовите преимущества поддержки нативного sql в Hibernate.
26. Что такое Named SQL Query?
27. Каковы преимущества Named SQL Query?
28. Расскажите о преимуществах использования Hibernate Criteria API.

## БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Перри Б. У.* Java сервлеты и JSP: сборник рецептов / Б. У. Перри. – Изд. 2-е / пер. с англ. – Москва : КУДИЦ-ПРЕСС, 2006. – 768 с.

2. *Хабибуллин И. Ш.* Создание распределенных приложений на Java2 / И. Ш. Хабибуллин. – Санкт-Петербург : БХВ-Петербург, 2002. – 704 с.

3. Hibernate Java. – URL: https://proselytear.gitbooks.io/java/3orm/42-hibernate.html (дата обращения: 12.01.2021).

4. Ваше первое приложение на Hibernate. – URL: https://javarush.ru/groups/posts/hibernate-java. (дата обращения: 12.01.2021).

## ОГЛАВЛЕНИЕ

**Васюткина Ирина Александровна**

**РАЗРАБОТКА СЕРВЕРНОЙ ЧАСТИ WEB-ПРИЛОЖЕНИЙ НА JAVA**

**Учебное пособие**

Выпускающий редактор *И.П. Брованова*
Корректор *И.Е. Семенова*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *Н.В. Гаврилова*

Налоговая льгота – Общероссийский классификатор продукции
Издание соответствует коду 95 3000 ОК 005-93 (ОКП)

---

Подписано в печать 13.04.2021. Формат 60 × 84 1/16. Бумага офсетная
Тираж 50 экз. Уч.-изд. л. 4,88. Печ. л. 5,25. Изд. № 21. Заказ № 438
Цена договорная

---

Отпечатано в типографии
Новосибирского государственного технического университета
630073, г. Новосибирск, пр. К. Маркса, 20