Глава 19

JAVA SERVER PAGES

Технология Java Server Pages (JSP) была разработана компанией Sun Microsystems, чтобы облегчить создание страниц с динамическим содержанием.

В то время как сервлеты наилучшим образом подходят для выполнения контролирующей функции приложения в виде обработки запросов и определения вида ответа, страницы JSP выполняют функцию формирования текстовых документов типа HTML, XML, WML и некоторых других.

Под терминами "динамическое/статическое содержание" обычно понимаются не части JSP, а содержание Web-приложения:

- динамические ресурсы, изменяемые в процессе работы: сервлеты, JSP, а также java-код;
- статические ресурсы, не изменяемые в процессе работы – HTML, JavaScript, изображения и т.д.

Смысл разделения динамического и статического содержания в том, что статические ресурсы могут находиться под управлением HTTP-сервера, в то время как динамические нуждаются в движке (Servlet Engine) и в большинстве случаев в доступе к уровню данных.

Рекомендуется разделить и разрабатывать параллельно две части приложения: Web-приложение, состоящее только из динамических ресурсов, и Web-приложение, состоящее только из статических ресурсов.

Некоторые преимущества использования JSP-технологии над другими методами создания динамического содержания страниц:

- *Разделение динамического и статического содержания.*
  Возможность разделить логику приложения и дизайн Web-страницы снижает сложность разработки Web-приложений и упрощает их поддержку.
- *Независимость от платформы.*
  Так как JSP-технология, основанная на языке программирования Java, не зависит от платформы, то JSP могут выполняться практически на любом Web-сервере. Разрабатывать JSP можно на любой платформе.
- *Многократное использование компонентов.*
  Использование JavaBeans и Enterprise JavaBeans (EJB) позволяет многократно использовать компоненты, что ускоряет создание Web-сайтов.
- *Скрипты и теги.*
  Спецификация JSP объявляет собственные теги, кроме того, JSP поддерживают как JavaScript, так и HTML-теги. JavaScript обычно используется, чтобы добавить функциональные возможности на уровне HTML-страницы. Теги обеспечивают возможность использования JavaBean и выполнение обычных функций.

Чтобы облегчить внедрение динамической структуры, JSP использует ряд тегов, которые дают возможность проектировщику страницы вставить значение полей объекта JavaBean в файл JSP.

Содержимое Java Server Pages (теги HTML, теги JSP и скрипты) переводится в сервлет код-сервером. Этот процесс ответствен за трансляцию как динамических, так и статических элементов, объявленных внутри файла JSP. Об архитектуре сайтов, использующих JSP/Servlet-технологии, часто говорят как о thin-client (использование ресурсов клиента незначительно), потому что большая часть логики выполняется на сервере.

JSP составляется из стандартных HTML-тегов, JSP-тегов, action-тегов, JSTL и пользовательских тегов. В спецификации JSP 2.0 существует пять основных тегов:

<%@ *директива* **%>** – используются для установки параметров серверной страницы JSP.

<%! *объявление* **%>** – содержит переменные Java и методы, которые вызываются в expression-блоке и являются полями генерируемого сервлета. Объявление не должно производить запись в выходной поток **out** страницы, но может быть использовано в скриптлетах и выражениях.

<% *скриптлет* **%>** – вживление Java-кода в JSP-страницу. Скриптлеты обычно используют маленькие блоки кода и выполняются во время обработки запроса клиента. Когда все скриптлеты собираются воедино в том порядке, в котором они записаны на странице, они должны представлять собой правильный код языка программирования. Контейнер помещает код Java в метод **_jspService()** на этапе трансляции.

<%= *вычисляемое выражение* **%>** – операторы языка Java, которые вычисляются, после чего результат вычисления преобразуется в строку **String** и посылается в поток **out**.

<%-- *JSP-комментарий* --%> – комментарий, который не отображается в исходных кодах JSP-страницы после этапа выполнения.

Стандартные элементы action

Большинство тегов, объявленных выше, применяются не так уж часто. Наиболее используемыми являются стандартные действия версии JSP 2.0. Они позволяют создавать правильные JSP –документы с помощью следующих тегов:

- **jsp:declaration** – объявление, аналогичен тегу **<%! ... %>**;
- **jsp:scriptlet** – скриптлет, аналогичен тегу **<% ... %>**;
- **jsp:expression** – скриптлет, аналогичен тегу **<%= ... %>**;
- **jsp:text** – вывод текста;
- **jsp:useBean** – позволяет использовать экземпляр компонента Java Bean. Если экземпляр с указанным идентификатором не существует, то он будет создан с областью видимости **page** (страница), **request** (запрос), **session** (сессия) или **application** (приложение). Объявляется, как правило, с атрибутами **id** (имя объекта), **scope** (область видимости), **class** (полное имя класса), **type** (по умолчанию **class**).

```
<jsp:useBean id="ob"
             scope="session"
                   class="test.MyBean" />
```

Создан объект **ob** класса **MyBean**, и в дальнейшем через этот объект можно вызывать доступные методы класса. Специфика компонентов JavaBean в том, что если компонент имеет поле **field**, экземпляр компонента имеет параметр **field**, а метод, устанавливающий значение, должен называться **setField(type value)**, возвращающий значение – **getField()**.

```
package test;
public class MyBean {
     private String field = "нет информации";
     public String getField() {
           return info;
     }
     public void setField(String f) {
           field = f;
     }
}
```

- **jsp:setProperty** – позволяет устанавливать значения полей указанного в атрибуте **name** объекта. Если установить значение **property** в «*», то значения свойств компонента JavaBean будут установлены таким образом, что будет определено соответствие между именами параметров и именами методов-установщиков (setter-ов) компонента:

```
<jsp:setProperty name="ob"
                        property="field"
                        value="привет" />
```

- **jsp:getProperty** – получает значения поля указанного объекта, преобразует его в строку и отправляет в неявный объект **out**:

```
<jsp:getProperty name="ob" property="field" />
```

- **jsp:include** – позволяет включать файлы в генерируемую страницу при запросе страницы:

```
<jsp:include page="относительный URL"
                        flush="true"/>
```

- **jsp:forward** – позволяет передать запрос другой странице:

```
<jsp:forward page="относительный URL"/>
```

- **jsp:plugin** – замещается тегом **<OBJECT>** или **<EMBED>**, в зависимости от типа браузера, в котором будет выполняться подключаемый апплет или Java Bean.
- **jsp:params** – группирует параметры внутри тега **jsp:plugin**.
- **jsp:param** – добавляет параметры в объект запроса, например в элементах **forward**, **include**, **plugin**.
- **jsp:fallback** – указывает содержимое, которое будет использоваться браузером клиента, если подключаемый модуль не сможет запуститься. Используется внутри элемента **plugin**.

В качестве примера можно привести следующий фрагмент:

```
<jsp:plugin type="bean | applet"
                  code="test.com.ReadParam"
```

```
                    width="250"
                    height="250">
            <jsp:params>
               <jsp:param name="bNumber" value="7" />
               <jsp:param name="state" value="true" />
            </jsp:params>
            <jsp:fallback>
                  <p> unable to start plugin </p>
            </jsp:fallback>
      </jsp:plugin>
```

Код апплета находится в примере 5 главы 11, и пакет, в котором он объявлен, должен быть расположен в корне папки **/WEB-INF**, а не в папке **/classes**.

Элементы **<jsp:attribute>**, **<jsp:body>**, **<jsp:invoke>**, **<jsp:doBody>**, **<jsp:element>**, **<jsp:output>** используются в основном при включении в страницу пользовательских тегов.

JSP-документ

Предпочтительно создавать JSP-страницу в виде JSP-документа – корректного XML-документа, который ссылается на определенное пространство имен, содержит стандартные действия JSP, пользовательские теги и теги ядра JSTL, XML-эквиваленты директив JSP. В JSP-документе вышеперечисленные пять тегов неприменимы, поэтому их нужно заменять стандартными действиями и корректными тегами. JSP-документы необходимо сохранять с расширением **.jspx**.

Директива **taglib** для обычной JSP:

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

для JSP-документа:

```
<jsp:root xmlns:c="http://java.sun.com/jsp/jstl/core"/>
```

Директива **page** для обычной JSP:

```
<%@ page contentType="text/html"%>
```

для JSP-документа:

```
<jsp:directive.page contentType="text/html" />
```

Директива **include** для обычной JSP:

```
<%@ include file="file.jspf"%>
```

для JSP-документа:

```
<jsp:directive.include file="file.jspf" />
```

Ниже приведены два примера, демонстрирующие различие применения стандартных действий и тегов при создании JSP-страниц и JSP-документов.

```
<!--пример # 1 : обычная jsp-страница: page.jsp -->
<%@ page contentType="text/html; charset=Cp1251" %>
<html><head><title>JSP-страница</title></head>
<%! private int count = 0;
      String version = new String("J2EE 1.5");
      private String getName(){return "J2EE 1.6";} %>
```

```
<% out.println("Значение count: "); %>
<%= count++ %>
<br/>
<% out.println("Значение count после инкремента: " +
count); %>
<br/>
<% out.println("Старое значение version: "); %>
<%= version %>
<br/>
<% version=getName();
out.println("Новое значение version: " + version); %>
</html>
```

Версия в виде JSP-документа несколько более громоздка, но читать и искать ошибки в таком документе проще, нежели в предыдущем.

```
<!--пример # 2 : правильный jsp-документ : page.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
                                          version="2.0">
      <jsp:directive.page contentType=
                              "text/html; charset=UTF-8" />
      <html><body>
      <jsp:declaration>
            private int count = 0;
            String version = new String("J2EE 1.5");
            private String getName(){return "J2EE 1.6";}
      </jsp:declaration>
      <jsp:scriptlet>
            out.println("Значение count: ");
      </jsp:scriptlet>
      <jsp:expression>
            count++
      </jsp:expression>
      <br />
      <jsp:scriptlet>
            out.println("Значение count после инкремента:"
                                          + count);
      </jsp:scriptlet>
      <br/>
      <jsp:scriptlet>
      out.println("Старое значение version: ");
      </jsp:scriptlet>
      <jsp:expression> version </jsp:expression>
      <br/>
      <jsp:scriptlet> version=getName();
      out.println("Новое значение version: " + version);
      </jsp:scriptlet>
      </body></html>
</jsp:root>
```

Далее в главе примеры будут приведены в виде JSP-документов.

JSTL

JSP-страницы, включающие скриптлеты, элементы action (стандартные действия) и пользовательские теги, не могут быть технологичными без использования JSTL (JSP Standard Tag Library). Создание страниц с применением JSTL позволяет упростить разработку и отказаться от вживления Java-кода в JSP. Как было показано ранее, страницы со скриптлетами трудно читаемы, что вызывает проблемы как у программиста, так и у веб-дизайнера, не обладающего глубокими познаниями в Java.

Библиотеку JSTL версии 1.1.2 (**jstl-1.1.2.jar** и **standard-1.1.2.jar**) или более позднюю версию можно загрузить с сайта apache.org. Библиотеки следует разместить в каталоге **/lib** проекта. При указании значения параметра **xmlns** элемента **root** (для JSP-страницы значение параметра **taglib uri=""**) необходимо быть внимательным, так как если адрес будет указан неправильно, JSP-страница не сможет иметь доступ к тегам JSTL. Проверить правильность значение параметра **uri** (оно же справедливо и для параметра **xmlns**) можно в файле подключаемой библиотеки (например **c.tld**). Простейшая JSP с применением тега JSTL, выводящим в браузер приветствие будет выглядеть следующим образом:

```
<!--пример # 3  : правильный jsp-документ: simple.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
            xmlns:c=http://java.sun.com/jsp/jstl/core
            version="2.0">
<jsp:directive.page contentType=
            "text/html; charset=Utf-8"/>
<html><body>
<c:out value="Welcome to JSTL"/>
</body></html>
</jsp:root>
```

Тег **<c:out/>** отправляет значение параметра **value** в поток **JspWriter**.

JSTL предоставляет следующие возможности:

- поддержку Expression Language, что позволяет разработчику писать простые выражения внутри атрибутов тега и предоставляет “прозрачный” доступ к переменным в различных областях видимости страницы;
- организацию условных переходов и циклов, основанную на тегах, а не на скриптовом языке;
- простое формирование доступа (URL) к различным ресурсам;
- простую интернационализацию JSP;
- взаимодействие с базами данных;
- обработку XML, а также форматирование и разбор строк.

Expression Language

В JSTL вводится понятие Expression Language (EL). EL используется для упрощения доступа к данным, хранящимся в различных областях видимости (**page**, **request**, **application**) и вычисления простых выражений.

EL вызывается при помощи конструкции **"${**имя**}"**.

Начиная с версии спецификации JSP 2.0 / JSTL 1.1, EL является частью JSP и поддерживается без всяких сторонних библиотек. С версии web-app 2.4 атрибут

isELIgnored по умолчанию имеет значение **true**. В более ранних версиях необходимо указывать его в директиве **page** со значение **true**.

EL-идентификатор ссылается на переменную, возвращаемую вызовом **PageContext.findAttribute(**имя**)**. В общем случае переменная может быть сохранена в любой области видимости: **page**(**PageContext**), **request**(**HttpServletRequest**), **session** (**HttpSession**), **application** (**ServletContext**). В случае если переменная не найдена, возвращается **null**. Также возможен доступ к параметрам запроса через предопределённый объект **paramValues** и к заголовкам запроса через **requestHeaders**.

Данные приложения, как правило, состоят из объектов, соответствующих спецификации JavaBeans, или представляют собой коллекции, такие как **List**, **Map**, **Array** и др. EL предоставляет доступ к этим объектам при помощи операторов **".**" и **"[]"**. Применение этих операторов зависит от типа объекта. Например:

```
<c:out value="${student.name}"/>
```

```
<!--пример # 4 : правильный jsp-документ : simple2.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:c="http://java.sun.com/jsp/jstl/core"
          version="2.0">
<jsp:directive.page contentType=
                              "text/html; charset=Utf-8"/>
<html>
<head><title>Простое использование EL</title></head>
<body>
<c:set var="login" value="Бендер" scope="page"/>
<c:out value="${login} in Rio"/>
<br/>
<c:out value="Бендер в байтовом виде: ${login.bytes} "/>
</body></html>
</jsp:root>
```

С помощью оператора **".**" можно вызывать некоторые методы класса, к которому принадлежит объект. Вызов **login.bytes** в переводе на обычную Java означает **login.getBytes()**.

В результате запуска этого документа в браузер будет выведено:

Бендер in Rio
Бендер в байтовом виде: [B@edf730

Операторы в EL поддерживают наиболее часто используемые манипуляции данными.

Типы операторов:

Стандартные операторы отношения:
== (или **eq**), **!=** (или **neq**), **<** (или **lt**), **>** (или **gt**), **<=** (или **le**), **>=** (или **ge**).
Арифметические операторы: **+**, **-**, *****, **/** (или **div**), **%** (или **mod**).
Логические операторы: **&&** (или **and**), **||** (или **or**), **!** (или **not**).
Оператор **empty** – используется для проверки переменной на **null**, или "пустое значение". Термин "пустое значение" зависит от типа проверяемого объекта. Например, нулевая длина для строки или нулевой размер для коллекции.

Например:

```
<c:if test="${not empty user and user.name neq 'guest'}>
  User is Customer.
</c:if>
```

Автоматическое приведение типов

Данные не всегда имеют тот же тип, который ожидается в EL-операторе. EL использует набор правил для автоматического приведения типов. Например, если оператор ожидает параметр типа **Integer**, то значение идентификатора будет приведено к типу **Integer** (если это возможно).

Неявные объекты

JSP-страница всегда имеет доступ ко многим функциональным возможностям сервлета, создаваемым Web-контейнером по умолчанию. Неявный объект:

- **request** – представляет запрос клиента. Обычно объект является экземпляром класса, реализующего интерфейс **javax.servlet.http.HttpServletRequest**. Для протокола, отличного от HTTP, это будет объект реализации интерфейса **javax.servlet.ServletRequest**. Область видимости в пределах страницы.
- **response** – представляет ответ клиенту. Обычно объект является экземпляром класса, реализующего интерфейс **javax.servlet.http.HttpServletResponse**. Для протокола, отличного от HTTP, это будет объект реализации интерфейса **javax.servlet.ServletResponse**. Область видимости в пределах страницы.
- **pageContext** – определяет контекст JSP-страницы и предоставляет доступ к неявным объектам. Объект класса **javax.servlet.jsp.PageContext**. Область видимости в пределах страницы.
- **session** – создается контейнером в соответствии с протоколом HTTP и является экземпляром класса **javax.servlet.http.HttpSession**, предоставляет информацию о сессии клиента, если такая была создана. Область видимости в пределах сессии.
- **application** – контейнер, в котором исполняется JSP-страница, является экземпляром класса **javax.servlet.ServletContext**. Область видимости в пределах приложения.
- **out** – содержит выходной поток сервлета. Информация, посылаемая в этот поток, передается клиенту. Объект является экземпляром класса **javax.servlet.jsp.JspWriter**. Область видимости в пределах страницы.
- **config** – содержит параметры конфигурации сервлета и является экземпляром класса **javax.servlet.ServletConfig**. Область видимости в пределах страницы.

- **page** – ссылка **this** для текущего экземпляра данной страницы является объектом **java.lang.Object**. Область видимости в пределах страницы.
- **exception** – представляет собой исключение одного из подклассов класса **java.lang.Throwable**, которое передается странице сообщения об ошибках и доступно только на ней.

JSTL core

Библиотека тегов JSTL состоит из четырёх групп тегов: основные теги – **core**, теги форматирования – **fomatting**, теги для работы с SQL – **sql**, теги для обработки XML – **xml**.

Library	Actions	Description
core	14	Основные: **if**/**then** выражения и **switch** конструкции; вывод; создание и удаление контекстных переменных; управление свойствами JavaBeans компонентов; перехват исключений; итерирование коллекций; создание URL и импортирование их содержимого.
formatting	12	Интернационализация и форматирование: установка локализации; локализация текста и структуры сообщений; форматирование и анализ чисел, процентов, денег и дат.
sql	6	Доступ к БД: описание источника данных; выполнение запросов, обновление данных и транзакций; обработка результатов запроса.
xml	10	XML-анализ и преобразование: преобразование XML; доступ и преобразование XML через XPath и XSLT.

Библиотека **core** содержит в себе наиболее часто используемые теги.
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c" %> – для обычной JSP.
<jsp:root version="1.2" xmlns:c=
"http://java.sun.com/jstl/core"**> ...</jsp:root>** – для XML формата JSP.

Теги общего назначения:

<c:out /> — вычисляет и выводит значение выражения;

<**c:set />** — устанавливает переменную в указанную область видимости;

<c:remove /> — удаляет переменную из указанной области видимости;

<c:catch /> — перехватывает обработку исключения.

Теги условного перехода:

<c:if /> — тело тега вычисляется только в том случае, если значение выражения **true**;
<c:choose /> (**<c:when />**, **<c:otherwise />**) – то же что и **<c:if />** с поддержкой нескольких условий и действия, производимого по умолчанию.

Итераторы:

<c:forEach /> — выполняет тело тега для каждого элемента коллекции;

<c:forTokens /> — выполняет тело тега для каждой лексемы в строке.

Теги обработки URL:

<c:redirect /> — перенаправляет запрос на указанный **URL**;

<c:import/> — добавляет на JSP содержимое указанного WEB-ресурса;

<c:url /> — формирует адрес с учётом контекста приложения **request.getContextPath()**.

<c:param /> — добавляет параметр к запросу, сформированному при помощи **<c:url/>**.

Ниже приведено несколько примеров, иллюстрирующих применение основных тегов из группы **core**.

```
<!--пример #5 : демонстрация работы тегов  c:set, c:remove, c:if, c:out: core1.jspx
-->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      version="2.0">
<jsp:directive.page contentType=
                              "text/html; charset=Utf-8" />
<html><head><title>Демонстрация тегов core</title></head>
<h3> Демонстрация работы тегов c:set, c:remove, c:if, c:out
<br/></h3>
      <form>
Новое значение переменной:<input type="text" name="set" />
<br/>
Удалить переменную:<input type="checkbox" name="del" />
<br/>
<input type="submit" name="send" value="принять"/><br/>
      </form>
<c:if test="${not empty param.send }">
      <c:if test="${not empty param.set }">
            <c:set var="item" value="${param.set}"
                  scope="session"></c:set>
      </c:if>
      <c:if test="${not empty param.del }">
            <c:remove var="item"/>
      </c:if>
</c:if>
<c:if test="${not empty item }">
      <jsp:text>Значение переменной :</jsp:text>
      <c:out value="${item }"/>
</c:if>
<c:if test="${empty item and not empty param.send}">
      <jsp:text>Значение переменной: </jsp:text>
      <c:out value="пусто"/>
```

```
</c:if>
</html>
</jsp:root>
```

<!--пример # 6 : демонстрация работы тегов c:forEach, c:choose, c:when, c:otherwise : core2.jspx -->

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:c="http://java.sun.com/jsp/jstl/core"
          version="2.0">
<jsp:directive.page contentType=
          "text/html; charset=Utf-8" />
<html><head>
<title>Демонстрация тегов core</title>
</head>
<h3>Демонстрация работы тегов <br/>c:forEach, c:choose,
c:when, c:otherwise</h3>
<jsp:text>Ниже приведены случайно сгенерированные элементы
массива <br/> и сделана их оценка по отношению к числу 50:
<br/></jsp:text>
<jsp:useBean id="arr" class="chapt21.Arr" />
<c:set var="items" value="${arr.fillMap}"
       scope="session" />
    <c:forEach var="id" items="${items}">
        <c:out value="${id}" />
            <c:choose>
                <c:when test="${id > 50}" >
                    <c:out value=" - число больше 50"/>
                </c:when>
                <c:otherwise>
                    <c:out value=" - число меньше 50"/>
                </c:otherwise>
            </c:choose>
            <br/>
    </c:forEach>
</html>
</jsp:root>
```

Элемент JavaBean, используемый в данном примере, – класс **Arr**, генерирующий массив из пяти случайных чисел. Его описание хранится в файле **Arr.java**. Исходный Java-файл хранится в каталоге на верхнем уровне приложения (например каталог **build**, **src** или **JavaSourse**) в зависимости от настроек web-приложения. Значение атрибута **class** равно **chapt21.Arr** тега **jsp:useBean** и означает, что файл находится в пакете **chapt21**, а имя класса – **Arr**. EL-выражение **arr.fillMap** означает вызов метода **getfillMap()**.

// пример # 7 : javabean класс : Arr.java

```
package chapt21;
public class Arr  {
      public Arr(){}
      public  String[] getfillMap(){
```

```
      String str[] = new String[5];
      for (int i =0; i < str.length ; i++){
String r = Integer.toString((int)(Math.random()*100));
            str[i] = r;
      }
      return str;
      }
}
```

В результате в браузер будет выведено:

Демонстрация работы тегов
c:forEach, c:choose, c:when, c:otherwise

Ниже приведены случайно сгенерированные элементы массива и сделана их оценка по отношению к числу 50:
8 - число меньше 50
68 - число больше 50
84 - число больше 50
5 - число меньше 50
36 - число меньше 50

Следующие примеры показывают работу тегов по взаимодействию с другими документами и ресурсами.

```
<!--пример # 8 : демонстрация работы тегов  c:import, c:url, c:redirect, c:param :
url.jsp -->
<%@ page language="java" contentType=
            "text/html; charset=Cp1251"
            pageEncoding="Cp1251"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
            prefix="c" %>
<html><head><title>Переход по ссылке</title></head>
<body>
<h3>Данная страница демонстрирует работу тегов
<br/>c:import, c:url, c:param, c:redirect</h3><br/>
<c:import url="\WEB-INF\jspf\imp.jspf"
      charEncoding="Cp1251"/>
<c:url value="redirect.jspx" var="myUrl" />
<a href='<c:out value="${myUrl}"/>' />Перейти</a>
</body></html>
<!--пример #  9 : фрагмент, включаемый с помощью тега c:import (находится
в каталоге WEB-INF/jspf/) : imp.jspf -->
<h5>importing by using c:import from jspf</h5>
```

В результате запуска страницы **url.jsp** в браузер будет выведено:

Данная страница демонстрирует работу тегов
c:import, c:url, c:param, c:redirect.
importing by using c:import from jspf
Перейти

В **url.jsp** был импортирован фрагмент **imp.jspf**, а также с помощью тега **c:url** была задана активная ссылка, при активации которой будет осуществлен переход на страницу **redirect.jspx**.

```
<!--пример # 10 : демонстрация работы тегов  c:redirect, c:param: redirect.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
            xmlns:c="http://java.sun.com/jsp/jstl/core"
            version="2.0">
<jsp:directive.page contentType=
            "text/html; charset=Utf-8" />
<html><head><title>Демонстрация тегов core</title></head>
<body>
<c:redirect url="urldestination.jspx">
      <c:param name="fname" value="Ostap"/>
      <c:param name="lname" value="Bender"/>
</c:redirect>
</body></html>
</jsp:root>
```

В документе были объявлены два параметра и заданы их значения, а также был автоматически выполнен переход на документ **urldestination.jspx**.

```
<!--пример # 11 : конечная страница, на которую был перенаправлен запрос
и переданы данные: urldestination.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      version="2.0">
<jsp:directive.page contentType=
      "text/html; charset=Utf-8"/>
<html><head>
      <title>Демонстрация работы тега c:url</title>
</head>
<body>
<jsp:text>
      Ваш запрос был перенаправлен на эту страницу<br/>
      Параметры, переданные с помощью тега c:param:<br/>
</jsp:text>
<c:forEach var="ps" items="${param}">
<c:out value="${ps.key} - ${ps.value}"/><br/>
</c:forEach>
</body></html>
</jsp:root>
```

В результате работы документа в браузер будет выведено:

Ваш запрос был перенаправлен на эту страницу.
Параметры, переданные с помощью тега c:param:
lname - Ostap
fname - Bender

JSTL fmt

Библиотека содержит теги форматирования и интернационализации.

<%@taglib uri=”http://java.sun.com/jstl/fmt” prefix=”fmt”%>

– для обычной страницы JSP;

<jsp:root version=”1.2” xmlns:fmt=

"http://java.sun.com/jstl/fmt">...</jsp:root> — для JSP-документа.

Теги интернационализации:

<fmt:setLocale/> — устанавливает объект класса **Locale**, используемый на странице;

<fmt:setBundle/>, **<fmt:bundle/>** — устанавливают объект **ResourceBundle**, используемый на странице. В зависимости от установленной локали выбирается **ResourceBundle**, соответствующий указанному языку, стране и региону;

<fmt:message/> – выводит локализованное сообщение.

Теги форматирования:

<fmt:timeZone/>, **<fmt:setTimeZone/>** — устанавливает часовой пояс, используемый для форматирования;

<fmt:formatNumber/>, **<fmt:formatDate/>** — форматирует числа/даты с учётом установленной локали (региональных установок) либо указанного шаблона;

<fmt:parseNumber/>, **<fmt:parseDate/>** — переводит строковое представление числа/даты в объекты подклассов **Number** / **Date**.

Ниже приведены три примера на использование тегов из группы **fmt**.

Документ **formatdatejsp.jspx** выводит на экран текущую дату и время с учётом установленного объекта класса **Locale**.

```
<!--пример # 12 : вывод даты и времени :  formatdate.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
xmlns:fmt="http://java.sun.com/jsp/jstl/fmt" version="2.0">
<jsp:directive.page contentType=
      "text/html; charset=Utf-8"/>
<html><head><title>Формат даты</title></head>
<body>
<jsp:useBean id="now" class="java.util.Date" />
<fmt:setLocale value="en-EN"/>
<jsp:text>Вывод даты в формате English</jsp:text><br/>
Сегодня: <fmt:formatDate value="${now}" /><br/>
<fmt:setLocale value="ru-RU"/>
<jsp:text>Вывод даты в формате Russian</jsp:text><br/>
Сегодня: <fmt:formatDate value="${now}" /><br/>

Время(стиль-short): <fmt:formatDate value="${now}"
type="time" timeStyle="short" /><br/>
Время(стиль-medium): <fmt:formatDate value="${now}"
type="time" timeStyle="medium" /><br/>
Время(стиль-long): <fmt:formatDate value="${now}"
type="time" timeStyle="long" /><br/>
Время(стиль-full): <fmt:formatDate value="${now}"
type="time" timeStyle="full" /><br/>
</body></html>
</jsp:root>
```

В результате работы документа в браузер будет выведено:
Вывод даты в формате English
Сегодня: Aug 14, 2007
Вывод даты в формате Russian
Сегодня: 14.08.2007
Время (стиль-short): 23:23
Время (стиль-medium): 23:23:02
Время (стиль-long): 23:23:02 EEST
Время (стиль-full): 23:23:02 EEST

В следующем примере реализован ещё один способ вывода времени и даты

```
<!--пример # 13 : полный вывод даты и времени : timezone.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:fmt="http://java.sun.com/jsp/jstl/fmt"
      version="2.0">
<jsp:directive.page contentType=
      "text/html; charset=Utf-8"/>
<html><head><title>timezone</title></head>
<body>
<jsp:useBean id="now" class="java.util.Date" />
<jsp:text>
Вывод даты и времени с помощью тега<br/> fmt:formatDate
и установки TimeZone
</jsp:text><br/>
<fmt:setLocale value="ru-RU"/>
<fmt:timeZone value="GMT+4:00">
<fmt:formatDate value="${now}" type="both"
                dateStyle="full" timeStyle="full"/><br/>
</fmt:timeZone>
</body></html>
</jsp:root>
```

В результате работы документа в браузер будет выведено:
Вывод даты и времени с помощью тега
fmt:formatDate и установки TimeZone
15 Август 2007 г. 0:26:38 GMT+04:00

Страница **formatnumber.jspx** выводит формат числа в соответствии с установленными региональными установками.

```
<!--пример # 14 : формат чисел :  formatnumber.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
xmlns:fmt="http://java.sun.com/jsp/jstl/fmt" version="2.0">
<jsp:directive.page contentType=
      "text/html; charset=Utf-8"/>
<html><head><title>fomatnumber</title></head>
<body>
Вывод формата числа 9876543.21: <br/>
<jsp:text>Обычный формат - </jsp:text>
<fmt:formatNumber value="9876543.21" /><br/>
<jsp:text>Процентный формат - </jsp:text>
```

```
<fmt:formatNumber value="9876543.21" type="percent"/><br/>
<fmt:setLocale value="ru-RU"/>
<jsp:text>Русская валюта - </jsp:text>
<fmt:formatNumber value="9876543.21" type="currency"/><br/>
<fmt:setLocale value="en-EN"/>
<jsp:text>Английская валюта - </jsp:text>
<fmt:formatNumber value="9876543.21" type="currency"/><br/>
<jsp:text>Французская валюта - </jsp:text>
<fmt:setLocale value="fr-FR"/>
<fmt:formatNumber value="9876543.21" type="currency"/><br/>
</body></html>
</jsp:root>
```

В результате работы документа в браузер будет выведено:

Вывод формата числа 9876543.21:
Обычный формат - 9 876 543,21
Процентный формат - 987 654 321%
Русская валюта - 9 876 543,21 руб.
Английская валюта - ¤9,876,543.21
Французская валюта - 9 876 543,21 €

JSTL sql

Используется для выполнения запросов SQL непосредственно из JSP и обработки результатов запроса в JSP.

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

— для обычной страницы JSP;

<jsp:root version="1.2" xmlns:sql=

"http://java.sun.com/jstl/sql">...</jsp:root> — для JSP-документа.

Теги:

<sql:dateParam> – определяет параметры даты для **<sql:query>** либо **<sql:update>**;

<sql:param> – определяет параметры **<sql:query>** либо **<sql:update>**;

<sql:query> – выполняет запрос к БД;

<sql:setDataSource> – усанавливает data source для **<sql:query>**, **<sql:update>**, и **<sql:transaction>** тегов;

<sql:transaction> – объединяет внутренние теги **<sql:query>** и **<sql:update>** в одну транзакцию;

<sql:update> - выполняет преобразование БД.

В промышленном программировании данная библиотека не используется из-за прямого доступа из JSP в СУБД, что является явным нарушением шаблона MVC.

JSTL xml

Используется для обработки данных XML в JSP-документе.

<% @taglib uri="http://java.sun.com/jstl/xml" prefix="x" %>

— для обычной JSP-страницы,

<jsp:root version="1.2" xmlns:x=

"http://java.sun.com/jstl/xml">...</jsp:root> — для XML формата JSP.

Теги:
<x:forEach> – XML-версия тега **<c:choose>**;
<x:choose> – XML-версия тега **<c:forEach>**;
<x:if> – XML-версия тега **<c:if>**;
<x:otherwise> – XML-версия тега **<c:otherwise>**;
<x:out> – XML-версия тега **<c:out>**;
<x:param> – XML-версия тега **<c:param>**, определяющая параметры для другого тега **<x:transform>**;
<x:parse> – разбор XML-документа;
<x:set> – XML-версия тега **<c:set>**;
<x:transform> – трансформация XML-документа;
<x:when> – XML-версия тега **<c:when>**;
<x:choose> – XML-версия тега **<c:choose>**;
<x:forEach> – XML-версия тега **<c:forEach>**.

Включение ресурсов

В реальных проектах JSP-страницы часто состоят из статических элементов. Для этого используется директива **include**, а файл, содержащий необходимый статичный элемент, сохраняется с расширением **.jspf**, что означает «фрагмент JSP». При необходимости включения содержимого в JSP-страницу каждый раз, когда та получает запрос, используется стандартное действие **jsp:include**. В этом случае включаемые сегменты имеют доступ к объектам **request**, **session** и **application** исходной страницы и ко всем атрибутам, которые имеют эти объекты. Если использовать директиву **include**, то изменения включаемого сегмента отразятся только после изменения исходной страницы (контейнер JSP перекомпилирует исходную страницу). Для включения содержимого в JSP-документ также используется стандартное действие **jsp:include**. При этом не обязательно, чтобы включаемый JSP-фрагмент был правильным XML-документом. Главное, чтобы он возвращал текст в виде правильного XML и не нарушал структуру исходного JSP-документа.

```
<!--пример #15 : включение в код статического содержимого :  incl_title.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:c="http://java.sun.com/jsp/jstl/core"
          version="2.0">
<jsp:directive.page contentType=
          "text/html; charset=Utf-8" />
<html><head>
<jsp:directive.include file="\WEB-INF\jspf\title.jspf" />
</head>
<body>
<h1>JSP-страница, использующая директиву include</h1>
<h3>Директива include используется для включения статиче-
ского содержимого, например заголовка страницы.</h3>
</body></html>
</jsp:root>
```

```
<!-- пример # 16 : код включаемого фрагмента : title.jspf -->
<title>Title from title.jspf</title>
```

Ниже приведен пример включения динамического содержимого. Включаемый фрагмент получает данные из объектов **request** и **session**. Для передачи значения параметра можно использовать строку запроса. Запрос может выглядеть следующим образом:

http://localhost:8082/home/thanks.jspx?lname=username.

Установка кодировки в фрагменте необходима для того, чтобы устранить неполадки при включении русского текста.

```
<!-- пример # 17 : использование действия include для динамического включения :
thanks.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
version="2.0">
<jsp:directive.page contentType="text/html;charset=Utf-8"/>
<html><head><title>Действие include</title></head>
<body>
      Данные, сформированные динамически при включении jsp-
фрагмента<br/>
      Включаемый фрагмент получает данные из объекта
session <br/>
      <jsp:include page="\WEB-INF\jspf\params.jsp"/>.
</body></html>
</jsp:root>
```

```
<!-- пример #18 : включаемый фрагмент : params.jsp -->
<jsp:directive.page contentType="text/html;charset=Utf-8"/>
ID сессии -
    <jsp:expression>session.getId()</jsp:expression>
```

В результате работы документа в браузер будет выведено:

Данные, сформированные динамически при включении jsp-фрагмента.
Включаемый фрагмент получает данные из объектов request, session
ID сессии - 08C51EEC60A97E90C734101F54EA310E .

Также для включения содержимого можно использовать тег **<c:import>**. Его использование уже было приведено выше.

Обработка ошибок

При выполнении web-приложений, как и любых других, могут возникать ошибки и исключительные ситуации. Три основных типа исключительных ситуаций:

- код «404 Not Found». Возникает при неправильном наборе адреса или обращении к станице, которой не существует;
- код «500 Internal Server Error». Возникает при вызове сервлетом метода **sendError(500)** для объекта **HttpServletResponse**;
- исключения времени исполнения. Исключения, генерируемые web-приложением и не перехватываемые фильтром, сервлетом или JSP.

Для обработки исключений в зависимости от типа в приложении может существовать несколько JSP-страниц, сервлетов или обычных HTML-страниц. Для настройки соответствия ошибок и обработчиков используется элемент **error-page** файла **web.xml**. Например:

```
<error-page>
   <error-code>404</error-code>
   <location>/error404</location>
</error-page>
```

или

```
<error-page>
   <exception-type>java.io.IOException</exception-type>
   <location>/errorIo</location>
</error-page>
```

В элементе **error-code** указывается код ошибки, в элементе **exception-type** – тип исключения.

Для указания страницы, обрабатывающей ошибки, возникающие при выполнении текущей страницы, можно использовать директиву

<jsp:directive.page errorPage="path" />, где **path** – эту путь к странице-обработчику. Ниже приведен пример, использующий именно такой способ. При нажатии на кнопку генерируется ошибка **java.lang.NullPointerException**, и управление передается странице **error_hand.jsp**

```
<!--пример # 19 : генерация ошибки : gen_error.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
            xmlns:c="http://java.sun.com/jsp/jstl/core"
            version="2.0">
<jsp:directive.page contentType=
            "text/html; charset=Utf-8"/>
<jsp:directive.page errorPage="/error_hand.jspx" />
<html><head><title>Генерация исключения</title></head>
<body>
<h2>При нажатии кнопки будет сгенерирована ошибка!</h2>
<form>
<input type="submit" name="gen"
       value="Сгенерировать ошибку"/>
</form>
<c:if test="${not empty param.gen}">
 <jsp:declaration>String str;</jsp:declaration>
 <jsp:scriptlet>str.length();</jsp:scriptlet>
</c:if>
</body></html>
</jsp:root>
```

Страница, вызываемая при ошибках, может иметь статический вид, но при необходимости сообщает о типе и месте возникшего исключения в понятной для клиента приложения форме.

```
<!--пример # 20 : ERROR PAGE : error_hand.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
            version="2.0">
```

```
<jsp:directive.page contentType=
            "text/html; charset=Utf-8" />
<jsp:directive.page isErrorPage="true" />
<html><head><title>Сообщение об ошибке</title></head>
<body>
<p>Сгенерирована ошибка! <br/>
<jsp:expression>exception.toString()</jsp:expression>
</p></body></html>
</jsp:root>
```

Извлечение значений полей

Библиотеки JSLT и EL позволяют легко обрабатывать данные, полученные из форм, так как JSP-страница имеет доступ к неявному объекту **param**, который состоит из объектов типа **java.util.Map.Entry**, что позволяет обращаться к данным как к парам «ключ-значение».

В следующем примере в документе **params.jspx** производится извлечение значений параметров, передаваемых из страницы **form.jspx**.

```
<!--пример # 21 : страница, которая выводит форму и передает данные
странице param.jspx: form.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
            xmlns:c="http://java.sun.com/jsp/jstl/core"
            version="2.0">
<jsp:directive.page contentType=
            "text/html; charset=Utf-8" />
<html><head><title>Форма для заполнения</title></head>
<body>
<form action="params.jspx">
Введите, пожалуйста, ваши данные: <br/>
Фамилия: <input type="text" name="fname" value="" /><br/>
Имя:     <input type="text" name="lname" value="" /><br/>
E-mail:  <input type="text" name="e-mail" value="" /><br/>
<input type="submit" value="Отправить" /><br/>
</form>
</body></html>
</jsp:root>
```

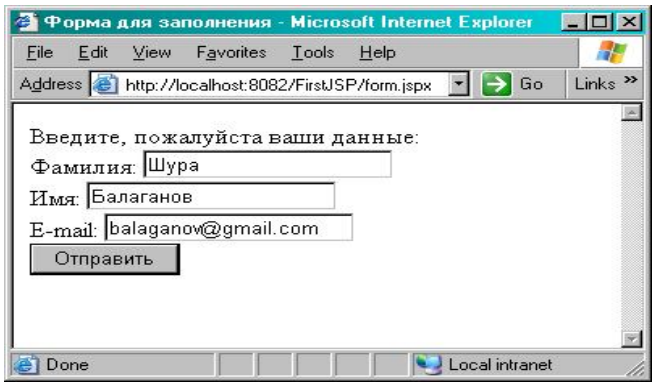

Рис. 19.1. Документ для задания и передачи параметров

```
<!--пример # 22 : считывание информации и генерация ответа : params.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:c="http://java.sun.com/jsp/jstl/core"
          version="2.0">
<jsp:directive.page contentType=
          "text/html; charset=Utf-8" />
<html><head><title>Обработка данных</title></head>
<body>
Вывод данных с помощью JSTL и EL<br/>
<c:forEach var="items" items="${param}">
<b><c:out value="${items.key}"></c:out></b>:
<c:out value="${items.value}"></c:out><br/>
</c:forEach>
<c:if test="${not empty param.fname}">
<b>Имя:</b><c:out value="${param.fname}"/><br/>
</c:if>
<c:if test="${not empty param.lname}">
<b>Фамилия:</b><c:out value="${param.lname}"/>
</c:if>
</body></html>
</jsp:root>
```

В результате работы документа в браузер будет выведено:

Вывод данных с помощью JSTL и EL
lname: Балаганов
fname: Шура
e-mail: balaganov@gmail.com
Имя: Шура
Фамилия: Балаганов

В вышеприведенном примере с помощью тега **c:forEach** перебираются все данные, полученные из формы. Так же можно выводить отдельные параметры, обращаясь к ним с помощью EL. Конструкция **${param.lname}** возвращает значение параметра **lname**.

С помощью тега **jsp:forward** можно добавлять данные к запросу.

```
<!--пример # 23: добавление параметра add_param и перенаправление запроса
к странице form.jspx: forward.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          version="2.0">
<jsp:directive.page contentType=
          "text/html; charset=Utf-8" />
<html><head><title>Добавление параметра</title></head>
<body>
<jsp:forward page="params.jspx">
<jsp:param name="addparam" value="added"/>
</jsp:forward>
</form>
</body></html>
</jsp:root>
```

Если обратиться к этой странице, передавая в строке запроса параметры (например **`http://localhost:8082/FirstJSP/forward.jspx?name=UserName`**), то, кроме этих параметров, странице **`param.jspx`** будет передан параметр **`addparam`** со значением **`added`**.

Технология взаимодействия JSP и сервлета

В большинстве приложений используются не сервлеты или JSP, а их сочетание. В JSP представляется, как будут выглядеть результаты запроса, а сервлет отвечает за вызов классов бизнес-логики и передачу результатов выполнения бизнес-логики в соответствующие JSP и их вызов. Т.е. сервлеты не генерируют ответа сами, а только выступают в роли контроллера запросов. Такая архитектура построения приложений носит название MVC (Model/View/Controller). Model – классы бизнес-логики и длительного хранения, View – страницы JSP, Controller – сервлет.

Реализацию достаточно простой, но эффективной технологии построения распределенного приложения можно рассмотреть на примере решения задачи проверки логина и пароля пользователя с выводом приветствия в случае положительного результата. Схематично организацию данного приложения можно представить в виде следующей диаграммы:

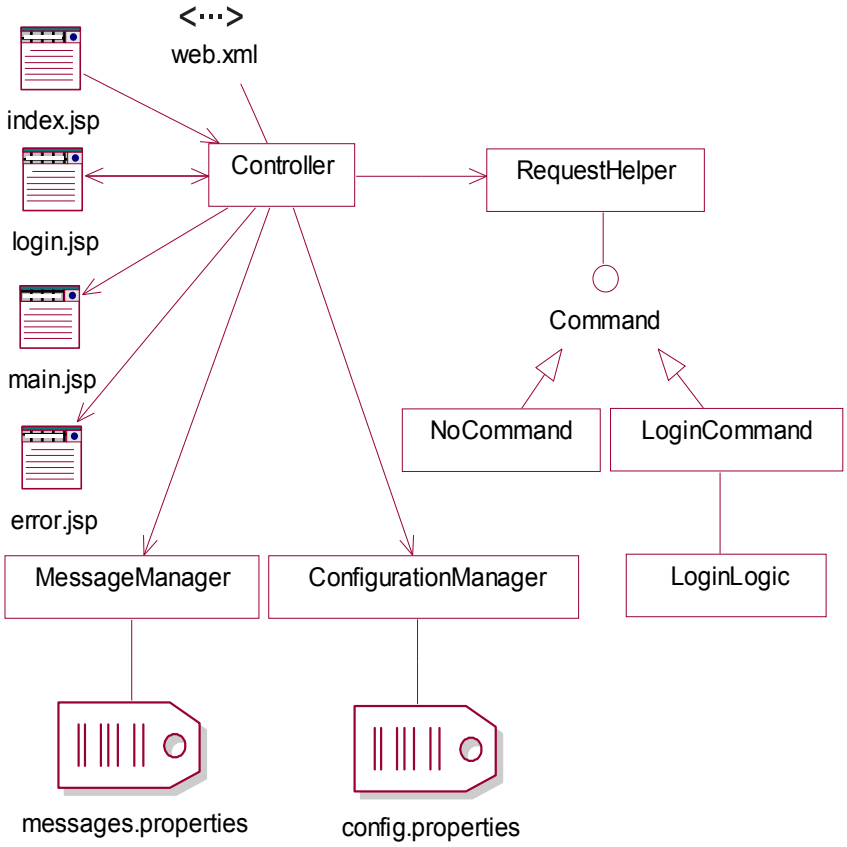

Рис. 19.2. Диаграмма взаимодействия классов и страниц JSP приложения.

```
<!--пример # 24 : прямой вызов контроллера : index.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
version="2.0">
<jsp:directive.page contentType="text/html; charset=Utf-8"
/>
<html><head><title>Index JSP</title></head>
<body>
<a href="controller">Main Controller</a>
</body></html>
</jsp:root>
```

Следующая страница **login.jsp** содержит форму для ввода логина и пароля для аутентификации в системе:

```
<!--пример # 25 : форма ввода информации и вызов контроллера : login.jsp -->
<%@ page language="java" contentType="text/html;
  charset=ISO-8859-1"  pageEncoding="ISO-8859-1"%>
    <html><head><title>Login</title></head>
    <body><h3>Login</h3>
    <hr/>
    <form name="loginForm" method="POST"
    action="controller">
    <input type="hidden" name="command" value="login" />
        Login:<br/>
        <input type="text" name="login" value=""><br/>
        Password:<br/>
        <input type="password" name="password" value="">
      <br/>
        <input type="submit" value="Enter">
    </form><hr/>
    </body></html>
```

Код сервлета-контроллера **Controller**:

```
/* пример # 26 : контроллер запросов : Controller.java */
package by.bsu.famcs.jspservlet;
import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import by.bsu.famcs.jspservlet.commands.Command;
import by.bsu.famcs.jspservlet.manager.MessageManager;
import
by.bsu.famcs.jspservlet.manager.ConfigurationManager;

public class Controller extends HttpServlet
            implements  javax.servlet.Servlet {
//объект, содержащий список возможных команд
      RequestHelper requestHelper =
            RequestHelper.getInstance();
```

```
      public Controller() {
            super();
      }
      protected void doGet(HttpServletRequest request,
                  HttpServletResponse response)
            throws ServletException, IOException{
            processRequest(request, response);
      }
      protected void doPost(HttpServletRequest request,
                  HttpServletResponse response)
            throws ServletException, IOException{
            processRequest(request, response);
      }
      private void processRequest(HttpServletRequest
            request, HttpServletResponse response)
      throws ServletException, IOException {

            String page = null;
            try {
//определение команды, пришедшей из JSP
                  Command command =
                        requestHelper.getCommand(request);
/*вызов реализованного метода execute() интерфейса Command и передача
параметров классу-обработчику конкретной команды*/
            page = command.execute(request, response);
// метод возвращает страницу ответа
            } catch (ServletException e) {
                  e.printStackTrace();
//генерация сообщения об ошибке
request.setAttribute("errorMessage",
      MessageManager.getInstance().getProperty(
      MessageManager.SERVLET_EXCEPTION_ERROR_MESSAGE));
//вызов JSP-страницы с сообщением об ошибке
page = ConfigurationManager.getInstance()
      .getProperty(ConfigurationManager.ERROR_PAGE_PATH);
            } catch (IOException e) {
                  e.printStackTrace();
                  request.setAttribute("errorMessage",
 MessageManager.getInstance()
.getProperty(MessageManager.IO_EXCEPTION_ERROR_MESSAGE));

            page = ConfigurationManager.getInstance()
.getProperty(ConfigurationManager.ERROR_PAGE_PATH);
            }
//вызов страницы ответа на запрос
            RequestDispatcher dispatcher =
getServletContext().getRequestDispatcher(page);
            dispatcher.forward(request, response);
      }
}
```

```
/* пример # 27 : класс контейнер команд : RequestHelper.java */
package by.bsu.famcs.jspservlet;
import java.util.HashMap;
import javax.servlet.http.HttpServletRequest;
import by.bsu.famcs.jspservlet.commands.Command;
import by.bsu.famcs.jspservlet.commands.LoginCommand;
import by.bsu.famcs.jspservlet.commands.NoCommand;

public class RequestHelper {
      private static RequestHelper instance = null;

      HashMap<String, Command> commands =
                  new HashMap<String, Command>();

      private RequestHelper() {
//заполнение таблицы командами
            commands.put("login", new LoginCommand());
      }
   public Command getCommand(HttpServletRequest request) {
//извлечение команды из запроса
      String action = request.getParameter("command");
//получение объекта, соответствующего команде
      Command command = commands.get(action);
            if (command == null) {
//если команды не существует в текущем объекте
                  command = new NoCommand();
            }
            return command;
      }
//создание единственного объекта по шаблону Singleton
   public static RequestHelper getInstance() {
            if (instance == null) {
                  instance = new RequestHelper();
            }
            return instance;
      }
}
/* пример # 28 : интерфейс, определяющий контракт и его реализации :
Command.java: LoginCommand.java : NoCommand.java */
package by.bsu.famcs.jspservlet.commands;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import java.io.IOException;

public interface Command {
      public String execute(HttpServletRequest request,
                              HttpServletResponse response)
                  throws ServletException, IOException;
}
```

```
package by.bsu.famcs.jspservlet.commands;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import by.bsu.famcs.jspservlet.logic.LoginLogic;
import
by.bsu.famcs.jspservlet.manager.ConfigurationManager;
import by.bsu.famcs.jspservlet.manager.MessageManager;

public class LoginCommand implements Command {

private static final String PARAM_NAME_LOGIN = "login";
private static final String PARAM_NAME_PASSWORD
                                        = "password";

      public String execute(HttpServletRequest request,
                              HttpServletResponse response)
                  throws ServletException, IOException {

            String page = null;
//извлечение из запроса логина и пароля
String login = request.getParameter(PARAM_NAME_LOGIN);
String pass = request.getParameter(PARAM_NAME_PASSWORD);
//проверка логина и пароля
            if (LoginLogic.checkLogin(login, pass)) {
                  request.setAttribute("user", login);
//определение пути к main.jsp
                  page = ConfigurationManager.getInstance()
.getProperty(ConfigurationManager.MAIN_PAGE_PATH);
            } else {
                  request.setAttribute("errorMessage",
 MessageManager.getInstance()
.getProperty(MessageManager.LOGIN_ERROR_MESSAGE));
                  page = ConfigurationManager.getInstance()
.getProperty(ConfigurationManager.ERROR_PAGE_PATH);
            }
            return page;
      }
}
package by.bsu.famcs.jspservlet.commands;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import
by.bsu.famcs.jspservlet.manager.ConfigurationManager;
```

```
public class NoCommand implements Command {

      public String execute(HttpServletRequest request,
             HttpServletResponse response)
             throws ServletException, IOException {
/*в случае прямого обращения к контроллеру переадресация на страницу ввода
логина*/
      String page = ConfigurationManager.getInstance()
.getProperty(ConfigurationManager.LOGIN_PAGE_PATH);
            return page;
      }
}
/* пример # 29 : служебные классы, извлекающие из properties-файлов
необходимую для функционирования приложения информацию :
ConfigurationManager.java: MessageManager.java */
package by.bsu.famcs.jspservlet.manager;
import java.util.ResourceBundle;

public class ConfigurationManager {
      private static ConfigurationManager instance;
      private ResourceBundle resourceBundle;

//класс извлекает информацию из файла config.properties
      private static final String BUNDLE_NAME = "config";

      public static final String DATABASE_DRIVER_NAME =
                                    "DATABASE_DRIVER_NAME";
      public static final String DATABASE_URL =
                                          "DATABASE_URL";
      public static final String ERROR_PAGE_PATH =
                                    "ERROR_PAGE_PATH";
      public static final String LOGIN_PAGE_PATH =
                                    "LOGIN_PAGE_PATH";
      public static final String MAIN_PAGE_PATH =
                                    "MAIN_PAGE_PATH";

      public static ConfigurationManager getInstance() {
            if (instance == null) {
                  instance = new ConfigurationManager();
                  instance.resourceBundle =
                  ResourceBundle.getBundle(BUNDLE_NAME);
            }
            return instance;
      }
      public String getProperty(String key) {
            return (String)resourceBundle.getObject(key);
      }
}
```

```
package by.bsu.famcs.jspservlet.manager;
import java.util.ResourceBundle;
public class MessageManager {
      private static MessageManager instance;
      private ResourceBundle resourceBundle;
//класс извлекает информацию из файла messages. properties
      private static final String BUNDLE_NAME = "messages";
      public static final String LOGIN_ERROR_MESSAGE = "LO-
GIN_ERROR_MESSAGE";
      public static final String SERV-
LET_EXCEPTION_ERROR_MESSAGE =
            "SERVLET_EXCEPTION_ERROR_MESSAGE";
      public static final String IO_EXCEPTION_ERROR_MESSAGE
= "IO_EXCEPTION_ERROR_MESSAGE";

      public static MessageManager getInstance() {
            if (instance == null) {
                  instance = new MessageManager();
                  instance.resourceBundle =
                  ResourceBundle.getBundle(BUNDLE_NAME);
            }
            return instance;
      }

      public String getProperty(String key) {
            return (String)resourceBundle.getObject(key);
      }
}
```

Далее приведено содержимое файла *config.properties:*

```
################################
## Application configuration ##
################################
DATABASE_DRIVER_NAME=com.mysql.jdbc.Driver
DATABASE_URL=jdbc:mysql://localhost:3306/db1?user=
root&password=root
ERROR_PAGE_PATH=/jsp/error.jspx
LOGIN_PAGE_PATH=/jsp/login.jspx
MAIN_PAGE_PATH=/jsp/main.jspx
```

Далее приведено содержимое файла *messages.properties:*

```
################################
##         Messages      ##
################################
LOGIN_ERROR_MESSAGE=Incorrect login or password
SERVLET_EXCEPTION_ERROR_MESSAGE=ServletException: Servlet
encounters difficulty
IO_EXCEPTION_ERROR_MESSAGE=IOException: input or output er-
ror while handling the request
```

Ниже приведен код класса бизнес-логики **LoginLogic**, выполняющий проверку правильности введенных логина и пароля с помощью запроса в БД:

```
/* пример # 30 : бизнес-класс проверки данных пользователя : LoginLogic.java */
package by.bsu.famcs.jspservlet.logic;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.DriverManager;
import
by.bsu.famcs.jspservlet.manager.ConfigurationManager;

public class LoginLogic {
      public static boolean checkLogin(
            String login, String password) {
            // проверка логина и пароля
            try {
//организация простейшего соединения с базой данных
String driver = ConfigurationManager.getInstance()
.getProperty(ConfigurationManager.DATABASE_DRIVER_NAME);

                  Class.forName(driver);
                  Connection cn = null;
                  try {
String url = ConfigurationManager.getInstance()
.getProperty(ConfigurationManager.DATABASE_URL);
            cn = DriverManager.getConnection(url);
                        PreparedStatement st = null;
                        try {
                  st = cn.prepareStatement(
"SELECT * FROM USERS WHERE LOGIN = ? AND PASSWORD = ?");
                              st.setString(1, login);
                              st.setString(2, password);
                              ResultSet rs = null;
                              try {
                                    rs = st.executeQuery();
      /* проверка, существует ли пользователь
       с указанным логином и паролем */
                                    return rs.next();
                              } finally {
                                    if (rs != null)
                                          rs.close();
                              }
                        } finally {
                              if (st != null)
                                    st.close();
                        }
                  } finally {
```

```
                    if (cn != null)
                        cn.close();
                }
            } catch (SQLException e) {
                e.printStackTrace();
                return false;
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
                return false;
            }
        }
}
```

Страница **main.jsp** показывается пользователю в случае успешной аутентификации в приложении:

```
<!--пример # 31 : сообщение о входе : main.jsp -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:c=http://java.sun.com/jsp/jstl/core
      version="2.0">
<jsp:directive.page contentType="text/html;
      charset=Utf-8" />
      <html><head><title>Welcome</title></head>
      <body><h3>Welcome</h3>
      <hr />
      <c:out value="${user}, Hello!"/>
      <hr />
      <a href="controller">Return to login page</a>
      </body></html>
</jsp:root>
```

Страница **error.jsp** загружается пользователю в случае возникновения ошибок (например, если неправильно введены логин и пароль):

```
<!-- пример # 32 : страница ошибок, предлагающая повторить процедуру ввода
информации : error.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:c=
      "http://java.sun.com/jsp/jstl/core" version="2.0">
      <jsp:directive.page contentType=
                  "text/html; charset=Utf-8" />
      <html><head><title>Error</title></head>
      <body>
      <h3>Error</h3>
      <hr />
      <jsp:expression>
(request.getAttribute("errorMessage") != null)
? (String) request.getAttribute("errorMessage")
: "unknown error"</jsp:expression>
      <hr />
      <a href="controller">Return to login page</a>
```

```
      </body></html>
</jsp:root>
```

И последнее, что надо сделать в приложении, – это настроить файл **web.xml**, чтобы можно было обращаться к сервлету-контроллеру по имени **controller**, т.е. необходимо настроить mapping.

```
<!--пример # 33 : имя сервлета и путь к нему : web.xml -->
   <?xml version="1.0" encoding="UTF-8"?>
   <web-app id="WebApp_ID" version="2.4" >
      <display-name>Project</display-name>
      <servlet>
            <description>
            </description>
            <display-name>
            Controller</display-name>
            <servlet-name>Controller</servlet-name>
            <servlet-class>
      by.bsu.famcs.jspservlet.Controller</servlet-class>
      </servlet>
      <servlet-mapping>
            <servlet-name>Controller</servlet-name>
            <url-pattern>/controller</url-pattern>
      </servlet-mapping>
      <welcome-file-list>
            <welcome-file>index.jsp</welcome-file>
      </welcome-file-list>
   </web-app>
```

В данном случае в поле **<servlet-name>** было занесено имя **controller,** а в поле **<url-pattern>** – соответственно **/controller**.

Запуск примера производится из командной строки Web-браузера при запущенном контейнере сервлетов Tomcat 5.5.*, например в виде:

http://localhost:8082/Project/index.jspx

В этом случае при вызове сервлета в браузере будет отображен путь и имя в виде:

http://localhost:8082/Project/controller

Задания к главе 19

Вариант А

Реализовать приложение, используя технологию взаимодействия JSP и сервлетов. Вся информация должна храниться в базе данных.

1. **Банк**. Осуществить перевод денег с одного счета на другой с указанием реквизитов: Банк, Номер счета, Тип счета, Сумма. Таблицы должны находиться в различных базах данных. Подтверждение о выполнении операции должно выводиться в JSP с указанием суммы и времени перевода.
2. **Регистрация пользователя**. Должны быть заполнены поля: Имя, Фамилия, Дата рождения, Телефон, Город, Адрес. Система должна при-

сваивать уникальный ID, генерируя его. При регистрации пользователя должна производиться проверка на несовпадение ID. Результаты регистрации с датой регистрации должны выводиться в JSP. При совпадении имени и фамилии регистрация не должна производиться.

3. **Телефонный справочник**. Таблица должна содержать Фамилию, Адрес, Номер телефона. Поиск должен производиться по части фамилии или по части номера. Результаты должны выводиться вместе с датой выполнения в JSP.
4. **Склад**. Заполняются поля Товар и Количество. Система выводит промежуточную информацию о существующем количестве товара и запрашивает подтверждение на добавление. При отсутствии такого товара на складе добавляется новая запись.
5. **Словарь**. Ввод слова. Системой производится поиск одного или нескольких совпадений и осуществляется вывод результатов в JSP. Перевод может осуществляться в обе стороны. Одному слову может соответствовать несколько значений.
6. **Каталог библиотеки**. Выдается список книг, наличествующих в библиотеке. Запрос на заказ отправляется пометкой требуемой книги. Система проверяет в БД, свободна книга или нет. В случае занятости выводится информация о сроках возвращения книги в фонд.
7. **Голосование**. Выводится вопрос и варианты ответа. Пользователь имеет возможность проголосовать и просмотреть результаты голосования по данному вопросу. БД должна хранить дату и время каждого голосования и выводить при необходимости соответствующую статистику по датам подачи голоса.

Вариант В

Для заданий варианта В главы 4 создать информационную систему, использующую страницы JSP на стороне клиента, сервлет в качестве контроллера и БД для хранения информации.

Тестовые задания к главе 19

Вопрос 19.1.

Как правильно объявить и проинициализировать переменную **j** типа **int** в тексте JSP?

1) `<%! int j = 1 %>;`
2) `<%@ int j = 2 %>;`
3) `<%! int j = 3; %>;`
4) `<%= int j = 4 %>;`
5) `<%= int j = 5; %>.`

Вопрос 19.2.

Какие из перечисленных переменных можно использовать в выражениях и скриптлетах JSP без предварительного объявления?

1) `error;`
2) `page;`

3) `this;`
4) `exception;`
5) `context.`

Вопрос 19.3.

Какой из следующих интерфейсов объявляет метод **`_jspService()`**?

1) `javax.servlet.jsp.Jsp;`
2) `javax.servlet.jsp.JspServlet;`
3) `javax.servlet.jsp.JspPage;`
4) `javax.servlet.jsp.HttpJspPage;`
5) `javax.servlet.jsp.HttpJspServlet.`

Вопрос 19.4.

Тег **`jsp:useBean`** объявлен как

```
<jsp:useBean   id="appJsp"
               class="main.ApplicationJSP"
               scope="application" />
```

В объекте какого типа должен быть сохранен созданный экземпляр?

1) `ServletConfig;`
2) `HttpApplication;`
3) `ServletContext;`
4) `ServletConfig;`
5) `ApplicationContext.`

Вопрос 19.5.

Какой тег JSP используется для извлечения значения поля экземпляра JavaBean в виде строки?

1) `jsp:useBean.toString;`
2) `jsp:param.property;`
3) `jsp:propertyType;`
4) `jsp:getProperty;`
5) `jsp:propertyToString;`