

Учреждение образования  
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ  
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

**В. В. Смелов**

# **ОСНОВЫ WEB-ПРОГРАММИРОВАНИЯ НА JAVA**

*Рекомендовано  
учебно-методическим объединением высших учебных заведений  
Республики Беларусь по образованию в области информатики  
и радиоэлектроники в качестве учебно-методического пособия  
для студентов учреждений, обеспечивающих получение  
высшего образования по направлению специальности  
«Информационные системы и технологии  
(издательско-полиграфический комплекс)»*

Минск 2009

УДК 004.434:004.738.5(075.8)

ББК 22.18я7

C50

Рецензенты:

кафедра управления информационными ресурсами  
Академии управления при Президенте Республики Беларусь  
(кандидат технических наук, заведующий кафедрой *В. И. Новиков*);  
кандидат технических наук, доцент кафедры информационных  
технологий автоматизированных систем Белорусского  
государственного университета информатики  
и радиоэлектроники *О. В. Герман*

*Все права на данное издание защищены. Воспроизведение всей книги или ее части не может быть осуществлено без разрешения учреждения образования «Белорусский государственный технологический университет».*

**Смелов, В. В.**

C50 Основы web-программирования на Java : учеб.-метод. пособие для студентов по направлению специальности «Информационные системы и технологии (издательско-полиграфический комплекс)». – Минск : БГТУ, 2009. – 140 с.

ISBN 978-985-434-871-1

Основное внимание в учебно-методическом пособии уделено практическому применению технологий Java Servlet и JSP для обработки HTTP-запросов. Кроме того, рассматриваются технологии JDBC и JNDI. Изложение сопровождается многочисленными схемами и примерами. Приведенные примеры ориентированы на применение в среде web-сервера Apache Tomcat. Заключительная глава пособия представляет собой практикум, который может быть использован для закрепления изученного материала. Предполагается, что читатель имеет навыки программирования на языке Java, опыт применения HTML, JavaScript, XML, SQL, а также понимает принципы работы компьютерных сетей.

УДК 004.434:004.738.5(075.8)

ББК 22.18я7

ISBN 978-985-434-871-1

© УО «Белорусский государственный  
технологический университет», 2009

© Смелов В. В., 2009

## ПРЕДИСЛОВИЕ

Язык программирования Java завоевал серьезную популярность только в начале XXI в., хотя уже был известен в середине 90-х гг. XX в. Как пишет один из его создателей Патрик Ноутон [1], первоначально предполагалось создать независимую от платформы систему программирования для встроенных систем в электронные устройства типа микроволновых печей, телевизоров, пультов управления и т. п. Но потом стало понятно, что проблемы разработки мобильного программного обеспечения для контроллеров и для Internet являются схожими.

Первой и основной составляющей успеха языка Java является его нейтральность относительно программно-аппаратной архитектуры компьютера. Появление такого инструмента делает прозрачными границы между основными конкурирующими операционными средами Microsoft, UNIX, IBM, Macintosh. Наиболее ощутимо это преимущество при разработке распределенных в сети Internet приложений, ведь Internet представляет собой объединение компьютеров с различной архитектурой и операционными системами. Поэтому именно Internet выдвинул Java на передние рубежи программирования.

Второй составляющей успеха системы программирования Java является его подобие с C++. Разработчики намеренно наделили Java практически одинаковым с языком C++ синтаксисом. Помимо того, что сама система программирования C++ является успешной, хорошо продуманной и развитой объектно-ориентированной технологией, с ней работает огромное количество опытных программистов. Освоение ими Java-программирования происходит быстро и без особых сложностей.

Следует, однако, отметить, что Java не заменяет C++, обе системы программирования будут сосуществовать, т. к. они предназначены для решения разных задач.

На момент подготовки к изданию книги, по оценкам Sun Microsystems Inc. [8], сообщество Java-программистов насчитывает более 4 млн. человек и общий объем доходов от продуктов Java составляет 110 млрд. долл., а с учетом сопутствующих продуктов этот доход удваивается.

Лицензиатами Java-технологий являются ведущие компании-производители программного обеспечения: IBM, Oracle Corp., Hewlett-Packard, Borland Corp., SAP и др.

Эта книга посвящена только одному направлению программирования на Java – разработке web-приложений. Задача, которую ставит перед собой автор, – научить читателя разрабатывать простые web-приложения и познакомить с основными технологиями и приемами программирования, необходимыми для достижения этой цели.

Предполагается, что читатель знаком с основами программирования на языке Java в объеме источника [1]. Кроме того, потребуются знания языков разметки HTML [2], XML [3], а также языка запросов к реляционным базам данных SQL [4]. Примеры web-приложений ориентированы на работу в среде web-сервера Apache Tomcat [5], некоторые из них содержат код JavaScript [6].

Первая глава является введением и посвящена обзору java-технологий, а также принципам их разработки и поддержки.

Вторая глава посвящена архитектуре web-приложения. Для определенности изложения здесь описывается гипотетическая программная система, в рамках которой функционирует web-приложение. По ходу изложения эта модель уточняется и наполняется конкретным содержанием.

В третьей и четвертой главах пособия рассматриваются две основные технологии, используемые при разработке web-приложений на языке Java: технология сервлетов и Java Server Pages (JSP).

Пятая глава обобщает материал, изложенный в предыдущих двух главах, и знакомит с моделью среды выполнения web-приложения на языке Java.

В шестой главе рассматривается технология Java Database Connectivity (JDBC), применяемая web-приложениями для связи с реляционными базами данных. Кроме того, эта глава знакомит читателя с технологией Java Naming and Directory Interface (JNDI), часто используемой совместно с JDBC.

Последняя глава пособия представляет собой практикум, предназначенный для закрепления пройденного материала. Опытные программисты могут сразу начинать с него. Последовательность практических работ построена по принципу «от простого к сложному», и все они сопровождаются ссылками на разделы пособия. Каждая практи-

ческая работа состоит из нескольких заданий, последовательное выполнение которых приводит к построению несложного web-приложения.

В заданиях практических работ, как правило, используются результаты предшествующих заданий практических работ, поэтому последовательность их выполнения является важной. Для продолжения практического изучения web-программирования на языке Java автор рекомендует книгу Брюса Перри [7].

Следует предостеречь читателя от использования учебно-методического пособия в качестве справочной литературы по web-программированию на языке Java. Издание предназначено лишь для того, чтобы ознакомить с основными принципами и технологиями этого направления в программировании, а для полного осмысления следует обратиться к первоисточникам – документации, которую всегда можно найти на официальном сайте разработчика Java – Sun Microsystems Inc. [8].

## Глава 1. ВВЕДЕНИЕ

Разработку и поддержку всех спецификаций Java-технологий осуществляет специально для этого созданная компанией Sun Microsystems Inc. открытая и независимая организация – Java Community Process (JCP). К моменту издания книги в этой организации состояло более 800 представителей различных компаний и частных лиц. Основным предназначением JCP является разработка документов, имеющих название Java Specification Request (JSR-запрос на спецификацию). Формальная процедура разработки JSR регламентируется специальным документом – JSR2:Process Document, который определяет основную терминологию процесса разработки JSR, перечень возможных инициаторов разработки или ревизии JSR, состав, стадии разработки и утверждения JRS и т. п. Всем JSR-документам присваивается уникальный номер. Например, JSR 206: Java™ API for XML Processing (JAXP) 1.3 – спецификация очередной версии программного интерфейса для работы с XML-документами. С этим документом, как, впрочем, и со всеми другими JRS, можно ознакомиться на сайте JCP [9]. Законченный документ JSR (имеющий статус Final), как правило, состоит из заголовка (рисунок) и трех секций: **Identification** (идентификация), **Request** (запрос), **Contributions** (вклады).



JSRs: Java Specification Requests			
JSR 206: Java™ API for XML Processing (JAXP) 1.3			
JAXP 1.3 is the next version of JAXP, an implementation independent portable API for processing XML with Java™.			
<b>Status: Final</b>			
<b>Stage</b>		<b>Start</b>	<b>Finish</b>
Maintenance Release	<a href="#">Download page</a>	30 Nov, 2006	
Maintenance Draft Review	<a href="#">Download page</a>	09 Jan, 2006	13 Feb, 2006
Final Release	<a href="#">Download page</a>	30 Sep, 2004	
Final Approval Ballot	<a href="#">View results</a>	31 Aug, 2004	13 Sep, 2004
Proposed Final Draft 2	<a href="#">Download page</a>	24 Aug, 2004	
Proposed Final Draft	<a href="#">Download page</a>	05 Aug, 2004	
Public Review	<a href="#">Download page</a>	16 Dec, 2003	15 Jan, 2004
Community Draft Ballot	<a href="#">View results</a>	09 Sep, 2003	15 Sep, 2003
Community Review	<a href="#">Login page</a>	15 Aug, 2003	15 Sep, 2003
Expert Group Formation		18 Mar, 2003	
JSR Review Ballot	<a href="#">View results</a>	04 Mar, 2003	17 Mar, 2003
JCP version in use: 2.6			
Java Specification Participation Agreement version in use: 2.0			
Please direct comments on this JSR to: <a href="mailto:jsr-206-comments@jcp.org">jsr-206-comments@jcp.org</a>			
<b>Specification Lead</b>			
Jeff Suttor	Sun Microsystems, Inc.		
Norman Walsh	Sun Microsystems, Inc.		
<b>Expert Group</b>			
BEA Systems	Galbraith, Ben	IBM	
Novell, Inc.	Oracle	Sabin, Miles	
SAP AG	SAS Institute Inc.	SeeBeyond Technology Corp.	
Sun Microsystems, Inc.	Tmax Soft, Inc.		
 <a href="#">Expert Group Private Page</a>			
 <a href="#">JSR Community Update Page</a>			

Рисунок. Заголовок JSR 206: Java™ API for XML Processing (JAXP) 1.3

Секция Identification несколько дублирует информацию заголовка и содержит сведения об инициаторах запроса, организациях, поддерживающих JSR, сведения о разработчиках JSR.

Секция Request содержит основную информацию о новой спецификации и/или об изменениях предшествующих спецификаций.

Секция Contributions содержит ссылки на все документы, необходимые для изучения и внедрения технологии, описанной в JRS.

Важно понимать, что JCP разрабатывает только спецификации. Реализацией этих спецификаций (т. е. разработкой программного обеспечения, соответствующих спецификации) занимаются компании-производители программного обеспечения. По понятным причинам основной вклад в разработку такого программного обеспечения внесла компания Sun Microsystems Inc.

На сегодняшний день имеется три семейства Java-технологий (и соответственно JSR-спецификаций):

- Java Platform, Standard Edition (Java SE);
- Java Platform, Enterprise Edition (Java EE);
- Java Platform, Micro Edition Specification (Java ME).

В составе Java SE содержится два основных продукта: Java Runtime Environment (JRE) и Java Development Kit (JDK). JRE представляет собой библиотеки, виртуальную Java-машину (JVM) и другие компоненты для исполнения приложений, разработанных на языке Java. Кроме того, в JRE включены технологии Java Plug-In (для запуска апплетов в web-браузерах) и Java Web Start (для разворачивания внешних приложений из сети Internet). JDK включает в себя все, что содержится в JRE, а также компиляторы языка Java, отладчик и дополнительные библиотеки. Спецификация Java SE определяет Java SE Application Interface (Java SE API) – программный интерфейс, позволяющий вызывать методы java-ядра и стандартные функции для создания настольных (desktop) и клиентских приложений, использовать графику. Более подробно с составом и назначением технологий Java SE можно ознакомиться на сайте [10].

Java Micro Edition представляет собой набор технологий, применяемых для устройств с ограниченной вычислительной мощностью. Эта технология является самой молодой и сейчас бурно развивается. Java ME применяется, как правило, для программирования контроллеров, средств связи (например, мобильных телефонов) и встроенных систем (программирование бытовых приборов, пультов управления и т. п.). Более подробно с технологиями Java ME можно ознакомиться на сайте [11].

Наибольший интерес для нас представляет семейство технологий Java EE. В состав этого семейства включены технологии, позволяющие создавать web-приложения, которые являются частным случаем Java EE-приложений. В основе технологии Java EE лежат четыре основных документа:

- **Java EE Platform Specification** (спецификация платформы Java EE);
- **Java EE Reference Implementation** (образцовые реализации платформы Java EE);
- **Java EE Blueprints** (модель приложений Java EE);
- **Java Compatibility Test Suite** (набор тестов на совместимость платформы Java EE).

Спецификация Java EE Platform [12] определяет компонентную структуру Java EE-приложения и содержит минимальный набор свойств, которыми должен обладать сервер приложений (Application server), поддерживающий эту платформу. Сервер приложений – это сервер, умеющий исполнять прикладные программы, специальным образом установленные на нем. Если говорят о Java EE-сервере приложений (далее просто Java EE-сервер), то подразумевается, что он соответствует некоторой версии спецификации Java EE и может исполнять Java EE-приложения. Существует достаточно много различных Java EE-серверов: Sun GlassFish Enterprise Server (ранее Sun Java System Application Server) [13], IBM WebSphere Application Server [14], Oracle Application Server [15], JBOSS [16], BEA WebLogic [17] и т. д. Важным является то, что, если любые два сервера приложений соответствуют спецификации Java EE Platform, то любое Java EE-приложение, которое может быть исполнено на одном сервере без перекомпиляции, может быть исполнено и на другом (с учетом соответствия версий спецификаций). Разница может заключаться только в процедурах установки и настройки приложения. При этом приложение остается нейтральным относительно программно-аппаратной среды, в которой работает сервер приложений.

Составной частью любого сервера приложений является web-сервер (его часто называют web-контейнером). В некоторых случаях это может быть отдельный продукт, который встраивается в сервер приложений (например, в JBOSS используется web-сервер Apache Tomcat [18]), в других случаях web-сервер может являться неотделимой составной частью сервера приложений (например, GlassFish) или вообще могут использоваться как несколько различных web-серверов, так и собственный встроенный (WebSphere).



Образцовые реализации платформы Java EE – это практические указания по разработке программных продуктов соответствующих спецификации этой платформы, а также сами действующие программные продукты, которые могут быть использованы в качестве образца. Компания Sun Microsystems Inc. предлагает в качестве образцовой реализации платформы Java EE свой продукт – сервер приложений Sun GlassFish Enterprise Server, который поддерживает весь спектр технологий, описанных в спецификации Java EE. С помощью этого модельного сервера разработчики серверов приложений могут проверить переносимость приложений между собственной реализацией сервера и образцовой реализацией, а разработчики Java EE-приложений – для разработки прототипов приложений.

Модель приложений Java EE предназначена для прикладных программистов, разрабатывающих приложения для этой платформы. В этом документе содержатся практические рекомендации по разработке Java EE-приложений и примеры, которые могут служить в качестве образцов реализации.

Набор тестов на совместимость платформы Java EE предназначен в основном для разработчиков серверов приложений, реализующих платформу Java EE. С помощью предложенных здесь тестов можно проверить разработанный продукт на соответствие спецификациям (иногда говорят – стандартам) платформы Java EE. Перечень программных продуктов, успешно прошедших проверку на наборе тестов и получивших от Sun Microsystems Inc. сертификат соответствия, публикуются на сайте компании.

Перечень технологий, предлагаемых платформой Java EE, достаточно большой и при этом постоянно пополняется. Каждая из технологий развивается, как правило, независимо от других и имеет собственную траекторию версий. В документации любого Java EE-сервера всегда можно найти перечень поддерживаемых им технологий с указанием версий.

В этом учебно-методическом пособии рассматривается только некоторая часть технологий, входящих в состав платформы Java EE, которые применяются для разработки web-приложений. Основными web-технологиями являются JavaServlet (технология сервлетов) и Java ServerPages.

## Глава 2. АРХИТЕКТУРА WEB-ПРИЛОЖЕНИЯ

### 2.1. Предисловие к главе

Существует две основные отличительные особенности web-приложения: 1) среда исполнения приложения; 2) его многокомпонентная структура.

Традиционные приложения исполняются в среде, которую предоставляет операционная система. Любое web-приложение может функционировать только в специальной среде, называемой web-контейнером. Web-контейнер по своей сути является набором классов и интерфейсов, имеющих то же предназначение, что и API (Application program interface) для операционных систем. Контейнер, в принципе, может быть встроен в любую другую программную систему, но, как правило, является составной частью web-серверов или серверов приложений. Важно понимать, что web-приложение ничего не «знает» о среде, в которой работает web-контейнер и тем более сервер, – это является одним из необходимых условий обеспечения независимости web-приложения от платформы.

В отличие от приложения, операционной системы, которые, как правило, являются монолитными, web-приложение состоит из нескольких компонентов. В общем случае эти компоненты могут быть распределены в сети. Спецификация Java EE Platform Specification, о которой уже говорилось выше, определяет состав типов компонентов Java EE-приложения и описывает спецификацию для каждого типа компонента. Не все типы, описанные в этой спецификации, обязательно поддерживаются web-серверами. Перечень типов компонентов и их версии определяется спецификацией конкретного web-сервера.

Дальнейшее изложение материала опирается на модель гипотетической программной системы WStudy, архитектура которой изображена на рис. 2.1. По мере освоения web-технологий эта модель будет уточняться, конкретизироваться и изменяться.

На этом этапе определим WStudy как систему, позволяющую удаленному пользователю сети Internet с помощью web-браузера установить связь с web-приложением ANaive, передать ему запрос и получить от него ответ. Web-приложение функционирует в среде некоторого web-сервера, который в свою очередь работает в среде некоторой операционной системы. Приложение ANaive состоит из нескольких компонентов (количество их будет наращиваться по ходу изложения).

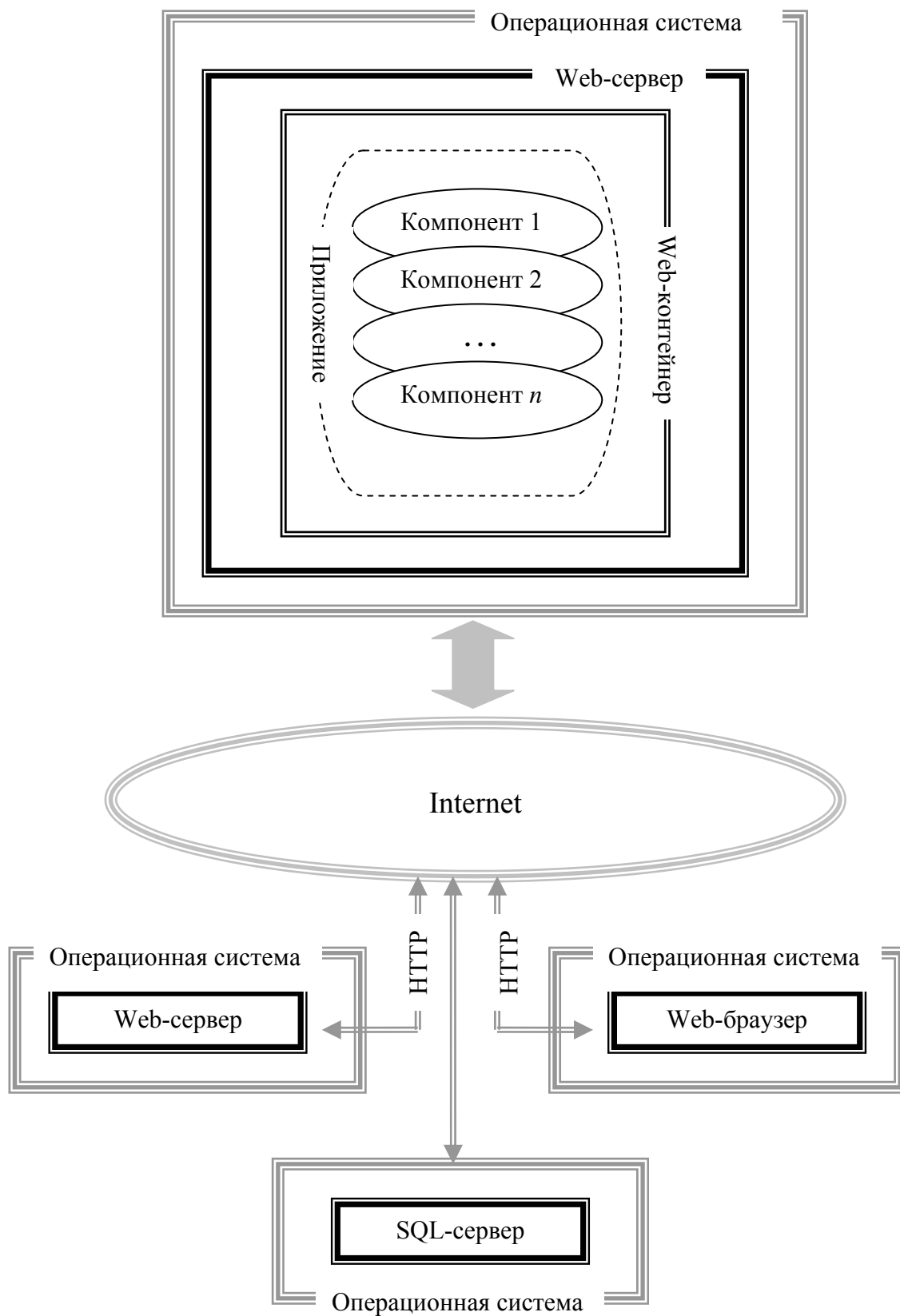


Рис. 2.1. Архитектура программной системы WStudy

В процессе своей работы приложение может вызывать другие web-приложения, которые могут работать как рамках того же, так и другого (в общем случае удаленного) web-сервера. При обработке запросов пользователя приложение ANaive устанавливает соединение с SQL-сервером, записывает и извлекает информацию из базы данных.

## 2.2. Web-сервер Apache Tomcat

Процедура установки и настройки любого web-приложения зависит от типа web-сервера, в среде которого предполагается его работа и, как правило, сводится к заполнению (или созданию) специальных текстовых файлов. Информация в этих файлах описывает конфигурацию самого web-приложения, а также способы его взаимодействия с различными службами и сервисами web-сервера.

Для того чтобы дальнейшее изложение было предметным, уточним модель WStudy (п. 2.1) и будем предполагать, что в качестве web-сервера здесь используется Apache Tomcat [18].

Apache Tomcat (далее просто Tomcat) является продуктом (проект Jakarta [20]) некоммерческой организации Apache Software Foundation (ASF, фонд программ Apache) [19], распространяемым под лицензией Apache Software License (ASL). Она дает право использовать программное обеспечение для любых целей, свободно распространять, изменять и распространять измененные копии.

К моменту написания этой книги последней стабильной версией Tomcat была 6.0.18. Она гарантирует соответствие спецификациям Java Servlet 2.5 и Java Server Pages 2.1, входящим в состав Java Platform EE. Более того, Tomcat является официальной эталонной реализацией технологий, определяемых этими спецификациями.

HTML-файлы, Java Servlet и Java Server Pages являются тремя основными типами компонентов, входящих в состав web-приложения. Используемая версия языка HTML, как правило, не связана с версией web-сервера, т. к. HTML-файлы интерпретируются web-браузером, а web-сервер осуществляет лишь их пересылку, используя для этого протокол HTTP. Поддержку и развитие спецификаций языка HTML и протокола HTTP обеспечивает консорциум W3C [21]. В дальнейшем изложении будем ориентироваться на версии HTML 4.1 (или XHTML 1.0) и HTTP 1.1.

Дистрибутивная копия продукта Tomcat 6.0.18 доступна для скачивания с официального сайта Apache Tomcat [18]. Инсталляция сервера Tomcat является простой процедурой и занимает, как правило, не

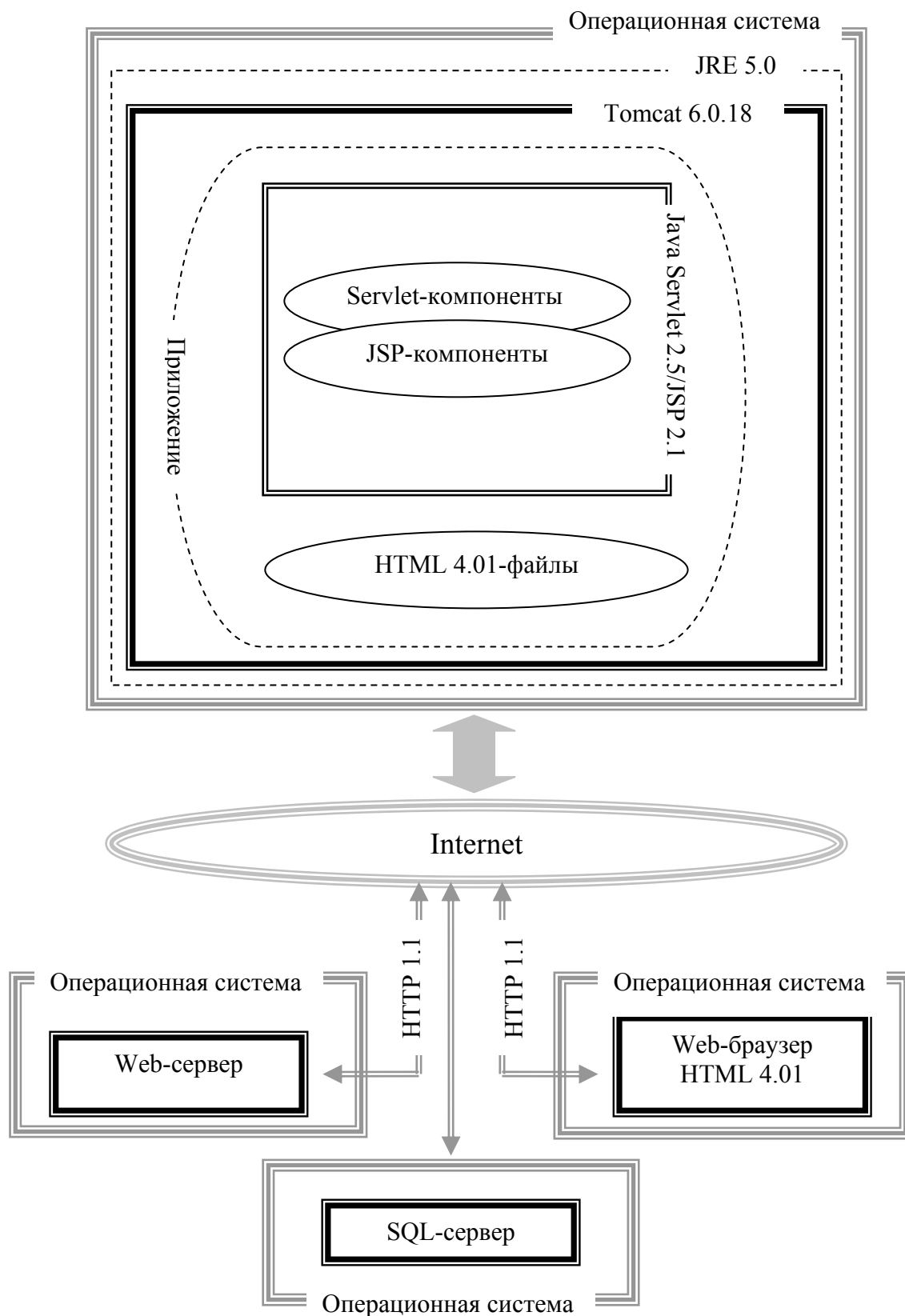


Рис. 2.2. Уточненная модель программной системы WStudy

более 10 мин. Необходимым условием для успешной инсталляции является наличие предустановленного программного обеспечения Java Runtime Environment версии не ниже 5.0.

С учетом уточнений, сделанных здесь, модель программной системы WStudy приобретает более определенные черты (рис. 2.2).

### 2.3. Структура директорий Tomcat

На рис. 2.3 приведен результат работы команды **dir** для директории web-сервера Tomcat 6.0.18 после его инсталляции в операционной системе Windows.

```
Содержимое папки C:\Program Files\Apache Software Foundation\Tomcat 6.0

21.01.2008 02:58 <DIR> .
21.01.2008 02:58 <DIR> ..
14.01.2008 17:41 <DIR> bin
21.01.2008 02:58 <DIR> common
09.03.2008 14:41 <DIR> conf
22.04.2008 08:09 <DIR> lib
09.03.2008 14:36 <DIR> logs
14.01.2008 17:41 <DIR> temp
09.03.2008 14:36 <DIR> webapps
14.01.2008 17:41 <DIR> work
                10 папок 3 878 817 792 байт свободно
```

Рис. 2.3. Директория инсталляции Tomcat 6.0

Директория **Tomcat 6.0\bin** содержит исполняемые файлы и jar-файлы, необходимые для установки параметров и запуска сервера Tomcat.

Директория **Tomcat 6.0\common** предназначена для хранения документов для всех web-приложений и ядра Tomcat jar-файлов и классов.

Директория **Tomcat 6.0\conf** содержит конфигурационные файлы Tomcat.

Директория **Tomcat 6.0\lib** содержит jar-файлы, доступные для всех web-приложений и недоступных для ядра Tomcat.

Директория **Tomcat 6.0\logs** содержит журналы сервера.

Директории **Tomcat 6.0\temp** и **Tomcat 6.0\work** используются для хранения промежуточных и временных файлов.

Директорий **Tomcat 6.0\webapps** – месторасположение web-приложений по умолчанию, которые могут быть исполнены сервером.

На рис. 2.4 приведен результат выполнения команды **dir** для директории **Tomcat 6.0\webapps**. В этой директории содержатся файлы

web-приложения, поставляемые с Tomcat (встроенные web-приложения), а также файлы новых разработанных web-приложений.

```
Содержимое папки C:\Program Files\Apache Software Foundation\Tomcat
6.0\webapps

21.08.2008 10:21 <DIR> .
21.08.2008 10:21 <DIR> ..
21.08.2008 10:21 <DIR> ANaive
14.01.2008 17:41 <DIR> docs
14.01.2008 17:41 <DIR> examples
14.01.2008 17:41 <DIR> host-manager
14.01.2008 17:41 <DIR> manager
14.01.2008 17:41 <DIR> ROOT
      8 папок 3 624 312 832 байт свободно
```

Рис. 2.4. Директория web-приложений Tomcat 6.0

Директория **ANaive** предназначена для хранения файлов компонентов приложения ANaive. Как правило, наименование web-приложения совпадает с наименованием директории, предназначенной для файлов этого приложения. Процесс установки приложения ANaive будет обсуждаться ниже. Здесь же поясним, что «ручное» создание директории ANaive не приведет к автоматическому появлению соответствующего web-приложения, т. к. необходимо внести информацию в конфигурационные файлы Tomcat.

Директория **docs** содержит документацию Tomcat. Эта директория необходима для того, чтобы с помощью сервера можно было просматривать документацию Tomcat.

Директория **examples** содержит множество примеров web-приложений, которые, как правило, используются для проверки инсталляции Tomcat.

Директории **host-manager** и **manager** используется специальным web-приложением Tomcat Web Application Manager, поставляемым с Tomcat. С помощью этого приложения можно выполнять некоторые операции, необходимые для установки и настройки web-приложений.

Директория **ROOT** используется для web-приложения Tomcat, которое будет выполняться по умолчанию, если не указано никакого подконтекста в URL сервера. Сразу после установки эта директория содержит web-приложение, поставляемое с Tomcat и предназначенное для просмотра исходной информации о сервере и вызова других встроенных приложений. Если на компьютере с установленным Tomcat в web-браузере указать в адресной строке **http://localhost:8080**, то это привет к вызову приложения ROOT.

На рис. 2.5 приведен результат выполнения команды **dir** для директории **Tomcat 6.0\webapps\ANaive**. Структура директории web-приложения ANaive определяется спецификацией Java Servlet 2.5. В данном случае директория содержит обязательную поддиректорию WEB-INF, предназначенную для хранения классов, библиотек и конфигурационных файлов web-приложения ANaive, а также HTML-файл index.html, который не является обязательным, но, как правило, файл с таким именем используется в качестве стартовой страницы web-приложения. На самом деле, имя стартовой страницы определяется в специальном конфигурационном файле, речь о котором пойдет ниже. Структура директории приложения ANaive, приведенная здесь, соответствует простому web-приложению. По мере изложения материала эта структура будет дополняться.

```
Содержимое папки C:\Program Files\Apache Software Foundation\Tomcat 6.0\
webapps\ANaive

21.08.2008 22:38 <DIR> .
21.08.2008 22:38 <DIR> ..
06.08.2008 02:24 256 index.html
21.08.2008 11:46 <DIR> WEB-INF
      1 файлов 256 байтов
      3 папок 3 731 587 072 байт свободно
```

Рис. 2.5. Директория web-приложения ANaive

На рис. 2.6 приведен результат выполнения команды **dir** для директории **Tomcat 6.0\webapps\ANaive\WEB-INF**. Директория содержит две поддиректории – **classes** и **lib**.

```
Содержимое папки C:\Program Files\Apache Software Foundation\
Tomcat 6.0\webapps\ANaive\WEB-INF

21.08.2008 23:08 <DIR> .
21.08.2008 23:08 <DIR> ..
21.08.2008 11:46 <DIR> classes
21.08.2008 11:46 <DIR> lib
06.08.2008 02:21 633 web.xml
      1 файлов 633 байтов
      4 папок 3 731 570 688 байт свободно
```

Рис. 2.6. Директория WEB-INF web-приложения ANaive

Директория **classes** предназначена для хранения исполняемых файлов java-классов web-приложения ANaive. Принципы организации этой директории будут обсуждаться ниже.



Директория **lib** предназначена для хранения jar-файлов web-приложения ANaive.

Файл **web.xml** – конфигурационный файл web-приложения ANaive, который принято называть *дескриптором развертывания web-приложения*.

## 2.4. Дескриптор развертывания web-приложения

Дескриптор развертывания является важной частью web-приложения, предназначенной для хранения его основных параметров. На рис. 2.7 представлен пример содержимого дескриптора простого web-приложения ANaive.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  version="2.5">

  <display-name>ANaive</display-name>

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>

</web-app>
```

Рис. 2.7. Пример дескриптора развертывания приложения ANaive

Как это видно на рис. 2.8, дескриптор развертывания приложения представляет собой xml-файл, корневым элементом которого является тег **<web-app>**. Дескриптор приложения может содержать достаточно много различных и повторяющихся элементов. Порядок элементов внутри **<web-app>** и их синтаксис определяется схемой XML, которая, как это видно из параметров тега **<web-app>**, находится на **http://java.sun.com/xml/ns/j2ee/web-app\_2\_5.xsd**. С описанием элементов дескриптора развертывания, их назначением и применением можно ознакомиться в спецификации Java Servlet 2.5 [12] или в источнике [5].

В самом простом случае дескриптор развертывания состоит только из одного тега – **<web-app>**, внутри которого ничего нет. В нашем случае имеется еще три тега: **<display-name>**, **<welcome-file-list>** и **<welcome-file>**.

Тег **<display-name>** не является обязательным, но если есть, то не может повторяться более одного раза. Он предназначен для указания имени web-приложения, которое потом может быть использовано в графическом интерфейсе. Для этого имени не требуется уникальность, и его значение не оказывает влияния на работу приложения.

Тег **<welcome-file-list>** тоже не является обязательным и предназначен для указания списка стартовых страниц web-приложения. Имена файлов страниц указываются внутри тега **<welcome-file-list>** с помощью одного или более тегов **<welcome-file>**.

В нашем примере в качестве стартовой страницы используется файл **index.html**. В общем случае может быть указано несколько стартовых страниц для одного web-приложения. В этом случае поиск их осуществляется в указанном порядке. Если ни одного файла не было найдено, сервер возвращает сообщение об ошибке HTTP Status 404. Пусть, например, для вызова приложения ANaive используется адресная строка **http://xxx:8080/ANaive**, где xxx – разрешаемое символическое имя компьютера. Тогда первой отобразится страница **index.html**, т. к. именно она указана в списке стартовых страниц дескриптора развертывания. При отсутствии тега **<welcome-file-list>** (и соответственно тегов **<welcome-file>**) имена стартовых страниц Tomcat определяет с помощью списка в теге **<welcome-file-list>** конфигурационного файла **Tomcat 6.0/conf/web.xml**. На рис. 2.8 отображен фрагмент этого файла, содержащий список стартовых страниц.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  version="2.5">

<!-- ..... -->

<welcome-file-list>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.htm</welcome-file>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>

</web-app>
```

Рис. 2.8. Список стартовых страниц по умолчанию в конфигурационном файле Tomcat

Дескриптор развертывания web-приложения содержит информацию, необходимую web-контейнеру для взаимодействия с приложением.

ем. По мере изложения материала сведения о структуре и составе этого файла будут постоянно уточняться и дополняться.

## 2.5. Утилита Apache Ant

Реальное web-приложение обычно включает достаточно большое количество файлов, размещенных в разветвленной системе директорий файловой системы. Компоновка такого приложения «вручную» представляет собой трудоемкое занятие. Поэтому для выполнения компоновки, как правило, используют специальные программные средства, позволяющие упростить этот процесс.

*Apache Ant* – средство автоматизации, созданное на базе Java и XML, распространяемое фондом ASF [19] как программное обеспечение (утилита **Ant**) с открытым исходным кодом, которое представляет собой инструмент, позволяющий автоматизировать процесс компоновки java-проекта. Сюда относится компиляция java-файлов, создание java-библиотек (jar-файлы) и специальных файлов для размещения приложения на сервере (war-файлы), а также выполнение задач, связанных с файловой системой (создание, перемещение, копирование, удаление файлов и директорий). Важной особенностью утилиты Ant является ее независимость от платформы и возможность расширения. На момент написания книги доступной для скачивания на официальном сайте [22] была версия Ant 1.7.0. Необходимым условием для инсталляции утилиты **Ant** является предварительная установка **Java Development Kit**. Процесс установки не сложен и занимает не более 10 мин. Для проверки правильности установки следует вызвать утилиту **Ant** с параметром **version** в командной строке. Результат выполнения команды **ant – version** должен быть примерно такой же, как на рис. 2.9.

```
C:\ant -version
Apache Ant version 1.7.0 compiled on December 13 2006
```

Рис. 2.9. Проверка правильности установки утилиты Ant

Для получения справки о параметрах команды **ant** можно воспользоваться вызовом **ant–help**.

Порядок компоновки web-приложения описывается в файле сборки, который содержит входную информацию для утилиты **Ant**. Файл сборки – это xml-файл, описывающий с помощью xml-элементов и атрибутов последовательность действий, которые должна исполнить

утилита **Ant**. По умолчанию имя файла сборки – **build.xml**. Если в параметрах вызова команды **ant** не указан путь, то предполагается, что он находится в текущей директории.

Файл сборки имеет единственный корневой элемент **<project>**. Внутри него может находиться один или несколько элементов **<target>** (эти элементы называют *целями*), которые объединяют группу элементов *задач*. Каждой задаче соответствует свой тег. Например, задача, выполняющая компиляцию java-файлов, описывается с помощью тега **<javac>**, задача копирования файлов – с помощью тега **<copy>**, задача создания jar-файла – с помощью тега **<jar>** и т. д.

На рис. 2.10 приведен пример файла сборки для web-приложения ANaive. При этом предполагается, что рабочие экземпляры файлов приложения находятся в директории **D:\Workspace\_ANaive** (рис. 2.11).

```
<project name="ANaive" default = "copy" >

<target name="init">
  <echo message="Building ANaive Project" />
</target>

<target name="create" depends = "init">
  <delete dir= "C:\Program Files\Apache Software Foundation\Tomcat
6.0\webapps\ANaive\" />
  <mkdir dir= "C:\Program Files\Apache Software Foundation\Tomcat
6.0\webapps\ANaive\" />
  <mkdir dir= "C:\Program Files\Apache Software Foundation\Tomcat
6.0\webapps\ANaive\WEB-INF" />
</target>

<target name="copy" depends = "create">
  <copy file= "D:\Workspace_ANaive\WEB-INF\web.xml"
    todir="C:\Program Files\Apache Software Foundation\Tomcat
6.0\webapps\ANaive\WEB-INF\" />
  <copy file= "D:\Workspace_ANaive\html\index.html"
    todir="C:\Program Files\Apache Software Foundation\Tomcat
6.0\webapps\ANaive\" />
</target>

</project>
```

Рис. 2.10. Пример файла сборки для web-приложения ANaive

В рабочей директории проекта (рис. 2.11) хранятся все необходимые файлы разрабатываемого приложения. Структурировать эту директорию можно различными способами, каждый разработчик может сделать это по-своему. При установке же приложения на web-сервер

(часто этот процесс называют *публикацией web-приложения*) необходимо разместить эти файлы в соответствии с правилами, определяемыми спецификациями сервера, компонента и логикой самого приложения. В данном примере будем использовать директорию **ant** для хранения файла сборки **build.xml**, директорию **html** – для хранения html-файлов, а назначение директории WEB-INF будет таким же, как и прежде.

```
Содержимое папки D:\Workspace_ANaive

23.08.2008 17:23 <DIR> .
23.08.2008 17:23 <DIR> ..
23.08.2008 16:59 <DIR> ant
23.08.2008 16:57 <DIR> html
23.08.2008 16:57 <DIR> WEB-INF
                    5 папок 27 275 943 936 байт свободно
```

Рис. 2.11. Пример рабочей директории проекта ANaive

Файл сборки, представленный на рис. 2.10, имеет единственный корневой элемент **<project>**, включающий три xml-элемента **<target>**. Для именования всех этих элементов используется атрибут **name**.

Элемент **<target>** с именем **init** (далее просто цель **init**) включает один элемент **<echo>** (задача **echo**), который предназначен для вывода на консоль сообщения «**Building ANaive Project**», указанного атрибутом **message**.

Цель **create** включает три задачи. Первая удаляет указанную атрибутом **dir** директорию вместе ее содержимым. Если указанной директории нет, то задача не выполняется и ошибки не возникает. Две другие задачи **mkdir** создают директории, имена которых также указаны атрибутом **dir**.

Цель **copy** включает две задачи типа **copy**. Каждая из них копирует файл, указанный атрибутом **file**, в директорию, указанную атрибутом **todir**.

Два тега **<target>** содержат атрибут с именем **depends**, который указывает имена одной или нескольких целей, которые должны быть успешно выполнены предварительно. В примере на рис. 2.10 цель **copy** может быть выполнена только после успешного выполнения задач цели **create**, а цель **create** – после выполнения задач цели **init**.

В теге **<project>** атрибут **default** указывает цель, задачи которой должны быть выполнены. Этот атрибут принимается во внимание

только в том случае, если при вызове команды **ant** явно не указана цель.

В примере на рис. 2.10 для атрибута **default** тега **<project>** установлено значение **copy**. Учитывая во внимание атрибуты **depends** тегов **<target>**, сначала выполнится цель **init**, в случае ее корректного завершения – цель **create** и, наконец, если не возникло ошибок в **create**, выполнится цель **copy**.

На рис 2.12 приводится протокол работы, выводимый утилитой **ant** при исполнении файла сборки **D:\Workspace\_ANaive\ant\build.txt**, содержимое которого приведено на рис. 2.10.

```
D:\Workspace_ANaive\ant>ant
Buildfile: build.xml
init:
    [echo] Build ANaive Project

create:
    [delete] Deleting directory C:\Program Files\Apache Software
Foundation\Tomcat 6.0\webapps\ANaive
    [mkdir] Created dir: C:\Program Files\Apache Software
Foundation\Tomcat 6.0\webapps\ANaive
    [mkdir] Created dir: C:\Program Files\Apache Software Foundation\
Tomcat 6.0\webapps\ANaive\WEB-INF

copy:
    [copy] Copying 1 file to C:\Program Files\Apache Software Foundation\
Tomcat 6.0\webapps\ANaive\WEB-INF

BUILD SUCCESSFUL
Total time: 0 seconds
```

Рис. 2.12. Пример протокола работы утилиты Ant

В дальнейшем примеры web-приложений, приведенные в учебно-методическом пособии, будут часто содержать файлы сборок утилиты Ant. По мере необходимости будут вводиться новые задачи Ant, позволяющие компоновать более сложные web-приложения.

Для детального изучения возможностей утилиты рекомендуется ознакомиться с ее документацией, которую можно всегда скачать с официального сайта [22]. Примеры применения утилиты Ant можно найти в источнике [7].

Следует отметить, что существуют и другие программные средства компоновки web-приложений. Например, в последнее время популярным стал продукт Apache Maven 2 [23], который позволяет не только компоновать web-приложения, но и управлять процессом его разработки в распределенной среде.

## 2.6. Разработка простейшего web-приложения

Будем предполагать, что на компьютере уже установлены JDK, сервер Apache Tomcat и утилита Apache Ant. Ограничим возможности гипотетического web-приложения ANaive до минимума и пусть его работа сводится только к выдаче на экран web-браузера приветственного сообщения «Hello, World».

Для построения такого приложения следует выполнить следующую последовательность действий:

- 1) создать рабочую директорию, в которой предполагается хранить разрабатываемые файлы web-приложения;
- 2) создать в рабочей директории файл **index.html**, выводящий на браузер сообщение «Hello, World»;
- 3) создать в рабочей директории файл **web.xml** – дескриптор развертывания приложения;
- 4) создать в рабочей директории файл **build.xml** – файл сборки web-приложения;
- 5) выполнить команду **ant** (текущим должна быть директория с файлом сборки);
- 6) остановить (если был запущен) сервер Tomcat и стартовать его снова;
- 7) запустить web-браузер, вызвать web-приложение ANaive и убедиться в его работоспособности.

Структура рабочей директории может быть выбрана произвольно. В качестве примера можно использовать структуру, предложенную на рис. 2.11.

В качестве содержимого для файла **index.html** можно использовать html-текст, представленный на рис. 2.13.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
< title>My First Web-Application</title>
</head>

<body>
Hello, World!
</body>
</html>
```

Рис. 2.13. Пример файла index.html

Для построения дескриптора развертывания web-приложения можно использовать текст на рис. 2.10. В принципе, внутреннее содержимое тега **<web-app>** можно сделать пустым, так как имя **index.html** установлено по умолчанию в конфигурационном файле сервера **Tomcat 6.0/conf/web.xml**.

В качестве образца для построения файла сборки (**build.xml**) можно использовать текст на рис. 2.10. Изменению подлежат только имя рабочей директории, если оно выбрано иное, чем в разобранным примере.

При исполнении команды **ant** следует обратить внимание на то, чтобы в текущей директории был файл **build.xml**. Кроме того, следует проанализировать протокол работы улиты Ant: при правильной работе протокол должен быть похожим на рис. 2.12.

Перезапуск сервера Tomcat необходим для того, чтобы он «узнал» о наличии нового приложения (определяется появлением новой директории). Остановка и запуск Tomcat осуществляется с помощью специальной графической утилиты **Tomcat 6.0\bin\tomcat6w.exe**, входящей в состав программного обеспечения сервера.

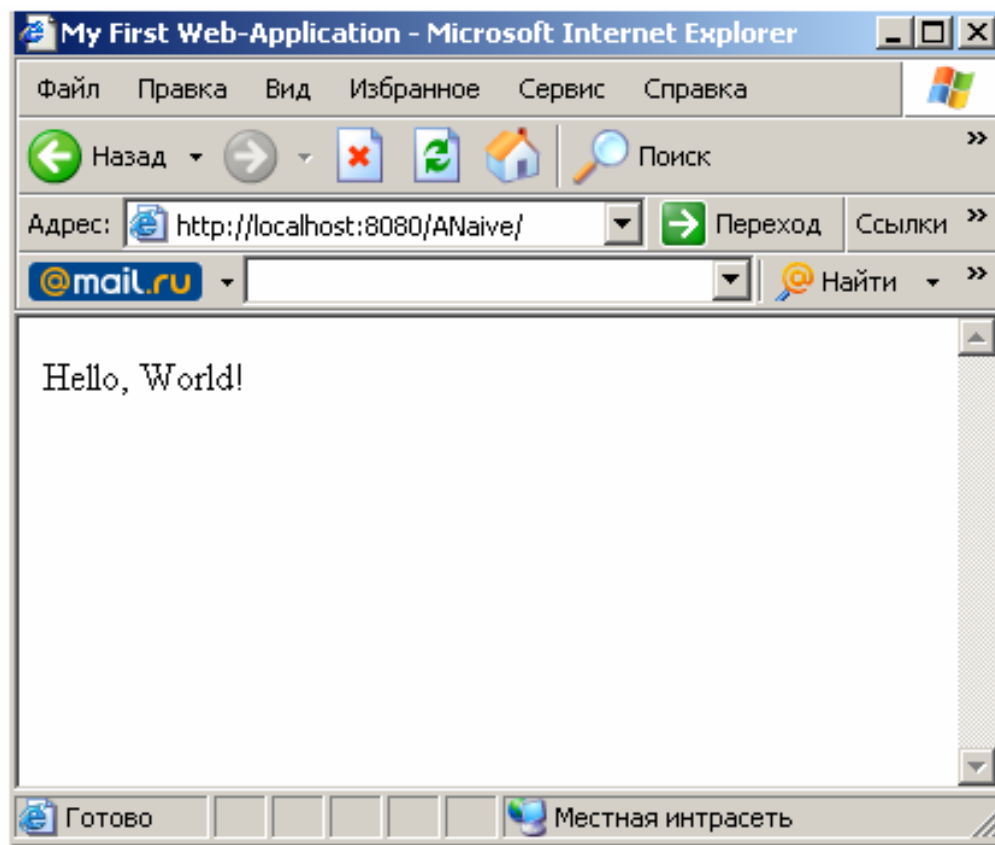


Рис. 2.14. Вызов приложения ANaive с помощью web-браузера Internet Explorer



Обычно вызов этой утилиты возможен с помощью ярлыка **Пуск/Все программы/Apache Tomcat 6.0/Configure Tomcat**, который устанавливается во время инсталляции Tomcat.

Самый простой способ проверки работоспособности приложения ANaive – вызвать приложение с помощью web-браузера, запустив его на том же компьютере, где запущен сервер Tomcat. В адресной строке браузера следует набрать **http://localhost:8080/ANaive**. В случае правильной работы приложения результат его вызова должен быть похожим на рис. 2.14.

## 2.7. Итоги главы

1. Необходимым условием для работы любой программной системы, использующей java-технологии, является предварительная инсталляция продукта Java Runtime Environment (JRE). Если при этом предполагается разработка программ на Java, то необходимо расширить JRE до продукта Java Development Kit (JDK).

2. Основными особенностями web-приложения являются среда его выполнения и многокомпонентная структура. Среда выполнения web-приложения – это web-контейнер, который, как правило, встраивается в web-сервер или в сервер приложений. Web-контейнер является набором классов и интерфейсов, имеющих то же предназначение, что и API (Application program interface) для операционных систем. В состав web-приложения могут входить объекты различного назначения: java-классы, html-файлы, xml-файлы, графические файлы и т. п.

3. Web-приложение является частным случаем J2EE-приложения. Состав компонентов web-приложения зависит от возможностей web-контейнера, в среде которого оно работает. Состав компонентов J2EE-приложения определяется спецификацией Java EE Platform Specification. Web-контейнеры поддерживают только некоторую часть этой спецификации. Как правило, это спецификации технологий Java Servlet и Java Server Pages.

4. Разработка web-приложения сводится к разработке его компонентов, размещению их в определяемой спецификацией Java Servlet структуре директорий web-сервера и подготовке специального xml-файла, называемого дескриптором развертывания web-приложения.

5. Процесс создания внутри структуры директорий web-сервера структуры директорий web-приложения и заполнение их файлами web-приложения называется компоновкой web-приложения. Для выполнения компоновки, как правило, используются специальные программные средства, например утилиты Apache Ant и Apache Maven.

6. Для вызова web-приложения, как правило, используется web-браузер. Для вызова web-приложения необходимо в адресной строке набрать его адрес (Uniform Resource Locator, URL – унифицированный локатор ресурса). В простейшем случае адрес представляет собой строку `http://cccc:8080/ppp`, где `http` – протокол связи, `ccc` – разрешаемое символическое имя или ip-адрес компьютера, `8080` – номер tcp-порта (этот номер, как правило, используется в java-серверах по умолчанию) и `ppp` – имя web-приложения (оно совпадает с именем директории web-приложения).

## Глава 3. РАЗРАБОТКА СЕРВЛЕТА

### 3.1. Предисловие к главе

**Сервлет** – это web-компонент, представляющий собой класс Java, предназначенный для динамического формирования содержимого ответов на запросы клиентского приложения (обычно web-браузера). Сервлеты размещаются на сервере и выполняются в специальной среде – контейнере сервлетов (Servlet engine), – на виртуальной Java-машине (JVM) сервера. Контейнер сервлетов является составной частью web-контейнера и определяется спецификацией Java Servlet. В общем случае сервлеты не зависят от протокола связи, но наиболее часто они применяются для обработки http-запросов.

Количество сервлетов на web-сервере не ограничено. Один сервлет может отвечать на один или несколько типов запросов клиента. Можно говорить о двух крайних случаях реализации сервлетов: 1) для каждого типа запроса реализуется свой сервлет; 2) все запросы обрабатываются одним общим сервлетом. В реальности запросы, обрабатываемые сервлетом, объединяются по функциональному признаку.

Следует подчеркнуть, что сервлет управляется web-контейнером, т. е. имеет API, управляемый событиями (even-driven API). Загрузку, выгрузку сервлетов, вызов методов, предварительную обработку запросов и ответов клиента, а также многое другое выполняет web-контейнер.

В спецификации Java Servlet предусмотрена следующая функциональность сервлета:

- 1) прием и чтение данных, посылаемых клиентом в качестве запроса;
- 2) получение любой информации о запросе (свойства запроса, имя хоста-отправителя, свойства браузера и т. п.);
- 3) генерация и форматирование ответа на запрос; установка необходимых параметров ответа;
- 4) в процессе обработки запроса сервлет может обращаться к базе данных, использовать файловую систему, осуществлять необходимые вычисления и вызывать другие приложения (в т. ч. другие сервлеты);
- 5) отсылка сформированного ответа клиенту.

В дальнейшем изложение в основном будет ориентировано на сервлеты, обрабатывающие http-запросы.

### 3.2. Структура и интерфейсы сервлета

Java Servlet API организован в виде двух пакетов, содержащих два набора базовых классов: **javax.servlet** – общие для всех сервлетов интерфейсы и классы; **javax.servlet.http** – расширение пакета **javax.servlet**, обеспечивающее поддержку протокола HTTP.

На рис. 3.1 представлен java-класс **Sss**, расширяющий класс **javax.servlet.http.HttpServlet** и реализующий интерфейс **javax.servlet.Servlet**. Объект класса **Sss** называется сервлетом **Sss**.

```
import java.io.IOException;           // исключения ввода/вывода
import javax.servlet.*;              // интерфейсы и классы общего типа
import javax.servlet.http.*;         // расширение javax.servlet для http

public class Sss extends HttpServlet implements Servlet {
    public Sss() {
        super();
        System.out.println("Sss:constructor");
    }
    public void init(ServletConfig sc) throws ServletException {
        // инициализация сервлета
        super.init();
        System.out.println("Sss:init");
    }
    public void destroy() {
        // перед уничтожением сервлета
        super.destroy();
        System.out.println("Sss:destroy");
    }
    protected void service(HttpServletRequest rq, HttpServletResponse rs)
        throws ServletException, IOException {

        // обработка http-запроса
        System.out.println("Sss:service:"+rq.getMethod());
    }
}
```

Рис. 3.1. Пример класса сервлета

Следует подчеркнуть, что пакеты **javax.servlet** и **javax.servlet.http.HttpServlet** не входят в состав JDK, а принадлежат web-серверу. В нашем случае эти пакеты следует искать в библиотеке **Tomcat 6.0\lib\servlet-api.jar**.

Это важный момент, поскольку именно web-сервер обеспечивает среду выполнения сервлета. Разные web-серверы организуют свою работу различными способами и каждый web-сервер имеет свои пакеты для сервлетов, но интерфейс любого сервлета будет с точностью до версии одинаков.

Все сервлеты должны реализовать интерфейс **javax.servlet.Servlet** (далее просто **Servlet**). Этот интерфейс предполагает три основных метода (реализация их представлена на рис. 3.1) и два вспомогательных, которые будут рассмотрены ниже.

Метод **init** вызывается сервером при инициализации сервлета. Этот метод позволяет программисту выполнить некоторые действия перед началом работы сервлета. В качестве параметра метод получает объект, реализующий интерфейс **ServletConfig**. Методы этого интерфейса реализуются сервером, они позволяют сервлету получить информацию о своей конфигурации.

Метод **destroy** тоже вызывается сервером, но при его выгрузке. Этот метод используется разработчиком сервлета для выполнения действий, связанных с окончанием работы, – освобождение ресурсов, закрытие соединений с сервером базы данных и т. п.

Метод **service** предназначен для обработки запроса клиента. Метод вызывается сервером при получении запроса клиента на вызов сервлета. Сервер формирует два параметра. Первый реализует интерфейс **HttpServletRequest** и используется для того, чтобы получить информацию о http-запросе. Вторым параметром, реализующим интерфейс **HttpServletResponse**, дает возможность сервлету формировать http-ответ клиенту. В данном примере в функции **service** используется вызов метода **getMethod** интерфейса **HttpServletRequest**. Функция **getMethod** позволяет определить тип http-запроса (**get**, **post**, **put**, **delete**, **options** и т. д.).

Класс **Sss** расширяет класс **HttpServlet**, который в свою очередь расширяет один из базовых классов API сервлета – **GenericServlet**. Класс **GenericServlet** реализует интерфейсы **Servlet** и **ServletConfig**. Основное назначение этого класса – поддержка свойств сервлета вне зависимости от протокола связи. Класс **HttpServlet** тоже реализует интерфейс **Servlet**, но он ориентирован на обработку http-запросов.

Обратим внимание еще раз на принцип построения web-приложения. Стандартными (прописанными в спецификациях Java Servlet) являются только спецификации интерфейсов и классов API сервлетов. Реализацию же интерфейсов осуществляют классы, входящие в состав web-сервера или разработчик сервлета.

Выше уже отмечалось, что сервлет имеет API управляемый событиями. В реальности это означает, что основные методы сервлета вызываются web-контейнером. Для того чтобы web-сервер «узнал» о наличии сервлета в приложении ANaive, необходимо отразить это в дескрипторе развертывания приложения.

На рис. 3.2 представлен дескриптор развертывания web-приложения, содержащего сервлет с именем **Sss**. Комментариями выделен фрагмент текста, предназначенного для описания сервлета. Здесь используется две группы элементов, обозначенные тегами **<servlet>** и **<servlet-mapping>**.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  version="2.5">

  <display-name>ANaive</display-name>
  <!-- ----- начало описания сервлета ----- -->
  <servlet>
    <servlet-name>Sss</servlet-name>
    <servlet-class>Sss</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Sss</servlet-name>
    <url-pattern>/Sss</url-pattern>
  </servlet-mapping>
  <!-- ----- конец описания сервлета ----- -->

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Рис. 3.2. Пример описание сервлета Sss в дескрипторе развертывания приложения

В элементе **<servlet>** содержится информация об именах сервлета (вложенный тег **<servlet-name>**) и классе сервлета (вложенный тег **<servlet-class>**). Откомпилированный класс (в нашем случае файл **Sss.class**) должен находиться, как уже говорилось выше, в стандартной директории **Tomcat 6.0/webapps/WEB-INF/classes**. Имя, задаваемое тегом **<servlet-name>**, не обязательно должно совпадать с именем класса сервлета, как это сделано в примере, а может быть произвольным.

Элемент **<servlet-mapping>** указывает на суффикс URL (тег **<url-pattern>**), с помощью которого можно вызвать сервлет. В данном случае для вызова сервлета в адресной строке web-браузера нужно указать **http://xxx:8080/ANaive/Sss**, где xxx, как и прежде, – ip-адрес

или разрешаемое символическое имя компьютера с установленным сервером Tomcat. Указанная в элементе **<url-pattern>** строка не обязательно должна совпадать с именами сервлета или класса.

На рис. 3.3 приведен пример файла сборки **build.xml**, позволяющий скомпоновать web-приложение ANaive, имеющее в своем составе сервлет Sss. Структура этого файла в значительной степени совпадает со структурой файлов сборки в рассмотренных выше примерах. Поэтому рассмотрим только отличия.

```
<project name="ANaive" default = "copy" >

<target name="init">
    <echo message="Build ANaive Project" />
    <property name = "server"
        value = "C:\Program Files\Apache Software Foundation\Tomcat 6.0\" />
    <property name = "work"
        value = "D:\Workspace_ANaive\" />
</target>

<target name="compile" depends = "init">
    <delete verbose = "true">
        <fileset dir="${work}build"/>
    </delete>
    <javac srcdir      = "${work}src"
        destdir       = "${work}\build"
        classpath     = "${server}lib\servlet-api.jar"/>
</target>

<target name="create" depends = "compile">
    <delete dir= "${server}webapps\ANaive\" />
    <mkdir dir= "${server}webapps\ANaive\" />
    <mkdir dir= "${server}webapps\ANaive\WEB-INF" />
</target>

<target name="copy" depends = "create">
    <copy file= "${work}WEB-INF\web.xml"
        todir="${server}webapps\ANaive\WEB-INF\" />
    <copy file= "${work}html\index.html"
        todir="${server}webapps\ANaive\" />
    <copy todir="${server}webapps\ANaive\WEB-INF\classes">
        <fileset dir="${work}build"/>
    </copy>
</target>

</project>
```

Рис. 3.3. Пример файла сборки web-приложения

Прежде всего, в файле сборки появилась новая цель **compile**, которая использует новую задачу **javac**, позволяющую откомпилировать все java-файлы, которые находятся в директории, указанной атрибутом

**srcdir**, и поместить результаты компиляции (class-файлы) в директорию, указанную атрибутом **destdir**. Для компиляции сервлета необходима библиотека, поставляемая вместе с сервером Tomcat. Месторасположение этой библиотеки указывается атрибутом **classpath** задачи **javac**.

Для сокращенной записи имен директорий применена задача **property** (цель **init**). Применение этой задачи позволяет существенно упростить написание и сопровождение этого файла.

Кроме того, следует обратить внимание, что в рабочей директории **Workspace\_ANaive** появились две новые поддиректории: **src** – для хранения исходных java-файлов, **build** – для хранения откомпилированных class-файлов.

Как и в предыдущих примерах, последняя цель **copy** предназначена для копирования файлов web-приложения в специальную структуру директорий web-сервера Tomcat.

Для запуска приложения, как и прежде, можно использовать вызов **http://xxx:8080/ANaive/**. Это приведет к такому же результату, как и в примере, приведенном в разд. 2.

Для вызова сервлета необходимо использовать локатор **http://xxx:8080/ANaive/Sss**, суффикс и класс которого, напомним, указан в дескрипторе развертывания web-приложения. Результатом вызова будет пустой экран браузера (в примере сервлет ничего не пересылает браузеру). Текст, выводимый функциями **System.out.println** (вывод в стандартный поток), будет помещен в файл, который находится в директории **Tomcat 6.0\logs**. Имя файла, содержащее результат вывода в стандартный поток, имеет формат **stdout\_yyyymmdd.log**, где уууу – год, мм – месяц, dd – день месяца. Сразу после однократного вызова сервлета **Sss** содержимое этого файла будет примерно таким, как это показано на рис. 3.4

```
Sss:constructor  
Sss:init  
Sss:service:GET
```

Рис. 3.4. Содержимое журнала стандартного вывода после первого вызова сервлета Sss

После повторного вызова сервлета **Sss** независимо от того, было это выполнено с того же терминала или с другого, содержимое файла для стандартного вывода изменится примерно так, как это представлено на рис. 3.5. Следует обратить внимание, что инициализация класса (выполнение конструктора класса) и сервлета (метод



**init**) осуществляется только в том случае, если вызываемый сервлет до вызова не был загружен сервером либо по каким то причинам (например, из-за нехватки оперативной памяти) был в момент вызова выгружен.

```
Sss:constructor  
Sss:init  
Sss:service:GET  
Sss:service:GET
```

Рис. 3.5. Содержимое журнала стандартного вывода после второго вызова сервлета Sss

Если же после второго вызова сервлета остановить web-сервер и снова распечатать этот же файл стандартного вывода, то результат будет похожим на рис. 3.6, т. е. перед окончанием работы web-сервер разрушает сервлет, предварительно вызвав его метод **destroy**.

```
Sss:constructor  
Sss:init  
Sss:service:GET  
Sss:service:GET  
Sss:destroy
```

Рис. 3.6. Содержимое журнала стандартного вывода после остановки web-сервера

### 3.3. Обработка запросов и ответов HTTP

В примере сервлета, который рассматривался выше, метод **service** получает управление после того, как http-запрос клиента поступает на обработку в сервлет. Для того чтобы определить тип запроса протокола HTTP, использованный клиентом для вызова сервлета, можно воспользоваться методом **getMethod** объекта класса **HttpServletRequest**. Объект этого класса, как уже отмечалось выше, создается web-контейнером и передается методу **service** в качестве первого параметра. Вторым параметром метода **service** является объектом класса **HttpServletResponse** и тоже формируется web-контейнером, он предназначен для подготовки и пересылки http-ответа клиенту.

Класс **HttpServlet** имеет собственную реализацию метода **service**, который в зависимости от типа http-запроса передает управление в один из следующих переопределяемых методов: **doGet**, **doPost**, **doPut**,

**doDelete, doHead, doOptions, doTrace.** Перечисленные методы соответствуют различным типам http-запросов. Наиболее часто используемые типы запросы в web-приложениях – это **GET** и **POST**. Именно обработка только этих запросов будет рассматриваться ниже. С полным перечнем http-запросов, их назначением и способом применения можно ознакомиться в документе RFC 2068, который доступен на официальном сайте организации-разработчика Internet Engineering Task Force [24].

### 3.3.1. Обработка http-запросов типа GET

Тип **GET** http-запроса используется по умолчанию и часто применяется клиентскими приложениями (как правило, это web-браузер) в том случае, если необходимо просто доставить html-файл, находящийся на web-сервере в распоряжении клиентского приложения. В простейшем случае этот файл может быть статическим, который просто перемещается из директории сервера в адрес клиента. В другом случае пересылаемый файл может быть сформирован динамически, например на основе запроса к базе данных. GET-запрос может сопровождаться параметрами, которые уточняют этот запрос.

Самый простой способ продемонстрировать GET-запрос – с помощью адресной строки web-браузера запросить какой-нибудь ресурс в сети Internet. Обратите внимание, что при вызове сервлета **Sss**, который рассматривался в примерах выше, в методе **service** мы обнаруживали с помощью функции **getMethod** запрос типа **GET**. Это происходило, потому что web-браузер использует метод **GET** по умолчанию при первоначальном вызове ресурса.

На рис. 3.7 приведен пример сервлета, обрабатывающего html-запрос **GET**. Сервлет реализован в виде класса **Ggg**, который, как и прежде, расширяет класс **HttpServlet** и реализует интерфейс **Servlet**. Класс не реализует, как это сделано в предыдущих примерах, методы **init**, **destroy** и **service** – все они реализуются классом **HttpServlet**, но используется метод **doGet**, унаследованный от того же **HttpServlet**. Строки функции **doGet** для удобства пронумерованы.

Функция **doGet** получает управление из функции **service**, реализованной классом **HttpService**, в том случае, если для вызова сервлета используется http-запрос типа **GET**. При этом параметры, полученные функцией **service**, без изменения передаются в функцию **doGet**. Кроме того, по сравнению с предыдущими примерами, в классе **Ggg** отсутствует конструктор. Это допускается правилами языка Java, в этом случае он генерируется компилятором автоматически.

```

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Ggg extends HttpServlet implements Servlet {
    protected void doGet(HttpServletRequestRequest rq, HttpServletResponse rs)
        throws ServletException, IOException {
        String parm = rq.getParameter("page");           // 1
        System.out.println("Ggg:doGet:page=" + parm);    // 2
        rs.setContentType("text/html");                 // 3
        PrintWriter pw = rs.getWriter();                 // 4
        pw.println("<html> "                             // 5
            + "<body> "                                     // 6
            + "Hello from Servlet"                       // 7
            + "<br>Ggg:doGet:page=" + parm                 // 8
            + "</body>"                                     // 9
            + "</html>");                                  // 10
        pw.close();                                     // 11
    }
}

```

Рис. 3.7. Пример сервлета, обрабатывающего GET-запрос

В рассматриваемом примере сервлет выводит два сообщения: одно, как и прежде, в журнал для стандартного вывода (строка 2), другое – в адрес web-браузера (строки 5–10), который будет пытаться интерпретировать сообщение как html-файл. Для получения значения параметра с именем **page** применяется метод **getParameter** интерфейса **HttpServletRequest** (строка 1).

Для вывода данных на web-браузер используется метод **println** класса **PrintWriter** (строки 5–10). Объект **PrintWriter** создается web-сервером, а ссылка на него может быть получена с помощью метода **getWriter** интерфейса **HttpServletResponse** (строка 4). Для того чтобы web-браузер смог правильно распорядиться данными, с помощью метода **setContentType** устанавливается MIME-тип данных (строка 3) ответа. С полный список MIME-типов можно ознакомиться в документации по HTTP [24]. Завершается вывод вызовом метода **close** объекта **PrintWriter** (строка 11).

На рис. 3.8 приведен фрагмент описания двух сервлетов в дескрипторе развертывания web-приложения. Следует обратить внимание на элемент **<url-pattern>** для сервлета **Ggg** – суффикс URL не совпадает с именем сервлета и класса.

Для того чтобы включить сервлет в приложение **ANaive**, необходимо поместить файл **Ggg.java** в директорию **Workspace\_ ANiave/scr**, откорректировать файл **web.xml** и запустить утилиту **Ant**. При этом файл **build.xml** необходимо оставить без изменения, т. е. таким же, как в предыдущих примерах.

```

<!-- ..... -->

<servlet>
    <servlet-name>Sss</servlet-name>
    <servlet-class>Sss</servlet-class>
</servlet>
<servlet>
    <servlet-name>Ggg</servlet-name>
    <servlet-class>Ggg</servlet-class>
</servlet>

    <servlet-mapping>
        <servlet-name>Sss</servlet-name>
        <url-pattern>/Sss</url-pattern>
    </servlet-mapping>
<servlet-mapping>
    <servlet-name>Ggg</servlet-name>
    <url-pattern>/GetExample</url-pattern>
</servlet-mapping>

<!-- ..... -->

```

Рис. 3.8. Фрагмент дескриптора развертывания web-приложения с двумя сервлетами

Для вызова сервлета Ggg требуется набрать в адресной web-браузера **http://xxx:8080/ANaive/GetExample**. При этом параметр **page** и его значение указывается в адресной строке так, как это показано в примере на рис. 3.9. Если требуется указать несколько параметров, то они разделяются символом амперсанда.

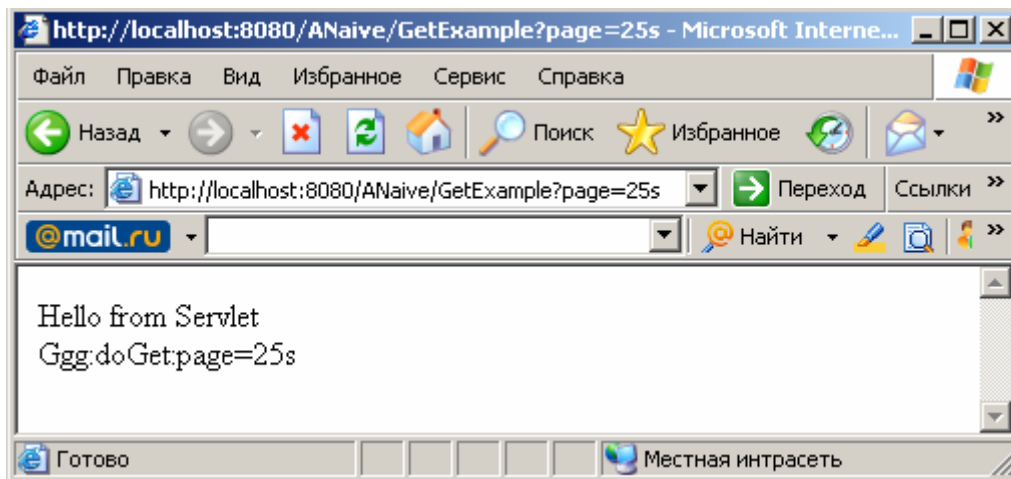


Рис. 3.9. Результат вызова сервлета Ggg

После добавления сервлета **Ggg** приложение ANaive содержит три независимые компоненты: два сервлета и один html-файл. Как

правило, web-приложение имеет иерархическую структуру и предполагает, что у него есть стартовая страница, с которой можно вызвать другие компоненты. На рис 3.10 изображен пример иерархической структуры приложения ANaive. Эта структура подразумевает наличие стартовой страницы **index.html**, с которой можно вызвать два других компонента приложения. Причем при вызове сервлета **Ggg** ему передаются два параметра: **page** и **name**.

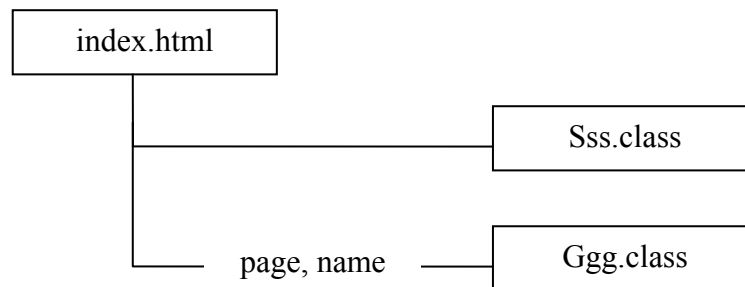


Рис. 3.10. Связь между компонентами приложения

Для вызова сервлетов со html-страницы можно использовать теги **<a>**, определяющие гиперссылки на сервлеты. На рис 3.11 представлен пример файла **index.html**, имеющего две гиперссылки для вызова сервлетов **Sss** и **Ggg**.

```

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>My First Web-Application</title>
  </head>
  <body>
    Hello, World!
    <br><a href = "Sss">Go to Sss</a>
    <br><a href = "GetExample?page=27s&name=Vlad"> Go to Ggg </a>
  </body>
</html>
  
```

Рис. 3.11. Вызов сервлетов с помощью гиперссылок

Для вызова сервлета с помощью гиперссылки необходимо указать соответствующее ему значение элемента **<url-pattern>** из дескриптора развертывания приложения в атрибуте **href** тега **<a>**. В примере на рис. 3.11 при вызове сервлета **Ggg** передаются два параметра

с именами **page** и **name**. Значение параметра **page**, как и прежде, осталось 27s, а параметру **name** присвоено значение Vlad. Следует обратить внимание, что параметры отделены от URL запроса символом вопроса, а между параметрами устанавливается амперсанд. Последовательность символов, находящаяся справа от знака вопроса, часто называют *строкой запроса*.

На рис. 3.12 приведен пример сервлета **Ggg**, получающего два параметра (строки 1–2) и выводящего html-сообщение в адрес http-клиента. Кроме того, здесь используются несколько полезных методов интерфейса **HttpServletRequest** и класса **HttpServlet**. Эти методы позволяют получить информацию о запросе ip-адреса клиентского хоста (строка 11); суффиксе URL, соответствующего сервлету (строка 12); имени web-сервера (строка 13); URL приложения (строка 14); имени сервлета (15).

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Ggg extends HttpServlet implements Servlet {
    protected void doGet(HttpServletRequest rq, HttpServletResponse rs)
        throws ServletException, IOException {
        String parm1 = rq.getParameter("page");           // 1
        String parm2 = rq.getParameter("name");           // 2
        System.out.println("Ggg:doGet:page=" + parm1);     // 3
        System.out.println("Ggg:doGet:name=" + parm2);     // 4
        rs.setContentType("text/html");                   // 5
        PrintWriter pw = rs.getWriter();                  // 6
        pw.println("<html><body>"                          // 7
            +"Hello from Servlet"                          // 8
            +"<br>Ggg:doGet:                                page=" + parm1 // 9
            +"<br>Ggg:doGet:                                name=" + parm2 // 10
            +"<br>Ggg:getRemoteHost:                        "+rq.getRemoteHost() // 11
            +"<br>Ggg:getServletPath:                        "+rq.getServletPath() // 12
            +"<br>Ggg:getServerName:                         "+rq.getServerName() // 13
            +"<br>Ggg:rq.getContextPath:                    "+rq.getContextPath() // 14
            +"<br>Ggg:getServletName:                       "+this.getServletName() // 15
            +"</body></html>"); // 16
        pw.close(); // 17
    }
}
```

Рис. 3.12. Пример сервлета, обрабатывающего два параметра GET-вызова

Если осуществить все разобранные выше изменения в файлах **index.html** и **Ggg.java** и выполнить утилиту **Ant**, то вызов приложения ANaive с помощью web-браузера должен привести к результату, как на рис. 3.13.

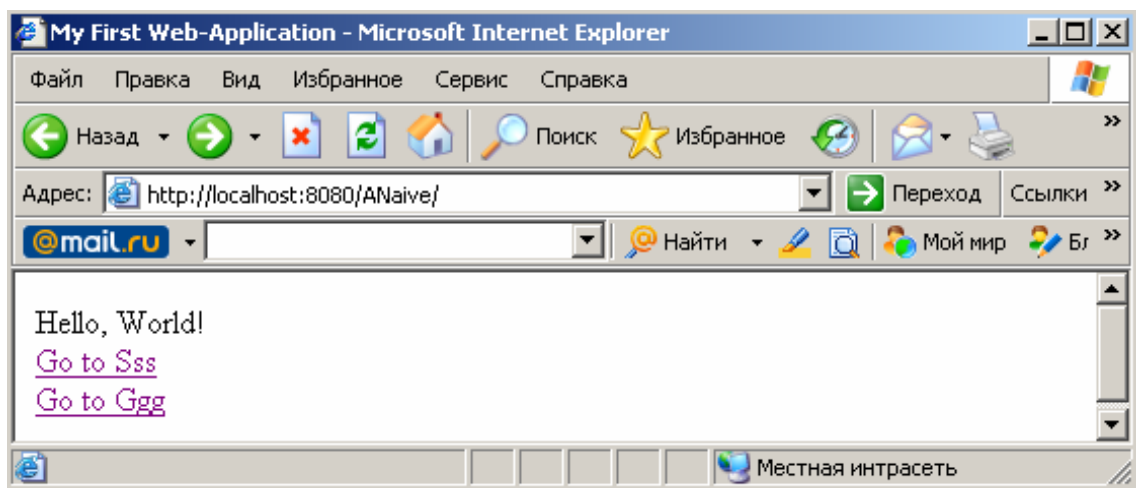


Рис. 3.13. Результат вызова приложения ANaive с помощью браузера Internet Explorer

Переход по гиперссылке **Go to Ggg** приведет к выводу на экран браузера сообщения, как на рис. 3.14.

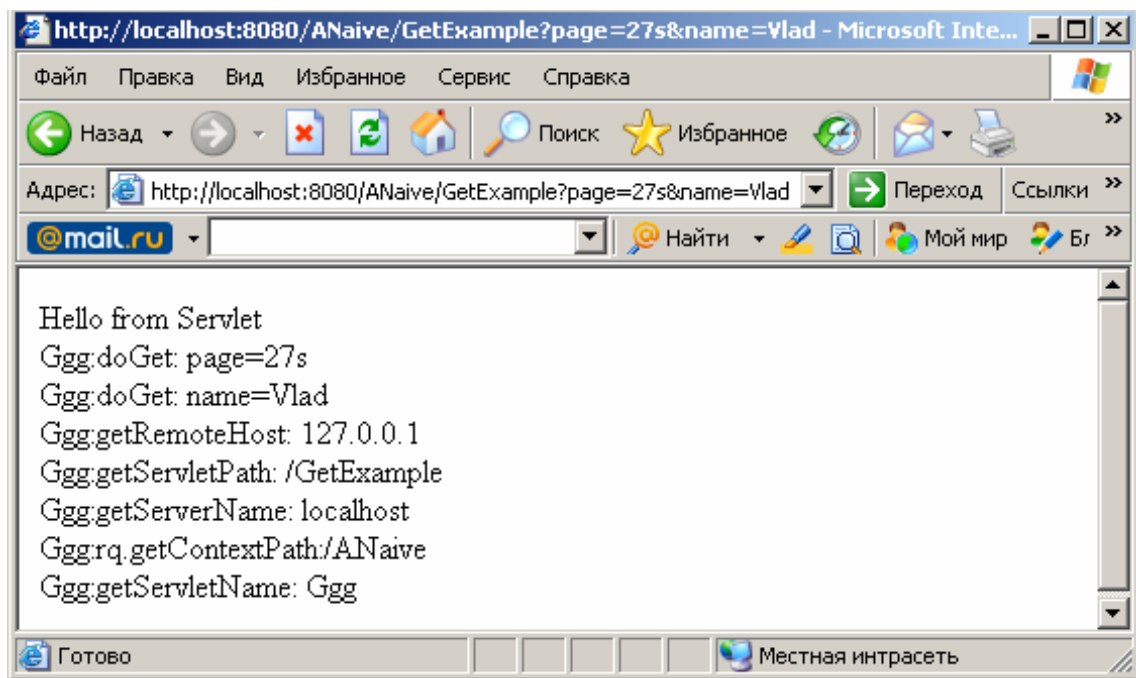


Рис. 3.14. Результат вызова сервлета Ggg

Следует еще раз обратить внимание на URL при вызове сервлета – он содержит наименование протокола связи (http), символическое имя или ip-адрес (localhost), номер порта (8080), имя приложения (совпадает с наименованием директории ANaive), значение элемента

**<url-pattern>** для данного сервлета из дескриптора развертывания приложения (GetExample) и строку запроса с заданными параметрами (page и name).

### 3.3.2. Обработка http-запросов типа POST

Разработка сервлета, обрабатывающего http-запрос **POST**, принципиально ничем не отличается от разработки сервлета, обрабатывающего запрос **GET**. Для генерации запроса **POST**, как правило, применяется html-форма, а обработка этого запроса осуществляется в методе **doPost** вызываемого сервлета. Параметры в этом случае не передаются как часть URL, а содержатся в теле http-запроса.

На рис. 3.15 приводится пример html-формы, генерирующей запрос **POST** для вызова сервлета **PPP**. Тег **<form>**, определяющий начало формы, содержит два атрибута: **action** и **method**.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                        "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Person data</title>
</head>
<body>
  <form action="PostExample" method = "post">
    <table>
      <tr> <td><label for = "firstname">firstname</label> </td>
        <td> <input type = "text" name="firstname" size = "20"> </td>
      </tr> <tr> <td><label for = "lastname">lastname</label></td>
        <td><input type = "text" name="lastname" size = "20"></td>
      </tr> <tr> <td><label for = "password">password</label></td>
        <td><input type = "password" name="password" size = "20"></td>
      </tr> <tr> <td>sex</td>
        <td> <input type = "radio" name="sex" value = "male" checked
          = "checked">male
          <input type = "radio" name="sex" value = "female">female </td>
      </tr>
    </table>
    <br> <input type = "submit" name="press" value = "OK" size = "20">
      <input type = "submit" name= "press" value = "Cancel" size = "20">
    </form>
  </body>
</html>
```

Рис. 3.15. Пример html-файла с формой, генерирующей POST-запрос

Атрибут **action** определяет ссылку, куда должны быть пересланы для обработки данные из формы. В нашем случае эта ссылка должна совпадать со значением элемента **<url-pattern>** дескриптора развертывания приложения. Второй атрибут **method** управляет методом пе-



редачи данных программе – обработчику и может принимать два значения: **get** или **post**, что соответствует рассматриваемым здесь типам http-запросов.

Внутри элемента **<form>** расположены 7 элементов **<input>**, которые предназначены для ввода информации. Следует обратить внимание на то, что для всех элементов задан атрибут **name**, а также на то, что для некоторых элементов значение этого атрибута совпадает. Именно значения атрибутов **name** передаются в http-запросе в качестве имен параметров. Значение самих передаваемых параметров определяется в зависимости от значения атрибута **type**. Так, например, значением параметра с именем **firstname** будет последовательность символов, которая была введена пользователем в соответствующее поле на экране web-браузера. Значением параметра с именем **press** (два элемента **<input type="submit">** имеют атрибут **name** с таким значением) будет **OK** или **Cancel** в зависимости от того, какая из изображенных в браузере кнопок будет нажата. Аналогичным образом будет определяться значение параметра **sex**.

Для определенности назовем html-файл, содержимое которого представлено на рис. 3.15, именем **persondata.html**, сервлет и его класс – **Ppp**, а также включим эти два компонента в состав приложения **Anaïve**, как это показано на схеме, изображенной на рис. 3.16, которая является продолжением рис. 3.10.

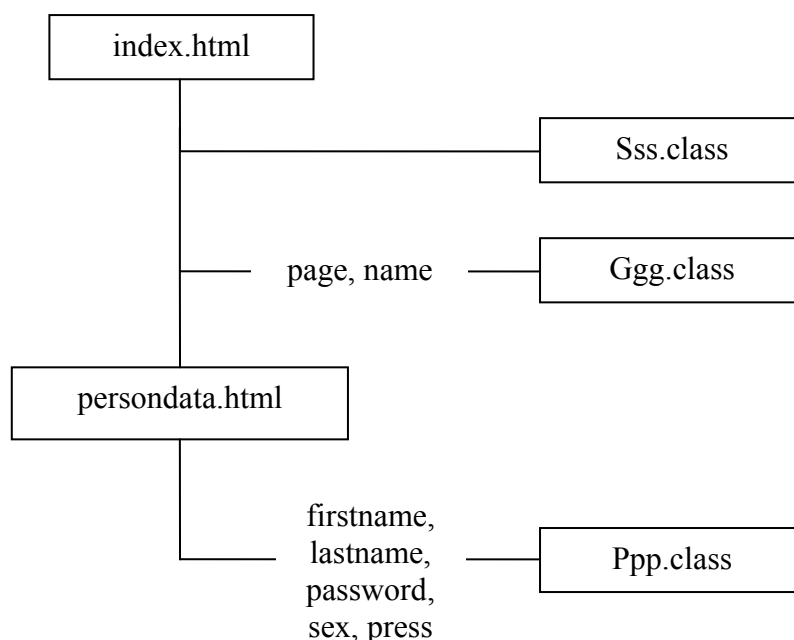


Рис. 3.16. Связь между компонентами приложения

В соответствии со схемой на рис. 3.16, страница **persondata.html** должна вызываться из страницы **index.html**. Проще всего это можно сделать с помощью гиперссылки (тега **<a>**), как поступали раньше. На рис. 3.17 представлен фрагмент файла **index.html** с добавленной гиперссылкой для вызова **persondata.html**.

```
<!-- ..... -->
<body>
    Hello, World!
    <br><a href = "Sss">Go to Sss</a>
    <br><a href = "GetExample?page=27s&name=Vlad"> Go to Ggg </a>
    <br><a href = "persondata.html"> Go to persondata </a>
</body>
<!-- ..... -->
```

Рис. 3.17. Фрагмент файла index.html с гиперссылкой на persondata.html

Сервлет **Ppp**, как и все другие сервлеты, должен быть описан в дескрипторе развертывания приложения. Атрибут **action** тега **<form>** указывает, что в описании сервлета должен присутствовать элемент **<url-pattern>** с таким же значением **PostExample**. На рис. 3.18 представлен фрагмент файла **web.xml** приложения ANaive с описанием сервлета **Ppp**.

```
<!-- ..... -->
<servlet>
    <servlet-name>Ppp</servlet-name>
    <servlet-class>Ppp</servlet-class>
</servlet>
<!-- ..... -->
<servlet-mapping>
    <servlet-name>Ppp</servlet-name>
    <url-pattern>/PostExample</url-pattern>
</servlet-mapping>
<!-- ..... -->
```

Рис. 3.18. Описание сервлета Ppp в файле web.xml

Пример класса сервлета **Ppp** представлен на рис. 3.19. По своему построению принципиально он отличается от сервлета **Ggg** только тем, что обработка запроса осуществляется в методе **doPost**.

Для выполнения сборки приложения ANaive с новыми компонентами – страницей **persondata.html** и сервлетом **Ppp** – необходимо модифицировать файл сборки **build.xml**. Изменению подлежит только одна задача **copy** в цели с таким же названием. Дело в

том, что вторая задача **copy** копирует только один файл из директории **Workspace\_ANaive/html**. Изменим цель **copy** так, так это показано на рис. 3.20. Выполнение ее приведет к копированию всех файлов директории **Workspace\_ANaive/html** в соответствующую директорию сервера **Tomcat**.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Ppp extends HttpServlet implements Servlet {
    protected void doPost(HttpServletRequest rq, HttpServletResponse
rs)
        throws ServletException, IOException {
        String firstname = rq.getParameter("firstname");
        String lastname = rq.getParameter("lastname");
        String password = rq.getParameter("password");
        String sex = rq.getParameter("sex");
        String press = rq.getParameter("press");
        rs.setContentType("text/html");
        PrintWriter pw = rs.getWriter();
        pw.println("<html> " + "<body> "
            + "<br>Ppp:doPost:firstname=" + firstname
            + "<br>Ppp:doPost:lastname=" + lastname
            + "<br>Ppp:doPost:password=" + password
            + "<br>Ppp:doPost:sex=" + sex
            + "<br>Ppp:doPost:press=" + press
            + "</body>" + "</html>");
        pw.close();
    }
}
```

Рис. 3.19. Пример сервлета, обрабатывающего  
html-запрос типа POST

```
<project name="ANaive" default = "copy" >
    <!-- ..... -->
    <target name="copy" depends ="create">
        <copy file= "${work}WEB-INF\web.xml"
            todir="${server}webapps\ANaive\WEB-INF" />
        <copy todir="${server}webapps\ANaive\">
            <fileset dir="${work}html" />
        </copy>
        <copy todir="${server}webapps\ANaive\WEB-INF\classes">
            <fileset dir="${work}build"/>
        </copy>
    </target>
</project>
```

Рис. 3.20. Фрагмент файла сборки, обеспечивающий  
копирование всех html-файлов

После сборки приложения ANaive вызов его привет к результату, похожему на рис. 3.21.

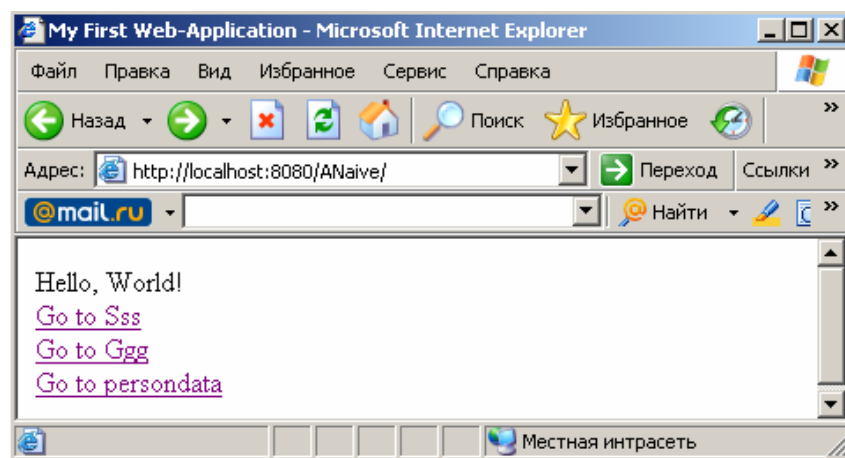


Рис. 3.21. Результат вызова приложения ANaive после добавления новой гиперссылки

Если перейти по ссылке **Go to persondata**, то сервер отправит клиенту файл **persondata.html**, а web-браузер проинтерпретирует его примерно так, как на рис. 3.22.

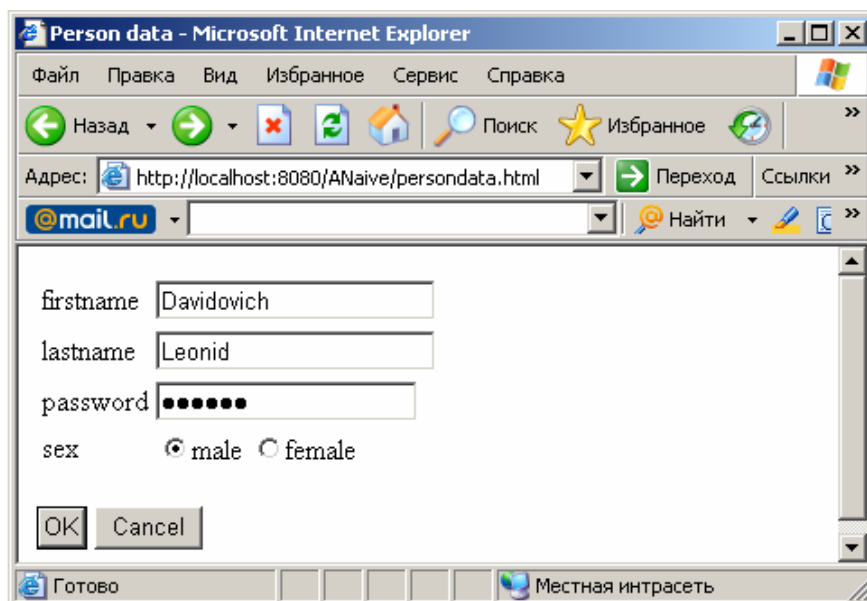


Рис. 3.22. Страница persondata.html, отображенная web-браузером

После заполнения полей для ввода (например, как на рис. 3.22) и нажатия клавиши **OK** произойдет генерация web-браузером запроса **POST**, вызывающего сервлет **Ppp**. Сервлет выведет в окно web-браузера

результат своей работы, в нашем случае это значения полученных параметров (рис. 3.23).

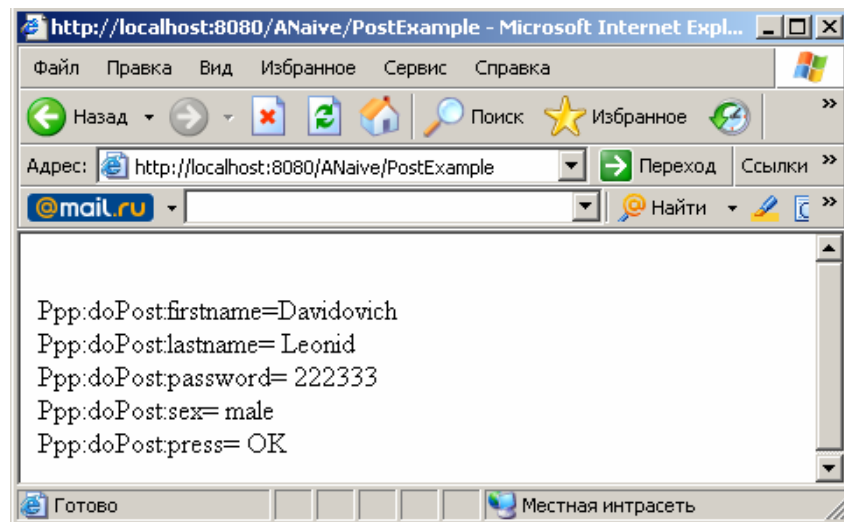


Рис. 3.23. Результат работы сервлета Ppp после запроса типа POST

Следует обратить внимание на то, что параметры не отображаются в URL запроса, а передаются в теле запроса. Кроме того, значение параметра **press** (ему соответствуют два элемента `<input type="submit">` в файле **persondata.html**) равно значению атрибута **value**, выбранного элемента. По этой же причине значение параметра **sex** равно **male**.

Если в файле **persondata.html** изменить атрибут **method** элемента **form** на значение **get**, в файле **Ppp.java** имя функции **doPost** на **doGet**, пересобрать и перезапустить приложение, то конечный результат будет похожим на рис. 3.24.

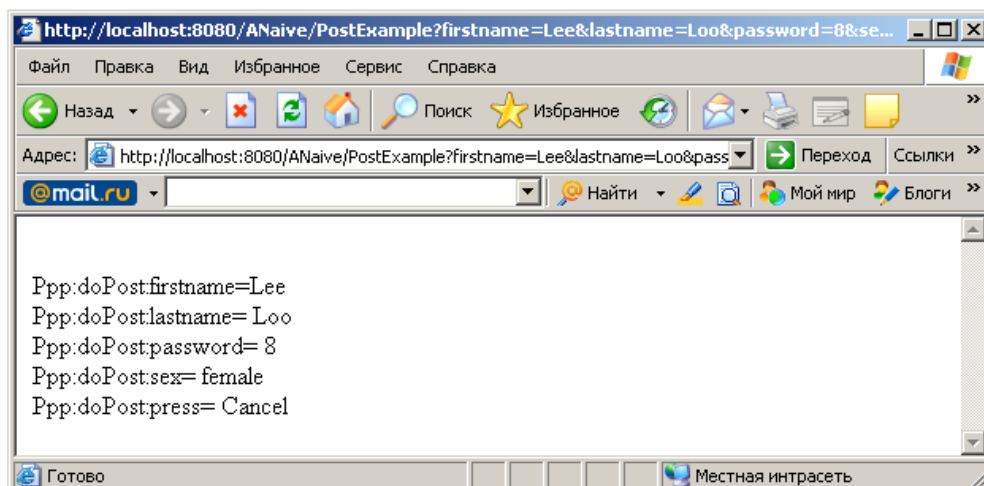


Рис. 3.24. Результат работы сервлета Ppp после запроса типа GET

Обратите внимание на адресную строку web-браузера, она содержит строку запроса с параметрами, имена которых совпадают со значениями атрибутов **name** в элементах формы. Кроме того, при этом вызове сервлета введены другие значения в текстовые поля, а также выбрано значение переключателя **female** и нажата клавиша **Cancel**.

### 3.3.3. Обработка http-запросов типа GET и POST в одном сервлете

Часто бывает необходимым разработать сервлет, который бы мог обрабатывать и **GET**-, и **POST**-запросы одновременно. При этом могут быть два подхода: 1) сервлет должен работать одинаково независимо от типа запроса; 2) сервлет должен по-разному обрабатывать разные запросы.

В первом случае правильным будет использовать метод **service**, хотя это не единственный способ решения задачи. Метод **service**, как уже говорилось выше, является методом интерфейса **Servlet** и должен быть обязательно реализован любым сервлетом. В предыдущих примерах сервлеты **Ggg** и **Ppp** использовали реализацию метода **service**, унаследованную от класса **HttpServlet**. Реализация **service** классом **HttpServlet** является очень простой: метод определяет тип http-запроса и передает управление обработчику запроса данного типа. В нашем случае обработчиками запросов являются методы **doGet** и **doPost**. Установить тип запроса в **service** всегда можно с помощью метода **getMethod**, класса **HttpServletRequest**, как это сделано в примере на рис. 3.1. Таким образом, если необходимо, чтобы сервлет одинаково реагировал на **GET** и **POST**, необходимо запросы обрабатывать в методе **service**. Например, для того, чтобы сервлет **Ppp** (рис. 3.19) стал обладать таким свойством, необходимо изменить имя функции **doPost** на **service** и снова выполнить сборку.

Во втором случае просто в одном сервлете можно реализовать два одинаковых обработчика запросов: **doGet** и **doPost**.

### 3.3.4. Переопределение http-запроса типа GET

Не всегда правильно формировать html-текст ответа непосредственно в теле сервлета, как это сделано в предшествующих примерах. Такой подход оправдан только в тех случаях, когда сервлет должен генерировать специфические ответы, например, при использовании технологии Ajax [25]. Чаще для ответов используются html или jsp-страницы. Применение технологии Java Server Page будет рассматриваться в следующей главе, а здесь разберем пример применения html-страницы для генерации сервлетом ответа.

Пусть в результате обработки сервлетом http-запроса в окне браузера должна отобразиться html-страница. Иначе говорят, что сервлет должен переопределить запрос на заданную html-страницу. Эту можно сделать, применив метод **forward** класса **Request Dispatcher**. Объект этого класса создается сервером, а ссылка на него становится доступной сервлету при помощи метода **getRequestDispatcher** класса **HttpServletRequest**.

Предположим, что в результате обработки запроса типа **GET** сервлет **Ppp** (рис. 3.19) должен отобразить в окне браузера страницу **male.html** в том случае, если на форме **persondata.html** переключатель **sex** был установлен в состояние **male**, а в другом случае должна быть выведена **female.html**.

На рис. 3.25 приведен фрагмент текста метода-обработчика запроса (**doGet** или **service**), который использует вызов html-страниц. Если заменить содержимое обработчика запроса в сервлете **Ppp**, который рассматривался выше на предлагаемый текст, подготовить файлы **male.html** и **female.html**, а также выполнить сборку приложения, то при нажатии клавиши **OK** на html-форме (рис. 3.21) будет осуществляться вызов **male.html** или **female.html** в зависимости от значения переключателя **sex**. Если же выбрана для нажатия клавиша **Cancel**, то вызывается исходная страница **index.html**.

```
//.....
RequestDispatcher rd = null;
String sex          = rq.getParameter("sex");
String press        = rq.getParameter("press");
if (press.equalsIgnoreCase("OK")) {
    if (sex.equalsIgnoreCase("male"))
        rd = rq.getRequestDispatcher("/male.html");
    else if (sex.equalsIgnoreCase("female"))
        rd = rq.getRequestDispatcher("/female.html");
} else
    rd = rq.getRequestDispatcher("/index.html");
rd.forward(rq, rs);
//.....
```

Рис. 3.25. Фрагмент сервлета, переопределяющего html-запрос

Аналогичным способом запрос можно переопределить на другой сервлет, имеющий обработчик **doGet**. На рис. 3.26 представлен фрагмент метода **doGet** сервлета, переопределяющего запрос типа **GET** на сервлеты **Mmm** или **Fff** в зависимости от состояния переключателя **sex** или на страницу **index.html** при нажатии клавиши **Cancel**.

```
// .....
String firstname      = rq.getParameter("firstname");
String lastname       = rq.getParameter("lastname");
String sex            = rq.getParameter("sex");
String press          = rq.getParameter("press");
RequestDispatcher rd = null;
if (press.equalsIgnoreCase("OK")) {
    String parmstr = "firstname=" + firstname + "&"
        + "lastname=" + lastname;
    if (sex.equalsIgnoreCase("male"))
        rd = rq.getRequestDispatcher("/Mmm?" + parmstr);
    else if (sex.equalsIgnoreCase("female"))
        rd = rq.getRequestDispatcher("/Fff?" + parmstr);
} else
    rd = rq.getRequestDispatcher("/index.html");
rd.forward(rq, rs);
// .....

```

Рис. 3.26. Фрагмент сервлета, переопределяющего запрос на другой сервлет

В качестве параметра функции **getRequestDispatcher** передается суффикс URL, значение которого представляет конкатенацию элемента **<url-pattern>**, описывающего соответствующий сервлет в дескрипторе развертывания приложения и динамически формируемая строка запроса, позволяющая передать параметры.

### 3.3.5. Формирование http-запроса в сервлете

Иногда требуется в сервлете выполнить html-запрос для вызова другого сервлета или jsp-страницы. При этом вызываемый объект не обязательно должен находиться (и соответственно исполняется) на том же web-сервере, что и вызывающий сервлет. Такой вызов принципиально отличается от переопределения запроса, рассмотренного выше.

На рис. 3.27. изображены схемы взаимодействия сервлетов в двух случаях: при переопределении запроса (а) и при генерации запроса (б). Пунктирными линиями на рисунке изображаются запросы, а сплошными — движение данных ответа. В обоих случаях предполагается, что браузер вызывает сервлет **Rpp**, который в зависимости от значения передаваемого параметра в первом случае переопределяет запрос на сервлеты **Mmm** или **Fff**, а во втором — генерирует http-запрос, вызывающий **Mmm** или **Fff**.

Обратите внимание, что при переопределении запроса (а), связь браузера с сервлетом **Rpp** фактически прекращается и данные ответа сервлетов **Mmm** или **Fff** поступают на обработку ему напрямую. Если бы, например, **Mmm** в свою очередь переопределял запрос на другой сервлет, то данные ответа очередного сервлета в цепочке переопределений все равно бы поступили непосредственно браузеру. Более того,



попытка вывода ответа сервлетом переопределяющим запрос вызовет ошибку исполнения.

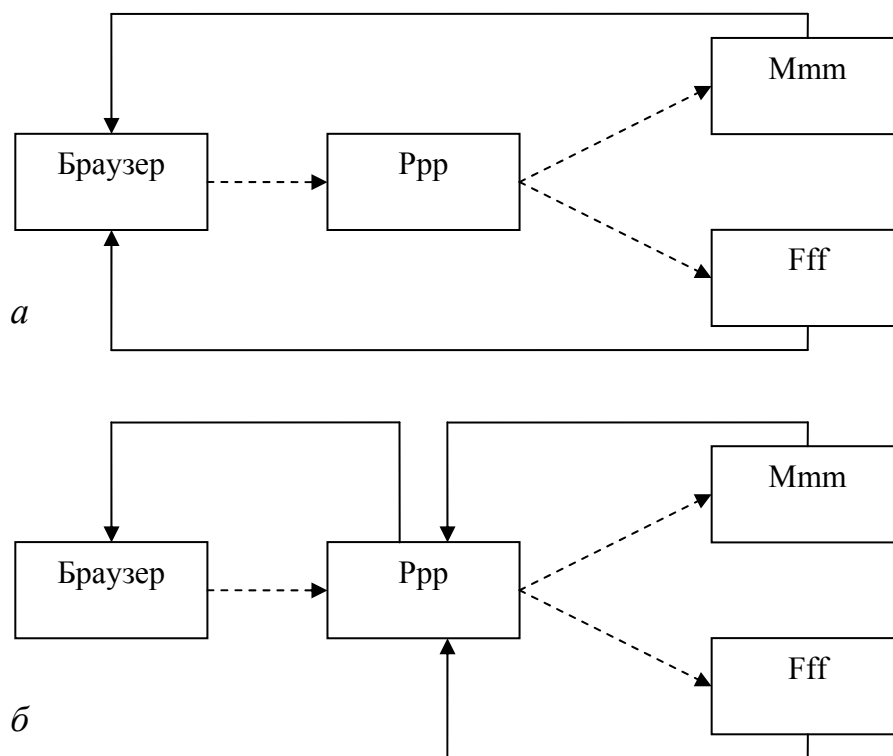


Рис. 3.27. Схемы взаимодействия сервлетов:  
*a* – при переопределении запроса; *б* – при генерации запроса

В случае генерации запроса (*б*) ответы сервлетов **Mmm** и **Fff** поступают и обрабатываются в вызывающем сервлете **Ppp** и только сгенерированный **Ppp** ответ может быть интерпретирован браузером.

Для генерации http-запросов будем использовать пакет **org.apache.commons.httpclient** (далее **httpclient**), который всегда можно скачать с сайта [26]. На момент подготовки пособия доступной была версия 3.1 пакета, выгружаемая в виде файла **commons-httpclient-3.1.jar**. В процессе своей работы классы **httpclient** вызывают методы еще двух пакетов – **org.apache.commons.logging** и **org.apache.commons.codec**. Скачать jar-файлы, содержащие эти пакеты, можно с того же сайта [26]. Там же есть исчерпывающая информация о назначении и применении этих пакетов.

Для того чтобы пакеты были доступны для всех приложений web-сервера, необходимо поместить их в директорию **Tomcat6.0/common/lib**. На рис. 3.28 приведен результат выполнения команды **dir** для этой директории после того, как туда были помещены jar-файлы с новыми пакетами.

Содержимое папки C:\Program Files\Apache Software Foundation\Tomcat 6.0\common\lib									
14.09.2008	18:03	<DIR>	.						
14.09.2008	18:03	<DIR>	..						
10.07.2004	16:13		46	725	commons-codec-1.3.jar				
18.08.2007	11:00		305	001	commons-httpclient-3.1.jar				
22.11.2007	00:28		141	401	commons-logging-1.1.1-javadoc.jar				
22.11.2007	00:28		74	976	commons-logging-1.1.1-sources.jar				
22.11.2007	00:28		60	841	commons-logging-1.1.1.jar				
22.11.2007	00:28		26	520	commons-logging-adapters-1.1.1.jar				
22.11.2007	00:28		52	313	commons-logging-api-1.1.1.jar				
22.11.2007	00:28		111	279	commons-logging-tests.jar				
	8 файлов		819	056	байт				
	2 папок	3	511	332	864	байт	свободно		

Рис. 3.28. Состав файлов в директории Tomcat 6.0\common\lib, необходимый для работы пакета org.apache.commons.httpclient

Кроме того, перед запуском приложения следует проверить установку (и если необходимо, то выполнить ее) параметра **common.loader** в конфигурационном файле **Tomcat 6.0\conf\catalina.properties**. Именно этот параметр указывает загрузчику классов сервера Tomcat их месторасположение. В комментариях файла **catalina.properties** написана инструкция по установке параметров и приведены примеры.

На рис. 3.29 представлен пример сервлета **Sss**, генерирующего запрос типа **POST**, вызывающий сервлет **Ppp**.

Для генерации http-запроса в рассматриваемом примере используются объекты трех классов: **HttpClient** (строка 1), **PostMethod** (строки 2–3), **NameValuePair** (строки 4–6). При создании объекта класса **PostMethod** в параметре конструктора указывается URL вызываемого серверного объекта (в нашем случае это сервлет **Ppp**). Созданный массив объектов класса **NameValuePair** предназначен для хранения списка пар «параметр – значение», которые будут передаваться вызываемому объекту в теле http-запроса. Кроме того, значение еще одного параметра указывается в строке запроса URL точно так, как это делалось раньше в запросе типа **GET**. Метод **addParameters** (строка 7) позволяет указать месторасположение передаваемых в запросе параметров, а метод **executeMethod** (строка 8) – выполнить запрос. Результат выполнения запроса проверяется с помощью метода **getStatusCode** – при успешном завершении он возвращает значение **SC\_OK** (строка 9). Вызванный сервлет **Ppp** на полученный запрос генерирует http-ответ, тело которого считывается и обрабатывается в вызывающем сервлете **Sss**. Для считывания ответа используется класс **InOutServlet** (строки 11–13), пример реализации которого приводится на рис. 3.30.

```

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.commons.httpclient.*;
import org.apache.commons.httpclient.methods.*;

public class Sss extends HttpServlet implements Servlet {
    protected void service(HttpServletRequest rq, HttpServletResponse rs)
        throws ServletException, IOException {
        HttpClient hc = new HttpClient(); // 1
        PostMethod pm = new PostMethod("http://localhost:8080" // 2
            + rq.getContextPath() + "/PostExample?press=OK"); // 3
        NameValuePair[] parms = { new NameValuePair("firstname", "Gurin"), // 4
            new NameValuePair("lastname", "Nik"), // 5
            new NameValuePair("sex", "male") }; // 6
        pm.addParameters(parms); // 7
        hc.executeMethod(pm); // 8
        if (pm.getStatusCode() == HttpStatus.SC_OK) { // 9
            InOutServlet inout = new InOutServlet(rs.getOutputStream(), // 10
                pm.getResponseBodyAsStream()); // 11
            inout.perform(); // 12
        } else // 13
            System.out.println("Sss:service:getStatusCode()=" // 14
                + pm.getStatusCode()); // 15
        } // 16
    }
}

```

Рис. 3.29. Пример сервлета, генерирующего http-запрос типа POST

```

import java.io.*;
import javax.servlet.*;
public class InOutServlet {
    protected InputStream ist;
    protected ServletOutputStream ost;
    public InOutServlet( ServletOutputStream ost, InputStream ist){
        this.ist = ist;
        this.ost = ost;
    }
    public void perform() throws IOException{
        int buf;
        while ((buf = this.ist.read()) > 0)
            this.ost.write(buf);
        this.ost.flush();
    }
}

```

Рис. 3.30. Пример реализации класса InOutServlet

Класс **InOutServlet** состоит из конструктора и одного метода – **perform**. При вызове конструктора класса ему передаются два параметра – объекты выходного и входного потоков. В нашем примере конструктор принимает ссылки на выходной поток сервлета и входной поток, позволяющий считать тело http-ответа. Метод **perform**

выполняет побайтную перезапись всех данных входного потока в выходной. Вызов **perform** в сервлете **Sss** позволяет переписать данные http-ответа сервлета **Ppp** в выходной поток сервлета **Sss**. В нашем случае принимать выходные данные сервлета **Sss** будет web-браузер (рис. 3.27, б).

На рис. 3.31 приводится пример новой реализации сервлета **Ppp**, принимающего запрос типа **POST**.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.commons.httpclient.*;
import org.apache.commons.httpclient.methods.*;
public class Ppp extends HttpServlet implements Servlet {
    protected void doPost(HttpServletRequest rq, HttpServletResponse rs)
        throws ServletException, IOException {
        String firstname = rq.getParameter("firstname");
        String lastname = rq.getParameter("lastname");
        String password = rq.getParameter("password");
        String sex = rq.getParameter("sex");
        String press = rq.getParameter("press");
        HttpClient hc = new HttpClient();
        GetMethod gm;
        String uri = "http://localhost:8080" + rq.getContextPath();
        if (press.equalsIgnoreCase("OK")) {
            String parmstr = "firstname=" + firstname + "&"
                + "lastname=" + lastname;

            if (sex.equalsIgnoreCase("male"))
                uri += "/Mmm?" + parmstr;
            else if (sex.equalsIgnoreCase("female"))
                uri += "/Ggg?" + parmstr;
        } else
            uri += "/index.html";
        hc.executeMethod(gm = new GetMethod(uri));
        rs.setContentType("text/html");
        PrintWriter pw = rs.getWriter();
        pw.println(gm.getResponseBodyAsString());
        pw.flush();
    }
}
```

Рис. 3.31. Пример реализации сервлета, генерирующего http-запрос типа GET

Кроме того, сервлет **Ppp** обрабатывает входные параметры и формирует запрос типа **GET** (обратите внимание, что используется класс **GetMethod**), вызывающий в зависимости от значений параметров **press** и **sex** сервлеты **Mmm** и **Ggg** или статическую страницу **index.html**. При вызове сервлетов параметры генерируемого запроса передаются в строке

запроса URL. Ответ в сервлете **Ppp** генерируется уже рассмотренным ранее способом – с помощью объекта класса **PrintWriter**. Впервые встречается здесь применение метода **getResponseBobyAsString** класса **GetMethod** (такой же метод есть и у класса **PostMethod**). Метод **getResponseBobyAsString** позволяет преобразовать выходной поток http-ответа вызываемого объекта в строку (тип **String**). В документации к пакету **httpclient** рекомендуется использовать этот метод с осторожностью и только тогда, когда достоверно известно, что ответ не является длинным. В тех случаях, когда заранее нельзя предсказать длину ответа, рекомендуется побайтное считывание (рис. 3.29, 3.30).

На рис. 3.32 представлен пример реализации сервлета **Mmm**, который в нашем случае вызывается сервлетом **Ppp** с помощью запроса типа **GET**. Сервлет обрабатывает входные параметры и выводит html-текст в выходной поток с помощью методов класса **PrintWriter**.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Mmm extends HttpServlet implements Servlet {
    public void doGet(HttpServletRequest rq, HttpServletResponse rs)
        throws ServletException, IOException {
        String firstname = rq.getParameter("firstname");
        String lastname = rq.getParameter("lastname");
        rs.setContentType("text/html");
        PrintWriter pw = rs.getWriter();
        pw.println("<html> <body> <br>Mmm:doGet:firstname=" +
firstname
                + "<br>Mmm:doGet:lastname=" + lastname + "<body><html>");
        pw.flush();
    }
}
```

Рис. 3.32. Пример реализации сервлета, обрабатывающего запросы типа GET

Обратите внимание на то, что сервлет **Mmm** (как, впрочем, и сервлеты **Ppp** и **Sss**) заранее «не знает», кем он будет вызван. Он просто принимает переданные ему параметры и формирует ответ в выходной поток. Если этот сервлет вызывать с помощью web-браузера, то выходной поток будет интерпретироваться web-браузером. Очевидно, что можно сделать этот сервлет независимым и от типа запроса, для этого надо просто изменить имя обработчика на **service**.

Сборка приложения ANaive с применением перечисленных выше дополнительных пакетов требует внести изменения в цель **compile** файла сборки **build.xml**. На рис. 3.33 приведен фрагмент этого файла,

позволяющий откомпилировать классы приложения с jar-файлами этих пакетов.

```
<!-- ..... -->
<target name="compile" depends="init">
  <delete verbose="true">
    <fileset dir="${work}build"/>
  </delete>
  <javac srcdir="${work}src" destdir="${work}\build">
    <classpath>
      <fileset file="${server}lib\servlet-api.jar" />
      <fileset dir="${server}common\lib" />
    </classpath>
  </javac>
</target>
<!-- ..... -->
```

Рис. 3.33. Фрагмент файла сборки для компиляции приложения с использованием дополнительных библиотек

По сравнению с предыдущим вариантом файла сборки, в элементе **javac** отсутствует атрибут **classpath**, но появляется новый внутренний элемент с таким же именем, который в свою очередь содержит два элемента **<fileset>**. Первый **<fileset>**, как и прежде, указывает на jar-файл **servlet-api.jar**, а второй – на директорию с jar-файлами дополнительных пакетов.

### 3.5. Параметры инициализации сервлета

Часто при разработке сервлета необходимо вынести за пределы программного кода некоторые данные, которые, как правило, представляют собой информацию, связанную с настройкой сервлета для его применения в конкретных условиях. Например, если предполагается, что сервлет должен обрабатывать запросы только в определенный интервал времени суток, то удобно значения времени начала и окончания этого интервала поместить где-нибудь во внешнем файле и считывать его каждый раз при вызове сервлета. В этом случае при изменении интервала работы сервлета нет необходимости перекомпилировать сервлет, а требуется лишь изменить этот файл.

В спецификации сервлетов предусмотрено специальное средство, позволяющее очень просто разметить некоторые данные за пределами программного кода. Для этого можно использовать параметры инициализации сервлета.

Параметры инициализации сервлета и их значения описываются в теле соответствующего тега `<servlet>` дескриптора развертывания приложения. На рис. 3.34 представлен пример фрагмента дескриптора развертывания приложения, описывающего сервлет **Hhh** с одним параметром инициализации.

```
<!-- ..... -->
<servlet>
    <servlet-name>Hhh</servlet-name>
    <servlet-class>Hhh</servlet-class>
    <init-param>
        <param-name>fhhtml</param-name>
        <param-value>male.html</param-value>
    </init-param>
</servlet>
<!-- ..... -->
```

Рис. 3.34. Фрагмент дескриптора развертывания с описанием параметра инициализации сервлета

Единственный параметр инициализации в примере имеет имя **fhhtml** и значение **male.html**. Значение параметров инициализации всегда имеют строкой тип. В общем случае параметров инициализации может быть больше, чем один.

Получить значение параметра инициализации в сервлете можно с помощью метода **getInitParameter** интерфейса **Servlet**. На рис. 3.35 приведен фрагмент сервлета, считывающего значение параметра **fhhtml**.

```
// .....
public class Hhh extends HttpServlet implements Servlet {

    protected void doGet(HttpServletRequest rq,
                        HttpServletResponse rs)
        throws ServletException, IOException {

        // .....

        String fhml = getInitParameter("fhhtml");
        rq.getRequestDispatcher("/"+fhml).forward(rq, rs);

        // .....
    }
}
```

Рис. 3.35. Фрагмент сервлета, использующего параметр инициализации

### 3.5. Итоги главы

1. Сервлет – это web-компонент, расположенный в серверной части web-приложения.
2. Сервлеты выполняются в специальной среде – контейнере сервлетов, который является составной частью web-контейнера.
3. Среда, в которой может работать web-контейнер, определяется его спецификацией – обычно это web-сервер или сервер приложений.
4. Сервлет не зависит от программно-аппаратной платформы, на которой работает web-контейнер. Более того, сервлет переносим на уровне бинарных файлов между разными web-контейнерами, если эти контейнеры реализуют одну и ту же версию спецификации контейнера сервлетов.
5. Основное назначение сервлетов – динамическое формирование содержимого ответов на запросы клиентского приложения. В общем случае сервлеты не зависят от протокола связи, но наиболее часто они применяются для обработки http-запросов.
6. С точки зрения программиста сервлет – это класс Java, реализующий интерфейс **Servlet**. Для обработки http-запросов используют класс **HttpServlet**, реализующий методы интерфейса **Servlet** и предоставляющий свои методы, предназначенные для обработки http-запросов и формирования http-ответов.
7. Объекты классов, используемые сервлетом для приема запросов и формирования ответов, создаются контейнером, поэтому классы этих объектов (**HttpServlet**, **HttpServletRequest**, **HttpServletResponse** и др.) находятся в составе API web-контейнеров.
8. Чаще всего сервлеты используют для обработки http-запросов типа **GET** и **POST**. Сервлет может принимать эти запросы, обрабатывать параметры, выполнять необходимые вычисления и формировать ответ. Кроме того, сам сервлет может тоже формировать http-запросы, а также получать и обрабатывать http-ответы.
9. Сервлет является простым и удобным типом компонента web-приложения, позволяющим гибко организовать взаимодействие его серверной и клиентской частей. Использование сервлета никак не связано с его местом расположения в сети – для обращения к нему необходимо знать только его URL и перечень принимаемых параметров. Обработку его ответа можно осуществлять любым программным средством, поддерживающим протокол HTTP (в т. ч. и сервлетом). Для вызывающего объекта прозрачна среда исполнения (операционная система, web-сервер, сервер приложений) вызываемого сервлета.



## Глава 4. ТЕХНОЛОГИЯ JAVA SERVER PAGES

### 4.1. Предисловие к главе

Большая часть разрабатываемых web-приложений в конечном счете сводится к выводу данных в формате HTML для интерпретации web-браузером. В приведенных выше примерах html-код для вывода на web-браузер формировался непосредственно в программе или для этого использовались готовые html-страницы. Оба способа имеют ряд недостатков. Если html-код содержится внутри программного кода, то любое изменение в форме представления данных будет приводить к необходимости перекомпиляции программы. При использовании html-страниц программный код становится независимым от представления данных, но в этом случае некоторые задачи становятся неразрешимыми (например, отображение информации, выбранной из базы данных), а в остальных случаях такой подход приводит к необходимости создавать (или, как часто говорят, верстать) для каждого случая вывода данных отдельный html-файл.

Технология Java Server Pages (JSP) предназначена для создания специального серверного компонента web-приложения, называемой jsp-страницей и обладающей одновременно свойствам html-страницы и сервлета. В самом первом приближении jsp-страница – это html-страница с вкраплениями java-кода. Как и в случае с сервлетом, для исполнения jsp-страницы требуется специальный контейнер (JSP Engine), который отвечает за разбор (parsing) страницы JSP и преобразование ее в сервлет, генерирующий при исполнении html-код.

Применение технологии JSP не отрицает, а скорее дополняет технологию Java Servlet. Два основных архитектурных подхода при реализации приложений по технологии JSP имеют специальные названия: JSP Model 1 (рис. 4.1) и JSP Model 2 (рис. 4.2).

В первом архитектурном решении jsp-страница полностью отвечает за получение запроса клиента, его обработку, подготовку и отправку ответа. Для доступа к данным, как правило, используется объект `JavaBean`, который создается и инициализируется jsp-страницей. Методы этого объекта предназначены для формирования запроса к источнику данных (например, к серверу базы данных), а свойства для хранения данных запроса и результатов ответа.

Во втором случае запрос принимает и обрабатывает сервлет. Он же создает и инициализирует объект `JavaBean`, имеющий такое же

предназначение, что и в первом случае. Страница JSP в этой модели применяется только для формирования ответа на основе данных хранящихся в свойствах JavaBean-объекта.

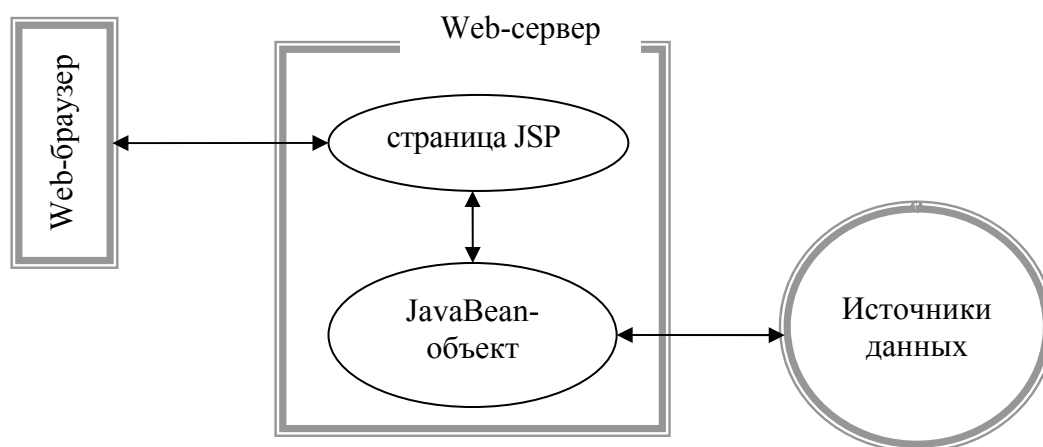


Рис. 4.1. Архитектура JSP Model 1

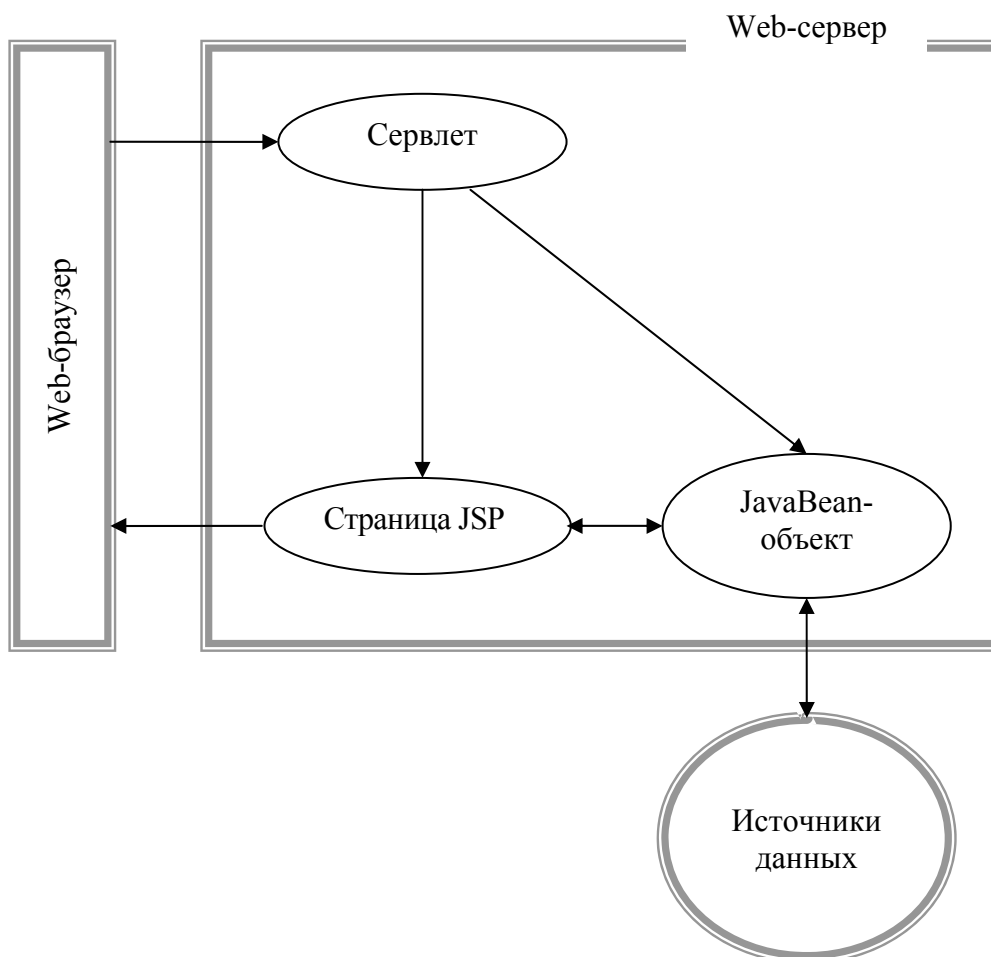


Рис. 4.2. Архитектура JSP Model 2

Преимущества второй модели становятся тем заметнее, чем сложнее разрабатываемое web-приложение.

## 4.2. Структура страницы JSP

Страница JSP – текстовый документ, обычно имеющий расширение **jsp** и содержащий данные двух типов: 1) статические данные, как правило, в формате HTML, XML, JavaScript; 2) динамические элементы – фрагменты java-кода. Может показаться странным, что JavaScript относится к статическим данным, но это потому, что динамика рассматривается относительно сервера. Web-сервер не интерпретирует код JavaScript, а лишь пересылает его клиенту (как правило, web-браузеру). По-другому обстоит дело с java-кодом – он может быть разобран и исполнен контейнером JSP. Более того, встроенный в jsp-страницу java-код, помимо html, может генерировать и JavaScript-код, который потом может исполняться web-браузером.

Любая jsp-страница, кроме обычных html-тегов, содержит специальные jsp-теги следующих категорий: директивы, объявления, скриплеты, выражения и комментарии.

### 4.2.1. Директивы JSP

Директивы предоставляют информацию контейнеру JSP, необходимую на стадии трансляции и имеют следующий синтаксис.

```
<% @ директива имя_атрибута_1 = "значение"  
имя_атрибута_2 = "значение" ... %>
```

Существует три типа директив: **page**, **taglib** и **include**.

Директива **page** определяет свойства страницы JSP. На рис. 4.3 приведен пример простейшей jsp-страницы с директивой **page**.

Значение атрибута **language** директивы **page** определяет язык (в примере Java), используемый в скриплетах (фрагментах программного кода), в выражениях или других включаемых файлах. Значение атрибута **contentType** устанавливает MIME-тип ответа и кодировку страницы.

Директива **taglib** указывает на то, что в странице JSP будут использоваться библиотеки тегов. В связи с тем, что разработка и применение библиотеки тегов будет рассматриваться отдельно,

отложим детальное описание этой директивы до соответствующего раздела главы.

```
<%@ page language="java"
contentType="text/html; charset=ISO-8859-1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>IS&T-2009 </title>
</head>
<body>
<h2>JSP-directives</h2>
</body>
</html>
```

Рис. 4.3. Пример использования директивы page

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head> <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
</head>
<body>
<!-- output comment --> <% // not output comment %>
<script type="text/javascript">
<%@ include file="head.js"%>
Head("Web-programming");
</script>
<h2>JSP-directives</h2>
<table >
<tr><td valign ="top"><b>page</b></td>
<td>
<tt><%@ include file="jsp-directives-page.html" %></tt>
</td></tr>
<tr><td valign ="top"><b>taglib</b></td>
<td>
<tt><%@ include file="jsp-directives-taglib.html"
</td></tr>
<tr><td valign ="top"><b>include</b></td>
<td>
<tt><%@ include file="jsp-directives-include.html" %></tt>
</td></tr>
</table>
</body>
</html>
```

Рис. 4.4. Пример применения директивы include

Директива **include** позволяет вставлять текст и код в процессе трансляции jsp-страницы. На рис. 4.4 приведен пример jsp-страницы (пусть для определенности это страница с именем **jsp-directives.jsp**) с директивой **include**. Директива здесь используется для вставки одного файла с инструкциями JavaScript (далее js-файл) и двух html-файлов.

Кроме того, в последнем примере демонстрируется два типа комментариев, которые можно применять в jsp-страницах: **выводимые** (комментарии в стиле HTML) и **невыводимые** комментарии. При вызове страницы JSP выводимые комментарии будут доставлены на http-клиент, а невыводимые – нет.

Для того чтобы продемонстрировать результат вызовов страницы **jsp-directives.jsp**, приведем здесь содержимое файлов **head.js** (рис. 4.5) и **jsp-directives-page.jsp** (рис. 4.6), а содержимое остальных двух html-файлов станет очевидным позже.

```
function Head(txt){
    var td = new Date();
    if (txt == null) txt='';
    document.write('<div style="width:500px;text-align:right">'
    + '<tt>'+txt+' ['+td+']</tt>'
    + '<hr align="right" width="500px" color="Olive" /></div>');
}
```

Рис. 4.5. Пример файла с инструкциями JavaScript, вставляемого в jsp-страницу с помощью директивы include

```
<table >
<tr> <td>&lt;%@ page
    <td> language = "javaScript|..." </td></tr>
<tr><td></td><td> extends = "superclass" </td></tr>
<tr><td></td><td> import = "packages list" </td></tr>
<tr><td></td><td> Buffer = "none|buffer size" </td></tr>
<tr><td></td><td> Session="true|false" </td></tr>
<tr><td></td><td> autoFlush="true|false" </td></tr>
<tr><td></td><td> isThreadSafe="true|false" </td></tr>
<tr><td></td><td> info = "string" </td></tr>
<tr><td></td><td> errorPage = "string" </td></tr>
<tr><td></td><td> isErrorPage = "true|false" </td></tr>
<tr><td></td><td> contentType = "MIME-type; charset..." </td></tr>
<tr><td></td><td> pageEncoding = "charset" %&gt; </td></tr>
</table>
```

Рис. 4.6. Пример html-файла, вставляемого в jsp-страницу с помощью директивы include

В простейшем случае разработанную jsp-страницу необходимо поместить в директорию приложения. На рис. 4.7 приводится пример

фрагмента файла сборки, выполнение которого приводит к копированию jsp-страницы, js-файла и всех html-файлов из рабочей директории в директорию приложения web-сервера. При этом предполагается, что новые jsp- и js-файлы хранятся в поддиректориях **jsp** и **js** соответственно.

```
<!-- ..... -->
<target name="copy" depends="create">
<!-- ..... -->
  <copy todir="${server}webapps\ANaive\ ">
    <fileset dir="${work}jsp" />
  </copy>
  <copy todir="${server}webapps\ANaive\ ">
    <fileset dir="${work}js" />
  </copy>
</target>
<!-- ..... -->
```

Рис. 4.7. Фрагмент файла сборки, копирующий js и jsp-файлы в директории сервера

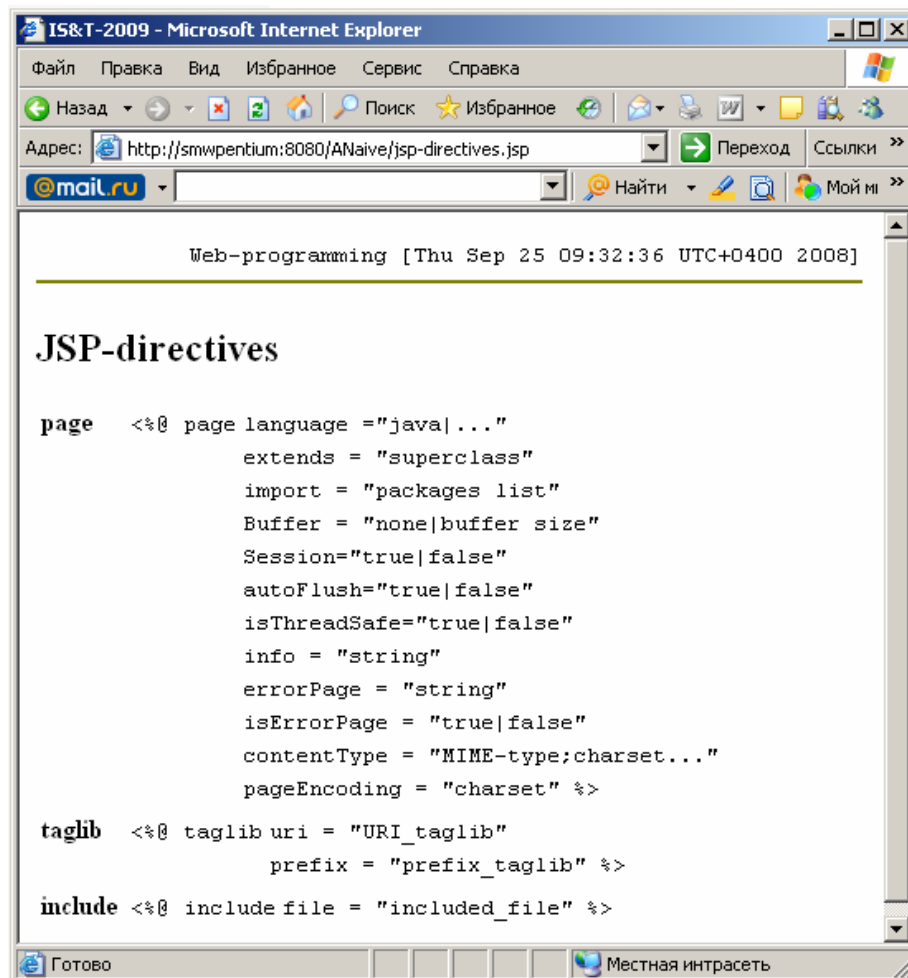


Рис. 4.8. Результат вызова jsp-страницы

Страницу JSP можно вызвать с помощью web-браузера, сервлета или другой jsp-страницы. Более того, ее можно указать в элементе **welcome-file-list** дескриптора развертывания приложения точно так, как это демонстрировалось в предыдущих примерах для страницы **index.html**.

Вызов страницы **jsp-directives.jsp** с помощью браузера приведет к результату, подобному рис. 4.8. Принцип работы jsp-контейнера становится очевидным, если вывести html-код, сгенерированный им в результате обработки **jsp-directives**. На рис. 4.9 приводится фрагмент html-кода, сгенерированного из **jsp-directives.jsp**. Если используется Microsoft Internet Explorer, то этот код можно получить с помощью команды браузера **Вид/Просмотр HTML-кода (View/Source)**.

```
<!-- output comment -->
<script type="text/javascript">
    function Head(txt) {
    var td = new Date();
    if (txt == null) txt='';
    document.write('<div style="width:500px;text-align:right">'
        +'<tt>'+txt+' ['+td+']</tt> <hr align="right"
        width="500px" color="Olive" /></div>');
    }
    Head("Web-programming");
</script>

<h2>JSP-directives</h2>
<table >
<tr><td valign ="top"><b>page</b> </td>
<td><tt>
<table >
<tr> <td>&lt;%@ page </td>
    <td> language ="java|..." </td></tr>
<tr><td></td><td> extends = "superclass" </td></tr>
<tr><td></td><td> import = "packages list" </td></tr>
<tr><td></td><td> Buffer = "none|buffer size" </td></tr>
<tr><td></td><td> Session="true|false" </td></tr>
<tr><td></td><td> autoFlush="true|false" </td></tr>
<tr><td></td><td> isThreadSafe="true|false" </td></tr>
<tr><td></td><td> info = "string" </td></tr>
<tr><td></td><td> errorPage = "string" </td></tr>
<tr><td></td><td> isErrorPage = "true|false" </td></tr>
<tr><td></td><td> contentType = "MIME-type;charset..."</td></tr>
<tr><td></td><td> pageEncoding = "charset" %&gt;</td></tr>
</table>
</tt></td></tr>
```

Рис. 4.9. Фрагмент html-кода, сгенерированного с jsp-страницы

Фрагмент html-кода на рис. 4.9 демонстрирует преобразование двух типов комментариев и двух директив **include**, содержащихся в jsp-странице на **jsp-directives.jsp** (рис. 4.4).

#### 4.2.2. Объявления JSP

Тег JSP, применяемый для объявлений, имеет следующий синтаксис.

`<% ! декларации переменных | декларация методов %>`

Объявления не производят никакого вывода в выходной поток. Переменные и методы, декларированные в объявлениях, становятся доступными для скриплетов и других объявлений в момент инициализации страницы JSP.

#### 4.2.3. Выражения JSP

Тег JSP, применяемый для выражений, имеет следующий синтаксис.

`<% = исполняемое выражение на языке скрипта %>`

Выражение в jsp-странице – это исполняемое выражение, написанное на языке скрипта, указанного атрибутом **language** в директиве **page** (в нашем случае это язык Java). Результат выражения автоматически приводится к типу String и выводится в стандартный поток. Если выражение не может быть преобразовано к типу String, то возникает ошибка выполнения.

На рис. 4.10 демонстрируется фрагмент jsp-страницы с тегами объявления и выражения.

В теге объявления содержатся декларации трех переменных и метода **Salutation**, в теге выражения осуществляется вызов объявленного метода. При объявлении и инициализации одной из переменных используется класс **Regimex**. Для того чтобы при трансляции jsp-страницы имя этого класса разрешилось, необходимо в атрибуте **import** (действие этого атрибута схоже с действием одноименного оператора в тесте java-программы) директивы **page** указать месторасположение класса.

В рассматриваемом примере класс **Regimex** содержится в пакете с именем **jspclass**.

Для определенности на рис. 4.11 приведен пример реализации класса **Regimex**, а на рис. 4.12 – результат вызова страницы **jsp-directives.jsp**, в которую добавлен фрагмент, представленный на рис. 4.10.



```

<%@ page language="java"
    contentType="text/html; charset=ISO-8859-1"%>
<%@ page import="jspclass.*"%>
<!-- ..... -->
<head>
<meta http-equiv="Content-Type"
content="text/html; charset=ISO-8859-1">
<title>IS&T-2009</title>
<%! String name = null;
    Regimex r = new Regimex();
    Integer n = r.GetHOURL();
    protected String Salutation(Integer h) {
        String rc = "Good ";
        if ((h > 0) && (h <= 5))
            rc += "night";
        else if ((h > 5) && (h <= 12))
            rc += "morning";
        else if ((h > 12) && (h <= 17))
            rc += "afternoon";
        else
            rc += "evening";
        return rc;
    }%>
</head>
<body>
<!-- ..... -->
<%=Salutation(n)%>
<!-- ..... -->

```

Рис. 4.10. Фрагмент jsp-страницы, содержащей теги объявления и выражения

```

package jspclass;
import java.util.*;

public class Regimex {
    protected String R = null;
    Calendar C;
    protected Date d = new Date();

    public Regimex(Calendar c) {
        this.C = c;
    }
    public Regimex() {
        this.C = Calendar.getInstance();
    }
    public Integer GetHOURL() {
        return ((Integer)C.get(Calendar.HOUR_OF_DAY));
    }
}

```

Рис. 4.11. Пример класса, используемого в jsp-странице

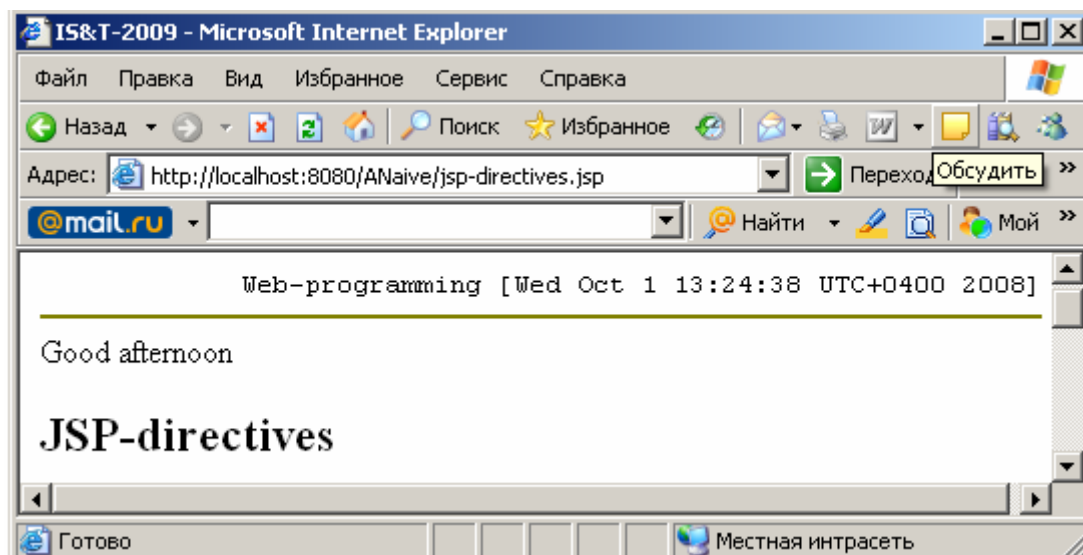


Рис. 4.12. Результат выполнения выражения JSP

#### 4.2.4. Скриплеты JSP

Скриплеты должны содержать фрагменты кода на языке скрипта, который указывается в атрибуте **language** директивы **page** (в нашем случае это язык Java).

Тег JSP, применяемый для скриплетов, имеет следующий синтаксис.

```
<% скрипт на языке Java %>
```

На рис. 4.13 приведен фрагмент jsp-страницы **jsp-directives.jsp**, демонстрирующий применение скриплетов для динамического формирования выходного потока.

В примере на рис. 4.13 jsp-страница в зависимости от содержимого строки запроса генерирует разный html-текст выходного потока. Строка запроса получена здесь с помощью метода **getQueryString** неявного (встроенного) объекта **request**, доступного в скриплете. При get-запросе jsp-страницы в строку **name** будет помещаться часть URL, находящаяся справа от знака вопроса, отделяющего адресную часть URL от параметров. Аналогичный метод есть и у объекта **HttpServletRequest**, который используется в сервлете. На рис. 4.14 изображен результат вызова страницы **jsp-directives.jsp**. Обратите внимание на то, что передача и обработка параметров в этом запросе типа **GET** осуществляется способом, отличным от рассмотренного ранее.

```

<!-- ..... -->
<%name = request.getQueryString();%>
<%= "Parms:" + name%>
<table>
<%if (name.indexOf("page") >= 0) {%>
    <tr>
        <td valign="top"><b>page</b></td>
        <td><tt> <%@ include file="jsp-directives-page.html"%>
        </tt></td>
    </tr>
<%} if (name.indexOf("taglib") >= 0) {%>
    <tr>
        <td valign="top"><b>taglib</b></td>
        <td><tt><%@ include file="jsp-directivestaglib.html"%></tt></td>
    </tr>
<%} if (name.indexOf("include") >= 0) {%>
    <tr>
        <td valign="top"><b>include</b></td>
        <td><tt><%@ include file="jsp-directives-include.html"%></tt></td>
    </tr>
<%}%>
<!-- ..... -->

```

Рис. 4.13. Фрагмент страницы JSP, содержащий скриплеты

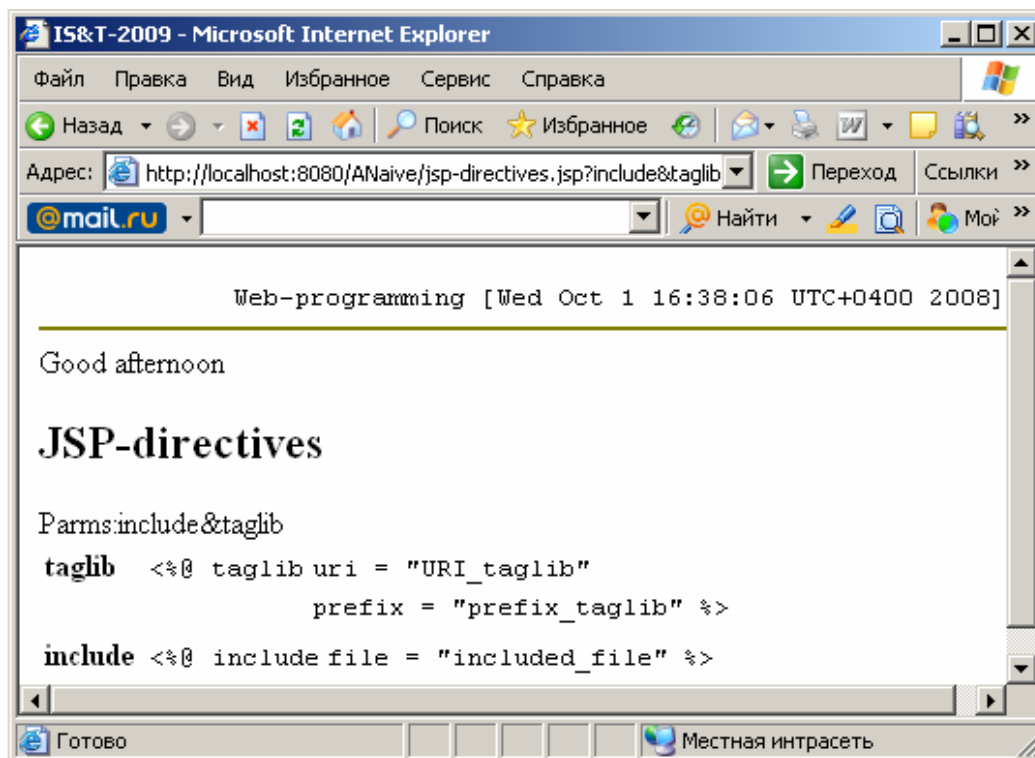


Рис. 4.14. Результат работы страницы JSP, содержащей скриплеты

Последний пример демонстрирует реализацию архитектуры JSP Model 1, рассмотренную выше.

#### 4.2.5. Неявные объекты JSP

**Неявные объекты** (Implicit objects) – это объекты, автоматически доступные в скриплетях JSP без специального их объявления или импорта. Перечень неявных объектов приведен в табл. 4.1.

Таблица 4.1

**Неявные объекты JSP**

Объект	Тип	Назначение
request	javax.servlet.HttpServletRequest	Описывает http-запрос
response	javax.servlet.HttpServletResponse	Описывает http-ответ
PageContext	javax.servlet.HttpServletPageContext	Доступ к содержимому jsp-страницы
session	javax.servlet.HttpSession	Описывает сессию клиента jsp-страницы, выполнившего запрос
application	javax.servlet.ServletContext	Контекст jsp-страницы
out	javax.servlet.JspWriter	Используется для записи в выходной поток
config	javax.servlet.ServletConfig	Конфигурация jsp-страницы
page	java.lang.Object	Как правило, не используется
exception	java.lang.Throwable	Используется для обработки ошибок

Полное описание неявных объектов и их методов можно найти в спецификации Java Server Page [12]. Некоторые из них будут использоваться и поясняться в примерах. Следует обратить внимание на совпадение типа основных (чаще всего используемых) неявных объектов JSP с типом объектов, применяемых в сервлете. Это говорит о единой природе технологий сервлетов и JSP.

#### 4.3. Библиотеки тегов

С точки зрения разработчика web-приложения **библиотека тегов** (Tag Library) – это технология, позволяющая создавать собственные теги (будем их далее называть tdl-тегами), которые потом можно использовать в jsp-страницах.

Для того чтобы воспользоваться этой технологией, необходимо выполнить следующее:

- 1) создать дескриптор библиотеки тегов (Tag library descriptor, TDL) и поместить его в директорию приложения;
- 2) создать обработчики тегов (Tag handler) – java-классы, генерирующие html-текст, замещающий tdl-теги, в выходном потоке jsp-страницы;
- 3) поместить на jsp-странице директиву **taglib**, указывающую на месторасположение дескриптора библиотеки тегов и задающую префикс (пространство имен) для имен tdl-тегов в данной странице;
- 4) добавить tdl-теги в jsp-страницу.

Изложенная здесь технология библиотеки тегов соответствует спецификации JSP 1.2. На сегодняшний день существует более поздняя версия – JSP 2.0. Особенности применения новой версии изложены в источнике [7].

#### 4.3.1. Дескриптор библиотеки тегов

Дескриптор библиотеки тегов представляет собой текстовый файл, выполненный в формате XML. Он содержит описание библиотеки тегов и элементов библиотеки.

На рис. 4.15 приведен пример дескриптора, описывающего библиотеку, содержащую два tld-тега.

Тег **<taglib>** открывает описание библиотеки, которое располагается до закрывающего тега **</taglib>**. Описание библиотеки состоит из пролога и описаний tld-тегов библиотеки.

Пролог содержит теги **<taglib-version>** для установки версии пользовательской библиотеки (в нашем случае установлена версия 1.0), **<jsp-version>** для указания применяемой спецификации JSP (в примере 1.2), **<short-name>** для символического обозначения (наименования) библиотеки (в примере – **StaffTag**) и **<uri>**, содержащего идентификатор ресурса библиотеки тегов (в примере – **StaffTag.tld**).

Описание каждого tld-тега библиотеки начинается с тега **<tag>** и заканчивается закрывающим тегом **</tag>**. В примере приводится описание двух различных тегов.

Для первого tld-тега с именем **surname** (указывается элементом **name**) используется класс-обработчик с именем **stafftag.Surname.class** (значение элемента **tag-class**). Этот tld-тег не содержит тела (значение **EMPTY** элемента **body-content**), но имеет один необязательный атрибут (значение **false** элемента **required**) с именем **value** (значение элемента **name**) строкового типа (значение **java.lang.String** элемента **type**).

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library
1.2//EN"
"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd" >
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>StaffTag</short-name>
  <uri>stafftag.tld</uri>
  <tag>
    <name>surname</name>
    <tag-class>stafftag.Surname</tag-class>
    <body-content>EMPTY</body-content>
    <attribute>
      <name>value</name>
      <required>false</required>
      <type>java.lang.String</type>
    </attribute>
  </tag>
  <!-- ..... -->
  <tag>
    <name>dossier</name>
    <tag-class>stafftag.Dossier</tag-class>
    <body-content>JSP</body-content>
    <attribute>
      <name>action</name>
      <required>true</required>
      <type>java.lang.String</type>
    </attribute>
  </tag>
</taglib>

```

Рис. 4.15. Пример дескриптора библиотеки тегов

Второй tld-тег с именем **dossier** допускает использование тела и имеет один обязательный атрибут **action** тоже строкового типа.

Если необходимо описать несколько атрибутов для tld-тега, то внутри тега **<tag>** необходимо поместить несколько тегов **<attribute>** с соответствующим описанием.

### 4.3.2. Обработчик тега

Обработчик тега – это java-класс, реализующий один из интерфейсов **Tag**, **IterationTag** или **BodyTag**. На практике обработчики тега представляют собой расширение одного из классов **TagSupport**, **BodyTagSupport** или **TagExtraInfo**, которые реализуют эти интерфейсы.

На рис. 4.16 приводится пример обработчика tld-тега **surname**, описанного в дескрипторе библиотеки тегов **Stafftag** (рис. 4.15).

Обработчик **Surname** расширяет класс **TagSupport** и реализует три метода: **doStartTag**, **setValue** и **getVlaue**.

```

package stafftag;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;
import java.io.IOException;

public class Surname extends TagSupport {
    static String in = "<label>Surname&nbsp; &nbsp;</label>"
        + "<input name =\"surname \" type = \"text\" width = \"150\" \"
        + \" maxlength= \"30\" \" ";
    public String value = "";
    public String getValue() {
        return value;
    }
    public void setValue(String value) {
        this.value = value;
    }

    public int doStartTag() throws JspException {
        JspWriter out = pageContext.getOut();
        try {
            out.print(in + (this.value.equals(""))
                ? " "
                : "value =\"" + this.value + "\" />" );
        } catch (IOException e) {
            System.out.println("stafftag.Surname: " + e);
        }
        return SKIP_BODY;
    }
}

```

Рис. 4.16. Пример обработчика простого тега

Метод **doStartTag** предназначен для генерации текста, замещающего начальный тег.

В данном примере tld-тег **<surname>** не имеет тела, поэтому начальный тег является единственным. Генерация текста осуществляется с помощью объекта **JspWriter**, ссылка на который может быть получена с помощью неявного объекта **pageContext**, доступного в классе-обработчике. Следует сразу отметить, что в обработчике могут использоваться все неявные объекты JSP.

Метод **setValue** используется контейнером JSP, вызывающим его при разборе jsp-страницы для передачи значения атрибута **value**, указанного в теге **<surname>**. Имя **setValue** образовано как конкатенация префикса **set** и суффикса, совпадающего с именем атрибута **value**.

Метод **getValue** используется в обработчике для доступа к значению атрибута в обработчике. Имя **getValue** образуется по тому же принципу, что и в предыдущем случае.

На рис. 4.17 приводится пример обработчика tld-тега **dossier**, описанного в дескрипторе библиотеки тегов **Stafftag** (рис. 4.15). В отличие от предыдущего случая, тег **<dossier>** предполагает наличие тела и конечно-го тега **</dossier>**. Для обработки начального тега, как и прежде, используется метод **doStartTag**, а обработка атрибута **action** осуществляется точно также. Отличительными особенностями этого обработчика является реализация дополнительного метода **doEndTag** и значение кода **EVAL\_BODY\_INCLUDE** возврата, формируемого методом **doStartTag**.

```
package stafftag;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;
import java.io.IOException;

public class Dossier extends TagSupport {
    static String in = "<label>Surname&nbsp; &nbsp;</label>"
        + "<input name =\"surname \" type = \"text\" width =
\"150\" \"
        + \" maxlength= \"30\" \" ";
    public String action = "";
    public String getAction() {
        return this.action;
    }
    public void setAction(String action) {
        this.action = action;
    }
    public int doStartTag() throws JspException {
        String in = "<form id =\"f01\" name = \"f01\" method =
\"post\" \"
        + \" action= \"" + this.action + "\">";
        JspWriter out = pageContext.getOut();
        try {
            out.print(in);
        } catch (IOException e) {
            System.out.println("stafftag.Dossier: " + e);
        }
        return EVAL_BODY_INCLUDE;
    }
    public int doEndTag() throws JspException {
        String in = "</form>";
        JspWriter out = pageContext.getOut();
        try {
            out.print(in);
        } catch (IOException e) {
            System.out.println("stafftag.Dossier: " + e);
        }
        return EVAL_BODY_INCLUDE;
    }
}
```

Рис. 4.17. Пример обработчика тега, имеющего тело



Метод **doEndTag** предназначен для генерации текста, замещающего конечный тег. Как и прежде, генерация осуществляется с помощью объекта **JspWriter**.

#### 4.3.3. Применение библиотечных тегов

На рис. 4.18 приводится пример jsp-страницы, разработанной с применением tld-тегов: **dossier**, **surname**, **lastname**, **submit**.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri = "stafftag.tld" prefix ="staff" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html;
        charset=ISO-8859-1" />
    <title>Dossier</title>
</head>
<body>
    <staff:dossier action= "PostExample" >
        <br /><staff:surname value="Smelov" />
        <br /><staff:lastname value="Vladimir" />
        <br /><staff:submit name ="press" />
    </staff:dossier>
</body>
</html>
```

Рис. 4.18. Пример применения библиотечных тегов

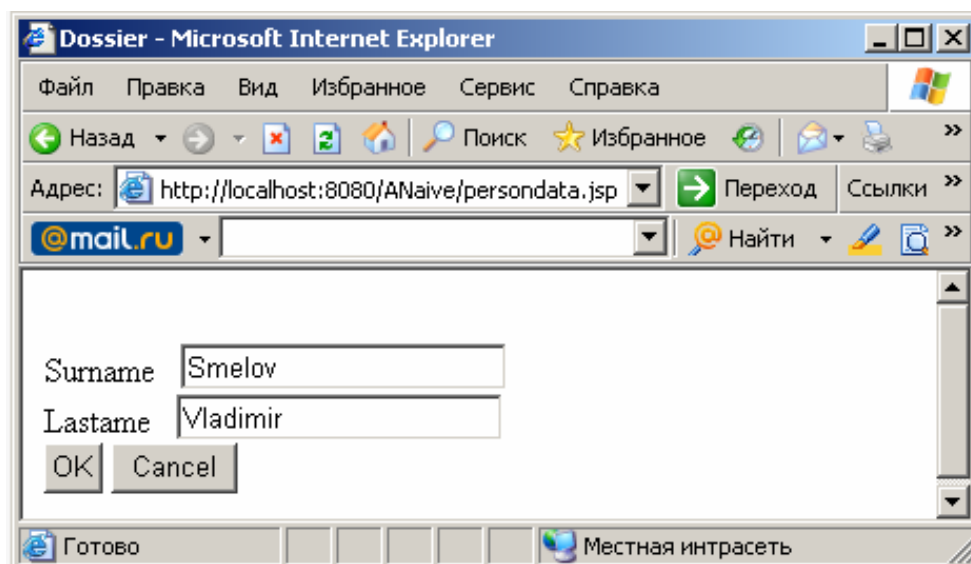


Рис. 4.19. Результат вызова jsp-страницы с библиотечными тегами

Тег `<%@ taglib >` описывает jsp-директиву **taglib**, которая указывает на месторасположение библиотеки тегов и устанавливает префикс (пространство имен) для tld-тегов. В нашем случае дескриптор библиотеки тегов должен находиться в корне директории WEB-INF приложения, а префикс имеет значение **staff**.

Вызов jsp-страницы, приведенной на рис. 4.18, приведет к результату, как на рис. 4.19.

Если вывести html-текст, соответствующий изображению jsp-страницы в окне браузера (рис. 4.19), то будет результат, как на рис. 4.20.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"
/>
<title>Dossier</title>
</head>
<body>
<form id = "f01" name = "f01" method = "post" action = "PostExecute " >
<br /><label>Surname&nbsp; &nbsp;</label>
<input name = "surname " type = "text" width = "150" maxlength= "30"
value = "Smelov" />
<br /><label>Lastame&nbsp; &nbsp;</label>
<input name = "lastname " type = "text" width = "150" maxlength= "30"
value = "Vladimir" />
<br /><input name = "press" type = "submit" value = "OK" size = "20"
/>&nbsp; 
<input name = "press" type = "submit" value = "Cancel" size = "20" />
</form>
</body>
</html>
```

Рис. 4.20. Код, сгенерированный в результате обработки библиотечных тегов

Следует обратить внимание, что вместо тегов `<staff:dossier>` и `</staff: dossier>` сгенерированы теги `<form>` и `</form>`, а вместо тега `<staff: surname>` – тег `<input>`. Несложно догадаться, что обработчик тега `<staff: lastname>` будет аналогичен обработчику тега `<staff:surname>`, а обработчик тега `<staff:submit>` генерирует два html-тега `<input type=submit>`.

На рис. 4.21 приведен фрагмент файла сборки web-приложения, использующего библиотеку тегов. Изменения претерпели две цели сборки: **copy** (теперь следует копировать не только дескриптор приложения, но и библиотеку тегов) и **javac** (при компиляции и исполнении классов обработчиков требуется дополнительная библиотека **jsp-api.jar**).

```

<!-- .....-->
<target name="compile" depends="init">
  <delete verbose="true">
    <fileset dir="${work}build"/>
  </delete>
  <javac srcdir="${work}src"
    destdir="${work}\build">
    <classpath>
      <fileset file="${server}lib\servlet-api.jar"/>
      <fileset file="${server}lib\jsp-api.jar"/>
      <fileset dir="${server}common\lib" />
    </classpath>
  </javac>
</target>
<!-- .....-->
<target name="copy" depends="create">
  <copy todir="${server}webapps\ANaive\WEB-INF" >
    <fileset dir="${work}WEB-INF" />
  </copy>
</target>
<!-- .....-->

```

Рис. 4.21. Фрагменты файла сборки web-приложения, использующего библиотеку тегов

#### 4.4. Стандартные действия JSP

Web-сервер может иметь свой набор действий, определенных для jsp-страниц в виде тегов, распознаваемых компилятором контейнера JSP. Однако существует стандартный перечень действий, обязательно поддерживаемых любым сервером. Такие стандартные действия представлены в табл. 4.2.

Таблица 4.2

Стандартные действия JSP

Конструкция XML	Назначение
<jsp:directive.page ... />	Эквивалентно действию тега <%@page ... %>
<jsp:directive.include ... />	Эквивалентно действию тега <%@include ... %>
<jsp:declaration ... > </jsp:declaration>	Эквивалентно действию тега <%! ... %>
<jsp:expression ... > </jsp:expression>	Эквивалентно действию тега <%=! ... %>
<jsp:scriptlet ... > </jsp:scriptlet>	Эквивалентно действию тега <% ... %>

Конструкция XML	Назначение
<code>&lt;jsp:include ... /&gt;</code> <code>&lt;jsp:include ...</code> <code>&gt;...&lt;/jsp:include&gt;</code>	Используется для включения статических и динамических ресурсов, например выходной информации сервлета
<code>&lt;jsp:forward ... /&gt;</code> <code>&lt;jsp:forward ... &gt;</code> <code>...&lt;/jsp:forward&gt;</code>	Переопределение запроса на другой ресурс (html, jsp-страница, сервлет)
<code>&lt;jsp:param ... /&gt;</code>	Используется для задания значений параметров для <code>&lt;jsp:include &gt;</code> , <code>&lt;jsp:forward &gt;</code> , <code>&lt;jsp:plugin&gt;</code>
<code>&lt;jsp:useBean ... /&gt;</code> <code>&lt;jsp:useBean ... &gt;...</code> <code>&lt;/jsp:useBean&gt;</code>	Объявление объекта JavaBean, используемый в jsp-странице
<code>&lt;jsp:setProperty ... /&gt;</code>	Установка значений свойств JavaBean
<code>&lt;jsp:getProperty ... /&gt;</code>	Чтение значений свойств JavaBean
<code>&lt;jsp:plugin ..&gt; ...</code> <code>&lt;/jsp:plugin ..&gt;</code>	Внедрение в JSP дополнительных программных модуля

Обратите внимание, что все имена стандартных действий имеют префикс `jsp`, а исполнение некоторых действий аналогично исполнению директив.

На рис. 4.22 приведен фрагмент страницы JSP, использующей тег `<jsp:include>`, вставляющий выходной поток сервлета **Salutation** в выходной поток jsp-страницы. При этом передается параметр с именем **hour** и значением, которое будет определено динамически на этапе компиляции jsp-страницы.

```

<!-- ..... -->
<head>
<!-- ..... -->
<%!
String name;
Regimex r = new Regimex();
%>
</head>
<body>
<script type="text/javascript">
  <jsp:directive.include file="head.js"%>
    Head("Web-programming");
  </script>
  <jsp:include page = "Salutation" >
    <jsp:param name="hour" value="<%=r.GetHOURL().toString()%" />
  </jsp:include>
<!-- ..... -->

```

Рис. 4.22. Применение стандартных действий JSP

Следует отметить существенное отличие между директивой `<%@ include %>` и стандартным действием `<jsp:include>`. Если директива внедряет в jsp-страницу текст до компиляции страницы, то стандартное действие внедряет результат исполнения вызываемого объекта (сервлет, html или jsp-страница) после компиляции на этапе исполнения.

На рис. 4.23 приведен пример сервлета **Salutation**, который вызывается с помощью стандартного действия `<jsp:include>`.

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.*;
import javax.servlet.http.*;

public class Salutation extends HttpServlet implements Servlet {
    public Salutation() {
        super();
    }
    protected void service(HttpServletRequest rq,
                           HttpServletResponse rs)
        throws ServletException, IOException {
        String hour = rq.getParameter("hour");
        Integer h;
        try {
            h = new Integer(hour);
        } catch (NumberFormatException e) {
            h = -1;
        }
        rs.setContentType("text/html");
        PrintWriter pw = rs.getWriter();
        String rc = "Good ";
        if ((h > 0) && (h <= 5))
            rc += "night";
        else if ((h > 5) && (h <= 12))
            rc += "morning";
        else if ((h > 12) && (h <= 17))
            rc += "afternoon";
        else if ((h > 17) && (h <= 24))
            rc = "evening";
        else
            rc = "Hello";
        pw.println(rc);
        pw.flush();
    }
}
```

Рис. 4.23. Сервлет, генерирующий сообщение

Как видно из текста сервлета, он не имеет никаких особенностей и может быть вызван любым другим способом, следует лишь позаботиться об обработке или отображении выходного потока сервлета.

Аналогичным способом с помощью стандартного действия `jsp:include` могут быть вызваны `html`- и `jsp`-страницы.

#### 4.5. Стандартная библиотека тегов JSTL

Разработка собственной библиотеки тегов требует от разработчика дополнительных знаний, навыков и усилий и к тому же может занять много времени. Поэтому на этапе проектирования требуется тщательно оценить необходимость применения этой технологии.

С другой стороны, применение библиотеки тегов дает возможность создавать повторно используемые элементы `jsp`-страниц, что позволяет избежать детального программирования и в конечном счете приводит к повышению производительности разработчика и уменьшению количества ошибок.

Компромиссом может служить решение использовать уже готовую стандартную библиотеку тегов – **JavaServer Pages Standard Tag Library (JSTL)**. Спецификация JSTL появилась в результате коллективного труда `java`-разработчиков и была оформлена в виде JRS-052. Первая реализация JSTL (версия 1.0) была разработана участниками проекта Apache Jakarta Project. На момент написания книги на сайте проекта [20] была доступна для скачивания версия 1.1 и готовилась к выпуску версия 1.2.

JSTL имеет широкую функциональность, позволяющую существенно сократить время на разработку любой `jsp`-страницы. С помощью стандартных тегов можно устанавливать атрибуты объектов, выводить текст на `jsp`-страницы, организовывать циклы обработки данных, форматировать числа и даты, преобразовывать `xml`-конструкции, применять SQL для работы с базами данных и др.

В качестве начального изучения технологии JSTL рекомендуется книга [7]. Для профессионального применения необходимо ознакомиться с документацией на сайте [20].

Следует отметить, что, кроме стандартной библиотеки JSTL, на сайте Apache Jakarta Project [20] доступны другие специализированные библиотеки тегов: **Session Tag Library** (набор тегов для работы с объектом `HttpSession`), **Mailer Tag Library** (набор тегов для работы с электронной почтой), **Regexp Tag Library** (набор тегов для работы с регулярными выражениями) и др. Для каждой библиотеки существует свой сайт, с которого можно скачать доступные версии библиотеки и документацию.

## 4.6. Обработка параметров запроса в JSP

Выше уже отмечалось, что JSP-страницы и сервлеты являются родственными технологиями. В конечном счете, jsp-страница преобразуется в сервлет и исполняется в том же контейнере сервлетов. В табл. 4.1 были приведены неявные объекты, доступные программисту при разработке jsp-страницы. Проанализировав типы этих объектов, не сложно заметить их совпадение с типами объектов, используемых в сервлетах, и догадаться, что они обладают схожими свойствами.

При вызове страницы JSP можно осуществлять передачу параметров запроса точно таким же способом, как это делалось в сервлете. Единственным отличием при обработке параметра в JSP является применения неявного объекта **request** вместо передаваемого параметра типа **HttpServletRequest**.

## 4.7. Обработка ошибок в JSP

Для обработки ошибок в JSP предусмотрен специальный механизм, основанный на применении двух атрибутов директивы **page: errorPage** и **isErrorPage**.

Атрибут **errorPage** устанавливается в директиве **page** основной страницы JSP и содержит URL дополнительной jsp-страницы, которая будет вызвана вместо основной, если там при исполнении возникла ошибка.

Атрибут **isErrorPage** директивы **page** должен иметь значение **true** на дополнительной странице JSP. Этот служит для указания компилятору JSP на необходимость сделать доступным неявный объект **exception**, свойства и методы которого позволяют идентифицировать и обработать возникшую ошибку.

## 4.8. Итоги главы

1. Технология Java Server Pages предназначена для создания специального серверного компонента, называемого jsp-страницей и обладающего одновременно свойствами html-страницы и сервлета.
2. Применение технологии JSP дополняет технологию сервлетов. Чаще всего эти две технологии используются совместно.

3. Существует два основных архитектурных подхода при реализации приложений с помощью технологии JSP: JSP Model 1 и JSP Model 2. В первом случае для обработки запроса и формирования ответа используется jsp-страница, во втором – запрос обрабатывается сервлетом, а ответ формируется jsp-страницей.

4. Страница JSP, кроме обычных html-тегов, содержит специальные теги следующих категорий: директивы, объявления, скриплеты, выражения и комментарии. Эти теги предназначены для управления процессом компиляции jsp-страницы в сервлет.

5. Программисту JSP доступны для применения в скриплетах специальные объекты, созданные контейнером JSP и называемые неявными объектами. Практически каждому неявному объекту JSP можно поставить в соответствие равносильный объект сервлета, который обладает похожими свойствами. Это делает технику программирования jsp-страниц очень похожим на программирование сервлетов.

6. Библиотека тегов – это одна из технологий, входящих в состав спецификации JSP, которая позволяет создавать собственные (пользовательские) теги для использования их при разработке jsp-страниц. Разработка библиотеки сводится к созданию дескриптора библиотеки (xml-файла, описывающего теги-элементы библиотеки) и разработке java-классов (обработчики тегов, генерирующих html-текст, замещающий пользовательские теги при компиляции jsp-страницы).

7. Каждый web-сервер может дополнять спецификацию JSP своим набором стандартных действий (в виде дополнительных тегов), распознаваемых компилятором контейнера JSP. Существует стандартный набор действий, который обязательно должен поддерживать любой web-сервером. Каждое такое действие оформлено в виде тега в нотации XML и использует пространство имен jsp. Функциональные возможности стандартных действий перекрывают и расширяют возможности классических тегов JSP.

8. Существует стандартная библиотека тегов JSTL, обладающая широкой функциональностью и доступная для скачивания с сайта [20]. Кроме стандартной библиотеки, на этом же сайте доступно более 10 различных специализированных библиотек тегов. Применение готовых библиотек может значительно снизить трудоемкость разработки jsp-страниц.

9. При разработке компонента web-приложения выбор между технологией JSP и технологией сервлетов необходимо склоняться к JSP в том случае, если предполагается малый объем вычислений и большой объем html-вывода в стандартный поток.



## Глава 5. СРЕДА ВЫПОЛНЕНИЯ WEB-ПРИЛОЖЕНИЯ

### 5.1. Предисловие к главе

Сервлеты и jsp-страницы, входящие в состав web-приложения, исполняются в специальной среде, называемой web-контейнером, и не могут (вернее, это запрещено спецификацией) взаимодействовать с внешним миром и между собой непосредственно. Более того, всем жизненным циклом сервлетов и jsp-страниц управляет контейнер, вызывая реализованные ими методы стандартных интерфейсов. Любое нарушение этих правил приводит к утрате свойства переносимости приложения.

Разработчику web-приложения контейнер представляется в виде predetermined объектов, которые передаются в виде параметров, ссылок или неявных объектов. На рис. 5.1 изображена объектная модель среды выполнения web-приложения.

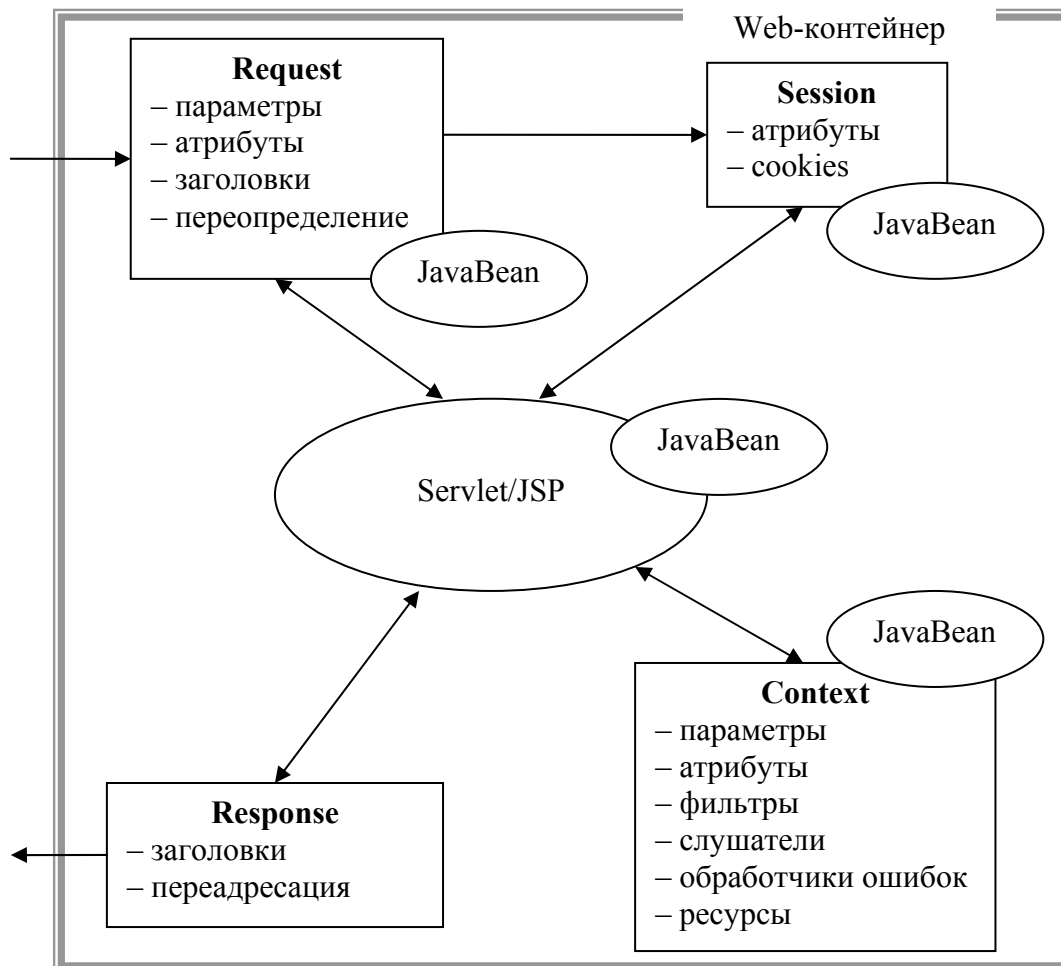


Рис. 5.1. Модель среды выполнения web-приложения

Объект **Request** создается контейнером при получении http-запроса к компоненту web-приложения и инкапсулирует всю необходимую информацию о запросе клиента. Этот объект существует и доступен только в рамках обработчика запроса (в нашем случае сервлета или jsp-страницы).

Объект **Response** создается контейнером тоже при получении запроса. Методы этого объекта позволяют инкапсулировать всю информацию, необходимую контейнеру, для того, чтобы сформировать и передать ответ клиенту. Этот объект существует и доступен только в рамках обработчика запроса.

Объект **Context** создается контейнером при его инициализации на основе дескриптора развертывания приложения. Помимо общего контекста создаются контексты для каждого сервлета и jsp-страницы.

Говорят, что два компонента web-приложения работают в одном контексте, если они связаны с общим дескриптором развертывания web-приложения. Для web-программиста контекст приложения – это, прежде всего, информация, характеризующая само web-приложение и среду, в которой оно работает. Эту информацию можно разделить на две части: статическую и динамическую. Статическая часть описывается в дескрипторе развертывания и в других конфигурационных файлах web-контейнера. Эта информация не чувствительна к перезагрузке сервера. Динамическая часть информации может быть добавлена, изменена или удалена с помощью специальных методов контекста и перезагрузка web-сервера приводит к ее разрушению.

Объект **Session** создается контейнером при получении первого запроса клиента. Объект существует до тех пор, пока интервал времени между последовательными запросами клиента не превысит установленное пороговое значение. Для обозначения каждой сессии используется специальный уникальный идентификатор, позволяющий разделить запросы разных клиентов.

Методы перечисленных объектов позволяют не только извлекать информацию о среде выполнения приложения, но и изменять ее, а также динамически создавать новые свойства этих объектов.

## 5.2. Контекст web-приложения

Наиболее часто web-программист сталкивается с необходимостью использовать параметры инициализации (статический элемент контекста) и атрибуты контекста (динамический элемент контекста).

### 5.2.1. Параметры инициализации контекста

Параметры инициализации контекста web-приложения описываются в дескрипторе web-приложения и используются для его настройки. Например, значением этого параметра может быть URL базы данных или внешнего для web-приложения сервлета. Параметры инициализации следует использовать для хранения данных, которые нецелесообразно «зашивать» в программный код из-за того, что они могут измениться при установке и перенастройке web-приложения. Размещение таких данных за пределами (в данном случае в дескрипторе развертывания) программного кода избавляет от необходимости поддержки нескольких версий одной и той же программы.

На рис. 5.2 приведен фрагмент дескриптора web-приложения, содержащего описание параметра инициализации.

```
<!--..... -->
display-name>ANaive</display-name>
<context-param>
    <description>Servlet URL</description>
    <param-name> ServletURL</param-name>
    <param-value>http://smwcore2:8080/owpj5/Rrr</param-value>
</context-param>
<servlet>
    <servlet-name>Ppp</servlet-name>
    <servlet-class>Ppp</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Ppp</servlet-name>
    <url-pattern>/PostExecute</url-pattern>
</servlet-mapping>
<!--..... -->
<welcome-file-list>
    <welcome-file>persondata.html</welcome-file>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
<!--..... -->
```

Рис. 5.2. Фрагмент дескриптора развертывания web-приложения с параметром инициализации контекста

Описание параметров инициализации контекста в дескрипторе приложения должно предшествовать описанию сервлетов и JSP. Каждый параметр описывается в дескрипторе развертывания приложения с помощью тегов, заключенных между тегами **<context-param>** и **</context-param>**.

Тег **<description>** не является обязательным и используется для комментариев. Теги **<param-name>** и **<param-value>** используются для задания имени и значения параметра инициализации. Именно это имя

используется в качестве параметра при вызове метода **getInitParameter** для получения значения параметра.

На рис. 5.3 приведен пример сервлета, который получает значение параметра **ServletURL** инициализации контекста и использует его в качестве URL для get-вызова сервлета.

```
import org.apache.commons.httpclient.*;
import org.apache.commons.httpclient.methods.*;

public class Ppp extends HttpServlet implements Servlet {
    protected void doPost(HttpServletRequest rq,
        HttpServletResponse rs)
        throws ServletException, IOException {
        GetMethod gm;
        HttpClient hc = new HttpClient();
        String firstname = rq.getParameter("firstname");
        String lastname = rq.getParameter("lastname");
        String press = rq.getParameter("press");
        ServletContext sc = getServletContext();
        String uri = sc.getInitParameter("ServletURL")+"?";
        if (press.equalsIgnoreCase("OK")) {
            uri+= ( "firstname=" + firstname + "&"
                + "lastname=" + lastname);
        } else
            uri += "/index.html";
        hc.executeMethod(gm = new GetMethod(uri));
        rs.setContentType("text/html");
        PrintWriter pw = rs.getWriter();
        pw.println(gm.getResponseBodyAsString());
        pw.flush();
    }
}
```

Рис. 5.3. Пример сервлета, извлекающего значение параметра инициализации контекста

На рис. 5.4 демонстрируется получение параметра инициализации в jsp-странице.

```
<!-- ..... -->
<body>
<% String parm = request.getParameter("parm");
String uri = getServletContext().getInitParameter("ServletURL");
%>
<br /> <%=parm%>
<br /> <%=uri%>
</body>
<!-- ..... -->
```

Рис. 5.4. Фрагмент jsp-страницы, извлекающей значение параметра инициализации контекста

На рис. 5.5 приводится фрагмент дескриптора развертывания приложения, содержащего несколько параметров инициализации контекста, а на рис. 5.6 – фрагмент страницы JSP, которая выводит в стандартный поток все имена и значения параметров инициализации.

```
<!-- ..... -->
<context-param>
  <param-name>parm_1 </param-name>
  <param-value>1</param-value>
</context-param>
<context-param>
  <param-name>parm_2</param-name>
  <param-value>2</param-value>
</context-param>
<context-param>
  <param-name>parm_3</param-name>
  <param-value>3</param-value>
</context-param>
<!-- ..... -->
```

Рис. 5.5. Фрагмент дескриптора развертывания web-приложения с тремя параметром инициализации контекста

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"
import="java.util.*" %>
<!-- ..... -->
<body>
<%
ServletContext sc = getServletContext();
Enumeration en = sc.getInitParameterNames();
String x;
while (en.hasMoreElements()) {
  x = (String) en.nextElement(); %>
  <br /><%=x+"=" %><%=sc.getInitParameter(x)%>
<%}%>
<!-- ..... -->
</body>
<!-- ..... -->
```

Рис. 5.6. Фрагмент jsp-страницы, выводящей в окно web-браузера все имена и значение параметров инициализации контекста

Следует отметить, что помимо параметров инициализации контекста web-приложения, есть возможность использовать параметры инициализации конкретного сервлета. Доступ к значению параметра в этом случае осуществляется с помощью метода **getInitParameter**

класса **HttpServlet** точно таким же способом, как и в случае с параметрами инициализации контекста.

### 5.2.2. Атрибуты контекста

Атрибуты являются динамическими элементами контекста и могут быть установлены, изменены или удалены в методах сервлетов или в скриплетах JSP.

На рис. 5.7 приводится фрагмент сервлета, использующего атрибуты контекста.

```
// .....
public void init() throws ServletException {
    super.init();
    this.Ab = new ABean("qwerty");
    ServletContext sc = getServletContext();
    String a1 = "a1";
    Integer n1 = 3;
    sc.setAttribute("atr_a1", a1);
    sc.setAttribute("atr_n1", n1);
    sc.setAttribute("atr_Ab", Ab);
}

protected void service(HttpServletRequest rq,
    HttpServletResponse rs)
    throws ServletException, IOException {
    ServletContext sc = getServletContext();
    PrintWriter pw = rs.getWriter();
    Enumeration en = sc.getAttributeNames();
    String x;
    while (en.hasMoreElements()) {
        x = (String) en.nextElement();
        pw.println("<br />" + x + " = " + sc.getAttribute(x));
    }
}
// .....
```

Рис. 5.7. Фрагмент сервлета, устанавливающего и считывающего значения атрибутов контекста

В методе **init** сервлета осуществляется установка значений атрибутов с помощью метода контекста **setAttribute**. Первый параметр этого метода задает имя атрибута, второй – его значение. Обратите внимание на атрибут с именем **atr\_Ab**, его значением является ссылка на объект java-класса с именем **ABean**.

В методе **service** сервлета с помощью методов контекста **getAttributeNames** и **getAttribute** в окно браузера выводится имена всех атрибутов и их значения. Следует знать, что некоторые атрибуты могут создаваться и устанавливаться самими web-контейнером. Для того чтобы не изменить или не удалить их, необходимо в документации

чтобы не изменить или не удалить их, необходимо в документации сервера выяснить имена этих атрибутов и не использовать их в своих приложениях.

Метод **setAttribute** предназначен как для создания атрибута, так и для его корректировки. При необходимости удалить атрибут используют метод **removeAttribute**.

На рис. 5.8 приводится пример java-класса с именем **ABean**. Объект этого класса создавался в методе **init** сервлета (рис. 5.7), а ссылка на объект этого класса сохранялась как значение атрибута с именем **atr\_Ab**.

```
public class ABean {
    protected String s = null;
    public ABean(String s) {
        this.s = s;
    }
    public String GetS() {
        return (this.s);
    }
    public void SetS(String s) {
        this.s = s;
    }
    public String toString () {
        return (this.s);
    }
}
```

Рис. 5.8. Пример класса простейшего JavaBean-объекта

Применение такого приема программирования является типичным. Связав ссылку на объект с именем атрибута, мы получаем возможность получать эту ссылку на объект и использовать его методы в любом приложении контекста.

### 5.2.3. Обработка исключений

В некоторых случаях web-приложения выдают сообщения об ошибках, которые хотелось бы скрыть от пользователей. В распоряжении разработчиков имеются специальные средства, позволяющие обрабатывать ошибки времени исполнения java-кода и коды состояния HTTP.

Настройка обработки кодов исключения или кодов состояния HTTP осуществляется с помощью тега **<error-page>** в дескрипторе развертывания приложения. С помощью этого тега можно установить соответствие между типом исключения или кодом состояния и web-ресурсом, обрабатывающим это событие. В качестве ресурса можно указать сервлет, jsp- или html-страницы.

Более подробно с методами и примерами обработки исключений и кодов состояния в web-приложении можно ознакомиться в источнике [7].

### 5.3. Запрос

При http-запросе информация передается от клиента к серверу в виде http-заголовков и тела сообщения запроса. Web-контейнер, получив запрос, создает объект типа **HttpServletRequest**, который затем передается методу **service** сервлета в качестве параметра или представляется как неявный объект **request** в JSP.

Объект типа **HttpServletRequest** инкапсулирует базовый http-запрос и предоставляет методы, позволяющие определить тип запроса, используемую кодировку, MIME-тип, ip-адрес клиента, номер порта и т. п., обработать параметры, атрибуты, заголовки, тело сообщения запроса.

#### 5.3.1. Параметры запроса

Параметрами запроса являются строковые данные, которые передаются от клиента к web-контейнеру в составе запроса. Параметры сохраняются контейнером как набор пар «имя – значение». В общем случае одному имени может соответствовать несколько значений. Доступ к параметрам обеспечивается методами **getParameter**, **getParameterValues**, **getParameterNames** интерфейса **HttpServletRequest**.

Метод **getParameter** возвращает первое значение заданного параметра или **null**, если такого параметра нет.

Метод **getParameterValues** возвращает все значения заданного параметра в виде массива или **null**.

С помощью метода **getParameterNames** можно получить список всех передаваемых в запросе параметров.

В разд. 3 и 5 приведены и разобраны примеры сервлетов и jsp-страниц, обрабатывающих параметры запроса. Для более полного знакомства рекомендуется книга [7] и спецификация Servlet API [12].

#### 5.3.2. Атрибуты запроса

Атрибутами запроса могут быть любые объекты, ассоциируемые с запросом. Для доступа к атрибутам запроса используются следующие методы интерфейса **HttpServletRequest**: **getAttribute** (получить значение атрибута), **setAttribute** (установить значение атрибута), **getAttributeNames** (получить список имен атрибутов), **removeAttribute** (удалить атрибут).

На рис. 5.9 приведен фрагмент сервлета, устанавливающего значение трех атрибутов – **atr1**, **atr2** и **atr3** перед выполнением переопределения запроса на jsp-страницу. При вызове таким способом страницы JSP ей передается объект запроса, с помощью методов которого в скрипте jsp-страницы доступны ассоциированные с запросом атрибуты.



```

public class Hhh extends HttpServlet implements Servlet {

protected void doGet(HttpServletRequestRequest rq,
    HttpServletResponse rs)
    throws ServletException, IOException {
    String is = "qwerty";
    Integer ia = 12345;
    jspclass.Regimex oa = new jspclass.Regimex();
    rq.setAttribute("atr1", "qwerty");
    rq.setAttribute("atr2", ia);
    rq.setAttribute("atr3", oa);
    rq.getRequestDispatcher("/jjj.jsp").forward(rq, rs);
    }
}

```

Рис. 5.9. Пример класса сервлета, устанавливающего атрибуты запроса

На рис. 5.10 приведен фрагмент jsp-страницы, в скрипте которой осуществляется доступ к атрибутам запроса с помощью методов неявного объекта **request**.

```

<!-- ..... -->
<body>
<%
String vatr1 = (String)request.getAttribute("atr1");
Integer vatr2 = (Integer)request.getAttribute("atr2");
jspclass.Regimex vatr3 =
    (jspclass.Regimex)request.getAttribute("atr3");
%>
<br> vatr1 = <%=vatr1%>
<br> vatr2 = <%=vatr2%>
<br> vatr2 = <%=vatr3.GetHOURL() %>
</body>
<!-- ..... -->

```

Рис. 5.10. Фрагмент jsp-страницы, обрабатывающей атрибуты запроса

Другой важный момент – это применение в качестве значения атрибута ссылки на объект. В этом примере атрибут с именем **atr3** является ссылкой на объект типа **Regimex**. После приведения результата выполнения метода **getAttribute** к типу **Regimex** можно использовать все методы этого класса.

На рис. 5.11 представлен фрагмент сервлета, выводящего в стандартный поток все существующие атрибуты запроса и их значения. Список всех атрибутов получен с помощью метода **getAttributeNames**, который возвращает объект типа **Enumeration**, позволяющий организовать цикл перебора всех имен атрибутов.

```

<!-- ..... -->
protected void service(HttpServletRequest rq,
    HttpServletResponse rs)
    throws ServletException, IOException {
    PrintWriter pw = rs.getWriter();
    Enumeration en = rq.getAttributeNames();
    String x;
    while (en.hasMoreElements()) {
        x = (String) en.nextElement();
        pw.println("<br />" + x + "= " + rq.getAttribute(x));
    }
}
<!-- ..... -->

```

Рис. 5.11. Фрагмент сервлета, обрабатывающего атрибуты запроса

На рис. 5.12 представлен результат работы сервлета на рис. 5.11.

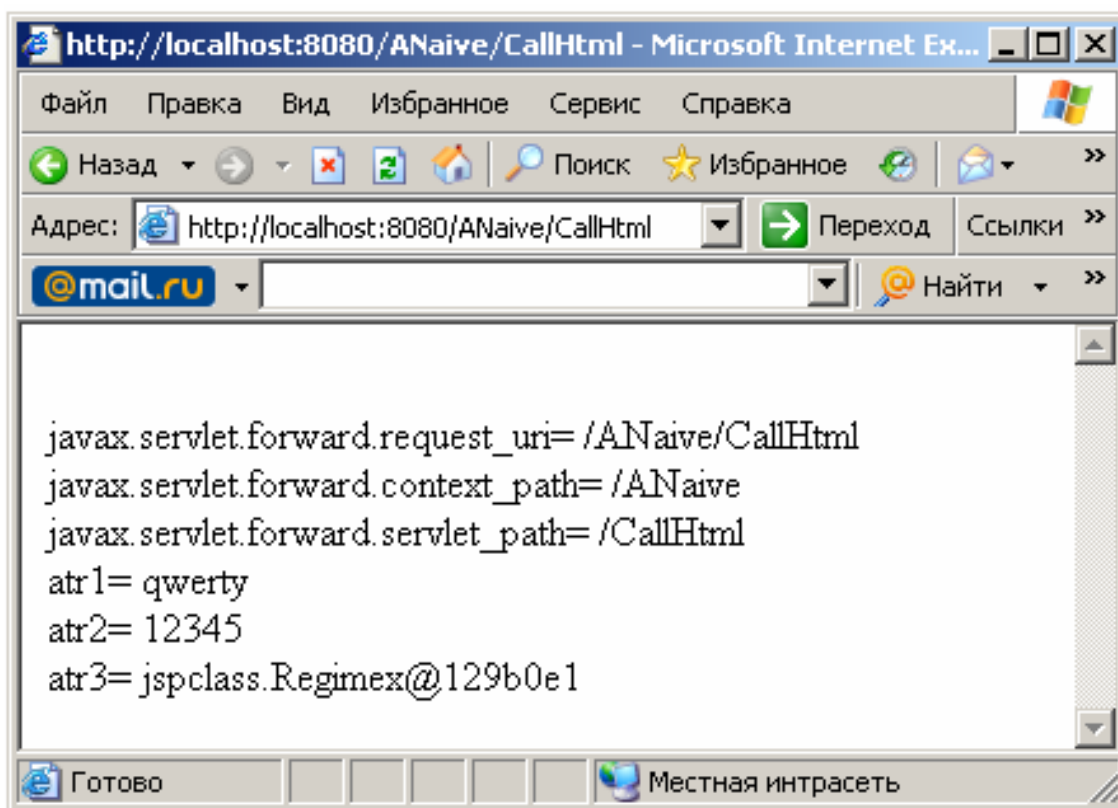


Рис. 5.12. Результат работы сервлета, обрабатывающего атрибуты запроса

При задании имен атрибутов соблюдается соглашение о наименовании пакетов в языке Java. Кроме того, имена атрибутов, начинающиеся с префиксов «java.» или «javax.», зарезервированы для определения атрибутов спецификации сервлетов, а имена с префиксом «sun.»

и «com.sun» зарезервированы компанией Sun Microsystems для собственных атрибутов.

Обратите внимание на то, что помимо установленных в сервлете (рис. 5.9) значений атрибутов, в окно браузера выведены имена и значения еще трех атрибутов запросов, созданных контейнером.

### 5.3.3. Заголовки запроса

Заголовки http-запроса представляют собой предшествующие телу запроса пары «имя – значение», которые формируются клиентом web-приложения и пересылаются серверу. В общем случае одному заголовку может соответствовать несколько значений.

В простом сервлете или jsp-странице разработчику доступны только три метода для работы с заголовками запроса: **getHeader** (получить первое значение заголовка по его имени), **getHeaders** (получить все значения заголовка по имени) и **getHeaderNames** (получить все имена заголовков запроса).

Если же запрос формируется с помощью классов пакета **httpclient** (гл. 3.4), то метод **addRequestHeader** объекта типа **GetMethod** (или **PostMethod**) позволяет добавить собственные заголовки запроса, как это сделано в примере на рис. 5.13.

```
//.....
protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {

    HttpClient hc = new HttpClient();
    String uri = "http://localhost:8080/ANaive/Jjj";
    GetMethod gm = new GetMethod(uri);
    gm.addRequestHeader("MyHeader1", "MyHeaderValue1");
    gm.addRequestHeader("MyHeader2", "MyHeaderValue2");
    gm.addRequestHeader("MyHeader3", "MyHeaderValue3");
    hc.executeMethod(gm);
    response.setContentType("text/html");
    InOutServlet inout = new InOutServlet(response.getOutputStream(),
        gm.getResponseBodyAsStream());
    inout.perform();
}
//.....
```

Рис. 5.13. Фрагмент сервлета, формирующего заголовки запроса

На рис. 5.13 представлен фрагмент сервлета, формирующего три дополнительных заголовка запроса с именами **MyHeader1**, **MyHeader2**, **MyHeader3**. В данном примере все значения заголовков имеют строковый формат.

Пусть сформированный http-запрос отправляется на обработку, сервлету **Jjj**, фрагмент которого представлен на рис. 5.14, а ответ, сформированный сервлетом **Jjj**, обрабатывается и выводится в окно браузера снова в вызывающем сервлете (рис. 5.13).

```
// .....  
  
protected void service(HttpServletRequest rq,  
    HttpServletResponse rs)  
    throws ServletException, IOException {  
  
    PrintWriter pw = rs.getWriter();  
    String s;  
    rs.setContentType("text/html");  
    Enumeration enh = rq.getHeaderNames();  
    while (enh.hasMoreElements()) {  
        s = (String) enh.nextElement();  
        pw.println("<br />" + s + "=" + rq.getHeader(s));  
    }  
  
    // .....
```

Рис. 5.14. Фрагмент сервлета, обрабатывающего заголовки запроса

В фрагменте сервлета на рис. 5.14 используется методы **getHeaderNames** и **getHeader** для формирования ответа, который направляется в вызывающий сервлет (рис. 5.13). Вывод сервлета на рис. 5.13 в окно браузера будет примерно таким, как на рис. 5.15.

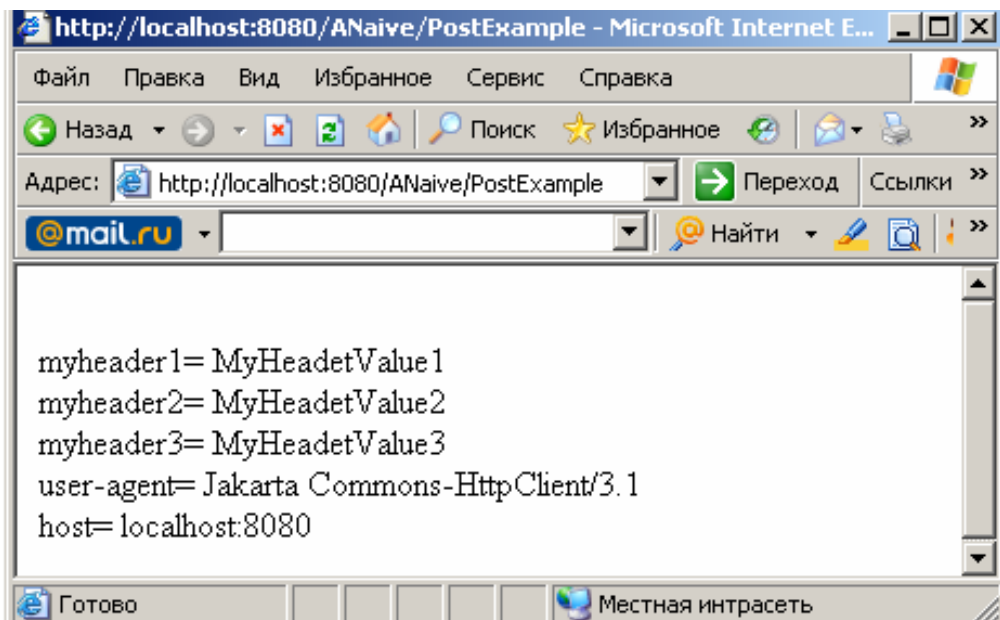


Рис. 5.15. Наименования и значения заголовков запроса

Кроме строкового, значения заголовков запроса могут быть целочисленного типа или типа `Date`. В этих случаях можно воспользоваться двумя другими разновидностями метода **getHeader**: **getIntHeader** или **getDateHeader**.

#### 5.3.4. Объект **RequestDispatcher**

Объект **RequestDispatcher** может быть создан в сервлете и применяется для переопределения запроса типа **GET** (см. 3.3.4). Для создания этого объекта применялся метод **getRequestDispatcher** объекта запроса **HttpServletRequest**.

В дополнение лишь отметим, что создание этого объекта также возможно с помощью методов **getNamedDispatcher** и **getServletDispatcher**, а кроме метода **forward** объекта **RequestDispatcher** можно использовать метод **include**, позволяющий включать в выходной поток данные.

Более детально с особенностями применения объекта **RequestDispatcher** можно ознакомиться в книге [7] и спецификации Servlet API [12].

### 5.4. Ответ

**Ответ** – это объект типа **HttpServletResponse**, который инкапсулирует информацию, передаваемую от сервлета или JSP клиенту. В качестве клиента может выступать как web-браузер, так и любой другой компонент web-приложения, поддерживающий http-протокол. Объект ответа создается web-контейнером после получения запроса. В сервлет ответ передается в виде второго параметра метода **service**, а в JSP он представлен в виде неявного объекта **response**.

#### 5.4.1. Заголовки ответа

Как и http-запрос, http-ответ всегда можно дополнить собственными заголовками, которые потом могут быть получены и обработаны на стороне клиента. На рис. 5.16 изображен фрагмент сервлета, формирующего заголовки ответа.

В общем случае заголовок с одним именем может соответствовать несколько значений. Заголовки могут быть сформированы с помощью метода **addHeader**, как это сделано в примере на рис. 5.16, или метода **setHeader**. В первом случае заголовок создается (если заголовок с таким именем нет) или добавляется следующее значение (если такой

заголовок уже есть). Метод **setHeader** тоже создает новый заголовок (если заголовка с таким именем нет) или заменяет значение.

```
// .....  
  
protected void service(HttpServletRequest rq,  
    HttpServletResponse rs)  
    throws ServletException, IOException {  
    rs.setContentType("text/html");  
    rs.addHeader("Jjj-Header1", "Jjj-HeaderValue1");  
    rs.addHeader("Jjj-Header2", "Jjj-HeaderValue2");  
}  
// .....
```

Рис. 5.16. Фрагмент сервлета, формирующего заголовки ответа

Доступ к заголовкам ответа осуществляется точно по такому же принципу, что и к заголовкам запроса. На рис. 5.17 приводится вывод заголовков ответа, полученных с помощью метода **getHeaderNames** и **getHeader**.

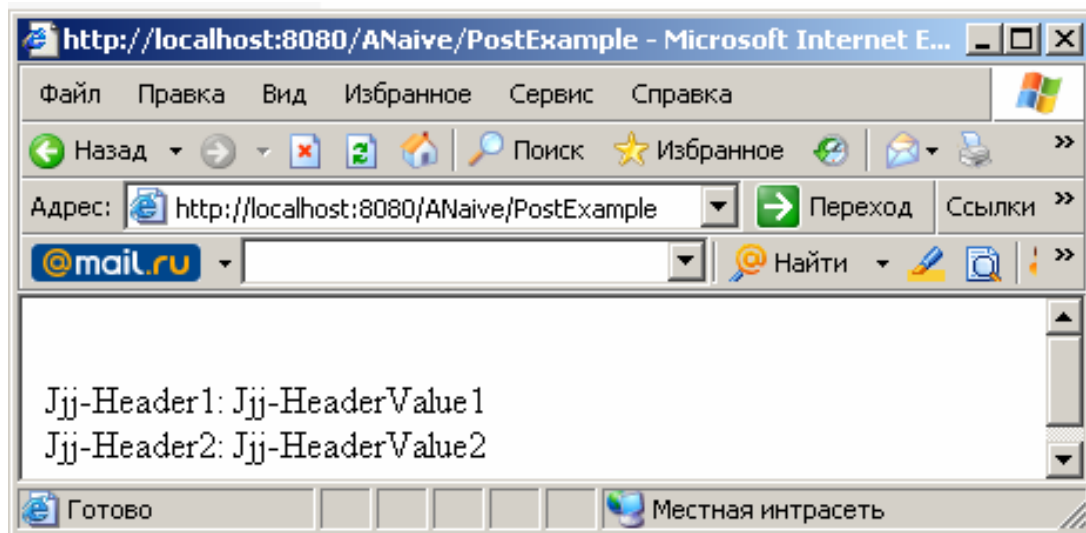


Рис. 5.17. Заголовки ответа

Перед использованием собственных заголовком ответа, как и в случае заголовков запроса, следует ознакомиться с существующими ограничениями на их имена, накладываемые протоколом HTTP и web-сервером.

Кроме перечисленных методов, объект типа **HttpServletResponse** обладает рядом других полезных свойств и методов, с которыми можно ознакомиться в спецификации Servlet API [12].

#### 5.4.2. Переадресация

В разд. 3, а также 5.3.3 уже обсуждался механизм переопределения запроса, позволяющий передать управление от одного сервлета другому. Существует еще один способ передачи управления – с помощью переадресации.

Принципиальное отличие переадресации от переопределения запроса заключается в том, что передача управления от сервлета А к сервлету В осуществляется через посредника – клиента сервлета А.

На рис. 5.18 изображен процесс переадресации. Штриховыми линиями изображается вызов браузером сервлетов, штрихпунктирными – переадресация, а сплошными – движение информации. Браузер вызывает сервлет **Ppp**, в котором в зависимости от некоторого условия осуществляется переадресация на сервлет **Mmm** или сервлет **Fff**. Процесс переадресации осуществляется через клиента (в этом случае это браузер) сервлета **Ppp**. При выполнении переадресации браузером всегда генерируется запрос типа **GET**.

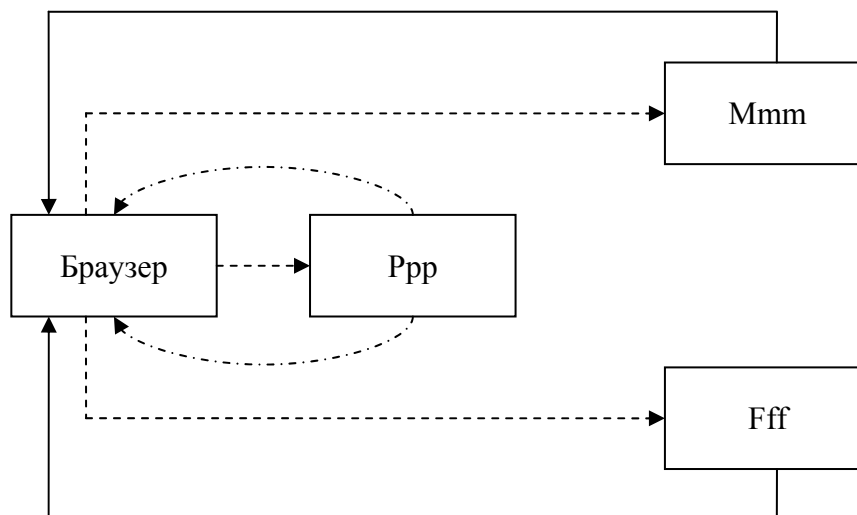


Рис. 5.18. Схема взаимодействия сервлетов при переадресации запроса

Переадресация выполняется с помощью метода **sendRedirect** объекта типа **HttpServletResponse**. На рис. 5.19 представлен фрагмент сервлета, выполняющего переадресацию запроса. В качестве параметра метода **sendRedirect** указывается URI нового ресурса. Ресурсом, на который осуществляется переадресация, в общем случае, может быть любой компонент приложения, допускающий запрос типа **GET**.

```
// .....

protected void service(HttpServletRequest rq, HttpServletResponse rs)
    throws ServletException, IOException {

// .....
    rs.sendRedirect("http://localhost:8080/ANaive/CallHtml");
}

// .....
```

Рис. 5.19. Фрагмент сервлета, осуществляющего переадресацию

Следует отметить, что переадресация является функцией протокола HTTP. Метод **sendRedirect** просто генерирует http-ответ с кодом **303 (See Other)**, который сопровождается URI нового ресурса. Если в качестве клиента выступает сервлет, то при получении ответа он должен анализировать код возврата и в случае необходимости генерировать запрос на новый ресурс.

## 5.5. Сеанс связи (сессия)

Протокол HTTP является *stat less* протоколом, т. е. протоколом, не сохраняющим информацию о своем состоянии. Это означает, что каждый запрос и ответ имеют свой жизненный цикл, никак не связанный с предшествующими им запросами и ответами. Поэтому управление сеансом связи с пользователем является важной и нетривиальной задачей.

Объект **Session** (сеанс связи, или сессия) реализует интерфейс **HttpSession** и служит для представления пользователя, работающего с клиентской частью web-приложения. Объект сессии создается, как правило, web-контейнером и становится доступным в сервлете или JSP с помощью метода **getSession** объекта **Request**.

Время существования сессии зависит от параметра **Session timeout**, который может быть установлен в дескрипторе развертывания или в конфигурационных файлах сервера и может быть изменен динамически, например в сервлете. Этот параметр устанавливает максимальное время между запросами одного клиента. Превышение этого значения приводит к удалению объекта **Session**.

В принципе, объект сессии может разрушить и изнутри, выполнив метод **invalidate** интерфейса **HttpSession**.

Во время сеанса связи любой объект, связанный с сеансом связи, доступен любому сервлету или JSP, находящемуся в этом же контексте,



и недоступен для сервлетов и JSP другого контекста. Состояние сеанса позволяет отследить два специальных механизма: **cookies** и **URL rewriting**. В первом случае информация о сессии записывается в специальном файле на компьютере клиента, во втором случае такая же информация записывается непосредственно в URL каждого вызова.

### 5.5.1. Идентификатор сессии

С каждой сессией связан специальный уникальный идентификатор, который может быть получен с помощью метода **getId** интерфейса **HttpSession**. Идентификатор представляет собой строку, состоящую из 32 символов. Все приемы программирования, направленные на сохранение информации о состоянии сессии, в итоге опираются на использование этого идентификатора.

### 5.5.2. Атрибуты сессии

Как и с контекстом приложения и запросом, с сессией (с объектом, реализующим интерфейс **HttpSession**) можно динамически связать атрибуты. Создание, корректировка и удаление атрибутов осуществляется точно таким же способом, как и в предыдущих случаях. Единственным отличием является то, что методы, выполняющие эти операции, являются собственностью интерфейса **HttpSession**.

На рис. 5.20 представлен фрагмент сервлета, создающего атрибут сессии, имеющего имя, которое совпадает с идентификатором сессии.

```
// .....
public class Sss extends HttpServlet implements Servlet {

    protected void service(HttpServletRequest rq, HttpServletResponse rs)
        throws ServletException, IOException {

        HttpSession ss = rq.getSession();
        String idss = ss.getId();
        if (ss.getAttribute(idss) == null) {
            mybeans.ABean ab = new mybeans.ABean("qwerty");
            ss.setAttribute(idss, ab);
        }

        // .....

        rq.getRequestDispatcher("/Jjj.jsp").forward(rq, rs);
    }
}
// .....
```

Рис. 5.20. Пример сервлета, использующего атрибуты сессии для сохранения своего состояния

Перед созданием осуществляется проверка на существование атрибута с таким именем, а в качестве значения атрибута используется ссылка на java-объект.

На рис. 5.21 представлена страница JSP, на которую выполняется переопределение запроса из сервлета, рассмотренного в предыдущем примере. Объект сессии в jsp-странице становится доступным с помощью метода **getSession** неявного объекта **request**.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-
1">
<title>Insert title here</title>
</head>
<body>
<%
HttpSession ss = request.getSession();
String idss = ss.getId();
mybeans.ABean ab = (mybeans.ABean)ss.getAttribute(idss);
%>
<br> <%=ab.GetS()%>
</body>
</html>
```

Рис. 5.21. Применение атрибута сессии в jsp-странице

Прием программирования, представленный на рис. 5.20 и 5.21, используется для создания java-объектов, связанных с сессией.

С помощью таких объектов можно сохранять и использовать некоторое время информацию о состоянии сессии, но следует помнить, что она будет уничтожена вместе с объектом **Session**.

### 5.5.3. Файлы cookies

**Cookies** – это небольшие фрагменты текстовой информации, которым обменивается web-сервер и браузер в процессе установки соединения. Получая информацию в виде cookies от браузера, сервер может «узнать» клиента и предоставить ему заранее обусловленный перечень услуг. Объем **cookie** ограничен 4 Кб.

Для отправки **cookie** от сервлета браузеру используется метод **addCookie** интерфейса **HTTPServletResponse**. Извлечь **cookie**, отправленный браузером, можно с помощью метода **getCookie** интер-

фейса **HttpServletRequest**. Работа с **cookie** поддерживается классом **Cookie**.

Более подробно о применении этой технологии можно ознакомиться в книге [7] и в спецификации Servlet API [12].

## 5.6. Фильтры

**Фильтр** – это класс Java, реализующий интерфейс **Filter** и выполняющий роль препроцессора запроса. С помощью фильтра можно осуществить доступ к ресурсу непосредственно перед запросом, блокировать, предварительно обработать или модифицировать запрос, изменить ответ и т. п.

Фильтры могут быть использованы для различных ресурсов: сервлетов, jsp-страниц, html-страниц и др. Один и тот же фильтр может быть назначен разным ресурсам. С другой стороны, одному ресурсу может быть назначено несколько фильтров – в этом случае они образуют *цепочку фильтров*.

Технология фильтров основывается на использовании трех основных интерфейсов: **Filter**, **FilterChain** и **FilterConfig**.

Интерфейс **Filter** является важнейшим интерфейсом и предназначен для поддержки действий, связанных с основной функцией – фильтрацией запросов и ответов.

Интерфейс **FilterChain** реализуется web-контейнером и используется для организации цепочки фильтров. С помощью этого интерфейса можно заблокировать доступ к ресурсу.

Интерфейс **FilterConfig** тоже реализуется web-контейнером и предназначен для информирования фильтра о его окружении.

Любой фильтр должен реализовать три обязательных метода: **init**, **destroy** и **doFilter**. Все методы вызываются web-контейнером, а последовательность их вызова определяет жизненный цикл фильтра. На рис. 5.22 приведен пример простейшего фильтра.

Метод **init** вызывается контейнером во время инициализации фильтра – при загрузке web-сервера, а метод **destroy** – при завершении работы сервера.

Метод **doFilter** получает управление перед получением запроса фильтруемым ресурсом. В качестве параметров этому методу передаются объекты запроса и ответа, предназначенные для ресурса, а также объект с интерфейсом **FilterChain**.

```

package flt;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Flt1 implements Filter {

    public void init(FilterConfig cfg) throws ServletException {
        System.out.println("Flt1:init");
    }

    public void destroy() {
        System.out.println("Flt1:destroy");
    }

    public void doFilter(ServletRequest rq,
        ServletResponse rs,
        FilterChain ch)
        throws IOException, ServletException {
        FltRequest frq = new FltRequest((HttpServletRequest)rq);
        frq.SetAA("Hello from Flt1");
        System.out.println("Flt1:doFilter");
        ch.doFilter(frq, rs);
    }

}

```

Рис. 5.22. Пример простейшего фильтра

Если в результате работы фильтра предполагается предать запрос далее по цепочке фильтров или при единственном фильтре непосредственно ресурсу, то работа фильтра должна быть завершена вызовом метода **doFilter** интерфейса **FilterChain**. В другом случае (при блокировке доступа) этот метод не вызывается.

В примере на рис. 5.22 следует обратить внимание на способ обработки запроса. Здесь применяется *обертывание запроса* (wrapping). Пояснение этого приема программирования будет сделано ниже.

Перехват фильтром запроса к ресурсу осуществляется на основе информации в дескрипторе развертывания приложения. С помощью тега **<filter>** описывается имя и java-класс фильтра, а тег **<filter-mapping>** позволяет связать этот фильтр с фильтруемым ресурсом.

На рис. 5.23 приведен фрагмент дескриптора, описывающего фильтр для сервлета **Sss**.

```

<!-- ..... -->
<servlet>
    <servlet-name>Sss</servlet-name>
    <servlet-class>Sss</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Sss</servlet-name>
    <url-pattern>/Sss</url-pattern>
</servlet-mapping>
<!-- ..... -->
<filter>
    <filter-name>Flt1</filter-name>
    <filter-class>flt.Flt1</filter-class>
</filter>
<filter-mapping>
    <filter-name>Flt1</filter-name>
    <url-pattern>/Sss</url-pattern>
</filter-mapping>
<!-- ..... -->

```

Рис. 5.23. Пример описания фильтра в дескрипторе развертывания web-приложения

На рис. 5.24 представлен фрагмент фильтруемого ресурса (сервлета **Sss**). Следует обратить внимание на использование объекта запроса, для которого использовалось обертывание.

```

//.....
public class Sss extends HttpServlet implements Servlet {

    protected void service(HttpServletRequest rq,
        HttpServletResponse rs)
        throws ServletException, IOException {
        flt.FltRequest frq = (flt.FltRequest)rq;
        System.out.println("Sss:service:"+ frq.GetAA());

        //.....
    }
}

```

Рис. 5.24. Обработка обернутого запроса в сервлете

На рис. 5.25 представлен класс-обертка запроса. Наиболее близкий термину «обертывание» является термин «инкапсуляция». Для создания класса обертки запроса используется специальный класс-обертка **HttpServletRequestWrapper**, реализующий методы

интерфейса **HttpServletRequest**. Применение такой технологии позволяет снабдить запрос новыми свойствами или заместить существующие.

```
package flt;
import javax.servlet.http.*;

public class FltRequest extends
HttpServletRequestWrapper implements HttpServletRequest {
    protected String AA = null;
    protected String BB = null;
    public FltRequest(HttpServletRequest request) {
        super(request);
        if (request.getClass().getName().equals("flt.FltRequest")){
            this.AA = ((FltRequest)request).GetAA();
            this.BB = ((FltRequest)request).GetBB();
        }
    }
    public FltRequest(FltRequest request) {
        super((HttpServletRequest)request);
        this.AA = request.GetAA();
        this.BB = request.GetBB();
    }
    public void SetAA(String aa) {
        this.AA = aa;
    }

    public String GetAA() {
        return this.AA;
    }
    public void SetBB(String bb) {
        this.BB = bb;
    }
    public String GetBB() {
        return this.BB;
    }
}
```

Рис. 5.25. Пример простейшего класса-обертки запроса

Подменив методы и свойства объектов запроса и ответа сервлета на свои (т. е. поменяв поведение объектов), можно эффективно управлять работой фильтруемого ресурса. В спецификации сервлетов такой метод управления считается предпочтительным.

Если предполагается использовать цепочку фильтров для некоторого ресурса, то это должно быть отражено в дескрипторе развертывания приложения. На рис. 5.26 представлен фрагмент дескриптора, описывающего цепочку из двух фильтров к сервлету **Sss**, и фильтр к странице JSP.

```

<!-- ..... -->
<filter>
    <filter-name>Flt1</filter-name>
    <filter-class>flt.Flt1</filter-class>
</filter>
<filter-mapping>
    <filter-name>Flt1</filter-name>
    <url-pattern>/Sss</url-pattern>
</filter-mapping>
<filter>
    <filter-name>Flt2</filter-name>
    <filter-class>flt.Flt2</filter-class>
</filter>
<filter-mapping>
    <filter-name>Flt2</filter-name>
    <url-pattern>/Sss</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>Flt1</filter-name>
    <url-pattern>/jjj.jsp</url-pattern>
</filter-mapping>
<!-- ..... -->

```

Рис. 5.26. Пример описания нескольких фильтров в дескрипторе приложения

Более глубокие сведения о технологии фильтрования можно получить в книге [7] и спецификации Servlet API [12].

## 5.7. Слушатели событий

**Слушатель событий** – это класс Java, реализующий один или более интерфейсов слушателей событий интерфейсов.

Перечень интерфейсов слушателей и их назначение приведен в таблице.

Таблица

**Интерфейсы слушателей событий**

Идентификатор интерфейса	Обрабатываемые события
ServletContextListener	Создание и разрушение контекста
ServletContextAttributeListener	Добавление, удаление, модификация атрибута контекста
HttpSessionListener	Создание и уничтожение сеанса
HttpSessionAttributeListener	Добавление, удаление, модификация атрибута сессии

Как и фильтры, слушатели создаются и инициализируются web-контейнером при загрузке web-сервера. На рис. 5.27 представлен пример

слушателя, реализующего интерфейс **HttpSessionListener** и обрабатывающего события двух типов: создание и уничтожение сеанса. Для обработки событий используются два метода интерфейса **HttpSessionListener**: **sessionCreated** и **sessionDestroyed**.

```
package lst;

import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

public class Lst1 implements HttpSessionListener{

    public void sessionCreated(HttpSessionEvent se) {
        HttpSession ss = se.getSession();
        System.out.println("Lst1:sessionCreated:Id="
            +ss.getMaxInactiveInterval());
        ss.setMaxInactiveInterval(10);
        System.out.println("Lst1:sessionCreated:Id="+ss.getId());
    }

    public void sessionDestroyed(HttpSessionEvent se) {
        System.out.println("Lst1:sessionDestroyed:Id="
            +se.getSession().getId());
    }

}
```

Рис. 5.27. Пример простейшего слушателя с интерфейсом HttpSessionListener

Метод **sessionCreated** получает управление от web-контейнера сразу после создания объекта **Session**, а метод **sessionDestroyed** – сразу после удаления.

Обратите внимание, что в методе **sessionCreated** осуществляется изменение параметра **Session timeout**. Новое значение этого параметра будет равно 10 мин.

На рис. 5.28 приведен пример еще одного слушателя событий. Этот слушатель реализует интерфейс **ServletContextAttributeListener** и обрабатывает события трех типов: добавление, удаление и изменение атрибутов контекста приложения.

Для обработки событий используются три метода **attributeAdded** (обработка добавления атрибута), **attributeRemoved** (обработка удаления атрибута) и **attributeReplaced** (обработка изменения атрибута).

На рис. 5.9 представлен фрагмент дескриптора развертывания приложения, в котором описываются два слушателя событий, реализованных в виде java-классов с именами **lst.Lst1** и **lst.Lst2**.



```

package lst;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextAttributeEvent;
import javax.servlet.ServletContextAttributeListener;
import javax.servlet.http.HttpSessionListener;

public class Lst2 implements ServletContextAttributeListener{

    public void attributeAdded(ServletContextAttributeEvent e) {
        System.out.println("Lst2:attributeAdded:AttributeName= "
            +e.getName());
        System.out.println("Lst2:attributeAdded:AttributeValue "
            +e.getValue());
    }
    public void attributeRemoved(ServletContextAttributeEvent e) {
        System.out.println("Lst2:attributeRemoved:AttributeName= "
            +e.getName());
        System.out.println("Lst2:attributeRemoved:AttributeValue "
            +e.getValue());
    }
    public void attributeReplaced(ServletContextAttributeEvent e) {
        System.out.println("Lst2:attributeReplaced:AttributeName="
            +e.getName());
        System.out.println("Lst2:attributeReplaced:AttributeOldValue "
            +e.getValue());
        ServletContext sc = e.getServletContext();
        System.out.println("Lst2:attributeReplaced:AttributeNewValue "
            +sc.getAttribute(e.getName()));
    }
}

```

Рис. 5.28. Пример простейшего слушателя с интерфейсом  
ServletContextAttributeListener

```

<!-- ..... -->
<listener>
<listener-class>lst.Lst1</listener-class>
</listener>
<listener>
<listener-class>lst.Lst2</listener-class>
</listener>
<!-- ..... -->

```

Рис. 5.29. Фрагмент дескриптора развертывания  
web-приложения с описанием слушателей

Для более детального знакомства с методами создания и использования слушателей событий следует обратиться к спецификации Servlet API [12].

## 5.8. Методология построения многопользовательского web-приложения

Методология построения многопользовательского приложения для простоты изложения будет излагаться для сервлетов. Практически без изменений все сказанное здесь можно распространить и на страницы JSP.

Сложность создания многопользовательского web-приложения возникает в основном из-за трех моментов: 1) протокол HTTP является протоколом, не сохраняющим свое состояние (типа *stateless*); 2) все сеансы сервера разделяют один общий экземпляр сервлета; 3) управление работой приложения осуществляется web-контейнером с помощью API, ориентированного на события (*event-driven API*). Каждый из перечисленных моментов имеет и другую, положительную сторону: протокол HTTP прост для применения, использование одного экземпляра сервлета для многих сеансов не перегружает память web-сервера, а применение API, ориентированного на события, позволяет избежать многих деталей программирования сервлета.

Если не предпринимать никаких дополнительных мер, то одновременное использование одного сервлета многими сеансами (многими пользователями одного приложения) может привести к его непредсказуемой работе.

Например, пусть сервлет S в рамках сессии с идентификатором A складывает полученные с html-формы два числа – 2 и 3. Если в рамках другой сессии B, параллельно работающей с сессией A, с html-формы были считаны для сложения сервлетом S числа 7 и 10, то нет никакой гарантии, что результат в первом случае будет 5, а во втором 17. В обеих сессиях могут быть получены 6 вариантов ответа: 5, 9, 10, 12, 13 и 17.

Для того чтобы результаты работы сервлета были корректными, следует для каждой сессии отвести свою область памяти для хранения значений промежуточных результатов. В этом случае сервлет будет всегда манипулировать с экземпляром данных, отведенным для текущей сессии. Очевидно, что эти области памяти должны существовать и быть доступными как минимум на протяжении сеанса.

Наиболее простой способ решения этой проблемы уже демонстрировался ранее в примерах на рис. 5.20 и 5.21. Область памяти, которая отводится для каждой сессии, выделяется при первом вызове сервлета в виде *java*-класса. Хранение ссылки на этот класс удобно организовать в виде атрибута, имеющего имя, совпадающее с идентификатором сессии.

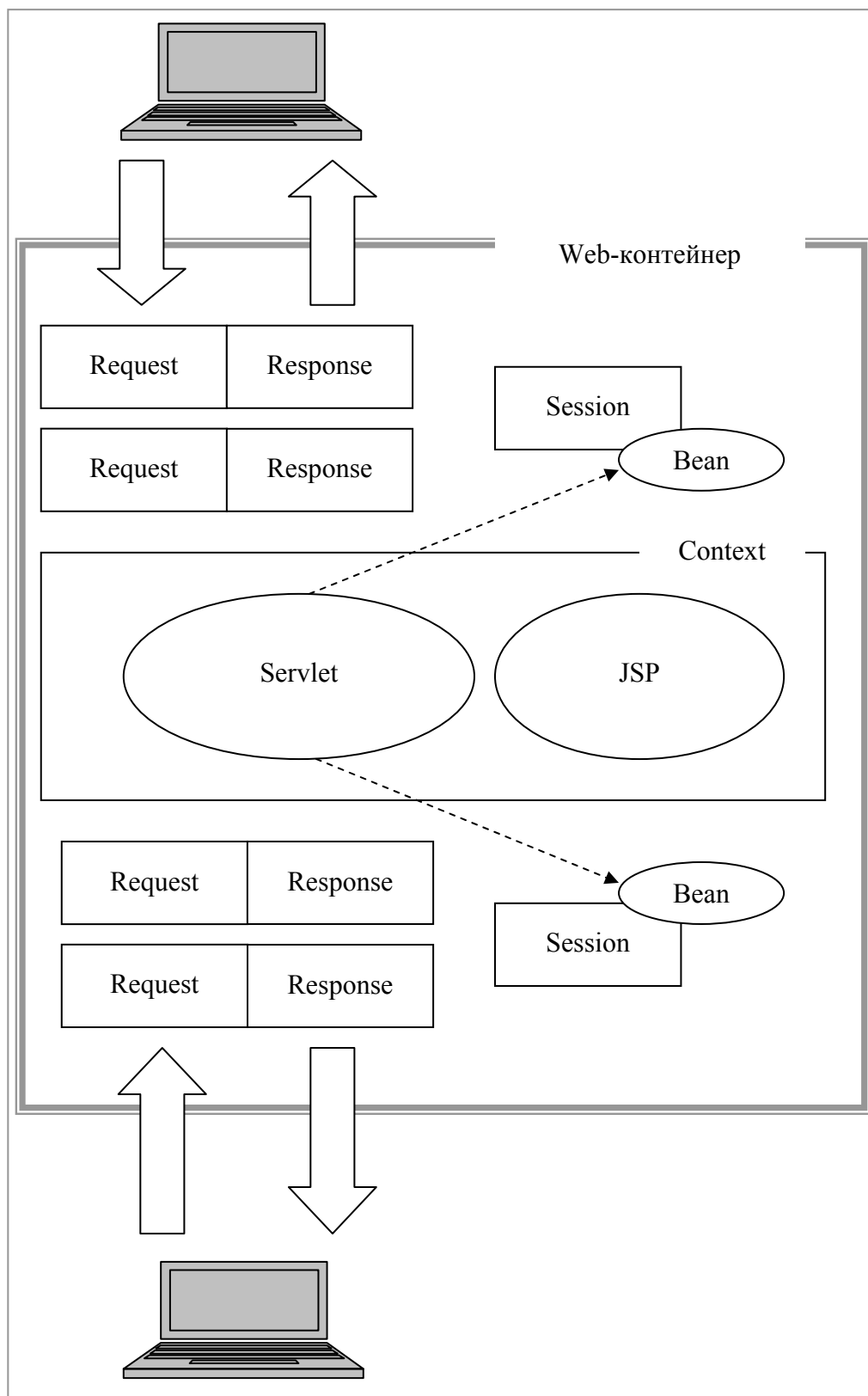


Рис. 5.30. Структурная схема многопользовательского web-приложения

В этом случае всегда можно определить, является ли вызов сервлета первым (атрибут с именем, равным идентификатору сессии существует) или нет. Такой подход хорошо согласуется с архитектурой JSP Model 2, которая уже рассматривалась выше.

На рис. 5.30 изображена структурная схема простого многопользовательского приложения, реализующего эту архитектуру.

На схеме изображено приложение, состоящее из сервлета, встречающего и обрабатывающего запросы и jsp-страницы, используемой для генерации ответа. Каждый запрос пользователя (на схеме изображена работа двух пользователей) сопровождается созданием пары объектов **Request** и **Response**, первоначально обрабатываемых сервлетом. При первом запросе в сессии сервлетом создается (штриховая стрелка) java-класс (обозначен овалом с надписью **Bean**), представляющий область памяти для данной сессии (область данных). Весь обмен данными между сервлетом, JSP и базой данных (для простоты не изображена на схеме) осуществляется с помощью методов этого класса.

Решение, предложенное выше, можно сделать более профессиональным, если создавать область данных не в сервлете, а в слушателе событий, реализующего интерфейс **HttpSessionListener**, или в фильтре. В последнем случае область данных можно «вмонтировать» в запрос с помощью технологии обертывания.

Предложенная методология помогает решить проблему только в рамках одного сеанса пользователя. Об этом следует помнить и предусмотреть при разработке приложения, ведь завершение сеанса влечет за собой потерю области хранения.

## 5.9. Итоги главы

1. Основными объектами, доступными для использования web-приложению, являются: **Request** (запрос), **Response** (ответ), **Context** (контекст приложения) и **Session** (сеанс связи – сессия).

2. Объект **Request** web-контейнером на основе http-запроса, поступающего от клиента. Кроме того, с помощью методов объекта **Request** разработчику становятся доступными параметры, атрибуты, заголовки запроса, а также объект **RequestDispatcher**, предназначенный для переопределения запроса.

3. Объект **Response** создается web-контейнером одновременно с объектом **Request**. Он содержит всю необходимую информацию, позволяющую web-контейнеру сгенерировать http-ответ. С помощью методов

объекта **Response** разработчик может создавать, удалять и изменять заголовки ответа, а также осуществлять переадресацию запроса.

4. Объект **Context** (контекст web-приложения) – структура, хранящая информацию о самом web-приложении, его компонентах и среде, в котором оно работает. Основная часть информации контекста описана в дескрипторе развертывания приложения. Кроме того, разработчику доступны параметры инициализации и атрибуты контекста.

5. Объект **Session** (сеанс связи, сессия) представляет в web-приложении пользователя, выполняющего последовательность запросов и получающего ответы. Время жизни объекта зависит от параметра **Session timeout**, который определяет максимальное время между запросами клиента. С каждой сессией связан уникальный идентификатор, представляющий собой строку из 32 символов и доступный разработчику. С помощью методов объекта **Session** можно создавать, удалять и изменять атрибуты сессии, менять значение **Session timeout**.

6. Для предварительной обработки запроса и ответа применяется специальный компонент web-приложения, называемый фильтром. Фильтр – это java-класс, который инициализируется web-контейнером и вызывается перед обращением к web-ресурсу, с которым этот фильтр связан. С одним ресурсом может быть связано несколько фильтров. В этом случае они образуют цепочку фильтров. Часто фильтры относят к контексту web-приложения.

7. Для обработки некоторых событий контекста или сессии применяется специальный компонент web-приложения, называемый слушателем событий. С помощью слушателей событий можно обработать создание и удаление контекста и сессии, а также все манипуляции с атрибутами этих двух объектов.

8. Построение многопользовательского web-приложения требует от разработчика дополнительных усилий, связанных с синхронизацией данных.

## Глава 6. ТЕХНОЛОГИЯ JAVA DATABASE CONNECTIVITY

### 6.1. Предисловие к главе

*Java Database Connectivity (JDBC)* – это технология, применяемая для доступа приложений на языке Java к реляционным базам данных. Пользуясь интерфейсом JDBC java-приложения, можно выполнить SQL-запрос к СУБД, а также получить и обработать результат этого запроса. В самом первом приближении JDBC – это реализация интерфейса Microsoft ODBC, но для платформы Java.

Интерфейс JDBC реализован в виде двух пакетов **java.sql** и **javax.sql**. На момент написания книги последней известной версией была JDBC 4.0. Пакеты классов и документация JDBC API 4.0 доступны для скачивания с сайта [27].

### 6.2. Архитектура JDBC

На рис. 6.1 представлена модель архитектуры интерфейса JDBC web-приложения.

Основная задача, решаемая JDBC, – это организация интерфейса между компонентом web-приложения и сервером реляционной базы данных. Условно этот интерфейс можно разбить на две части: JDBC API и JDBC-драйвер.

JDBC API – это набор java-классов и интерфейсов, входящих в состав пакетов **java.sql** и **javax.sql**, которыми оперирует разработчик приложения. Этот набор классов является стандартным и не зависит от типа sql-базы данных, с которой осуществляет взаимодействие приложение.

Все приложения, использующие JDBC одинаковым способом, выполняют sql-операции. JDBC API входит в состав спецификации Java Platform Standard Edition.

JDBC-драйвер – это программное средство, взаимодействующее с JDBC API стандартным образом и учитывающее специфику конкретной sql-базы данных. JDBC-драйвер входит в состав программного обеспечения сервера базы данных и, как правило, разрабатывается компаниями-производителями СУБД. Спецификация JDBC предусматривает четыре типа драйверов.

Первый тип jdbc-драйверов является мостом между JDBC API и драйвером ODBC. Все вызовы JDBC-драйвера первого типа транслируются в вызовы odbc-драйвера, с помощью которого и выполняется доступ к базе данных.

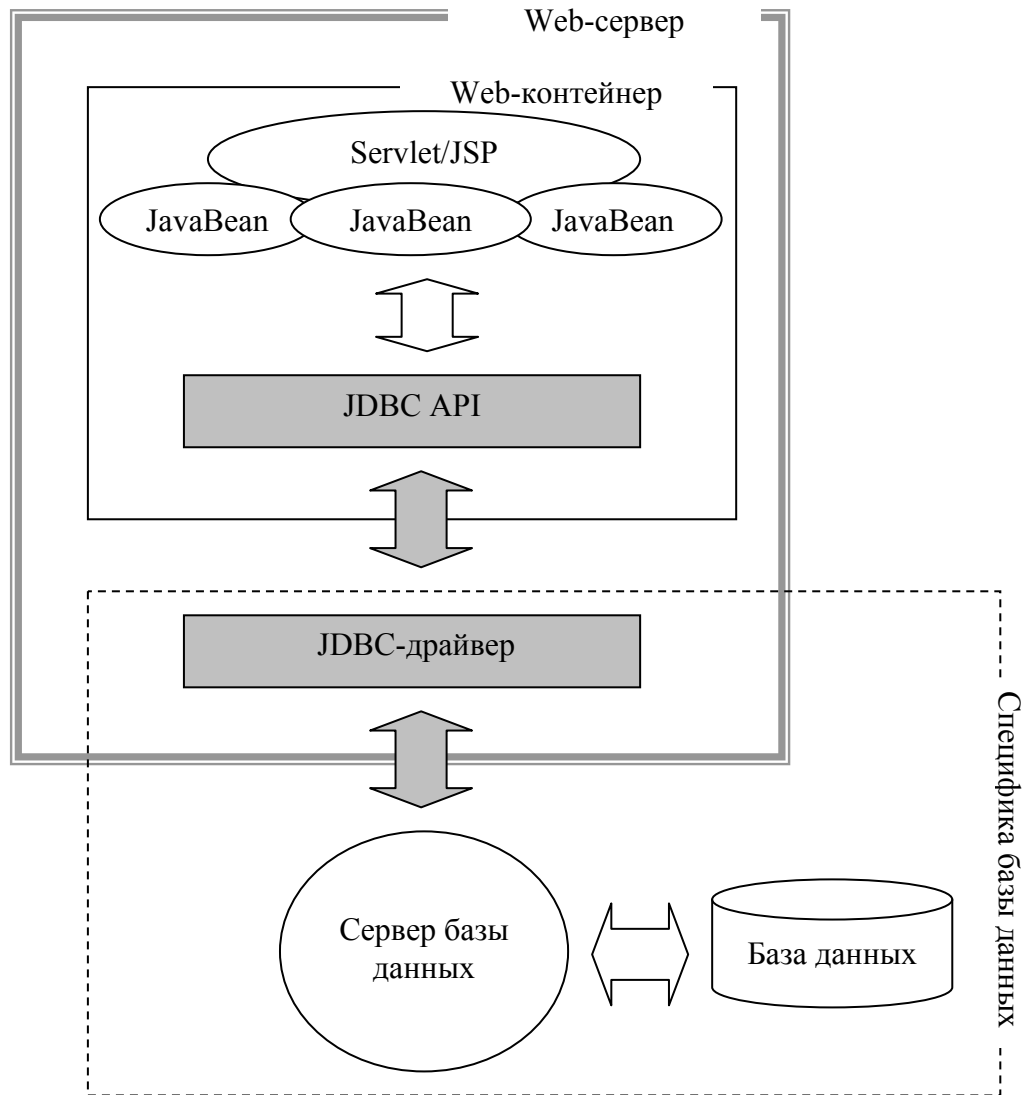


Рис. 6.1. Архитектура JDBC

Второй тип драйверов только частично реализован на языке Java и является зависимым от платформы.

Третий тип драйверов полностью реализован на языке Java, но взаимодействует с сервером через клиентскую часть СУБД.

Четвертый тип драйверов тоже полностью реализован на языке Java и не требует никакого посредника для связи с сервером базы данных (тонкий клиент).

Информация о принципах работы jdbc-драйвера содержится в спецификации JDBC [27], а с перечнем и характеристиками существующих jdbc-драйверов для всех СУБД можно ознакомиться на сайте [28]. Обычно jdbc-драйвер представляет собой jar-файл, который всегда можно скачать с официального сайта производителя СУБД.

### 6.3. Объектная модель JDBC API

JDBC API представляет собой набор взаимосвязанных java-классов, которые используются разработчиком java-приложения для доступа к базе данных. Сам sql-текст запросов к базе данных передается серверу СУБД в виде символьной строки. Для динамического формирования запроса в эту строку можно вставить специальный символ «?», называемый *маркером запроса*, который может быть динамически замещен при формировании запроса.

На рис. 6.2 представлена упрощенная объектная модель JDBC API. Прямоугольниками на рисунке изображены java-интерфейсы, а направленными линиями – методы, связывающие интерфейсы.

**Connection** является пограничным интерфейсом между JDBC API и JDBC-драйвером. Объект с интерфейсом **Connection** создается jdbc-драйвером с помощью его стандартного метода **getConnection**. После того, как этот объект был создан, дальнейшая работа осуществляется, как правило, только средствами JDBC API.

Объект с интерфейсом **Statement** создается с помощью метода **getStatement** интерфейса **Connection** и предназначен для создания статических запросов к базе данных.

Объект с интерфейсом **PreparedStatement** создается с помощью метода **prepareStatement** интерфейса **Connection** и предназначен для подготовки динамических запросов к базе данных.

Объект с интерфейсом **CallableStatement** создается с помощью метода **prepareCall** интерфейса **Connection** и предназначен для динамического вызова хранимых процедур или функций базы данных.

Методы **executeQuery**, **execute**, **getResult**, **getResultSet** позволяют инициировать передачу серверу sql-запроса. Результатом выполнения этих методов является объект с интерфейсом **ResultSet**, инкапсулирующий результат sql-запроса.

Методы объекта с интерфейсом **ResultSet** позволяют осуществить навигацию и доступ к результату выполнения sql-запроса.



В тех случаях, когда результатом вызываемой функции или хранимой процедуры являются код возврата или выходные параметры, можно обойтись без объекта **ResultSet** и получить результат напрямую через параметры запроса.

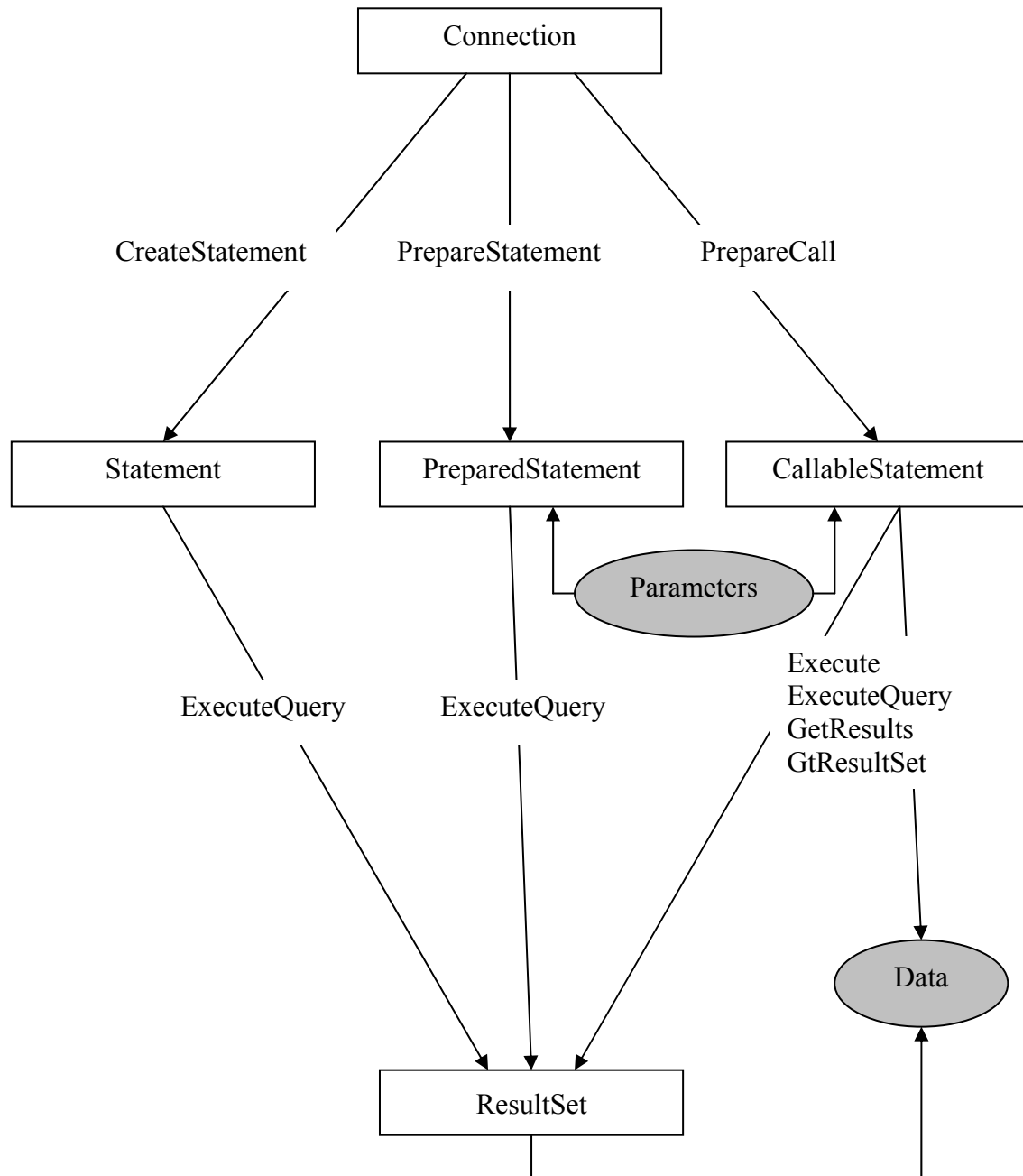


Рис. 6.2. Объектная модель пакета `java.sql`

Более подробную информацию о классах и интерфейсах JDBC API можно получить из спецификации JDBC [27].

## 6.4. Применение JDBC API

На рис. 6.3 представлен пример сервлета, который использует интерфейс JDBC четвертого типа для доступа к базе данных СУБД Oracle 10g.

```
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
import oracle.jdbc.pool.OracleDataSource;

public class Rrr extends HttpServlet implements Servlet {
    protected static String SqlPersonBrw = "select person, surname,"
        + "firstname, fathurname, sex, rowid from person "
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setCharacterEncoding("CP1251");
        PrintWriter pw = response.getWriter();
        try {
            OracleDataSource ods = new OracleDataSource();
            ods.setDriverType("thin");
            ods.setServerName("smw-oracle");
            ods.setDatabaseName("SHL");
            ods.setPortNumber(1521);
            ods.setUser("STAFFCORE");
            ods.setPassword("STAFFCORE");
            Connection conn = ods.getConnection();
            PreparedStatement stm =
            conn.prepareStatement(SqlPersonBrw);
            ResultSet rss = stm.executeQuery();
            pw.println("<table>");
            while (rss.next()) {
                pw.println("<tr>"+<td>"+rss.getString(2)+"</td>"
                    +<td>"+rss.getString(3)+"</td>"
                +<td>"+ rss.getString(5)+"</td>"
                +</tr>");
            }
            pw.println("</table>");
            conn.close();
        } catch (SQLException e) {
            System.out.println("Rrr:doGet:SQLException:" + e);
        }
    }
}
```

Рис. 6.3. Сервлет, использующий JDBC для доступа к базе данных

Процесс работы с базой данных в этом примере можно разбить на следующие этапы:

- 1) настройка JDBC-драйвера;
- 2) установка соединения с базой данных;
- 3) создание и выполнение sql-запроса;
- 4) обработка результатов запроса.

Настройка jdbc-драйвера сводится к созданию объекта класса **OracleDataSource**, входящего в состав API драйвера, а также установке параметров подключения: режима работы (в примере **thin** – тонкий клиент), имени сервера (**smw-oracle**), наименования базы данных (**SHL**), номера TCP-порта (**1521**), имени и пароля пользователя (в примере имя и пароль совпадают – **STAFFCORE**).

Установка соединения с базой данных осуществляется с помощью метода **getConnection** класса **OracleDataSource**. При успешном завершении соединения этот метод возвращает объект типа **Connection**.

Текст sql-запроса к базе данных хранится как строка с именем **SQLPersonBrw**. Запрос является статическим и представляет собой простой sql-оператор **SELECT**. Он оформляется как объект типа **PreparedStatement**, который создается с помощью метода **prepareStatement** интерфейса **Connection**. Пересылка sql-запроса на сервер базы данных выполняется методом **executeQuery** интерфейса **PreparedStatement**.

При успешном выполнении запроса (в результате выполнения метода **executeQuery**) создается объект типа **ResultSet**, инкапсулирующий результат запроса.

Результатом выполнения sql-оператора **SELECT** является таблица, инкапсулированная в объекте типа **ResultSet**. Навигация по этой таблице осуществляется с помощью метода **next** интерфейса **ResultSet**. Вызов этого метода позволяет позиционировать курсор интерфейса **ResultSet** на следующую строку таблицы. Доступ к элементам текущей строки выполняется с помощью метода **getString** интерфейса **ResultSet**.

На рис. 6.4 представлен пример еще одного сервлета, применяющего интерфейс **JDBC** для доступа к базе данных. Отличия этого примера от предыдущего заключаются в следующем:

- 1) строка запроса описывает вызов функции и содержит маркеры;
- 2) используется метод **prepareCall**;
- 3) динамически устанавливаются параметры запроса;
- 4) используется метод **execute**;
- 5) результатом запроса является значение, возвращаемое функцией.

Текст sql-запроса, как и прежде, храниться в строковой переменной, но сам запрос – это вызов функции базы данных.

В тексте вызова функции присутствуют два маркера, которые обозначаются знаком вопроса и нумеруются слева направо, начиная с единицы.

```
//.....

static String SqlPersonNumber = "{?=call personpkg.getnumber(?)}";

//.....

protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setCharacterEncoding("CP1251");
    PrintWriter pw = response.getWriter();
    String sample = request.getParameter("sample");
    pw.println("sample="+sample);
    try {
        OracleDataSource ods = new OracleDataSource();
        ods.setDriverType("thin");
        ods.setServerName("smw-oracle");
        ods.setDatabaseName("SHL");
        ods.setPortNumber(1521);
        ods.setUser("STAFFCORE");
        ods.setPassword("STAFFCORE");
        Connection conn = ods.getConnection();
        CallableStatement stm = conn.prepareCall(SqlPersonNumber);
        stm.registerOutParameter(1, Types.INTEGER);
        stm.setString(2, sample);
        stm.execute();
        Integer rc = stm.getInt(1);
        pw.println("number="+rc);
        conn.close();
    } catch (SQLException e) {
        System.out.println("Rrr:doGet:SQLException:" + e);
    }
}

//.....
```

Рис. 6.4. Фрагмент сервлета, вызывающего хранимую процедуру базы данных

Вызов хранимых процедур и функций базы данных оформляется как объект типа **CallableStatement**, который создается на основе строки запроса с помощью метода **prepareCall** объекта типа **Connection**.

С помощью **registerOutParameter** интерфейса **Callable Statement** регистрируется выходной параметр, который в строке запроса обозна-

чен первым маркером. Значение входного параметра (второй маркер) вызываемой функции устанавливается с помощью метода **setString**.

Метод **execute** интерфейса **CallableStatement** выполняет пересылку запроса серверу базы данных для исполнения.

Результатом выполнения запроса является значение типа **int**, возвращаемое функцией базы данных к точке возврата и зарегистрированное ранее как выходной параметр. Для получения этого значения применяется метод **getInt** интерфейса **CallableStatement**. В качестве параметра этому методу передается номер маркера выходного параметра.

В реальных программных системах вызов функций JDBC API редко осуществляется непосредственно из сервлета или JSP. Обычно для этого создаются специальные классы, инкапсулирующие эти вызовы.

Следует отметить, что существуют специальные программные технологии, настроенные над JDBC API и позволяющие быстро создавать web-приложения, взаимодействующие с базами данных. Наиболее известной такой технологией является Hibernate [29].

## 6.5. Использование технологии JNDI

Приведенные выше примеры применения JDBC API обладают рядом недостатков.

1. В сервлетах используется API-драйвера базы данных. При смене сервера базы данных потребуются вносить изменения в текст программы и перекомпилировать сервлет.

2. Параметры подключения к серверу (имя сервера, номер порта, логин, пароль, имя базы данных) «защиты» в тексте программы. Любое изменение этих параметров приведет к необходимости перекомпиляции сервлета.

3. Количество открытых соединений с базой данных равно количеству работающих сервлетов. Такой подход может привести к нерациональному использованию ресурсов web-сервера.

Последний из перечисленных недостатков обычно может быть устранен с помощью пула jdbc-соединений (несколько постоянно открытых соединений с базой данных, используемых последовательно по мере необходимости).

Эта функциональность может обеспечиваться производителем jdbc-драйвера или web-контейнером.

Для решения двух первых проблем технологию JDBC используют совместно с другой технологией – **Java Naming and Directory Interface (JNDI)**, которая входит в состав технологий Java Platform Enterprise Edition [12].

JNDI – технология, обеспечивающая прикладной программный интерфейс для функции наименования и каталогов. В состав этой технологии входит JNDI API, позволяющий работать со службой наименований (Naming service) или службой каталогов (Directory service).

**Служба наименований** – это специальное программное обеспечение, которое управляет системой наименований и решает две основные задачи: 1) устанавливает связь между именем и объектом; 2) осуществляет поиск объекта по имени.

**Служба каталогов** – это специальное программное обеспечение, которое управляет системой наименования объектов специального вида – каталогов. **Каталог** – это объект, обладающий набором атрибутов, описывающих поименованный объект в некоторой системе. Например, в качестве объекта может выступать web-ресурс. В этом случае атрибутами этого объекта могут быть URL ресурса, перечень параметров и т. д.

Службы наименований и каталогов обычно являются частью операционной системы. Примером службы наименований может служить Domain Name System (служба имен доменов), а примером службы каталогов – Microsoft Active Directory. Свои службы наименований или каталогов могут иметь сервера приложений, баз данных и другие программные системы.

Применение JNDI позволяет вынести описание любого объекта за пределы программного модуля. Доступ к объекту и его свойствам в этом случае осуществляется по символическому имени.

Применительно к нашему случаю объектом наименования является jdbc-соединение с базой данных. Все параметры jdbc-соединения выносятся за пределы программы в структуру службы наименования или в каталоги. В программном коде соединение будет представлено в виде символического имени. В простейшем случае внешнее (по отношению к программному коду) описание может быть помещено в xml-файл.

Более подробно с технологией JNDI можно ознакомиться на официальном сайте технологии Java Platform Enterprise Edition [12], а с примерами использования – в источнике [7].

## 6.6. Итоги главы

1. JDBC – это входящая в состав спецификаций Java Platform Standard Edition-технология доступа java-приложений к реляционным базам данных.

2. JDBC состоит из двух компонентов: классов JDBC API, входящих в состав пакетов **java.sql** и **javax.sql**, а также jar-файла, называемого драйвером JDBC. Драйвер JDBC входит в состав программного обеспечения сервера базы данных и обычно разрабатывается и поставляется разработчиком СУБД.

3. Спецификация JDBC определяет 4 типа драйверов. Наиболее распространенными являются драйверы четвертого типа, полностью реализованные на языке Java и не требующие никаких дополнительных установок, кроме JRE, на клиентском компьютере.

4. В реальных web-приложениях технология JDBC часто используется совместно с технологией JNDI. Кроме того, существуют другие технологии, построенные над JDBC API. Наиболее популярной из них является технология Hibernate.

## **Глава 7. ПРАКТИКУМ**

### **7.1. Предисловие к главе**

Для выполнения практических заданий, предложенных в этой главе, требуется компьютер с операционной системой Windows XP (локальный компьютер), включенный в сеть TCP/IP с установленным на нем следующим программным обеспечением:

- Java Developer Kit версии не ниже 1.5;
- сервер Tomcat 6.0;
- утилита Ant версии не ниже 1.7.

Кроме того, следует отдельно установить еще один сервер Tomcat на другом компьютере (удаленный компьютер), подсоединенном к той же компьютерной сети. Второй сервер потребуется для моделирования взаимодействия приложений (или отдельных компонентов приложения), расположенных на разных серверах.

Желательно (но не обязательно) иметь какое-нибудь средство IDE (Integrated Development Environment), позволяющее упростить разработку компонент приложения. Наиболее популярными java-IDE на момент написания пособия являются Eclipse, IDEA и NetBeans.

Каждое практическое работа состоит из нескольких заданий, выполнение которых основывается на сведениях и примерах, изложенных в пособии. Задания всего практикума, как правило, связаны между собой: результаты некоторых заданий могут быть использованы в следующих заданиях. Поэтому последовательность их выполнения является важной.

### **7.2. Практическая работа № 1 Разработка простейшего web-приложения**

#### **7.2.1. Цель и результаты работы**

Целью практической работы является первоначальное знакомство с технологией разработки web-приложения.

Результатом практической работы является подготовленная рабочая директория для хранения проекта web-приложения и разработанное простейшее web-приложение.

#### **7.2.2. Теоретические сведения**

Теоретические сведения, необходимые для выполнения практической работы, изложены в разд. 2 пособия.



### **7.2.3. Подготовка рабочей директории**

**Задание 1.** Создайте рабочую директорию для хранения проекта. Предусмотрите возможность хранения страниц HTML, JSP, текстов JavaScript, сервлетов (исходных текстов и откомпилированных классов), файлов сборки, а также графических файлов (директория **image**).

### **7.2.4. Разработка компонент web-приложения**

**Задание 2.** Разработайте html-страницу с именем **index.html**. Страница должна обеспечивать вывод в окно web-браузера фамилию и инициалы обучающегося. Поместите эту страницу в рабочую директорию проекта.

**Задание 3.** Разработайте дескриптор развертывания приложения. Обеспечьте автоматический вызов страницы **index.html** при запуске приложения. Поместите разработанный дескриптор в рабочую директорию.

**Задание 4.** Разработайте файл сборки приложения, осуществляющий установку приложения на локальный web-сервер. Имя web-приложения должно иметь следующий вид: **AS\_xxxxx**, где **xxxxx** – уникальный номер, выданный преподавателем.

### **7.2.5. Установка и проверка web-приложения на локальном сервере**

**Задание 5.** Запустите утилиту Ant для исполнения файла сборки, разработанного в предыдущем задании. Визуально проверьте правильность создания директорий web-приложения в структуре директорий сервера.

**Задание 6.** Запустите локальный сервер. Убедитесь в работоспособности сервера, вызвав его стартовую страницу (страницу сервера Tomcat) с помощью браузера.

**Задание 7.** Вызовите приложение **AS\_xxxxx** и убедитесь в его работоспособности.

### **7.2.6. Установка и проверка web-приложения на удаленном сервере**

**Задание 8.** Измените файл сборки таким образом, чтобы сборка приложения **AS\_xxxxx** выполнялась на удаленный сервер. Запустите утилиту Ant для исполнения файла сборки. Визуально убедитесь в правильности создания приложения на удаленном сервере.

**Задание 9.** Перезапустите удаленный сервер и вызовите с помощью локального браузера приложение **AS\_xxxxx** с удаленного сервера. Убедитесь в работоспособности приложения.

### 7.2.7. Исследование возможностей простейшего web-приложения

**Задание 10.** Разработайте еще две html-страницы **address.html** и **education.html**, которые содержат информацию об адресе проживания и образовании. Внесите изменения в **index.html** для вызова новых страниц с помощью гиперссылки. Проверьте работоспособность приложения на локальном и удаленном серверах. Исследуйте возможность вызова новых страниц напрямую без вызова **index.html**.

**Задание 11.** Исследуйте возможность вызова страницы **education.html** с удаленного сервера, работающего на локальном сервере приложением.

**Задание 12.** Внесите такие изменения в дескриптор развертывания приложения, чтобы стартовой стала страница **address.html**.

**Задание 13.** Просмотрите файл **web.xml** локального сервера, задающего значение параметров дескриптора развертывания приложения по умолчанию. Предложите и реализуйте пример web-приложения, демонстрирующий установку значений параметров дескриптора приложения по умолчанию.

**Задание 14.** Верните состояние web-приложения на момент выполнения задания 10. Разместите на странице **index.html** фотографию. Для хранения графического файла с фотографией создайте в рабочей директории проекта поддиректорию **images**. Проверьте работоспособность приложения на локальном и удаленном серверах.

## 7.3. Практическая работа № 2 Разработка простейшего сервлета

### 7.3.1. Цель и результаты работы

Целью практической работы является знакомство с технологией сервлетов.

Результатом практической работы является разработанный сервлет, включенный в состав простого web-приложения.

### 7.3.2. Теоретические сведения

Теоретические сведения, необходимые для выполнения практической работы, изложены в разделах 3.1–3.3.3 и 4.2.4 пособия.

### 7.3.3. Разработка сервлета

**Задание 1.** Разработайте класс сервлета с именем **Sss**. Реализуйте конструктор класса и методы **init**, **destroy**, **service**. В каждом реализо-

ванном методе класса обеспечьте вывод сообщения в стандартный поток вывода. Поместите разработанный класс в рабочую директорию проекта. Внесите изменения в дескриптор приложения и файл **hosts** локального компьютера, которые позволили бы вызывать сервлет помощью URL **http://LocalServer\_xxxxx:8080/AS\_XXXXX/GoSss** с локального сервера **http://RemoteServer\_XXXXX:8080/AS\_XXXXX/GoSss** с удаленного сервера, где **XXXXX** имеет то же значение, что и в предшествующей работе. Внесите изменения в файл сборки приложения, которые бы обеспечивали компиляцию класса сервлета и перемещение результатов компиляции в структуру директорий локального и удаленного серверов.

**Задание 2.** Выполните утилиту **Ant** и визуально убедитесь в правильности размещения классов сервлета и дескриптора развертывания приложения в директориях серверного приложения.

**Задание 3.** Запустите локальный сервер **Tomcat** и вызовите с помощью браузера сервлет. Убедитесь, что сообщения, выведенные сервлетом, помещены в журнал локального сервера.

#### 7.3.4. Исследование работы сервлета

**Задание 4.** Исследуйте порядок вызова методов класса сервлета: 1) при первом вызове сервлета; 2) при повторном вызове сервлета; 3) при останове и перезапуске сервера.

**Задание 5.** Повторите все исследования, проведенные в задании 4, для сервлета на удаленном сервере.

**Задание 6.** Внесите изменения в сервлет **Sss**, обеспечивающие дублирование выводимых сервлетом сообщений в окно web-браузера.

**Задание 7.** Определите тип вызова сервлета **Sss** и выведите соответствующее сообщение в окно браузера.

**Задание 8.** Определите имя сервера и ip-адрес хоста, вызывающего сервлет **Sss**. Выведите соответствующие сообщения в окно браузера.

**Задание 9.** Повторите все исследования, проведенные в заданиях 7 и 8, для сервлета на удаленном сервере.

**Задание 10.** Внесите изменения в сервлет **Sss**, позволяющие обрабатывать GET-запрос с помощью метода **doGet**. Убедитесь, что приложение корректно работает на локальном и удаленном серверах.

**Задание 11.** Внесите изменения в содержимое страницы **index.html**, которые позволили вызвать сервлет **Sss** с этой страницы с помощью запроса **GET**. Убедитесь в работоспособности приложения на локальном и удаленном серверах.

**Задание 12.** Внесите изменения в содержимое страницы **index.html**, которые позволили вызвать сервлет **Sss** с этой страницы

с помощью запроса **POST**. Убедитесь в работоспособности приложения на локальном и удаленном серверах.

#### **7.3.5. Обработка параметров, передаваемых сервлету в запросе**

**Задание 13.** Обеспечьте передачу параметров с именами **firstname** и **lastname** сервлету **Sss** при GET-вызове со страницы **index.html**. Выведите значение параметров в окно браузера. Убедитесь в работоспособности приложения на локальном и удаленном серверах.

**Задание 14.** Получите и выведите в окно web-браузера всю строку GET-запроса. Убедитесь в работоспособности приложения на локальном и удаленном серверах.

**Задание 15.** Обеспечьте передачу параметров с именами **firstname** и **lastname** сервлету **Sss** при POST-вызове со страницы **index.html**. Выведите значение параметров в окно браузера. Убедитесь в работоспособности приложения на локальном и удаленном серверах.

### **7.4. Практическая работа № 3**

#### **Переопределение и переадресация запросов, формирование и обработка GET- и POST-запросов в сервлете**

##### **7.4.1. Цель и результаты работы**

Целью практической работы является знакомство с технологией сервлетов.

Результатом практической работы является несколько разработанных сервлетов, взаимодействующих в рамках одного и/или нескольких web-приложений.

##### **7.4.2. Теоретические сведения**

Теоретические сведения, необходимые для выполнения практической работы, изложены в разд. 3 и гл. 5.4 пособия.

##### **7.4.3. Переопределение get-запроса сервлета**

**Задание 1.** Разработайте два сервлета, **Sss** и **Ggg**, обрабатывающие запрос типа **GET**. Обеспечьте GET-вызов сервлета **Sss** со стартовой страницы с помощью гиперссылки. Переопределите запрос сервлета **Sss** на сервлет **Ggg**. Осуществите трассировку работы сервлетов с помощью вывода в стандартный поток. Исследуйте возможность передачи параметров при переопределении запроса.

**Задание 2.** Разработайте на стартовой странице форму, вызывающую сервлет **Sss** с помощью POST-запроса, и обеспечьте обработку запроса в методе **service** сервлета **Sss**. Исследуйте возможность переопределения запроса на сервлет **Ggg**. Объясните полученный результат.

**Задание 3.** Разработайте html-страницу и исследуйте возможность переопределения GET-запроса на html-страницу из сервлета.

**Задание 4.** Исследуйте возможность двойного переопределения GET-запроса с сервлета **Sss** на **Ggg** и с **Ggg** на html-страницу.

**Задание 5.** Проверьте возможность вывода данных в окно браузера из сервлета, осуществляющего переопределение запроса, и из сервлета, на который выполнено переопределение. Объясните полученный результат.

#### **7.4.4. Переадресация запроса**

**Задание 6.** Выполните задания 1–5, но применяйте не переопределение, а переадресацию запроса. Объясните, в каких случаях необходимо использовать переопределение, а в каких переадресацию запроса.

#### **7.4.5. Формирование и обработка get-запроса в сервлете**

**Задание 7.** Сформируйте get-запрос в сервлете **Sss** к сервлету **Ggg** с помощью классов пакета **org.apache.commons.httpclient**. Организуйте передачу параметров запроса. В сервлете **Ggg** сгенерируйте http-ответ, содержащий значения параметров запроса. Обработайте ответ в сервлете **Sss** и выведите полученный ответ от сервлета **Ggg** в окно браузера.

**Задание 8.** Разместите сервлет **Sss** на локальном web-сервере, а **Ggg** – на удаленном. Измените значение URL для вызова **Ggg** с удаленного сервера в сервлете **Sss**. Проверьте работоспособность приложения.

#### **7.4.6. Формирование и обработка post-запроса в сервлете**

**Задание 9.** Выполните задание 6, но в сервлете **Sss** сформируйте, а в **Ggg** обработайте запрос типа **POST**.

**Задание 10.** Выполните задание 7, но для сервлетов **Sss** и **Ggg**, разработанных в задании 8.

### **7.5. Практическая работа № 4** **Разработка простейшей jsp-страницы**

#### **7.5.1. Цель и результаты работы**

Целью практической работы является знакомство с технологией Java Server Pages.

Результатом практической работы являются разработанные jsp-страницы.

### 7.5.2. Теоретические сведения

Теоретические сведения, необходимые для выполнения практической работы, изложены в разд. 4 пособия.

### 7.5.3. Разработка jsp-страницы

**Задание 1.** Разработайте jsp-страницу **index.jsp**, которая бы динамически выводила в окно web-браузера одно из приветствий Good night, Good morning, Good afternoon или Good evening в зависимости от времени суток. Добавьте страницу в рабочую директорию, внесите необходимые изменения в файл сборки приложения и выполните утилиту **Ant**. Вызовите страницу с помощью web-браузера и убедитесь в ее работоспособности.

**Задание 2.** Доработайте страницу **index.jsp** таким образом, чтобы после приветствия в окно браузера динамически выводилась таблица, состоящая из двух столбцов и семи строк. В первом столбце должны находиться даты в формате **dd.mm.yyyy**, начиная (в первой строке) с текущей, а во втором – соответствующий дате номер дня недели (1 – понедельник, 2 – вторник и т. д.)

**Задание 3.** Сделайте страницу **index.jsp** стартовой для web-приложения. Убедитесь, что она вызывается автоматически при запуске web-приложения.

**Задание 4.** Убедитесь в работоспособности приложений, разработанных в заданиях 1–3, на удаленном сервере.

### 7.5.4. Взаимодействие jsp-страниц

**Задание 5.** Разработайте четыре jsp-страницы: **night.jsp**, **morning.jsp**, **afternoon.jsp** и **evening.jsp**. Каждая jsp-страница должна выводить собственное имя в окно браузера и в файл **logs\stdout** web-сервера. На странице **index.jsp** создайте кнопку **press**. Нажатие ее должно приводить к вызову одной из jsp-страниц в зависимости от времени суток с помощью директивы **include**. Выполните все необходимые операции для сборки нового приложения. Убедитесь в работоспособности приложения. Исследуйте html-текст в окне браузера.

**Задание 6.** Замените директивы **include** в странице **index.jsp** на стандартные действия **jsp:include** и убедитесь в работоспособности приложения. Исследуйте html-текст в окне браузера.

**Задание 7.** Разработайте сервлет **afternoon**, который формирует вывод сообщения **Servlet:Good afternoon** в http-поток. Замените в **index.jsp** вызов страницы **afternoon.jsp** на вызов сервлета **afternoon**. Добейтесь вызова сервлета **afternoon** со страницы **index.jsp** и убедитесь в работоспособности приложения. Исследуйте html-текст в окне браузера.

**Задание 8.** Замените вызовы **jsp:include** в странице **index.jsp** на вызовы **jsp:forward**. Убедитесь в работоспособности приложения. Исследуйте html-текст в окне браузера.

**Задание 9.** Убедитесь в работоспособности приложений, разработанных в заданиях 5–8, на удаленном сервере.

**Задание 10.** Объясните отличия в применении директивы **include** и стандартных действий **jsp:include** и **jsp:forward**.

#### **7.5.5. Взаимодействие сервлета и jsp-страницы**

**Задание 11.** Разработайте сервлет **Jjj**, который переопределяет запрос на одну из 4 страниц, описанных в задании 4, в зависимости от времени суток. Убедитесь в работоспособности приложения.

**Задание 12.** Измените сервлет **Jjj** таким образом, чтобы он вызывал (по тому же принципу, что и раньше) одну из jsp-страниц с помощью GET-запроса, обработал ответ и вывел его в окно браузера. Убедитесь в работоспособности приложения.

**Задание 13.** Измените в сервлете **Jjj** GET-вызовы jsp-страниц на POST-вызовы и убедитесь в работоспособности приложения.

**Задание 14.** Убедитесь в работоспособности приложений, разработанных в заданиях 11–13, на удаленном сервере.

### **7.6. Практическая работа № 5** **Разработка библиотеки тегов**

#### **7.6.1. Цель и результаты работы**

Целью практической работы является знакомство с технологией разработки библиотеки тегов Java Server Pages.

Результатом практической работы является разработанная библиотека тегов.

#### **7.6.2. Теоретические сведения**

Теоретические сведения, необходимые для выполнения практической работы, изложены в гл. 4.3 пособия.

### 7.6.3. Разработка дескриптора библиотеки тегов

**Задание 1.** Разработайте дескриптор библиотеки тегов, содержащий описание следующих тегов:

**dossier** – тег, предполагающий использование тела и предназначенный для создания html-тега **<form>**, генерирующего http-запрос типа POST;

**surname** – тег, предназначенный для ввода фамилии (html-тег **<input type=text>**);

**lastname** – тег, предназначенный для ввода имени (html-тег **<input type=text>**);

**sex** – тег, предназначенный для выбора одного из значений **male** или **female** (html-тег **<input type=radio>**);

**submit** – тег, предназначенный для отображения двух кнопок – **OK** и **Cancel** (html-тег **<input type=submit>**).

**Задание 2.** Поместите разработанный в задании 1 дескриптор тегов в рабочую директорию и внесите необходимые изменения в файл сборки приложения.

### 7.6.4. Разработка обработчиков тегов

**Задание 3.** Разработайте для каждого тега, описанного в дескрипторе библиотеки тегов, свой обработчик.

**Задание 4.** Поместите разработанные в задании 3 обработчики тегов в рабочую директорию и внесите необходимые изменения в файл сборки приложения.

### 7.6.5. Разработка jsp-страницы с применением библиотечных тегов

**Задание 5.** Разработайте jsp-страницу **Ttt.jsp** с использованием тегов, разработанных в заданиях 1, 3. Страница должна позволять ввести значения **surname**, **lastname** и **sex**. На форме должны отображаться две кнопки – **OK** и **Cancel**. Нажатие кнопки **OK** должно приводить к вызову сервлета **Ttt**, обрабатывающего сгенерированный запрос POST и выводящего в окно браузера переданные параметры (значения полей ввода).

**Задание 6.** Разработайте сервлет с именем **Ttt** и выполняющего функции, описанные в задании 5.

### 7.6.6. Использование библиотеки тегов

**Задание 7.** Обеспечьте вызов jsp-страницы **Ttt.jsp** со стартовой страницы с помощью гиперссылки. Убедитесь в работоспособности приложения.



**Задание 8.** Исследуйте возможность размещения страницы **Ttt.jsp** и библиотеки тегов на разных web-серверах.

## **7.7. Практическая работа № 6**

### **Применение параметров инициализации и атрибутов контекста**

#### **7.7.1. Цель и результаты работы**

Целью практической работы является исследование возможностей среды выполнения web-приложения.

Результатом практической работы являются разработанные сервлеты и jsp-страницы, использующие параметры инициализации и атрибуты контекста web-приложения.

#### **7.7.2. Теоретические сведения**

Теоретические сведения, необходимые для выполнения практической работы, изложены в гл. 5.2 пособия.

#### **7.7.3. Использование параметров инициализации контекста web-приложения**

**Задание 1.** Добавьте в дескриптор развертывания приложения два параметра с именами **URL1** и **URL2**. Значение параметра **URL1** – URL html-страницы на удаленном сервере. Значение параметра **URL2** – URL сервлета на удаленном сервере.

**Задание 2.** Разработайте сервлет, обрабатывающий запрос типа GET. Сервлет должен обрабатывать параметр с именем **urln**. Если значение параметра равно **n**, то сервлет должен считывать параметр инициализации с именем **URLn**. Если параметр инициализации существует, то сервлет должен сгенерировать запрос типа GET к ресурсу с URL равным значению соответствующего параметра инициализации и вывести в окно браузера ответ, иначе должен выдать сообщение **parameter URLn not found**.

**Задание 3.** Разработайте страницу JSP, выводящую в окно браузера значения параметров **URL1** и **URL2**.

#### **7.7.4. Применение атрибутов контекста web-приложения**

**Задание 4.** Разработайте класс с именем **CBean**, предназначенный для установки и хранения значений переменных **Value1**, **Value2** и **Value3**. Класс имеет три метода с именами **SetValue1**, **SetValue2** и **SetValue3** для установки значений переменных и три метода – **GetValue1**, **GetValue2** и **GetValue3** – для считывания значений.

**Задание 5.** Разработайте сервлет с именем **Ccc**, обрабатывающий запросы типа GET и POST. Сервлет должен обрабатывать параметры с именами **CBean**, **Value1**, **Value2** и **Value3**.

Параметр **CBean** может принимать два значения – **old** и **new**. Значение **new** является указанием сервлету создать новый объект класса **CBean** и записать ссылку на него в атрибут контекста с именем **atrCBean**. Значение **old** устанавливается по умолчанию и предполагает использование старого объекта.

Если параметры **Value1**, **Value2** и **Value3** указаны при вызове сервлета, то является указанием сервлету на установку нового значения соответствующей переменной класса **CBean**. Если параметры отсутствуют, то значение переменных не изменяются.

Сервлет **Ccc** создает объект класса **CBean** при инициализации, а ссылку на объект записывает в атрибут контекста с именем **atrCBean**. Первоначальные значения переменных класса **CBean** должны быть **null**. Сервлет **Ccc** переопределяет запрос на jsp-страницу с именем **Ccc.jsp**.

**Задание 6.** Разработайте jsp-страницу с именем **Ccc.jsp**. В скрипте jsp-страницы должна быть извлечена ссылка на объект типа **CBean** и с помощью get-методов объекта выведены в окно браузера значения переменных **Value1**, **Value2** и **Value3**.

#### 7.7.5. Исследование свойств атрибутов контекста web-приложения

**Задание 7.** Убедитесь в работоспособности сервлета **Ccc** и страницы **Ccc.jsp**. Для этого выполните установку значений переменных, создание нового класса и установку значений переменных, изменение отдельных переменных.

**Задание 8.** Исследуйте сохранность значения атрибутов контекста после перезапуска приложения. Для этого следует запустить сервлет два раза.

Первый раз с созданием нового класса и установкой переменных, а второй раз без создания класса и установки переменных.

**Задание 9.** Исследуйте сохранность значения атрибутов контекста после перезапуска сервера. Для этого следует запустить сервлет два раза. Первый раз с созданием нового класса и установкой переменных, а второй раз после перезагрузки web-сервера и без создания класса и без установки переменных.

**Задание 10.** Исследуйте сохранность значения атрибутов при работе нескольких сессий. Для этого сервлет **Ccc** и страницу **Ccc.jsp**

следует установить на удаленном сервере. Далее с двух различных компьютеров в сети необходимо вызвать сервлет. В первом случае с созданием объекта и установкой значений переменных, во втором – без создания объекта и установки значений переменных.

## **7.8. Практическая работа № 7**

### **Применение атрибутов запроса и сессии**

#### **7.8.1. Цель и результаты работы**

Целью практической работы является исследование возможностей среды выполнения web-приложения.

Результатом практической работы являются разработанные сервлеты и jsp-страницы, использующие атрибуты сессии web-приложения.

#### **7.8.2. Теоретические сведения**

Теоретические сведения, необходимые для выполнения практической работы, изложены в гл. 5.5 пособия.

#### **7.8.3. Использование атрибутов запроса**

**Задание 1.** Измените сервлет **Ссс** и страницу **Ссс.jsp** следующим образом: для хранения ссылки на класс используйте атрибут запроса.

**Задание 2.** Проведите исследования аналогичные тем, которые выполнялись в заданиях 7–10 практической работы 6.

#### **7.8.4. Использование атрибутов сессии**

**Задание 3.** Измените сервлет **Ссс** и страницу **Ссс.jsp** следующим образом: для хранения ссылки на класс используйте атрибут сессии, а в качестве имени атрибута – идентификатор сессии.

**Задание 4.** Проведите исследования, аналогичные тем, которые выполнялись в заданиях 7–10 практической работы 6.

## **7.9. Практическая работа № 8**

### **Использование заголовков запроса и ответа**

#### **7.9.1. Цель и результаты работы**

Целью практической работы является исследование возможностей среды выполнения web-приложения.

Результатом практической работы являются разработанные сервлеты и jsp-страницы, использующие заголовки запросов и ответов.

### **7.9.2. Теоретические сведения**

Теоретические сведения, необходимые для выполнения практической работы, изложены в 5.3.2 и 5.4.1 пособия.

### **7.9.3. Использование заголовков запроса**

**Задание 1.** Разработайте два сервлета – **Aaa** и **Bbb**. Сервлет **Aaa** расположить на локальном сервере, **Bbb** – на удаленном. Сервлет **Aaa** выполняет http-запрос типа POST к сервлету **Bbb** с передачей трех параметров и трех дополнительных заголовков, а также обработать и вывести в окно браузера полученный ответ. Имена параметров и заголовков, а также их значения должны совпадать. Сервлет **Bbb** должен сформировать http-ответ, в котором содержаться значения входных параметров и заголовков запроса.

### **7.9.4. Использование заголовков ответа**

**Задание 2.** Внесите изменения в сервлет **Aaa**: кроме полученного от сервлета **Bbb** ответа он должен выводить в окно браузера значения всех заголовков ответа. Внесите изменения в сервлет **Bbb**: сформируйте два заголовка ответа перед формированием текста http-ответа.

## **7.10. Практическая работа № 9**

### **Использование фильтров и слушателей событий**

#### **7.10.1. Цель и результаты работы**

Целью практической работы является исследование возможностей среды выполнения web-приложения.

Результатом практической работы являются разработанные фильтры и слушатели событий.

#### **7.10.2. Теоретические сведения**

Теоретические сведения, необходимые для выполнения практической работы, изложены в гл. 5.6 и 5.7 пособия.

#### **7.10.3. Разработка фильтров**

**Задание 1.** Разработайте фильтр с именем **F1** для сервлета **Css**, созданного в практической работе 7. С помощью трассировки убедит-

тес, что фильтр инициализируется и выполняется. Объясните, в какой момент происходит инициализация фильтра. Проверьте возможность блокировки доступа к сервлету **Ссс** с помощью фильтра.

**Задание 2.** Разработайте фильтр с именем **F2** для страницы **Ссс.jsp**, созданной в практической работе 7. Проверьте возможность блокировки доступа к странице **Ссс.jsp** с помощью фильтра.

**Задание 3.** Разработайте цепочку фильтров **F1**, **F2**, **F3** для сервлета **Ссс**. Проверьте возможность блокировки доступа к сервлету различными фильтрами в цепочке.

#### **7.10.4. Разработка слушателей событий**

**Задание 4.** Разработайте слушатель событий с именем **L1**, реагирующий на создание и удаление контекста web-приложения. Убедитесь в работоспособности слушателя с помощью трассировки.

**Задание 5.** Разработайте слушатель событий с именем **L2**, реагирующий на создание, модификацию и удаление атрибутов сессии. Убедитесь в работоспособности слушателя с помощью трассировки.

### **7.11. Практическая работа № 10 Применение JDBC**

#### **7.11.1. Цель и результаты работы**

Целью практической работы является исследование возможностей технологии доступа к базе данных JDBC.

Результатом практической работы являются разработанные сервлеты, осуществляющие sql-запросы к базе данных с помощью JDBC.

#### **7.11.2. Теоретические сведения**

Теоретические сведения, необходимые для выполнения практической работы, изложены в главе 6 пособия.

#### **7.11.3. Выполнение статического запроса к базе данных**

**Задание 1.** Установите выданный преподавателем jdbc-драйвер на локальный и удаленный серверы, а также выясните все параметры подключения к базе данных, ее структуру, прототипы хранимых процедур и функций.

**Задание 2.** Разработайте сервлет, выполняющий статический запрос SELECT запрос к базе данных. Результаты запроса следует отразить в окне браузера.

Установите выданный преподавателем jdbc-драйвер на локальный и удаленный сервера, а также выясните все параметры подключения к базе данных и ее структуру.

#### **7.11.4. Выполнение динамического запроса к базе данных**

**Задание 3.** Измените сервлет, разработанный в задании 2, таким образом, чтобы условие **WHERE** оператора **SELECT** устанавливалось динамически с помощью маркеров.

#### **7.11.5. Выполнение хранимых процедур**

**Задание 4.** Разработайте сервлет, выполняющий вызов хранимой процедуры базы данных. При этом в сервлете должны динамически устанавливаться входные параметры процедуры и обрабатываться выходные параметры. Значения выходных параметров следует вывести в окно браузера.

### **7.12. Итоги главы**

1. Практическая разработка web-приложения на языке Java основывается на двух основных технологиях: сервлетов и JSP .
2. Процесс установки и настройки web-приложения зависит от возможностей web-сервера.
3. Для взаимодействия web-приложение с реляционной базой данных применяется технология JDBC. Драйвер JDBC следует искать для скачивания на сайтах производителей СУБД.
4. Профессиональная разработка web-приложений, как правило, требует привлечения дополнительных библиотек, большинство из которых свободно можно скачать с сайта производителя.
5. Программирование многопользовательских web-приложений требует от разработчика дополнительных усилий для разграничения совместной работы клиентов с приложением.

## ЛИТЕРАТУРА

1. Ноутон, П. Java™ 2 / П. Ноутон, Г. Шилдт. – СПб.: БХВ-Петербург, 2006. – 1072 с.
2. HTML, XHTML и CSS: библия пользователя / Б. Пфаффенбергер [и др.]. – М.: Вильямс, 2007. – 752 с.
3. Хабибуллин, И. Ш. Самоучитель XML / И. Ш. Хабибуллин. – СПб.: БХВ-Петербург, 2003. – 336 с.
4. Клайн, К. SQL: справочник / К. Клайн. – М.: КУДИЦ-ОБРАЗ, 2006. – 832 с.
5. Apache Tomcat для профессионалов / Бакор А. [и др.]. – М.: КУДИЦ-ОБРАЗ, 2005. – 544 с.
6. Пауэл, Т. Полный справочник по JavaScript / Т. Пауэл, Ф. Шнайдер. – М.: Вильямс, 2006. – 960 с.
7. Перри, Б. У. Java-сервлеты и JSP: сборник рецептов / Б. У. Перри. – М.: КУДИЦ-ПРЕСС, 2006. – 768 с.
8. <http://java.sun.com> – официальный сайт Sun Microsystems Inc.
9. <http://jcp.org> – официальный сайт организации Java Community Process.
10. <http://java.sun.com/javase> – официальная страница Java Platform Standard Edition.
11. <http://java.sun.com/javame> – официальная страница Java Platform Micro Edition.
12. <http://java.sun.com/javaee> – официальная страница Java Platform Enterprise Edition.
13. <http://www.sun.com/software/products/appsrvr> – официальная страница Sun GlassFish Enterprise Server.
14. <http://www-142.ibm.com/software/dre/ecatalog> – каталог программных продуктов IBM.
15. <http://www.oracle.com/global/ru/ip/10g/as> – официальная страница Oracle Application Server 10g.
16. <http://www.jboss.org/> – официальный сайт компании JBOSS.
17. <http://www.bea.com/> – официальный сайт компании BEA.
18. <http://tomcat.apache.org/> – официальный сайт Apache Tomcat.
19. <http://www.apache.org/> – официальный сайт ASF.
20. <http://jakarta.apache.org/> – официальный сайт проекта Jakarta.
21. <http://www.w3.org/> – официальный сайт консорциума W3C.

- 22. <http://ant.apache.org/> – официальный сайт Apache Ant.
- 23. <http://maven.apache.org/> – официальный сайт Apache Maven.
- 24. <http://www.ietf.org> – официальный сайт Internet Engineering Task Force.
- 25. Маклафлин, Б. Изучаем Ajax / Б. Маклафлин. – СПб.: Питер, 2008. – 443 с.
- 26. <http://commons.apache.org/downloads/> – сайт ASF, предназначенный для скачивания java-библиотек общего назначения.
- 27. <http://developers.sun.com/product/jdbc/download.html> – специфика-  
ция и документация JDBC API.
- 28. <http://developers.sun.com/product/jdbc/drivers> – перечень JDBC-  
драйверов.
- 29. <http://www.hibernate.org> – официальный сайт проекта Hibernate.



## ОГЛАВЛЕНИЕ

<b>ПРЕДИСЛОВИЕ .....</b>	<b>3</b>
<b>Глава 1. ВВЕДЕНИЕ.....</b>	<b>6</b>
<b>Глава 2. АРХИТЕКТУРА WEB-ПРИЛОЖЕНИЯ .....</b>	<b>10</b>
2.1. Предисловие к главе.....	10
2.2. Web-сервер Apache Tomcat.....	12
2.3. Структура директорий Tomcat .....	14
2.4. Дескриптор развертывания web-приложения .....	17
2.5. Утилита Ant.....	19
2.6. Разработка простейшего web-приложения .....	23
2.7. Итоги главы.....	25
<b>Глава 3. РАЗРАБОТКА СЕРВЛЕТА.....</b>	<b>27</b>
3.1. Предисловие к главе .....	27
3.2. Структура и интерфейсы сервлета.....	28
3.3. Обработка запросов и ответов HTTP .....	33
3.3.1. Обработка http-запросов типа GET .....	34
3.3.2. Обработка http-запросов типа POST .....	40
3.3.3. Обработка http-запросов типа GET и POST в одном сервлете .....	46
3.3.4. Переопределение http-запроса типа GET .....	46
3.3.5. Формирование http-запросов .....	48
3.4. Параметры инициализации сервлета .....	54
3.5. Итоги главы .....	56
<b>Глава 4. ТЕХНОЛОГИЯ JAVA SERVER PAGES (JSP).....</b>	<b>57</b>
4.1. Предисловие к главе .....	57
4.2. Структура страницы JSP .....	59
4.2.1. Директивы JSP .....	59
4.2.2. Объявления JSP.....	64
4.2.3. Выражения JSP.....	64
4.2.4. Скриплеты JSP .....	66
4.2.5. Неявные объекты JS .....	68
4.3. Библиотека тегов.....	68
4.3.1. Дескриптор библиотеки тегов.....	69
4.3.2. Обработчик тега.....	70
4.3.3. Применение библиотечных тегов .....	73
4.4. Стандартные действия JSP.....	75
4.5. Стандартная библиотека тегов JSTL .....	78

4.6. Обработка параметров запроса в JSP .....	79
4.7. Обработка ошибок в JSP .....	79
4.8. Итоги главы .....	79
<b>Глава 5. СРЕДА ВЫПОЛНЕНИЯ WEB-ПРИЛОЖЕНИЯ .....</b>	<b>81</b>
5.1. Предисловие к главе .....	81
5.2. Контекст web-приложения .....	82
5.2.1. Параметры инициализации контекста .....	83
5.2.2. Атрибуты контекста .....	86
5.2.3. Обработка исключений .....	87
5.3. Запрос .....	88
5.3.1. Параметры запроса .....	88
5.3.2. Атрибуты запроса .....	88
5.3.3. Заголовки запроса .....	91
5.3.4. Объект RequestDispatcher .....	93
5.4. Ответ .....	93
5.4.1. Заголовки ответа .....	93
5.4.2. Переадресация .....	95
5.5. Сеанс связи (сессия) .....	96
5.5.1. Идентификатор сессии .....	97
5.5.2. Атрибуты сессии .....	97
5.5.3. Файлы cookies... ..	98
5.6. Фильтры .....	99
5.7. Слушатели событий .....	103
5.8. Методология построения многопользовательского web-приложения .....	106
5.9. Итоги главы .....	107
<b>Глава 6. ТЕХНОЛОГИЯ JAVA DATABASE CONNECTIVITY (JDBC) .....</b>	<b>110</b>
6.1. Предисловие к главе .....	110
6.2. Архитектура JDBC .....	110
6.3. Объектная модель JDBC API .....	112
6.4. Применение JDBC API .....	114
6.5. Использование технологии JNDI .....	117
6.6. Итоги главы .....	119
<b>Глава 7. ПРАКТИКУМ .....</b>	<b>120</b>
7.1. Предисловие к главе .....	120
7.2. Практическая работа № 1. Разработка простейшего web-приложения .....	120

7.3. Практическая работа № 2.	
Разработка простейшего сервлета.....	122
7.4. Практическая работа № 3.	
Переопределение и переадресация запросов, формирование и обработка GET и POST запросов.....	124
7.5. Практическая работа № 4.	
Разработка простейшей jsp-страницы .....	125
7.6. Практическая работа № 5.	
Разработка библиотеки тегов .....	127
7.7. Практическая работа № 6.	
Применение параметров инициализации и атрибутов контекста web-приложения .....	129
7.8. Практическая работа № 7.	
Применение атрибутов запроса и сессии .....	131
7.9. Практическая работа № 8.	
Использование заголовков запроса и ответа .....	131
7.10. Практическая работа № 9.	
Использование фильтров и слушателей событий .....	132
7.11. Практическая работа № 10. Применение JDBC.....	133
7.12. Итоги главы .....	134
<b>ЛИТЕРАТУРА .....</b>	<b>135</b>

Учебное издание

**Смелов Владимир Владиславович**

**ОСНОВЫ  
WEB-ПРОГРАММИРОВАНИЯ  
НА JAVA**

Учебно-методическое пособие

Редактор *Л. Г. Кишко*

Компьютерная верстка *И. А. Ткаченко*

Подписано в печать 29.03.2008. Формат 60×80<sup>1</sup>/<sub>16</sub>.  
Бумага офсетная. Гарнитура Таймс. Печать офсетная.  
Усл. печ. л. 8,1. Уч.-изд. л. 8,0.  
Тираж 60 экз. Заказ .

Учреждение образования  
«Белорусский государственный технологический университет».  
220006. Минск, Свердлова, 13а.  
ЛИ № 02330/0133255 от 30.04.2004.

Отпечатано в лаборатории полиграфии учреждения образования  
«Белорусский государственный технологический университет».  
220006. Минск, Свердлова, 13.  
ЛП № 02330/0150477 от 16.01.2009.