

Министерство науки и высшего образования Российской Федерации

Томский государственный университет
систем управления и радиоэлектроники

В.Г. Резник

РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СЕТИ

Учебное пособие

Тема 4. Web-технологии распределенных систем

Томск
2020

Резник, Виталий Григорьевич

Распределенные вычислительные сети. Учебное пособие. Тема 4. Web-технологии распределенных систем / В.Г. Резник. – Томск : Томск. гос. ун-т систем упр. и радиоэлектроники, 2019. – 53 с.

В учебном пособии рассмотрены основные современные технологии организации распределенных вычислительных сетей, которые уже получили достаточно широкое распространение и подкреплены соответствующими инструментальными средствами реализации распределенных приложений. Представлены основные подходы к распределенной обработке информации. Проводится обзор организации распределенных вычислительных систем: методы удалённых вызовов процедур, многослойные клиент-серверные системы, технологии гетерогенных структур и одноранговых вычислений. Приводится описание концепции GRID-вычислений и сервис-ориентированный подход к построению распределенных вычислительных систем. Рассматриваемые технологии подкрепляются описанием инструментальных средств разработки программного обеспечения, реализованных на платформе языка Java.

Пособие предназначено для студентов бакалавриата по направлению 09.03.01 «Информатика и вычислительная техника» при изучении курсов «Вычислительные системы и сети» и «Распределенные вычислительные системы».

Одобрено на заседании каф. АСУ протокол №_____ от _____

УДК 004.75

© Резник В. Г., 2019

© Томск. гос. ун-т систем упр. и радиоэлектроники, 2019

Оглавление

4 Тема 4. Web-технологии распределенных систем.....	4
4.1 Общее описание технологии web.....	5
4.1.1 Унифицированный идентификатор ресурсов (URI).....	5
4.1.2 Общее представление ресурсов (HTML).....	7
4.1.3 Протокол передачи гипертекста (HTTP).....	9
4.2 Модели «Клиент-сервер».....	11
4.2.1 Распределение приложений по уровням.....	12
4.2.2 Типы клиент-серверной архитектуры.....	13
4.3 Технология Java-сервлетов.....	15
4.3.1 Классы Servlet и HttpServlet.....	18
4.3.2 Контейнер сервлетов Apache Tomcat.....	20
4.3.3 Диспетчер запросов — RequestDispatcher.....	30
4.3.4 Технология JSP-страниц.....	33
4.3.5 Модель MVC.....	44
Вопросы для самопроверки.....	53

4 Тема 4. Web-технологии распределенных систем

Данная глава посвящена изучению технологий *web*, зародившихся как частная задача интеграции научного информационного документооборота сотрудников «Европейского совета по ядерным исследованиям» (CERN), но со временем превратившихся в глобальный гипертекстовый проект, известный как «Всемирная паутина». Согласно Википедии [41]: «**Всемирная паутина** (англ. *World Wide Web*) — распределенная система, предоставляющая доступ к связанным между собой документам, расположенным на различных компьютерах, подключённых к сети Интернет. Для обозначения Всемирной паутины также используют слово **веб** (англ. *web* «паутина») и аббревиатуру **WWW**. Всемирную паутину образуют сотни миллионов веб-серверов. Большинство ресурсов Всемирной паутины основано на технологии гипертекста. Гипертекстовые документы, размещаемые во Всемирной паутине, называются веб-страницами. Несколько веб-страниц, объединённых общей темой или дизайном, а также связанных между собой ссылками и обычно находящихся на одном и том же веб-сервере, называются веб-сайтом. Для загрузки и просмотра веб-страниц используются специальные программы — браузеры (англ. *browser*). Всемирная паутина вызвала настоящую революцию в информационных технологиях и дала мощный толчок развитию Интернета. ...».

Первоначально, *web* не создавался для реализации широкого класса распределённых систем, но он предоставил три технологических новинки:

1. **URI** (*Uniform Resource Identifier*) — унифицированный идентификатор ресурса.
2. **HTML** (*HyperText Markup Language*) — гипертекстовый язык разметки.
3. **Браузер** (*web browser*) — прикладная программа для просмотра содержимого HTML-файлов, включающих файлы рисунков.

Существуют различные мнения, но на мой взгляд — именно браузеры, которые стали распространяться с каждой ОС, сделали общедоступным язык HTML, что привлекло к участию в развитии web-технологий широкий круг специалистов различных областей, сделало *web* популярным и экспериментальной площадкой для новых технологических решений в Интернет. Появился даже термин «*Тонкий клиент*», обозначающий браузер — как «*Автоматизированное рабочее место*» (*APM*). Что касается нашей предметной области, то наиболее важной является идея URI, которая на практике показала свою состоятельность и обратила внимание теоретиков распределенных систем на проблему адресации большого количества ресурсов.

Сам учебный материал данной главы разделен на три части:

- *первая часть* — общее описание технологии web;
- *вторая часть* — развитие модели «*Клиент-сервер*»;
- *третья часть* — реализация web-технологий с использованием языка Java.

4.1 Общее описание технологии web

Считается, что Тимоти Джон Бернерс-Ли придумал web в 1990 году. Им были изобретены идентификаторы URI, язык HTML и протокол HTTP. В период с 1991 по 1993 годы он усовершенствовал эти стандарты и опубликовал их. Кроме того, им впервые в мире были написаны:

- а) web-сервер, названный «*httpd*»;
- б) web-браузер, названный «*WorldWideWeb*».

В концептуальном плане Бернерс-Ли ввёл понятие **ресурса**, которым может быть что угодно: произвольные рисунки, записи, чертежи и все остальное, что может быть идентифицировано и передано с одного компьютера на другой. В такой проекции, распределенная система рассматривается как некий «механизм» по перемещению ресурсов с одной машины на другую. Дополнительно, Бернерс-Ли показал, что такой «механизм» может быть реализован и для этого нужны три простые технологии:

1. *Адресация ресурсов.* Необходимы гибкие и расширяемые способы именования произвольных ресурсов, например, URI.
2. *Представление ресурсов.* Необходимо такое представление ресурсов, чтобы они могли быть представлены в виде потока бит и переданы по сети. Кроме того, такое представление должно быть понятно всем и принято всеми, например, HTML.
3. *Передача ресурсов.* Необходим протокол передачи по типу «Клиент-сервер», который поддерживает минимальный необходимый набор операций передачи данных, например, HTTP.

В такой последовательности мы и рассмотрим технологии web, понимая, что в конце 80-х годов единственным претендентом на глобальное применение был лишь стек протоколов TCP/IP.

4.1.1 Унифицированный идентификатор ресурсов (URI)

Официально, адресация ресурсов web осуществляется с помощью **URI** [42]: «URI — символьная строка, позволяющая идентифицировать какой-либо ресурс: документ, изображение, файл, службу, ящик электронной почты и т. д. Прежде всего, речь идёт о ресурсах сети Интернет и Всемирной паутины. URI предоставляет простой и расширяемый способ идентификации ресурсов. Расширяемость URI означает, что уже существуют несколько схем идентификации внутри URI, и ещё больше будет создано в будущем. ...».

URI имеет две формы представления:

1. **URL** — определяет, где и как найти ресурс. В 2002 году (см. RFC 3305) анонсировано

устаревание термина URL.

2. **URN** — определяет, как ресурс идентифицировать.

Согласно [43]: «**Единый указатель ресурса** (от англ. *Uniform Resource Locator* — унифицированный указатель ресурса, **URL** — система унифицированных адресов электронных ресурсов, или единообразный определитель местонахождения ресурса (файла). Используется как стандарт записи ссылок на объекты в Интернет (Гипертекстовые ссылки во «всемирной паутине» www). ...».

Именно такие ссылки мы обычно используем в работе с Интернет, а также будем использовать в учебной работе, поэтому рассмотрим формат URL(URI) более подробно:

`<схема>:[//[<логин>[:<пароль>]@]<хост>[:<порт>]][/<URL-путь>][?<параметры>][#<якорь>]`

где, в этой записи:

- **квадратные скобки** - необязательные конструкции;
- **<схема>** - схема обращения к ресурсу, обычно обозначает протокол: *http*, *https*, *ftp* и другие;
- **<логин>** - имя пользователя, используемое для доступа к ресурсу;
- **<пароль>** - пароль пользователя;
- **<хост>** - доменное имя компьютера или его цифровой IP-адрес;
- **<порт>** - номер порта транспортного уровня;
- **<URL-путь>** - путь к ресурсу относительно корневого каталога, определённого программным обеспечением сервера;
- **<параметры>** - строка запроса к серверу, допустимая в методе **GET** протокола HTTP (HTTPS); параметры передаются парами **имя=значение** и разделяются символами «&»;
- **<якорь>** - элемент языка HTML, дающий команду браузеру переместиться к нужной части принятой страницы.

Недостатки адресации URL — хорошо известны:

- а) использование ограниченного набора ASCII-символов, делающего нечитаемыми слова национальных языков;
- б) сильная привязка к стеку протоколов TCP/IP, требующая указания адреса сети и порта транспортного соединения;
- в) наличие привязок к тексту HTML и методам протокола HTTP.

В целом, адресация URL имеет достаточно простую семантику, что обеспе-

чило ей большую популярность, хотя сам Бернерс-Ли считает ее излишней, поскольку раскрывает структуру распределенных систем.

Другой подход, связанный с идентификацией ресурсов, имеет следующие характеристики [44]: «URN (англ. *Uniform Resource Name*) — единообразное название (имя) ресурса. ... URN — это *постоянная* последовательность символов, идентифицирующая абстрактный или физический ресурс. URN является частью концепции **URI** (англ. *Uniform Resource Identifier*) — единообразных идентификаторов ресурса. Имена URN призваны в будущем заменить локаторы **URL** (англ. *Uniform Resource Locator*) — единообразные определители местонахождения ресурсов. Но имена URN, в отличие от URL, не включают в себя указания на местонахождение и способ обращения к ресурсу. Стандарт URN специально разработан так, чтобы он мог включать в себя другие пространства имён. ...».

Общий формат URN:

urn:<NID>:<NSS>

В этой записи:

- **<NID>** - нечувствительный к регистру идентификатор пространства имён (*Namespace Identifier*), представляющий собой статическую интерпретацию NSS;
- **<NSS>** - строка некоторого определённого пространства имён (*Namespace Specific String*), состоящая, как и URL, из ограниченного набора ASCII-символов.

Мы не будем подробно рассматривать адресацию URN, отправляя желающих узнать подробности к источникам RFC 3401 - RFC 3406.

4.1.2 Общее представление ресурсов (HTML)

Язык HTML безусловно является символом Интернет [45]: «**HTML** (от англ. *HyperText Markup Language* — «язык гипертекстовой разметки») — стандартизированный язык разметки документов во Всемирной паутине. Большинство веб-страниц содержат описание разметки на языке HTML (или XHTML). Язык HTML интерпретируется браузерами; полученный в результате интерпретации форматированный текст отображается на экране монитора компьютера или мобильного устройства. Язык HTML до 5-й версии определялся как приложение SGML (стандартного обобщённого языка разметки по стандарту ISO 8879). Спецификации HTML5 формулируются в терминах DOM (объектной модели документа). Язык XHTML является более строгим вариантом HTML, он следует синтаксису XML и является приложением языка XML в области разметки гипертекста. ...».

Язык HTML — достаточно прост в изучении. Имеет множество учебников, описывающих его. Базовые принципы этого языка можно изучить даже в Википе-

дии [46]. Нас он интересует прежде всего как средство представления ресурсов распределенных систем. В этом плане, он первоначально обеспечивал:

- а) предоставление ссылок (адресацию) на различные ресурсы Интернет;
- б) форматирование текста, включая представление списков и таблиц;
- в) отображение рисунков в формате *.gif*;
- г) интерактивные средства взаимодействия клиента с удаленным приложением сервера с помощью конструкции `<FORM>`, включающей поля ввода текста, радиокнопок, кнопок отправки формы по методам GET и POST протокола HTTP, а также кнопки очистки формы.

В целом, язык HTML содержит мало средств, ориентированных на описание самих ресурсов распределенных приложений. К тому же он только даёт описание разметки, а само представление обеспечивает приложение — браузер. Поэтому разработчики браузеров стали включать в HTML различные расширения:

- вставку различных объектов: не-HTML документов и *media*-файлов;
- вставку различных языков сценариев, например, *JavaScript* [47].

Существенным событием web-технологий стала возможность включения в текст языка HTML объектов языка Java, известных как *апплеты* [48]: «**Java-апплет** — прикладная программа, чаще всего написанная на языке программирования Java в форме байт-кода. Java-апплеты выполняются в веб-обозревателе с использованием виртуальной Java машины (JVM), или в Sun's AppletViewer, автономном средстве для испытания апплетов. *Java-апплеты были внедрены в первой версии языка Java в 1995 году.* Java-апплеты обычно пишутся на языке программирования Java, но могут быть написаны и на других языках, которые компилируются в байт код Java, таких, как Jython. Поддержка апплетов исключена из Java, начиная с версии 11. Апплеты используются для предоставления интерактивных возможностей веб-приложений, которые не могут быть предоставлены HTML. Так как байт-код Java — платформонезависим, то Java-апплеты могут выполняться с помощью плагинов браузерами многих платформ, включая Microsoft Windows, UNIX, Apple Mac OS и GNU/Linux. ...».

В настоящее время, наиболее популярной является технология AJAX [49]: «**AJAX**, Ajax (от англ. *Asynchronous Javascript and XML* — «асинхронный JavaScript и XML») — подход к построению интерактивных пользовательских интерфейсов веб-приложений, заключающийся в «фоновом» обмене данными браузера с веб-сервером. В результате, при обновлении данных веб-страница не перезагружается полностью, и веб-приложения становятся быстрее и удобнее. ...».

Серьёзные претензии к HTML возникли в процессе создания проекта электронного документа. Были предложения полностью перейти на язык XML [50], но пока этого не случилось и язык HTML остаётся полноправным членом всех существующих web-технологий.

4.1.3 Протокол передачи гипертекста (HTTP)

Не менее значимым чем HTML в web-технологиях является протокол HTTP [51]: «**HTTP** (англ. *HyperText Transfer Protocol* — «протокол передачи гипертекста») — протокол прикладного уровня передачи данных изначально — в виде гипертекстовых документов в формате «HTML», в настоящий момент используется для передачи произвольных данных. Основой HTTP является технология «клиент-сервер», то есть предполагается существование:

- а) **потребителей** (клиентов), которые иницируют соединение и посылают запрос;
- б) **поставщиков** (серверов), которые ожидают соединения для получения запроса, производят необходимые действия и возвращают обратно сообщение с результатом. ...

HTTP используется также в качестве «транспорта» для других протоколов прикладного уровня, таких как SOAP, XML-RPC, WebDAV. Основным объектом манипуляции в HTTP является ресурс, на который указывает URI (Uniform Resource Identifier) в запросе клиента. Обычно такими ресурсами являются хранящиеся на сервере файлы, но ими могут быть логические объекты или что-то абстрактное. Особенностью протокола HTTP является возможность указать в запросе и ответе способ представления одного и того же ресурса по различным параметрам: формату, кодировке, языку и т. д. (в частности, для этого используется HTTP-заголовок). Именно благодаря возможности указания способа кодирования сообщения, клиент и сервер могут обмениваться двоичными данными, хотя данный протокол является текстовым. ...».

Широкую популярность в приложениях протокол HTTP получил благодаря опоре на стек протоколов транспортного уровня TCP/IP. Это делает его универсальным способом прикладного взаимодействия в Интернет. Кроме того, благодаря текстовому характеру передаваемых сообщений, он легко адаптируется, подвергается анализу и отладке во время проектирования распределенного взаимодействия, а также включает в себя достаточно большой набор стандартизированных **методов**:

- а) OPTIONS — метод, используемый для получения информации о поддерживаемых расширениях сервера;
- б) GET — основной метод запроса ресурсов в браузерах;
- в) POST — дополнительный метод запроса ресурса, используемый в браузерах с помощью специальных конструкций <FORM>;
- г) HEAD — метод, по форме запроса аналогичный методу GET, но ответ сервера не содержит тела ресурса;
- д) PUT — аналогичен методу POST, но содержит более детальный диалог;
- е) PATCH - аналогичен методу PUT, но применяется к фрагменту ресурса;

- ж) DELETE — предназначен для удаления ресурсов;
- з) TRACE — обеспечивает трассировку запросов;
- и) CONNECT — использует соединение с сервером как TCP/IP-туннель.

Таким образом, идея Бернерса-Ли о трёх базовых ресурсах Всемирной паутины получила не только простейшую реализацию, применимую для научного информационного документооборота, но и стала развиваться в других приложениях, которые, в свою очередь, стимулировали ее развитие и все большую популярность. Такая ситуация очень быстро привела к необходимости дополнительных исследований, связанных со все более возрастающим объёмом обрабатываемой информации и источников ее размещения. Это потребовало более подробного изучения базовой модели сетевого взаимодействия модели «Клиент-сервер». Рассмотрим этот вопрос более подробно.

4.2 Модели «Клиент-сервер»

Во всем предшествующем учебном материале, начиная с первой главы (см. пункт 1.2.1), присутствует базовая сетевая парадигма «Клиент-сервер» [14]. Она присутствует как в концепции самого сетевого взаимодействия модели OSI [13], так и в базовом программном обеспечении (см. подраздел 2.5, «Управление сетевыми соединениями»). Далее ее классическая интерпретация рассматривается в подразделе 2.6 как «Организация доступа к базам данных», и, наконец, вся тема 3 полностью посвящена брокерным архитектурам, относящихся к группе «Объектных распределенных систем».

Фактически, с момента своего появления словосочетание «Клиент-сервер» превратилось в некий зонтичный термин, в котором, по определению, присутствуют две две части:

1. **Клиент** — активная часть, инициирующая сетевое взаимодействие и являющаяся конечным потребителем результатов вычислений.
2. **Сервер** — пассивная часть, обслуживающая одного или множество клиентов.

Простое масштабирование данной парадигмы, как модели построения распределённых вычислительных сетей (РВ-сетей), потребовало введение промежуточного программного обеспечения (*Middleware*), которое для объектного подхода было реализовано в виде ПО *брокеров*, регистрирующих наличие и местоположение серверов. Сами брокеры не участвовали в конечном прикладном взаимодействии клиента и сервера. Они обеспечивали только начало такого взаимодействия, реализуя «Службу имён». Широкое распространение web-технологий актуализировало другой аспект сетевого взаимодействия, предложив использовать парадигму «Тонкий клиент» взамен парадигмы «Автоматизированное рабочее место» (*АРМ*).

Сразу заметим, что идея *тонкого клиента*, под которым понимается программное обеспечение браузеров, является отражением более общей проблемы, с которой сталкиваются все проектировщики РВ-сетей. Она выражается в виде простой дилеммы о распределении функциональной нагрузки между ПО клиента и ПО сервера. В таком виде, использование тонкого клиента напоминает концепцию виртуальной машины Java (JVM), реализуемой для каждой платформы ЭВМ, только на более высоком (прикладном) уровне. Но прямое применение этой идеи позитивно лишь до тех пор, пока в силу не вступают вопросы надёжности, безопасности и другие технические или юридические ограничения.

В общем случае, к приложениям можно применить два общих типа распределения: *вертикальные* и *горизонтальные*. В следующих пунктах мы рассмотрим:

- а) распределение приложений по уровням, отдельно выделяющим и отражающим концепцию тонкого клиента;
- б) выделение ряда общих типов клиент-серверной архитектуры, включающих вертикаль-

ные и горизонтальные распределения.

4.2.1 Распределение приложений по уровням

Парадигма «Тонкий клиент», ставшая основой публичного использования web-технологий, породила много дебатов и споров. В результате, сторонники бурно развивающихся в 90-е годы информационных технологий предложили трёх-уровневую архитектуру построения РВ-сетей. Она включала:

- верхний уровень представления (пользовательского интерфейса);
- средний уровень бизнес-логики (функциональная поддержка приложений);
- нижний уровень данных (накопление, хранение и извлечение данных).

В чем-то такая архитектура напоминала идеи сетевых моделей OSI и DoD (*Department of Defense*), перенесённых в понятийное пространство прикладных систем. Рассмотрим эту архитектуру более подробно.

Уровень представления — ПО поддержки пользовательского интерфейса, расположенное на машине клиента. Оно может варьироваться от простейших терминальных систем, обеспечивающих только символьный доступ к ПО сервера, до сложных мультимедиа систем, обеспечивающих все возможности графики, звука и других интерактивных технологий. Появился даже новый термин: «*Богатый клиент*».

Уровень бизнес-логики — это совокупность правил, принципов и зависимостей поведения объектов предметной области системы. Фактически, — это и есть функциональная часть приложения, реализованная на сервере, но не включающая хранилище используемых данных, которое реализовано как отдельная система.

Уровень данных — это сервер или набор серверов, содержащих программы, которые хранят и предоставляют данные программам уровня бизнес-логики. Фактически, — это сервера, содержащие базы данных управляемые некоторыми СУБД.

Хотя такая архитектура может показаться устаревшей и напоминающей эпоху использования мейнфреймов, она имеет серьёзные позитивные обоснования:

- тенденция создания сложных масштабных приложений, работоспособность которых может быть обеспечена только организациями, содержащими высококлассных специалистов;
- тенденция создания важных приложений, которые юридически могут обслуживаться только специализированными организациями;
- тенденция мобильности и низкого уровня подготовки клиентов, которые требуют централизованного и профессионального обслуживания.

Сложно прогнозировать, будет ли указанная архитектура преобладающей в будущих РВ-сетях, поскольку все современные тенденции негативно относятся к

централизованным решениям, но её простота и наглядность вполне применимы к небольшим проектам уровня предприятия.

4.2.2 Типы клиент-серверной архитектуры

В предыдущем пункте был рассмотрен *вертикальный тип* распределения приложений, который называется «*Трёхзвенной архитектурой*». В целом, можно выделить три варианта таких архитектур.

Однозвенная архитектура — модель, по функциональности эквивалентная терминальному доступу к мэйнфреймам. В простейшем виде она соответствует системам телеобработки данных, ранее показанным на рисунке 1.3. В более сложных случаях, это могут быть, например, графические X-станции, которые в своё время выпускала компания Sun Microsystems. Такие станции представляли собой компьютер со встроенным сетевым программным обеспечением. После включения, такая станция соединялась с сервером, загружала программу *login* и, после успешной авторизации, загружала программное обеспечение графического X-сервера, который обеспечивал доступ к размещённой на самом сервере рабочей области пользователя.

Двухзвенная архитектура — модель, демонстрирующая функциональность АРМ или «*толстого клиента*». Она показана на рисунке 4.1.

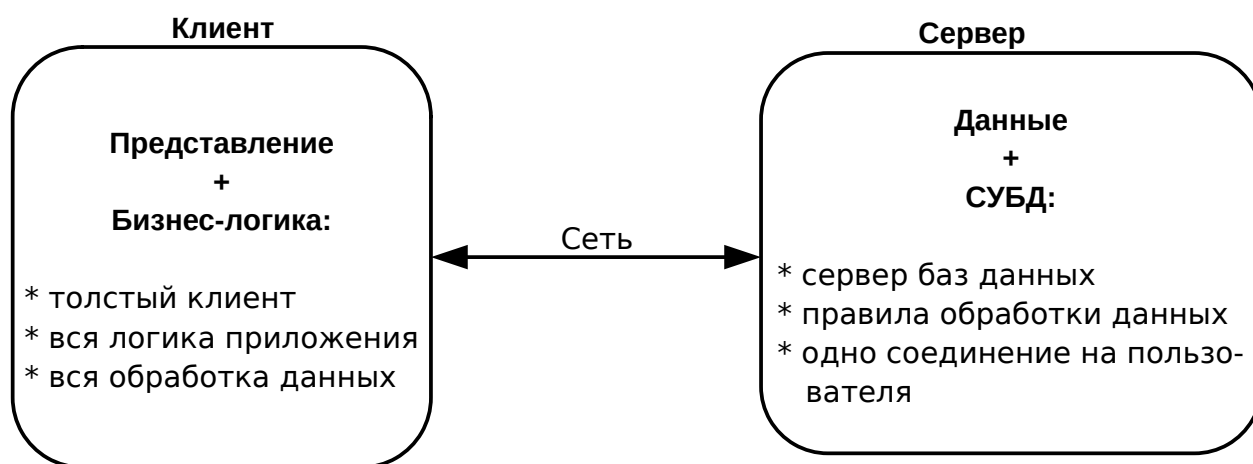


Рисунок 4.1 — Двухзвенная архитектура «Клиент-сервер»

Примером такой архитектуры является приложение *Example11*, реализованное в пункте 2.6.3 главы 2, а также подобные реализации в вариантах технологий CORBA и RMI. Тот факт, что в этих примерах использовался вариант встроенной СУБД Apache Derby не играет никакой роли, поскольку сама технология доступа к СУБД предполагает соединение с сервером баз данных.

Трёхзвенная архитектура - модель, демонстрирующая функциональность «*тонкого клиента*». Ее архитектура представлена на рисунке 4.2, а особенности

функционирования — следующие:

- **Клиент** обеспечивает только представление данных и диалог с сервером приложений;
- **Сервер приложений** несёт всю нагрузку обработки данных, определённую бизнес-логикой приложения.



Рисунок 4.2 — Трёхзвенная архитектура «Клиент-сервер»

Общей характерной особенностью **вертикального типа распределения** приложений является *размещение на разных машинах логически разных компонент приложения*.

Альтернативой вертикальному распределению является **горизонтальное распределение**, когда логически одинаковые компоненты клиентов и серверов размещаются на разных машинах. Схематически, такой вариант показан на рисунке 4.3, где центральный элемент — «*Сервер распределения*» — распределяет нагрузки запросов от множества клиентов по множеству серверов.

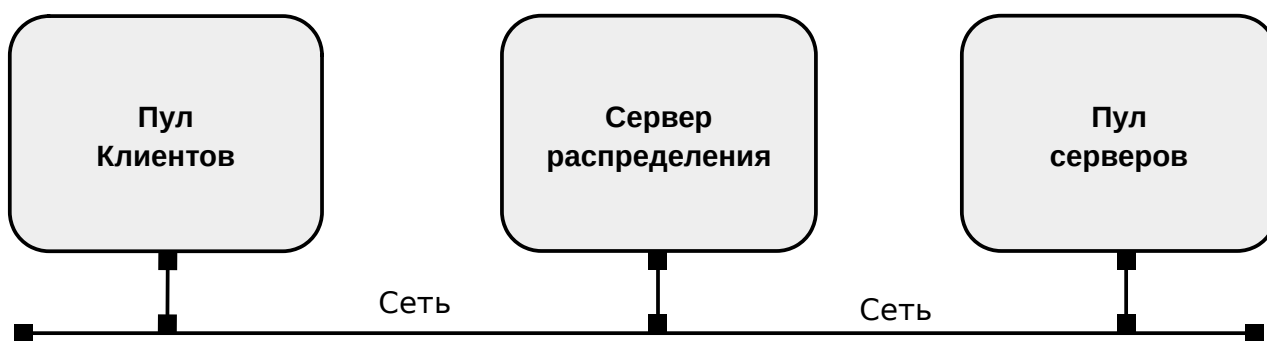


Рисунок 4.3 — Горизонтальное распределение архитектуры «Клиент-сервер»

В учебных условиях достаточно сложно реализовать горизонтальную архитектуру распределения компонент приложения, поэтому мы её больше не будем рассматривать. Главное, что необходимо отметить, — горизонтальное распределение обычно создаётся *методами репликации*, когда логически одинаковые компоненты приложений постоянно синхронизируются между собой. Другими словами, если какой-то логический компонент изменил своё значение, то аналогичные компоненты изменяют своё значение на всех машинах сети.

4.3 Технология Java-сервлетов

Первоначально, *www-технологии* зародились из желания иметь публикации, которые по качеству представления были бы сравнимы с книгами, газетами и другими традиционными документами, но могли бы находиться на компьютерах и свободно распространяться по сети Интернет. Для этой цели были разработаны:

- а) язык *HTML*, для упрощённой разметки электронного документа;
- б) *browser*, способный обеспечить разметку и просмотр электронного документа, написанного на языке HTML;
- в) технология *CGI (Common Gateway Interface)*, описывающая общие требования к web-серверу, способному хранить или формировать страницы HTML;
- г) протокол *HTTP*, обеспечивающий взаимодействие *браузера* и *web-сервера* по передаче запросов на получение документов.

При таком подходе, простейшие публикации, предоставляющие статический текст на языке HTML, могли формироваться любой программной или в любом текстовом редакторе.

Пример такого простейшего текста показан на листинге 4.1.

Листинг 4.1 — Исходный текст простейшей HTML-страницы

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Проект proj14</title>
</head>
<body>
  <h2 align="center">
    РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СЕТИ
  </h2>
  <h4 align="center">
    бакалавриат кафедры АСУ
  </h4>
  <hr> Адрес этой страницы:
  <a href="http://localhost:8080/proj14/Title.html">
    http://localhost:8080/proj14/Title.html
  </a>
  <hr>
  <a href="http://localhost:8080/proj14/Example14">
    Запуск сервлета Example14
  </a>
  <hr>
</body>
</html>
```

Текст листинга содержит: *титульную часть*, *заголовок* в виде надписей разным размером шрифта и *две адресные ссылки*. Если этот текст поместить, например в файл ***\$HOME/src/rvs/Title.html***, то его можно запустить из файлового менеджера и отразить в браузере, что и показано на рисунке 4.4.

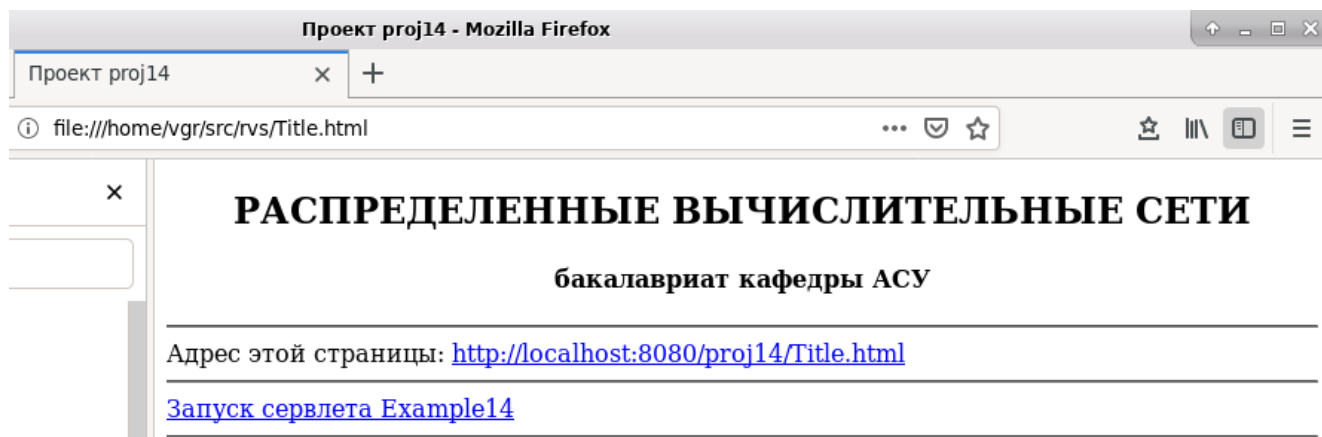


Рисунок 4.4 — Демонстрация в браузере статического HTML-текста

В 1995 году, на основе общих требований CGI, появляются новые инструменты:

- свободный web-сервер Apache [52]: «**Apache HTTP-сервер** (... назван именем группы племён североамериканских индейцев апачей; ...) — свободный веб-сервер. Apache является кроссплатформенным ПО, поддерживает операционные системы Linux, BSD ... Основными достоинствами Apache считаются надёжность и гибкость конфигурации. Он позволяет подключать внешние модули для предоставления данных, использовать СУБД для аутентификации пользователей, модифицировать сообщения об ошибках и т. д. Поддерживает IPv6. ...»;
- серверный препроцессорный язык PHP [53]: «**PHP** (англ. *PHP: Hypertext Preprocessor* — «PHP: препроцессор гипертекста»; первоначально *Personal Home Page Tools* — «Инструменты для создания персональных веб-страниц») — скриптовый язык общего назначения, интенсивно применяемый для разработки веб-приложений. В настоящее время поддерживается подавляющим большинством хостинг-провайдеров и является одним из лидеров среди языков, применяющихся для создания динамических веб-сайтов. ...»;
- встраиваемый язык JavaScript [47]: «**JavaScript** (... аббр. **JS** ...) — мультипарадигменный язык программирования. ... Является реализацией языка ECMAScript (стандарт ECMA-262). JavaScript обычно используется как встраиваемый язык для программного доступа к объектам приложений. Наиболее широкое применение находит в браузерах как язык сценариев для придания интерактивности веб-страницам. ...».

Благодаря новым технологиям возможности Всемирной паутины значительно расширились, появилась возможность стандартными средствами языка PHP создавать динамические web-страницы. Тем не менее для разработки приложений уровня предприятия они ещё не годились. Причина была в том, что сценарии PHP, которые на стороне сервера вставлялись в текст HTML-страниц, постоянно под-

вергались синтаксическому контролю. Это не позволяло кэшировать страницы, что снижало быстродействие и требовало больших ресурсов web-серверов.

В 1997 году, Sun Microsystems предложила *технологии сервлетов* [54]: «Сервлет является интерфейсом Java, реализация которого расширяет функциональные возможности сервера. Сервлет взаимодействует с клиентами посредством принципа запрос-ответ. Хотя сервлеты могут обслуживать любые запросы, они обычно используются для расширения веб-серверов. Для таких приложений технология Java Servlet определяет HTTP-специфичные сервлет классы. Пакеты **javax.servlet** и **javax.servlet.http** обеспечивают интерфейсы и классы для создания сервлетов. Первая спецификация сервлетов была создана в Sun Microsystems (версия 1.0 была закончена в июне 1997). ...». В сочетании с апплетами, которые были созданы ещё в 1995 году, появилась возможность проектировать полноценные графические приложения, используя только функциональные возможности браузеров.

Несмотря на предоставление новых возможностей, технологии Java встретили и *недружественное противодействие*:

- а) разработчикам браузеров не нравилось, что апплеты, имея своё собственное окно, делают ненужным и сам рендеринг HTML-страниц; поэтому, обосновывая сложную поддержку «громоздкой» Java-машины (JRE), которая ещё и может нарушать безопасность компьютера, они «спускали на тормозах» нормальную работу апплетов;
- б) необходимость создавать HTML-страницы на языке Java не повышало популярности предложенной технологии;
- в) использование апплетами пакета *javax.swing* делает их графическое представление «чужеродным».

С 1999 года, Sun Microsystems стала формировать отдельную платформу для разработки приложений уровня предприятия — **J2EE** (Java EE), а также выпускать новый сервер [55]: «**Tomcat** (в старых версиях – **Catalina**) — контейнер сервлетов с открытым исходным кодом, разрабатываемый Apache Software Foundation. Реализует спецификацию сервлетов, спецификацию JavaServer Pages (JSP) и JavaServer Faces (JSF). Написан на языке Java. **Tomcat** позволяет запускать веб-приложения и содержит ряд программ для самоконфигурирования. **Tomcat** используется в качестве самостоятельного веб-сервера, в качестве сервера контента в сочетании с веб-сервером Apache HTTP Server, а также в качестве контейнера сервлетов в серверах приложений JBoss и GlassFish. ...». Таким образом, был создан полный набор инструментов, позволяющих развивать web-технологии Java, даже без ориентации на технологию апплетов.

Чтобы иметь более полное представление о сказанном выше, мы рассмотрим ряд конкретных примеров:

- а) краткое описание классов *Servlet* и *HttpServlet*;
- б) контейнер сервлетов Apache Tomcat;

- в) технологию JSP-страниц;
- г) модель проектирования MVC, - применительно к технологии сервлетов.

4.3.1 Классы *Servlet* и *HttpServlet*

Официальную документацию на описание сервлетов можно найти, например, в источнике [56], где указано, что в пакете *javax.servlet* имеется *публичный интерфейс Servlet*, содержащий описание пяти методов:

- а) `void destroy()` — вызывается контейнером сервлета, чтобы указать сервлету, что он выводится из эксплуатации;
- б) `ServletConfig getServletConfig()` — возвращает объект *ServletConfig*, который содержит параметры инициализации и запуска для этого сервлета.
- в) `String getServletInfo()` — возвращает информацию о сервлете, такую как автор, версия и авторские права.
- г) `void init(ServletConfig config)` — вызывается контейнером сервлета, чтобы указать сервлету, что он вводится в эксплуатацию.
- д) `void service(ServletRequest req, ServletResponse res)` — вызывается контейнером сервлета, чтобы сервлет мог ответить на запрос.

Используемый в программировании сервлет *HttpServlet* является расширением абстрактного класса *GenericServlet*, включающего интерфейс *Servlet*, как это показано на рисунке 4.5.

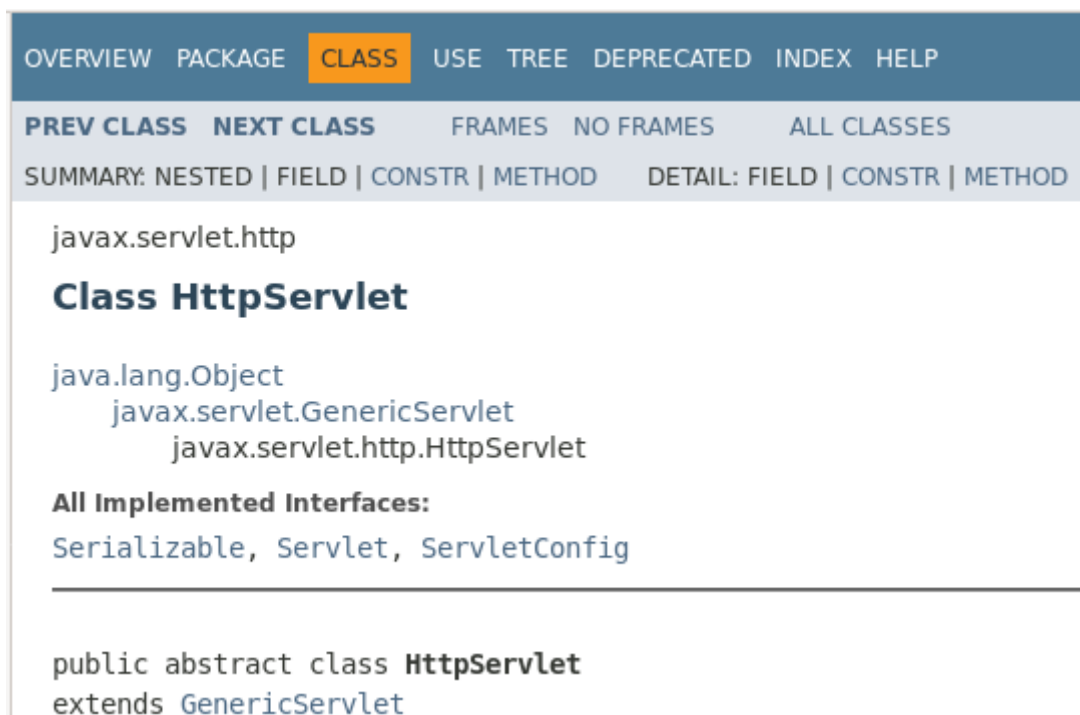


Рисунок 4.5 — Структура зависимостей для абстрактного класса `HttpServlet`

Документация [56] так характеризует **HttpServlet**: «... предоставляет абстрактный класс, который можно разделить на подклассы для создания HTTP-сервлета, подходящего для веб-сайта. Подкласс HttpServlet должен переопределить хотя бы один метод, обычно один из следующих:

- а) **doGet(...)**, если сервлет поддерживает запросы HTTP GET;
- б) **doPost(...)**, для запросов HTTP POST;
- в) **doPut(...)**, для запросов HTTP PUT;
- г) **doDelete(...)**, для запросов HTTP DELETE;
- д) **init(...)** и **destroy(...)**, чтобы управлять ресурсами, которые управляют жизненным циклом сервлета;
- е) **getServletInfo(...)**, когда необходимо предоставить информацию о себе.

Нет необходимости переопределять метод **service(...)**. Он обрабатывает стандартные HTTP-запросы, отправляя их методам-обработчикам для каждого типа HTTP-запроса (перечисленные выше методы **doXXX**). Аналогично, почти нет причин переопределять методы **doOptions(...)** и **doTrace(...)**.

Сервлеты обычно работают на многопоточных серверах, поэтому имейте в виду, что *сервлет должен обрабатывать параллельные запросы*, и соблюдайте осторожность при синхронизации доступа к общим ресурсам. К общим ресурсам относятся данные в памяти, такие как переменные экземпляра или класса, и внешние объекты, такие как файлы, соединения с базой данных и сетевые соединения. ...».

Любой сервлет, который создаёт проектировщик, является обычным публичным Java-классом, который расширяет абстрактный класс **HttpServlet**. Его «жизненный цикл» состоит из трёх периодов:

1. Когда сервер стартует, то загружает доступные ему сервлеты. При этом, каждый сервлет выполняет метод **init(...)**. При необходимости, проектировщик может переопределить этот метод.
2. В процессе работы сервера, сервлет выполняет методы, которые запрашивает клиент, кроме методов **init(...)** и **destroy(...)**. При необходимости, проектировщик переопределяет нужные методы, обычно — методы **doGet(...)** и **doPost(...)**.
3. Когда сервер завершает работу, он для каждого сервлета вызывает его метод **destroy(...)**. При необходимости, проектировщик может переопределить этот метод.

В процессе работы, *сервер параллельно обслуживает запросы клиентов*. Если необходимо, чтобы сервлет обслуживал только одного клиента в определённый момент времени, нужно реализовать интерфейс **SingleThreadModel** в добавление к наследованию абстрактного класса **HttpServlet**. При этом, нет необходимости вносить каких либо изменений в реализацию самого сервлета.

На практике обычно переопределяются два метода: **doGet(...)** и **doPost(...)**.

Оба они имеют одинаковые аргументы:

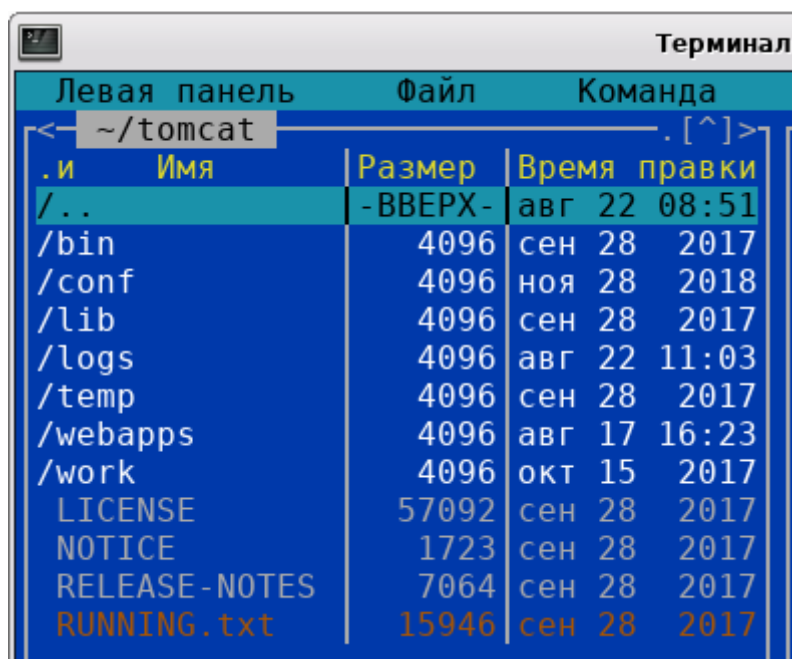
- `ServletRequest req` — объект запроса, получающий информацию от клиента (браузера);
- `ServletResponse res` — объект ответа, передаваемый клиенту (браузеру).

Каждый из этих объектов имеет достаточно много собственных методов, которые нужно изучать по представленной в [56] документации. Мы, когда будем демонстрировать конкретные примеры, рассмотрим только наиболее важные из них и выполним их в среде разработки Eclipse EE.

4.3.2 Контейнер сервлетов Apache Tomcat

Как было отмечено ранее, классическим представителем контейнера сервлетов является Apache Tomcat [55]. Его можно загрузить с сайта [57]. Для учебных целей данного курса используется *Tomcat v8.5 Server*, установленный в домашней директории пользователя: *\$HOME/tomcat*.

На рисунке 4.6 показано содержимое этой директории.



Имя	Размер	Время	правки
./	-ВВЕРХ-	авг 22	08:51
/bin	4096	сен 28	2017
/conf	4096	ноя 28	2018
/lib	4096	сен 28	2017
/logs	4096	авг 22	11:03
/temp	4096	сен 28	2017
/webapps	4096	авг 17	16:23
/work	4096	окт 15	2017
LICENSE	57092	сен 28	2017
NOTICE	1723	сен 28	2017
RELEASE-NOTES	7064	сен 28	2017
RUNNING.txt	15946	сен 28	2017

Рисунок 4.6 — Структура каталогов дистрибутива Apache Tomcat

В лучших традициях ОС UNIX, назначение основных каталогов дистрибутива — следующее:

- *bin* — содержит служебные сценарии для администрирования сервера; причём, сценарии, имеющие расширение *.sh*, предназначены для ОС UNIX/Linux, а сценарии, имеющие расширение *.bat*, - для ОС MS Windows;

- **conf** — содержит различные файлы конфигурации сервера, включая обеспечение безопасности;
- **lib** — содержит JAR-архивы библиотек;
- **webapps** — базовое расположение реализованных на сервере проектов.

Минимальная настройка дистрибутива Apache Tomcat требует задания системной переменной *CATALINA_HOME*, что для нашего варианта установки имеет значение показанное на рисунке 4.7.

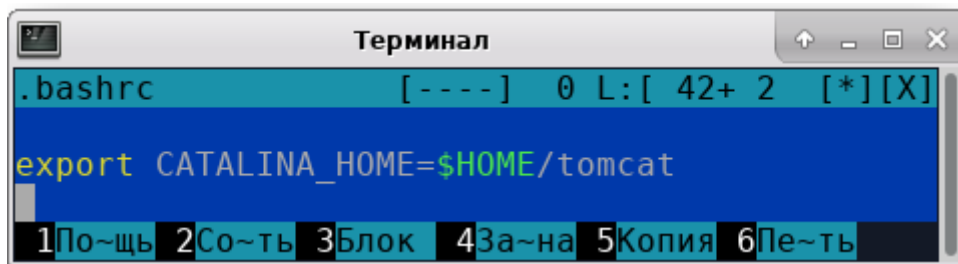


Рисунок 4.7 — Задание системной переменной *CATALINA_HOME*

Минимальная проверка работоспособности дистрибутива Apache Tomcat осуществляется с помощью сценария *startup.sh*, который запускается из каталога *\$CATALINA_HOME/bin*, как это демонстрируется рисунком 4.8.

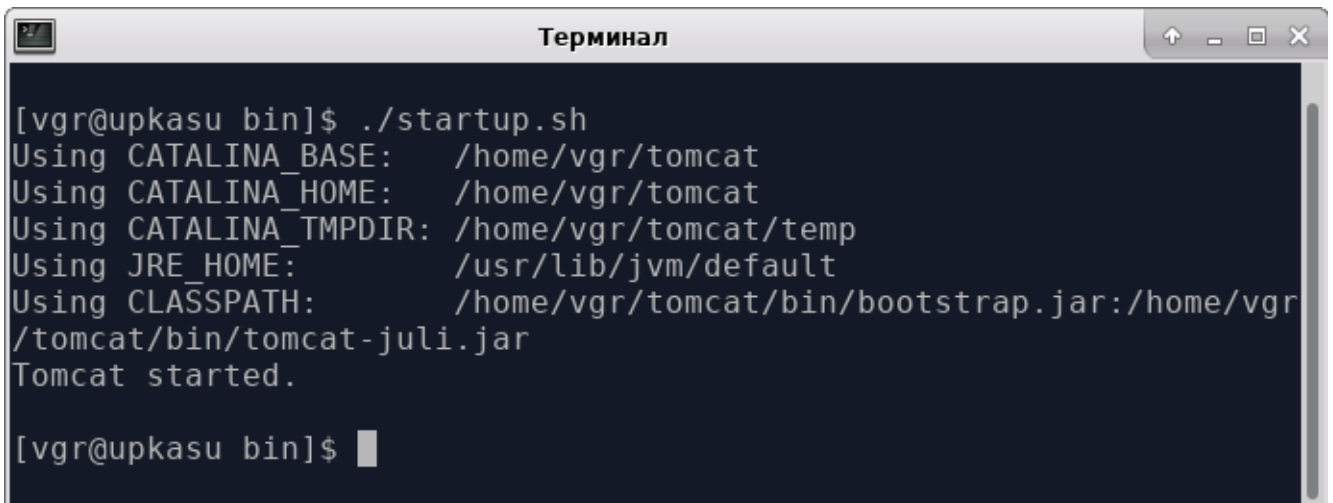


Рисунок 4.8 — Нормальный старт сервера Apache Tomcat

После старта, сервер Apache Tomcat начинает прослушивать **порт 8080**. Это отличает его от обычных web-серверов, которые по умолчанию прослушивают **порт 80**. Соответственно, для остановки запущенного сервера используется сценарий *shutdown.sh*.

В учебных примерах (при создании и тестировании сервлетов), мы будем использовать инструментальные средства среды Eclipse EE. Эта среда сама стартует сервер Apache Tomcat, отображая его дистрибутив в своём адресном пространстве реализации проектов. Стартуемый в Eclipse EE сервер Apache также использует порт 8080, поэтому, *перед началом использования среды разработки, Apache Tomcat должен быть остановлен!*

Демонстрацию запуска сервера из среды Eclipse EE проведём в рамках отдельного проекта с именем *proj14*. Для этого, выберем **File→New→Dynamic Web Project** и, в появившемся окне, укажем имя проекта, используемую версию Apache Tomcat и местоположение его дистрибутива. Правильный результат установок показан на рисунке 4.9.

The screenshot shows the 'New Dynamic Web Project' dialog box in Eclipse. The title bar says 'New Dynamic Web Project'. The main heading is 'Dynamic Web Project' with a subtitle 'Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.' The fields are as follows:

- Project name:** proj14
- Project location:** ☒ Use default location. Location: /home/vgr/rvs/proj14. There is a 'Browse...' button.
- Target runtime:** Apache Tomcat v8.5. There is a 'New Runtime...' button.
- Dynamic web module version:** 3.1.
- Configuration:** Default Configuration for Apache Tomcat v8.5. There is a 'Modify...' button. Below this, it says: 'A good starting point for working with Apache Tomcat v8.5 runtime. Additional facets can later be installed to add new functionality to the project.'
- EAR membership:** ☐ Add project to an EAR. EAR project name: EAR. There is a 'New Project...' button.
- Working sets:** ☐ Add project to working sets. There is a 'New...' button. Working sets: [empty field]. There is a 'Select...' button.

At the bottom, there is a help icon (?) and four buttons: '< Back', 'Next >', 'Cancel', and 'Finish'.

Рисунок 4.9 — Открытие первого проекта Dynamic Web Project и привязка к нему дистрибутива сервера Apache Tomcat

После нажатия кнопки «*Finish*», проект **proj14** откроется с необходимой привязкой к используемому контейнеру сервлетов. На рисунке 4.10 показана базовая структура этого вновь созданного проекта. Эту структуру необходимо изучить и знать назначение ее основных каталогов.

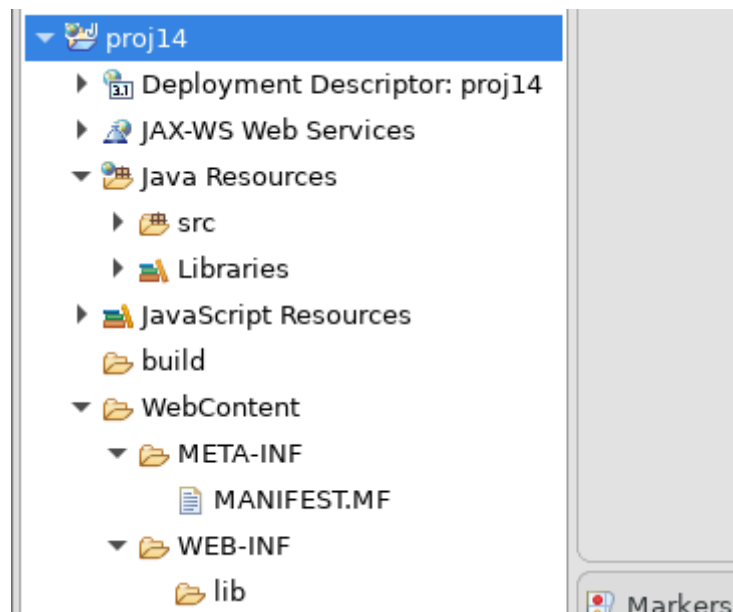


Рисунок 4.10 — Начальная структура каталогов проекта proj14

Общее назначение основных каталогов — следующее:

- а) **src** — каталог, предназначенный для хранения исходных текстов сервлетов проекта;
- б) **WebContent** — корневой (*root*) каталог проекта; помещённые в него каталоги и файлы задаются в абсолютной адресации, например, файл **index.html**, помещённый в него, будет адресоваться как **/index.html**;
- в) **META-INF** — внутренний для проекта каталог, предназначенный для хранения его манифеста; созданный после завершения разработки JAR-архив проекта будет содержать этот манифест;
- г) **WEB-INF** - внутренний для проекта каталог, предназначенный для хранения JSP-страниц и других файлов, *доступных в относительной адресации и только для программного обеспечения этого проекта*;
- д) **lib** — каталог, в который помещают дополнительные библиотеки, используемые только самим проектом.

Проведём демонстрацию работы Apache Tomcat как обычного web-сервера, для чего выделим мышкой каталог WebContent и правой кнопкой активируем меню, в котором выберем **New→HTML File**, а далее:

- указываем имя файла **Title.html** и нажимаем кнопку «**Next >**» (см. рисунок 4.11);

- выбираем тип HTML-файла: *New HTML File (5)* (см. рисунок 4.12);
- нажатием кнопки «*Finish*» получаем шаблон HTML-страницы (см. рисунок 4.13).

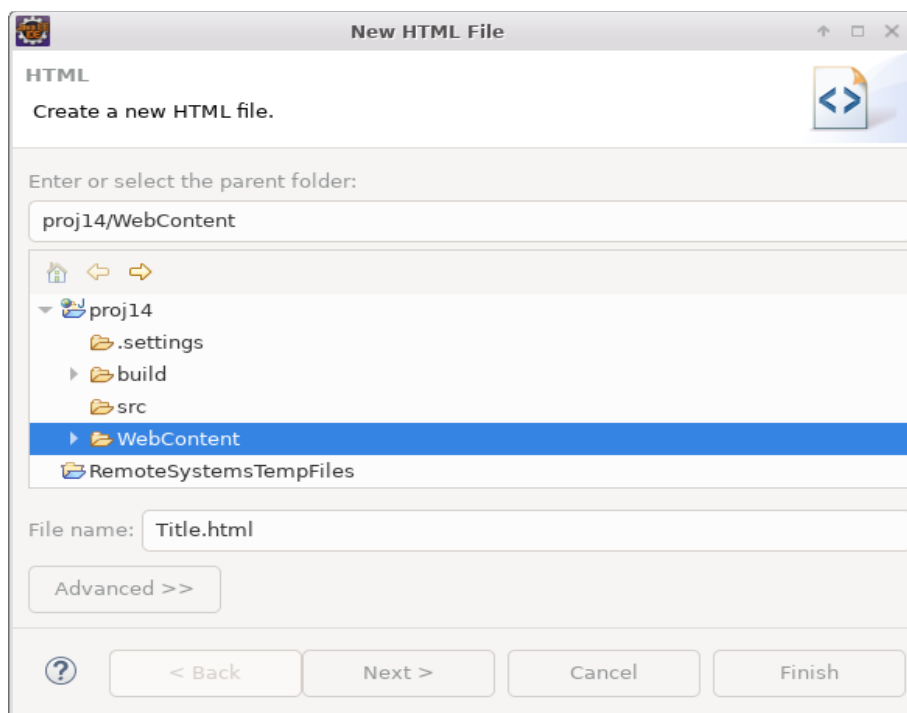


Рисунок 4.11 — Задание имени создаваемого HTML-файла

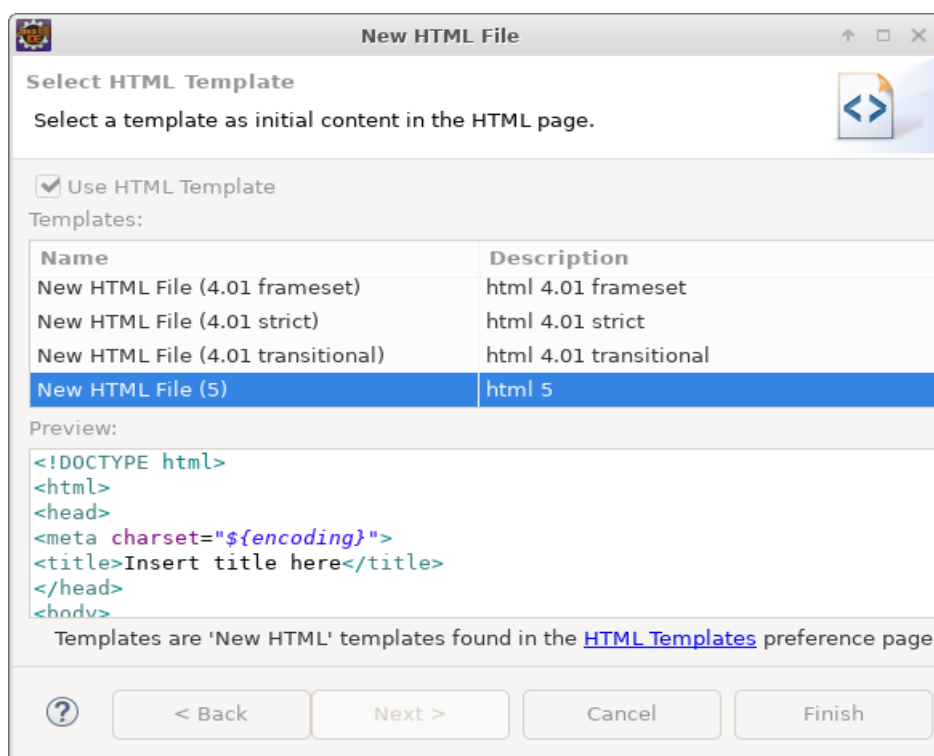
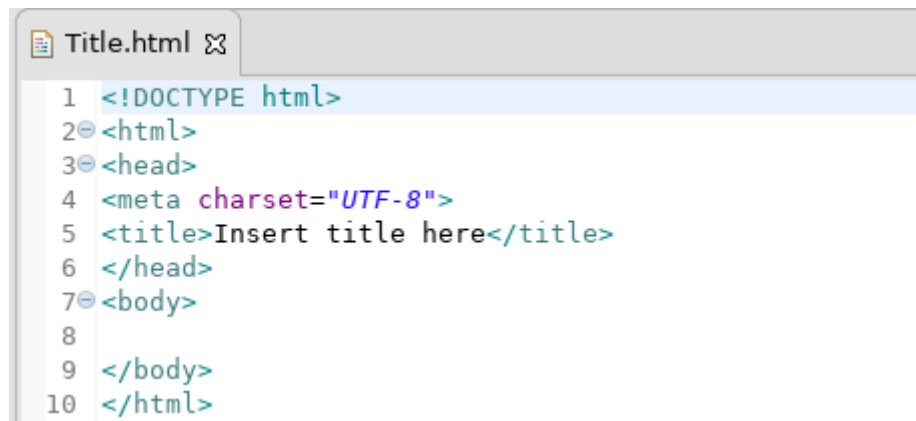


Рисунок 4.12 — Выбор версии шаблона создаваемого HTML-файла



```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset="UTF-8">
5 <title>Insert title here</title>
6 </head>
7 <body>
8
9 </body>
10 </html>
```

Рисунок 4.13 — Вкладка редактора Eclipse EE с шаблоном файла Title.html

Таким образом создаются и размещаются все статические HTML-страницы, адресуемые и доступные для просмотра из любого браузера.

Теперь, заменим содержимое данного шаблона на текст листинга 4.1, а затем запустим проект на выполнение.

Сначала появится окно с предложением выбрать и запустить сервер. Здесь нужно согласиться и появится отображение содержимого файла во встроенном браузере Eclipse EE. Результат такого запуска показан на рисунке 4.14.

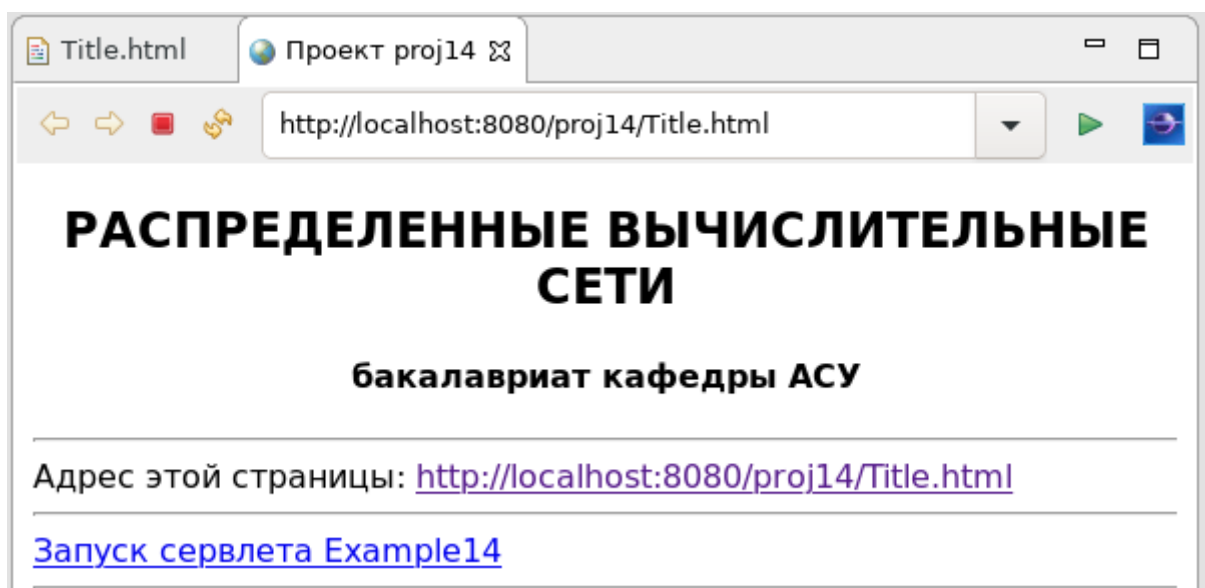


Рисунок 4.14 — Отображение файла Title.html во встроенном браузере Eclipse EE

Попытка запустить сервлет по второй ссылке закончится неудачей, поскольку сервлет *Example14* — ещё не создан. Браузер выведет стандартное сообщение, показанное на рисунке 4.15.

Таким образом, мы научились создавать статический контент реализуемого проекта и запускать его из среды Eclipse EE.

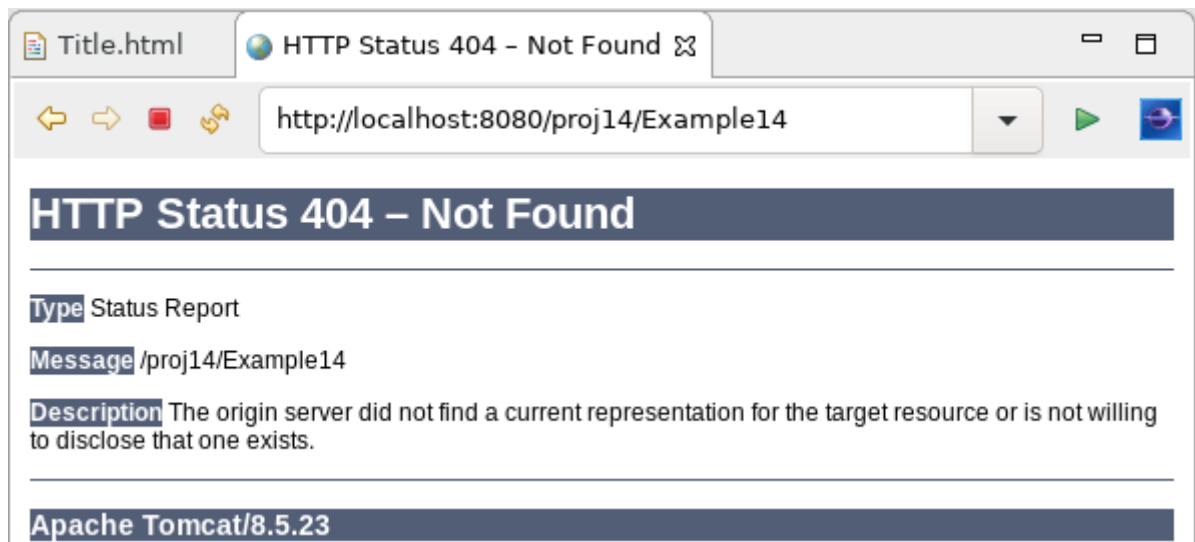


Рисунок 4.15 — Сообщение встроенного браузера об отсутствии адресуемого ресурса

Теперь создадим первый сервлет с именем *Example14*.

Для этого, в проекте *proj14* выделим каталог *src*, а затем правой кнопкой мышки активируем меню: *New*→*Servlet*. В появившемся окне укажем нужный пакет и имя сервлета, как это показано на рисунке 4.16.

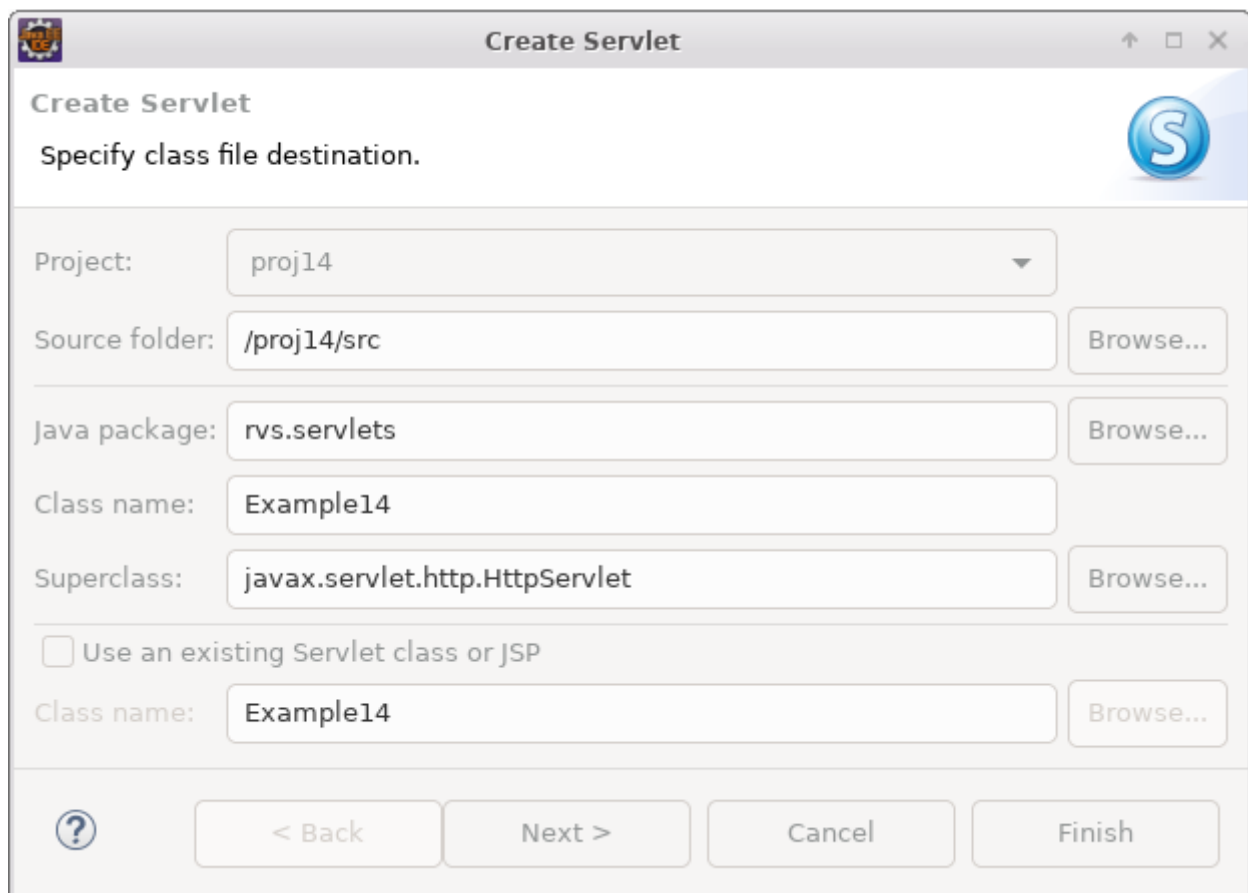


Рисунок 4.16 — Окно открытия сервлета Example14

Два раза нажав кнопку «*Next >*», мы переходим к окну, показанному на рисунке 4.17, где происходит выбор используемых сервлетом методов.

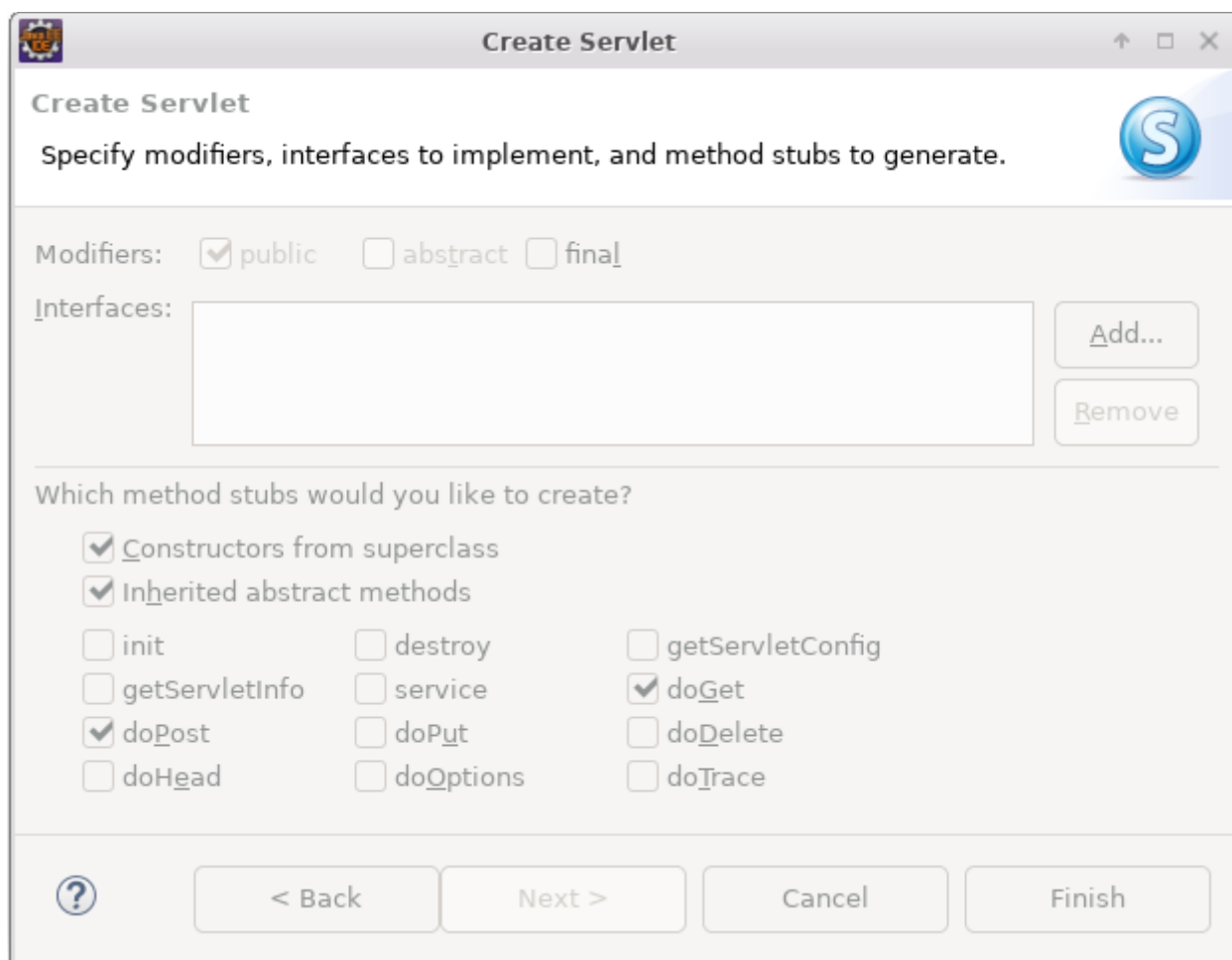


Рисунок 4.17 — Методы сервлета, предлагаемые по умолчанию

Обычно используются два метода *doGet(...)* и *doPost(...)*, которые запрашивают все браузеры, поэтому — нажимаем кнопку «*Finish*» и получаем результат, представленный на листинге 4.2.

Листинг 4.2 — Стандартный шаблон сервлета *Example14*

```
package rvs.servlets;

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

/**
 * Servlet implementation class Example14
 */
```

```

@WebServlet("/Example14")

public class Example14 extends HttpServlet {

    private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public Example14() {
        super();
        // TODO Auto-generated constructor stub
    }

    /**
     * @see HttpServlet#doGet(HttpServletRequest request,
     * HttpServletResponse response)
     */
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
    {
        // TODO Auto-generated method stub
        response.getWriter().append("Served at: ").append(
            request.getContextPath());
    }

    /**
     * @see HttpServlet#doPost(HttpServletRequest request,
     * HttpServletResponse response)
     */
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        // TODO Auto-generated method stub
        doGet(request, response);
    }
}

```

Первоначальный шаблон сервлета содержит: *имя пакета*, импортируемые по умолчанию классы и *определение публичного класса **Example14***, расширяющего абстрактный класс **HttpServlet**. Тело шаблона сервлета содержит:

- а) статическую константу **serialVersionUID**, используемую для идентификации сервлета;
- б) конструктор **Example14()**;
- в) метод **doGet(...)**, посылающий в качестве ответа текстовое сообщение;
- г) метод **doPost(...)**, просто вызывающий метод **doGet(...)**.

Если запустить этот сервер на выполнение, то он выдаст сообщение, показанное на рисунке 4.18.

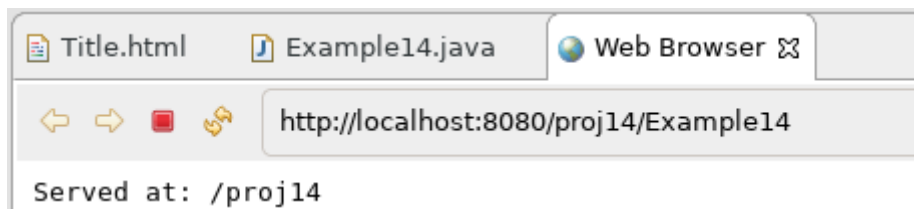


Рисунок 4.18 — Ответ шаблона сервлета Example14

Для дальнейшего изучения сервлетов необходимо учитывать следующие правила взаимодействия браузера и web-сервера:

1. Основным методом запроса браузера является метод GET, поэтому первым в сервлете нужно реализовать метод *doGet(...)*.
2. Браузеры разных ОС настроены на разные кодировки символов: **Cp1251** — для MS Windows и **UTF-8** — для UNIX/Linux. Сервлеты должны самостоятельно учитывать используемую браузерами кодировку, для этого, объекты запроса (*request*) и ответа (*response*) методов сервлета имеют соответствующие методы *getCharacterEncoding()* и *setCharacterEncoding(String str)*, обрабатывающие эту ситуацию.
3. Сервлет должен возвращать браузеру HTML-страницу с правильным типом контента, обычно - *text/html*. Поэтому объект *response* обязательно должен использовать метод *setContentType(String str)*.
4. При первом обращении к сервлету он компилируется в новый Java-класс, поэтому после изменения его исходного кода и запуске на выполнение, среда Eclipse EE предлагает перезапустить сервер и подключить к нему новый вариант сервлета.

Демонстрацию важности перечисленных правил провести очень просто. Для этого, в методе *doGet(...)* шаблона сервлета можно заменить текст "Served at: " на русский текст "Запрашивает: ". В результате, запрос к сервлету даст ответ, который будет выглядеть как показано на рисунке 4.19.

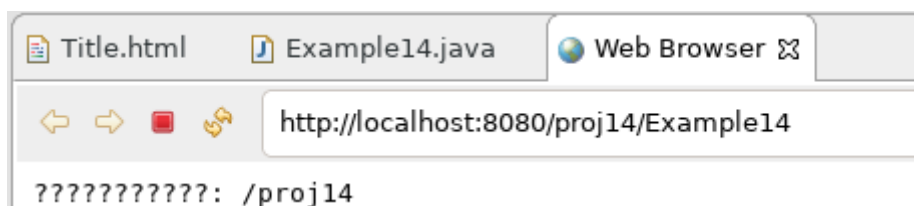


Рисунок 4.19 — Ответ шаблона сервлета, содержащего русскоязычный текст

4.3.3 Диспетчер запросов — *RequestDispatcher*

Сервлет получает запрос от браузера в виде объекта *request*, определённого интерфейсом *HttpServletRequest*, и проводит его анализ. Для этого объект запроса содержит множество методов, определяющих детали протокола HTTP:

- а) *String getCharacterEncoding()* — определение символьной кодировки запроса;
- б) *String getContentType()* — определение MIME-типа (Multipurpose Internet Mail Extension) пришедшего запроса;
- в) *String getProtocol()* — определение названия и версии протокола;
- г) *String getServerName()*, *getServerPort()* — определение имени сервера, принявшего запрос, и порта, на котором запрос был соответственно принят сервером;
- д) *String getRemoteAddr()*, *getRemoteHost()* — определение IP-адреса и имени хоста клиента, от которого пришел запрос;
- е) *String getRemoteUser()* — определение имени пользователя, выполнившего запрос;
- ж) *ServletInputStream getInputStream()*, *BufferedReader getReader()* — получение ссылки на поток, ассоциированный с содержимым полученного запроса.

После анализа запроса, разработчик должен привязать к объекту *request* некоторый ресурс сервера, который должен быть передан клиенту. Это делается с помощью реализации объекта интерфейса *RequestDispatcher*:

```
RequestDispatcher disp =  
    request.getRequestDispatcher(String path);
```

где *path* — абсолютный путь (в пределах сервлета) к подключаемому ресурсу.

Сама передача ресурса клиенту осуществляется методом *forward(...)* в виде:

```
disp.forward(request, response);
```

Для демонстрации этого стандартного решения, подключим к запросу уже имеющийся ресурс — файл *Title.html*. Для этого, преобразуем метод *doGet(...)* из листинга 4.2 к виду показанному на листинге 4.3.

Листинг 4.3 — Подключение *Title.html* в методе *doGet(...)* сервлета *Example14*

```
/**  
 * @see HttpServlet#doGet(HttpServletRequest request,  
 * HttpServletResponse response)
```

```

*/
protected void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException
{
    /**
     * Явная установка кодировок объектов запроса и ответа.
     * Стандартная установка контекста ответа.
     */
    request.setCharacterEncoding("UTF-8");
    response.setCharacterEncoding("UTF-8");
    response.setContentType("text/html");

    /**
     * Стандартное подключение ресурса сервлета.
     */
    RequestDispatcher disp =
        request.getRequestDispatcher("/Title.html");
    disp.forward(request, response);
}

```

После внесённых изменений и перезапуска сервлета, обращения по адресам <http://localhost:8080/proj14/Title.html> и <http://localhost:8080/proj14/Example14> выдают одинаковый результат показанный ранее на рисунке 4.14.

Указанный пример демонстрирует доступ сервлета к общедоступному ресурсу. Для внутренних ресурсов, недоступных прямой адресации из браузеров, в архитектуре сервлета имеется директория **WEB-INF**. Чтобы показать это, создадим в этом каталоге файл с именем **post1.html**, содержащий форму запроса к приложению ведения записей в базе данных, как это делалось в примерах использования технологий CORBA и RMI.

Содержимое такого файла показано на листинге 4.4.

Листинг 4.4 — Исходный текст файла post1.html сервлета Example14

```

<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Форма запроса</title>
</head>
<body>
<hr>
<b>Запрос к таблице ведения записей</b>
<hr>
<form action="Example14" method="post" accept-charset="UTF-8">
  <p> Введи ключ :
    <input type="text" size="10" name="key">
  </p>
  <p> Введи текст: <br>
    <textarea rows="10" cols="40" name="text"></textarea>
  </p>
  <p>
    <input type="submit">
  </p>
</form>

```

```
</p>
</form>
<hr>
</body>
</html>
```

Если мы напрямую обратимся к файлу *post1.html*, то получим ответ, показанный на рисунке 4.20, а если мы в методе *doGet(...)* создадим объект диспетчера в виде:

```
RequestDispatcher disp =
    request.getRequestDispatcher("/WEB-INF/post1.html");
```

то вызов сервлета покажет нужную страницу, приведённую на рисунке 4.21.

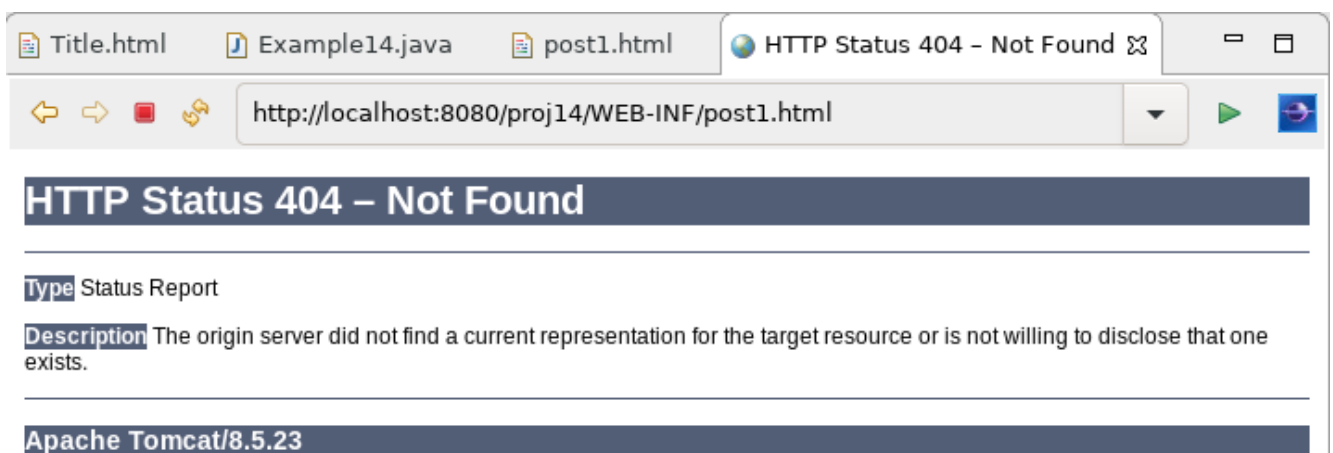


Рисунок 4.20 — Прямое обращение к файлу *post1.html*

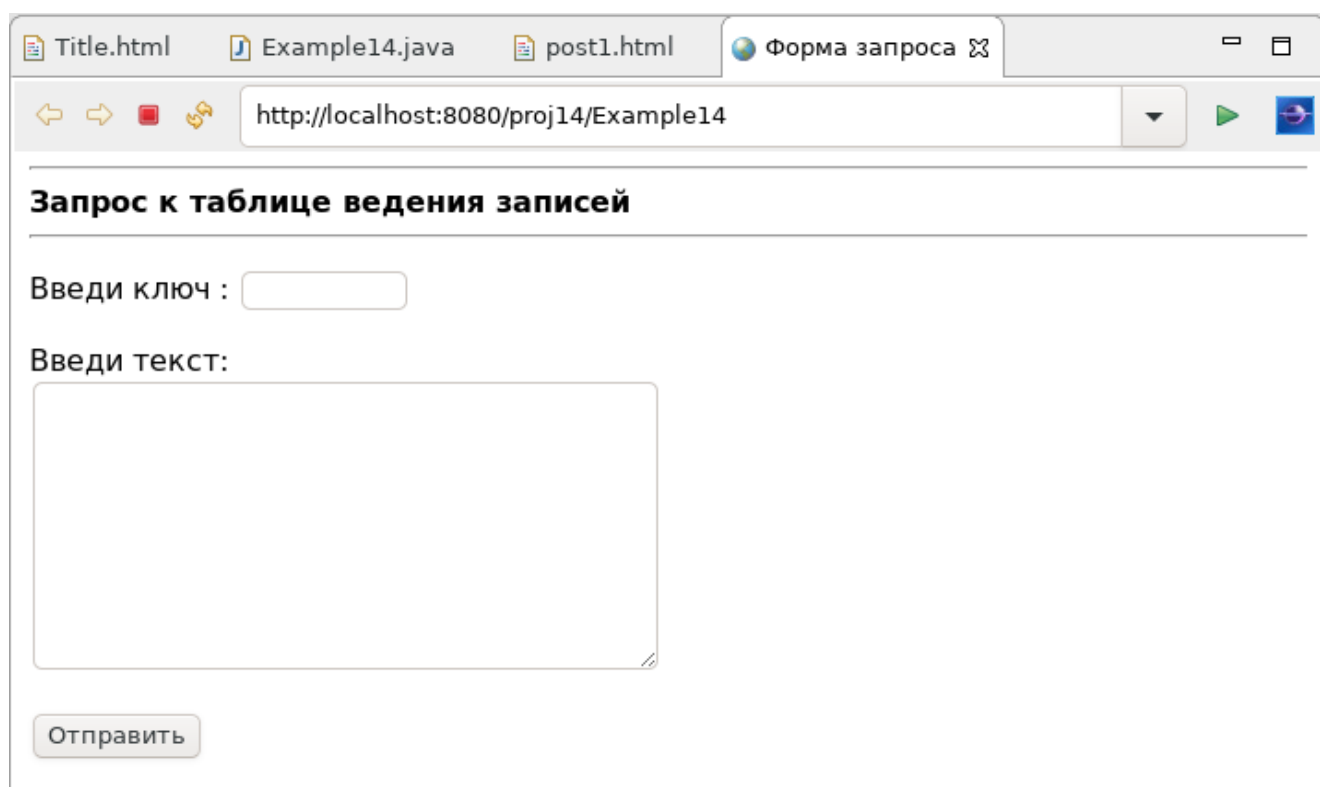


Рисунок 4.21 — Обращение к файлу post1.html из метода doGet(...)

4.3.4 Технология JSP-страниц

Наиболее существенным свойством сервера **Apache Tomcat** является реализация в нём технологии JSP-страниц [58]: «**JSP (JavaServer Pages)** — технология, позволяющая веб-разработчикам создавать содержимое, которое имеет как статические, так и динамические компоненты. Страница JSP содержит текст двух типов: *статические исходные данные*, которые могут быть оформлены в одном из текстовых форматов HTML, SVG, WML, или XML, и *JSP-элементы*, которые конструируют динамическое содержимое. Кроме этого могут использоваться библиотеки JSP-тегов, а также Expression Language (EL), для внедрения Java-кода в статичное содержимое JSP-страниц. Код JSP-страницы транслируется в Java-код сервлета с помощью компилятора JSP-страниц **Jasper**, и затем компилируется в байт-код виртуальной машины Java (JVM). Контейнеры сервлетов, способные исполнять JSP-страницы, написаны на платформенно-независимом языке Java. JSP-страницы загружаются на сервере и управляются из структуры специального Java server packet, который называется Jakarta EE Web Application. Обычно страницы упакованы в файловые архивы **.war** и **.ear**. Технология JSP, является платформенно-независимой, переносимой и легко расширяемой, для разработки веб-приложений. ...».

Создадим в нашем проекте JSP-страницу с именем **post2.jsp**, используя имеющиеся шаблоны среды Eclipse EE.

Для этого, в проекте выделим каталог **WEB-INF** и правой кнопкой мыши активируем меню: **New→JSP File**. В появившемся окне (см. рисунок 4.22) введём нужное имя файла и нажмём кнопку «**Next >**».

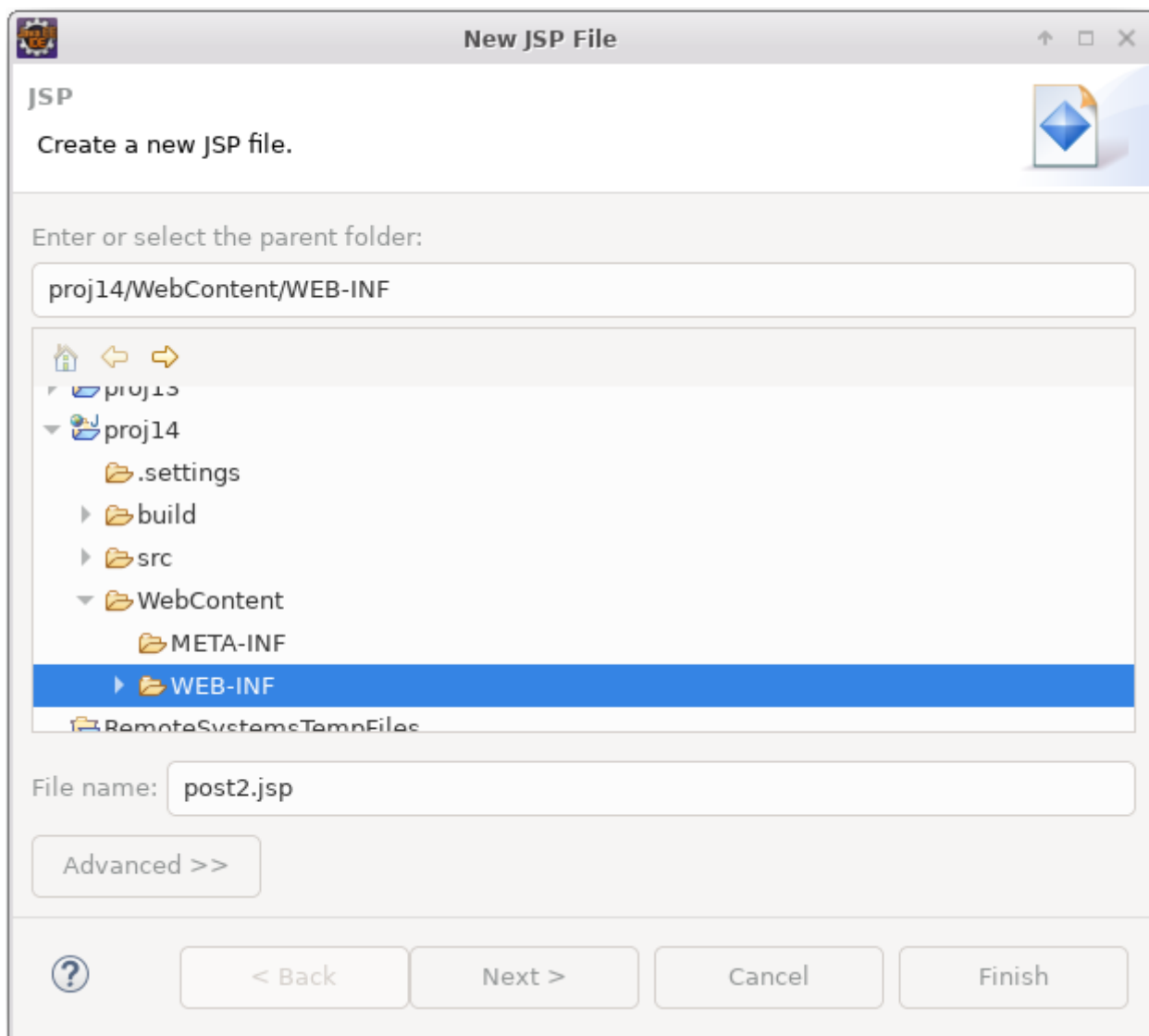


Рисунок 4.22 — Задание имени файла JSP-страницы

Появится новое окно, показанное на рисунке 4.23, в котором выбирается один из доступных шаблонов JSP-страниц. Выберем предложенное по умолчанию и нажмём кнопку «**Finish**».

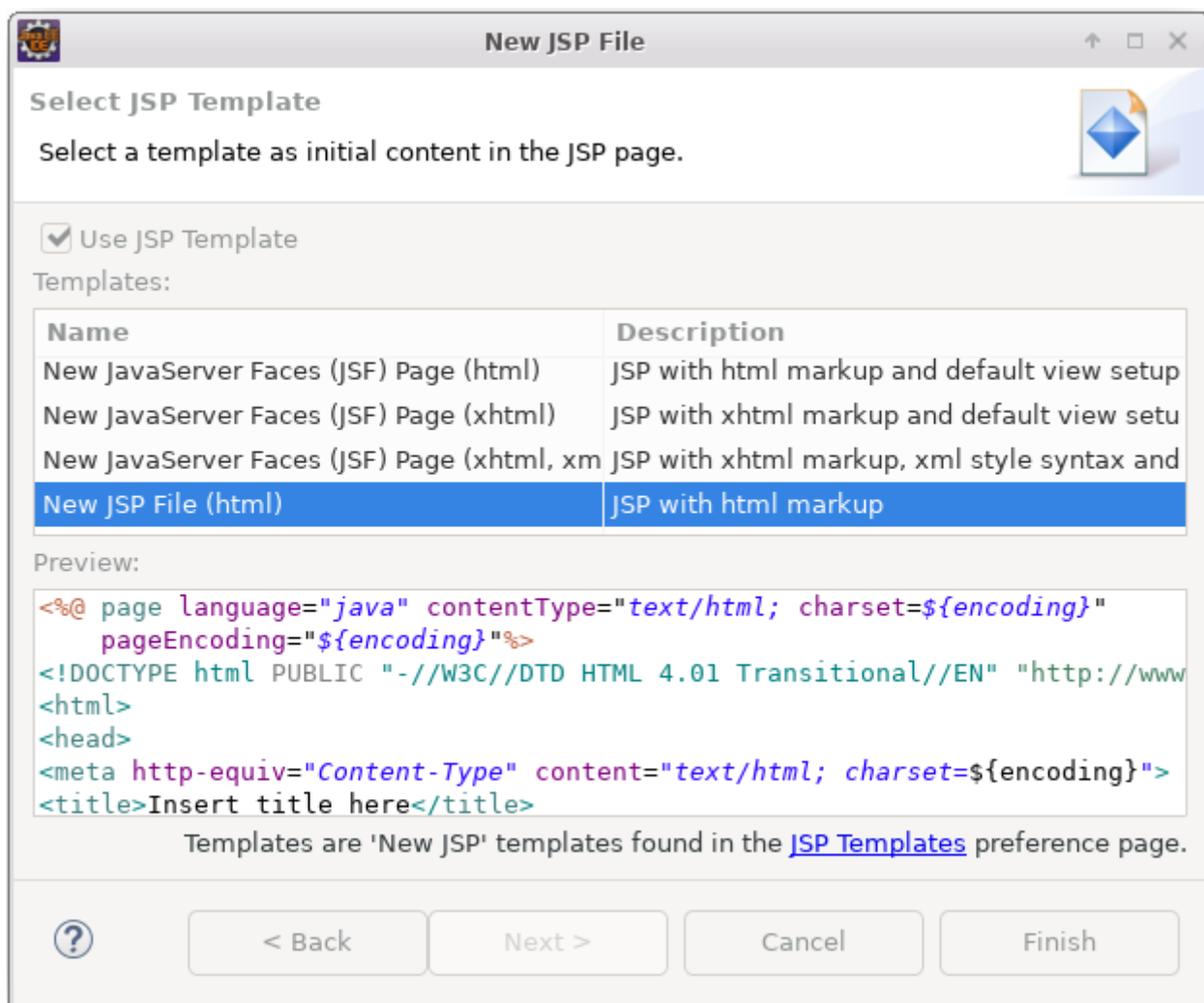


Рисунок 4.23 — Выбор шаблона файла JSP-страницы

В результате, в редакторе Eclipse EE появится новая вкладка с текстом выбранного шаблона для файла **post2.jsp**, что показано на листинге 4.5.

Листинг 4.5 — Исходный текст JSP-файла *post2.jsp* сервлета *Example14*

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!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>Insert title here</title>
</head>

<body>

</body>
</html>
```

Представленный **шаблон JSP-страницы**, является *типичным шаблоном страницы XHTML версии 4.01* дополненный в начале текста директивой, указывающей на используемый язык программирования, тип контента и используемую кодировку символов, которую можно заменить на любую другую. Далее, в этот текст можно вставлять конструкции языка HTML и синтаксические конструкции JSP-кода, представленные в таблице 4.1.

Таблица 4.1 — Синтаксические конструкции JSP-кода страниц

Элемент JSP	Синтаксис	Описание
Выражение JSP	<code><%= выражение %></code>	Выражение обрабатывается и направляется на вывод.
Скриплет JSP	<code><% код %></code>	Код добавляется в метод service.
Объявление JSP	<code><%! код %></code>	Код добавляется в тело класса сервлета, вне метода service.
Директива JSP page	<code><%@ page att="значение" %></code>	Директивы для движка сервлета с информацией об основных настройках.
Директива JSP include	<code><%@ include file="url" %></code>	Файл в локальной системе, подключаемый при трансляции JSP в сервлет.
Комментарий JSP	<code><%-- комментарий --%></code>	Комментарий; игнорируется при трансляции JSP страницы в сервлет.
Действие jsp:include	<code><jsp:include page="относительный URL" flush="true"/></code>	Подключает файл при запросе страницы.
Действие jsp:useBean	<code><jsp:useBean att=значение*/> or <jsp:useBean att=значение* ... </jsp:useBean></code>	Найти или создать Java Bean.
Действие jsp:setProperty	<code><jsp:setProperty att=значение*/></code>	Устанавливает свойства bean, или явно, или указанием на соответствующее значение параметра, передаваемое при запросе.
Действие jsp:getProperty	<code><jsp:getProperty name="ИмяСвойства" value="значение"/></code>	Получение и вывод свойств bean.
Действие jsp:forward	<code><jsp:forward page="относительныйURL"/></code>	Передаёт запрос другой странице.
Действие jsp:plugin	<code><jsp:plugin attribute="значение"*> ... </jsp:plugin></code>	Генерирует теги OBJECT или EMBED, в зависимости от типа браузера, в котором будет выполняться апплет, использующий Java Plugin.

В целом, все синтаксические конструкции таблицы 4.1 подразделяются на пять групп: *директивы*, *объявления*, *выражения*, *скриплеты* и *действия*.

Первая группа — *директивы JSP* распространяются на всю структуру класса, в который компилируется страница. Общий формат этой группы:

```
<%@ директива атрибут1="значение1"
    атрибут2="значение2"
    ...
    атрибутN="значениеN" %>
```

Существуют два основных типа директив:

- а) **page**, которая позволяет совершать такие операции, как импорт классов, изменение суперкласса сервлета и другие;
- б) **include**, которая даёт возможность вставлять файлы в тело JSP-страницы, при трансляции JSP файла в сервлет; эта директива имеет проблемы преобразования символов, поэтому ей нужно пользоваться с осторожностью.

Конкретный вариант использования директивы **page** показан в начале листинга 4.5. Общий список вариантов директивы **page** — следующий:

- а) `import="назем.class1,...,назем.classN"`.

Позволяет задать пакеты, которые должны быть импортированы.

Например:

```
<%@ page import="java.util.*" %>
import - единственный атрибут, допускающий многократное применение.
```

- б) `contentType="MIME-Tun"` или `contentType="MIME-Tun; charset=Кодировка-Символов"`
Задаёт тип MIME для вывода. По умолчанию используется **text/html**.

К примеру, директива:

```
<%@ page contentType="text/plain" %>
приводит к тому же результату, что и использование скриплета:
<% response.setContentType("text/plain"); %>
```

- в) `isThreadSafe="true|false"`.

Значение **true** - "истина", принимается по умолчанию, задаёт нормальный режим выполнения сервлета, когда множественные запросы обрабатываются одновременно с использованием одного экземпляра сервлета. Значение **false** ("ложь") сигнализирует о том, что сервлет должен наследовать *SingleThreadModel* (однопоточную модель), при которой последовательные или одновременные запросы обрабатываются отдельными экземплярами сервлета.

г) *session*="true|false".

Значение **true** - "истина", принимается по умолчанию, сигнализируя, что заранее определённая переменная *session* типа *HttpSession* должна быть привязана к существующей сессии, если таковая имеется. В противном случае, создаётся новая сессия, к которой и осуществляется привязка. Значение **false** ("ложь") определяет, что сессии не будут использоваться, и попытки обращения к переменной *session* приведут к возникновению ошибки при трансляции JSP страницы в сервлет.

д) *buffer*="размерkb|none".

Задаёт размер буфера для *JspWriter out*. Значение принимаемое по умолчанию зависит от настроек сервера, но должно превышать 8kb.

е) *autoflush*="true|false".

Значение **true**, принимаемое по умолчанию, устанавливает, что при переполнении буфер должен автоматически очищаться. Значение **false**, которое крайне редко используется, устанавливает, что переполнение буфера должно приводить к возникновению исключительной ситуации. При установке значения атрибута *buffer*="none", установка значения **false** для этого атрибута недопустимо.

ж) *extends*="нак.класс".

Задаёт суперкласс для генерируемого сервлета. Этот атрибут следует использовать с большой осторожностью, поскольку возможно что сервер уже использует какой-нибудь суперкласс.

з) *info*="сообщение".

Задаёт строку, которая может быть получена при использовании метода *getServletInfo*.

и) *errorPage*="url".

Задаёт JSP страницу, которая вызывается в случае возникновения каких-либо событий *Throwables*, которые не обрабатываются на данной странице.

к) *isErrorPage*="true|false".

Сигнализирует о том, может ли эта страница использоваться для обработки ошибок для других JSP страниц. По умолчанию принимается значение **false** ("ложь").

л) *language*="java".

Данный атрибут предназначен для задания используемого языка программирования. По умолчанию принимается значение "java", поскольку на сегодняшний день это единственный поддерживаемый язык программирования.

Директива **include** позволяет включать файлы в процессе трансляции JSP страницы в сервлет. Её использование имеет следующий формат:

`<%@ include file="абсолютный или относительный url" %>`

Рассмотрим четыре — наиболее часто используемые конструкции языка JSP, которые представлены таблицей 4.2.

Таблица 4.2 — Часто используемые конструкции языка JSP

Группа	Пояснение
<code><jsp:include page="url" /></code>	Подключение внешних файлов к странице JSP, в процессе обращения к ней.
<code><%! код %></code>	Объявление: объявление глобальных типов языка Java, в пределах JSP-страницы.
<code><%= выражение %></code>	Выражение языка Java, которое вычисляется и направляется на вывод в текстовом виде.
<code><% код %></code>	Скриплет: любой код на языке Java.

JSP-действие (`<jsp:include .../>`) является удобным, когда у нас имеются уже готовые HTML-страницы, которые можно включить в JSP-страницу. Например, файлы *Title.html* и *post1.html* можно включить в созданный шаблон двумя **действиями**:

```
<jsp:include page="/Tille.html" />
<jsp:include page="post1.html"/>
```

Остальные три конструкции таблицы 4.2 вставляют код языка Java в JSP-страницу. Для эффективности их использования, в странице доступны **четыре predefined типа объектов**:

- а) **request** (тип `HttpServletRequest`) — объект запроса к сервлету;
- б) **response** (тип `HttpServletResponse`) — объект ответа клиенту;
- в) **session** (тип `HttpSession`) — ассоциируется с запросом, если таковой имеется;
- г) **out** (тип `PrintWriter`) — используется для отсылки выводимых клиенту данных.

Кроме predefined объектов, в JSP-странице могут быть объявлены любые типы языка Java. Например, если мы хотим подсчитывать число обращений к странице, то можем создать переменную *accessCount* в виде **объявления**:

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

затем, использовать **выражение** для самого подсчёта:

```
<%= ++accessCount %>
```

Если необходимо проводить более сложные расчёты, используются конструкции *скриплетов*, например, вывод текущего времени и параметров запроса для нашего проекта будет выглядеть:

```
<%  
    out.println("Текущее время: " + new java.util.Date()+ "<br>");  
  
    out.println("Параметры запроса: <br>"  
        + "key = " + request.getParameter("key") + "<br>"  
        + "text = " + request.getParameter("text") );  
%>
```

Перенесём эти примеры в созданный шаблон *post2.jsp*, как это показано на листинге 4.6.

Листинг 4.6 — Изменённый текст JSP-файла post2.jsp сервлета Example14

```
<%@ page language="java" contentType="text/html; charset=UTF-8"  
    pageEncoding="UTF-8"%>  
<!DOCTYPE html>  
<html>  
<head>  
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  
<title>Файл post2.jsp</title>  
</head>  
<body>  
    <!-- Первое действие include -->  
    <jsp:include page="/Title.html" />  
    <p>  
    Это тело JSP-страницы post2.jsp, <br>  
    </p>  
  
    <!-- Объявление -->  
    <%! private int accessCount = 0; %>  
  
    <!-- Выражение -->  
    Количество обращений к странице:  
    <%= ++accessCount %><br>  
  
    <!-- Скриплет -->  
    <%  
        out.println("Текущее время: " + new java.util.Date()  
            + "<br>");  
  
        out.println("Параметры запроса:<br>"  
            + "key = " + request.getParameter("key") + "<br>"  
            + "text = " + request.getParameter("text"));  
    %>
```



```

<%-- Второе действие include --%>
<jsp:include page="post1.html" />

</body>
</html>

```

Теперь учтём, что форма, которая первоначально предоставляется клиенту методом *doGet(...)*, вызывает метод сервлета *doPost(...)*. Поэтому метод *doPost(...)* можно преобразовать так, чтобы он вызвал JSP-страницу *post2.jsp* (см. листинг 4.7).

Листинг 4.7 — Изменённый метод doPost(...) сервлета Example14

```

/**
 * @see HttpServlet#doPost(HttpServletRequest request,
 * HttpServletResponse response)
 */
protected void doPost(HttpServletRequest request,
                        HttpServletResponse response)
                        throws ServletException, IOException {
    /**
     * Явная установка кодировок объектов запроса и ответа.
     * Стандартная установка контекста ответа.
     */
    request.setCharacterEncoding("UTF-8");
    response.setCharacterEncoding("UTF-8");
    response.setContentType("text/html");

    /**
     * Стандартное подключение ресурса сервлета.
     */
    RequestDispatcher disp =
        request.getRequestDispatcher("/WEB-INF/post2.jsp");
    disp.forward(request, response);
}

```

Теперь, после запуска сервлета, можно заполнить форму запроса, например, как показано на рисунке 4.18.

Example14.java post2.jsp Форма запроса

http://localhost:8080/proj14/Example14

Запрос к таблице ведения записей

Введи ключ :

Введи текст:

Текст обращения к сервлету Example14.

Рисунок 4.24 — Заполнение формы запроса при первом запуске сервлета

Нажав кнопку «*Отправить*», мы получим ответ (см. рисунок 4.25).

Напомним, что сама JSP-страница *post2.jsp*, при первом обращении к ней, компилируется в сервлет и кэшируется сервером Apache Tomcat. Это делает технологию JSP-страниц очень мощной и приемлемой для разработки приложений уровня предприятий.

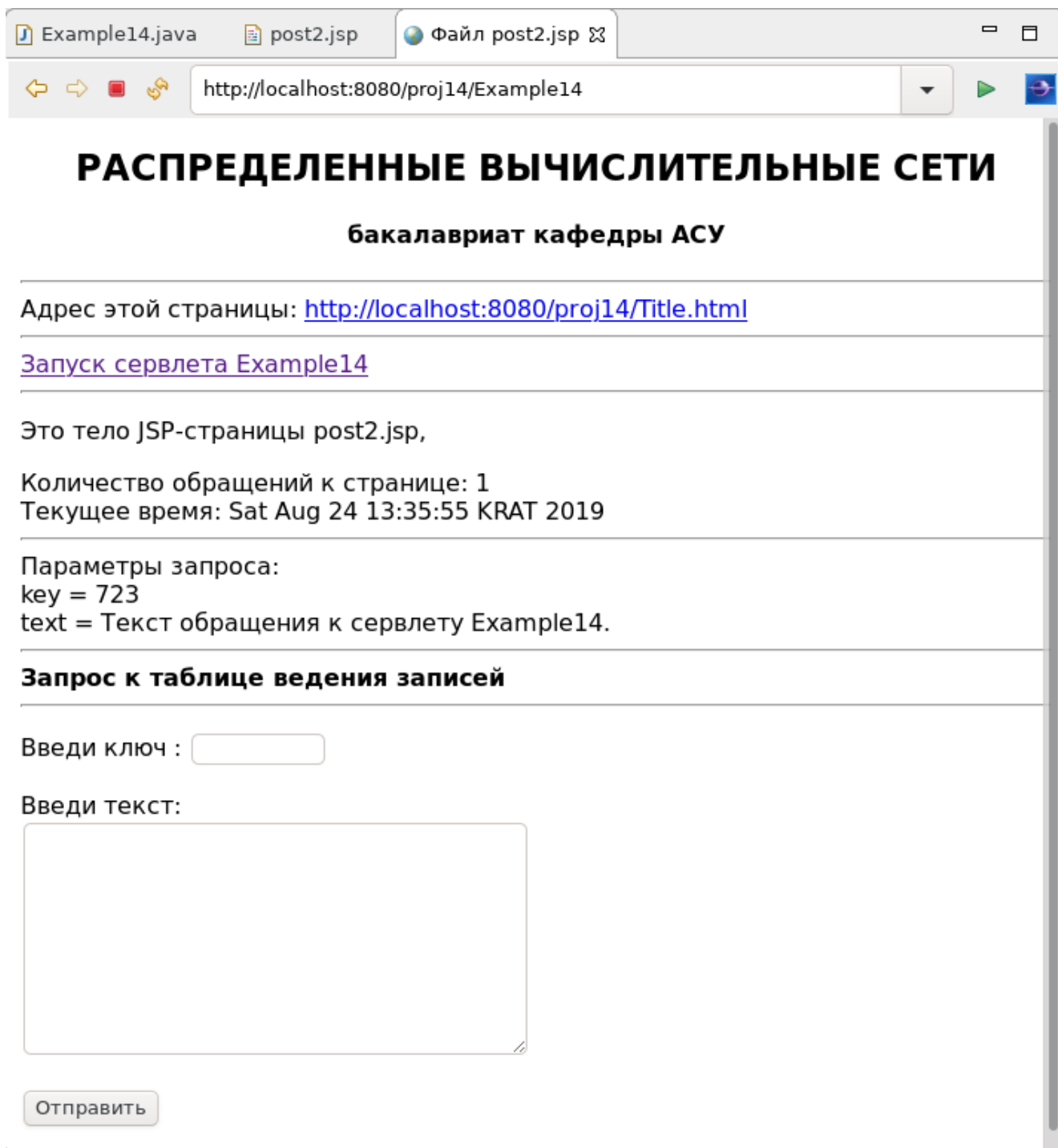


Рисунок 4.25 — Ответ сервлета методом doPost(...)

4.3.5 Модель MVC

Практическая реализация любой РВ-сети начинается с создания некоторой первичной схемы проекта, которая бы создавала основу для дальнейшего функционального наполнения системы. Такая схема должна создаваться на основе принципов, обеспечивающих простоту и надёжность последовательного процесса реализации окончательного варианта системы.

Теоретическим универсальным принципом разработки сложных систем является подход, предполагающий разделение ее на ряд специализированных подсистем, удовлетворяющих следующим двум признакам:

- а) большая функциональная часть таких подсистем может реализовываться и развиваться независимо от других частей;
- б) имеется простое и понятное взаимодействие между подсистемами.

Таким признакам удовлетворяет проектная схема MVC, которую мы и рассмотрим в данном пункте, связав её с технологией сервлетов:

- а) **MVC** [*Model View Controller*] - это *шаблон проектирования*, который впервые был опубликован в 1970-х годах. Он представляет собой образец архитектуры программного обеспечения, основанный на *принципе разделения представления данных и функционала*, где эти данные формируются и обрабатываются.
- б) **MVC** — это *аббревиатура*, состоящая из трёх слов:
 - а) - *модель* [Model];
 - б) - *вид* (представление) [View];
 - в) - *контроллер* [Controller].

Модель — *некое хранилище* (поставщик) *данных* в рамках всего проекта. Главные задачи модели заключаются в представлении доступа к данным для их просмотра или актуализации (добавления, редактирования, удаления).

Представление — *это компонент MVC*, где выполняется вывод данных на экран. При классическом подходе к web-разработкам в представлении будет формироваться HTML код.

Контроллер (контроллеры) — *это компонент MVC*, предназначенный для связи между моделями и представлениями, а также для обработки данных, которые пришли от пользователя к серверу через формы запроса и другие источники. После того, как контроллер получил информацию, в зависимости от необходимости задачи, он передаст данные в представление для вывода или в модель для актуализации (добавления, редактирования, удаления).

Взаимодействие работы перечисленных компонент, данного шаблона, можно представить рисунком 4.26, на котором каждая из компонент реализуется средствами сервера Apache Tomcat:

- а) **модель** (model) – некоторое приложение, поставляющее данные в сервлет;
- б) **представление** (view) – набор JSP-страниц, подготавливающий для сервлета ответ клиенту (браузеру);
- в) **контроллер** (controller) – сервлет, который организует доступ к приложению модели, принимает от него данные, передаёт на подготовку HTML-страниц в службу представления и возвращает результат клиенту (браузеру).

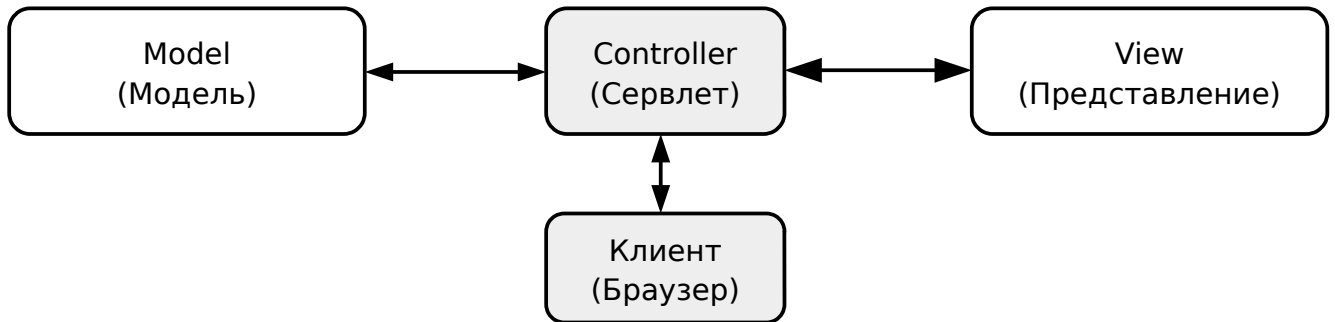


Рисунок 4.26 — Трёхзвенная архитектура, реализуемая моделью MVC

Представленная трёхзвенная архитектура РВ-сети легко может быть реализована в нашем проекте. Для этого воспользуемся, например, классом *NotePad*, который находится в JAR-архиве *\$HOME/lib/notepad.jar*, а чтобы наш сервлет имел доступ к этому архиву, мы:

- а) выделим мышкой проект *proj14*;
- б) правой кнопкой мышки активируем меню и выберем *Build Path*→*Configure Build Path*;
- в) в появившемся окне добавим внешний JAR-архив, как это показано на рисунке 4.27.

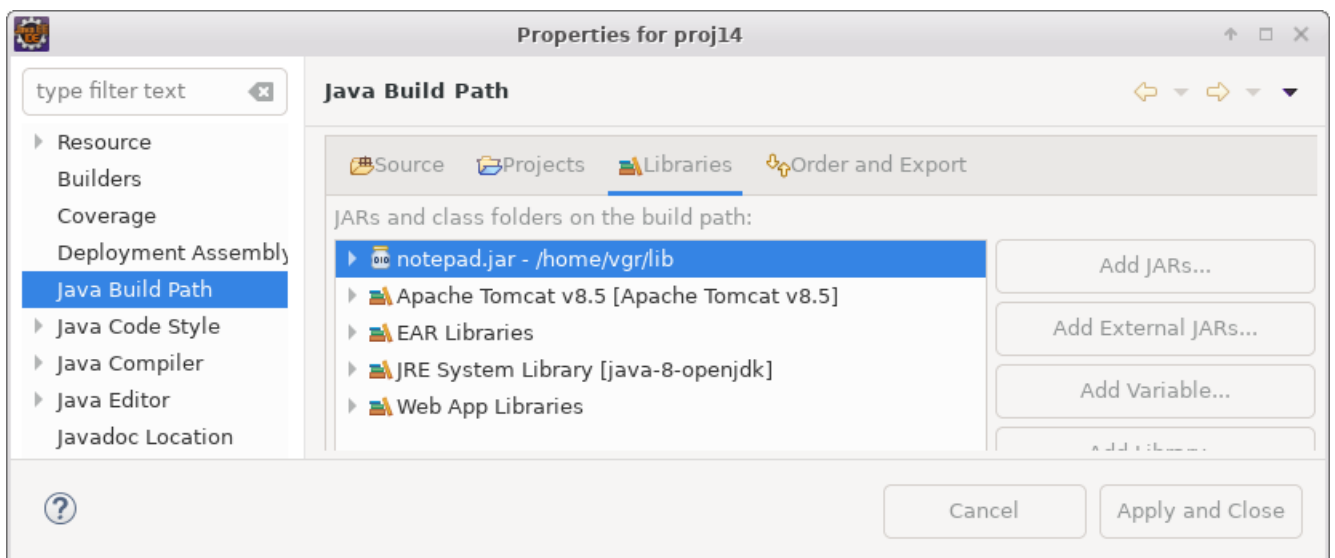


Рисунок 4.27 — Подключение архива notepad.jar к проекту proj14

Дополнительно, необходимо подключить архив *\$HOME/lib/notepad.jar* к серверу Apache Tomcat, иначе он не сможет стартовать по причине невозможности загрузить класс *NotePad*. Для этого необходимо:

- а) в Project Explorer выделить используемый сервер;
- б) из главного меню Eclipse EE выбрать **Run**→**Run Configurations...**;
- в) в появившемся окне (см. рисунок 4.28), на вкладке **Classpath** подключить внешний архив *notepad.jar*.

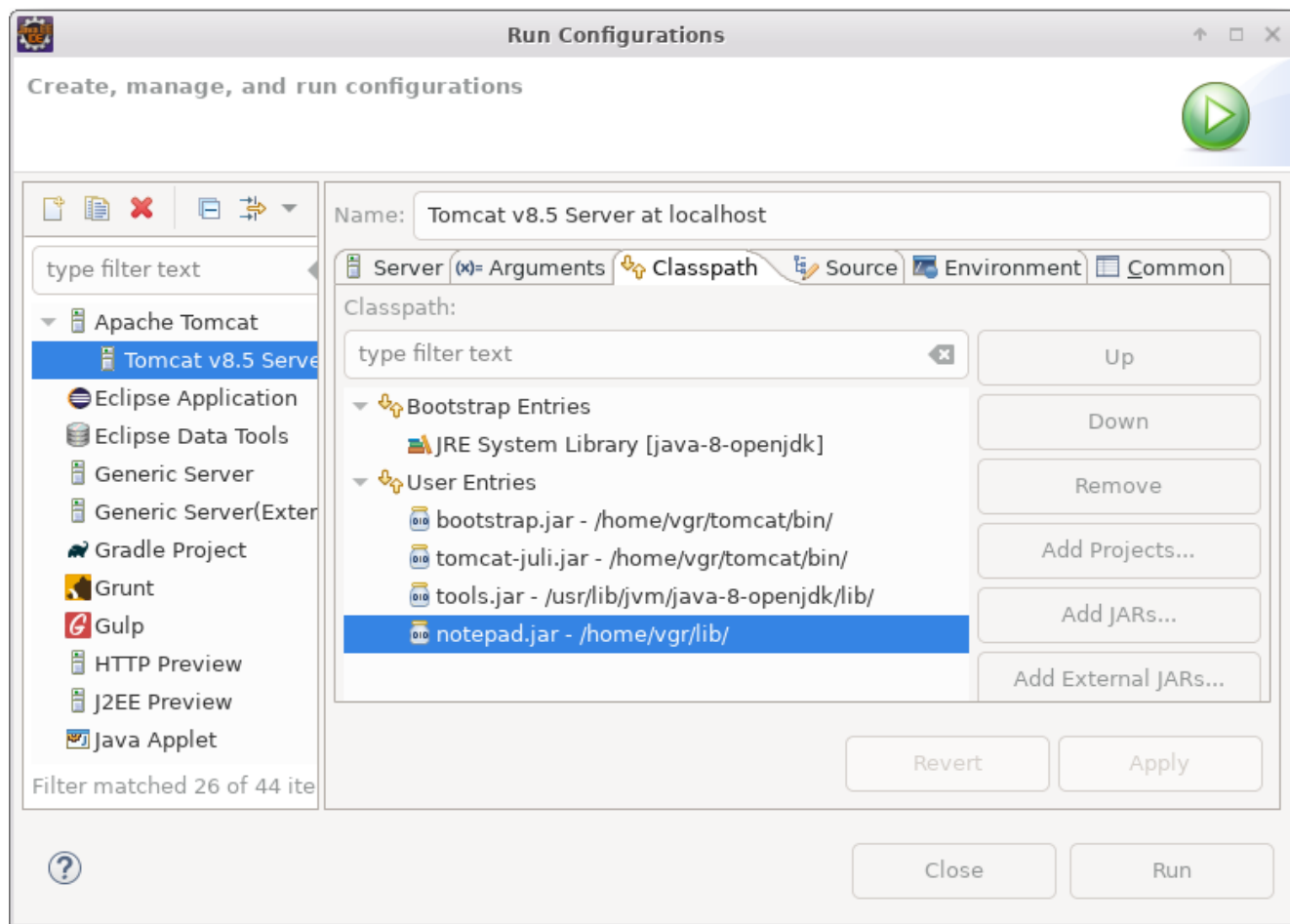


Рисунок 4.28 — Подключение архива notepad.jar к среде запуска сервера Apache Tomcat

Теперь все готово для реализации проекта по схеме трёхзвенной РВ-сети. Исключительно для целей демонстрации, ограничим объем реализации нашего примера только чтением, с помощью класса *NotePad*, содержимого таблицы *notepad* базы данных *exampleDB* и предоставлением его пользователю в виде списка отдельных записей.

Действуя по схеме MVC, сначала обеспечим доступ к модели, что реализуется с помощью следующих статических конструкций языка Java, добавляемых в начало сервлета *Example14*:

- а) `NotePad notepad` — ссылка на объект, реализующий доступ к БД;
- б) `boolean isOpen` — статус состояния класса `NotePad`;
- в) `boolean isError` — статус ошибки создания объекта класса `NotePad`;
- г) `boolean dbOpen()` — метод создания объекта класса `NotePad` и контроля его статуса, который возвращает значение ***true***, если дальнейшие действия с БД являются допустимыми;
- д) `void dbClose()` — закрывает объект класса ***NotePad*** и устанавливает статусы доступа, что — необходимо, если будет переопределяться метод сервлета ***destroy()***;
- е) `String[] dbList()` — метод получения списка записей БД.

Использование указанных статических конструкций обосновано тем, что запросы могут осуществляться многими пользователями параллельно, а открытый объект ***notepad*** должен быть один. Указанные объекты добавлены в начало сервлета ***Example14*** и представлены на листинге 4.8.

Листинг 4.8 — Новые статические конструкции сервлета ***Example14***

```
/**
 * Servlet implementation class Example14
 */
@WebServlet("/Example14")
public class Example14 extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * Объекты доступа к классу NotePad
     */
    private static NotePad notepad = null;
    private static boolean isOpen = false;
    private static boolean isError = false;

    /**
     * Открытие объекта NotePad.
     * @return
     */
    protected static boolean dbOpen()
    {
        if(isError)
            return false;
        if(!isOpen)
            notepad = new NotePad();
        if(notepad == null)
        {
            isError = true;
            System.out.println("Не могу открыть NotePad");
            return false;
        }else
        {
            isOpen = notepad.isConnected();
        }
    }
}
```

```

        if(isOpen)
            System.out.println("NotePad - открыта!");
        else
            System.out.println("NotePad - не открыта!");

        return isOpen;
    }
}

/**
 * Закрытие объекта NotePad.
 */
protected static void dbClose()
{
    if(isError || !isOpen)
        return;
    notepad.setClose();
    isOpen = false;
    notepad = null;
}

/**
 * Получение списка записей объекта Notepad.
 * @return String[]
 */
protected static String[] dbList()
{
    if(!dbOpen())
        return null;

    Object[] obj = notepad.getList();

    if(obj == null)
    {
        System.out.println("getList() вернул null");
        return null;
    }
    int ns =
        obj.length;
    System.out.println("Получено " + ns + " строк(и)");
    String[] ss =
        new String[ns];

    for(int i=0; i < ns; i++)
        ss[i] = obj[i].toString();

    return ss;
}
// Далее идут стандартные методы класса Example14.

```

Теперь перейдём к реализации проектной части представления. Для этого учтём, что методы запроса *request* позволяют взаимодействовать сервлету и JSP-странице посредством передачи значений атрибутов:

- а) request.setAttribute(String name, Type value) — устанавливает значение атрибута с именем *name*;
- б) request.getAttribute(String name) — читает значение атрибута с именем *name*.

Новое представление для метода *doPost(...)* создадим в виде JSP-страницы *post3.jsp*, показанной на листинге 4.9.

Листинг 4.9 — Исходный текст post3.jsp сервлета Example14

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Страница post3.jsp</title>
</head>
<body>
    <!-- Первое действие include -->
    <jsp:include page="/Title.html" />
    <p>
        Список сообщений таблицы notepad: <br>
    </p>
    <ul>

    <!-- Объявление -->
    <%! private int ns = 0;%>

    <!-- Скриплет -->
    <%
        String[] ss =
            (String[])request.getAttribute("list");
        if(ss == null)
            out.println("<li>Нет данных, возможно - ошибка БД!</li>");
        else
        {
            ns = ss.length;

            for(int i=0; i<ns; i++)
                out.println("<li>" + ss[i] + "</li>");
        }
    %>
    </ul><hr>

    Получено <%=ns %> строк(и)<br>

    <!-- Второе действие include -->
    <jsp:include page="post1.html" />

</body>
</html>
```

Завершаем реализацию данного примера посредством изменения метода *doPost(...)*, показанного на листинге 4.10.

Листинг 4.10 — Изменённый метод doPost(...) сервлета Example14

```
/**
 * @see HttpServlet#doPost(HttpServletRequest request,
```

```

    * HttpServletResponse response)
    */
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
                          throws ServletException, IOException {

        System.out.println("Вызывается метод doPost(...)");
        /**
         * Явная установка кодировок объектов запроса и ответа.
         * Стандартная установка контекста ответа.
         */
        request.setCharacterEncoding("UTF-8");
        response.setCharacterEncoding("UTF-8");
        response.setContentType("text/html");
        /**
         * Обращение к модели и установка результата
         * в виде атрибута к JSP-странице:
         */
        request.setAttribute("list", dbList());
        /**
         * Далее, на основе анализа request должны
         * реализовываться методы добавления и
         * удаления строк таблицы notepad БД exampleDB.
         */
        /**
         * Стандартное подключение ресурса сервлета.
         */
        RequestDispatcher disp =
            request.getRequestDispatcher("/WEB-INF/post3.jsp");
        disp.forward(request, response);
    }

```

Начальный результат запуска изменённого сервлета ***Example14*** соответствует форме запроса приведённого ранее на рисунке 4.24, а после отправки сообщения, браузер покажет список записей таблицы ***notepad***, подобно рисунку 4.29.

Example14.java Страница post3.jsp

http://localhost:8080/proj14/Example14

РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СЕТИ

бакалавриат кафедры АСУ

Адрес этой страницы: <http://localhost:8080/proj14/Title.html>

[Запуск сервлета Example14](#)

Список сообщений таблицы notepad:

- 124 Разработал программу для технологии RMI.
- 125 Исправил исходные тексты.
- 126 Стартовал RMI-сервер из архива ~/lib/rmipadserver.jar.

Получено 3 строк(и)

Запрос к таблице ведения записей

Введи ключ :

Введи текст:

Рисунок 4.29 — Форма отображения с помощью JSP-страницы post3.jsp

Таким образом, мы завершили заявленный объем реализации приложения с трёхзвенной архитектурой РВ-сети, которое естественным образом может быть развито для поддержки остальных функций класса *NotePad*: добавление и удаление записей таблицы *notepad* базы данных *exampleDB*. И, в плане этой реализации, мы ограничимся только следующими замечаниями:

- а) для использования объекта класса *NotePad*, в сервлет *Example14* были включены дополнительные статические объекты, которые необходимы для синхронизации обращений многих клиентов к встроенной базе данных; принципиально, можно было бы создать новый класс, решающий как задачу синхронизации, так и предоставляющий нужные методы без «загромождения» текста самого сервлета; такой класс должен быть помещён в поддерево каталога *WEB-INF*, чтобы он был недоступен для прямого вызова программой клиента;

- б) представление *post1.jsp* метода *doGet(...)* преследует чисто учебные цели, поэтому его можно и нужно заменить на более информативное содержание, описывающее и рекламирующее само распределенное приложение; в любом случае, оно должно содержать форму, хотя бы в виде одной кнопки, переводящей запросы клиента на метод *doPost(...)*;
- в) полная реализация метода *doPost(...)* предполагает чтение параметров запроса и их анализ, и, как следствие, это порождает различные обращения к объекту класса *NotePad* и вызов различных JSP-страниц, соответствующих протоколу диалога клиента и сервлета.

Рассмотренным примером завершается учебный материал главы, посвящённой web-технологиям распределенных систем. Наряду с общим обзором, включающим мотивы её появления и тенденции развития, нами были изучены и подкреплены конкретными примерами: модели развития базовой архитектуры «Клиент-сервер», а также мощная и перспективная технология Java-сервлетов.

В процессе своего развития, web-технологии всегда претендовали на нечто большее, чем простой информационный обмен HTML-страницами. Технология Java-сервлетов наглядно продемонстрировала возможность создания распределённых РВ-сетей уровня предприятия с использованием «Тонкого клиента». И хотя язык Java представляет лишь часть инструментальных средств создания РВ-сетей, он занимает достойное место среди других и может быть более модных, инструментальных технологий.

В заключении данной главы отметим, что рассмотренные нами инструменты создания РВ-сетей лишь актуализировали проблему адресации все возрастающего объёма распределенных приложений. Теоретические и практические подходы к решению этой проблемы рассмотрим в следующей главе.

Вопросы для самопроверки

1. Что такое — URI и его формы представления?
2. Чем отличается URI от URL?
3. В чем состоит назначение языка HTML и его отличие от языка XML?
4. Что такое - «Тонкий клиент» и как это понятие связано с АРМ?
5. Когда были сформированы стандарты на технологию World Wide Web?
6. Какой уровень стека протоколов TCP/IP описывает протокол HTTP?
7. Объясните назначение и соотношение терминов JavaScript, Applet и Servlet?
8. Какие методы запросов клиента к серверу обеспечивает протокол HTTP?
9. Какие три уровня предлагает вертикальная модель распределения классической архитектуры «Клиент-сервер»?
10. В чем состоит назначение горизонтальной модели распределения классической архитектуры «Клиент-сервер»?
11. Что такое — двухзвенная архитектура «Клиент-сервер»?
12. Что такое — трёхзвенная архитектура «Клиент-сервер» и в чем состоит ее отличие от двухзвенной архитектуры?
13. Что такое — технология CGI?
14. В чем состоит назначение Apache Tomcat?
15. Что такое Servlet и какой абстрактный класс он порождает?
16. Какой пакет Java содержит ПО сервлетов?
17. Какие два основных метода запросов формирует браузер к web-серверу?
18. Какие два базовых метода использует класс HttpServlet?
19. Что такое — JSP-страница и для чего она предназначена?
20. Что делает сервер с JSP-страницей, когда она вызывается сервлетом?