JSP

1. Понятие о JSP

Сервлеты позволяют получать запросы от клиента, совершать некоторую работу и выводить результаты на экран. Сервлет работает до того момента, пока речь идет об обработке информации, т.е. до вывода информации на экран. В сервлет можно вставить достаточно сложную логику, сделать вызовы к базе данных и многое-многое другое, что необходимо для приложения. Но вот осуществлять вывод на экран внутри самого сервлета - очень неудобно.

Технология проектирования Java Server Pages (JSP) - это одна из технологий JavaEE, которая представляет собой расширение технологии сервлетов для упрощения работы с Web-содержимым. Страницы JSP позволяет легко разделить Web-содержимое на статическую и динамическую часть, допускающую многократное использование ранее определенных компонентов. Разработчики Java Server Pages могут использовать компоненты JavaBeans и создавать собственные библиотеки нестандартных тегов, которые инкапсулируют сложные динамические функциональные средства.

Во многих случаях сервлеты и JSP-страницы являются взаимозамеяемыми. Подобно сервлетам, JSP-страницы обычно выполняются на стороне Web-сервера, который называют контейнером JSP.

Когда Web-сервер, поддерживающий технологию JSP, принимает первый запрос на JSP-страницу, контейнер JSP транслирует эту JSP-страницу в сервлет Java, который обслуживает текущий запрос и все последующие запросы к этой странице. Если при компиляции нового сервлета возникают ошибки, эти ошибки приводят к ошибкам на этапе компиляции. Контейнер JSP на этапе трансляции помещает операторы Java, которые реализует ответ JSP-страницы, в метод _jspService. Если сервлет компилируется без ошибок, контейнер JSP вызывает метод _jspService для обработки запроса.

JSP-страница может обработать запрос непосредственно, либо вызвать другие компоненты Web-приложения, чтобы содействовать обработке запроса. Любые ошибки, которые возникают в процессе обработки, вызывают исключительную ситуацию в Web-сервере на этапе запроса.

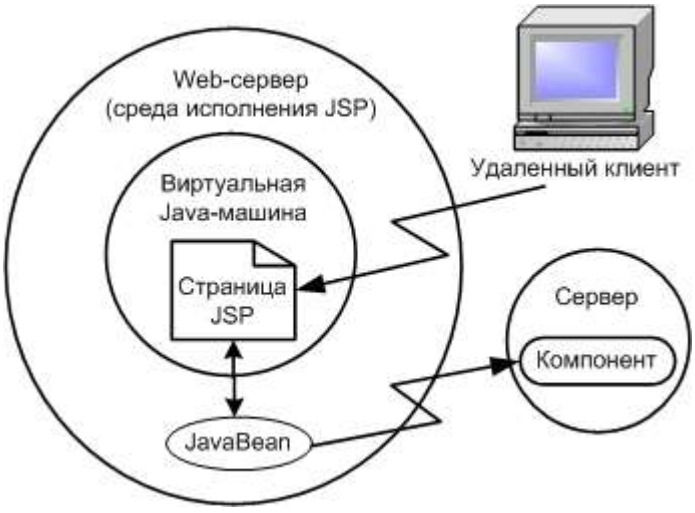
Весь статический текст HTML, называемый в документации JSP шаблоном HTML (template HTML), сразу направляется в выходной поток. Выходной поток страницы буферизуется. Буферизацию обеспечивает класс JspWriter, расширяющий класс Writer.

Таким образом, достаточно написать страницу JSP, сохранить ее в файле с расширением jsp и установить файл в контейнер, так же, как и страницу HTML, не заботясь о компиляции. При установке можно задать начальные параметры страницы JSP так же, как и начальные параметры сервлета.

Архитектура JSP-страницы

Базовая архитектура Java Server Pages в самом общем виде представлена на рисунке. Страница JSP располагается на Web-сервере в среде виртуальной Java-машины. Доступ к страниц JSP, как и в случае сервлета, осуществляется через Web с использованием протокола HTTP.

Страница JSP функционирует под управлением JSP Engine (среды исполнения JSP). Страница JSP может взаимодействовать



с программным окружением с помощью компонентов JavaBeans, получая и устанавливая его параметры, используя теги:<*jsp:useBean*>, <*jsp:getProperty*>, <*jsp:setProperty*>. Компонент JavaBean сам может участвовать в других процессах, предоставляя результаты в виде своих параметров, доступных страницам JSP, участвующим в сеасе, а через них - всем пользователям, запрашивающим эти страницы JSP.

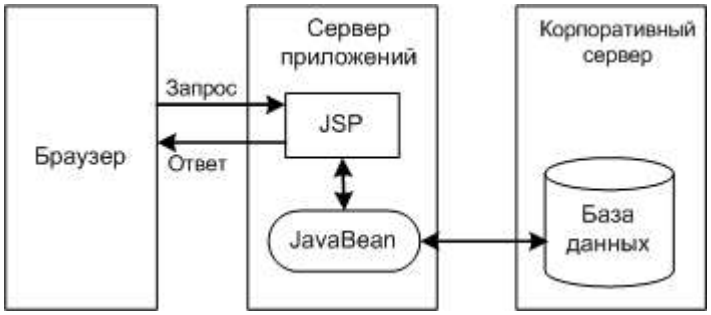
Основные модели архитектуры JSP

Возможны различные подходы к использованию технологии JSP. Два основных архитектурных подхода, нашедшие применение при реализации приложений уровня предприятия, имеют специальные названия:

- JSP Model 1 (Первая модель архитектуры JSP);
- JSP Model 2 (Вторая модель архитектуры JSP);

Первая модель архитектуры JSP

JSP Model 1 (Первая модель архитектуры JSP) практически реализует базовую архитектуру JSP. В архитектурном решении JSP Model 1 полностью отвечает за получение запроса от клиента, его обработку, подготовку ответа и доставку ответа пользователю. Разделение представления и динамического содержания обеспечивается тем, что доступ к данным осуществляется через компоненты JavaBeans. В сценарии *JSP Model 1* предполагается следующая последовательность действий:

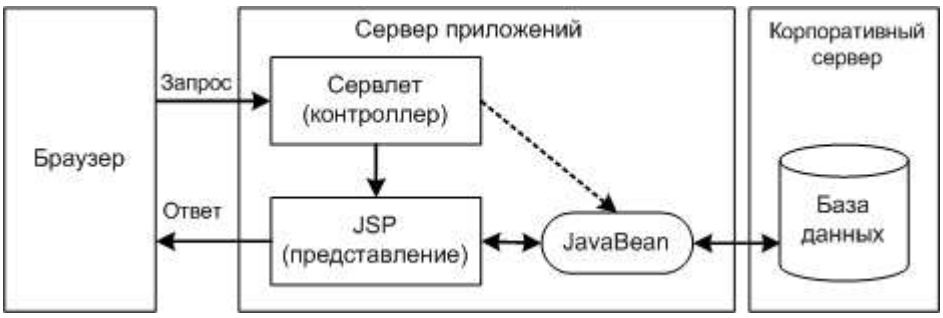


1. Запрос пользователя посылается через Web-браузер странице JSP.
2. Страница JSP компилируется в сервлет (*при первом обращении*).
3. Скомпилированный сервлет обращается к некоторому компоненту JavaBean, запрашивая у него информацию.
4. Компонент JavaBean, в свою очередь, осуществляет доступ к информационным ресурсам (непосредственно или через компонент Enterprise JavaBeans).
5. Полученная информация отображается в свойствах компонента JavaBeans, доступных странице JSP.
6. Формируется ответ в виде страницы HTML с комбинированным содержанием (статическое, динамическое).

Архитектура *JSP Model 1* может с успехом применяться для небольших приложений. Однако использование данной модели для более сложных задач вызывает определенные трудности и не является технологичным из-за большого объема встроенных в страницу программных фрагментов. Для сложных корпоративных приложений рекомендуется применение второй модели архитектуры JSP.

Вторая модель архитектуры JSP

JSP Model 2 (Вторая модель архитектуры JSP) реализует гибридный подход к обслуживанию динамического содержания Web-страницы, при котором совместно используется сервлет и страница JSP. Эта модель позволяет эффективно использовать преимущества обеих технологий: сервлет поддерживает задачи, связанные с обработкой запроса и созданием объектов JavaBeans, используемых JSP, а страница JSP отвечает за визуальное представление информации. Сервлет используется как управляющее устройство (контроллер). Схематично вторая модель представлена на рисунке. Сценарии *JSP Model 2*, как правило реализует следующую типовую последовательность действий:



1. Запрос пользователя посылается через Web-браузер сервлету.
2. Сервлет обрабатывает запрос, создает и инициализирует объект JavaBean или другие объекты, используемые страницей JSP, и запрашивает динамическое содержание у компонента JavaBean.
3. Компонент JavaBean осуществляет доступ к информации непосредственно или через компонент Enterprise JavaBeans.
4. Сервлет, направляющий запрос, вызывает сервлет, скомпилированный из страницы JSP.
5. Сервлет, скомпилированный из страницы JSP, встраивает динамическое содержание в статический контекст HTML-страницы и отправляет ответ пользователю.

Необходимо отметить, что в рамках этой модели страница JSP сама не реализует никакую логику, это входит в функции сервлета-контроллера. Страница JSP отвечает только за получение информации от компонента JavaBean, который был предварительно создан сервлетом, и за визуальное представление этой информации в удобном для клиента виде.

Архитектуры *JSP Model 2* в большей степени, чем архитектура *JSP Model 1*, соответствует идее отделения представления от содержания. Эта модель позволяет четко выделить отдельные части приложения и связанные с ними роли и обязанности персонала, занятого в разработке:

Чем сложнее разрабатываемая система, тем заметнее становятся преимущества архитектуры *JSP Model 2*.

Функционирование JSP

Работа со страницей JSP становится возможной только после ее преобразования в сервлет. В процессе трансляции как статическая, так и динамическая части JSP преобразуются в Java-код сервлета, который передает преобразованное содержимое браузеру через выходной поток Web-сервера.

Технология JSP является технологией серверной стороны, поэтому все процессы обработки JSP протекают на стороне сервера. Страница JSP - текстовый документ, который в соответствии со спецификацией JSP, проходит две фазы:

- фазу трансляции;
- фазу выполнения.

При трансляции, которая выполняется один раз для каждой страницы JSP, создается или локализуется класс типа *Servlet*, реализующий JSP. Трансляция JSP может производиться как

отдельно, до ее использования, так и в процессе размещения JSP на Web-сервере или сервере приложений.
Во второй фазе осуществляется обработка запросов и подготовка ответов.

Синтаксис JSP-страницы

Страницы JSP имеют комбинированный синтаксис: объединение стандартного синтаксиса, соответствующего спецификации HTML, и синтаксиса JSP, определенного спецификацией Java Server Pages. Синтаксис JSP определяет правила записи страниц JSP, состоящих из стандартных тегов HTML и тегов JSP.

Страницы JSP, кроме HTML-тегов, содержат теги JSP следующих категорий:

- директивы (directives);
  - page;
  - taglib;
  - include;
- объявления (declarations);
- скриптлеты (scriptlets);
- выражения (expressions);
- комментарии (comments);

Директивы JSP

Директивы обеспечивают глобальную информацию, касающихся конкретных запросов, направляемых в JSP, и предоставляют информацию, необходимую на стадии трансляции.

Директивы всегда помещаются в начале JSP-страницы до всех остальных тегов, чтобы *parser* (анализатор) JSP при разборе текста в самом начале выделил глобальные инструкции. Таким, образом, JSP Engine (среда исполнения JSP), анализируя код, создает из JSP сервлет. *Директивы* представляют собой сообщения контейнеру JSP.

Синтаксис *директив* JSP выглядит следующим образом:

```
<%@ директива имяАтрибута="значение" %>
```

Синтаксис задания *директив* на XML:

```
<jsp:directive.директива имяАтрибута="значение" />
```

Директива может иметь несколько атрибутов. В этом случае *директива* может быть повторена для каждого из атрибутов. В то же время пары *"имяАтрибута=значение"* могут располагаться под одной директивой с пробелом в качестве разделителя.

Существует три типа директив:

- page (страница)
- taglib (библиотека тегов)
- include (включить)

Директива page

Директива *page* определяет свойства страницы JSP, которые воздействуют на транслятор. Порядок следования атрибутов в директиве *page* не имеет значения. Нарушение синтаксиса или наличие нераспознанных атрибутов приводит к ошибке трансляции. Примером директивы *page* может служить следующий код:

```
<%@ page buffer="none" isThreadSafe="yes" errorPage="/error.jsp" %>
```

Эта директива объявляет, что данная страница JSP не использует буферизацию, что возможно одновременное обращение к данной странице JSP многих пользователей, и что поддерживается страница ошибок с именем *error.jsp*. Директива *page* может содержать информацию о странице:

```
<%@ page info = "JSP Sample 1" %>
```

Директива taglib

Директива *taglib* объявляет, что данная страница JSP использует библиотеку тегов, уникальным образом идентифицируя ее с помощью URI, и ставит в соответствие префикс тега, с помощью которого возможны действия в библиотеке. Если контейнер не может найти библиотеку тегов, возникает фатальная ошибка трансляции.

Директива *taglib* имеет следующий синтаксис:

```
<%@ taglib uri="URI включаемой библиотеки тегов" prefix="имяПрефикса" %>
```

Префикс "*имяПрефикса*" используется при обращении к библиотеке. Пример использования библиотеки тегов *mytags*:

```
<%@ taglib uri="http://www.taglib/mytags" prefix="customs" %>
. . .
<customs:myTag>
```

В данном примере библиотека тегов имеет URI-адрес *"http://www.taglib/mytags"*, в качестве префикса назначена строка *customs*, которая используется в странице JSP при обращении к элементам библиотеки тегов.

Директива include

Директива *include* позволяет вставлять текст или код в процессе трансляции страницы JSP в сервлет. Синтаксис директивы *include* имеет следующий вид:

```
<%@ include file="Относительный URI включаемой страницы" %>
```

Директива *include* имеет один атрибут - *file*. Она включает текст специфицированного ресурса в файл JSP. Эту директиву можно использовать для размещения стандартного заголовка об авторских правах на каждой странице JSP:

```
<%@ include file="copyright.html" %>
```

Контейнер JSP получает доступ к включаемому файлу. Если включаемый файл изменился, контейнер может перекомпилировать страницу JSP. Директива *include* рассматривает ресурс, например, страницу JSP, как статический объект.

Заданный URI обычно интерпретируется относительно JSP страницы, на которой расположена ссылка, но, как и при использовании любых других относительных URI, можно задать системе положение интересующего ресурса относительно домашнего каталога WEB-сервера добавлением в начало URI символа "/". Содержимое подключаемого файла обрабатывается как обычный текст JSP и поэтому может включать такие элементы, как статический HTML, элементы скриптов, директивы и действия.

Многие сайты используют небольшую панель навигации на каждой странице. В связи с проблемами использования фреймов HTML часто эта задача решается размещением небольшой

таблицы сверху или в левой половине страницы, HTML код которой многократно повторяется для каждой страницы сайта. Директива *include* - это наиболее естественный способ решения данной задачи, избавляющий разработчика от кошмара рутины копирования HTML в каждый отдельный файл.

Поскольку директива *include* подключает файлы в ходе трансляции страницы, то после внесения изменений в панель навигации требуется повторная трансляция всех использующих ее JSP страниц. Если же подключенные файлы меняются довольно часто, можно использовать действие *jsp:include*, которое подключает файл в процессе обращения к JSP странице.

Объявления JSP

Declarations (Declarations) предназначены для определения переменных и методов на языке скриптов, которые в дальнейшем используются на странице JSP. Синтаксис *declarations* имеет следующий вид :

```
<%! код Java %>
```

Объявления располагаются в блоке объявлений, а вызываются в блоке выражений страницы JSP. Код в блоке объявлений обычно пишется на языке Java, однако серверы приложений могут использовать синтаксис и других скриптов. *Объявления* иногда используются для того, чтобы добавить дополнительную функциональность при работе с динамическими данными, получаемыми из свойств компонентов JavaBeans. Примеры *объявлений* представлены в таблице.

<%! int i ; %>	Объявление глобальной целочисленной переменной *i*.
<%! String s = "Hello, World!"; %>	Объявление и инициализация глобальной строковой переменной *s*.
<%! int n = 100; %>	Объявление и инициализация глобальной целочисленной переменной *n*.
<%! public int adding (int a, int b){return a + b}; %>	Объявление метода *adding* сложения двух целочисленных значений, глобального для всей страницы JSP

Объявление может содержать несколько строк, как например, в приведенном ниже коде вычисления значения функции *fact (int n)*, которая должна быть равна 1 при n меньше 2 и n! при положительном значении n;

```
<%!
public static int fact (int n)
{
  if (n <= 1)
     return 1;
  else
     return n * fact (n - 1);
}
%>
```

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

Скриптлеты JSP

Скриптлеты включают различные фрагменты кода, написанного на языке скрипта, определенного в директиве *language*. Фрагменты кода должны соответствовать синтаксическим конструкциям языка *скриптлетов*, т.е., как правило, синтаксису языка Java. *Скриптлеты* имеют следующий синтаксис:

```
<% текст скриптлета %>
```

Эквивалентом синтаксиса *скриптлета* для XML является:

```
<jsp:scriptlet> текст скриптлета </jsp:scriptlet>
```

Если в тексте *скриптлета* необходимо использовать последовательность символов %> именно как сочетание символов, а не как тег - признак окончания *скриптлета*, вместо последовательности %> следует использовать следующее сочетание символов %\>.

В спецификации JSP приводится простой и понятный пример *скриптлета*, обеспечивающего динамическое изменение содержимого страницы JSP в течение дня.

```
<% if (Calendar.getInstance ().get (Calendar.AM_PM) == Calendar.AM) {%>
  Good Morning
<% } else { %>
  Good Afternoon
<% } %>
```

Необходимо заметить, что код внутри *скриплета* вставляется в том виде, как он записан, и весь статический HTML-текст (текст шаблона) до или после *скриплета* конвертируется при помощи оператора *print*. Это означает что скриплеты не обязательно должны содержать завершенные фрагменты на Java, и что оставленные открытыми блоки могут оказать влияние на статический HTML-текст вне *скриплета*.

Скриплеты имеют доступ к тем же автоматически определенным переменным, что и выражения. Поэтому, например, если есть необходимость вывести какую-либо информацию на страницу, необходимо воспользоваться переменной *out*.

```
<%
   String queryData = request.getQueryString ();
   out.println ("Дополнительные данные запроса: " + queryData);
%>
```

Выражения JSP

Выражение в странице JSP - это исполняемое выражение, написанное на языке скрипта, указанного в объявлении *language* (как правило Java). Результат *выражения* JSP, имеющий обязательный тип *String*, направляется в стандартный поток вывода *out* с помощью текущего объекта *JspWriter*. Если результат *выражения* не может быть приведен к типу *String*, возникает либо ошибка трансляции, если проблема была выявлена на этапе трансляции, либо возбуждается исключение *ClassCastException*, если несоответствие было выявлено в процессе выполнения запроса. *Выражение* имеет следующий синтаксис:

```
<%= текст выражения %>
```

альтернативный синтаксис для *выражений* JSP при использовании XML:

```
<jsp:expression> текст выражения </jsp:expression>
```

Порядок выполнения *выражений* в странице JSP слева-направо. Если *выражение* появляется более чем в одном атрибуте времени выполнения, то оно выполняется слева-направо в данном теге. *Выражение* должно быть полным выражением на определенном скрипте (как правило Java).

Выражения выполняются во время работы протокола HTTP. Значение выражения преобразуется в строку и включается в соответствующую позицию файла JSP.

Выражения обычно используются для того, чтобы вычислить и вывести на экран строковое представление переменных и методов, определенных в блоке объявлений страницы JSP или полученных от компонентов JavaBeans, которые доступны из JSP. Следующий код *выражения* служит для отображения даты и времени запроса данной страницы:

```
Текущее время: <%= new java.util.Date () %>
```

Для того чтобы упростить *выражения* существует несколько заранее определенных переменных, которые можно использовать. Наиболее часто используемые переменные:

- request, HttpServletRequest;
- response, HttpServletResponse;
- session, HttpSession - ассоциируется с запросом , если таковой имеется;
- out, PrintWriter - буферизированный вариант типа JspWriter для отсылки данных клиенту.

Пример:

```
<%@page import="by.iba.Calculator"%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>JSP Application</title>
</head>
<body>
<h2>Square of 3 = <%= new Calculator().square(3) %></h2>
</body>
</html>
```

Комментарии JSP

В страница JSP можно использовать два типа комментариев:

1. Выводимый комментарий (output comment).
2. Закомментированный блок (commented block).

Выводимый комментарий - это такой стиль комментирования, при котором комментарий выводится в выходной поток и отображается в браузере. Синтаксис комментариев данного типа следующий:

```
<!-- Текст комментария -->
```

Можно также использовать комментарии с динамическим содержимым. Для этого внутрь закомментированного блока должен быть встроен тег скриптлета, например:

```
<!-- Начало комментария <%= expression %> продолжение комментария -->
```

Закомментированный блок - это стиль комментирования, при котором тело закомментированного блока полностью игнорируется, не выводится в выходной поток и, следовательно, не отображается в браузере. Этот стиль обычно используется для поясняющих записей, касающихся фрагментов программного кода страницы JSP.Закомментированный блок имеет следующий синтаксис:

```
<%-- Текст комментария --%>
```

Неявные объекты

Неявные объекты (implicit objects) - это объекты, автоматически доступные как часть стандарта JSP без их специального объявления или импорта. Эти объекты, список которых представлен в таблице, можно использовать в коде JSP.

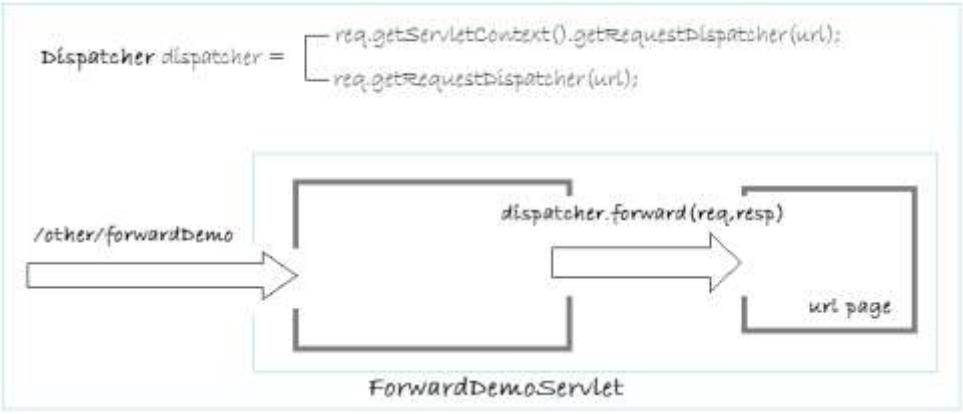
Наименование объекта	**Тип объекта**	**Назначение**
request (запрос)	javax.servlet.HttpServletRequest	Запрос, требующий обслуживания. Область видимости - запрос. Основные методы : *getAttribute, getParameter, getParameterNames, getParameterValues* Таким образом, запрос *request* обеспечивает обращение к параметрам запроса через метод *getParameter*, типу запроса (GET, POST, HEAD, и т.д.), и входящим HTTP заголовкам (cookies, Referer и т.д.).
response (ответ)	javax.servlet.HttpServletResponse	Ответ на запрос. Область видимости - страница. Поскольку поток вывода (см. out далее) буферизован, можно изменять коды состояния HTTP и заголовки ответов, даже если это недопустимо в обычном сервлете, но лишь в том случае, если какие-то данные вывода уже были отправлены клиенту.
out (вывод)	javax.servlet.jsp.JspWriter	Объект, который пишет в выходной поток. Область видимости - страница. Основные методы : *clear, clearBuffer, flush, getBufferSize, getRemaining.* Необходимо помнить, размер буфера можно изменять и даже отключить буферизацию, изменяя значение атрибута *buffer*директивы *page*. Также необходимо обратить внимание, что *out* используется практически исключительно скриплетами, поскольку выражения JSP автоматически помещаются в поток вывода, что избавляет от необходимости явного обращения к *out*.

pageContext (содержание страницы)	javax.servlet.jsp.pageContext	Содержимое JSP-страницы. Область видимости - страница. *pageContext* поддерживает доступ к полезным объектам и методам, обеспечивающим явный доступ реализации JSP к специфическим объектам. Основные методы : *getSession, getPage, findAttribute, getAttribute, getAttributeScope, getAttributeNamesInScope, getException.*
session (сеанс)	javax.servlet.HttpSession	Объект типа *Session*, создаваемый для клиента, прислявшего запрос. Область видимости - страница. Основные методы *getId, getValue, getValueNames, putValue.* Сессии создаются автоматически, и переменная *session* существует даже если нет ссылок на входящие сессии. Единственным исключением является ситуация, когда разработчик отключает использование сессий, используя атрибут*session* директивы *page*. В этом случае ссылки на переменную *session* приводят к возникновению ошибок при трансляции JSP страницы в сервлет.
application (приложение)	javax.servlet.ServletContext	Контекст сервлета, полученный из объекта конфигурации сервлета при вызове методов : *getServletConfig или getContext*. Область видимости - приложение. Основные методы : *getMimeType, getRealPath.*
config (конфигурация)	javax.servlet.ServletConfig	Объект *ServletConfig* текущей страницы JSP. Область видимости - страница. Основные методы : *getInitParameter, getInitParameterNames*
page (страница)	java.lang.Object	Экземпляр класса реализации текущей страницы JSP, обрабатывающий запрос. Область видимости - страница. Объект доступен, но, как правило, используется редко. По сути является синонимом для *this*, и не нужен при работе с Java.
exception (исключение)	java.lang.Throwable	Объект *Throwable*, выводимый в страницу ошибок *error page*. Область видимости - страница. Основные методы : *printStackTrace, toString, getMessage, getLocalizedMessage*.

Пересылка и перенаправление

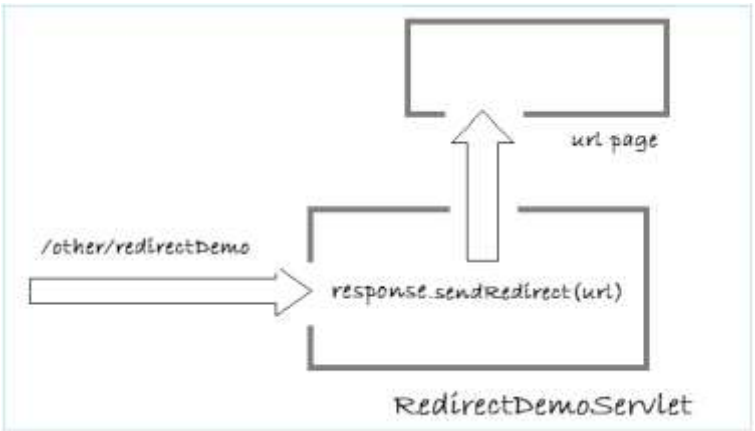
Пересылка (Forward): Когда запрос (request) браузера отправлен к **Servlet**, он может переслать запрос на другую страницу (или другой servlet). Адрес на браузере пользователя так же является ссылкой первой страницы, но содержание страницы создается страницей пересылки.
Пересылающая страница обязательно должна быть одной страницей (или Servlet) в вашем веб приложении.

С Forward вы можете использовать **request.setAttribute()** для передачи данных от страницы 1 к странице 2.



Перенаправление (Redirect) позволяет вам перенаправлять на сайты, включая те, что за пределами вебсайта.

Перенаправление (Redirect): Когда запрос (request) от пользователя к Servlet (Страница A), этот Servlet может направить запрос на другую страницу (страница B), и закончить свою задачу. Страница может быть перенаправлена на страницу в вашем приложении, или может быть любой страницей. Адрес на браузере пользователя теперь отображает ссылку страницы B. В отличии от пересылки (Forward). С Redirect вы не можете использовать **request.setAttribute (..)** для передачи данных от страницы A на страницу B.



requestDispatcher - метод forward()

- Когда мы используем метод forward, запрос передается другому ресурсу на том же сервере для дальнейшей обработки.

- В случае пересылки веб-контейнер обрабатывает весь процесс внутри, а клиент или браузер не задействованы.
- Когда forward вызывается на объект requestdispatcher, мы передаем объекты запроса и ответа, чтобы наш старый объект запроса присутствовал на новом ресурсе, который будет обрабатывать наш запрос.
- Визуально мы не можем видеть перенаправленный адрес, он прозрачен.
- Использование метода forward() выполняется быстрее, чем перенаправление отправки.
- Когда мы перенаправляем с помощью forward и хотим использовать одни и те же данные в новом ресурсе, мы можем использовать request.setAttribute(), поскольку у нас есть объект запроса.

SendRedirect

- В случае sendRedirect запрос передается другому ресурсу в другой домен или другой сервер для дальнейшей обработки.
- При использовании sendRedirect контейнер передает запрос клиенту или браузеру, поэтому URL-адрес, указанный внутри метода sendRedirect, отображается как новый запрос клиенту.
- В случае вызова sendRedirect старые объекты запроса и ответа теряются, поскольку он рассматривается как новый запрос браузером.
- В адресной строке мы можем видеть новый перенаправленный адрес. Его непрозрачность.
- sendRedirect медленнее, потому что требуется одно дополнительное путешествие туда, потому что создается новый новый запрос и теряется старый объект запроса. Требуется два запроса браузера.
- Но в sendRedirect, если мы хотим использовать, мы должны хранить данные в сеансе или передавать вместе с URL.

Если вы хотите, чтобы управление передавалось на новый сервер или контекст, и оно рассматривается как совершенно новая задача, тогда мы отправляемся на Send Redirect. Как правило, переадресация должна использоваться, если операция может быть безопасно повторена при перезагрузке веб-страницы браузером, не повлияет на результат.

Работа с параметрами

Станицы JSP могут получать отправленные данные, например, через параметры или в виде отправленных форм, так же, как это происходит в сервлете. Для этого внутри страницы jsp доступен объект request, который позволяет получить данные посредством следующих методов:

- getParameter(String param): возвращает значение определенного параметра, название которого передается в метод. Если указанного параметра в запросе нет, то возвращается значение null.
- getParameterValues(String param): возвращает массив значений, который представляет определенный параметр. Если указанного параметра в запросе нет, то возвращается значение null.

Например

```
<!DOCTYPE html>
<html>
```

```
<head>
    <meta charset="UTF-8" />
    <title>User Info</title>
</head>
<body>
<p>Name: <%= request.getParameter("username") %></p>
<p>Country: <%= request.getParameter("country") %></p>
<p>Gender: <%= request.getParameter("gender") %></p>
<h4>Selected courses</h4>
<ul>
    <%
        String[] courses = request.getParameterValues("courses");
        for(String course: courses){
            out.println("<li>" + course + "</li>");
        }
    %>
</ul>
</body>
</html>
```

Передача данных из сервлета в jsp

Нередко страница jsp обрабатывает запрос вместе сервлетом. В этом случае сервлет определяет логику, а jsp - визуальную часть. И при обработке запроса сервлет может перенаправить дальнейшую обработку странице jsp. Соответственно может возникнуть вопрос, как передать данные из сервлета в jsp?

Есть несколько способов передачи данных из сервлета в jsp, которые заключаются в использовании определенного контекста или scope. Есть несколько контекстов для передачи данных:

- **request** (контекст запроса): данные сохраняются в HttpServletRequest
- **session** (контекст сессии): данные сохраняются в HttpSession
- **application** (контекст приложения): данные сохраняются в ServletContext

Данные из контекста запроса доступны только в пределах текущего запроса. Данные из контекста сессии доступны только в пределах текущего сеанса. А данные из контекста приложения доступны постоянно, пока работает приложение.

Но вне зависимости от выбранного способа передача данных осуществляется с помощью метода setAttribute(name, value), где name - строковое название данных, а value - сами данные, которые могут представлять различные данные.

Наиболее распространенный способ передачи данных из сервлета в jsp представляют атрибуты запроса. То есть у объекта HttpServletRequest, который передается в сервлет, вызывается метод setAttribute(). Этот метод устанавливает атрибут, который можно получить в jsp.

Для вывода атрибутов применяется специальный синтаксис EL: в фигурные скобки {} заключается выводимое значение

```
<p>Name: ${name}</p>
<p>Age: ${age}</p>
```

Expression Language

Expression Language или сокращенно EL предоставляет компактный синтаксис для обращения к массивам, коллекциям, объектам и их свойствам внутри страницы jsp. Он довольн прост. Вставку окрывает знак $, затем в фигурные скобки {} заключается выводимое значение

Откуда эти данные берутся? EL пытается найти значения для этих данных во всех доступных контекстах.

И EL просматривает все эти контексты в следующем порядке:

1) Контекст страницы (данные сохраняются в PageContext)
2) Контекст запроса
3) Контекст сессии
4) Контекст приложения

Соответственно, если контексты запроса и сессии содержат атрибут с одним и тем же именем, то будет использоваться атрибут из контекста запроса.

Затем найденное значение (если оно было найдено) конвертируется в строку и выводится на страницу.

```
<%
    pageContext.setAttribute("name", "Tom");
%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>JSP Application</title>
</head>
<body>
<p>Name: ${name}</p>
</body>
</html>
```

Однако может сложиться ситуация, что сразу в нескольких контекстах одновременно содержатся данные с одним и тем же именем, например, "name". Соответственно EL будет получать данные в порядке просмотра контекстов. Но, возможно, нам захочется выводить данные из какого-то определенного контекста. В этом случае перед названием данных мы можем указать название контекста: pageScope, requestScope, sessionScope или applicationScope. Например:

```
<%
    pageContext.setAttribute("name", "Bob");
%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>JSP Application</title>
</head>
<body>
<p>Name: ${requestScope.name}</p>
</body>
</html>
```

Встроенные объекты Expression Language

По умолчанию Expression Language предоставляет ряд встроенных объектов, которые позволяют использовать различные аспекты запроса:

- param: объект, который хранит все переданные странице параметры
- paramValues: хранит массив значений для определенного параметра (если для параметра передается сразу несколько значений)
- header: хранит все заголовки запроса
- headerValues: предоставляет массив значений для определенного заголовка запроса
- cookie: предоставляет доступ к отправленным в запросе кукам

- initParam: возвращает значение для определенного параметра из элемента context-param из файла web.xml
- pageContext: предоставляет доступ к объекту PageContext, который представляет контекст текущей страницы jsp

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>User Info</title>
</head>
<body>
<p>Name: ${param.name}</p>
<p>Age: ${param.age}</p>
</body>
</html>
```

Через объект param здесь получаем из запроса значения параметров name и age. Значения для параметров можно передать как через строку запроса, так и через отправку формы.

Java Bean

Компоненты JavaBeans – это многократно используемые классы Java, позволяющие разработчикам существенно ускорять процесс разработкии WEB-приложений путем их сборки из программных компонентов. JavaBeans и другие компонентные технологии привели к появлению нового типа программирования – сборки приложений из компонентов, при котором разработчик должен знать только сервисы компонентов; детали реализации компонентов не играют никакой роли.

Компоненты JavaBean – это одноуровневые объекты, использующиеся для того, чтобы инкапсулировать в одном объекте сложный код, данные или и то и другое. Компонент JavaBean может иметь свойства, методы и события, открытые для удаленного доступа.

Компонент JavaBean – это java-класс, удовлетворяющий определенным соглашениям о наименовании методов и экспортируемых событий. Одним из важных понятий технологии JavaBeansявляется внешний интерфейс properties (свойства). Property – это пара методов (getter и setter), обеспечивающих доступ к информации о внутреннем состоянии компонента JavaBean.

Для обращения к компонентам JavaBeans на странице JSP необходимо использовать следующее описание тега в разделе head :

<jsp:useBean id="BeanID" [scope="page | request | session | application"] class="BeanClass" />

BeanID определяет имя компонента JavaBean, являющееся уникальным в области видимости, заданной атрибутом scope. По умолчанию принимается область видимости scope="page", т.е. текущая страница JSP.

Обязательный атрибут класса компонента "class" может быть описан следующим способом:

class="имя класса" [type="полное имя суперкласса"]

Свойство компонента JavaBean с именем myBean устанавливается тегом:

<jsp:setProperty name="myBean" property="Имя свойства" value="Строка или выражение JSP" />

Для чтения свойства компонента JavaBean с именем myBean используется тег:

<jsp:getProperty name="myBean" property="Имя свойства" />

В следующем листинге приведен пример компонента JavaBean, содержащего строку mystr, используемую в качестве свойств.

Листинг компонента JavaBean

```
package beans;

public class myBean
{
    private String mystr;
    //-----------------------------------------------
    public void setMystr(String mystr)
    {
          this.mystr = mystr;
    }
    //-----------------------------------------------
    public String getMystr()
    {
          return mystr;
    }
    //-----------------------------------------------
}
```