JavaServer Pages.

Основной недостаток технологии сервлетов при использовании в качестве компонентов представления (presentation components) состоит в смешении в одном исходном файле кода на различных языках. В случае простого веб-приложения это как минимум Java и HTML, но сложные веб-приложения на клиентской стороне требуют помимо описания структуры страницы (HTML) также информацию о ее внешнем оформлении (таблицы стилей CSS) и поведении (скрипты JavaScript). Сопровождать исходный код, состоящий из смеси языков, очень тяжело. Интуитивно ощущается потребность в отделении представления данных пользователей от логики обработки данных и от структуры их хранения. При этом желательно иметь возможность описывать представление данных в декларативной форме в виде некоторого шаблона, абстрагируясь при этом от низкоуровневой механики генерации вывода, от деталей, связанных с буферизацией, и т.п. Необходимо средство конструирования страниц ответов непосредственно как HTML-страниц с включением в них при необходимости элементов специального вида, генерирующих нужные фрагменты структуры динамически на этапе обработки запроса. Причем эти элементы должны вписываться в идеологию HTML, то есть оформляться открывающим и закрывающим тегом, иметь атрибуты, тело и прочее.

В качестве примера рассмотрим страницу следующего вида.

Здесь лишь тело таблицы является тем фрагментом, который нужно генерировать динамически на этапе обработки запроса в соответствии с информацией, извлеченной из БД, которая на этапе разработки приложения неизвестна и в ходе его разработки может меняться. Все остальные части страницы не изменяются и известны заранее, при этом заголовок отчета и шапка таблицы специфична для данной конкретной страницы приложения, а логотип, горизонтальное и вертикальное меню (на рис. отсутствуют) могут быть общими для всех страниц приложения. В этих условиях формировать вывод сервлетом неэффективно и необходимо высокоуровневое средство для декларативного

модульного описания шаблона страницы. Таким средством на платформе Java EE является технология JSP.

Основные элементы стандартного синтаксиса JSP.

JSP-страницы – это текстовые файлы, которые конструируются из элементов пяти видов:

1) шаблон (template);
2) директива (directive),
3) скриптовый элемент (scripting element),
4) действие (action),
5) EL-элемент (EL-element).

Еще до того, как JSP-страница сможет обработать первый поступивший к ней запрос, она преобразуется так или иначе в соответствующий ей сервлет, который называется Page Implementation Class (PIC). Данное преобразование выполняется в два этапа. Сначала исходный файл JSP-страницы по определенным правилам транслируется в исходный файл PIC. Эта фаза фактически есть преобразование одного текста в другой. Спецификация JSP предусматривает свои правила трансляции для каждого из пяти видов элементов. На втором этапе исходный файл PIC компилируется стандартным компилятором Java. На этапе обработки запросов, относящихся к JSP, используется экземпляр соответствующего ей PIC.

Момент времени, когда выполняется преобразование JSP-страницы в PIC, специфичен для сервера приложений. Спецификация JSP предусматривает три варианта:

1) Трансляция может быть выполнена явным образом до того, как будет выполнена сборка и установка приложения. Для этого используется специальный компилятор, который входит в комплект поставки сервера приложений. Для SJS AS 8.1 это утилита командной строки `jspc`. После явной трансляции на этапе сборки приложения уже работают не с JSP, а с сервлетом.

2) Трансляция может быть выполнена сервером приложений на этапе установки приложения. В SJS AS 8.1 для включения этого режима необходимо выполнить соответствующую настройку сервера приложений.

3) Трансляция может быть выполнена «на лету» при первом обращении к JSP-странице. При этом сервер отслеживает соответствие исходной версии JSP и соответствующего PIC. При обновлении JSP-страниц приложения повторная трансляция выполняется автоматически. В SJS AS 8.1 данный режим используется по умолчанию.

Шаблоны.

Шаблон содержит статические части, генерируемые HTML-страницей, или ответа другого вида (например, простого текста или XML-документа). Шаблон оформляется непосредственно в том виде, в котором он должен быть передан клиенту. Шаблон включается в ответ без трансформации.

Директивы.

Это специальные элементы, которые обрабатываются на первой фазе преобразования и влияют на нее. Они не предназначены для генерации вывода. По завершении трансляции директив не существует. Директивы JSP в этом смысле аналогичны директивам в языках программирования (например, в Ассемблере). Директивы оформляются так:

```
<%@ имя_директивы атрибут1="значение1" атрибут2="значение2" ... %>
```

Конкретные примеры:

```
<%@ page isErrorPage="true" %>
<%@ include file="menu.html" %>
```

Скриптовые элементы.

Выделяют три вида скриптовых элементов:

1) объявления (declarations);
2) выражения (expressions);
3) скриптлеты (scriptlets).

В любом случае скриптовые элементы содержат код на языке Java, который включается в ходе первой фазы преобразования в исходный код PIC в точности в том виде, в каком он написан в исходном файле JSP.

Объявления предназначены исключительно для описания переменных и функций, которые можно использовать в других скриптовых элементах этой же страницы. После трансляции объявления переменные становятся полями PIC, а объявления функций – его методами. Объявления оформляются так:

```
<%! текст_объявления %>
```

Примеры:

```
<%! int i = 0; // объявление переменной %>
<%! // объявление функции
      int m(...) {...} %>
```

Выражения используются для включения в состав ответа фрагментов, вычисляемых динамически на этапе обработки запросов. Вычисления описываются выражением на языке Java. Результат вычисления преобразуется в строку. Выражения оформляются так:

```
<%= текст_выражения %>
```

Выражения могут использоваться также для формирования значений атрибутов действий на этапе обработки действий:

```
<doSomethig source="<%= ... %>" >
```

Пример:

```
<%= request.getParameter("username") %>
<%= product.get("model") %>
```

Скриптлеты предназначены для описания фрагмента кода, который непосредственно включается в тело определяемого в PIC специального метода `jspService()`, вызываемого на этапе обработки запроса и формируемого на первом этапе трансляции JSP-страницы. Скриптлеты оформляются так:

```
<% текст_скриптлета %>
```

Пример:

```
<% if(...) { %>
      <!—Шаблон -->
<% } else if (...) { %>
      <!—Шаблон -->
<% } else { %>
      <!—Шаблон -->
<% } %>
```

Действия.

Действия – это основной механизм технологии JSP для декларативного использования некоторой функциональности. Действия позволяют избавить исходный код JSP-страницы от скриптовых элементов, то есть от фрагментов кода на языке Java, и таким образом избежать смешения языков или минимизировать его. За редкими исключениями следует стремиться использовать действия вместо скриптовых элементов. Действия оформляются в соответствии с синтаксисом XML вне зависимости от используемого синтаксиса JSP (стандартный или XML). Действия оформляются так:

```
<префикс:имя_действия атрибут1="значение1" атрибут2="значение2" ... >
      <!—тело действия -->
```

```
    ...
</префикс:имя_действия>
```

Если тело у действия отсутствует, то так:

```
<префикс:имя_действия атрибут1="значение1" атрибут2="значение2" ... />
```

Все действия делятся на стандартные и определяемые программистом (custom actions). Синтаксис и семантика стандартных действий строго закреплены спецификацией JSP, для стандартных действий зарезервирован префикс `jsp`, который нельзя использовать для custom actions.

Механизм custom actions позволяет разработчику расширять набор стандартных действий JSP собственными действиями, для чего необходимо разработать класс, реализующий функциональность действия, и описать соответствующие действию элементы (его имя, атрибуты, специфику тела, порождаемые переменные и др.), который и будет использован в JSP-страницах для активации функциональности, реализованной классом.

В любом случае действия обрабатываются на этапе выполнения приложения на сервере, то есть на первом этапе трансляции для них генерируется определенный код в составе PIC.

Для формального описания элементов действий и связанной с ними дополнительной информации используются XML-дескрипторы специального вида. Один дескриптор может описывать несколько действий (библиотеку действий), называется Tag Library Descriptor (TLD) и оформляется файлом с расширением `tld`.

EL-элементы.

EL-элементы могут использоваться в шаблоне, а также в атрибутах действий, вычисляемых во время выполнения. Данные выражения записываются на особом языке Expression Language (EL), подробное описание которого будет дано позднее.

Комментарии и esc-последовательности.

В JSP-страницы можно внедрять комментарии трех видов:

1) комментарии, оформленные в виде `<!-- комментарий -->`, считаются частью шаблона и выводятся в генерируемый ответ. В тексте таких комментариев распознаются элементы JSP, в том числе выражения и EL-элементы;
2) комментарии, оформленные в виде `<%-- комментарий --%>`, используются для отключения части функциональности JSP. JSP-элементы в таких комментариях не распознаются.
3) в теле скриптовых элементов можно использовать комментарии, специфичные для языка скриптов (то есть для Java). Например,

```
<% // комментарий
    if (…) { … } %>
```

В JSP-документах для комментирования используются стандартные средства XML `<!-- комментарий -->`. В таких комментариях JSP-элементы не распознаются и не передаются в составе ответа. Для передачи комментариев в составе ответа их нужно оформлять как шаблон. Комментарии в скриптовых элементах оформляются так же, как и в JSP-страницах.

В JSP предусмотрены следующие esc-последовательности:

1) в EL-элементах специальных esc-последовательностей не предусмотрено и `${` можно использовать литерально. Например, `${'${'}` – результат `${`. То же относится и к `}`: `${'}'}` – результат `}`;
2) в скриптовых элементах пара символов `%>` при необходимости их литерального использования заменяется на `%\>`;

3) в шаблоне `<%` заменяется на `<\%`, `$` – на `\$`;
4) в атрибутах `' – \'`, `" – \"`, `\ – \\`, `$ – \$`, `%> – %\>`, `<% – <\%`. Также можно использовать `&apos;` вместо `'` и `&quot;` вместо `"`. Например, `<%= "Joe said %\>%>`, результат – `Joe said %>`. Esc-последовательность `\$` используется только в случае, если интерпретация EL-элементов включена, в противном случае она воспринимается просто как пара символов;
5) в JSP-документах используются только стандартные для технологии XML esc-последовательности: `< – <`, `> – >`, `' – &apos;`. `" – &quot;`, `$ – \$`. В JSP-документах пара символов `\\` esc-последовательностью не является. Соответственно `\\${1+1}` будет вычислено в `\2` в стандартном синтаксисе при включенном EL, в `\${1+1}` в стандартном синтаксисе при отключенном EL, и в `\${1+1}` в XML-синтаксисе.

Стандартные директивы JSP.

Для JSP-страниц спецификацией предусмотрены следующие стандартные директивы: `page`, `taglib`, `include`.

Директива `page` определяет общие свойства страницы, которые влияют на ее трансляцию. В ней предусмотрены следующие атрибуты:

- `language` указывает язык, на котором записываются скриптовые элементы, единственным допустимым значением является значение `java` (оно используется по умолчанию);
- `extends` указывает полностью квалифицированное имя класса, который будет использован как базовый для PIC. Это позволяет заменить стандартный базовый класс, входящий в комплект поставки сервера приложений;
- `import` используется для импорта имен классов и интерфейсов из различных пакетов в пространство имен страницы. Импортированные имена можно использовать в скриптовых элементах без квалификации. Значение атрибута – разделенный запятыми список имен. Атрибут выполняет ту же функцию для JSP, что и директива `import` для Java-класса. Неявно импортируются следующие имена: `java.lang.*`, `javax.servlet.*`, `javax.servlet.http.*`, `javax.servlet.jsp.*`;
- `session` указывает, что JSP-страница требует участия в веб-сессии. Если да (по умолчанию), то в скриптовых элементах страницы доступен встроенный объект `session` типа `HttpSession`;
- `buffer` используется для управления буферизацией ответа, генерируемого JSP-страницей. Возможные значения: `none`, *X*`kb`, где *X* – размер буфера в килобайтах (по умолчанию – 8 кБ). Значение `none` отключает буферизацию, иначе используется буфер размером не менее указанного;
- `autoFlush` управляет режимом автоматического сброса буфера при переполнении. Если режим отключен, то при каждом переполнении возникает исключение, иначе содержимое буфера передается клиенту и буфер очищается (по умолчанию сброс включен). Сочетание значений `buffer="none"`, `autoFlush="false"` считается недопустимым;
- `isThreadSafe` указывает поточную модель для PIC. При `isThreadSafe="true"` используется многопоточная модель (по умолчанию), иначе – однопоточная. Последнее приводит к тому, что класс PIC реализует интерфейс `SingleThreadModel`. Использование однопоточной модели не рекомендуется;
- `info` указывает краткое описание JSP-страницы, которое возвращается из метода `getServletInfo()`, переопределяемого в PIC;
- `isErrorPage` указывает, что данная JSP-страница предназначена для обработки ошибок, вследствие чего в ней доступен встроенный объект с именем `exception`

типа `java.lang.Throwable`. Встроенный объект `exception` соответствует объекту исключения, возникшего при обращении к другому компоненту, в результате чего было выполнено перенаправление запроса на данную JSP-страницу с использованием механизма страниц обработки ошибок. Если к данной странице выполнено непосредственное обращение или перенаправление на нее было выполнено не в результате возникновения исключения (в том числе по коду статуса ответа), то значение объекта `exception` будет `null`. Если `isErrorPage="false"` (по умолчанию), то имя объекта `exception` не доступно, и попытка его использования приводит к ошибке трансляции JSP;

- `errorPage` указывает относительный URL компонента, на который будет сделано перенаправление запроса в случае возникновения исключения при обработке запроса на данной JSP-странице. Объект возникшего исключения перед перенаправлением помещается в запрос как атрибут с именем `javax.servlet.error.exception`. Указанный в данном атрибуте URL имеет приоритет перед теми страницами обработки ошибок, которые заданы в установочном дескрипторе веб-приложения. Перенаправление не может быть выполнено, если ответ находится в отправленном состоянии. Значение по умолчанию данного атрибута зависит от реализации сервера приложений;
- `contentType` указывает MIME-тип и, возможно, кодировку ответа, генерируемого данной JSP-страницей. Указывается в форме `"mime-тип"` или `"mime-тип;charset=`*кодировка*`"`. По умолчанию MIME-тип принимается `text/html` для JSP-страниц в стандартном синтаксисе и `text/xml` для JSP-страниц в XML-синтаксисе. Кодировка по умолчанию принимается Latin-1 в стандартном синтаксисе и UTF-8 в XML-синтаксисе. Например, `"text/html;charset=windows-1251"` – HTML в кодировке ANSI 1251; `"text/plain"` – простой текст в кодировке Latin-1;
- `pageEncoding` указывает кодировку, в которой записана JSP-страница. Для JSP-страниц в стандартном синтаксисе по умолчанию полагается кодировка Latin-1. Для JSP-страниц в XML-синтаксисе кодировка указывается в объявлении XML (XML declaration), например: `<?xml version="1.0" encoding="UTF-8" ?>`;
- `isELIgnored` позволяет включить (`false`) или выключить (`true`) обработку EL-элементов на данной JSP-странице. Значение по умолчанию определяется настройками в установочном дескрипторе веб-приложения. Если обработка отключена, то EL-элементы рассматриваются как часть шаблона или как литеральные значения атрибутов действий.

Директива `taglib` используется для подключения дескриптора библиотеки действий, определенных программистом, к данной JSP-странице и назначения некоторого префикса, который будет использоваться совместно с именами действий этой библиотеки для предотвращения возможного конфликта имен. Последний может возникнуть, когда одна JSP-страница использует две библиотеки, в которых имена некоторых действий совпадают. Введенный префикс должен быть уникальным в пределах JSP-страницы. В директиве `taglib` предусмотрены следующие атрибуты:

- `prefix` вводит непустой префикс для имен действий подключаемой библиотеки. Префиксы `jsp`, `jspx`, `java`, `servlet`, `sun`, `sunw` зарезервированы. Введенный префикс используется так: *`префикс`*`:`*`имя_действия`*, например, `c:forEach`;
- `uri` указывает глобально-уникальный идентификатор (URI) библиотеки действий. Этот идентификатор может формироваться произвольно, но принято формировать его в формате URL. Этот идентификатор не используется непосредственно для получения дескриптора библиотеки. Файл дескриптора (TLD – tag library descriptor) должен быть расположен в веб-модуле. Абстрактный URI, идентифицирующий библиотеку, связывается с физическим расположением ее дескриптора в установочном дескрипторе веб-приложения. Связка «абстрактный URI – физический

TLD» может отсутствовать для так называемых well-known URIs, которые распознаются сервером приложений и для которых в комплект поставки сервера приложений входит и TLD, и реализующие действия классы. Идентификаторы библиотек, вводимые в JSP Standard Tag Library (JSTL), являются примером well-known URI.

- `tagdir` указывает каталог в архиве веб-приложения, в котором расположены custom actions в виде тэг-файлов тегов (tag files). Этот каталог должен находиться в подкаталоге `/WEB-INF/tags`. Если он расположен в другом месте или каталог не существует, возникает ошибка трансляции JSP. При использовании этого атрибута задействуется так называемый неявный TLD, что позволяет использовать новые возможности механизма custom actions, введенные в JSP 2.0.
- Одновременно может использоваться только один из атрибутов `uri` и `tagdir`, иначе возникает ошибка трансляции JSP.

Директива `include` используется для включения содержимого некоторого файла в состав исходного текста JSP на этапе трансляции. Атрибут `file` указывает путь к включаемому файлу относительно исходного JSP-файла. Включаемый файл должен быть доступен на этапе трансляции, иначе возникает ошибка. Этот файл может содержать элементы, специфичные для JSP, которые обрабатываются на этапе трансляции страницы.

Встроенные объекты JSP (JSP implicit objects).

Встроенные объекты JSP-страницы инициализируются автоматически еще до того, как начинает работу та часть PIC, которая была сгенерирована по исходному тексту JSP. Эти объекты можно использовать в выражениях и скриптлетах, но нельзя использовать в объявлениях и EL-элементах (для последних существуют свои встроенных объекты). Встроенные объекты доступны по их именам. У каждого объекта свой тип и своя область видимости, которая определяет доступность и время жизни соответствующего объекта.

Существует четыре области видимости объектов JSP (как встроенных, так и определяемых пользователем): `page`, `request`, `session`, `application`. Понятие области видимости относится не только к встроенным объектам, но и к пользовательским объектам, которые создаются и используются в рамках JSP.

Область видимости `page` ограничивает доступность объектов пределами конкретной JSP-страницы и временем, в течение которого эта страница участвует в обработке запроса. Так, объекты с областью видимости `page`, созданные в пределах одной JSP-страницы, не будут доступны другой странице после перенаправления запроса. Объекты с областью видимости `page` могут быть либо полями PIC (вводятся с помощью объявлений), либо локальными переменными метода `_jspService` (вводятся в скриптлетах), либо атрибутами объекта типа `javax.servlet.jsp.PageContext`, который поддерживается сервером приложений для каждой JSP-страницы (этот объект называют контекстом JSP-страницы).

Область видимости `request` ограничивает доступность объектов теми веб-компонентами, которые участвуют в обработке запроса (за счет перенаправления запросов таких компонентов может быть несколько) и временем, в течение которого идет обработка запроса. Объекты с этой областью видимости сохраняются как атрибуты запроса.

Область видимости `session` ограничивает доступность объектов веб-компонентами приложения (у каждого приложения своя отдельная сессия) и временем, в течение которого эта сессия поддерживается сервером приложений для конкретного пользователя. Объекты с этой областью видимости сохраняются как атрибуты сессии.

Область видимости `application` ограничивает доступность объектов веб-компонентами приложения и временем, в течение которого это приложение доступно пользователям (в штатном режиме это время с момента запуска до момента останова

сервера приложений). Объекты с этой областью видимости сохраняются как атрибуты контекста веб-приложения.

Спецификация JSP определяет встроенные объекты, приведенные в табл. 7.1.

Имена объектов, начинающихся с префиксов `jsp`, `_jsp`, `jspx`, `_jspx` в любом сочетании регистра, зарезервированы, их нельзя использовать для пользовательских объектов.

Стандартные действия JSP.

Стандартные действия имеют предопределенные синтаксис и семантику и обязательно поддерживаются любым сервером приложений. Любые действия в JSP-страницах, и стандартные в том числе, оформляются по правилам XML.

jsp:useBean

Действие `jsp:useBean` выполняет поиск пользовательского объекта по имени в указанной области видимости и при необходимости создает его, после чего делает доступным для скриптлетов и выражений. Атрибуты:

- `id` – указывает имя искомого объекта (фактически это имя атрибута, запроса, сессии и др.); это же имя используется для объявления локальной переменной, в которую помещается ссылка на найденный объект. Именно эта переменная далее может быть использована скриптлетами и выражениями. Данный атрибут обязателен.
- `scope` – указывает область видимости, возможные значения: `page` (по умолчанию), `request`, `session`, `application`.
- `type` – указывает тип (полностью квалифицированное имя класса), с которым будет объявлена порождаемая действием переменная. Тип найденного или созданного объекта будет приведен к этому типу.
- `class` – указывает полностью квалифицированное имя класса, который будет использован для создания объекта, если он не будет найден. Класс должен быть не абстрактным и содержать публичный конструктор без параметров.

Допустимы следующие сочетания атрибутов: `type` или `class`, или `type` и `class`.

Типы, указанные атрибутами `type` и `class` должны быть совместимы, иначе при преобразовании возникнет `ClassCastException`. Если атрибут `class` указан, а `type` нет, то тип переменной будет соответствовать указанному классу.

Последовательность действий, выполняемая этими стандартными действиями:

1) выполняется поиск атрибута с указанным именем в указанной области видимости;
2) объявляется переменная с указанным именем и типом, указанным атрибутом `type` (если он есть) или `class`;
3) если объект найден, ссылка на него сохраняется в объявленной переменной, при этом выполняется преобразование типа объекта к типу переменной, и если типы несовместимы, то возникает исключение `java.lang.ClassCastException`;
4) если объект не найден, а атрибут `class` не указан или указывает имя абстрактного класса или интерфейса, а также если в классе, указанном в атрибуте `class`, отсутствует публичный конструктор без параметров, то возникает исключение `java.lang.InstantiationException`, в противном случае создается объект указанного класса, при этом вызывается публичный конструктор без параметров.

Если действием `useBean` объект был найден в указанной области видимости, то тело действия игнорируется. Если же объект не был найден, и его пришлось создать, то тело действия обрабатывается обычным образом. Как правило, действие `useBean` содержит вложенные действия, используемые для инициализации созданного объекта.

Если действие `useBean` создает объект, то ссылка на него записывается не только в порожденную переменную, но и в атрибут указанной области видимости с указанным именем.

jsp:setProperty

Действие `jsp:setProperty` используется для установки свойств компонента JavaBean, созданного или найденного действием `jsp:useBean`. Атрибуты:

- `name` – указывает имя компонента JavaBean, созданного или найденного ранее с помощью действия `jsp:useBean`.
- `property` – указывает устанавливаемые действием свойства, при этом используются соглашения по именам свойств, принятые для JavaBeans.
- `value` – указывает значение свойства, оно может формироваться на этапе обработки запроса выражением или EL-элементом.
- `param` – указывает имя параметра запроса, из которого берется значение устанавливаемого свойства.

При этом имеют место следующие особенности:

1) если указанный атрибутом `param` параметр запроса отсутствует либо содержит пустую строку, то значение свойства JavaBean не изменяется;
2) запрещается одновременно указывать атрибуты действия `param` и `value`;
3) допускается не указывать ни один из атрибутов `param` и `value`. В этом случае значение свойства извлекается из параметра запроса, имя которого совпадает с именем свойства;
4) если не указан ни один из атрибутов `param` и `value`, то атрибут `property` может иметь значение `"*"`. При этом установка свойств JavaBean будет выполнена на основе соответствия имен и типов свойств JavaBean и параметров запроса.

Часто действие `jsp:setProperty` используют в теле действия `jsp:useBean` для инициализации свойств JavaBean, если он не был найден.

Примеры:

```
<jsp:useBean id="customer" scope="request"
   class="ru.johnheadlong.bean.Customer">
      <jsp:setProperty name="customer" property="*" />
</jsp:useBean>
<jsp:setProperty name="customer" property="firstName" param="first-name" />
<jsp:setProperty name="customer" property="firstName" value='<%=
request.getParameter("first-name") %>' />
<jsp:setProperty name="customer" property="firstName"
   value="${param.first-name}" />
```

jsp:getProperty

Действие `jsp:getProperty` позволяет получить значение свойства компонента JavaBean, созданного или найденного ранее действием `jsp:useBean`, и вывести это значение в генерируемый объект. Атрибуты:

- `name` – имя объекта JavaBean;
- `property` – имя свойства.

Пример:

```
<jsp:useBean id="customer" scope="request" />
<h1>Hello, <jsp:getProperty name="customer" property="login" /></h1>
<h1>Hello, ${customer.login}</h1>
```

jsp:include

Действие `jsp:include` позволяет включить в ответ, генерируемый данной JSP-страницей, ответ, генерируемый другим веб-компонентом. В отличие от директивы `include` включение выполняется на этапе обработки запроса, а не на этапе трансляции

JSP-страницы. Относительный URL компонента, чей ответ включается, указывается обязательным атрибутом `page`. Этот относительный URL интерпретируется относительно URL данной JSP-страницы, значение атрибута формируется выражением или EL-элементом. Необязательный атрибут `flush` определяет, выполняется ли сброс буфера перед включением (`true`) или нет (`false`, по умолчанию).

К действию `jsp:include` применимы те же ограничения, что и к методу `include()` интерфейса `javax.servlet.RequestDispatcher`. В теле действия `jsp:include` можно использовать действия `jsp:param` для установки дополнительных параметров запроса, которые будут доступны компоненту, ответ которого включается.

jsp:forward

Действие `jsp:forward` позволяет перенаправить обработку запроса другому веб-компоненту. Относительный URL указывается обязательным атрибутом `page`. Значение атрибута формируется выражением или EL-элементом. К этому действию применимы те же ограничения, что и к методу `forward()` интерфейса `javax.servlet.RequestDispatcher`. Здесь переход ответа в отправленное состояние происходит в одном из двух случаев:

1) если вывод буферизуется, то при первом переполнении буфера;
2) если буферизация отключена, то при генерации любого вывода.

В теле действия `jsp:forward` можно использовать действия `jsp:param` для установки дополнительных параметров запроса, которые будут доступны компоненту, которому перенаправляется запрос.

jsp:param

Действие `jsp:param` используется в теле действий для указания дополнительных параметров запроса. Эти параметры добавляются в оригинальный запрос и имеют приоритет перед существующими параметрами при совпадении имен, то есть в массиве значений, возвращаемом методом `getParameterValues()` интерфейса `javax.servlet.ServletRequest` новые значения будут идти первыми.

У действия `jsp:param` имеется два обязательных атрибута:

- `name` – имя добавляемого параметра запроса;
- `value` – значение параметра, может формироваться выражением или EL-элементом.

jsp:plugin

Действие `jsp:plugin` используется для внедрения в генерируемый ответ (HTML-страницу) Java-аплета. Java-аплет – это один из видов Java-программ. По сути это написанный и упакованный по определенным правилам компонент, который может внедряться по ссылке в HTML-страницы, загружаться браузером по протоколу HTTP и исполняться в среде браузера, то есть Java-аплет – это клиентский веб-компонент.

Несмотря на то, что язык HTML предлагает стандартные средства для внедрения в состав веб-страницы произвольных внешних объектов (элемент `object`) и аплетов (устаревший элемент `applet`), при их использовании могут возникнуть проблемы. Они связаны с тем, что аплет может использовать API, добавленный в версии J2SE (например, библиотеку Swing) более поздние, чем установленная на клиенте JRE (поставляемая вместе с браузером или установленная отдельно и интегрированная с ним). Поэтому необходим механизм, который позволяет указать номер версии JRE, которая требуется для аплета, проверить наличие ее на клиенте и в случае ее отсутствия предложить клиенту загрузить нужную версию JRE с того или иного сайта. Этот функционал реализуется с помощью Java plug-in, который является частью JRE. Активация Java plug-in выполняется

разными способами в зависимости от браузера. Для генерации нужного HTML-кода для внедрения аплета в зависимости от типа браузера и предусмотрено действие `jsp:plugin`.

Атрибуты:

- `type` – предусмотрены значения `bean` и `applet`. Для внедрения аплетов используется значение `applet`;
- `jversion` – указывает минимальный номер версии JRE (значение по умолчанию – 1.2);
- `nsplugin` и `ieplugin` – указывают URL'ы (первый – для Netscape Navigator / Mozilla Firefox, а второй – для MS Internet Explorer), откуда браузер может предложить пользователю загрузить нужную версию JRE. Значение по умолчанию зависит от реализации;

Остальные атрибуты действия `jsp:plugin` аналогичны тем, что определены для элемента applet спецификации HTML:

- `code` – полностью квалифицированное имя класса аплета;
- `codebase` – URL, с которого можно загрузить класс аплета или содержащий его архив. Если аплет распространяется в виде архива, то имя файла архива указывается атрибутом `archive` (это может быть jar- или zip-файл, MS IE поддерживает также cab-файлы, но использовать его не рекомендуется);
- остальные атрибуты влияют на размещение аплета при отображении веб-страницы: `height` – высота аплета, `width` – ширина аплета, `hspace` и `vspace` – горизонтальное и вертикальное поля, `align` – выравнивание по вертикали.

В теле действия `jsp:plugin` можно использовать действия `jsp:params` и `jsp:fallback`. Действие `jsp:params` используется как контейнер для вложенных в него действий `jsp:param`, по которым генерируется HTML-код для инициализации аплета. Действие `jsp:fallback` используется как контейнер для фрагмента ответа, который отображается в случае, если Java-аплет по тем или иным причинам не смог запуститься.

Вопросы локализации.

Для JSP-страницы в стандартном синтаксисе кодировка исходного файла определяется так:

1) по значению соответствующего данному файлу свойства `page-encoding`, указанного в установочном дескрипторе (см. конфигурирование JSP);
2) по значению атрибута `pageEncoding` директивы `page`. Если значение атрибута не совпадает со значением свойства `page-encoding`, заданного для этого файла, то возникает ошибка трансляции;
3) по значению `charset` в атрибуте `contentType` директивы page. Используется, только если не задано ни одно из перечисленных выше значений;
4) если не указано ни одно из вышеперечисленных значений, то используется кодировка Latin-1.

Для JSP-страниц в XML-синтаксисе кодировка исходного файла указывается в XML-объявлении. По умолчанию используется UTF-8.

Кодировка ответа, генерируемого JSP-страницей, определяется значением `charset` в атрибуте `contentType` директивы `page`. Если оно не указано, то кодировка ответа определяется так:

1) для JSP-страницы в XML-синтаксисе используется кодировка UTF-8;
2) для JSP-страницы в стандартном синтаксисе используется значение атрибута `pageEncoding` директивы `page` или значение свойства `page-encoding`, указанного для данного JSP-файла. Если они не указаны, то используется кодировка Latin-1.

Разработчик JSP может изменить устанавливаемую автоматически кодировку генерируемого ответа на этапе обработки запроса либо методом `setCharacterEncoding()` интерфейса `javax.servlet.ServletResponse` (не рекомендуется), либо действиями JSTL.

Технология JSP не предоставляет средства для непосредственной работы с кодировкой запроса. Для этого используется либо Servlet API (не рекомендуется), либо действие JSTL `fmt:requestEncoding`.

Конфигурирование JSP.

В состав установочного дескриптора могут внедряться специальные конфигурационные элементы для управления JSP-контейнером. В установочном дескрипторе для каждой коллекции JSP-страниц, заданной шаблоном URL, можно выполнить следующие настройки:

1) отключить вычисление EL-элементов;
2) запретить использование скриптовых элементов;
3) указать кодировку исходных файлов JSP;
4) указать, что соответствующие JSP-страницы записаны в XML-синтаксисе независимо от расширения файла;
5) указать JSP-фрагменты, которые будут автоматически включаться (неявной директивой `include`) в самое начало (`include-prelude`) и в самый конец (`include-coda`) каждого JSP-файла коллекции.

В составе шаблона URL, определяющего коллекцию JSP-файлов, можно использовать подстановочный символ *.

По умолчанию:

1) EL-элементы игнорируются, если веб-модуль версии 2.3, и вычисляются для более старших версий;
2) скриптовые элементы разрешены;
3) определение кодировки описано в разделе «Вопросы локализации»;
4) распознавание JSP-документов описано в разделе «XML-синтаксис описания JSP-страниц».

Если не сделано дополнительных указаний, то сервер приложений считает JSP-страницей любой файл с расширением `jsp` (в стандартном синтаксисе) и `jspx` (в XML-синтаксисе), а также все файлы, которые они включают явно директивами `include` или неявно с помощью JSP-свойств `include-prelude` и `include-coda`.

Кроме того, любая из веб-коллекций, используемых для конфигурирования JSP, неявно указывает серверу приложений, что все файлы, входящие в коллекцию, являются JSP-страницами.

Пример JSP-страницы

Данная JSP-страница предназначена для вывода таблицы со сведениями о всех клиентах.

```
<%-- директива --%>
<%@ page contentType="text/html" pageEncoding="UTF-8"
import="java.util.Collection,model.*,dao.*" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<%-- объявление --%>
<%!   String getClientFullName(Client client) {
            return client.getLastName() + " " + client.getFirstName();
        } %>
```

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Client List</title>
    </head>
    <body>

    <%@ include file="/header.jsp" %>
    <h1>All Clients</h1>
    <table border="1">
        <thead>
            <tr>
                <th>Name</th>
                <th>Place</th>
                <th>Email</th>
            </tr>
        </thead>
        <tbody>
            <%-- скриптлет --%>
            <%  // получаем список клиентов из DAO-класса
                Collection<Client> clients = ClientDAO.getAll();
                for (Client client : clients) {
                    pageContext.setAttribute("client", client);
            %>
            <tr>
                <!-- полное имя клиента -->
                <td><a href='<%=
response.encodeURL("infoClient.jsp")%>?id=<%= client.getId() %>'>
                    <%= getClientFullName(client) %></a></td>
                <td>${client.place}</td>
                <%-- выражение --%>
                <td><%= client.getEmail() %></td>
            </tr>
            <% } %>
        </tbody>
    </table>

    </body>
</html>
```

Внешний вид HTML-страницы, полученной в результате обращения к данной JSP-странице, приведен на рис. выше.

Данная JSP-страница при установке содержащего ее веб-приложения на сервер Apache Tomcat 5.5.17 транслируется в следующий PIC:

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import java.util.Collection;
import jdbcsample.model.*;
import jdbcsample.dao.*;

public final class listClients_jsp
    extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

      String getClientFullName(Client client) {
            return client.getLastName() + " " + client.getFirstName();
        }
  private static java.util.List _jspx_dependants;

  public Object getDependants() {
    return _jspx_dependants;
```

```
  }

  public void _jspService(HttpServletRequest request, HttpServletResponse
response) throws java.io.IOException, ServletException {

    JspFactory _jspxFactory = null;
    PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    JspWriter _jspx_out = null;
    PageContext _jspx_page_context = null;


    try {
      _jspxFactory = JspFactory.getDefaultFactory();
      response.setContentType("text/html;charset=UTF-8");
      pageContext = _jspxFactory.getPageContext(this, request, response,
                  null, true, 8192, true);
      _jspx_page_context = pageContext;
      application = pageContext.getServletContext();
      config = pageContext.getServletConfig();
      session = pageContext.getSession();
      out = pageContext.getOut();
      _jspx_out = out;

      out.write("\n");
      out.write("\n");
      out.write("\n");
      out.write("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01
Transitional//EN\"\n");
      out.write("   \"http://www.w3.org/TR/html4/loose.dtd\">\n");
      out.write('\n');
      out.write("\n");
      out.write("\n");
      out.write("<html>\n");
      out.write("    <head>\n");
      out.write("        <meta http-equiv=\"Content-Type\"
content=\"text/html; charset=UTF-8\">\n");
      out.write("        <title>Client List</title>\n");
      out.write("    </head>\n");
      out.write("    <body>\n");
      out.write("\n");
      out.write("\n");
      out.write("\n");
      out.write("<h1>MegaCompany Ltd.</h1>\n");
      out.write("<hr>");
      out.write("\n");
      out.write("    <h1>All Clients</h1>\n");
      out.write("    <table border=\"1\">\n");
      out.write("        <thead>\n");
      out.write("            <tr>\n");
      out.write("                <th>Name</th>\n");
      out.write("                <th>Place</th>\n");
      out.write("                <th>Email</th>\n");
      out.write("            </tr>\n");
      out.write("        </thead>\n");
      out.write("        <tbody>\n");
      out.write("            ");
      out.write("\n");
      out.write("            ");
  // получаем список клиентов из DAO-класса
```

```
                Collection<Client> clients = new
ClientDAO().findAllClients();
                for (Client client : clients) {
                    pageContext.setAttribute("client", client);

      out.write("\n");
      out.write("            <tr>\n");
      out.write("                <!-- полное имя клиента -->\n");
      out.write("                <td><a href='");
      out.print( response.encodeURL("infoClient.jsp"));
      out.write("?id=");
      out.print( client.getId() );
      out.write("'>\n");
      out.write("                    ");
      out.print( getClientFullName(client) );
      out.write("</a></td>\n");
      out.write("                <td>");
      out.write((java.lang.String)
org.apache.jasper.runtime.PageContextImpl.proprietaryEvaluate("${client.place
}", java.lang.String.class, (PageContext)_jspx_page_context, null, false));
      out.write("</td>\n");
      out.write("                ");
      out.write("\n");
      out.write("                <td>");
      out.print( client.getEmail() );
      out.write("</td>\n");
      out.write("            </tr>\n");
      out.write("            ");
 }
      out.write("\n");
      out.write("        </tbody>\n");
      out.write("    </table>\n");
      out.write("\n");
      out.write("    </body>\n");
      out.write("</html>\n");
    } catch (Throwable t) {
      if (!(t instanceof SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
          out.clearBuffer();
        if (_jspx_page_context != null)
_jspx_page_context.handlePageException(t);
      }
    } finally {
      if (_jspxFactory != null)
_jspxFactory.releasePageContext(_jspx_page_context);
    }
  }
}
```

Таблица 7.1

Встроенные объекты

Название	Назначение	Тип	Область видимости
`request`	Запрос	`javax.servlet.ServletRequest`[1]	request[2]
`response`	Ответ	`javax.servlet.ServletResponse`[3]	page
`session`	Веб-сессия. Объект доступен, только если для взаимодействия с JSP используется протокол HTTP и JSP-страница участвует в сессии (см. атрибут `session` директивы `page`)	`javax.servlet.http.HttpSession`	session
`application`	Контекст веб-приложения	`javax.servlet.ServletContext`	application
`config`	Конфигурационные параметры JSP-страницы, указанные в установочном дескрипторе	`javax.servlet.ServletConfig`	page
`pageContext`	Контекст текущей JSP-страницы	`javax.servlet.jsp.PageContext`	page
`out`	Используется для генерации вывода в ответ, формируемый текущей JSP-страницей. Данный объект необходимо использовать при необходимости программной генерации части ответа в скриптлете или обработчике тэга для визуального custom action[4]	`javax.servlet.jsp.JspWriter`	page
`page`	Соответствует экземпляру PIC (фактически это ссылка на экземпляр сервлета, реализующего данную JSP-страницу, эквивалентная ссылке `this`)	`java.lang.Object`	page
`exception`	Объект исключения, выброшенного другим веб-компонентом, в результате чего за счет механизма страниц обработки ошибок запрос был перенаправлен на данную JSP-страницу. Объект доступен, только если атрибут `isErrorPage` директивы `page` в данной JSP-странице равен `true`.	`java.lang.Throwable`	page

1. Один из потомков типа `javax.servlet.ServletRequest` в зависимости от протокола, используемого для взаимодействия с JSP. Для протокола HTTP это тип `javax.servlet.http.HttpServletRequest`.
2. Объект `request` не хранится как атрибут запроса, так как сам представляет собой запрос.
3. Один из потомков типа `javax.servlet.ServletResponse` в зависимости от протокола, используемого для взаимодействия с JSP. Для протокола HTTP это тип `javax.servlet.http.HttpServletResponse`.
4. Не следует для этих целей использовать объекты типа `PrintWriter` или `OutputStream`, которые можно попытаться получить из встроенного объекта `response` методами `getPrintWriter()` и `getOutputStream()`, соответственно.