top
КОМПЬЮТЕРНАЯ АКАДЕМИЯ

C++

</>

C++

ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ

JAVA

Урок № 7

JavaServer Pages,
Tags в JSP, cookies,
session и другое
в JSP

Содержание

1. Что такое JSP? История возникновения JSP

В предыдущем уроке были рассмотрены сервлеты, Вы также узнали о запросах обработки, ответах, параметрах запросов. Java — весьма мощный язык, он имеет множество возможностей, которые делают его полезным, гибким и простым в использовании.

Рассмотрим пример, который иллюстрирует формирование HTML-содержимого в Java-строках:

```
PrintWriter writer = response.getWriter();
writer.append("<!DOCTYPE html>\r\n")
      .append("<html>\r\n")
      .append("    <head>\r\n")
      .append("        <title>Poem App</title>\r\n")
      .append("    </head>\r\n")
      .append("    <body>\r\n")
      .append("        Speech: \"To be, or not to be,
                      that is the question\"\r\n")
      .append("    </body>\r\n");
      .append("</html>\r\n");
```

Этот способ является весьма громоздким и неудобным. Для получения результата необходимо написать сравнительно много кода, а для его хранения требуется дополнительное пространство. Также, во время написания и тестирования такого кода много времени тратится впустую. Например, для того чтобы в браузере было можно получить читабельный исходный HTML-код —

необходимо применять многострочность с использованием символа окончания строки (\r\n). Любые кавычки в HTML, должны быть экранированы так, чтобы они преждевременно не завершали строковый литерал. И, возможно, одна из худших проблем — редакторы кода не могут распознавать и проверять HTML-код в строках, чтобы сообщить программисту об ошибках. Конечно же, есть лучший способ.

Предыдущий пример в обычном HTML-файле, выглядит так:

```
<!DOCTYPE html>
<html>
    <head>
        <title> Poem App </title>
        <style>
            q {
                  quotes: "\0022" "\0022";
              }
        </style>
    </head>

    <body>
        Speech: <q>To be, or not to be,
                    that is the question</q>
    </body>
</html>
```

Таким образом, создатели спецификации Java EE поняли, что эта система быстро станет громоздкой и спроектировали *JavaServer Pages* (JSP), чтобы избежать этой проблемы.

И в самом деле, когда речь идет о многоуровневой разработке Java EE, первое, что должно прийти на ум это

JavaServer Pages (JSP). Фактически, JSP используются для большинства приложений Java EE. Если создается динамическое веб-приложение, то есть вероятность того, что используется технология JSP. Это означает, что JSP может использоваться как в самых простых приложениях, так и в самых сложных и требовательных.

2. Цели и задачи JSP

Правильное использование JSP позволяет выполнить чистую реализацию паттерна Model-View-Controller, позволяющую четко отделить представление от бизнес-логики. Эта технология также позволяет считывать, повторно использовать и обслуживать страницы, то есть, в принципе, мы имеем что-то более читабельное, чем сервлет.

JSP позволяет использовать код для разметки. Цель JSP-технологии состоит в том, чтобы отделить контент от логики, и позволить «не программистам» создавать необходимые страницы, которые содержат пользовательские теги. JSP относительно просты в сборке и предоставляют полный набор Java API.

Проблема в самом последнем примере заключается в том, что это статический HTML-документ. Возможно, так было-бы проще писать, и, вероятно, будет бесконечно легче поддерживать, чем пример, написанный на Java, но здесь нет ничего динамичного. JSP — это, по сути, гибридное решение, сочетающее Java-код и HTML-теги. JSP могут содержать любой HTML-тег в дополнение к Java-коду, встроенным JSP-тегам, пользовательским JSP-тегам и тому, что называется языком выражений (*Expression Language*).

Следующий пример, использует JSP вместо сервлета для отображения цитаты из известной поэмы.

```
<%@ page contentType="text/html; charset=UTF-8"
language="java" %>
<!DOCTYPE html>
```

```
<html>
    <head>
        <title> Poem App </title>
        <style>
            q {
                quotes: "\0022" "\0022";
              }
        </style>
    </head>

    <body>
        Speech: <q>To be, or not to be,
                that is the question</q>
    </body>
</html>
```

Этот пример почти идентичен предыдущему HTML-примеру. Единственное отличие — первая строка. Это одна из нескольких директив JSP, в этой директиве задается тип содержимого и кодировка символов на странице. То, что было можно сделать с использованием методов setContentType и setCharacterEncoding HttpServletResponse. Все остальное в этом JSP — это простой HTML, переданный клиенту «как есть» в ответе.

3. Что происходит с JSP во время выполнения

Скорей всего, ранее Вы уже слышали фразу «синтаксический сахар». Так вот, JSP — это и есть одна из форм синтаксического сахара. Во время выполнения JSP-код интерпретируется JSP-компилятором, который в свою очередь разбирает все специальные функции в JSP-коде и переводит их в Java-код. Класс Java, созданный из каждого JSP, реализует сервлет. Затем код Java проходит тот же цикл, который обычно выполняется. Еще во время выполнения он компилируется в байт-код, а затем в машинный код. Наконец, JSP перенаправляет сервлет в ответ на запросы, как любой другой сервлет.

Для того чтобы понять это, необходимо выполнить следующие действия:

1. Создайте и скомпилируйте проект Poem-JSP в своей среде IDE и откройте браузер по ссылке *http://localhost:8080/poem/*. На экране можно увидеть знакомую цитату.
2. Откройте каталог Tomcat (*C:\Program Files\Apache Software Foundation\Tomcat* в Windows) и перейдите в папку *\Catalina\localhost\poem*. Tomcat помещает все скомпилированные JSP для приложения в этот каталог, кроме того, он оставляет за собой промежуточные файлы Java, которые он создает для просмотра.
3. Найдите файл *index_jsp.java* и откройте его (не файл *index_jsp.class*) в текстовом редакторе.

Найдем класс, который наследуется от org.apache.jasper.runtime.HttpJspBase. Этот абстрактный класс наследуется от HttpServlet. HttpJspBase предоставляет некоторые базовые функции, которые будут использоваться всеми JSP. Они компилируются Tomcat, и когда JSP выполняется — выполняется метод службы на этом сервлете, который, в конечном итоге, выполняет метод _jspService.

Если Вы просмотрите метод _jspService, то найдете серию вызовов методов, записывающих HTML в выходной поток. Этот код должен быть Вам знаком, потому что он не отличается от кода Java, который Вы заменили этим JSP. Разумеется, класс JSP сервлет не выглядит одинаково на каждом веб-контейнере. Классы org.apache.jasper, например, являются классами, специфичными для Tomcat. Ваш JSP компилируется по-разному на каждом веб-контейнере, в котором Вы его запускаете. Важным моментом является то, что существует стандартная спецификация поведения и синтаксиса JSP, и пока используемые веб-контейнеры соответствуют спецификации, JSP должны работать одинаково на всех из них, даже если они интерпретировали код Java по-разному.

JSP, как и обычные сервлеты, также можно отладить во время выполнения. Для этого необходимо поместить точку останова на строку в JSP, содержащую «Speech…», а затем обновить браузер. Когда отладчик попадет в точку останова в JSP — Вы должны заметить несколько вещей. Во-первых, Вы можете устанавливать точки останова непосредственно в коде JSP — Java, Tomcat и IDE могут соответствовать точке останова в JSP для отладки кода во время выполнения. Вы также должны заметить, что,

хотя точка останова может быть в коде JSP, отладчик явно этого не отображает. Стек показывает, что время выполнения приостановлено в методе _jspService, а окно переменных отображает все экземпляры и локальные переменные, определенные в этой области в классе index_jsp.

Как и все другие сервлеты, запущенные в веб-контейнере, JSP имеют жизненный цикл. В некоторых веб-контейнерах, таких как Tomcat, JSP преобразовывается и компилируется как раз во время, когда приходит первый запрос на этот JSP. Для будущих запросов, JSP уже скомпилирован и готов к использованию. Это влияет на производительность. Хотя удар по производительности, как правило, приходит только по первому запросу, все последующие запросы работают на приличной скорости, это все еще нежелательно в некоторых производственных средах. Из-за этого многие веб-контейнеры предоставляют возможность предварительно скомпилировать все JSP-приложения при развертывании. Это, конечно же, значительно увеличивает время развертывания для крупных приложений. Если у Вас много тысяч JSP, для развертывания приложения потребуется, например, десять минут, а не одна. Какую конфигурацию использовать, зависит от организации-заказчика и ее потребностей. Независимо от времени компиляции, после поступления первого запроса, сервлет JSP будет создан и инициализирован, а затем первый запрос может быть обслужен.

К этому моменту нужно понимать, что код, который, написанный в JSP, в конечном счете, переведен в некоторую версию кода, который пришлось бы писать в любом случае, если бы не было JSP. Возникает закономерный во-

прос, почему стоит беспокоиться о JSP? Факт остается фактом: JSP — гораздо более простой формат файла для создания разметки и для отображения в веб-браузере, чем прямой Java-код. Если это может повысить скорость, эффективность и точность процесса разработки, на самом деле возникает вопрос: «Почему бы не использовать JSP?»

4. Создание первого JSP

После того как были изучены сервлеты. Необходимо ответить на вопрос «Что должно происходить в методе service?». Метод service класса Servlet обслуживает все входящие запросы. Он должен анализировать и обрабатывать данные по входящему запросу на основе используемого протокола, а затем возвращать ответ клиенту. Если метод service возвращает ответ обратно в сокет без данных, клиент, скорее всего, получит сетевую ошибку, например «сброс соединения».

В HTTP протоколе метод service должен понимать заголовки и параметры, которые клиент отправляет и затем возвращать валидный HTTP-ответ, который включает в себя минимальные заголовки HTTP, но поскольку HttpServlet заботится обо всем этом, методы doGet и doPost могут быть буквально пустыми. Во время выполнения JSP должно произойти много вещей, но все эти «должны» обрабатываются за нас.

Чтобы продемонстрировать это, создадим файл с именем *blank.jsp* в корневом каталоге пустого проекта. Удалим все его содержимое (IDE может поместить туда какой-то код). Произведем redeploy нашего проекта. После перехода к *http://localhost:8080/poem/blank.jsp*, не получим никаких ошибок. Все работает нормально, мы просто получаем бесполезную пустую страницу. Теперь добавим в него следующий код, произведем redeploy и перезагрузим страницу:

```
<!DOCTYPE html>
<html>
    <head>
        <title> Poem App </title>
        <style>
            q {
                quotes: "\0022" "\0022";
            }
        </style>
    </head>

    <body>
        Speech: <q>To be, or not to be,
                that is the question</q>
    </body>
</html>
```

Сейчас существует только небольшая разница между *blank.jsp* и *index.jsp* — отсутствующий специальный тег, который находится в первой строке *index.jsp*. И все же, контент все еще отображается одинаково. Это связано с тем, что JSP по умолчанию имеют тип содержимого text/html и кодировку символов ISO-8859-1. Однако, эта кодировка по умолчанию, несовместима со многими специальными символами, которые не содержатся в английском языке, что может помешать локализовать приложение. Итак, как минимум, JSP должен содержать HTML для отображения информации пользователю. Тем не менее, чтобы убедиться, что HTML отображается правильно во всех браузерах во всех системах на многих языках, необходимо включить определенные теги JSP для управления данными, отправленными клиенту, такими как установка кодировки UTF-8.

В JSP можно использовать несколько различных типов тегов. Один из типов тегов директивы Вы уже видели:

```
<%@ page ... %>
```

Этот тег директивы предоставляет некоторые элементы управления передачей JSP клиенту, отображением и обратной передачей. В примере *index.jsp* директива page выглядит следующим образом:

```
<%@ page contentType="text/html;charset=UTF-8"
language="java" %>
```

Атрибут language сообщает контейнеру, какой JSP-скриптовый язык используется для этого JSP. Язык сценариев JSP (не путайте с интерпретируемыми языками сценариев) — это язык, который может быть встроен в JSP для написания определенных действий.

Технически, можно опустить этот атрибут. Поскольку Java является единственным поддерживаемым языком сценариев JSP, и используется по умолчанию в спецификации. Атрибут contentType сообщает контейнеру значение заголовка Content-Type, которое должно быть отправлено обратно вместе с ответом. Заголовок Content-Type содержит как тип содержимого, так и кодировку символов, разделенных точкой с запятой. Если Вы помните, в файле *index_jsp.java*, содержится:

```
response.setContentType("text/html;charset=UTF-8");
```

Следует отметить, что предыдущий фрагмент кода является эквивалентом следующих двух строк:

```
response.setContentType("text/html");
response.setCharacterEncoding("UTF-8");
```

И, более того, они эквивалентны следующей строке кода:

```
response.setHeader("Content-Type",
                   "text/html;charset=UTF-8");
```

Таким образом, нужно отметить, что существует несколько способов выполнения одной и той же задачи. Методы setContentType и setCharacterEncoding являются удобными. Какой вариант использовать — зависит от Вас. Как правило, необходимо выбрать один и придерживаться его, чтобы избежать путаницы. Однако, поскольку большая часть кода будет основана на JSP, в основном мы будем иметь дело только с атрибутом contentType директивы page.

5. Комментирование кода

Как и в любом другом языке, в JSP есть способы комментирования кода:

1. Комментарии XML (также известный как комментарий HTML):

```
<!-- это комментарий HTML/XML -->
```

   Этот тип комментариев передается клиенту, поскольку он является стандартной разметкой XML и HTML. Браузер игнорирует его, но он появляется в источнике ответа. Теги JSP в этом комментарии будут по-прежнему вычислены. Это важно помнить, потому что комментирование кода таким способом не препятствует исполнению кода Java внутри.

2. Однострочный и блочный комментарии Java:

```
// строчный комментарий
/* блочный комментарий */
```

   Вы можете использовать любой Java комментарий в объявлениях и сценариях в JSP, внутри которых код не будет выполняться.

3. Комментарии JSP:

```
<%-- это комментарий JSP --%>
```

   Как и в комментарии XML/HTML, все, что внутри скобок считается закомментированным. Он не толь-

ко не отправляется в браузер, он полностью игнорируется компилятором JSP.

Если первые два типа комментариев содержатся в java-файле JSP Servlet, то, последний — нет. Это особенно полезно для комментирования некоторого кода, который включает в себя сценарии, выражения, декларации, директивы и разметку, которые Вы не хотите отправлять в браузер.

6. Добавление import в JSP

Если в JSP Java-код использует класс напрямую, необходимо либо ссылаться на него, используя его полное имя класса, либо включать директиву импорта в JSP-файл. И так же, как каждый класс в пакете *java.lang* импортируется неявно в файлы Java, каждый класс пакета *java.lang* неявно импортируется в JSP-файлы. Для того чтобы импортировать один или несколько классов необходимо добавить атрибут импорта в директиву страницы:

```
<%@ page import = "java.util.*, java.io.IOException" %>
```

В этом примере используется запятая для разделения нескольких импортов, и в результате импортируется класс java.io.IOException и все члены пакета *java.util*. Не обязательно использовать отдельную директиву для импорта классов, можно комбинировать с другими атрибутами:

```
<%@ page contentType="text/html;charset = UTF-8"
    language="java"
    import="java.util.*,java.io.IOException" %>
```

Можно также использовать несколько директив:

```
<%@ page import="java.util.Map" %>
<%@ page import="java.util.List" %>
<%@ page import="java.io.IOException" %>
```

Каждый тег JSP, который ничего не выводит, а также директивы, декларации и сценарии, приводят к пу-

стым строкам при отображении клиенту. Таким образом, если у Вас есть много директив, за которыми следуют различные объявления и сценарии, Вы можете получить десятки пустых строк. Чтобы компенсировать это, разработчики JSP часто связывают конец одного тега с началом следующего:

```
<%@ page import="java.util.Map"
%><%@ page import="java.util.List"
%><%@ page import="java.io.IOException" %>
```

Этот пример кода дает тот же результат, что и предыдущий, но в результате он выводит только одну пустую строку вместо трех.

7. Понятие директива, объявление, сценарий и выражение

Кроме различных тегов HTML и JSP, которые можно использовать в JSP, существует также несколько уникальных структур, которые определяют своего рода JSP-язык. Это директивы, объявления, сценарии и выражения.

```
<%@ директива %>
```

Директивы используются, для того чтобы указать интерпретатору JSP, какое действие он должен выполнить (например, установить тип содержимого) или сделать предположение о файле (например, какой язык скриптинга он использует), импортировать класс, включить другой JSP во время преобразования или включения библиотеки тегов JSP.

```
<%! объявление %>
```

Объявления используются для того чтобы определить что-то в рамках класса JSP Servlet. Например, переменные экземпляра, методы или классы в теге объявления. Необходимо также помнить, что все классы, которые определяются, на самом деле являются внутренними классами класса JSP Servlet, так как все они объявлены в созданном классе JSP Servlet.

```
<% сценарий %>
```

Сценарий, так же как и объявление, содержит Java-код. Тем не менее, сценарии имеют разную область видимости. Код в объявлении копируется в тело класса JSP Servlet во время преобразования и поэтому должен использоваться для объявления какого-то поля или метода, сценарии же копируются в тело метода _jspService. Любые локальные переменные, которые находятся в области действия этого метода, будут находиться в пределах области сценариев, и любой код, который видимый в пределах тела метода, является видимым в сценарии.

Таким образом, можно определить локальные переменные, но не поля экземпляров. Можно использовать условные операторы, манипулировать объектами и выполнять арифметику, все, что невозможно сделать в объявлении. Можно даже определить классы, но классы не будут иметь области видимости вне метода _jspService. Класс, метод или переменная, определенные в объявлении, могут использоваться в сценарии, но класс или переменная, определенные в сценарии, не могут использоваться в объявлении.

```
<%= выражение %>
```

Выражения содержат простой Java-код, который что-то возвращает клиенту. Например, арифметический расчет внутри выражения, в результате которого будет возвращено и отображено числовое значение. По существу, любой код, который может находиться справа от оператора присваивания, может быть помещен в выражение. Выражения выполняются в рамках того же метода, что и сценарии, то есть выражения будут скопированы в метод _jspService.

В качестве примера рассмотрим следующий код, который содержит директивы, объявления, сценарии и выражения.

```
<%@ page contentType="text/html;charset=UTF-8"
    language="java" %>
<%!
    protected String str = "simple string";
    // Описание ниже, не является объявлением
    // и приведет к синтаксической ошибке
    // str = "test string";

    private final float a = 1.11F;
    public float addNum(float num)
    {
        return num + 1.5F;
    }

    public class ExampleInnerClass {  }
    ExampleInnerClass obj1 = new ExampleInnerClass();
    // ExampleClass имеет другую область видимости,
    // поэтому объявление ниже является
    // синтаксической ошибкой
    // ExampleClass obj2 = new ExampleClass();
%>
<%
    class ExampleClass    {    }

    ExampleClass obj3 = new ExampleClass();

    ExampleInnerClass obj4 = new ExampleInnerClass();
    long b;
    b = 7L;
%>
<%= "To be, or not to be, that is the question" %>
<br/>
<%= addNum(a) %>
```

Для того чтобы собрать проект, необходимо создать JSP-файл с именем *example.jsp* в корневом каталоге пустого проекта, и поместить в него код из предыдущего примера. Затем, нужно скомпилировать и запустить приложение, перейти по адресу *http://localhost:8080/poem/example.jsp*.

Чтобы лучше понять отличия директив, деклараций, сценариев и выражений, необходимо найти файл *example_jsp.java* в рабочем каталоге Tomcat (*\Tomcat\work\Catalina\localhost\poem \org\apache\jsp*). В классе JSP Servlet можно увидеть, как код из JSP был преобразован в Java-код.

Использование Директив

Существует три разных типа директив.

Директива свойств страницы

Директива страницы предоставляет нам элементы управления тем, как JSP интерпретируется, отображается и передается обратно клиенту. Рассмотрим некоторые из атрибутов, которые могут быть включены в эту директиву:

Атрибут	Значение по умолчанию	Описание
pageEncoding	ISO-8859-1	Определяет кодировку символов JSP.
session	True	Определяет участвует ли JSP в сессиях HTTP. Если true — будет предоставлен доступ к сессии в JSP, в противном случае — нельзя использовать сессии. Если приложение не использует сессии, и нужно повысить производительность — установите false.

Атрибут	Значение по умолчанию	Описание
isELIgnored	до JSP 2.0 — true с JSP 2.0 — false	Определяет, вычисляется ли EL для JSP.
buffer	8kb или none	Определяет размер выходного буфера JSP, если none выход происходит непосредственно в объект.
autoFlush	True	Определяет, будет ли буфер автоматически очищаться после достижения предела размера. Если значение false — при переполнении буфера возникает исключение.
errorPage	URL	Если во время выполнения JSP возникает ошибка, этот атрибут указывает контейнеру, на какой JSP переслать запрос.
isErrorPage	False	Определяет, является ли JSP страницей для обработки ошибок.
isThreadSafe	True	Сообщает контейнеру, что JSP может безопасно одновременно обслуживать несколько запросов. Если значение false — контейнер обслуживает только запросы к этому JSP. Лучше не изменять, так как JSP должен быть потоко-безопасным.
extends	полное имя расширяемого класса	Задает суперкласс для генерируемого сервлета.
info	Текст	Сообщение, которое можно прочитать методом getServletInfo().

Директива включения других JSP

Первым инструментом, который можно использовать для включения JSP в JSP, является include директивы.

```
<%@ include file="/path/to/example/file.jsp" %>
```

File — предоставляет контейнеру путь к JSP-файлу, который должен быть включен. Может быть абсолютным или относительным. Директива include включает файл на этапе компиляции. Прежде чем JSP будет преобразован в Java, директива include заменяется содержимым включенного JSP-файла. После этого комбинированное содержимое преобразуется в Java и компилируется. Таким образом, этот процесс является статическим и происходит только один раз.

Существует еще один способ включить другие JSP, которые приводят к динамическому включению вместо статического. Для этого используется тег <jsp: include>

```
<jsp:include page="/path/to/example/page.jsp" />
```

Тег <jsp:include> не имеет атрибут file, он имеет атрибут page. Путь по-прежнему может быть как абсолютным, так и относительным. Но он не включен во время компиляции. Вместо этого включенный файл компилируется отдельно. Во время выполнения запрос временно пересылается во включенный JSP, результирующий вывод этого JSP записывается в ответ, а затем элемент управления возвращается обратно к включенному JSP.

Оба этих метода включения имеют свои сильные и слабые стороны. Директива include является быстрой, потому что она компилируется только один раз, и все объявленные переменные в JSP находятся в одной области видимости со включенными JSP и могут на них ссылаться. Но при использовании данного метода, JSP файл становится больше, и важно помнить, что байт-код скомпилированных Java-методов не может превышать 65534 байта.

С тегом <jsp:include> не возникает такой проблемы, но он должен компилироваться при каждой загрузке страницы. Переменные, определенные во включенных JSP, недоступны и не могут быть использованы в текущем JSP.

Включение библиотеки тегов

Для того чтобы в JSP было можно использовать теги, определенные другой библиотекой необходимо применить директиву taglib для ссылки на нее:

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

Атрибут uri определяет пространство имен URI в котором определена библиотека тегов, а атрибут prefix определяет псевдоним, с которым Вы ссылаетесь на теги в этой библиотеке.

8. Использование <jsp> тегов

Когда Вы пишете JSP, обратите внимание, что одна библиотека тегов уже неявно включена для использования во всех ваших JSP. Это библиотека тегов JSP (префикс jsp), и Вам не нужно размещать taglib в JSP, чтобы использовать ее. Но в документе JSP, Вам нужно добавить объявление XMLNS для библиотека тегов jsp.

Рассмотрим стандартные <jsp> теги:

- Тег <jsp:forward> позволяет пересылать запрос от JSP, который он в настоящее время выполняет в другом JSP. В отличие от <jsp:include> , запрос не возвращается к исходному JSP. Это не перенаправление, браузер клиента не видит изменения. Кроме того, все, что JSP пишет для ответа, остается в ответе, когда происходит пересылка. Он не стирается, как это было бы с перенаправлением.

  Использование тега <jsp:forward>:

```
<jsp:forward page="/example/other/page.jsp"/>
```

В этом примере запрос внутренне перенаправляется на */example/other/page.jsp*. Любой контент ответа, созданный до тега, по-прежнему поступает в браузер клиента. Любой код, который появляется после тега, игнорируется.

- Тег <jsp:plugin> является удобным инструментом для встраивания Java-апплетов в обработанный HTML. Этот тег устраняет риск испортить тщательную структуру

тегов OBJECT и EMBED, необходимых для того, чтобы Java-апплеты могли работать во всех браузерах. Он обрабатывает создание этих HTML-тегов, так чтобы апплет работал во всех браузерах, поддерживающих плагин Java.

Ниже приведен пример использования тега <jsp:plugin>:

```
<jsp:plugin type="applet" code="MyApplet.class"
     jreversion="1.8">
<jsp:params>
<jsp:param name="appletParam1" value="paramValue1"/>
</jsp:params>
<jsp:fallback>
```

Ваш браузер не поддерживает Java-апплеты. Пожалуйста, смените свой браузер.

```
</jsp:fallback>
</jsp:plugin>
```

Обратите внимание, что <jsp:plugin> также может содержать стандартные атрибуты OBJECT и EMBED HTML, такие как name, align, height, width, hspace и vspace. Эти атрибуты копируются в разметку HTML.

9. Компонент JavaBean

Что такое JavaBean. Цели и задачи

Компонентами JavaBeans являются классы Java, которые необходимо многократно использовать. Они позволяют программистам значительно ускорить процесс разработки веб-приложений методом их сборки из программных компонентов. Различные компонентные технологии, и JavaBeans в том числе, повлекли за собой появление нового типа программирования — сборки приложений из компонентов. Где программисту нужно знать всего лишь сервисы компонентов — нюансы реализации компонентов совершенно не важны.

Возможности Bean не ограничены: он может выполнять простую функцию, такую как проверка орфографии документа, или более сложную — например, прогнозирование эффективности портфеля акций. Bean может быть, как видимым для конечного пользователя — кнопка на графическом пользовательском интерфейсе, так и невидимым — программное обеспечение для декодирования потока мультимедийной информации в режиме реального времени. Также Bean может быть создан для локального использования на компьютере пользователя или для совместной работы с набором других распределенных компонентов. Программное обеспечение для создания круговой диаграммы из набора точек данных, является примером компонента, который может выполняться локально. Однако, Bean который пре-

доставляет информацию о ценах на фондовой бирже в реальном времени, должен взаимодействовать с другим распределенным программным обеспечением для получения своих данных.

Технология JavaServer Pages напрямую поддерживает использование компонентов JavaBeans со стандартными элементами языка JSP. Вы можете легко создавать и инициализировать компоненты, а также получать и устанавливать значения своих свойств.

Соглашения о разработке компонентов JavaBeans определяют свойства класса и публичные методы, которые предоставляют доступ к свойствам.

Свойство не должно быть реализовано переменной экземпляра. Оно должно быть доступно с использованием публичных методов, которые соответствуют следующим соглашениям:

```
PropertyClass getProperty () {...}
setProperty (PropertyClass pc) {...}
```

В дополнение к методам свойств компонент JavaBeans должен определять конструктор, который не принимает никаких параметров.

Существуют следующие теги для работы с JavaBeans:

- **<jsp:useBean>** тег объявляет наличие JavaBean на странице.
- **<jsp:getProperty>** получение свойств (метод get) из объявленных JavaBeans с использованием <jsp:useBean>.
- **<jsp:setProperty>** установка свойств (метод set). JavaBean в этом случае является любым экземпляром объекта.

- **<jsp: useBean>** создает экземпляр класса для создания компонента, и этот компонент может быть доступен с помощью двух других bean-тегов (setProperty, getProperty).

Преимущество для объявления компонента, таким образом, заключается в том, что он делает JavaBeans доступными для других тегов JSP. Если Вы просто объявили JavaBean-компонент в сценарии, он будет доступен только для сценариев и выражений.

Пример использования

Рассмотрите пример компонента JavaBean:

```
package database;
public class CardDB
{
    private String clientName;
    public CardDB() { }
    public String getClientName()
    {
        return clientName;
    }

    void setClientName(String clientName)
    {
        this.clientName = clientName;
    }
}
```

JavaBean CardDB имеет свойство clientName, которое доступно и для чтения и для записи.

В дополнение к методам свойств компонент JavaBeans должен определять конструктор по умолчанию.

Используя метод <jsp:useBean> создадим экземпляр класса CardDB.

Свойство clientName установим и тут же получим с помощью тегов <jsp:setProperty> и <jsp:getProperty> соответственно:

```
<jsp:useBean id="cardDB" class="database.CardDB"
    scope="page" >
<jsp:setProperty name="cardDB" property="clientName"
    value="Ivanov" />
<jsp:getProperty name="cardDB" property="clientName" />
</jsp:useBean>
```

Где:

<jsp:useBean>

- **id="cardDB"** — идентификатор объекта;
- **class="database.CardDB"** — полное имя класса реализации объекта;
- **scope="page"** — область видимости объекта;

<jsp:setProperty>, <jsp:getProperty>;

- **name="cardDB"** — идентификатор объекта;
- **property="clientName"** — имя свойства;
- **value="Ivanov"** — новое значение свойства.

10. Что такое JSP Fragment?

По умолчанию веб-контейнеры преобразовывают и компилируют файлы, заканчивающиеся на *.jsp* и *.jspx* как JSP. Так же существует расширение *.jspf*. Файлы JSPF обычно называются фрагментами JSP и не компилируются веб-контейнером. Хотя нет жестких правил, регулирующих файлы JSPF, можно технически настроить большинство веб-контейнеров для их компиляции, если необходимо. Файлы JSPF представляют собой фрагменты JSP, которые не могут существовать самостоятельно и всегда должны быть включены, а не доступны напрямую. Вот почему веб-контейнеры обычно не компилируют их. Фактически, во многих случаях файл JSPF ссылается на переменные, которые могут существовать только в том случае, если они включены в другой JSP-файл. По этой причине файлы JSPF должны быть включены только с помощью директивы include, потому что переменные, определенные во включении JSP, должны быть включены в состав включенного JSP.

Веб-приложение обычно содержит раздел навигации, основное содержание и нижний колонтитул веб-страницы. Использование директивы include упрощает сохранение фрагмента веб-страницы, и когда нам нужно изменить раздел нижнего колонтитула, нам просто нужно изменить файл нижнего колонтитула, и вся страница, которая включает его, будет изменена.

Включение страницы с использованием директивы include будет происходить во время выполнения страницы, когда JSP транслирован в сервлет с помощью кон-

тейнера JSP. Мы можем использовать любое имя расширения файла для JSP Fragment, используемого директивой include.

Рассмотрим пример включения JSP Fragment в страницу JSP с директивой include. В этом примере мы используем расширение *.jspf*, которое является сокращением JSP Fragment.

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

    <body>
        <div id="header">
            <%@ include file=
                        "/example/other/header.jspf" %>
        </div>
        <div id="content">
            Content
        </div>
        <div id="footer">
            <%@ include file=
                       "/example/other/footer.jspf" %>
        </div>
    </body>
</html>
```

Содержимое файла *header.jspf*:

```
Header
<hr/>
```

Содержимое файла *footer.jspf*:

```
<hr/>
Footer
```

Более детальное применение файла JSPF рассмотрим в разделе Model View Controller.

11. Обработка ошибок в JSP

Вы уже знаете что, используя директиву page, можно управлять различными параметрами выполнения страницы JSP. Рассмотрим директивы, которые относятся к ошибкам буферизации и обработке ошибок.

Когда выполняется страница JSP, вывод, записанный в объект ответа, автоматически буферизуется. Вы можете установить размер буфера, используя директиву page:

```
<%@ page buffer="none|xxxkb" %>
```

Чем больше буфер, тем больше он позволяет записать контента до того, как что-либо действительно будет отправлено обратно клиенту, тем самым предоставив странице JSP больше времени для установки соответствующих кодов состояния и заголовков или для перехода на другой веб-ресурс. Меньший буфер уменьшает нагрузку на серверную память и позволяет клиенту быстрее получать данные.

При выполнении страницы JSP может возникнуть разное количество исключений. Для того чтобы указать, что веб-контейнер должен перенаправлять элемент управления на страницу с ошибкой, в том случае если возникает исключение, необходимо включить следующую директиву page в начале страницы JSP:

```
<%@ page errorPage="filename" %>
```

Предположим что страница */example/errorpage.jsp* содержит директиву page:

```
<%@ page errorPage="errorpage.jsp" %>
```

Атрибут errorPage позволяет передать исключение для обработки страницы JSP, которую предварительно создал программист. У этой страницы-обработчика исключения атрибут isErrorPage равен true:

```
<%@ page isErrorPage="true"%>
```

12. Model View Controller

Что такое Model View Controller? Цели и задачи Model View Controller

Model View Controller (MVC) — это паттерн, используемый в разработке программного обеспечения для отделения бизнес-логики приложения от пользовательского интерфейса. Как следует из названия, паттерн MVC имеет три уровня.

Модель определяет бизнес-уровень приложения, контроллер управляет потоком приложения, а представление определяет уровень представления приложения.

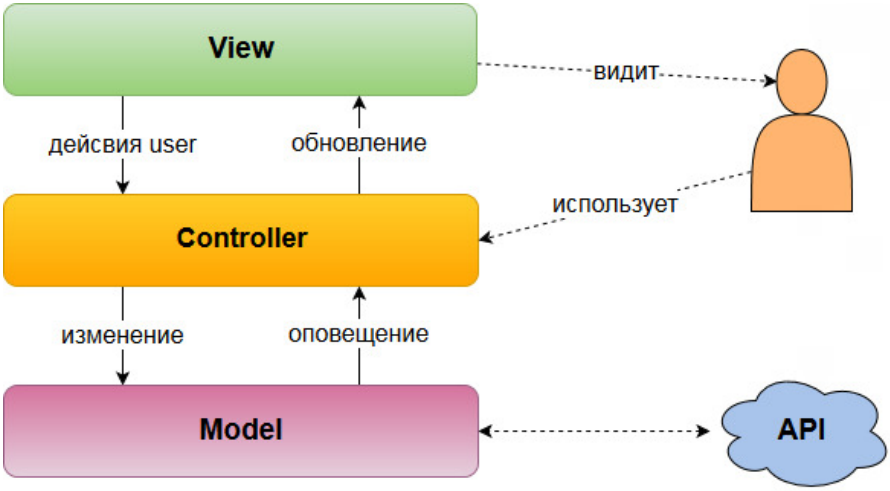

Рисунок 1.

Хотя паттерн MVC не является специфичным для веб-приложений, он очень хорошо подходит для прило-

жений такого типа. В контексте Java модель состоит из простых классов Java, контроллер состоит из сервлетов, а представление состоит из страниц JSP.

Ключевые особенности шаблона:

- отделяет слой представления от бизнес-уровня;
- контроллер выполняет действие вызова модели и отправки данных в представление;
- модель даже не знает, что она используется каким-либо веб-приложением или десктопным приложением.

Давайте посмотрим на каждый слой:

1. **Модель**. Это слой данных, который содержит бизнес-логику системы, а также представляет состояние приложения. Он не зависит от уровня представления, контроллер извлекает данные из уровня модели и отправляет их туда, где эти данные можно визуализировать — слой представления.
2. **Контроллер**. Уровень контроллера действует как интерфейс между представлением и моделью. Он получает запросы со слоя представление и обрабатывает их, включая необходимые проверки. Запросы затем отправляются на уровень модели для обработки данных, и как только они обрабатываются, данные отправляются обратно в контроллер и затем отображаются в представлении.
3. **Представление**. Этот уровень представляет собой результат приложения, обычно это форма пользовательского интерфейса. Уровень представления исполь-

зуется для отображения данных модели, полученных контроллером.

Примеры создания серверных решений с помощью MVC

Представим, у некоторого небольшого предприятия есть веб-сайт, который обслуживает клиентов. Необходимо добавить приложение поддержки клиентов, которое позволит клиентам задавать вопросы и позволять сотрудникам отвечать на эти вопросы.

На данном этапе проект будет довольно простым. Он состоит из трех страниц, обработанных doGet: список обращений, страница для создания обращения и страница просмотра. Он также имеет возможность принятия запроса POST для создания нового обращения.

Проект содержит классы Card и CardServlet. Класс Card — простой POJO (*plain old Java objects*) и выступает в качестве модели:

```
public class Card
{
    private String clientName;
    private String topic;
    private String message;
    public String getClientName()
    {
        return clientName;
    }
    void setClientName(String clientName)
    {
        this.clientName = clientName;
    }
```

```
    public String getTopic()
    {
        return topic;
    }

    void setTopic(String topic)
    {
        this.topic = topic;
    }

    public String getMessage()
    {
        return message;
    }

    void setMessage(String message)
    {
        this.message = message;
    }
}
```

В то время как CardServlet является контроллером:

```
@WebServlet(
        name = "cardServlet",
        urlPatterns = {"/cards"},
        loadOnStartup = 1
)

public class CardServlet extends HttpServlet
{
    private Map<Integer,
        Card> cardBase = new LinkedHashMap<>();
    private volatile int CARD_ID = 1;
    …
}
```

Так как на данном этапе нам не важен принцип хранения данных, в качестве базы данных обращений будем использовать Map.

Реализация doGet выглядит следующим образом:

```
@Override
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
    {
        String action = request.getParameter("action");
        if(action == null)
            action = "list";
        switch(action)
        {
            case "create":
                this.showCardForm(request, response);
                break;
            case "view":
                this.viewCard(request, response);
                break;
            case "list":
                this.listCards(request, response);
                break;
        }
    }
```

Метод doGet использует паттерн action/executor: действие передается через параметр запроса, а метод doGet отправляет запрос исполнителю (методу) на основе этого действия. Метод doPost аналогичен:

```
@Override
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
```

```
    {
        String action = request.getParameter("action");
        if(action == null)
            action = "list";

        switch(action)
        {
            case "create":
                this.createCard(request, response);
                break;
            case "list":
                response.sendRedirect("cards");
                break;
        }
    }
```

Здесь используется метод перенаправления. В этом случае, если клиент выполняет POST с отсутствующим параметром действия, его браузер перенаправляется на страницу со списком обращений.

Метод createCard использует параметры запроса для заполнения объекта Card и добавления его в базу данных.

```
private void createCard(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
{
    Card card = new Card();
    card.setClientName(request.
         getParameter("clientName"));
    card.setTopic(request.getParameter("topic"));
    card.setMessage(request.getParameter("message"));
    int id;
    synchronized(this)
    {
```

```
            id = this.CARD_ID++;
            this.cardBase.put(id, card);
        }
        response.sendRedirect(
                "cards?action=view&cardId=" + id);
        }
```

В методе createCard используется блок synchronized для блокировки доступа к базе данных обращений. В этом блоке кода происходит два действия: CARD_ID увеличивается и его значение извлекается, а обращение добавляется в Map. Обе эти переменные являются переменными экземпляра Servlet, что означает, что несколько потоков могут иметь к ним доступ одновременно. Включение этих действий в блок synchronized гарантирует, что ни один другой поток не сможет выполнять эти две строки кода одновременно. Поток, выполняющий этот блок кода, имеет исключительный доступ для выполнения блока до его завершения. Разумеется, всегда следует соблюдать осторожность при использовании методов или блоков synchronized, поскольку неправильное применение синхронизации может привести к дедлокам.

Настало время заняться уровнем представления

Директива page и многие атрибуты, которые она предоставляет, позволяют настроить то, как JSP будет интерпретирован, скомпилирован и обработан, но проект, который содержит много JSP с аналогичными свойствами, выглядит громоздко. Существует способ настройки общих свойств JSP в дескрипторе развертывания. В файле

web.xml, в котором пока находится только <display-name>Support Application</display-name> необходимо добавить следующее содержимое:

```
<jsp-config>
<jsp-property-group>
<url-pattern>*.jsp</url-pattern>
<url-pattern>*.jspf</url-pattern>
<page-encoding>UTF-8</page-encoding>
<scripting-invalid>false</scripting-invalid>
<include-prelude>/WEB-INF/jsp/base.jspf
    </include-prelude>
<trim-directive-whitespaces>true
    </trim-directive-whitespaces>
<default-content-type>text/html</default-content-type>
</jsp-property-group>
</jsp-config>
```

Тег <jsp-config> может содержать разное количество тегов <jsp-property-group>. Эти группы свойств используются для разделения свойств для разных групп JSP. Например, можно определить один набор общих свойств для всех JSP в папке */WEB-INF/jsp/admin* и другой набор общих свойств для всех JSP в папке */WEB-INF/jsp/help*. Вы выделяете эти группы свойств, определяя различные теги <url-pattern> для каждой <jsp-property-group>.

В примере теги <url-pattern> указывают, что эта группа свойств применяется ко всем файлам, заканчивающимся на *.jsp* и *.jspf*, в любом месте веб-приложения. Если необходимо обрабатывать JSP в одной папке иначе, чем JSP в другой, описанной выше, можно иметь два (или более) тега <jsp-property-group>, причем один из них имеет <url-

pattern> */WEB-INF/jsp/admin/*.jsp* </ url-pattern>, а другой — <url-pattern> */WEB-INF/jsp/help/*.jsp* </url-pattern>

Правила работы с тегом <url-pattern>:

- Если в приложении файл соответствует <url-pattern> как в <servlet-mapping>, так и в группе свойств JSP, преобладает тот, где указано более конкретное совпадение. Например, если первый <url-pattern> был **.jsp*, а другой — */WEB-INF/jsp/admin/*.jsp*, приоритет выше у того, у которого /WEB-INF/jsp/admin/*.jsp. Если теги <url-pattern> одинаковые, группа свойств JSP приоритетней, чем отображение сервлета.
- Если какой-либо файл совпадает с <url-pattern> в более чем одной группе свойств JSP, приоритетно более конкретное совпадение. Если они идентичны, то первая соответствующая группа свойств JSP идет в том порядке, в котором она появляется в дескрипторе развертывания.
- Если какой-либо файл совпадает с <url-pattern> в более чем одной группе свойств JSP, и более чем одна из этих групп свойств содержит правила <include-prelude> или <include-coda>, применяются правила включения из всех групп свойств JSP для этого файла, хотя для других свойств используется только одна из групп свойств.

Чтобы понять последний пункт, рассмотрим следующие группы свойств:

```
<jsp-property-group>
<url-pattern>*.jsp</url-pattern>
<url-pattern>*.jspf</url-pattern>
```

```
<page-encoding>UTF-8</page-encoding>
<include-prelude>/WEB-INF/jsp/base.jspf</include-prelude>
</jsp-property-group>
<jsp-property-group>
<url-pattern>/WEB-INF/jsp/admin/*.jsp</url-pattern>
<url-pattern>/WEB-INF/jsp/admin/*.jspf</url-pattern>
<page-encoding>ISO-8859-1</page-encoding>
<include-prelude>/WEB-INF/jsp/admin/include.jspf
    </include-prelude>
</jsp-property-group>
```

Файл с именем */WEB-INF/jsp/client.jsp* будет соответствовать только первой группе свойств. Он будет иметь кодировку символов UTF-8, а файл */WEB-INF/jsp/base.jspf* будет включен в начале. С другой стороны, */WEB-INF/jsp/admin/client.jsp* будет соответствовать обеим группам свойств. Поскольку вторая группа свойств является более конкретной, этот файл будет иметь кодировку символов ISO-8859-1. Однако оба параметра */WEB-INF/jsp/base.jspf* и */WEB-INF/jsp/admin/include.jspf* будут включены в начало этого файла. Чтобы не запутаться — необходимо максимально упрощать группы свойств JSP.

Тег <include-prelude> в дескрипторе развертывания проекта указывает контейнеру включить файл */WEB-INF/jsp/base.jspf* в начале каждого JSP, который принадлежит этой группе свойств. Это полезно для определения общих переменных, деклараций библиотеки тегов или других ресурсов, которые должны быть доступны для всех JSP в группе. Точно так же тег <include-coda> определяет файл, который будет включен в конце каждого JSP

в группе. Можно использовать оба эти тега более одного раза в одной группе JSP. Например, можно создать файлы *header.jspf* и *footer.jspf* для включения в начале и конце каждого JSP. Эти файлы могут содержать HTML-контент заголовка и нижнего колонтитула для работы в качестве своего рода шаблона для приложения. Необходимо позаботиться о том, чтобы иметь возможность легко включать эти файлы в другие места в будущем.

Тег <page-encoding> идентичен атрибуту pageEncoding директивы page. Поскольку JSP уже имеют тип содержимого text/html по умолчанию, можно просто указать <page-encoding> UTF-8, чтобы изменить кодировку символов типа содержимого JSP из text/html; ISO-8859-1 в text/html; UTF-8. Можно также использовать тег <default-content-type> для переопределения text/html с некоторым другим типом содержимого по умолчанию.

Особенно полезным свойством является <trim-directive-whitespaces>. Это свойство указывает транслятору JSP, удалить из ответа вывод любого пробела, созданного директивами, декларациями, сценариями и другими тегами JSP.

Тег <scripting-invalid> необходим для отключения Java в JSP. Значение false (по умолчанию), разрешает Java во всех JSP в группе.

<el-ignored> — по умолчанию false — тег аналогичен и соответствует атрибуту isELIgnored директивы page. Если значение true — язык выражений запрещен в JSP группы (что приводит к ошибке компиляции, если используется EL).

В проект добавлен */WEB-INF/jsp/base.jspf* во всех JSP в приложении. Веб-контейнер игнорирует это правило include к самому *base.jspf*. Его содержимое выглядит просто:

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

Он импортирует эти классы для всех JSP и объявляет библиотеку основных тегов JSTL с XMLNS префиксом «c». Этот файл помещается в каталог */WEB-INF/jsp*, а не в корневой каталог, потому что файлы в каталоге WEB-INF защищены от доступа из сети. Размещение JSP-файла в этом каталоге запрещает пользователям получать доступ к JSP из своего браузера. Это можно сделать для любого JSP, к которому браузеры не должны иметь доступ напрямую, например JSP, которые ссылаются на атрибуты сессии и запроса, предоставленные только передаваемым сервлетом и JSP.

Последнее, что осталось создать — это файл *index.jsp* в корневом каталоге веб-сайта проекта. Наличие этого файла в корневом каталоге означает, что он может отвечать на запросы для развернутого корневого приложения (/), не будучи напрямую идентифицированным в URL-адресе. Он содержит две строчки кода:

```
<%@ page session="false" %>
<c:redirect url="/cards" />
```

Вторая строка кода перенаправляет пользователя на URL-адрес сервлета /card относительно разверну-

того приложения. Первая строка кода отключает сессии в *index.jsp*.

Типичный паттерн объединения сервлетов и JSP заключается в том, чтобы сервлет принял запрос, выполнил любую обработку бизнес-логики хранения или извлечения данных, через модель, которая может быть легко использована в JSP, а затем перенаправляет запрос в JSP.

Сначала опишем метод showCardForm в CardServlet. Необходимо принять HttpServletRequest, а затем перейти на JSP:

```
private void showCardForm(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
{
    request.getRequestDispatcher(
            "/WEB-INF/jsp/view/cardForm.jsp")
            .forward(request, response);
}
```

Метод getRequestDispatcher получает javax.servlet.RequestDispatcher, который обрабатывает внутренние перенаправления и включает в себя определенный путь (в данном случае */WEB-INF/jsp/view/cardForm.jsp*). С помощью этого объекта можно перенаправить текущий запрос на этот JSP, вызвав метод forward.

Обратите внимание, что это не перенаправление: браузер пользователя не получает код статуса перенаправления, а содержимое адресной строки браузера не изменяется. Вместо этого обработка внутренних запросов пересылается в другую часть приложения.

После того, как Вы вызовите forward, ваш сервлет-код никогда не должен снова обрабатывать ответ. Это может привести к ошибкам или неустойчивому поведению. Теперь создайте JSP-файл, на который этот метод перенаправляет:

```
<%@ page session="false" %>
<!DOCTYPE html>

<html>
    <head>
        <title>Support service</title>
    </head>

    <message>
        <h1>Add a card</h1>
        <form method="POST" action="cards" >
            <input type="hidden" name="action"
                   value="create"/>
            Name<br/>
            <input type="text" name="clientName"><br/>
            Topic<br/>
            <input type="text" name="topic"><br/>
            Message<br/>
            <textarea name="message" rows="10"
                    cols="25"></textarea><br/>
            <input type="submit" value="Submit"/>
        </form>
    </message>

</html>
```

Все, что нужно сделать, это скопировать код с Java на JSP. В JSP сессии отключены, так как пока еще они не используются.

Теперь необходимо описать метод viewCard. Но, сперва, необходимо подумать о его представлении. Во-первых — какие элементы данных нужно использовать — и только потом писать код сервлета, чтобы предоставить эту информацию. Поэтому файл */WEB-INF/jsp/view/viewCard.jsp* будет выглядеть следующим образом:

```
<%@ page import="com.academy.Card" %>
<%@ page session="false" %>
<%
    String cardId = (String)request.
                    getAttribute("cardId");
    Card card = (Card)request.getAttribute("card");
%>

<!DOCTYPE html>
<html>
    <head>
        <title>Support service</title>
    </head>
    <message>
        <h1>Card #<%= cardId %>: <%= card.getTopic() %>
        </h1>
        <i>Client Name - <%= card.getClientName() %>
             </i><br />
        <%= card.getMessage() %><br />
        <a href="<c:url value="/cards" />">
            Back to other cards</a>
    </message>
</html>
```

Этот JSP показывает, что для уровня представления требуется идентификатор cardId и card для правильного отображения. Из метода viewCard можно предоставить эти переменные и передать запрос JSP:

```
private void viewCard(HttpServletRequest request,
             HttpServletResponse response)
             throws ServletException, IOException
{
    String idStr = request.getParameter("cardId");
    Card card = this.getCard(idStr, response);
    if(card == null)
       return;
    request.setAttribute("cardId", idStr);
    request.setAttribute("card", card);
    request.getRequestDispatcher(
            "/WEB-INF/jsp/view/viewCard.jsp")
            .forward(request, response);
}
```

Код метода getCard следующий:

```
private Card getCard(String idStr,
        HttpServletResponse response)
        throws IOException
{
    if(idStr == null || idStr.length() == 0)
    {
        response.sendRedirect("cards");
        return null;
    }
    try
    {
        Card card = this.cardBase.
                    get(Integer.parseInt(idStr));
        if(card == null)
        {
            response.sendRedirect("cards");
            return null;
        }
        return card;
```

```
        }
        catch(Exception e)
        {
            response.sendRedirect("cards");
            return null;
        }
    }
```

Первые несколько строк метода viewCard выполняют бизнес-логику анализа параметра запроса и получения карточки обращения из базы данных. Затем код добавляет два атрибута в запрос. Это основная цель атрибутов запроса. Они могут использоваться для передачи данных между различными элементами приложения, которые обрабатывают один и тот же запрос, например, между сервлетом и JSP. Атрибуты запроса отличаются от параметров запроса: атрибутами запроса являются Objects, в то время как параметры запроса — это Strings, а клиенты не могут передавать атрибуты, подобные параметрам. Атрибуты запроса существуют исключительно для внутреннего использования в приложении. Если Servlet помещает Card в атрибут запроса, JSP получает его как Card. В течение срока действия запроса любой компонент приложения, имеющего доступ к экземпляру HttpServletRequest, имеет доступ к атрибутам запроса. Когда запрос завершен, атрибуты запроса отбрасываются.

Последний метод, который нужно реализовать — метод listCards. Начнем с создания файла представления */WEB-INF/jsp/view/listCards.jsp*. Поскольку атрибуты запроса являются Objects, Вы должны отбрасывать их, ког-

да извлекаете. В этом случае приведение в Map <Integer, Card> является непроверенной операцией, поэтому Вам нужно скрыть предупреждение.

```
<%@ page session="false" import="java.util.Map" %>
<%
    @SuppressWarnings("unchecked")
    Map<Integer, Card> cardBase = (Map<Integer,
        Card>)request.getAttribute("cardBase");
%>

<!DOCTYPE html>
<html>

<head>
    <title>Support service</title>
</head>

<body>
    <h1>Cards</h1>
    <a href="<c:url value="/cards">
    <c:param name="action" value="create" />
    </c:url>">Add card</a><br /><br />
    <%
    if(cardBase.size() == 0)
    {
        %><b>Base is empty.</b><%
    }
    else
    {
        for(int id : cardBase.keySet())
        {
            String idStr = Integer.toString(id);
            Card card = cardBase.get(id);
            %>Card #<%= idStr %>:
            <a href="<c:url value="/cards">
            <c:param name="action" value="view" />
```

```
            <c:param name="cardId" value="<%= idStr %>" />
            </c:url>"><%= card.getTopic() %></a>
            (client:
            <%= card.getClientName() %>)<br /><%
        }
    }
%>

</body>

</html>
```

Этому JSP необходима база данных CardBase, поэтому в методе listCards необходимо предоставить эту возможность и переслать запрос:

```
private void listCards(HttpServletRequest request,
           HttpServletResponse response)
           throws ServletException, IOException
{
    request.setAttribute("cardBase", this.cardBase);
    request.getRequestDispatcher(
            "/WEB-INF/jsp/view/listCards.jsp")
            .forward(request, response);
}
```

Скомпилируйте приложение и запустите Tomcat в IDE. Перейдите в браузере по адресу *http://localhost:8080/service/*. Вы будете перенаправлены на *http://localhost:8080/service/cards* из-за кода перенаправления в файле *index.jsp*. Вы должны увидеть следующую страницу. Создайте несколько карточек с обращениями в службу поддержки и просмотрите их.

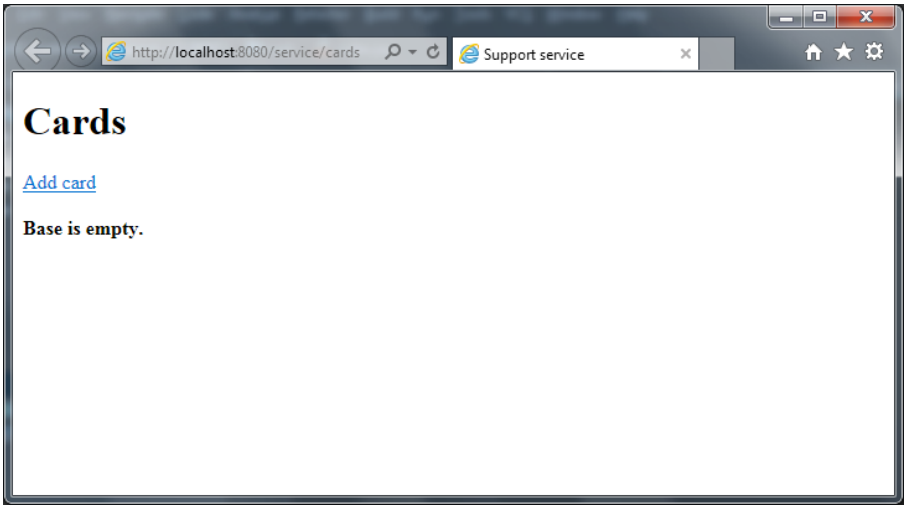
http://localhost:8080/service/cards
Support service
Cards
Add card
Base is empty.

Рисунок 1.

13. Работа с файлами в JSP (upload/download)

При обращении в службу сервисного центра клиенту зачастую может понадобиться прикрепить файл к сообщению. Реализуем данную логику для нашего приложения. Для этого мы будем использовать класс POJO — Attachment в качестве Model. Выглядеть он будет следующим образом:

```
public class Attachment {
    private String fileName;
    private byte[] fileContents;
    public String getFileName() {
        return fileName;
    }

    public void setFileName(String fileName) {
        this.fileName = fileName;
    }

    public byte[] getFileContents() {
        return fileContents;
    }

    public void setFileContents(byte[] fileContents) {
        this.fileContents = fileContents;
    }
}
```

После создания класса Attachment необходимо объявить Map объектов и необходимые методы для работы с вложениями в классе Card:

```
import java.util.Collection;
import java.util.LinkedHashMap;
import java.util.Map;
public class Card
{
…
    private Map<String, Attachment>
            attachments = new LinkedHashMap<>();
    public Attachment getAttachment(String name)
    {
        return this.attachments.get(name);
    }
    public Collection<Attachment> getAttachments() {
        return attachments.values();
    }
    public void addAttachment(Attachment attachment)
    {
        this.attachments.put(attachment.getFileName(),
            attachment);
    }
    public int getNumberOfAttachments()
    {
        return this.attachments.size();
    }
…
}
```

Далее необходимо внести изменения в CardServlet.

```
@MultipartConfig(
    fileSizeThreshold = 5_242_880, //5MB
    maxFileSize = 20_971_520L, //20MB
    maxRequestSize = 41_943_040L //40MB
)
```

Аннотации @MultipartConfig инструктируют веб-контейнер предоставлять поддержку загрузки файлов для этого сервлета. Она имеет несколько важных атрибутов, на которые Вы должны обратить внимание:

- **fileSizeThreshold** сообщает веб-контейнеру, насколько большой файл должен быть, прежде чем он будет записан во временный каталог. В этом примере загруженные файлы размером менее 5 мегабайт хранятся в памяти до тех пор, пока запрос не будет завершен, а затем они станут доступны для сборщика мусора. Если файл превышает 5 мегабайт, контейнер хранит его до тех пор, пока запрос не завершится, после чего он удалит файл с диска.
- **maxFileSize** указывает, что загруженный файл не должен превышать 20 мегабайт.
- **maxRequestSize** указывает, что общий размер запроса не должен превышать 40 мегабайт, независимо от количества загружаемых файлов.

В методе doGet добавим еще один case для загрузки вложения:

```
case "download":
      this.downloadAttachment(request, response);
      break;
```

Реализуем метод downloadAttachment:

```
private void downloadAttachment(HttpServletRequest
        request, HttpServletResponse response)
        throws ServletException, IOException{
```

```
        String idStr = request.getParameter("cardId");
        Card card = this.getCard(idStr, response);
        if(card == null)
            return;
        String fileName = request.
               getParameter("attachment");
        if(fileName == null)
        {
            response.sendRedirect(
               "cards?action=view&cardtId=" +
               idStr);
            return;
        }
        Attachment attachment =
            card.getAttachment(fileName);
        if(attachment == null)
        {
            response.sendRedirect(
                "cards?action=view&cardId=" + idStr);
            return;
        }
        response.setHeader("Content-Disposition",
                           "attachment; filename=" +
                            attachment.getFileName());
        response.setContentType(
                           "application/octet-stream");
        ServletOutputStream stream =
                            response.getOutputStream();
        stream.write(attachment.getFileContents());
    }
```

Часть кода, выделенная жирным шрифтом отвечает за передачу загрузки файла в браузер клиента. Заголовок Content-Disposition указывает браузеру запрашивать у клиента сохранить или загрузить файл вместо того, чтобы просто открывать файл в браузере. ServletOut-

putStream записывает содержимое файла в ответ. Это не самый эффективный способ, поскольку он может иметь проблемы с памятью для больших файлов. Для загрузки больших файлов, необходимо скопировать байты из InputStream в ResponseOutputStream, затем регулярно очищать ResponseOutputStream, чтобы байты перенаправлялись обратно в браузер пользователя, вместо буферизации в памяти.

Добавим логику добавления файла в обращение в метод createCard и создадим метод, который он использует, processAttachment. Метод processAttachment получает InputStream из многостраничного запроса и копирует его в объект Attachment. Он использует метод getSubmittedFileName, для определения исходного имени файла перед его загрузкой. Метод createCard использует этот метод и другие параметры запроса для заполнения объекта Card и добавления его в базу данных.

```
private void createCard(HttpServletRequest request,
             HttpServletResponse response)
             throws ServletException, IOException
{
    ...
    Part filePart = request.getPart("file1");
    if(filePart != null && filePart.getSize() > 0)
    {
        Attachment attachment = this.
                   processAttachment(filePart);
        if(attachment != null)
            ticket.addAttachment(attachment);
        }
    ...
}
```

```
private Attachment processAttachment(Part filePart)
                    throws IOException
{
    InputStream inputStream = filePart.getInputStream();
    ByteArrayOutputStream outputStream =
           new ByteArrayOutputStream();
    int read;
    final byte[] bytes = new byte[1024];
    while((read = inputStream.read(bytes)) != -1)
    {
        outputStream.write(bytes, 0, read);
    }
    Attachment attachment = new Attachment();
    attachment.setFileName(filePart.
                           getSubmittedFileName());
    attachment.setFileContents(outputStream.
                           toByteArray());
    return attachment;
}
```

Теперь внесем изменения на уровне представления. Добавим импорт в файле *base.jspf*:

```
<%@ page import="com.academy.Card,
    com.academy.Attachment" %>
```

Затем в JSP cardForm в тег <form> добавим атрибут:

```
enctype="multipart/form-data"
```

И в конец формы перед кнопкой Submit следующий HTML-код:

```
Attachments<br/>
<input type="file" name="file1"/><br/>
```

Далее файл viewCard необходимо дополнить кодом для отображения вложений:

```
<%
    if(card.getNumberOfAttachments() > 0)
    {
         %>Attachments: <%
         int i = 0;
         for(Attachment item : card.getAttachments())
         {
             if(i++ > 0)
                 out.print(", ");
                 %><a href="<c:url value="/cards">
                 <c:param name="action"
                          value="download" />
                 <c:param name="cardId"
                          value="<%= cardId %>" />
                 <c:param name="attachment"
                          value="<%=
                          item.getFileName() %>" />
             </c:url>"><%= item.getFileName() %></a><%
         }
       %><br /><br /><%
    }
%>
```

Запустите приложение и протестируйте работу с файлами в браузере.

14. Expression Language в JSP

Что такое Expression Language? История возникновения Expression Language

Expression Language (EL) первоначально был разработан как часть стандартной библиотеки тегов Java JSP (JSTL) для поддержки предоставления данных на страницах JSP без использования сценариев, деклараций или выражений. В значительной степени он был основан на ECMAScript (основа JavaScript) и языках XPath. В то же время он упоминался как с *Simplest Possible Expression Language* (SPEL), но позже сокращен до Expression Language. EL был частью спецификации JSTL 1.0, которая появилась в JSP 1.2 и использовался только в атрибутах тегов JSTL. В JSP 2.0 и JSTL 1.1 из-за популярности спецификация EL перешла от спецификации JSTL к спецификации JSP и EL стал доступен для использования в любом месте JSP, а не только в атрибутах тега JSTL.

В то же время, началась работа над JavaServer Faces, построенной на базе JSP 1.2 как альтернатива JSP. JSF также нуждался в собственном языке выражений. Однако повторное использование EL имело несколько недостатков. Во-первых, JSF необходимо было контролировать вычисление выражения для определенных точек жизненного цикла JSF. Выражение должно быть вычислено во время рендеринга страницы, а также во время обратной передачи на страницу JSF. Кроме того, JSF

нуждался в лучшей поддержке выражений метода, чем предлагаемый EL.

В результате сформировались два чрезвычайно похожих языка выражения — один для JSP 2.0? другой для JSF 1.0. Наличие двух отдельных языков выражения Java не являлось идеальным решением, поэтому, когда началась работа над JSP 2.1, попытались объединить EL JSP 2.0 с EL JSF 1.1. Результатом стал *Java Unified Expression Language* (JUEL) для JSP 2.1 и JSF 1.2.

Несмотря на то, что JSP и JSF были разделены, EL не получал свой собственный JSR, но продолжал оставаться частью спецификация JSP, хотя у него был свой собственный технический документ и артефакт JAR. Это осталось в EL в JSP 2.2. EL продолжает расширяться и улучшаться, а с Java EE 7 он перемещен в собственный JSR (JSR 341) и обновлен с поддержкой лямбда-выражений и эквивалентом Java 8 Collection Stream API, маркированный как *Java Unified Expression Language 3.0* (EL 3.0). EL 3.0 был выпущен с Java EE 7, Servlet 3.1, JSP 2.3 и JSF 2.2 в 2013 году. Сейчас мы рассмотрим EL 3.0, поскольку он напрямую относится к JSP.

Синтаксис Expression Language

Базовый синтаксис для EL определяет выражения, которые требуют вычисления. JSP-интерпретатор должен определить, когда выражение EL начинается и заканчивается, чтобы он мог анализировать и оценивать выражение отдельно от остальной части страницы. Существует два разных типа базового синтаксиса EL: немедленное и отложенное вычисление.

Немедленное вычисление EL — JSP-движок анализирует и вычисляет EL во время отображения страницы. По мере того как код JSP выполняется сверху вниз, выражение EL вычисляется, как только движок JSP сталкивается с ним и до продолжения остальной части страницы. Выражения EL, которые должны быть немедленно вычислены, выглядят следующим образом:

```
//expr – действительное выражение EL
${expr}
```

Знак доллара и открывающие и закрывающие скобки определяют границы выражения EL. Все что внутри скобок анализируется как выражение EL. Нельзя использовать этот синтаксис для каких-либо других целей в JSP, в противном случае это будет оцениваться как выражение EL и приведет к синтаксической ошибке. Если необходимо написать что-то с этим синтаксисом для вывода в ответ, Вам нужно будет экранировать знак доллара:

```
\${не является выражением EL}
```

Обратный слеш перед знаком доллара указывает движку JSP, что это не выражение EL. Также можно использовать XML-суффикс доллара $ вместо \$ — это приведет к такому же результату.

```
&#36;{не является выражением EL}
```

Механизм JSP также игнорирует это, но многие предпочитают использовать обратный слеш. Также может понадобиться выводить обратный слеш перед выражени-

ем, которое нужно вычислить. Для этого необходимо использовать XML-суффикс обратного слеша:

```
&#92;${выражение EL будет вычислено}
```

В этом случае выражение EL будет вычислено и отображено после обратного слеша.

Отложенное вычисление EL является частью языка унифицированных выражений, которые в первую очередь поддерживает потребности JavaServer Faces. Он обычно не используется в JSP. Отложенный синтаксис выглядит почти идентично немедленному синтаксису:

```
//expr – действительное выражение EL
#{expr}
```

В JSF отложенные выражения могут быть вычислены либо при визуализации страницы, либо во время обратной передачи на страницу или, возможно, в двух случаях. В JSP #{} отложенный синтаксис, который действителен только в атрибутах тегов JSP, может использоваться для отсрочки вычисления выражения EL в процессе рендеринга тега. EL выражение вычисляется до того, как значение атрибута привязано к тегу (как это было бы с ${}), атрибут тега получает ссылку на не вычисленное выражение EL. Затем тег может вызывать метод для вычисления EL-выражения, когда это уместно.

Потенциальная проблема отложенного синтаксиса заключается в том, что некоторые языки шаблонов и JavaScript фреймворки используют #{} синтаксис для замещений. Из-за этого, необходимо избегать эти под-

становки, чтобы не путать их с отсроченным анализом EL выражений:

```
\#{не является выражением EL}
&#35;{также не является выражением EL}
```

Однако это может не работать в некоторых фреймворках, что может быть реальной проблемой, если нужно использовать EL часто, или если у Вас есть много JSP, которые должны работать с EL 2.1 или выше. Также объект XML несовместим с JavaScript. Существует еще один вариант для предотвращения того, чтобы литерал #{} вычислялся как отложенное выражение. В <jsp-config> в разделе дескриптора развертывания можно добавить следующий тег <jsp-property-group>:

```
<deferred-syntax-allowed-as-literal>true
    </deferred-syntax-allowed-as-literal>
```

Это позволяет использовать #{} синтаксис буквально и не мешает Вам избежать хэш-тег в этом случае. Если нужно управлять ним для отдельных JSP, можно использовать атрибут deferredSyntaxAllowedAsLiteral="true" директивы page в любом JSP.

Размещение EL-выражений

Выражения EL можно использовать практически в любом месте JSP с несколькими незначительными исключениями. Во-первых, выражения EL не могут использоваться в каких-либо директивах. Директивы (<%@ page %> , <%@ include %> , а также <%@ taglib %>) анализируют-

ся при компиляции JSP, но EL выражения вычисляются позже, когда JSP отображается, поэтому они не сработают. Кроме того, EL-выражения недействительны в объявлениях JSP (<%! %>), сценариях (<% %>), или выражениях (<%= %>). В таком случае выражение EL просто игнорируется или приводит к синтаксической ошибке.

Выражения EL могут быть размещены где угодно. Например, в текстовом литерале:

```
Every ${expr} wants to know where the ${expr} is.
```

Этот пример включает два EL-выражения, которые во время вычисления помещаются в строку с текстом, который выводится. Если первое выражение оценивается как «hunter», а второе выражение оценивается «pheasant», пользователь увидит следующее:

```
Every hunter wants to know where the pheasant is.
```

Выражения могут использоваться в стандартных атрибутах HTML-тегов:

```
<input type="text" name="clientName" value="${expr}" />
```

Также можете использовать их в атрибутах тегов JSP:

```
<c:url value="/clientName/${expr}/${expr}" />
<c:redirect url="${expr}" />
```

Выражения EL не должны составлять все значение атрибута. Вместо этого одна или более частей значений атрибута может включать EL-выражения. Другие особенности HTML, такие как JavaScript или CSS также мо-

гут содержать EL-выражения в кавычной или литеральной форме. Механизм JSP не вычисляет их, а записывает в ответ так, как если бы они были буквальным текстом:

```
<script type="text/javascript" lang="javascript">
    var clientName = '${expr}';
    var topic = ${expr};
    var message = ${expr};
</script>
```

Выражения EL, как и любой другой язык, имеют определенный синтаксис. Как и в большинстве других языков, этот синтаксис является строгим, и его нарушение приведет к синтаксическим ошибкам, при отображении JSP. Однако, в отличие от Java, синтаксис EL слабо типизирован и имеет много встроенных неявных преобразований типов, похожий на такие языки, как PHP или JavaScript. Основным правилом выражения является то, что оно должно вычислять некоторое значение. Нельзя объявлять переменные в выражении в выражении или выполнять какое-то назначение или операцию, которая не приводит к значению. Например, ${obj.method()} действует только в том случае, если method имеет не пустой тип void. EL не предназначен для замены Java — он разработан для того, чтобы предоставить разработчику инструменты, необходимые для создания JSP без Java.

Зарезервированные ключевые слова

Как и в любом другом языке, одним из первых, что нужно знать об EL, является список его зарезервированных ключевых слов. Это слова, которые должны исполь-

зоваться только по назначению. Переменные, свойства и методы не должны иметь имена, равные этим зарезервированным словам.

- true
- false
- null
- instanceof
- empty
- div
- mod
- and
- or
- not
- eq
- ne
- lt
- gt
- le
- ge

Первые четыре ключевых слова Вам знакомы из Java. Ключевое слово empty используется для проверки любой коллекции, Map, или массива, который содержит любые значения, или String которые имеют длину больше одного символа. Если любой из них null или empty, выражение — true, в противном случае — false.

```
${empty x}
```

Ключевые слова div и mod сопоставляются математическим операциям Java деления (/) и деления по модулю (%), соответственно.

- and, or и not означают логические операторы Java && и ||, а также !, соответственно.
- eq, ne, lt, gt, le, и ge являются альтернативой Java операторов сравнения (==, !=, <, >, <=, и >=).

Если Вы предпочитаете привычные уже Вам традиционные математические символы, вместо ключевых слов — можете также их использовать.

Приоритеты операторов

Как и в других языках, все предыдущие операторы вместе с другими операторами в EL, имеют порядок приоритетности. Этот порядок не отличается от приоритета оператора в Java.

В данной таблице отображен порядок приоритетов от верхнего (самого высокого) до нижнего (самого низкого). Операторы с одинаковым приоритетом вычисляются в порядке отображения в выражении слева направо.

Приоритет	Оператор	Описание
1	[], .	квадратные скобки и точка
2	()	круглые скобки, используются для изменения приоритета других операторов.
3	-, !, not, empty	унарный минус, отрицание, проверка на пустоту.
4	*, /, div, %, mod	умножение, деление, остаток от деления, целая часть от деления.
5	+, –	сложение, бинарное вычитание
6	+=	конкатенация строк (появился в EL 3.0)
7	<,lt, >, gt, <=, le, >=, ge	операторы сравнения
8	==, eq, !=, ne	операторы отношения равенства
9	&&, and	логическое И
10	||, or	логическое ИЛИ
11	?, :	условные операторы
12	->	лямбда-выражения (появился в EL 3.0)
13	=	оператор присваивания
14	;	точка с запятой (появился в EL 3.0), позволяет описывать несколько выражений со значениями, которые кроме последнего отбрасываются

Рассмотрим следующий пример:

```
${a = b – 7.2; obj.method(a);
      "To be, or not to be"}
```

Эта комбинация EL-выражения содержит в себе четыре выражения.

1. Выражение b – 7.2 рассчитывается и в результате получается 6.0 при условии, что b = 13.2.
2. Это значение присваивается a
3. Вызывается метод method с параметром a(6.0).
4. Рассчитывается строковый литерал «To be, or not to be». Результат этого выражения является результатом выражение после последней точки с запятой: «To be, or not to be».

Результаты выражений a = b – 7.2 и obj.method(a) отбрасываются. Это особенно полезно для назначения некоторого значения переменной EL, а затем включения этого значения в какую-либо другую часть выражения вместо того, чтобы просто выводить значение.

Значения литералов

Язык унифицированного выражения имеет поддержку для указания литеральных значений с определенным синтаксисом. Ключевые слова true, false, а также null являются буквальными значениями.

EL так же может иметь строковые литералы. В отличие от Java, где строковые литералы всегда окружены двойными кавычками, строковые литералы в EL могут быть окружены или двойными или одинарными кавыч-

ками, как в PHP и JavaScript. Таким образом, оба выражения в следующем примере действительны.

```
${"Gamlet's Speech."}
${'Speech: "To be, or not to be, that is the question."'}
```

Если необходимо смешивать одиночные и двойные кавычки внутри строкового литерала, который существует в пределах значение атрибута это становится не читабельным:

```
<c:url value="${'Gamlet\'s Speech:\"
    To be, or not to be.\"'}" />
```

По возможности, лучше использовать более простой строковый литерал.

С EL 3.0 в Java EE 7, есть возможность конкатенировать строковые литералы в выражениях EL, так как в Java. Все три строки в следующем примере эквивалентны и приводят к тому же результату.

```
Every ${expr} wants to know where the ${expr} is.
${'Every ' += expr += " wants to know where the " +=
    expr += ' is.'}
${"Every " += expr += ' wants to know where the ' +=
    expr += " is."}
```

Если expr приводит к некоторому объекту, который не является строкой, он будет преобразован к строке через метод toString.

Числовые литералы в EL упрощены по сравнению с Java, и даже можно выполнить арифметику между не-

которыми объектами, которую не могли бы использовать в Java.

Рассмотрим следующие три целых типа числовых литералов:

$ {123}	литерал обрабатывается как int во время вычисления.
$ {-124567891234}	литерал слишком велик, чтобы поместится в int — неявно преобразовывается к long.
$ {138987604321987655807}	литерал слишком велик, чтобы поместится в long — неявно преобразовывается к BigInteger.

Эти преобразования происходят без Вашего участия.

Типы decimal подобны целым типам, неявно преобразовываются к float, double и BigDecimal. Хотя литерал дробного типа по умолчанию в Java является double, то в EL по умолчанию float. Когда Вы работаете с EL-выражениями, нельзя явно указать тип литерала — он всегда преобразовывается неявно.

Выражения EL облегчают математические операции, поскольку все преобразования типов происходят неявно и потому, что арифметические операторы могут использовать типы BigInteger и BigDecimal.

Три других примитивных литерала: char, byte и short. Обычно эти типы данных не используются в EL-выражениях. EL не содержит конкретных литералов для этих типов, но будет преобразовывать другие литераторы к char, byte и short когда это необходимо.

Для типа char: null, ' ' или " " будет принудительно введен в символ нулевого байта (0x00). Односимвольный

строковый литерал (одинарная или двойная кавычка) будет приведен к его эквиваленту char. Целое число также будет приведено к char, если его значение находится в диапазоне от 0 до 65 535. В противном случае, любая строка с несколькими символами или любое число вне диапазона 0 и 65 535 приведет к ошибке.

Любое число целочисленного типа также будет преобразовано к byte или short при необходимости, до тех пор, пока число не выходит за пределы диапазона byte или short. В противном случае, попытка приведения приведет к ошибке.

Свойства и методы объекта

EL предоставляет упрощенный синтаксис для доступа к свойствам в JavaBeans в дополнение к стандартному синтаксису, который используется для доступа к публичным методам. Невозможно получить доступ к публичным полям из EL-выражения. Рассмотрим класс с именем Client с публичным полем clientName. Можно подумать, что есть возможность получить доступ к переменной clientName класса Client при помощи выражения EL:

```
$ {client.clientName}
```

Однако это не допустимо. Когда EL видит этот синтаксис, он ищет свойство clientName, а не поле. Рассмотрим измененный класс, где clientName — инкапсулированное приватное поле со стандартными методами getClientName и setClientName. Теперь выражение client.clientName становится ярлыком для вызова client.getClientName(). Это

может работать для любого поля любого типа. До тех пор, пока у него есть стандартный метод доступа JavaBean, к нему можно получить доступ таким образом. Если в классе Client было поле под названием.

Это не единственный способ, при помощи которого можно использовать свойства при помощи JavaBean. Так же можно получить доступ к свойствам, используя оператор []. Следующее выражение также получает доступ к clientName с использованием getClientName.

```
${client["clientName"]}
```

В более ранних версиях EL можно получить доступ только к свойствам JavaBeans. Нельзя вызывать методы объектов. Однако EL 2.1 добавила возможность вызова методов объектов в JSP. Таким образом, можно получить clientName типа Client при помощи вызова ${client.getClientName()}, вместо ${client.clientName}.

EL функции

В EL функция представляет собой специальный инструмент, сопоставленный статическому методу класса. Подобно XML-схемам, функции отображаются в пространстве имен. Общий синтаксис вызова функции следующий:

```
$ {[ns]: [fn] ([a1 [, a2 [, a3 [, ...]]]])},
```

где

[ns] — пространство имен,
[fn] — имя функции,
[a1] — [an] — аргументы.

Функции определены в *Tag Library Descriptors* (TLD). Существует набор функций, определенных в JSTL, которые удовлетворяют многим потребностям разработчиков в JSP, например функции связанные со строками — обрезка, поиск, объединение, разделение, экранирование и многое другое. По соглашению библиотека функций JSTL имеет пространство имен fn.

Наиболее распространенные функций JSTL EL:

1. **${fn:contains (String, String) }** — проверяет, содержит ли первая строка один или несколько экземпляров второй строки и возвращает true, если это так.
2. **${fn:escapeXml (String) }** — если выводимая строка содержит специальные символы, функция, экранирует их. (< станет <, > станет >, & станет &, «"» станет "). Это является особенно важным инструментом в предотвращении атак с межсайтовыми скриптингами (XSS).
3. **${fn:join (String [], String) }** — объединяет массив строк вместе, используя указанную строку как разделитель.
4. **${fn:length (Object) }** — если аргумент строка, функция вызывает и возвращает результат вызова метода length в указанной строке. Если это Collection, Map или массив — возвращает размер. Никакие другие типы не поддерживаются.
5. **${fn:toLowerCase (String) }** и **${fn:toUpperCase (String) }** — изменяет регистр строки в нижний или верхний соответственно.
6. **$ {fn:trim (String) }** — обрезает все пробелы с обоих концов указанной строки.

Статические поля и методы доступа

Новшеством в EL 3.0 является получение доступа к публичным статическим полям и методам в любом классе на пути к JSP-классу.

Вы получаете доступ к статическим полям и методам так же, как и в Java — используя полное имя класса и имя поля или метода, разделенные точкой. Например, Вы можете получить доступ к константе MAX_VALUE в классе Integer со следующим выражением:

```
$ {java.lang.Integer.MIN_VALUE}
```

Имя класса должно быть полным, если класс не импортируется с помощью page директивы JSP. В JSP, все классы в java.lang неявно импортируются. Поэтому, прежнее выражение могло бы быть написано следующим образом:

```
$ {Integer.MIN_VALUE}
```

С помощью этого можно получить доступ к статическим полям или методам в любом классе, к которому имеет доступ JSP. Важно отметить, что можно только читать значение этих полей, но не записывать в них. Вызвать статический метод в классе так же просто. Предположим, Вы захотели найти максимальное число из двух заданных:

```
$ {java.lang.Integer.max(5,7)}
$ {Integer.max(5,7)}
```

Это выражение вызывает статический метод max в классе Integer и передает два числа 5 и 7 в качестве аргу-

ментов. Помимо вызова статических методов также можно вызвать конструктор класса, который возвращает экземпляр этого класса, после чего можно дополнительно получить доступ к свойствам, вызвать методы или просто вывести объект.

```
${com.academy.Employee()}
${com.academy.Employee('Ivan', 'Ivanov').firstName}
```

Хотя доступ к статическому методу может полностью заменить поведение EL-функций и библиотек функций, это не означает, что библиотеки функций не нужны. Предыдущий вызов статического метода в Integer.max является удобным, но с использованием статических методов класса Integer, следующее выражение еще более удобно:

```
$ {int:max (5,7)}
```

Это может показаться не на много короче, но если имя класса гораздо длиннее сразу становится понятно, почему библиотеки функций по-прежнему очень полезны. Наиболее удобно использовать доступ к статическому полю в перечислениях.

Перечисления

Перечисления весьма полезны и эффективны. Традиционно, перечисления в EL приводятся к строке, когда это необходимо. Например, JSP имеет переменную в области видимости с именем month и представляет одно из значений из перечисления java.time.Month в новом API-интерфейсе Java 8 Date and Time. Вы можете прове-

рить, является ли month июнем со следующим логическим выражением:

```
${month == 'JUNE'}
```

Переменная month здесь преобразуется в String и сравнивается с 'JUNE'. Это отличается от Java, где преобразование автоматически никогда не произойдет. Хотя это удобно, это, безусловно, не безопасно для типов. Если Вы пропускаете июнь, IDE, скорей всего, не поймает это, и если Вы будете компилировать JSP во время непрерывной сборки интеграции, чтобы проверять ошибки компиляции JSP, ошибки не будет. Тем не менее, с EL 3.0 Вы можете использовать синтаксис доступа к статическому полю, чтобы получить ссылку на константу типа enum. В конце концов, константы перечисления представляют собой только публичные статические константные поля типа перечисления:

```
${month == java.time.Month.JUNE}
```

И, если Вы импортируете Month в свой JSP, выражение такое же, как и строковое enum выражение:

```
${month == Month.JUNE}
```

Эти последние два метода безопасны по типу и будут проверяться IDE во время компиляции.

Лямбда выражения

EL 3.0 вычисляет лямбда-выражения среди множества других своих функций. Лямбда выражение — анонимная функция, которая, как правило, передается как

аргумент функции более высокого порядка (например, метод Java). В наиболее общем смысле лямбда-выражения представляют собой список имен параметров, а затем некоторый тип оператора и, наконец, тело функции.

EL-лямбда-выражения используют оператор стрелки -> в качестве разделителя параметров выражения слева от выражения в правой части. Кроме того, скобки вокруг параметра выражения являются необязательными, если имеется ровно один параметр. Пример EL лямбда-выражений:

```
x -> x - 8
(x, y) -> x * y.
```

Конечно, лямбда-выражения не являются полными EL-выражениями. Что-нибудь должно быть сделано с лямбда-выражениями. Они могут быть сразу вычислены. Например:

```
$ {(x -> x - 8) (6)}
$ {((x, y) -> x * y) (6, 3)}
```

В предыдущих EL-выражениях лямбда-выражения объявлены и вычислены немедленно. Результат двух EL-выражений -2 и 18. Обратите внимание, что само лямбда-выражение окружено круглыми скобками. Это устраняет неоднозначность выражения лямбда от всего, что есть вокруг, и позволяет немедленно выполнить его. Также можно определить EL лямбда-выражение для использования позднее:

```
$ {z = (x, y) -> x * y; z(6, 2)}
```

Результат второго выражения в этом случае равен 12, потому что он выполняет лямбда-выражение, определенное до точки с запятой. Теперь лямбда-выражение z можно использовать в любом другом выражении EL, которое следует за этим выражением на странице. Это особенно полезно, если лямбда очень сложна.

Также, можно передать EL лямбда-выражение в качестве аргумента методу, вызванному в EL-выражении.

```
${employees.stream().filter(item -> item.lastName ==
    'Ivanov' || item.lastName == 'Sidorov').toList()}
```

Коллекции

В EL коллекции можно легко получить с помощью операторов точка и квадратные скобки. Какие использовать операторы, зависит от типа коллекции. В Java все коллекции представляют собой либо Collection, либо Map. В иерархии Map есть много разных типов map, все имеют общую основу: ключ-значение. Иерархия Collection сложнее: Set, List и Queue. Поскольку каждый тип коллекции имеет разный способ доступа к своим значениям, EL поддерживает каждый из них несколько иначе.

Доступ к значениям в Map довольно прост и имитирует доступ к свойствам на JavaBeans. Предположим, у Вас есть значение с именем map с ключом lastName, сопоставленным со значением 'Sidorov', и ключ employeeId, сопоставленный со значением '15'. Вы можете получить доступ к этим двум свойствам Map с помощью операторов скобок, как в следующем примере:

```
${map["lastName"]}
${map["employeeId"]}
```

Однако это не единственный метод, который можно использовать для доступа к значениям Map. Также можно обрабатывать ключи и получать доступ к их значениям с помощью оператора точки:

```
$ {map.lastName}
$ {map.employeeId}
```

Отметим некоторые ограничения на использование оператора точки для доступа к значениям Map. Если ключ не может быть идентификатором в Java, необходимо использовать скобки вместо оператора точки для доступа к значению, сопоставленному с этим ключом. Это означает, что ключ не может содержать пробелы или дефисы, не может начинаться с числа и не может содержать большинство специальных символов. Если содержатся любые символы, которые недействительны в Java-идентификаторах, Вы должны использовать скобки. Если Вы не уверены, используйте скобки и тогда не ошибетесь.

Доступ к элементам List прост. Рассмотрим список со значениями «monday», «tuesday» и «wednesday» в порядке от 0 до 2. Вы получите доступ к значениям, используя оператор скобки, как если бы List фактически был массивом. Следующий код демонстрирует это.

```
${list[0]}
${list[1]}
${list[2]}
```

EL позволяет Вам использовать строковые литералы вместо чисел для индексации списка, как если бы это была Map с List индексами, служащая в качестве ключей:

```
${list [0]}
${list ['1']}
${list ["2"]}
```

Единственное правило при использовании строковых литералов состоит в том, что строки должны быть конвертируемыми в целые числа, в противном случае код приводит к ошибкам во время выполнения. Рекомендуется использовать числовые литералы.

Значения двух других типов коллекций, Set и Queue, не могут быть доступны с помощью EL. Эти коллекции не предоставляют средства прямого доступа к значению, как индекс в List или ключ в Map. В Set и Queue нет методов «get». Доступ к значениям в этих типах коллекций можно получить только с помощью итерации. Как и в случае со всеми типами коллекций, можно проверить, пусты ли Set и Queue при помощи оператора empty:

```
${empty set}
${empty queue}
```

Использование области видимости переменных в EL выражениях

JSP имеют набор неявных переменных (request, response, session, out, application, config, pageContext, page и exception), которые можно использовать для получения

информации из среды запроса, сессии, среды исполнения и воздействия на ответ. EL имеет аналогичный набор неявных переменных. Однако он также имеет представление о неявной области видимости, в которой разрешены неизвестные переменные. Это позволяет получать информацию из разных источников с минимальным кодом.

Существует четыре разных атрибута области видимости (страница, запрос, сессия и приложение). Каждая из этих областей имеет все большую и большую область действия, чем предыдущая.

Область запроса начинается, когда сервер получает запрос и заканчивается, когда сервер завершает отправку ответа клиенту. Область запроса существует в любом месте, где есть доступ к объекту запроса, и атрибуты, привязанные к запросу, больше не связаны после завершения запроса.

Область сессии сохраняется между запросами и любой код с доступом к объекту HttpSession может получить доступ к области сессии. Когда сессия недействительна, его атрибуты не привязаны и область действия заканчивается.

Область страниц и приложений несколько отличается. Область страницы включает в себя атрибуты для конкретной страницы (JSP) и запроса. Когда переменная привязана к области страницы, она доступна только для этой страницы JSP и только в течение срока действия запроса. Другие JSP и сервлеты не могут получить доступ к переменной, связанной с областью страницы, и когда запрос завершается, переменная не связана. При доступе к объекту JspContext или PageContext можно сохранять

и извлекать атрибуты, существующие в пределах области страницы, с помощью методов setAttribute и getAttribute.

Область приложения — это самая широкая область, существующая во всех запросах, сессиях, страницах JSP и сервлетах. Объект ServletContext представляет область приложения, а также атрибуты, которые хранятся в нем, живут в области приложения.

Использование неявной области видимости в EL

EL определяет 11 неявных переменных в области видимости EL-выражений. Неявная область видимости более полезна и более часто используется из-за ее способности разрешать атрибут в области запроса, сессии, страницы или приложения. Когда EL выражение ссылается на переменную, EL-интерпретатор разрешает переменную, используя следующую процедуру: он проверяет, является ли переменная одной из 11 неявных переменных.

Если переменная не является одной из 11 неявных переменных, тогда EL-интерпретатор ищет атрибут в области страницы, который имеет такое же имя в качестве переменной. Если он находит соответствующий атрибут в области страницы, он использует значение атрибута как значение переменной.

1. Если нет соответствующего атрибута страницы, интерпретатор ищет атрибут запроса с тем же именем, что и переменная, и использует атрибут, если он найден.
2. Интерпретатор ищет атрибут сессии и использует его, если найден.
3. Интерпретатор ищет атрибут приложения и использует его, если найден.

4. Если интерпретатор просмотрев все, не находит неявной переменной или атрибута, соответствующего имени переменной, он выдает ошибку.

Прелесть этой функции в том, что не нужно извлекать экземпляр HttpServletRequest или HttpSession для использования атрибутов на любом из этих объектов.

Использование неявных переменных EL

Имеется 11 неявных EL-переменных, доступных для использования в выражениях EL. За исключением одного все являются объектами Map. Большинство из них используются для доступа к атрибутам из некоторой области видимости, параметров запроса или заголовков.

- pageContext — экземпляр класса PageContext, не является Map. Можно получить доступ к данным ошибки страницы и объекту исключения, интерпретатору выражений, автору вывода, экземпляру Servlet JSP, запросу и ответу, ServletContext, ServletConfig и сессии.
- pageScope — Map<String, Object>, которая содержит все атрибуты, привязанные к PageContext (область видимости страницы).
- requestScope — Map<String, Object> всех атрибутов, связанных с ServletRequest. Можно получить доступ к этим атрибутам, не вызывая метод в объекте запроса.
- sessionScope — представляет собой Map<String, Object> и содержит все атрибуты сессии из текущей сессии.
- applicationScope — последняя из областей видимости, Map<String, Object>, которая содержит все атрибуты, связанные с экземпляром ServletContext.

- param и paramValues — обеспечивают доступ к параметрам запроса. Переменная param представляет собой Map<String, String> и содержит только первое значение из любого параметра с несколькими значениями, тогда как Map<String, String []> paramValues содержит все значения каждого параметра.
- header и headerValues — предоставляют доступ к заголовкам запроса. Map<String, String> header содержит только первое значение любых многозначных заголовков и Map<String, String []> headerValues содержит все значения для каждого заголовка.
- initParam — Map<String, String>, содержит все параметры контекста из экземпляра ServletContext для этого приложения.
- cookie — Map<String, javax.servlet.http.Cookie>, содержит все файлы cookie, отправленные браузером пользователя вместе с запросом. Ключами являются имена файлов cookie.

Доступ к коллекциям через stream API

Одним из самых больших дополнений к языку Expression 3.0 в Java EE 7 является поддержка Stream API коллекций, представленного в Java SE 8. Поскольку API поддерживается в EL 3.0, не нужно запускать приложение на Java 8, чтобы воспользоваться преимуществами этой новой функции EL.

Основой Stream API является stream метод без аргументов, присутствующий в каждой коллекции. Этот метод возвращает java.util.stream.Stream, который может

фильтровать и иным образом манипулировать копией коллекции. Класс java.util.Arrays также предоставляет множество статических методов для извлечения Stream из разных массивов. Используя Stream, можно выполнять много разных операций. Некоторые из этих операций возвращают другие Stream, что позволяет создать цепочку операций. Этот конвейер состоит из источника конвейера (Stream), промежуточных операций (таких как фильтрация и сортировка) и, наконец, финальной операции (например, преобразования результатов в List, который можно перебрать и отобразить).

В EL 3.0 можно вызвать stream метод для любой переменной EL, которая представляет собой массив Java или Collection. Возвращенный Stream на самом деле не является java.util.stream.Stream, потому что EL 3.0 должен работать в Java 7, где Stream еще не существует. Вместо этого возвращаемый Stream представляет собой реализацию интерфейса Stream API, специфичную для EL.

Промежуточные операции фильтруют, сортируют, уменьшают, преобразуют или иным образом изменяют коллекцию. Важно понимать, что при выполнении промежуточных операций над Stream исходная коллекция или массив никогда не изменяются. Операции влияют только на содержимое Stream.

Фильтрация содержимого Stream, как правило, уменьшает количество содержащихся в нем объектов. Операция filter принимает предикатный аргумент — лямбда-выражение, которое возвращает логическое значение и принимает один аргумент, тип которого является типом элемента Stream. В List<E>, где E — тип элемента, stream возвра-

щает Stream<E>. Вызывая filter для Stream<E>, Вы предоставляете Predicate<E> с сигнатурой E -> boolean. Затем Вы используете свойства E, чтобы определить, включать ли этот E в результирующий Stream<E>:

```
${bookStore.stream().filter(book -> book.author ==
    "Shakespeare")}
```

Предикатом в этом случае является лямбда-выражение, которое принимает book как аргумент и проверяет, является ли автором книги Shakespeare. Когда передается операция filter, предикат применяется к каждой книге в Stream, и результирующий Stream содержит только те книги, для которых предикат возвращает true.

Специальная операция distinct, позволяет убрать повторяющиеся значения. Следующее выражение удаляет дубликаты из списка:

```
${[9, 2, 2, 3, 4, 7, 7, 0, 2, 2].stream().distinct()}
```

Можно манипулировать значениями в потоке, используя операцию forEach. Подобно filter, forEach принимает лямбда-выражение, которое вычисляется для каждого элемента в Stream. Лямбда-выражение не имеет возвращаемого значения. Это используется для управления значениями в Stream, чтобы каким-то образом их преобразовать.

Один из возможных вариантов использования:

```
${bookStore.stream().forEach(book ->
    book.setLastSold(Instant.now()))}
```

Для операции сортировки Stream<E> принимает java.util.Comparator<E>. Данный интерфейс, может быть представлен лямбда-выражением (E, E) -> int. Лямбда-выражение или Comparator сравнивает два элемента в Stream, используя эффективный алгоритм сортировки, который не определен и специфичен для реализации.

Следующее выражение сортирует книги по автору:

```
${bookStore.stream().sorted((book1, book2) ->
    book1.author.compareTo(book2.author))}
```

Операция sorted — не принимает никакие аргументы. Вместо этого она предполагает, что элементы в Stream реализуют интерфейс java.lang.Comparable.

Можно ограничить количество элементов в Stream, используя операции limit и substream. limit — обрезает Stream после указанного количества элементов. Substream — более полезен для разбивки на пагинацию, поскольку можно указать начальный индекс (включительно) и конечный индекс (не включительно).

```
${bookStore.stream().limit(51)}
${bookStore.stream().substream(16, 35)}
```

Используя операцию map, можно преобразовать элементы в Stream в какой-то другой тип элемента. Операция map принимает mapper, который ожидает один тип элемента и возвращает число. Учитывая Stream<S>, map ожидает лямбда-выражение, с единственным аргументом типа S. Если лямбда-выражение возвращает другой тип R,

результирующий Stream представляет собой Stream<R>. Следующий пример принимает List<Book>, извлекает Stream<Book> и преобразует его в Stream<String>, содержащий только автора книг:

```
${bookStore.stream().map(book -> book.author)}
```

После того как Stream отфильтрован, отсортирован или иным образом трансформирован, необходимо выполнить окончательную операцию, которая преобразует Stream обратно в полезное значение, коллекцию или массив. Эта операция является терминальной, поскольку в отличие от промежуточных операций она не возвращает Stream. Она анализирует промежуточные операции, отложенные по соображениям производительности, а затем преобразует в конечный желаемый результат. Всегда необходимо использовать терминальную операцию.

Операции toArray и toList возвращают массив или List в качестве результата:

```
${bookStore.stream().map(book -> book.author).toArray()}
```

Если Вы выполнили какие-либо отсортированные промежуточные операции в Stream, результирующий массив или List будет содержать в себе выполненные операции. Вы также можете использовать операцию iterator, чтобы вернуть необходимый java.util.Iterator.

Используя операции min, max, average, sum и count — можно агрегировать значения в Stream. Операция count может работать на любом типе Stream, тогда как опера-

ции average и sum требуют, чтобы конечные типы элементов Stream были типа Number.

- Count — возвращает количество элементов в потоке типа long;
- Average — возвращает среднее значение элементов Stream как Optional<? extends Number>;
- Sum — возвращает сумму всех элементов Stream типа Number;
- Optional — заполнитель, который может сообщить, было ли возвращенное значение null и предоставить возвращаемое значение по запросу;
- Min и max — возвращают Optional<E>, где E — тип элемента результирующего Stream. Без каких-либо аргументов эти операции требуют, чтобы элементы Stream реализовали Comparable. Однако при необходимости можно указать аргумент Comparator для этих операций.

Следующие выражения представляют собой некоторые распространенные случаи использования этих агрегирующих терминальных операций:

```
${bookStore.stream().map(book -> book.cost()).max()}
${bookStore.stream().filter(book -> book.tittle ==
    "Gamlet").count()}
```

Операция findFirst возвращает первый элемент в результирующий Stream. Для Stream<E> он возвращает Optional<E>, потому что Stream может быть пустым, то есть нет первого элемента для возврата.

```
${bookStore.stream().filter(book -> book.author ==
    " Shakespeare").findFirst()}
```

Пример использования EL

Теперь настало время доработать приложение сервисной поддержки, используя EL-выражения. Начнем с файла /WEB-INF/jsp/base.jspf, который содержит объявление библиотеки тегов для функций библиотеки JSTL:

```
<%@ page import="com.academy.Card,
    com.academy.Attachment" %>
<%@ taglib prefix="c"
    uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn"
    uri="http://java.sun.com/jsp/jstl/functions" %>
```

Вам не потребуется вносить изменения в код Java. Все изменения будут в JSP. Во *viewCard.jsp* можно заменить некоторые строки. Новый код для этого файла приведен ниже.

```
<%@ page session="false" %>
<%--@elvariable id="cardId" type="java.lang.String"--%>
<%--@elvariable id="card" type="com.academy.Card"--%>
<%
    Card card = (Card)request.getAttribute("card");
%>
<!DOCTYPE html>
<html>
    <head>
        <title>Support service</title>
    </head>
    <message>
        <h1>Card #${cardId}: ${card.topic}</h1>
        <i>Client Name - ${card.clientName}</i><br />
        ${card.message}<br />
        <%
```

```
            if(card.getNumberOfAttachments() > 0)
            {
                %>Attachments: <%
                int i = 0;
                for(Attachment item :
                    card.getAttachments())
                {
                    if(i++ > 0)
                        out.print(", ");
                    %><a href="<c:url value="/cards">
                    <c:param name="action"
                             value="download" />
                    <c:param name="cardId"
                             value="${cardId}" />
                    <c:param name="attachment"
                             value="<%= item.
                             getFileName() %>" />
                    </c:url>"><%= item.getFileName()
                    %></a><%
                }
                %><br /><br /><%
            }
            %>
            <a href="<c:url value="/cards" />">
                Back to other cards</a>
    </message>
</html>
```

Обратите внимание, что новый код имеет аннотацию @elvariable в верхней части для cardId и card и что Java переменная cardId удалена. Но Java переменная card не была удалена, поскольку выражения EL не могут заменить все, для чего используется переменная card — например, итерация над вложениями. Новые EL-выражения выделены жирным шрифтом.

Теперь скомпилируйте и запустите приложение поддержки клиентов и перейдите по адресу *http://localhost:8080/service/* в браузере. Создайте несколько обращений и просмотрите их. Все должно работать так же, как раньше, но теперь EL заботится о некоторых выводах.

15. Cookies и Сессии

Что такое сессии? Цели и задачи сессии

Инструменты, которые были рассмотрены до текущего момента, не позволяют Вам связывать несколько запросов, поступающих от одного и того же клиента, и обмениваются данными между этими запросами. Предположим, что можно использовать IP-адрес как уникальный идентификатор и все запросы с IP-адреса в некоторых временных рамках должны принадлежать одному клиенту. К сожалению, из-за преобразования сетевых адресов (*Network Address Translation*) это ненадежно. Сотни студентов в корпусе академии могут буквально использовать один и тот же IP-адрес, скрытый за NAT-маршрутизатором. По этой причине концепция HTTP-сессий стали использоваться всеми HTTP-серверными технологиями, а Java EE поддерживает сессии, описанные в ее спецификации.

Не каждое приложение нуждается в сессиях. Примеры Hello World, безусловно, не нуждаются в сессиях.

Если взять, к примеру, приложение сервисного центра, в котором не применяются сессии — оно будет похоже на анонимную доску объявлений. Но если думать, о требованиях организации, которая имеет сайт сервисной поддержки, можно быстро понять, что в какой-то момент необходимо создать учетные записи пользователей, и эти пользователи должны авторизоваться в приложении. Служба поддержки запросов может

содержать личную информацию, которая не должна быть доступна другим клиентам. Конечно, нужен способ ограничить доступ к определенным обращениям в поддержку, чтобы только клиент, разместивший свое обращение, и члены команды поддержки могли получить доступ к нему. Можно было бы запрашивать имя и пароль пользователя на каждой странице, к которой они обращаются, но это плохая идея, клиенты не будут довольны этим решением.

Сессии используются для поддержки состояния между одним запросом и другим. HTTP-запросы полностью автономны. С точки зрения сервера, запрос начинается, когда веб-браузер пользователя открывает сокет для сервера, и он заканчивается, когда сервер отправляет последний пакет обратно для клиента и закрывает соединение. В этот момент больше нет связи между браузером и сервером, и когда приходит следующее соединение, невозможно связать новый запрос с предыдущим запросом.

Приложения часто не могут нормально функционировать, будучи автономными. Классическим примером является интернет-магазин. На сегодняшний день почти каждый интернет-магазин требует имя пользователя и пароль перед покупкой, но рассмотрим даже те магазины, которые этого не делают. При просмотре в магазине, Вы найдете товар, который Вам нравится, поэтому Вы добавляете этот товар в свою корзину покупок. Вы продолжаете просмотр магазина и находите другой товар и также добавляете его в свою корзину. Когда Вы просматриваете свою корзину покупок, Вы видите, что оба ранее добавленных товара остаются в корзине. Так или иначе,

веб-сайт знает, что все Ваши запросы поступают из одного и того же браузера на одном компьютере. Никто не может видеть Вашу корзину или ее содержимое — она исключительно привязана к Вашему компьютеру и браузеру. Этот сценарий является аналогом реального опыта покупок. Вы входите в свой любимый продовольственный магазин, и берете корзину (получаете сессию с сервера). Проходите по магазину, берете продукты и кладете их в свою корзину (добавляете их в сессию). На кассе, Вы достаете продукты из корзины и отдаете их кассиру, который сканирует их и принимает Ваши деньги (проверяете использование своей сессии). При выходе из магазина, Вы возвращаете свою корзину (закрываете свой браузер или выходите из системы, заканчивая сессию).

В этом примере корзина поддерживает Ваше состояние, пока Вы находитесь в магазине. Без корзины, ни Вы, ни магазин не можете быть в курсе о Ваших товарах для покупки. Если не поддерживать состояние между запросами, Вам придется заходить, брать один продукт, платить за него, и выходить (завершить запрос) и повторять весь процесс с каждым продуктом, который Вы хотите приобрести. Сессии — это механизм поддержки состояния между запросами.

Еще один сценарий для рассмотрения — это веб-сайт форума пользователей. Почти повсеместно в онлайн-форумах пользователи известны своими именами так называемыми «никнеймами». Когда пользователь заходит на форум, он входит в систему, предоставляя имя пользователя и пароль, чтобы подтвердить его личность. С этого момента он может добавить обсуждения на фо-

руме, учувствовать в обсуждениях с другими пользователями, отвечать на личные сообщения, сообщать о темах или отвечать модераторам и, возможно, пометить темы как избранные. Важно обратить внимание на то, что пользователь вошел в систему только один раз в течение всей этой временной шкалы. Системе нужен способ запоминать пользователя между каждым запросом, и сессия позволяет это сделать.

Пользователям зачастую требуются веб-приложения, использование, которых позволило бы автоматизировать выполнение задач для некоторых бизнес-процессов. Рассмотрим пример с созданием новостной статьи для публикации на сайте. Журналист сначала может перейти на экран, где он имеет возможность ввести название, слоган, тело статьи и форматировать элементы соответственно. На следующей странице он может выбрать фотографии и указать, как они должны размещаться. Также может загружать или записывать видео, которые будут помещены в статью. И наконец, ему будет представлен список похожих статей или поле поиска, чтобы их найти, для того чтобы указать, какие из них следует помещать в поле «Похожие новости». После того, как все эти этапы будут завершены, статья будет опубликована.

Весь этот сценарий представляет собой идею бизнес-процесса. Бизнес-процесс содержит в себе множество этапов, каждый из которых является частью одной задачи. Чтобы связать все эти этапы вместе, для завершения бизнес-процесса, необходимо поддерживать связь между запросами.

Что такое cookies? Цели и задачи cookies

Теперь, когда Вы понимаете важность сессий, важно разобраться в том, как они работают. Для определения этого существует две разных идеи. Во-первых, общая теория о веб-сессиях и о том, как они реализованы. И, во-вторых, особенности реализации сессий в Java EE web-приложениях.

В общей теории веб-сессий, сессия представляет собой некоторый файл, сегмент памяти, объект или контейнер управляемый сервером или веб-приложением, который содержит различные элементы данных, назначенные ему. Эти элементы данных могут быть именем пользователя, корзиной покупок, деталями бизнес-процесса и т.д. Пользовательский браузер не сохраняет и не поддерживает эти данные. Он управляется исключительно сервером или кодом веб-приложения. Единственный недостающий элемент — это связь между этим контейнером и браузером. С этой целью сессиям назначается произвольно сгенерированная строка, называемая идентификатором сессии. В первый раз сессия создается в результате получения запроса, идентификатор сессии передается обратно в браузер в качестве части ответа. Каждый последующий запрос от браузера включает идентификатор сессии. Когда приложение получает запрос с ID сессии, он может затем связать существующий сеанс с этим запросом. Это показано на рисунке 1.

Оставшаяся проблема, которая будет решена, заключается в том, как идентификатор сессии передается с сервера на браузер и обратно. Для этого используются два метода: Cookies сесии и переписывание URL-адресов

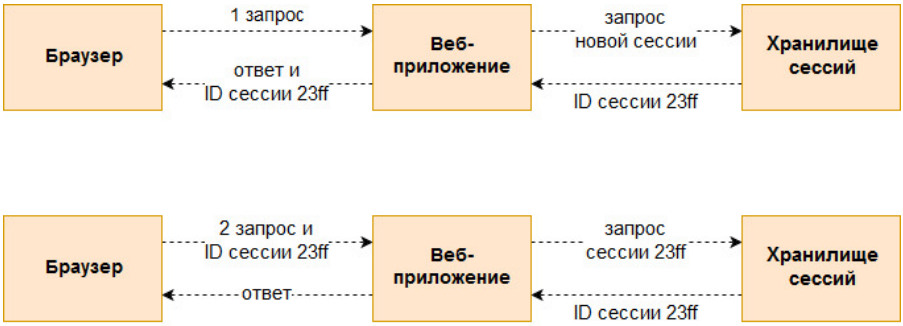

Рисунок 1.

В HTTP 1.1 уже существует решение, которое позволяет серверам отправлять ID сессий обратно для браузеров, чтобы браузеры включали идентификаторы сессий в будущих запросах. Это технология HTTP называется Cookies. Cookie файлы — это механизм, посредством которого произвольные данные могут передаваться с сервера на браузер через Set-Cookie ответный заголовок, хранится локально на компьютере пользователя, а затем передается обратно из браузера на сервер через Cookie-заголовок запроса. Файлы cookie могут иметь различные атрибуты, такие как имя домена, путь, срок действия или максимальный возраст, флаг безопасности и флаг только для HTTP.

Атрибут Domain — указывает браузеру, для каких доменных имен он должен отправить cookie обратно, тогда как атрибут Path позволяет файлу cookie дополнительно ограничиваться определенным URL-адресом относительно домена. Каждый раз, когда браузер делает запрос любого типа, он находит все файлы cookie, соответствующие домену и пути для сайта и отправляет эти cookie вместе с запросом. Expires определяет дату истечения срока дей-

ствия файла cookie, тогда как атрибут Max-Age определяет количество секунд до истечения срока действия файла cookie. Если дата истечения срока действия файла cookie прошла, браузер немедленно удаляет его. Если cookie не имеет атрибут Expires или Max-Age, он удаляется при закрытии браузера. Если атрибут Secure присутствует (не требует значения) браузер отправит файл cookie назад только через HTTPS. Это позволяет зашифровать cookie. В заключение, атрибут HttpOnly ограничивает cookie для прямого запроса браузера. Другие технологии, такие как JavaScript и Flash не будут иметь доступ к файлу cookie.

Веб-серверы и серверы приложений используют файлы cookie для хранения идентификаторов сессий на стороне клиента, чтобы они могли быть переданы обратно на сервер с каждым запросом. С серверами приложений Java EE имя этого cookie JSESSIONID по умолчанию.

Set-Cookie заголовки в ответах используются для отправки файлов cookie в браузер пользователя на хранение. Аналогичным образом, cookie-заголовки в запросах используются для отправки файлов cookie обратно на веб-сервер. Представим: пользователь переходит на некоторый сайт поддержки и перенаправляется на страницу входа. При перенаправлении, браузер пользователя также получает cookie идентификатора сессии с сервера. Когда браузер переходит на страницу входа в систему, он включает в себя файл cookie идентификатора сессии в своем запросе. С этого момента каждый раз, когда браузер отправляет новый запрос, он включает в себя JSESSIONID cookie. Сервер не отправляет его снова, так как знает, что браузер уже имеет его.

После успешного входа в систему сервер также отправляет обратно remusername cookie — метод, который использует сайт для автоматического заполнения имени пользователя всякий раз, когда он переходит на страницу входа в систему. Будущие запросы всегда будут содержать этот файл cookie. Хотя, будущее ответы не сбрасывают его. Обратите внимание, что JSESSIONID cookie не имеет срока годности, тогда как remusername cookie имеет. Remusername cookie истекает в 2021 году, тогда как JSESSIONID cookie истечет, как только пользователь закроет свой браузер.

Уязвимости сессий

Сессии имеют уязвимости, которые могут вызвать серьезные проблемы для пользователей, и если приложение оперирует конфиденциальной или личной информацией (например, номера кредитных карт), это может повлечь огромные потери для бизнеса. Существуют простые способы устранения этих уязвимостей. Разработчик должен быть всегда внимательным и хорошо информированным в вопросах безопасности. В критически важных и чувствительных приложениях было бы разумно использовать коммерческий сканер какого-либо типа, который сканирует приложение на наличие слабых мест.

Ошибка копирования и вставки

Возможно, одним из самых простых способов, с помощью которого можно скомпрометировать сессию, являются действия пользователя, когда он копирует и вставляет URL-адрес из своего браузера в электронную почту,

публикацию форума или другую общественную зону. Если пользователь решит поделиться страницей в приложении со своими друзьями и копипастит URL-адрес из адресной строки — все увидят ID сессии, который включен в URL. Если кто-то перейдет по URL до истечения срока действия сессии — он присвоит личность пользователя, который поделился URL-адресом. Таким образом, друзья пользователя могут случайно увидеть его личную информацию. Более опасный сценарий заключается в том, что злоумышленник может найти эту ссылку и использовать ее для захвата сессии пользователя. Затем может изменить адрес электронной почты учетной записи, получить ссылку на сброс пароля и наконец, изменить пароль.

Источником возникновения этой проблемы является — копирование и вставка URL-адреса пользователя с его адресной строки. Единственным безошибочным методом устранения этой уязвимости является полностью отключить внедрение ID сессий в URL-адреса.

Фиксация сессии

Атака фиксации сессии похожа на ошибку копирования и вставки, за исключением того, что «ничего не подозревающий пользователь» в этом случае является злоумышленником, а жертвами являются пользователи, которые используют ссылку, содержащую ID сессии. Злоумышленник может перейти на какой-либо веб-сайт, который, принимает идентификаторы сессии, встроенные в URL-адрес. Получит ID сессии, а затем отправит URL-адрес, содержащий этот идентификатор сессии, жертве,

через форум или электронное письмо. На этом этапе, когда пользователь нажимает на ссылку, чтобы перейти на сайт, его ID сессия привязана к тому идентификатору сессии что был в URL-адресе, о котором знает злоумышленник. Если пользователь логинится на веб-сайте во время этой сессии, злоумышленник также будет залогирован, потому что он распространил ID сессии, предоставляя доступ к учетной записи пользователя.

Существует два способа решения проблемы:

1. Как и в случае ошибки копирования и вставки, можно просто отключить вложение идентификаторов сессии в URL-адрес, а также запретить приложению принимать идентификаторы сессии через URL-адреса.
2. Использовать миграцию сессии (сеанс работы) после входа в систему. Когда пользователь входит в систему, изменяя идентификатор сессии, или копирует данные сессии в новую сессию и аннулирует исходную сессию. У злоумышленника все еще есть исходный идентификатор сессии, который больше не действителен и не подключен к сессии пользователя.

Межсайтовый скриптинг и перехват сессий

Существует еще одна форма перехвата сессии, которая использует JavaScript для чтения содержимого файла cookie. Злоумышленник, который использует уязвимость сайта для межсайтовых скриптовых атак, вставляет JavaScript на страницу, чтобы прочитать содержимое cookie с ID сессии с использованием свойства JavaScript DOM document.cookie. После того, как злоумышленник получает ID сес-

сии от ничего не подозревающего пользователя, он может похитить этот идентификатор сессии, создав cookie-файл его ПК или с использованием встраивания URL-адресов, тем самым присвоить личность жертвы.

Обеспечить защиту сайта от межсайтового скриптинга сложно, и злоумышленники постоянно находят новые способы осуществления кросс-атак сценариев сайта. Альтернативная защита, которую нужно всегда использовать — помечать все ваши cookie-файлы с HttpOnly атрибут. Этот атрибут позволяет использовать cookie только когда браузер выполняет HTTP (или HTTPS) запрос, независимо от того, происходит ли это запрос через ссылку, ручной ввод URL, отправку формы или запрос AJAX. Важно то, что HttpOnly полностью отключает возможности JavaScript, Flash или других скриптов или плагинов браузера, чтобы получить содержимое файла cookie. Это останавливает межсайтовый скриптинг от атак на сессии. Файлы cookie с идентификатором сессии должны всегда включать HttpOnly атрибут.

Небезопасные файлы cookie

Последней уязвимостью, которая будет рассмотрена, является атака «злоумышленник в середине» (*man-in-the-middle attack*). Классическая атака перехвата данных, при которой злоумышленник наблюдает за запросом или ответом, когда тот транспортируется между клиентом и сервером и получает информацию от запроса или ответа. Эта атака привела к появлению уровня защищенных сокетов и безопасности транспортного уровня (SSL/TLS), что является основой протокола HTTPS. Защита

веб-трафика с помощью HTTPS эффективно предотвращает атаку MitM и предотвращает кражу файлов cookie ID сессий. Проблема, в том, что пользователь может сначала попробовать перейти на сайт с помощью протокола HTTP. Даже если он будет перенаправлен на HTTPS, урон уже произошел: браузер пользователя передал cookie ID сессии на незашифрованный сервер и наблюдающий злоумышленник может украсть ID сессии.

Флаг Secure был создан для решения этой проблемы. Когда сервер отправляет ID сессии cookie клиенту в ответе, он устанавливает Secure флаг. Это сообщает браузеру, что cookie должен передаваться только через HTTPS. С этого момента cookie будет передаваться только зашифрованным, и злоумышленники не смогут его перехватить. Чтобы это работало, сайт должен всегда находиться за HTTPS. В противном случае, как только пользователь будет перенаправлен на HTTP, браузер не сможет более передавать файл cookie, и сессия будет потеряна. По этой причине необходимо взвесить безопасность потребностей приложения и определить, являются ли данные, которые Вы защищаете, достаточно чувствительными, чтобы гарантировать накладные расходы на производительность и сложность обеспечения каждого запроса с помощью HTTPS.

Самая сильная возможная защита

Одним из заключительных вариантов, которые важно понимать при работе с безопасностью сессий, является SSL/TLS ID сессии. Чтобы повысить эффективность протокола SSL, устраняя необходимость выполнения SSL-

рукопожатия по каждому запросу, протокол SSL определяет свой собственный ID сессии. SSL идентификатор сессии устанавливается во время установления связи SSL, а затем используется в последующих запросах для связывания запросов для определения того, какие ключи следует использовать для шифрования и дешифрования. Эта концепция дублирует понятие идентификатора сессии HTTP. Однако ID сессии SSL не передаётся при использовании файлов cookie или URL-адресов и является более безопасным. Получить идентификатор сессии SSL для которого Вы не авторизованы очень трудно. Некоторые сайты с повышенной безопасностью, например, различные финансовые учреждения, используют идентификатор сессии SSL в качестве протокола HTTP идентификатора сессии, тем самым, исключая использования файлов cookie, а также URL шифрование сохраняя состояния между запросами.

Это самый безопасный способ установления ID сессии по всем запросам и почти неуязвимый. Кроме того, при обнаружении уязвимостей SSL они обычно обрабатываются в течение нескольких недель и устраняются обновлениями браузера.

Настройка сессий в дескрипторе развертывания

Во многих случаях, HTTP-сессии готовы к использованию в Java EE и не требуют явной конфигурации. Однако в целях безопасности их следует сконфигурировать. Настраиваются сессии в дескрипторе развертывания, используя тег <session-config>. Внутри этого тега можно настроить метод отслеживания сессий, возраст истечения

срока ожидания сессии и сведения о файле cookie, идентификатор сессии, если используется. Многие из них имеют значения по умолчанию, которые вряд ли понадобится изменять. Следующий код демонстрирует все возможные параметры дескриптора развертывания для сессий.

```
<session-config>
    <session-timeout>30</session-timeout>
    <cookie-config>
        <name>JSESSIONID</name>
        <domain>example.org</domain>
        <path>/example</path>
        <comment><![CDATA[Keeps you logged in.
             See our privacy policy for
             more information.]]>
        </comment>
        <http-only>true</http-only>
        <secure>false</secure>
        <max-age>1800</max-age>
    </cookie-config>
    <tracking-mode>COOKIE</tracking-mode>
    <tracking-mode>URL</tracking-mode>
    <tracking-mode>SSL</tracking-mode>
</session-config>
```

Все теги в <session-config> а также <cookie-config> являются необязательными, но они должны отображаться в порядке, показанном в примере (за исключением пропущенных тегов). Тег <session-timeout> указывает, как долго сессии должны оставаться неактивными до того, как они будут признаны недействительными. Если значение равно 0 или меньше, сессия никогда не истекает. Если этот тег опущен, применяется значение по умолчанию для контейнера. По умолчанию контейнер Tomcat — 30, кото-

рый можно изменить в конфигурации Tomcat. В примере время ожидания составляет 30 минут. Каждый раз, когда пользователь с определенным идентификатором сессии делает запрос к приложению, таймер сбрасывает неактивную сессию. Если прошло более 30 минут без запроса, его сеанс считается недействительным, и ему дается новая сессия. Тег <tracking-mode>, который был добавлен в Servlet 3.0/Java EE 6, указывает какой метод должен использовать контейнер для отслеживания идентификаторов сессий. Допустимые значения:

1. **URL** — контейнер содержит только идентификаторы сессии в URL-адресах. Он не использует файлы cookie или идентификаторы сессии SSL. Этот подход не очень безопасен.
2. **COOKIE** — в контейнере используются файлы cookie сессии для отслеживания ID сессий. Этот метод достаточно безопасен.
3. **SSL** — контейнер использует ID сессий SSL в качестве ID сессий HTTP. Этот метод является наиболее безопасным подходом, но требует, чтобы все запросы были HTTPS для правильной работы.

Можно использовать <tracking-mode> более одного раза, чтобы сообщить контейнеру, что он может использовать несколько стратегий. Например, если указать оба COOKIE и URL, контейнер предпочтет файлы cookie, но будет использовать URL-адреса, когда файлы cookie недоступны. Указание COOKIE как единственного режима отслеживания указывает контейнеру никогда не встраивать сеансы в URL-адреса и всегда предполагать, что у пользователя есть файлы

cookie. Аналогично, указав URL как единственный режим отслеживания — Вы запретите контейнеру использование cookie. Если включен SSL режим отслеживания, также невозможно включить COOKIE или URL режимы. SSL-идентификаторы сессий должны использоваться сами по себе, то есть контейнер не может использовать файлы cookie или URL-адреса при отсутствии HTTPS.

Тег <cookie-config> применяется только тогда, когда COOKIE указан как один из режимов отслеживания. Теги внутри него настраивают файлы cookie, которые контейнер возвращает в браузер:

1. **<name>** позволяет настроить имя файла cookie сессии. По умолчанию используется JSESSIONID.
2. **<domain>** и **<path>** соответствуют домену и пути атрибутов файла cookie. Веб-контейнер использует их по умолчанию, поэтому их настраивать не нужно. Domain по умолчанию имя домена, используемое для выполнения запроса, в течение которого была создана сессия. Path по умолчанию использует имя развертываемого приложения.
3. **<comment>** добавляет атрибут комментарий в cookie идентификатора сессии, предоставляющий возможность добавить произвольный текст. Это часто используется для объяснения цели файла cookie и указания пользователей на политику конфиденциальности сайта.
4. **<http-only>** и **<secure>** теги соответствуют атрибутам HttpOnly и Secure cookie и оба false по умолчанию. Для повышения безопасности всегда нужно устанавливать <http-only> в true, <secure> следует изменить на true только если включен HTTPS.

5. **<max-age>** определяет атрибут cookie Max-Age, который контролирует время истечения срока действия файла cookie. По умолчанию cookie не имеет срока истечения, это означает, что он истекает, когда браузер закрывается. Установка этого параметра в -1 имеет тот же эффект. Это значение настраивается в секундах, но это может привести к истечению срока действия файла cookie и сбою сеанса, когда пользователь находится в середине активного использования приложения. Лучше не использовать этот тег.

Примеры использования сессий

Интегрируем сессии в службу поддержки клиентов. Добавим авторизацию пользователя в приложении.

Для того чтобы использовать сессии, необходимо удалить атрибут session = "false" из всех JSP в приложении поддержки клиентов. Также нужно добавить XML-файл <session-config> в дескриптор развертывания, чтобы сессии были настроены для лучшей безопасности, а идентификаторы сессий не попадали в URL-адреса.

```
<session-config>
    <session-timeout>30</session-timeout>
    <cookie-config>
        <http-only>true</http-only>
    </cookie-config>
    <tracking-mode>COOKIE</tracking-mode>
</session-config>
```

На данном этапе должно быть, очевидно, что приложение поддержки клиентов нуждается в некоторой фор-

ме пользовательской базы данных с учетными записями. Сейчас добавим элементарную, незащищенную функцию логина для приложения.

Добавим класс LoginServlet в приложение и создадим в нем статическую базу данных:

```
@WebServlet(
        name = "loginServlet",
        urlPatterns = "/login"
)
public class LoginServlet extends HttpServlet
{
    private static final Map<String,
            String> clientBase = new LinkedHashMap<>();
    static {
        clientBase.put("Ivanov", "qwert");
        clientBase.put("Petrov", "123456");
        clientBase.put("Sidorov", "master");
    }
}
```

База данных клиентов представляет собой простую Map имен и паролей. Клиенты могут либо получить доступ к системе, либо нет, а пароли не хранятся безопасным образом.

Метод getSession для HttpServletRequest имеет несколько перегрузок: getSession() а также getSession(boolean). Вызов getSession() вызывает getSession(true), который возвращает существующую сессию, если она существует, и создает новую — если еще не существует (никогда не возвращает null). Вызов getSession(false) возвращает существующую сессию, если она существует, и null, если — отсутствует.

Если нужно проверить была ли создана сессия — необходимо вызывать getSession с аргументом false. Метод getAttribute возвращает объект, сохраненный в сессии. У него есть аналог getAttributeNames, который возвращает перечисление имен всех атрибутов в сессии. Метод setAttribute связывает объект с сессией.

Метод doGet отвечает за отображение формы входа в систему, создадим его.

```
@Override
protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
{
    HttpSession session = request.getSession();
    if(session.getAttribute("clientName") != null)
    {
        response.sendRedirect("cards");
        return;
    }
    request.setAttribute("loginFailed", false);
    request.getRequestDispatcher(
        "/WEB-INF/jsp/view/login.jsp")
        .forward(request, response);
}
```

Первое, что делает метод в предыдущем примере — проверяет, вошел ли клиент в приложение (clientName атрибут существует) и перенаправляет на экран со страницу обращений, если это так. Если клиент не вошел в систему, устанавливается атрибут loginFailed в false и перенаправляет запрос на JSP входа в систему. Когда форма входа в JSP отправлена, она направляется в метод doPost:

```
@Override
protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
{
    HttpSession session = request.getSession();
    if(session.getAttribute("clientName") != null)
    {
        response.sendRedirect("cards");
        return;
    }

    String username = request.getParameter("clientName");
    String password = request.getParameter("password");
    if(username == null || password == null ||
        !LoginServlet.clientBase.
        containsKey(username) ||
        !password.equals(LoginServlet.clientBase.
        get(username)))
    {
        request.setAttribute("loginFailed", true);
        request.getRequestDispatcher(
                "/WEB-INF/jsp/view/login.jsp")
                .forward(request, response);
    }
    else
    {
        session.setAttribute("clientName", username);
        request.changeSessionId();
        response.sendRedirect("cards");
    }
}
```

В методе doPost нет ничего нового. Он снова проверяет, что пользователь еще не выполнил вход в систему, а затем проверяет имя пользователя и пароль в базе дан-

ных. Если данные не совпадают, он устанавливает атрибут loginFailed в true и отправляет пользователя обратно в JSP входа. Если учетные данные верны, он устанавливает атрибут clientName в сессии, изменяет ID сессии, а затем перенаправляет пользователя на экран запроса. Метод changeSessionId — защищает от атак фиксации сессии.

Создайте */WEB-INF/jsp/view/login.jsp* и вставьте в него форму входа:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Support service</title>
    </head>
    <body>
        <h1>Login</h1>
        Log in, please.<br />
        <%
        if(((Boolean)request.getAttribute("loginFailed")))
        {
            %>
            <b>The username or password incorrect.
            </b><br />
            <%
        }
        %>
        <form method="POST"
              action="<c:url value="/login" />">
            Client Name<br />
            <input type="text" name="clientName" /><br />
            Password<br />
            <input type="password" name="password" /><br />
            <input type="submit" value="Log In" />
        </form>
    </body>
</html>
```

Эта простая страница отображает форму входа и, используя атрибут loginFailed, уведомляет клиентов о том, что их учетные данные были отклонены. Вместе с LoginServlet, завершается функция входа. Однако это не мешает пользователям получать доступ к экранам запросов. Необходимо добавить проверку в CardServlet, чтобы убедиться, что пользователи вошли в систему, прежде чем отображать информацию о запросе или разрешить им отправлять запросы. Это легко осуществить, добавив следующий код в начало doGet и doPost методы в CardServlet:

```
if(request.getSession().
    getAttribute("clientName") == null)
{
    response.sendRedirect("login");
    return;
}
```

Теперь, когда пользователи регистрируются перед созданием запросов, Вам больше не нужно поле имя в форме запроса. В методе createCard сервлета CardServlet, необходимо изменить код, который устанавливает имя клиента запроса с помощью параметра запроса, так что он теперь использует имя пользователя из сессии, как показано в следующем коде. Также можно удалить поле ввода «Client Name» из */WEB-INF/jsp/view/cardForm.jsp*.

```
card.setClientName(
     (String)request.getSession().
     getAttribute("clientName")
);
```

Теперь, когда приложение требует входа в систему, выполним следующие действия, чтобы протестировать его:

1. Скомпилируем проект с помощью среды IDE.
2. Перейдем к приложению в браузере (*http://localhost:8080/service/*), и Вы сразу же должны быть отправлены на страницу входа в систему.
3. Попробуйте войти в систему с неправильным именем пользователя и паролем. Вам будет запрещен вход.
4. Попробуйте верные имя пользователя и пароль, и Вы должны переместится в список запросов.
5. Создайте несколько обращений, как Вы делали это ранее, и ваше имя пользователя должно быть прикреплено к ним.
6. Закройте браузер, повторно откройте его и снова войдите в систему, используя другое имя пользователя и пароль.
7. Создайте новый запрос, и Вы увидите, что новый запрос имеет имя пользователя, с которым Вы в настоящее время вошли в систему, а старые запросы связаны с другим пользователем.

При тестировании Вам пришлось закрыть браузер для выхода из приложения поддержки клиентов. Эта довольно таки неудобно при использовании приложения. Добавление ссылки для выхода из системы достаточно тривиально. Во-первых, настройте код в верхней части метода doGet в LoginServlet, чтобы добавить поддержку выхода из системы:

```
HttpSession session = request.getSession();
if(request.getParameter("logout") != null)
{
    session.invalidate();
    response.sendRedirect("login");
    return;
}
else if(session.getAttribute("clientName") != null)
{
    response.sendRedirect("cards");
    return;
}
```

Единственное, что Вам нужно сделать, это добавить ссылку для выхода в верхнюю часть *listcards.jsp*, *cardForm.jsp*, а также *viewCard.jsp* в */WEB-INF/jsp/view*, чуть выше заголовков <h1>:

```
<a href="<c:url value="/login?logout" />">Logout</a>
```

Теперь снова запустите, и войдите в свое приложение. На каждой странице Вы должны увидеть ссылку для выхода из системы. Нажмите на ссылку выхода из системы, и Вы вернетесь на страницу входа в систему, что означает, что Вы успешно вышли из системы.

16. JSP-теги и Java Standard Tag Library

Понятие Tag. Работа с тегами

JSP-теги — это особый синтаксис технологии JavaServer Pages, который выглядит как любой обычный тег HTML или XML. JSP-теги также называются actions. Тег JSP выполняет некоторые действия, такие как создание или ограничение вывода. Теги JSP требуют правильной ссылки на пространство имен XML так как они находятся за пределами любого стандартного тега, указанного в HTML. Даже опытным программистам иногда бывает нелегко придерживаться строгого синтаксиса XML-документа. Таким образом, идея синтаксиса тегов JSP содержит псевдонимы для упрощения написания JSP. Первое из них — директива taglib.

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

Эта директива является альтернативой атрибута XMLNS ссылки на пространства имен XML в XML-документах:

```
<jsp:root xmlns="http://www.w3.org/1999/xhtml"
          version="2.0"
xmlns:jsp="http://java.sun.com/JSP/Page"
xmlns:c="http://java.sun.com/jsp/jstl/core"
xmlns:fn="http://java.sun.com/jsp/jstl/functions">
```

Использование этой директивы предотвращает синтаксический анализ XML-кода, но разработчик должен соблюдать стандарты XML. Вместо этого JSP-движок в веб-контейнере понимает и знает, как анализировать JSP синтаксис, также все основные Java IDE могут сообщать о синтаксических ошибках и других проблемах JSP.

Атрибут prefix в директиве taglib (или пространстве имен XML) представляет пространство имен, на которое ссылается библиотека тегов на странице JSP. Префикс тега, рекомендуемый в файле дескриптора библиотеки тегов (*Tag Library Descriptor* — TLD) для этой библиотеки тегов, объявляется в директиве taglib с использованием атрибута prefix.

Атрибут uri указывает URI, определенный в TLD для этой библиотеки тегов. Таким образом, парсер JSP находит соответствующий TLD для указанной библиотеки тегов.

Когда JSP-парсер сталкивается с директивой taglib, он находит файл TLD для этой библиотеки тегов, используя URI, производит поиск в разных местах. Эти местоположения указаны в спецификации JSP следующим образом (от наивысшего до самого низкого приоритета):

1. Если контейнер совместим с Java EE, парсер ищет любые соответствующие файлы TLD, которые являются частью спецификации Java EE, включая библиотеку тегов JSP, стандартную библиотеку тегов Java (JSTL) и любые библиотеки JavaServer Faces.
2. Затем он проверяет явные объявления <taglib> в разделе <jsp-config> дескриптора развертывания.
3. Если парсер все еще не нашел соответствующий файл TLD, он проверяет все файлы TLD, содержащиеся в ка-

талоге META-INF, из любых файлов JAR, размещенных в каталоге приложения /WEB-INF/lib, или любых файлов TLD, размещенных в каталоге приложения /WEB-INF или в любых подкаталогах /WEB-INF, рекурсивно.

4. Наконец, парсер проверяет любые другие файлы TLD, которые встроены как часть веб-контейнера или сервера приложений.

Явное объявление <taglib> обычно не требуется, за исключением если указанный TLD не содержит URI, он не находится в одном из других ранее размещенных мест, или Вам необходимо переопределить TLD с конфликтующим URI, предоставленным отдельным JAR-файлом, который Вы не контролируете. Явные объявления <taglib> выглядят следующим образом:

```
<jsp-config>
...
<taglib>
    <taglib-uri>
        http://www.example.org/xmlns/jsp/client
    </taglib-uri>
    <taglib-location>/tld/client.tld</taglib-location>
</taglib>
...
</jsp-config>
```

В этом примере значение http://www.example.org/xmlns/jsp/client в теге <taglib-uri> будет сравниваться с атрибутом uri директивы taglib. Если они совпадут, он будет использовать указанный TLD (/tld/client.tld) по отношению к корню веб-приложения. Обратите внимание, что в этой конфигурации не указан префикс.

Так как это не объявление библиотеки тегов, как в директиве taglib. Это путь, который указывает контейнеру, где хранится файл TLD для указанной URI библиотеки тегов. Использование явных объявлений <taglib> всегда можно избежать.

После того, как директива taglib правильно настроена для доступа к TLD, можно использовать теги внутри этой библиотеки в JSP. Все теги JSP следуют одному и тому же базовому синтаксису:

```
<prefix:tagname[ attribute=value[ attribute=value[ ...]]]/>
<prefix:tagname[ attribute=value[ attribute=value[ ...]]]>
    content
</prefix:tagname>
```

В этой синтаксической нотации prefix обозначает префикс библиотеки тегов JSP, также известный как пространство имен (которое является стандартной номенклатурой XML). Tagname — это имя тега, как определено в TLD. Значения атрибутов обрамляются либо одинарными кавычками, либо двойными. Между атрибутами должно быть пустое пространство, но в самозакрывающемся теге пробел перед /> является необязательным. Все теги JSP, чтобы быть валидными должны иметь либо самозакрывающий тег (<prefix:tagname />), либо открывающий и закрывающий теги соответственно (<prefix:tagname></prefix:tagname>). Использование открывающих тегов без закрывающих тегов приведет к синтаксической ошибке (<prefix:tagname>).

Есть полные, Java EE-совместимые серверы приложений, и есть более ограниченные веб-контейнеры Java EE.

Серверы приложений реализуют всю спецификацию Java EE, тогда как веб-контейнеры реализуют спецификации Servlet и JSP и возможно несколько других спецификаций, которые разработчики веб-контейнера считали важными. Большинство веб-контейнеров также реализуют спецификацию EL, поскольку она была частью спецификации и сегодня остается неразрывно связанной со спецификацией JSP. Все веб-контейнеры поддерживают библиотеки тегов с JSP, потому что эта поддержка является частью спецификации JSP. Однако некоторые веб-контейнеры не реализуют спецификацию JSTL, поскольку библиотеки конкретных тегов в JSTL легко отделяются от общей концепции библиотек тегов. Tomcat является одним из этих веб-контейнеров, и по сей день он не реализует JSTL. Однако это не означает, что нельзя использовать JSTL в приложениях, которые планируется развертывать в Tomcat.

Ранее при разработке приложения для сервисного центра были добавлены зависимости Maven. Одной из них была JSP API, которая позволяет скомпилировать функции JSP в среде IDE. Другая зависимость для API Servlet. Эти зависимости Maven обладают «предоставленной» областью, поскольку Tomcat уже включает библиотеку API JSP, и поэтому ее не нужно включать в развернутое приложение. Другими двумя добавленными зависимостями были JSTL API (интерфейсы, абстрактные классы и описания тегов для JSTL) и реализация JSTL, предоставляемые GlassFish (JLD TLD, конкретные классы и реализации интерфейсов). Если Tomcat предоставил JSTL-реализацию, Вам все равно понадобятся зависимости Javen Maven, но они будут иметь «предоставленную» область. Поскольку

Tomcat не обеспечивает реализацию JSTL, эти библиотеки находятся в области «компиляции», поэтому они развертываются вместе с приложением. Это позволяет использовать JSTL в приложении, несмотря на отсутствие у Tomcat реализации JSTL.

Различные виды Tags

В спецификации библиотеки стандартных тегов Java есть пять библиотек тегов:

- Core (c)
- Formatting (fmt)
- Functions (fn)
- SQL (sql)
- XML (x).

Библиотеку функций мы рассмотрели, когда изучали язык выражений (EL). В качестве дополнительной информации Вы можете просмотреть документацию TLD для JSTL 1.1 Java EE 5 по ссылке https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/. К сожалению, нет доступной документации для JSTL 1.2 Java EE 6, но изменения между этими версиями были незначительными. Никаких новых тегов добавлено не было — только новые пояснения в спецификации. В Java EE 7 не было новой версии JSTL.

Core Tags

Библиотека тегов Core содержит почти все основные функции, необходимые для замены кода Java в JSP. Она включает в себя инструменты для условных и циклических конструкций, итерации и вывода содержимого. Вы

уже видели пару тегов из библиотеки Core, поэтому Вы должны быть знакомы с ее директивой taglib:

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

Тег <c:out> наиболее используемый тег, его цель — отобразить контент JSP. На первый взгляд, это не отличается от простого использования выражения EL для вывода содержимого. Более запутанным является то, что <c:out> почти всегда используется с одним или несколькими EL-выражениями.

```
<c:out value="${someVariable}" />
```

Хотя <c:out> вполне может быть эквивалентно простому написанию ${someVariable}, есть некоторые отличия. Во-первых, в этом случае использование <c:out> фактически эквивалентно ${fn:escapeXml(someVariable)}. Это связано с тем, что по умолчанию <c:out> экранирует зарезервированные символы XML (<,>, ‹, " и &), как и fn:escapeXml. Можно отключить это, установив для атрибута escapeXml значение false:

```
<c:out value="${someVariable}" escapeXml="false" />
```

Но этого делать не стоит, экранирование зарезервированных символов XML помогает защитить сайт от межсайтового скриптинга и различных атак при помощи инъекции, а также предотвратить нарушения функциональности сайта в следствии появления неожиданных специальных символов. Атрибут default указывает зна-

чение по умолчанию, если значение, указанное в атрибуте value, равно null.

```
<c:out value="${someVariable}" default="None value" />
```

Атрибут default может также содержать EL-выражение.

```
<c:out value="${someVariable}" default="${otherValue}" />
```

Вместо атрибута default можно использовать вложенное содержимое и получить тот же результат. Это позволяет использовать теги HTML, JavaScript и другие теги JSP для генерации значения по умолчанию.

```
<c:out value="${someVariable}">default value</c:out>
```

Обычно значение, указанное EL-выражением в атрибуте value, принудительно привязывается к String и эта String выводится. Если EL-выражение возвращает java.io.Reader, содержимое этого объекта считывается, а затем выводится.

Тег <c:url> правильно кодирует URL-адреса и перезаписывает их при необходимости, чтобы добавить идентификатор сессии, а также выводит URL-адреса в JSP. Тег отрабатывает вместе с тегом <c:param>, который задает параметры запроса для включения в URL. Если URL-адрес является относительным, тег добавляет URL-адрес с контекстным путем для приложения, чтобы браузер получил правильный абсолютный URL-адрес. Рассмотрим следующее использование тега:

```
<c:url value="http://www.example.net/content/books/
        world.html" />
```

Поскольку этот URL является абсолютным URL-адресом и не содержит пробелов или других специальных символов для шифрования, он не изменяется. Но, если есть параметры запроса, которые нужно включить в URL-адрес — тег <c:url> правильно сформирует и зашифрует строку запроса:

```
<c:url value=
    "http://www.example.net/content/books/world.jsp">
<c:param name="title" value="${titleId}" />
<c:param name="seo" value="${seoString}" />
</c:url>
```

Параметр title не вызовет проблемы, но параметр seo, так как это строка оптимизации поисковой системы, может содержать пробелы, вопросительные знаки, амперсанды и другие специальные символы, все из которых зашифрованы, чтобы обеспечить, что они не повреждают URL. Однако, тег <c:url> наиболее полезен при шифровании относительных URL-адресов.

Представьте, что приложение развернуто на *http://www.work.com/forums/*, и Вы поместите в HTML-код следующий тег ссылки:

```
<a href="/view.jsp?forumId=6">Job Vacancy Forum</a>
```

Когда пользователь перейдет по ссылке, он будет перенаправлен на *http://www.work.com/view.jsp?forumId=6*. Вероятно, Вы указали на этот URL-адрес относительно приложения форума, а не на весь сайт. Вы можете легко изменить свою ссылку, чтобы указывать на */forums/view.jsp?forumId=6*, но что, если Вы пишете приложение для

форумов, которое любой может загрузить и использовать на своем веб-сайте. Вы не знаете, будут ли они распространять приложение на /forums, /discussion, /boards или даже просто /. Здесь пригодится тег <c:url>.

```
<a href="<c:url value="/view.jsp">
<c:param name="forumId" value="6" />
</c:url>">Job Vacancy Forum</a>
```

Тег <c:url> здесь фактически встроен в атрибут HTML-тега. Тег <c:url> синтаксически анализируется и заменяется, когда JSP-движок отображает JSP и обрабатывает все, что не является специальным синтаксисом JSP, как обычный текст. Если приложение развернуто на */forums*, результирующая ссылка указывает на */forums/view.jsp?forumId=6*. Если оно будет развернуто на */boards*, ссылка будет */boards/view.jsp?forumId=6*. Вам не придётся беспокоиться о том, к какому контексту относится приложение. Если нужно, чтобы URL-адрес возвращался к корню сайта, или переходил к другому развертываемому приложению — добавьте атрибут context к тегу.

```
<c:url value="/index.html" context="/"/>
<c:url value="/item.jsp?userId=23"
       context="/organization" />
```

В первом теге создается URL-адрес к корневому контексту — */index.html*. Второй тег создает URL-адрес, идущий в контекст */organization* — */organization/item.jsp?userId=23*.

По умолчанию тег <c:url> выводит результирующий URL-адрес в ответ. Если URL-адрес необходимо использовать несколько раз на странице, можно сохранить полученный URL-адрес для скопированной переменной:

```
<c:url value="/index.jsp" var="mainUrl" />
<c:url value="/index.jsp" var="mainUrl" scope="request" />
```

Атрибут var указывает имя переменной EL для создания и сохранения результирующего URL-адреса. По умолчанию он сохраняется в области страницы, что обычно является достаточным. Если по какой-то причине нужно сохранить его в другой области, можно использовать атрибут scope, чтобы явно указать область. Значение var является простой строкой, а не EL-выражением. Хотя EL-выражение здесь может работать — оно бесполезно. Вы указываете JSP имя атрибута, который хотите создать в области, поэтому он всегда должен быть простым строковым значением.

Первый тег в предыдущем примере создает атрибут mainUrl в области страницы. Второй тег создает тот же атрибут, но в области запроса. Независимо от того, в какой области Вы сохраняете URL-адрес, можно ссылаться на URL-адрес позже на странице ${mainUrl}.

```
<a href="${mainUrl}">Main</a>
```

Для максимальной безопасности, гибкости и переносимости рекомендуется, чтобы все URL-адреса в JSP были зашифрованы с помощью <c:url>, если URL-адрес не является внешним URL-адресом без параметров запроса. Даже

в этом случае использование <c:url> по-прежнему является допустимым и даже рекомендуется, если URL содержит специальные символы, которым требуется шифрование.

Тег <c:if> является условным тегом для управления, отображением определенного контента. Синтаксис использования тега <c:if>:

```
<c:if test="${variable == otherVariable}">
    Run if true
</c:if>
```

Атрибут test содержит условие, результат которого должен быть true для вложенного содержимого в теге <c:if>. Если условие false, все внутри тега игнорируется. Если необходимо выполнить сложное условие только единожды, а результат использовать несколько раз на странице, можно сохранить его в переменной с помощью атрибута var:

```
<c:if test="${multiExpression}" var="isTrueExp" />
...
<c:if test="${isTrueExp}">
    Run if true
</c:if>
...
<c:if test="${isTrueExp}">
    Run if true
</c:if>
```

<c:if> предназначен для простых условных блоков, более сложная логика этим тегом не поддерживается.

Теги <c:choose>, <c:when> и <c:otherwise> являются более мощным эквивалентом тега <c:if> и обеспечивают

сложную логику if/else-if/else. Тег <c:choose> действует как блок, указывающий начало и конец сложного условного блока. Он не имеет атрибутов и может содержать только пробелы, <c:when> и <c:otherwise>, вложенные внутри. В теге <c:choose> должно быть вложено не менее одного тега <c:when>, и все они должны быть указаны перед тегом <c:otherwise>. Тег <c:when> имеет один атрибут test, который содержит условие. Любой контент или другие теги JSP могут быть вложены в <c:when>. Содержимое только одного тега <c:when> будет отрабатывать — первого, где test равен true. Блок <c:choose> закрывается после того, как тег <c:when> получил значение true.

Может быть не более одного (необязательного) тега <c:otherwise>, и он должен быть последним тегом внутри <c:choose>. Он не имеет атрибутов. Может содержать любой вложенный контент и отрабатывает только в том случае если ни один из тегов <c:when> не получил значение true.

```
<c:choose>
    <c:when test="${condition}">
            "if"
    </c:when>
    <c:when test="${otherCondition}">
            "else if"
    </c:when>
    ...
    <c:otherwise>
            "else"
    </c:otherwise>
</c:choose>
```

Из предыдущего кода видно, что первый <c:when> похож на if в Java. Если условие true — все внутри него выполняется, а все остальное игнорируется. Второй и все остальные <c:when> аналогичны else if. Они проверяются только в том случае, если каждый предыдущий — false. Тег <c:otherwise> похож на конечный else, который всегда выполняется, когда все остальное false.

Тег <c:forEach> используется для повторения его содержимого некоторое фиксированное количество раз или итерации по некоторой коллекции или массиву объектов. Он может действовать как циклы Java for или for-each, в зависимости от того, какие атрибуты будут использованы. Например, предположим, что Вы хотите заменить следующий цикл Java на <c:forEach>:

```
for(int i = 0; i < 21; i++)
{
    System.out.println("Num " + i + " ");
}
```

Эквивалентный тег <c: forEach> выглядит следующим образом:

```
<c:forEach var="i" begin="0" end="21">
    Num ${i}
</c:forEach>
```

В этом случае, каждое число от 0 до 21 отобразится на экране. Атрибут begin должен быть не менее 0. Если end меньше begin, цикл никогда не выполнится. Вы также можете инкрементировать i более чем на 1,

используя атрибут step (который должен быть больше или равен 1):

```
<c:forEach var="i" begin="0" end="21" step="2">
    Num ${i}
</c:forEach>
```

В этом случае каждое второе число от 0 до 21 отобразится на экране.

Для итерации по разным коллекциям объектов необходимо использовать разные атрибуты тега <c:forEach>:

```
<c:forEach items="${clients}" var="client">
    ${client.lastName}, ${client.firstName}<br />
</c:forEach>
```

Выражение внутри items должно вычисляться в некоторой Collection, Map, Iterator, Enumeration или массива объектов. Если items представляют собой Map, Map.Entry перебираются путем вызова метода entrySet. Если items являются Iterator или Enumeration — можно перебрать их только один раз. Если items равно null, никакая итерация не выполняется, как если бы коллекция была пуста. Исключение NullPointerException не произойдет. Если есть другой класс, который реализует Iterable, но не является одним из этих типов, можно использовать его, вызывая метод iterator для объекта (items="${object.iterator()}"). Атрибут var указывает имя переменной, которой каждый элемент должен быть назначен для каждой итерации цикла. Предыдущий пример эквивалентен следующему циклу Java for-each:

```
for(Client client:clients)
{
    System.out.println(client.getLastName() + " " +
           client.getFirstName() + "<br />");
}
```

Можно пропускать элементы коллекции в <c:forEach> с помощью атрибута step, как и для итерации по индексам. Также можно использовать атрибут begin для начала итерации по указанному индексу (включительно) и атрибуту end для завершения итерации по указанному индексу (включительно). Например, это полезно для реализации пейджинга коллекции объектов.

При использовании <c:forEach> как и в циклах for или foreach, можно использовать атрибут varStatus, чтобы объявить переменную доступную в цикле, которая содержит текущий статус итерации.

```
<c:forEach items="${clients}" var="client"
                              varStatus="info">
    ${info.begin}
    ${info.end}
    ${info.step}
    ${info.count}
    ${info.current}
    ${info.index}
    ${info.first}
    ${info.last}
</c:forEach>
```

В этом примере переменная info инкапсулирует статус текущей итерации. Свойства этого объекта состоя-

ния (экземпляр javax.servlet.jsp.jstl.core.LoopTagStatus) следующие:

1. **Begin** — значение атрибута begin из тега цикла.
2. **End** — значение атрибута end из тега цикла.
3. **Step** — значение атрибута step из тега цикла.
4. **Index** — возвращает текущий индекс из итерации. Это значение увеличивается на step для каждой итерации.
5. **Count** — возвращает количество выполненных до сих пор итераций (включая текущую итерацию). Это значение увеличивается на 1 для каждой итерации, даже если шаг больше 1. Значение начинается с 1 с первой итерацией и никогда не равно status.index.
6. **Current** — текущий элемент из итерации. В предыдущем примере status.current равен client.
7. **First** — true, если текущая итерация является самой первой итерацией. В противном случае — false.
8. **Last** — true, если текущая итерация является последней итерацией. В противном случае — false.

Последнее, что следует учитывать при использовании <c:forEach> — это влияние отложенного синтаксиса EL (#{}). Если Вы собираетесь использовать некоторый тег в цикле, который требует отложенного синтаксиса в атрибуте, и хотите использовать переменную, созданную как указано в var в этом отложенном синтаксисе, Вы также должны использовать отложенный синтаксис для EL-выражения в <c:forEach >. В противном случае отложенный синтаксис, ссылающийся на переменную элемента, не будет работать.

Тег <c:forTokens> почти идентичен тегу <c:forEach>. Он содержит много таких же атрибутов (var, varStatus, begin, end и step), которые ведут себя так же, как и в теге <c:forEach> при работе в цикле for-each по перебору объектов. Основное отличие от <c:forTokens> заключается в том, что атрибут items принимает String, а не коллекцию, а дополнительный атрибут delims указывает один или несколько символов, с помощью которых можно разделить String на токены. Затем тег перебирает эти токены.

```
<c:forTokens items="Every hunter wants to know, where
             the pheasant is." delims=" ,." var="token">
    ${token}<br />
</c:forTokens>
```

Вы уже знакомы с тегом <c:redirect>, который используется в файлах *index.jsp*. Этот тег перенаправляет пользователя на другой URL-адрес. После добавления заголовка HTTP-местоположения в ответ и изменения кода состояния ответа HTTP, он прерывает выполнение JSP. Поскольку он изменяет заголовки ответов, <c:redirect> должен быть вызван до того, как ответ начнет возвращаться обратно клиенту. В противном случае, перенаправить клиента не удастся, и вместо этого клиент получит усеченный ответ (со всем, что осталось после тега <c:redirect> в JSP). <c:redirect> следует тем же правилам, что и <c:url> относительно кодировки URL, переписывая идентификаторы сессии и добавляя параметры запроса с помощью вложенных тегов <c:param>. Следующие примеры — все возможные применения тега <c:redirect>:

```
<c:redirect url="http://www.work.org/" />
<c:redirect url="/cards">
<c:param name="action" value="add" />
<c:param name="cardId" value="${cardId}" />
</c:redirect>
```

Тег <c:import> позволяет извлекать содержимое ресурса по определенному URL-адресу. Затем содержимое может быть привязано к ответу, сохранено в переменной String или сохранено в переменной Reader. Как и в случае с <c:url> и <c:redirect>, URL-адрес может быть для локального контекста, для другого контекста или для внешнего сайта и надлежащим образом зашифрован и переписан, когда это необходимо. Вложенные теги <c:param> также могут указывать параметры запроса для шифрования в URL. Атрибут var указывает имя переменной String, в которой должен быть сохранен контент, и атрибут scope может указывать область действия переменной String. Атрибут varReader указывает имя переменной Reader, которая должна быть доступна для чтения содержимого.

Если используется varReader для экспорта переменной Reader, нельзя использовать <c:param>, и необходимо использовать Reader внутри вложенного содержимого тега <c:import>. Считывающая переменная не будет доступна после закрытия тега </ c:import>. Это гарантирует, что механизм JSP имеет возможность закрыть Reader.

Не следует использовать var и varReader вместе, это приведет к появлению исключительной ситуации. По-

скольку ни атрибуты var, ни атрибуты varReader не указаны, содержимое ресурса по URL-адресу встроено в JSP. Следующие примеры демонстрируют некоторые способы использования <c:import>.

```
<c:import url="/mainPage.jsp" />
<c:import url="/vac.jsp" context="/job"
          var="vacancy" scope="request">
<c:param name="specialization" value="${varSpec}" />
</c:import>
```

Первый пример выводит содержимое из локального *mainPag.jsp* приложения на странице. Второй сохраняет содержимое vac.jsp?specialization=${varSpec} в переменной с именем String в области запроса, правильно шифруя параметр запроса категории.

Вы можете использовать тег <c:set>, чтобы установить значение новой или существующей переменной области видимости и использовать ее экземпляр <c:remove>, чтобы удалить переменную из области видимости.

```
<c:set var="some" value="To be, or not to be" />
...
${some}
...
<c:remove var="some" scope="page" />
<c:set var="other" scope="request">
Here is other JSP tags
</c:set>
...
${other}
...
<c:remove var="other" scope="request" />
```

Как и в большинстве других тегов, которые предоставляют области видимости переменным, Вы можете использовать атрибут scope, чтобы указать в какой области видимости переменная определена. По умолчанию используется область страницы. Будьте осторожны с <c:remove>, потому что его атрибут области не всегда работает одинаково: если не указана область, все атрибуты с совпадающими именами во всех областях удаляются. Вероятно, это не то, что Вы хотите, поэтому нужно использовать атрибут scope. В дополнение к ранее показанным функциям также можно использовать <c:set> для изменения значения свойства в bean.

```
<c:set target="${obj}" property="propertyName"
    value="To be, or not to be" />
<c:set target="${obj}" property="propertyName">
Here is other JSP tags
</c:set>
```

В данном случае, при использовании атрибута target, он всегда должен быть EL-выражением, которое анализирует либо Map, либо какой-либо другой компонент с «сеттер» методами. Предыдущие примеры эквивалентны вызову obj.setPropertyName(...) для bean или obj.put ("propertyName", ...) для Map в Java-коде.

Пример использования библиотеки тегов Core

Чтобы понять, как работает библиотека тегов Core, рассмотрим пример новостной ленты. Создадим стандартное приложение с дескриптором развертывания, содержащим стандартные JSP. Файл index.jsp должен

перенаправляться на сервлет /list посредством тега <c:redirect>:

```
<c:redirect url="/list" />
```

Базовая страница JSP /WEB-INF/jsp/base.jsp должна содержать директивы taglib для библиотек Core и Function из JSTL:

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

Нам нужен список новостей, поэтому начнем с создания простого блока новостей POJO с базовой информацией. Обратите внимание, что в следующем примере кода, класс Story использует новый Java 8 Date and Time API и реализует Comparable, чтобы было можно его отсортировать соответствующим образом.

```
public class Story implements Comparable<Story>
{
    private String title;
    private String content;
    private String author;
    private String category;
    private MonthDay dateStory;
    private Instant dateCreatedStory;
    public Story()  {    }
    public Story(String title, String content,
          String author, String category,
          MonthDay dateStory, Instant dateCreatedStory)
    {
        this.title = title;
```

```
        this.content = content;
        this.author = author;
        this.category = category;
        this.dateStory = dateStory;
        this.dateCreatedStory = dateCreatedStory;
    }
    public String getTitle()
    {
        return title;
    }
    public void setTitle(String title)
    {
        this.title = title;
    }
    public String getContent()
    {
        return content;
    }
    public void setContent(String content)
    {
        this.content = content;
    }
    public String getAuthor()
    {
        return author;
    }
    public void setAuthor(String author)
    {
        this.author = author;
    }
    public String getCategory()
    {
        return category;
    }
    public void setCategory(String category)
    {
        this.category = category;
```

```
    }
    public MonthDay getDateStory() {
        return dateStory;
    }
    public void setDateStory(MonthDay dateStory) {
        this.dateStory = dateStory;
    }
    public Instant getDateCreatedStory()
    {
        return dateCreatedStory;
    }
    public void setDateCreatedStory(Instant
        dateCreatedStory)
    {
        this.dateCreatedStory = dateCreatedStory;
    }
    @Override
    public int compareTo(Story other)
    {
        return dateStory.compareTo(other.dateStory);
    }
}
```

StoryServlet отвечает на запросы и содержит статический Set новостей, который служит в качестве базы данных и предварительно заполняется несколькими статьями. Метод doGet добавляет пустые статьи к атрибуту запроса, если присутствует пустой параметр, или статический Set, если параметр отсутствует, а затем перенаправляется в файл */WEB-INF/jsp/view/list.jsp*.

```
@WebServlet(
        name = "storyServlet",
        urlPatterns = "/list"
)
```

```
public class StoryServlet extends HttpServlet
{
    private static final SortedSet<Story>
        STORIES = new TreeSet<Story>(){
        {
            add(new Story("News1", "Basketball",
                "Ivanov", "Sport",
                MonthDay.of(Month.JULY, 14),
                Instant.parse("2018-07-15T08:13:45Z")));
            add(new Story("News2", "Fish", null, "Food",
                MonthDay.of(Month.JULY, 13),
                Instant.parse("2018-07-16T08:17:45Z")));
            add(new Story("News3", "New president",
                "Petrov", "Politics",
                MonthDay.of(Month.JULY, 15),
                Instant.parse("2018-07-15T08:45:45Z")));
        }
    };

    @Override
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
    {
        if(request.getParameter("empty") != null)
            request.setAttribute("stories",
               Collections.<Story>emptySet());
        else
            request.setAttribute("stories", STORIES);
               request.getRequestDispatcher(
               "/WEB-INF/jsp/view/list.jsp").
               forward(request, response);
    }
}
```

Обратите внимание, что в list.jsp используется <c:choose>, <c:when> и <c:otherwise>, чтобы отобразить

сообщение, если новостная лента пуста, и в противном случае выполняется код для перебора списка. <c:forEach> выполняет эту задачу, а <c:out> обеспечивает правильное экранирование значений String, чтобы они не содержали XML символы. Наконец, тег <c:if> гарантирует, что имя автора отображается только в том случае, если оно не равно null.

```
<%--@elvariable id="stories"
     type="java.util.Set<com.acadamy.Story>"--%>
<!DOCTYPE html>
<html>
    <head>
        <title>News</title>
    </head>
    <body>
        <h1>Latest News</h1>
        <c:choose>
           <c:when test="${fn:length(stories) == 0}">
               <b>There are no news today.</b>
           </c:when>
           <c:otherwise>
               <c:forEach items="${stories}" var="story">
                    <b>
               <c:out value="Title: ${story.title}" />
               <br />
               <c:out value=
                      "Category: ${story.category}" />
                      <br />
               <c:out value="Date: ${story.
                      dateStory}" /><br />
                       </b>
               <c:out value="${story.content}" /><br />
               <c:if test="${story.author != null}">
               Author: ${story.author}<br />
               </c:if>
```

```
                Date created: ${story.dateCreatedStory}
                    <br /><br />
                </c:forEach>
            </c:otherwise>
        </c:choose>
    </body>
</html>
```

Скомпилируйте, запустите проект и перейдите по ссылке в *http://localhost:8080/news/list?empty* в браузере. Вы должны увидеть сообщение о том, что в новостной ленте нет контактов, что означает, что условие <c:when> — true. Теперь уберите с URL-адреса параметр empty, и Вы увидите такой экран, как на рисунке.

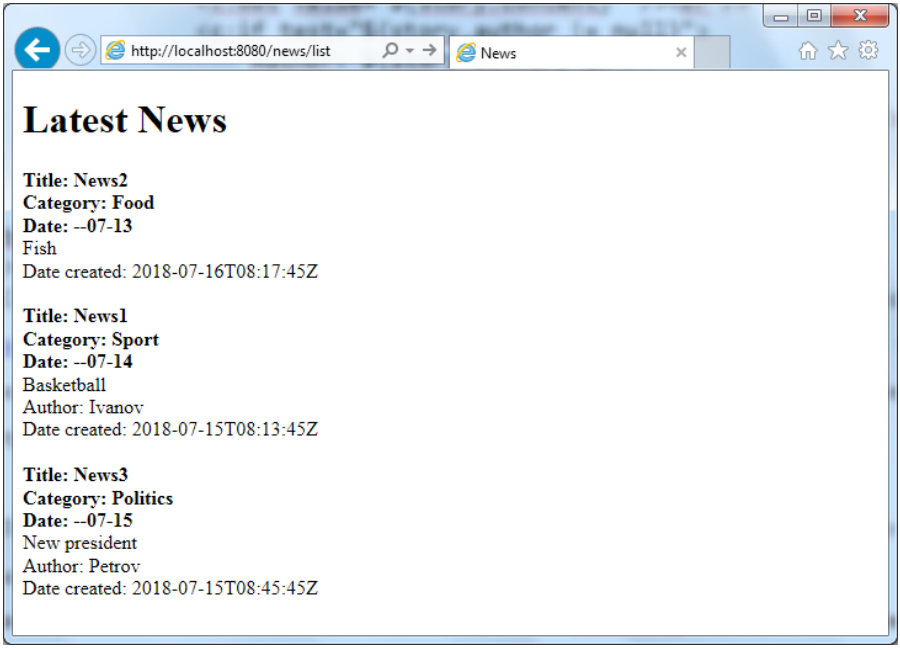

Рисунок 1.

По большей части это выглядит нормально, но даты не отформатированы, и нет поддержки альтернативных языков.

Formatting Tags

Если Вы планируете создавать корпоративные Java-приложения для развертывания в Интернете и привлекать значительную международную аудиторию, Вам будет необходимо локализовать приложение в определенных регионах мира. Это достигается за счет интернационализации (сокращенно *i18n*), что является процессом разработки приложения, чтобы он мог адаптироваться к различным регионам, языкам и культурам без перепроектирования или перезаписи приложения для новых регионов.

После того, как приложение интернационализировано, Вы можете использовать инфраструктуру интернационализации для локализации приложения, добавив поддержку целевых регионов, языков и культур. Часто термины локализация (сокращенно *L10n*) и интернационализация путаются, так как они тесно взаимосвязаны. Один удобный способ запомнить разницу: «Сначала Вы интернационализируетесь через архитектуру, а затем локализуетесь посредством перевода».

Существует три принципа интернационализации и локализации:

1. Текст должен быть переводимым и переведенным, чтобы пользователи, говорящие на других языках, могли использовать приложение.
2. Даты, время и цифры (включая валюты и проценты) должны быть отформатированы правильно для раз-

ных мест. Например, 123,456,789.00 в Англии на самом деле составляет 123 456 789,00 в Украине.

3. Цены должны быть конвертируемыми, чтобы они могли отображаться в местной валюте в соответствии с регионами пользователя.

Зачастую конвертирование валют опущена — и не зря. Отображение цен в разных валютах для пользователей может быть чрезвычайно сложным, неточным и устаревшим, и сегодня большинство бизнес-финансовых учреждений предоставляют механизмы конвертации валюты в валюту пользователя во время оформления заказа. Во многих случаях вполне достаточно просто отображать цены в долларах США.

JSTL предоставляет библиотеку тегов, которая поддерживает как интернационализацию, так и локализацию в приложениях: библиотеку интернационализации и форматирования, префиксом которой является fmt. Мы разберем, как использовать и настроить библиотеку форматирования для интернационализации приложения, чтобы впоследствии было можно локализовать его. Вы часто видите термин locale, который является идентификатором, представляющим определенный регион или культуру. Локали всегда содержат двухбуквенный код нижнего регистра, как указано в ISO 639-1.

Для языков, где код языка неоднозначен (например, американский английский язык немного отличается от британского английского), языковой стандарт может содержать двухбуквенный код страны с верхним регистром, как указано в ISO 3166-1.

В локальном коде, код языка всегда на первом месте. Если указывается код страны, это делается следующим образом, с знаком подчеркивания, разделяющим язык и коды стран. Иногда языковой стандарт может отличаться от того, который входит в состав кода локали. Дополнительная информация о них содержится в документации API для класса java.util.Locale. Locale используется для представления локалей в Java и JSTL.

Библиотека тегов интернационализации и форматирования разделена на две основные категории:

- теги, которые поддерживают интернационализацию (теги i18n);
- теги, которые поддерживают форматирование даты, времени и чисел (formatting tags).

Теги i18n являются пакетами ресурсов, которые определяют локальные объекты. Пакеты ресурсов состоят из ключей, которые соответствуют записям в пакете и определяются с использованием произвольного базового имени выбора разработчика с кодом локали, добавленным к базовому имени, чтобы сформировать полное имя пакета ресурсов. У данного ключа обычно есть записи в каждом наборе ресурсов, по одному для каждого языка и страны.

Ниже приведена директива taglib для библиотеки интернационализации и форматирования:

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

<fmt:message> — это тег i18n, который используется чаще всего. <fmt:message> позволяет использовать локали-

зованное сообщение в наборе ресурсов, а затем либо размещает это сообщение на странице, либо сохраняет значение в переменной EL. Требуемый атрибут key определяет ключ сообщения локализации для размещения в наборе ресурсов. Используется дополнительный атрибут bundle, чтобы указать, какой контекст локализации, созданный с помощью <fmt:setBundle>, должен использоваться для поиска ключа. Это переопределяет пакет по умолчанию. Необязательный атрибут var указывает переменную EL для сохранения локализованного сообщения, и соответствующий атрибут scope может управлять той областью, в которую входит переменная. Сообщения локализации также могут быть параметризованы с помощью тега <fmt:param>, вложенного в <fmt:message>. Предположим, у приложения был один пакет ресурсов, переведенный на два языка — британский английский (en_GB) и русский (ru_RU) — каждый перевод содержал запись для хранилища сообщений. Английское значение для film может выглядеть так:

```
film={0} days of Summer.
```

Русское значение похоже, за исключением того, что оно переведено:

```
film={0} дней лета.
```

Токен {0} в каждом сообщении указывает заполнитель, который должен заменить какой-то параметр. Заполнители основаны на нулевом значении, но они не должны отображаться в числовом порядке в сообщении. Некоторые языки используют очень разные порядки слов, что означает, что параметр может потребоваться заменить в разных местах на

разных языках. Заполнители также могут быть дублированы по желанию — возможно, Вы захотите вставить один и тот же параметр в сообщение несколько раз. В JSP Вы ссылаетесь на сообщение локализации, используя следующий код.

```
<fmt:message key="film">
    <fmt:param value="${numberOfDays}" />
</fmt:message>
```

Если numberOfDays было бы 500, пользователи увидели бы следующий результат в зависимости от выбранного региона:

Английский: *500 days of Summer.*

Русский: *500 дней лета.*

Вложенные теги <fmt:param> соответствуют заполнителям в порядке от 0. Таким образом, первый тег <fmt:param> указывает значение для {0}, второе для {1} и т. д., независимо от того, какой порядок заполнители фактически появляются в локализованном сообщении. Сообщения не обязательно должны содержать заполнители, и Вы можете использовать <fmt:message> без <fmt:param>. Если в сообщении есть заполнитель и нет тега <fmt:param>, то заполнитель остается как и есть в сообщении. Вместо атрибута value Вы можете указать значение параметра, поместив его в тег <fmt:param>.

```
<fmt:message key="film">
    <fmt:param>${numberOfDays}</fmt:param>
</fmt:message>
```

Тег <fmt:setLocale> устанавливает языковой стандарт, используемый для разрешения сообщений пакета ресур-

сов для i18n и форматирования. Атрибут value указывает локаль и может быть либо строковым языковым кодом (например, en_GB), либо EL-выражением, вычисляющим экземпляр Locale. Если value является код локали, атрибут variant также может быть указан для обозначения варианта локали. Локаль сохраняется в переменной EL с именем javax.servlet.jsp.jstl.fmt.locale и становится локалью по умолчанию в указанной scope. Если используется этот тег, он должен появиться перед любыми другими i18n или тегами форматирования, чтобы гарантировать, что используется правильный язык. Однако обычно Вам не нужен тег <fmt:setLocale>. В интернационализированных приложениях обычно есть механизм (например, загрузка сохраненных настроек языкового стандарта из учетной записи пользователя), с помощью которого языковой стандарт автоматически устанавливается до того, как запрос будет отправлен в JSP.

```
<fmt:setLocale value="en_GB" />
<fmt:setLocale value="${locale}" />
```

Теги i18n в JSTL полагаются на контекст локализации, чтобы получить текущий набор ресурсов и локали. Вы можете использовать <fmt:setLocale> и другие методы, чтобы указать локаль в текущем контексте локализации. <fmt:bundle> и <fmt:setBundle> — два способа указать ресурсный пакет, который должен использоваться. Когда тегу i18n необходимо узнать свой контекст локализации, он размещается в нескольких местах, используя первый указанный контекст локализации, который он находит в этом порядке приоритета:

- Если указан атрибут связки тега <fmt:message>, он использует это значение с предпочтением над всеми другими пакетами, которые могут применяться к тегу.
- Если тег i18n вложен в тег <fmt:bundle>, он использует этот пакет, если он не переопределен атрибутом bundle в <fmt:message>.
- Если контекст локализации по умолчанию указан с использованием проинициализированного параметра контекста или переменной EL с именем javax.servlet.jsp.jstl.fmt.localizationContext, он использует этот пакет.

Хотя тег <fmt:bundle> создает контекст одноранговой локализации, который влияет только на вложенные теги внутри, <fmt:setBundle> экспортирует контекст локализации в переменную EL, которую теги i18n могут впоследствии использовать, ссылаясь на переменную bundle в атрибуте bundle. Имя экспортируемой переменной указано в атрибуте var и имеет область, указанную в scope. Если Вы не укажете var, то контекст локализации сохраняется в переменной EL с именем javax .servlet.jsp.jstl.fmt.localizationContext и становится контекстом локализации по умолчанию для этой области. Следующий пример демонстрирует приоритет определения bundle.

```
<fmt:setBundle basename="News" var="newsBundle" />
<fmt:bundle basename="Categories">
<fmt:message key="categories.mainPage" />
<fmt:message key="news.notFound" bundle="${newsBundle}" />
</fmt:bundle>
<fmt:message key="someNews " />
```

Атрибут basename указывает базовое имя пакета ресурсов — то есть начало файла пакета ресурсов до того, как к нему добавится код локали. Тег <fmt:message>, выводящий сообщение categories.mainPage, будет использовать пакет Categories, определенный тегом <fmt:bundle>, внутри которого он вложен, тогда как сообщение news.notFound будет разрешено с помощью пакета News, определенного в <fmt:setBundle>. Наконец, сообщение someNews будет разрешено с использованием контекста локализации по умолчанию.

Тег <fmt:requestEncoding> устанавливает кодировку символов для текущего запроса с использованием атрибута var, чтобы параметры запроса были правильно расшифрованы с кодировкой символов, соответствующей данному языку.

Тег не используется по двум причинам:

- Java-сервлеты обрабатывают параметры запроса до того, как тег <fmt:requestEncoding> сможет изменить кодировку символов запроса.
- Все современные браузеры включают заголовок запроса Content-Type с кодировкой символов для любых запросов, кодировка которых отличается от ISO-8859-1. Все они также используют кодировку символов, возвращенную последним ответом от сайта, на который отправлен запрос. Это устраняет необходимость ручной установки кодировки символов запроса.

Этот тег является устаревшим тегом, который исходит из эпохи, когда кодировка атрибутов запроса редко известна и должна быть угадана. Сегодня, из-за по-

ведения современных браузеров, необходимость сделать это ушла.

<fmt:timeZone> и <fmt:setTimeZone> — теги форматирования, которые обрабатывают дату и время, требуют правильного функционирования языкового стандарта и часового пояса. Локаль происходит из контекста локализации, о котором мы говорили в <fmt:bundle> и <fmt:setBundle>. Однако идея часового пояса, иногда коррелированная с регионом или языком, указанным в локали, строго не привязана к языку. Например, кто-то из Украины может посетить Англию, и Вы хотите использовать свое приложение на украинском языке со стандартным форматом даты и времени Украины, видя даты и время в Англии. По этой причине и многие другие часовые пояса и локали разделены на Java и JSTL. Форматирование тегов, требующих часовых поясов, позволяет использовать часовой пояс с данной стратегией в порядке приоритета:

- Если тег форматирования даты и времени имеет значение, указанное для атрибута timeZone, используйте это значение с предпочтением по всем другим часовым поясам.
- Если тег вложен в тег <fmt:timeZone>, используйте часовой пояс, указанный <fmt:timeZone>.
- Если проинициализирован параметр контекста или указана EL-переменная javax.servlet.jsp.jstl.fmt.timeZone, используйте этот часовой пояс.
- В противном случае используйте часовой пояс, указанный контейнером, обычно это часовой пояс JVM, часовой пояс базовой операционной системы.

Тег <fmt:timeZone> — это часовой пояс, поведение которого аналогично <fmt:bundle>. Он создает специальную область часового пояса, в которой любые вложенные теги используют данный часовой пояс. Его единственным атрибутом является значение, которое может быть EL-выражением, вычисляющим java.util.TimeZone или String, соответствующей любому допустимому идентификатору часового пояса, указанному в базе данных часовых поясов IANA. Можно узнать об этих идентификаторах в документации API для TimeZone. Если значение равно null или пустое, предполагается часовой пояс GMT.

Тег <fmt:setTimeZone> аналогичен тегу <fmt: setBundle> и экспортирует указанное value часового пояса в переменную области видимости. Атрибут var указывает имя переменной EL для экспорта часового пояса в область, заданную scope. Если var опущен, часовой пояс сохраняется в переменной EL с именем javax.servlet.jsp.jstl.fmt.timeZone и становится часовым поясом по умолчанию для этой области.

```
<fmt:setTimeZone value="Ukraine/Kyiv"
    var="currentTimeZone" />

<fmt:timeZone value="${otherTimeZone}">
other time
</fmt:timeZone>

<fmt:timeZone value="${currentTimeZone}">
Kyiv time
</fmt:timeZone>
```

Тег <fmt:formatDate> форматирует указанную дату (и/или время) с использованием стандартного или ука-

занного языка и значения по умолчанию или указанного часового пояса. Отформатированная дата затем либо встраивается, либо сохраняется в переменной, указанной в атрибуте var с указанной scope. Атрибут timeZone указывает другой идентификатор TimeZone или String ID временной зоны для форматирования даты. Значение даты указано с использованием атрибута value. В настоящее время значение должно быть EL-выражением, вычисляющим экземпляр java.util.Date. Классы java.util.Calendar и Java 8 Date and Time API не поддерживаются.

Форматирование даты определяется с использованием языка в сочетании с атрибутами type, dateStyle, timeStyle и pattern. type должен быть одним из «дата», «время» или «дата/время», чтобы указать, что следует выводить. Атрибуты dateStyle и timeStyle соответствуют семантике, определенной в документации API для java.text.DateFormat, и должны быть отформатированы. Эти атрибуты определяют, как дата и время, соответственно, отформатированы относительно их локалей. Вы также можете указать собственный шаблон форматирования в соответствии с правилами java.text.SimpleDateFormat, используя атрибут pattern. В этом случае атрибуты type, dateStyle и timeStyle игнорируются. Выполнение этого также игнорирует стили, которые входят в локаль, поэтому лучше избегать использования шаблона, если это вообще возможно.

```
<fmt:formatDate value="${birthDate}"
    type="both" dateStyle="long"
    timeStyle="long" />
```

```
<fmt:formatDate value="${birthDate}"
     type="date" dateStyle="short"
     var="formattedDate" timeZone="${someTimeZone}"
/>
```

Для даты 15 августа 2018 года в 21:16:45 в часовом поясе Украина/Киев, первый пример выведет «15 серпня 2018 21:16:45 CDT» для украинского языка в Украине и «August 15, 2018 9:16:45 PM CDT» для английского языка в США. Второй пример форматирует только дату в более коротком формате в часовом поясе, указанном ${someTimeZone}, и сохраняет его в переменной formattedDate. Значение formattedDate — «15/08/18» для украинского языка в Украине и «8/15/18» для английского языка в США.

<fmt:parseDate> имеет все те же атрибуты и правила, что и <fmt:formatDate>, но он совершенно противоположен <fmt:formatDate>. Он принимает отформатированные Strings, как то, что <fmt:formatDate> будет выводить и анализирует их в объектах Date. Как правило, Вы всегда присваиваете это переменной с помощью атрибута var. Кроме того, вместо указания даты для синтаксического анализа, используется атрибут value, Вы можете указать его как содержимое тела в теге.

Тег <fmt:formatNumber> позволяет форматировать числа (целочисленные и десятичные), валюты и проценты. Он имеет много атрибутов, и не все они применимы ко всем ситуациям. Во-первых, этот тег, как и многие другие, имеет атрибуты var и scope. Если var опущен, форматированное число встроено в JSP. Теперь рассмо-

трим необходимость форматирования валюты и предположим, что число — это переменная области со значением 123456.7831.

```
<fmt:formatNumber type="currency" value="${num}" />
```

Это выводит «123 456,78₴» для украинского языка в Украине и «$123,456.78» для английского языка в США. Вы должны сразу увидеть проблему: число было представлено двумя разными символами валюты без конвертации валюты. Это довольно запутано и не точно. Из-за этого Вы всегда должны указывать атрибут currencyCode, который может быть любым действительным кодом валюты ISO 4217.

```
<fmt:formatNumber type="currency" value="${num}"
     currencyCode="UAH" />
```

Результат этого по-прежнему «123 456,78₴» для украинского языка в Украине, но теперь «123,456.78 UAH» для английского языка в США, который является правильным. Атрибут currencySymbol также может использоваться для переопределения используемого символа валюты. Тег может правильно определить символ валюты на основе указанного кода валюты. Оба эти атрибута игнорируются, если тип не является currency.

Другим допустимым значением type является number (по умолчанию). Он форматирует число как обычное число. По умолчанию округляет цифры до трех цифр и группирует цифры в числе в соответствии с локалью.

```
<fmt:formatNumber type="number" value="${num}" />
```

Это выводит «123 456.783» для украинского языка в Украине и «123,456.783» для английского языка в США. Используйте атрибут maxFractionDigits, чтобы увеличить и уменьшить точность округления, указав количество цифр после десятичного разделителя и maxIntegerDigits, чтобы указать максимальное количество цифр перед десятичным разделителем. Не рекомендуется использовать maxIntegerDigits, потому что он может урезать ваши числа. Например, если установлено значение 3, число 12345 становится равным 345. minFractionDigits используется для заполнения нулей в конце десятичной части числа до указанного количество цифр, а также minIntegerDigits используется для ввода нулей в начало целой части числа. Атрибут groupingUsed (по умолчанию true) указывает, следует ли группировать цифры в отформатированном числе. Если false, вывод из предыдущего примера был бы «123456.783» и «123456.783».

Третье и последнее значение type — percent и используется для форматирования числа в процентах.

```
<fmt:formatNumber type="percent" value="0.4863" />
```

Результат этого составляет «49%» для украинского языка в Украине и для английского языка в США, потому что стратегия по умолчанию — округлять проценты до целых чисел. Если для атрибута maxFractionDigits установлено значение 2, значения будут «48,63%» и «48.63%». Обратите внимание, что число автоматически умножается на 100, так что оно преобразуется в процент. Значение по умолчанию для maxFractionDigits зависит от локали для валют, 3 для чисел и 0 для процентов. Существует также

атрибут pattern, с помощью которого можно указать собственный шаблон для форматирования числа в соответствии с правилами java.text.DecimalFormat.

Так же как и <fmt:parseDate>, <fmt:parseNumber> противоположен <fmt:formatNumber>. Он не имеет атрибутов maxFractionDigits, maxIntegerDigits, minFractionDigits, minIntegerDigits и groupingUsed, потому что эта информация не нужна для разбора чисел. Он содержит дополнительные атрибуты integerOnly и parseLocale. IntegerOnly указывает, следует ли игнорировать часть долей числа, по умолчанию — false. ParseLocale указывает язык, который будет использоваться при разборе числа, если он отличается от локали по умолчанию. Также как в <fmt:parseDate> можно указать число для разбора в атрибуте value или в теге.

Пример использования i18n и библиотеки тегов форматирования

Доработаем и интернационализируем проект новостной ленты. Первое, что Вам необходимо сделать, это добавить новый проинициализированный параметр контекста в дескриптор развертывания.

```
<context-param>
    <param-name>javax.servlet.jsp.jstl.fmt.
        localizationContext</param-name>
    <param-value>NewsList-stories</param-value>
</context-param>
```

Он устанавливает ресурс, из которого могут быть загружены локализованные сообщения. Контейнер

ищет файл в любом месте пути вашего класса с именем *NewsList-stories_[language]_[region].properties*. Если он не находит этот файл, он ищет *NewsList-stories_[language].properties*. Если он этого не обнаружит, он переключается на резервную локаль и ищет свой пакет. Чтобы удовлетворить эту потребность, создайте файл с именем *NewsList-stories_ru_RU.properties* в каталоге источника *source/production/resources* проекта. Все файлы в каталоге resources будут скопированы в каталог */WEB-INF/classes* во время сборки.

```
title.browser=Новости
title.page=Свежие новости
message.noNews=На сегодня нет новостей.
label.title=Заголовок
label.category=Категория
label.dateStory=Дата проишествия
label.author=Автор
label.dateCreatedStory=Дата новости
```

Вам также нужна переведенная версия этого файла, чтобы Вы могли тестировать языки переключения, поэтому создайте файл с именем *NewsList-stories _en_US.properties* в том же каталоге.

```
title.browser=News
title.page=Latest News
message.noNews=There are no news today.
label.title=Title
label.category=Category
label.dateStory=Date
label.author=Author
label.dateCreatedStory=Date created
```

Для того чтобы менять язык, на котором отображается страница, добавьте следующий код в начало метода doGet в StoryServlet. Класс Config здесь импортируется из *javax.servlet.jsp.jstl.core.Config*.

```
String language = request.getParameter("language");
if("english".equalsIgnoreCase(language)) {
    Config.set(request, Config.FMT_LOCALE, "en_US");
}
```

Поскольку тег <fmt:formatDate> не поддерживает API Date and Time Java 8, Вам нужен способ доступа к объекту Date старого стиля, поэтому добавьте следующий метод в класс новости POJO.

```
public Date getOldDateCreated()
{
    return new Date(this.dateCreatedStory.
           toEpochMilli());
}
```

Добавьте директиву taglib, в файл */WEB-INF/jsp/base.jspf*.

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

Теперь, разберем новый *list.jsp*. Все литералы заменяются тегами <fmt:message>, ссылающимися на ключи из файлов properties. Процесс размещения тегов <fmt:message> в JSP — это интернационализация приложения. Процесс создания файлов свойств, содержащих переводы — это локализация приложения.

```
<%--@elvariable id="stories"
    type="java.util.Set<com.acadamy.Story>"--%>
<!DOCTYPE html>
<html>
   <head>
      <title><fmt:message key="title.browser" />
      </title>
   </head>

   <body>
      <h1><fmt:message key="title.page" /></h1>
      <c:choose>
         <c:when test="${fn:length(stories) == 0}">
             <b><fmt:message key="message.noNews" />
             </b>
         </c:when>
         <c:otherwise>
            <c:forEach items="${stories}"
                       var="story">
             <b>
                <fmt:message key="label.title" />:
                <c:out value="${story.title}" /><br />
                <fmt:message key="label.category" />:
                <c:out value="${story.category}" /><br/>
                <fmt:message key="label.dateStory" />:
                     ${story.dateStory.month.
                     getDisplayName('FULL',
                     pageContext.response.locale
                     )} ${story.dateStory.
                     dayOfMonth}<br />
              </b>
                <c:out value="${story.content}" /><br/>
                <c:if test="${story.author != null}">
                <fmt:message key="label.author"/>:
                          ${story.author}<br />
                </c:if>
```

```
                <fmt:message
                    key="label.dateCreatedStory" />:
                <fmt:formatDate
                    value="${story.oldDateCreated}"
                    type="both" dateStyle="long"
                    timeStyle="long" />
                <br /><br />
            </c:forEach>
        </c:otherwise>
    </c:choose>
  </body>
</html>
```

Также обратите внимание на использование <fmt:formatDate>, который форматирует dateCreatedStory для отображения на странице и на то, как изменился код, отображающий поле dateStory. Теперь протестируйте все это, скомпилируйте и запустите проект и перейдите по ссылке *http://localhost:8080/news/list* в браузере. Он выглядит также, за исключением того, что поле dateStory и dateCreatedStory отображаются в отформатированном виде и страница переведена на русский язык. Добавьте ?language=english к URL-адресу, и страница должна теперь отображаться на английском, а не на русском. Добавьте &empty к URL-адресу, и Вы увидите по-английски сообщение о том, что новостей нет. Вы успешно интернационализировали и локализовали свое приложение.

SQL Tags

JSTL содержит библиотеку тегов, которая обеспечивает транзакционный доступ к реляционным базам данных. Стандартный префикс для этой библиотеки —

sql, а его директива taglib аналогична предыдущим директивам:

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

Вообще говоря, выполнение действий с базами данных на уровне представления неодобрительно и его по возможности следует избегать. Вместо этого такой код должен идти в бизнес-логике приложения, как правило, в Servlet или, более подходящим образом, в репозитории, который использует Servlet. Однако эта библиотека тегов иногда полезна, особенно для прототипирования новых приложений или быстрого тестирования теорий или концепций.

Действия в библиотеке SQL предоставляют возможность запроса данных с помощью операторов SELECT, доступ и повторение результатов этих запросов, обновление данных с помощью инструкций INSERT, UPDATE и DELETE и выполнение любого количества этих действий в рамках транзакции. Как правило, теги в этой библиотеке работают с помощью javax.sql.DataSource. В тегах <sql:query>, <sql:update>, <sql:transaction> и <sql:setDataSource> есть атрибуты dataSource для указания источника данных, который должен использоваться для выполнения этого действия.

Атрибут dataSource должен быть либо DataSource, либо String. Если это DataSource, он используется как есть. Если это String, контейнер воспринимает String как JNDI-имя для DataSource. Если соответствующий Data-

Source не найден, контейнер выполняет последнее действие, принимая String как URL-адрес соединения JDBC и пытается подключиться с помощью java.sql.DriverManager. Если ни одно из них не работает, генерируется исключение. Для всех тегов атрибут dataSource является необязательным, и в этом случае контейнер ищет EL переменную с именем javax.servlet.jsp.jstl.sql.dataSource в области по умолчанию. Если это String или DataSource, применяется описанная ранее логика. В противном случае генерируется исключение.

Запросы выполняются с использованием тега <sql:query> и действия обновления с помощью тега <sql:update>. Для обоих тегов оператор SQL может быть указан в атрибуте sql или в содержимом вложенного тела. Вложенные теги могут использоваться для указания параметров подготовленных операторов. Вы можете создать транзакцию с помощью <sql:transaction>, и все вложенные теги запроса и обновления будут использовать эту транзакцию, но Вы должны помнить о двух правилах:

- Только тег <sql:transaction> может указывать атрибут dataSource (не может содержать вложенные теги).
- Если Вы запрашиваете данные, Вы должны также перебирать эти данные в транзакции.

XML Tags

Подобно библиотеке тегов SQL, библиотека тегов обработки XML не рекомендуется для использования. Ранее XML был единственным распространенным стандартом, с которым приложения обменивались данны-

ми, а способность анализировать и перемещать XML имела решающее значение. Сегодня все больше приложений поддерживают стандарт JSON в качестве альтернативы XML, а несколько высокоэффективных библиотек могут сопоставлять объекты с JSON или XML и обратно к объектам. Эти инструменты проще в использовании, чем библиотека тегов XML, и могут позаботиться о преобразовании данных там, где они содержатся — в бизнес-логике.

Библиотека тегов XML, префикс которой x, основан на стандарте XPath и состоит из узлов или множества узлов, переменных, функций и префиксов пространства имен. Он содержит много действий, аналогичных тегам в библиотеке тегов Core, но специально разработанных для работы с выражениями XPath в отношении XML-документа. Директива taglib библиотеки XML выглядит следующим образом.

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

Примеры использования JSTL и различных видов Tags

Ранее Вы заменили несколько строк кода Java на EL-выражения в JSP для приложения поддержки клиентов. Сейчас Вы замените почти весь Java-код в JSP на теги JSTL. Начните с просмотра страницы /WEB-INF/jsp/view/login.jsp. В первую строку страницы над doctype добавлена аннотация @elvariable, и единственный скрипт на этой странице был заменен тегом <c:if>.

```
<%--@elvariable id="loginFailed"
    type="java.lang.Boolean"--%>
<!DOCTYPE html>
<html>
…
Log in, please.<br />
<c:if test="${loginFailed}">
<b>The username or password incorrect.</b><br />
</c:if>
…
</html>
```

В файле /WEB-INF/jsp/view/viewCard.jsp внесено больше изменений. Сценарии и выражения, которые содержали список ссылок на вложения, заменены тегом <c:if>, который проверяет наличие вложений и <c:forEach> который перебирает их. Следующий тег <c:if> определяет, следует ли выводить запятую перед текущим вложением, используя переменную состояния тега цикла. Кроме того, большинство EL-выражений были помещены в атрибут value тегов <c:out>. Это защищает приложение от инъекций HTML и JavaScript. Теги <c:out> избегают любых зарезервированных символов XML, до тех пор, пока атрибут escapeXml равен true. Правилом хорошего тона является использовать <c:out> для вывода переменных String. Однако переменные, которые не являются char примитивами (целые числа, десятичные знаки), не требуют экранирования.

```
<%--@elvariable id="cardId"
    type="java.lang.String"--%>
<%--@elvariable id="card" type="com.academy.Card"--%>
```

```
<!DOCTYPE html>
<html>
    <head>
        <title>Support service</title>
    </head>

    <body>
        <a href="<c:url value="/login?logout" />">
            Logout</a>
        <h1>Card ${cardId}: <c:out value="${card.topic}"/>
        </h1>
        <i>Client Name -
            <c:out value="${card.clientName}" />
        </i><br />
        <c:out value="${card.message}" /><br />
        <c:if test="${card.numberOfAttachments > 0}">
            Attachments:
            <c:forEach items="${card.attachments}"
                       var="attachment"
                       varStatus="status">
                <c:if test="${!status.first}">, </c:if>
                <a href="<c:url value="/cards">
                    <c:param name="action"
                             value="download" />
                    <c:param name="cardId"
                             value="${cardId}" />
                    <c:param name="attachment"
                             value="${attachment.
                             fileName}" />
                 </c:url>"><c:out value="${attachment.
                             fileName}"/></a>
            </c:forEach><br />
        </c:if>
        <a href="<c:url value="/cards" />">
        Back to other cards</a>
    </body>
</html>
```

Файл */WEB-INF/jsp/view/listCards.jsp* использует многие из тех же функций для замены Java-кода как и предыдущий, но также использует более сложные <c:choose>, <c:when> и <c:otherwise> для замены кода if-else. <c:when> проверяет, пуста ли база данных обращений и выводит сообщение об этом, если это так. В противном случае цикл <c:forEach> выполняет итерацию по базе данных. Обратите внимание, что атрибут items <c:forEach> использует Map, поэтому каждая итерация предоставляет переменную Map.Entry<Integer, Card>. Можно получить доступ к ключу int с помощью entry.key и значение Card — с entry.value.

```
<%--@elvariable id="cardBase"
    type="java.util.Map<Integer,
    com.acadamy.Card>"--%>
<!DOCTYPE html>
<html>
    <head>
        <title>Support service</title>
    </head>
    <body>
        <a href="<c:url
           value="/login?logout" />">Logout</a>
        <h1>Cards</h1>
        <a href="<c:url value="/cards">
           <c:param name="action" value="create" />
           </c:url>">Add card</a><br/>
        <c:choose>
           <c:when test="${fn:length(cardBase) == 0}">
              <i>Base is empty.</i>
        </c:when>
        <c:otherwise>
            <c:forEach items="${cardBase}"
                       var="entry">
```

```
                Card # ${entry.key}: <a href="<c:url
                            value="/cards">
                        <c:param name="action"
                            value="view" />
                        <c:param name="cardId"
                            value="${entry.key}" />
                            </c:url>"><c:out
                            value="${entry.value.
                            topic}" /></a>
                Client Name: <c:out
                            value="${entry.value.
                            clientName}" />)<br />
            </c:forEach>
        </c:otherwise>
    </c:choose>
  </body>
</html>
```

Теперь скомпилируйте, запустите приложение и перейдите по ссылке *http://localhost:8080/service/*. Создайте несколько обращений и просмотрите их список. Добавьте теги HTML, цитаты и апострофы в названиях и в тексте обращений. В предыдущей версии проекта они были бы напечатаны буквально и интерпретировались как HTML в браузере. Вы можете просмотреть источник страницы, и проследить, что они теперь экранированы должным образом и больше не представляют опасности для вашего приложения.

17. Custom Tags. Tag Files

Понятие TLD, Tag Files и обработчиков Tag

Все теги JSP приводят к выполнению какого-либо обработчика тэгов. Обработчик тега представляет собой реализацию javax.servlet.jsp.tagext.Tag или javax.servlet.jsp.tagext.SimpleTag и содержит Java-код, необходимый для достижения желаемого поведения тега. Обработчик тега указан в определении тега в TLD, и контейнер использует эту информацию для сопоставления тега в JSP с кодом Java, который должен выполняться вместо этого тега.

Тем не менее, теги не всегда должны быть написаны как Java классы явно. Так же, как контейнер может переводить и компилировать JSP в HttpServlet, он также может переводить и компилировать файлы тегов в SimpleTag. Файлы тегов не такие мощные, как прямой Java-код, и Вы не можете делать синтаксический анализ вложенных тегов в файле тегов, но тег-файлы имеют преимущества использования простой разметки, такой как в JSP, и позволяют использование других тегов JSP внутри них. Определение тега в TLD может указывать либо на класс обработчика тега, либо на файл тега. Однако не нужно создавать TLD для использования тега, определенного в файле тега. Директива taglib позволяет сделать это, используя атрибут tagdir:

```
<%@ taglib prefix="myTags" tagdir="/WEB-INF/tags" %>
```

Обратите внимание, что эта директива taglib отличается от тех, которые Вы видели ранее. Вместо указания URI для файла TLD, содержащего определения библиотеки тегов, она указывает каталог, в котором файлы тегов могут быть найдены. В этом случае любые файлы *.tag* или *.tagx* в каталоге tagdir привязаны к пространству имен myTags. Файлы тегов в приложении должны находиться в каталоге */WEB-INF/tags*, но они также могут находиться в подкаталоге этого каталога. Можно использовать это, чтобы иметь несколько пространств имен файлов тегов в приложении, как показано в следующем примере.

```
<%@ taglib prefix="t" tagdir="/WEB-INF/tags/template" %>
<%@ taglib prefix="f" tagdir="/WEB-INF/tags/formats" %>
```

Разница между *.tag* и *.tagx* такая же, как и между *.jsp* и *.jspx*. Файлы *.tag* содержат синтаксис JSP, в то время как *.tagx*-файлы содержат синтаксис JSP Document (XML).

Файлы тегов JSP также могут быть определены в JAR-файлах в каталоге приложения */WEB-INF/lib*, но правила немного отличаются. Тогда как файлы тегов в приложении должны быть в */WEB-INF/tags* и могут быть объявлены либо в TLD, либо с директивой taglib, указывающей на каталог, файлы тегов в JAR-файле помещаются в каталог */META-INF/tags* и должны объявляться в TLD в */META-INF/* каталоге (или подкаталоге) одного и того же файла JAR.

Чтение библиотеки тегов Java TLD

Чтобы написать пользовательские библиотеки тегов и функций, необходимо понимать библиотеку JSP тегов XSD и как писать теги с ней. Лучший способ про-

демонстрировать это на примере, поэтому взгляните на TLD для библиотеки тегов Core из JSTL. Она находится в файле *org.glassfish.web:javax.servlet.jsp.jstl* Maven artifact JAR (найдите файл */META-INF/c.tld*). Начните с поиска первоначального объявления внутри файла:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--...Copyright (c) 2010 Oracle and/or its
    affiliates. All rights reserved...-->
<taglib xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/
    web-jsptaglibrary_2_1.xsd"
version="2.1">
...
</taglib>
```

Этот код устанавливает корневой элемент документа и объявляет, что документ использует определение схемы XML *web-jsptaglibrary_2_1.xsd*. Эта XML-схема определяет, как структурируется TLD, так же как *web-app_3_1.xsd* в *web.xml* определяет, как структурирован дескриптор развертывания. Единственное, что важно для библиотеки JSP тегов XSD — то, что схема использует строгую последовательность элементов, или файл TLD не будет валидным. Первые пять элементов в корне документа объявляют общую информацию о библиотеке тегов.

```
<description>JSTL 1.2 core library</description>
<display-name>JSTL core</display-name>
<tlib-version>1.2</tlib-version>
<short-name>c</short-name>
<uri>http://java.sun.com/jsp/jstl/core</uri>
```

Комментарии для кода:

- Элементы <description> и <display-name> предоставляют имена, которые инструменты XML (например IDE) могут отображать, но не имеют отношения к фактическому содержимому TLD и являются полностью необязательными. Фактически может быть столько тегов <description> и <display-name>, сколько хотите, позволяя Вам указывать разные отображаемые имена и описания для разных языков.
- <tlib-version> является обязательным элементом. Он определяет версию библиотеки тегов и должен содержать только числа и периоды (группы чисел, которые могут быть разделены только одним периодом).
- <short-name> указывает предпочтительный префикс по умолчанию (или пространство имен) для этой библиотеки тегов и также является обязательным. Он не может содержать пробелов или начинаться с цифры или знака подчеркивания.
- <uri>, определяет URI для этой библиотеки тегов. Этот элемент не обязательный, и отсутствие URI означает, что <jsp-config> в дескрипторе развертывания должен содержать объявление <taglib> для этой библиотеки тегов, которое можно использовать. Лучше всегда использовать <uri>, несмотря на то, что он является необязательным, и помните, что он не должен быть URL-адресом для фактического ресурса. В TLD может быть только один <uri>.

Определение валидаторов и слушателей

Следующий элемент TLD ядра определяет валидатор.

```
<validator>
    <description>
    …
    </description>
    <validator-class>
        org.apache.taglibs.standard.tlv.JstlCoreTLV
    </validator-class>
</validator>
```

Валидаторы расширяют класс javax.servlet.jsp.tagext.TagLibraryValidator для проверки JSP во время компиляции, чтобы он правильно использовал библиотеку. Валидатор в этом случае выполняет проверки, которые, например, удостоверяются, что тег <c:param> вложен только в теги, которые его поддерживают (например, <c:url> и <c:import>). Элементы валидатора имеют вложенные, по порядку, ноль или более элементов <description>, один требуемый элемент <validator-class> и ноль или более элементов <init-param>. Библиотека тегов может содержать ноль или более валидаторов.

Валидаторы должны быть объявлены после элемента <uri> и перед любыми элементами <listenener>. Объявление слушателей идентичны их аналогам в дескрипторе развертывания, и здесь может быть объявлен любой допустимый класс слушателя Java EE (ServletContextListener, HttpSessionListener и другие). Вы можете объявить ноль или более слушателей в TLD, и сразу же после этого можно объявить ноль или более тегов, которые являются следующим в Core TLD.

Определение тегов

Элемент <tag> является основополагающим компонентом TLD и отвечает за определение тегов в библиотеке тегов.

```
<tag>
    <description>
        Catch exceptions
    </description>
    <name>catch-exceptions</name>
    <tag-class>org.apache.taglibs.standard.tag.
                common.core.CatchTag</tag-class>
    <body-content>JSP</body-content>
    <attribute>
        <description>
        …
        </description>
        <name>Variable</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
</tag>
```

Тег может содержать ноль или более вложенных <description>, <display- name> и <icon>, так же как <taglib>. Как правило, здесь используется только <description>. После этого появляется требуемый элемент <name>, который указывает имя тега JSP. В этом случае полное имя тега <c:catch-exceptions>, где с — это <short-name> (префикс) для библиотеки тегов, а catch-exceptions — это <имя> тега. Очевидно, что у тега есть только одно имя. Далее идет элемент <tag-class> и указывает класс обработчика тега (Tag или SimpleTag), ответственный за выполнение тега. В этом примере не показан необя-

зательный элемент <tei-class>, который может быть следующим, и указывает расширение javax.servlet.jsp.tagext.TagExtraInfo для этого тега. Классы TagExtraInfo могут проверять атрибуты, используемые в теге во время интерпретации, чтобы убедиться, что их использование является правильным. В библиотеке тегов Core только теги <c:import> и <c:forEach> предоставляют дополнительную информацию классов.

Затем <body-content> указывает, какой тип содержимого вложенного в тег разрешен. Его допустимые значения:

- **empty** — означает, что тег не может содержать вложенный контент и должен быть пустым тегом.
- **scriptless** — теги с этим типом содержимого тела могут иметь текстовый шаблон, EL-выражения и JSP теги внутри них, но не скриптлеты или выражения. Объявления не допускаются в пределах содержимого вложенного тела тега.
- **JSP** — указывает, что вложенный контент тега может быть любым контентом, который в противном случае действителен в JSP, включая скриптлеты.
- **tagdependent** — указывает контейнеру не анализировать вложенный контент тега, а вместо этого позволяет тегу анализировать сам контент. Обычно это означает, что контент является другим языком, таким как SQL, XML или зашифрованными данными.

После <body-content> могут быть ноль или более <variable> элементов. Эти элементы предоставляют информацию о переменных, объявленных в результате использования этого тега. Элемент <variable> имеет следую-

щие подэлементы, которые предоставляют дополнительную информацию об объявленных переменных:

- **<description>** — необязательное описание для переменной.
- **<name-given>** — имя переменной, созданной этим тегом, должен быть допустимый идентификатор Java.
- **<name-from-attribute>** — имя атрибута, значение которого определяет, каким будет имя переменной. Этот элемент конфликтует с <name-given>. Вы должны указать оба этих элемента, но только один из них должен иметь значение. Другой всегда должен быть пустым. Элемент <name-from-attribute> всегда должен ссылаться на имя атрибута тега, тип которого является String, что не позволяет выражениям времени исполнения в его значении и используется для указания имени переменной.
- **<variable-class>** — необязательный элемент указывает полное имя класса Java для типа объявленной переменной. Если не указано, предполагается, что это String.
- **<declare>** — boolean-элемент, который по умолчанию имеет значение true, указывает, является ли заданная переменная новой переменной, для которой требуется объявление. Если false, это означает, что переменная уже определена в другом месте и просто изменяется.
- **<scope>** — указывает область, в которой будет определена эта переменная. Значение по умолчанию — NESTED, что означает, что переменная доступна только для кода и действий, вложенных в тег. Другими допустимыми значениями являются AT_BEGIN,

указывает на то, что переменная находится в области видимости кода, вложенного в тег и код, идущий после тега. AT_END — означает, что переменная относится только к области кода, которая приходит после тега, а не для кода, вложенного в тег.

После тегов <variable> можно определить ноль или более тегов <attribute>, что было рассмотрено в теге <c:catch-exceptions>, определенном ранее. <attribute> определяют атрибуты, которые могут быть указаны для этого тега. Существует несколько подэлементов <attribute>, которые определяют детали определяемого им атрибута.

- **<description>** — необязательный тег, указывает описание для этого атрибута.
- **<name>** — имя атрибута, значение должно быть допустимым идентификатором Java.
- **<required>** — необязательное булевское значение, указывающее, требуется ли этот атрибут при использовании этого тега. Значение по умолчанию false.
- **<rtexprvalue>** — необязательный булевский элемент, который по умолчанию имеет значение false, указывает, позволяет ли атрибут выражениям времени выполнения (EL или сценариям) определять значение атрибута. Если true, допустимы выражения времени выполнения. Если false, значения атрибута считаются статическими, а выражения времени выполнения приводят к ошибке на этапе компиляции.
- **<type>** — необязательный элемент указывает полное имя класса Java для типа атрибута. Если не указано, тип считается Object.

- **<deferred-value>** — необязательный элемент указывает, что значение атрибута является отложенным значением EL-выражения, и в результате обработчик тега будет передавать значение атрибута как выражение javax.el.ValueExpression. Обычно контейнер вычисляет выражение перед привязкой его возвращаемого значения к значению атрибута. При отложенном значении не вычисленное значение ValueExpression привязывается к атрибуту. Затем обработчик тега может вычислить это выражение ноль или более раз по мере необходимости. По умолчанию предполагается, что значение этого выражения будет принудительно приведено к Object, но вложенный элемент <type> может указать более точный тип, что особенно полезно при использовании этого атрибута в среде IDE.
- **<deferred-method>** — необязательный элемент указывает, что значение атрибута является отложенным методом EL-выражения, и в результате обработчик тега будет передавать значение атрибута как выражение javax.el.MethodExpression. Это похоже на <deferred-value>, за исключением типа выражения. По умолчанию предполагается, что сигнатура метода в выражении void method(), но Вы можете использовать вложенный элемент <method-signature> для указания более точной сигнатуры (например, void execute (java.lang.String) или boolean test (java.lang.Object)), так что ожидаемый тип возвращаемого значения, а также количество и тип параметров правильно документированы. Имя метода здесь фактически не важно, и контейнер игнорирует его.

- **<fragment>** — необязательный булевский элемент, если значение true — он создает атрибут type javax.servlet.jsp.tagext.JspFragment и сообщает контейнеру не анализировать содержимое JSP, содержащееся в значении атрибута. Обработчик тега может проанализировать фрагмент ноль или более раз. Значение по умолчанию — false. Код, который использует тег, может указывать значение атрибута с помощью вложенного тега <jsp:attribute> вместо фактического атрибута XML, как в следующем примере. Это так, даже если содержимое тела установлено как empty.

```
<myTags:myAction>
    <jsp:attribute name="myAttribute">
        Some text <fmt:message
                    key="including.jsp.tags" />.
    </jsp:attribute>
</myTags:doSomething>
```

Связанный элемент, <dynamic-attributes>, следует за тегами <attribute>. Можно указать этот булевский элемент не более одного раза. Значение по умолчанию — false, и оно указывает, разрешены ли еще атрибуты, не указанные другими элементами <attribute>. Это чаще всего встречается в теге JSP, который в итоге выводит HTML-тег. Код для этого тега может получить динамические атрибуты, а затем скопировать их из тега JSP в тег HTML во время выполнения. Динамические атрибуты всегда допускают EL-выражения и сценарии как значения. Чтобы использовать динамические атрибуты, класс обработчика тегов должен реализовать javax.servlet.jsp.tagext.DynamicAttributes.

Следующим <dynamic-attributes> является необязательный элемент <example> (может быть только один). Он связан с <description> и содержит простой текст, с примерами использования тега. Ваш тег может иметь ноль или более <tag-extension>. Этот тег предоставляет дополнительную информацию о теге, полезную для IDE и валидаторов. Расширения тегов никогда не влияют на поведение тега или контейнера. Они абстрактны и не содержат подэлементов, вместо этого разработчик отвечает за определение схемы для расширений тегов, которые он хочет реализовать.

Определение файлов тегов

Вы уже видели, как Вы можете использовать директиву taglib для сбора каталога тегов в пространстве имен пользовательских тегов, и теперь Вы знаете, что файлы тегов по существу являются JSP с немного отличающейся семантикой. Однако Вы также должны знать, как определять файлы тегов в дескрипторе библиотеки тегов. Помните, что файлы тегов, отправленные в библиотеку JAR, должны быть определены внутри TLD. Если есть файлы тегов, которые нужно группировать с обработчиками тегов или функциями JSP в одном и том же пространстве имен, необходимо определить файлы тегов в TLD, даже если они не отправлены в библиотеку JAR.

После всех элементов <tag> в TLD можно поместить ноль или более элементов <tag-file> для определения файлов тегов, принадлежащих библиотеке. Внутри элемента <tag-file> есть необязательные элементы <description>, <display-name> и <icon>. Как правило, указывается толь-

ко <description>. Элемент <name> указывает, какое имя тега следует за префиксом. Следующий элемент <path>, который указывает путь к файлу *.tag*, реализующему этот настраиваемый тег. Значение <path> должно начинаться с /WEB-INF/tags в веб-приложениях и /META-INF/tags в JAR-файлах. Последние два элемента: <example> и <tag-extension>, являются аналогами таких же тегов в элементе <tag>. Следующий XML демонстрирует основное использование <tag-file>.

```
<tag-file>
    <description>Some text.</description>
    <name>some</name>
    <path>/WEB-INF/tags/some.tag</path>
</tag-file>
```

Определение функций

После определения <tag-file> в TLD можно определить ноль или более JSP-функций, используя элемент <function>. Можно просмотреть функции JSTL, определенные в fn.tld, которые также можно найти в /META-INF/ в артефакте Maven. Этот файл содержит заголовок с описанием библиотеки тегов, отображаемым именем, версией, сокращенным именем и URI. Затем следует функция:

```
<function>
    <description>
        Occurrence of a substring in a string
    </description>
    <name>string-contains</name>
    <function-class>
        org.apache.taglibs.standard.functions.Functions
```

```
    </function-class>
    <function-signature>
       boolean contains(java.lang.String,
                        java.lang.String)
    </function-signature>
    <example>
       &lt;c:if test="${fn: string-contains(str1,
                        str2)}">
    </example>
</function>
```

Определение функции в TLD необычайно просто. Не смотря на элементы <description>, <display-name>, <icon> и <name>, с которыми Вы уже знакомы, важными элементами в этом примере являются <function-class> и <function-signature>. Класс функции — это полное имя стандартного Java класса, а сигнатура функции является сигнатурой статического метода в этом классе. Любой публичный статический метод в любом открытом классе может стать JSP-функцией. Последние два элемента, которые можно использовать в <function>, это <example> и <function-extension>, которые аналогичны элементам <example> и <tag-extension> в определениях <tag> и <tag-file>.

Определение расширений библиотеки тегов

После того, как все теги <tag>, <tag-file> и <function> есть в TLD, Вы можете определить ноль или более тегов расширения библиотеки тегов с помощью элемента <taglib-extension>. Расширения библиотеки тегов не влияют на поведение тегов или контейнеров и просто существуют для поддержки инструментария. Их концепция абстрактна, и в <taglib-extension> нет предопределенных подэлементов.

Сравнение директив JSP и директив Tag File

Файлы тегов работают также как файлы JSP. Они содержат один и тот же синтаксис и должны следовать одним и тем же основным правилам, и во время выполнения они переводятся и компилируются в Java, как JSP. Файлы тегов могут использовать любой обычный текст шаблона (включая HTML), любой другой тег JSP, декларации, сценарии, выражения и язык выражений. Однако не удивительно, что между двумя файловыми форматами существуют небольшие различия, в основном касающиеся директив, доступных для файлов тегов. Файлы тегов также могут использовать директивы include и taglib для включения файлов и других библиотек тегов в JSP, но в файлах тегов нет директивы page. Директива include может использоваться для включения файлов *.jsp*, *.jspf* и других *.tag* в .tag-файле или *.jspx* и других .tagx-файлах в .tagx-файле. Использование директивы taglib в файле тега идентично использованию одного в JSP-файле.

Вместо директивы page, файлы тегов имеют директиву tag. Эта директива заменяет необходимую функциональность директивы page из JSP, а также заменяет многие элементы конфигурации из элемента <tag> в файле TLD. Директива tag имеет следующие атрибуты, ни один из которых не является обязательным:

- **pageEncoding** — устанавливает кодировку символов вывода тега.
- **isELIgnored** — указывает контейнеру не вычислять EL выражения в файле тега и по умолчанию — false.
- **language** — указывает язык сценариев, используемый в файле тега.

- **deferredSyntaxAllowedAsLiteral** — указывает контейнеру игнорировать и не анализировать отложенный синтаксис EL в файле тега.
- **trimDirectiveWhitespaces** –указывает контейнеру обрезать пробел вокруг директив.
- **import** — можно указать один или несколько разделенных запятыми классов Java для импорта в этот атрибут, и можно использовать этот атрибут несколько раз.
- **description** — эквивалент элемента <description> в файле TLD.
- **display-name** — эквивалентно элементу <display-name> в файле TLD.
- **small-icon** и **large-icon** — аналогичны элементу <icon> в TLD.
- **body-content** — аналог <body-content> в TLD с одним незначительным изменением: его допустимые значения empty, scriptless и tagdependent. Значение JSP, доступное в TLD, недопустимо для содержимого тела тега, указанного в файле тега. Нельзя использовать сценарии или выражения внутри содержимого вложенного тела при использовании тега, который был определен в файле тега. Scriptless — значение по умолчанию для этого атрибута.
- **dynamic-attributes** — строковый атрибут указывает, включены ли динамические атрибуты. По умолчанию значение пустое, означает, что динамические атрибуты не поддерживаются. Чтобы включить динамические атрибуты, установите его значение в имя переменной EL, которую Вы хотите создать, чтобы сохранить все

динамические атрибуты. У переменной EL будет тип Map <String, String>. Где ключи — динамические имена атрибутов и значения — атрибуты значений.

- **example** — атрибут можно использовать для указания использования тега примера.

Эквивалентные атрибуты директивы отсутствуют для элементов <name>, <tag-class>, <tei-class>, <variable>, <attribute> и <tag-extension>. Имя тега всегда равно имени файла тега (минус расширение *.tag*), а <tag-class> не требуется, поскольку файл тега является обработчиком тега. Невозможно указать класс TagExtraInfo или расширение тега для файлов тегов, потому что нет эквивалента. Теги <variable> и <attribute> заменяются директивами variable и attribute соответственно.

Директива variable предоставляет атрибуты description, name-given, name-from-attribute, variable-class, declare и scope, которые эквивалентны их идентично названным элементам в TLD. Он также предоставляет дополнительный атрибут alias, который позволяет указать имя локальной переменной, которое можно использовать для ссылки на переменную в файле тега. Директива attribute имеет атрибуты description, name, required, rtexprvalue, type, fragment, deferredValue, deferredValueType, deferredMethod и deferredMethodSignature, которые соответствуют элементам в TLD.

Примеры использования Tag Files и Custom Tags

Теперь, когда Вы знакомы с деталями дескрипторов библиотеки тегов и файлов тегов, пришло время создать свой первый пользовательский тег JSP. Самый простой

способ создать собственный тег — написать файл тега и использовать директиву taglib с атрибутом tagdir. Реализуем проект TagExample. Для того чтобы отключить Java в JSP изменим стандартный дескриптор развертывания: <scripting-invalid> false </ scripting-invalid> изменен на <scripting-invalid> true </ scripting-invalid>. Проект также имеет файл *index.jsp* в корневом каталоге веб-сайтов, который перенаправляет /index с помощью <c:url> и */WEB-INF/jsp/base.jspf*-файл со следующими объявлениями библиотеки тегов:

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt"
    uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="fn"
    uri="http://java.sun.com/jsp/jstl/functions" %>
<%@ taglib prefix="template"
    tagdir="/WEB-INF/tags/template" %>
```

Затем создайте простой сервлет, который может отвечать на запросы в /index.

```
@WebServlet(
        name = " TagExampleServlet",
        urlPatterns = "/index"
)
public class IndexServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException
    {
```

```
        request.getRequestDispatcher(
                "/WEB-INF/jsp/view/main.jsp")
                .forward(request, response);
    }
}
```

Одной из самых мощных вещей, которые Вы можете делать с файлами тегов, является создание системы шаблонов HTML для приложения. Эта система шаблонов может позаботиться о многих повторяющихся задачах, необходимых на страницах приложения, сокращая дублированный код и упрощая изменение дизайна сайта. Чтобы продемонстрировать это, создайте файл */WEB-INF/tags/template/main.tag,* содержащий основной формат JSP, который используется большинством страниц в приложении.

```
<%@ tag body-content="scriptless"
        dynamic-attributes="dynamicAttributes"
        trimDirectiveWhitespaces="true" %>
<%@ attribute name="title"
        type="java.lang.String" rtexprvalue="true"
        required="true" %>
<%@ include file="/WEB-INF/jsp/base.jspf" %>

<!DOCTYPE html>
<html<c:forEach items="${dynamicAttributes}" var="i">
    <c:out value=' ${i.key}=
    "${fn:escapeXml(i.value)}"' escapeXml="false" />
</c:forEach>>
    <head>
        <title><c:out value="${fn:trim(title)}" />
        </title>
    </head>
```

```
    <body>
        <jsp:doBody />
    </body>

</html>
```

Разберем пример:

- Первая директива в файле устанавливает, что использование тега может содержать содержимое тела, что оно поддерживает динамические атрибуты, которые доступны с помощью переменной dynamicAttributes EL, и пустое пространство вокруг директивы должно быть обрезано. Для этого используется атрибут trimDirectiveWhitespace.
- Вторая директива устанавливает явный атрибут title.
- Третий атрибут включает файл base.jspf, потому, что <jsp-config> не влияет на файлы тегов.
- Цикл <c: forEach> копирует все динамические атрибуты в тег <html>.
- Тег <c:out> в цикле гарантирует, что пространство между каждым атрибутом не игнорируется и значения атрибута экранируются должным образом.
- Атрибут title выводится как документ <title>, используя <c:out> и EL-выражение обрезает его значение.
- Специальный тег <jsp:doBody> используется в HTML <body>. Этот тег может использоваться только в файлах тегов, сообщает контейнеру вычислить содержимое тега JSP-тега и поместить его в строку. Также можно указать атрибуты var или varReader и атрибут scope,

чтобы вывести вычисленное содержимое тела в переменную, а не встраивать ее.

Теперь, чтобы использовать все это, создайте файл */WEB-INF/jsp/view/main.jsp*, который вызывает тэг <template:main>, который Вы создали.

```
<template:main title="Example custom tag">
     Example page
</template:main>
```

Скомпилируйте и запустите приложение, перейдите по ссылке *http://localhost:8080/TagExample /index*. Если Вы просмотрите источник ответа для результирующей страницы, Вы увидите, что заголовок документа — «Example custom tag» и что текст «Example page» был помещен в тело документа. Вы успешно создали свой первый пользовательский тег JSP!

18. Почтовые возможности JSP

Рассмотрим создание простого веб-приложения для отправки сообщения электронной почты. Сперва создадим служебный класс ServiceEmail:

```
public class ServiceEmail {
    public static void sendEmail(String host,
           String port, String user, String password,
           String address, String topic, String text)
           throws AddressException,
           MessagingException {

       Properties properties = new Properties();
       properties.put("mail.smtp.host", host);
       properties.put("mail.smtp.port", port);
       properties.put("mail.smtp.auth", "true");
       properties.put("mail.smtp.starttls.enable", "true");

       Authenticator auth = new Authenticator() {
           public PasswordAuthentication
           getPasswordAuthentication() {
              return new PasswordAuthentication(user,
                     password);
           }
      };

      Session session =
          Session.getInstance(properties, auth);
      Message message = new MimeMessage(session);
      message.setFrom(new InternetAddress(user));
      InternetAddress[] addresses =
          { new InternetAddress(address) };
```

```
            message.setRecipients(Message.RecipientType.TO,
                    addresses);
            message.setSubject(topic);
            message.setSentDate(new Date());
            message.setText(text);
            Transport.send(message);
        }
}
```

Класс имеет один статический метод sendEmail, который принимает параметры SMTP-сервера и данные сообщения в качестве аргументов. Разместим настройки SMTP-сервера в файле *web.xml* приложения. Так же добавим файл *mail.jar* JavaMail в каталог *WEB-INF/lib*. Для этого Вам нужна библиотека *javax.mail*.jar. Загрузите ее с Java.net.

Теперь создадим форму электронной почты в JSP, используя шаблон из предыдущего примера. Форма отправки сообщения электронной почты будет выглядеть следующим образом:

```
<template:main title="SendEmail">
    <form action=" ServiceEmailServlet" method="post">
        <table border="0" width="40%" align="center">
            <caption><h2>Send E-mail</h2></caption>
            <tr>
                <td width="50%">Address </td>
                <td><input type="text" name="address"
                           size="30"/></td>
            </tr>
            <tr>
                <td>Topic </td>
                <td><input type="text" name="topic"
                           size="30"/></td>
            </tr>
```

```
            <tr>
                <td>Text </td>
                <td><textarea rows="5" cols="30"
                              name="text">
                    </textarea>
                </td>
            </tr>
            <tr>
                <td colspan="2" align="center">
                   <input type="submit" value="Send"/>
                </td>
            </tr>
        </table>
    </form>
</template:main>
```

Обратите внимание, что атрибут action формы указывает на URL сервлета, который будет создан следующий. Сервлет выполняет следующие задачи:

- Чтение настроек SMTP-сервера из файла *web.xml*.
- Получение данных на странице *FormSendEmail.jsp*.
- Вызов метода sendEmail класса ServiceEmail для отправки сообщения электронной почты.

Код сервлета:

```
@WebServlet("/ServiceEmailServlet")
public class ServiceEmailServlet extends HttpServlet
{
    private String user;
    private String pass;
    private String host;
    private String port;
     public void init() {
        ServletContext context = getServletContext();
```

```
        user = context.getInitParameter("user");
        pass = context.getInitParameter("pass");
        host = context.getInitParameter("host");
        port = context.getInitParameter("port");
    }

    protected void doPost(HttpServletRequest request,
            HttpServletResponse response) throws
            ServletException, IOException {
        String address =
            request.getParameter("address");
        String topic = request.getParameter("topic");
        String text = request.getParameter("text");

        ServiceEmail.sendEmail(host, port, user,
            pass, address, topic, text);
        }
    }
}
```

Как мы видим, сервлет ServiceEmailServlet — загружает конфигурацию для SMTP-сервера при инициализации в методе init и обрабатывает запросы пользователя в методе doPost. Он считывает информацию о сообщении (получателя, тему и содержимое) со страницы FormSendEmail.jsp и отправляет электронное письмо, вызывая метод sendEmail класса ServiceEmail.

Сконфигурируйте параметры для SMTP-сервера в файле дескриптора развертывания следующим образом:

```
<context-param>
    <param-name>host</param-name>
    <param-value>smtp.gmail.com</param-value>
</context-param>
```

```
<context-param>
    <param-name>port</param-name>
    <param-value>587</param-value>
</context-param>
<context-param>
    <param-name>user</param-name>
    <param-value>***EMAIL***</param-value>
</context-param>
<context-param>
    <param-name>pass</param-name>
    <param-value>***PASSWORD***</param-value>
</context-param>
```

Параметры SMTP настраиваются как параметры контекста. В этом приложении используется SMTP-учетная запись Gmail, и вы должны изменить адрес электронной почты отправителя и пароль в соответствии с настройками своей учетной записи. Проверьте поставщика услуг электронной почты, чтобы убедиться, что SMTP включен для вашей учетной записи.

Скомпилируйте и запустите приложение, перейдите по ссылке *http://localhost:8080/ServiceEmailServlet/*. Протестируйте работу приложения.

Доработайте приложение таким образом, чтобы система выдавала пользователю сообщение о состоянии отправки (сообщение отправлено успешно/ошибка отправки сообщения).

19. Домашнее задание

Создать упрощённую версию интернет магазина. Основной функционал:

- просмотр списка продуктов
- просмотр корзины
- добавление продуктов в корзину
- удаление продуктов из корзины.

Отображение списка продуктов и корзины покупок реализуйте через JSP. Реализовать навигацию по страницам. Использовать сессии и cookie-файлы.