Что такое JSP?

- JSP (JavaServer Pages) — платформенно-независимая, переносимая и легко расширяемая технология для разработки веб-приложений, работающая на виртуальной машине Java (JVM). JSP позволяет веб-разработчикам создавать содержимое, состоящее из статических исходных данных, которые могут быть оформлены в одном из текстовых форматов HTML, SVG, WML, или XML, и JSP-элементов, которые конструируют динамическое содержимое. Кроме этого могут использоваться библиотеки JSP-тегов, а также Expression Language (EL), для внедрения Java-кода в статичное содержимое JSP-страниц.

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 HTMLпримеру. 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.

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 JSPскриптовый 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 ContentType 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.

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

```
<%@ 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 пустым 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.

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 чтото 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.
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().

Использование 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.

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 предоставляет 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.

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 контейнера 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.

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" %>
```

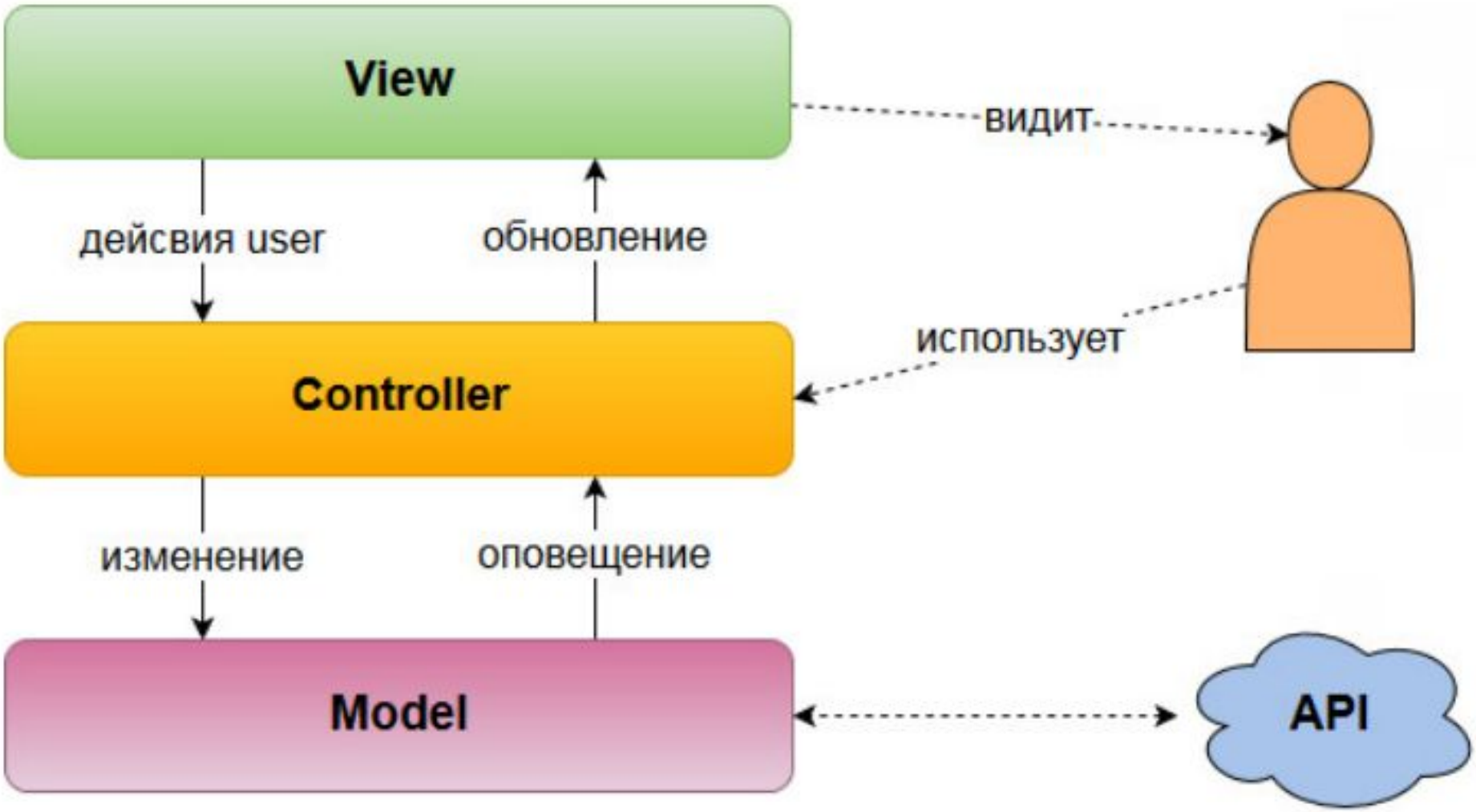
Атрибут errorPage позволяет передать исключение для обработки страницы JSP, которую предварительно создал программист. У этой страницы-обработчика исключения атрибут isErrorPage равен true:

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

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.


View
видит
дейсвия user
обновление
используует
Controller
изменение
оповещение
Model
API

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

- 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.

- 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.

- Представление. Этот уровень представляет собой результат приложения, обычно это форма пользовательского интерфейса. Уровень представления используется 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>
```

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. ServletOutputStream 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 /><%
    }
%>
```