Колок Саковича

[1. Платформа J2EE. Состав технологии J2EE (4 пункта, в т.ч. «образцовые реализации»). 2](#_Toc452548787)

[2. Обзор Java-технологий. Архитектура J2EE. 3](#_Toc452548788)

[3. Сервлеты. Жизненный цикл. Передача параметров. Интерфейс SingleThreadModel. 4](#_Toc452548789)

[4. Структура HTTP-запроса. Заголовки. 6](#_Toc452548790)

[5. Классификация кодов состояния. Примеры. 7](#_Toc452548791)

[6. Структура HTTP-ответа. Заголовки. 8](#_Toc452548792)

[7. Использование cookies. 8](#_Toc452548793)

[8. Поддержка сессии. 9](#_Toc452548794)

[9. Технология JSP (Java Server Pages). Элементы сценариев JSP. 10](#_Toc452548795)

[10. Директива page. Другие директивы. 11](#_Toc452548796)

[11. Включение файлов в JSP-документы. Интеграция сервлетов и JSP. Перенаправление запросов. 14](#_Toc452548797)

[12. JavaBeans в JSP. 15](#_Toc452548798)

[13. Определение новых JSP-дескрипторов. Обработка тела JSP-дескриптора. 16](#_Toc452548799)

[14. Определение новых JSP-дескрипторов. Вложенные JSP-дескрипторы. 16](#_Toc452548800)

[15. Библиотека JSTL. Обзор возможностей. 17](#_Toc452548801)

# Платформа J2EE. Состав технологии J2EE (4 пункта, в т.ч. «образцовые реализации»).

1. Спецификации - то, чему должен соответствовать веб сервер.
2. образцовые реализации - примеры серверов.
3. модель приложения - показывает разработчикам что хорошо делать, а что - плохо.
4. тесты - набор тестов для проверки веб сервера (для создателей сервера)

JMS (Java Message Service) — это набор интерфейсов и стандарт обмена сообщениями, позволяющий компонентам J2EE-приложения создавать, посылать, принимать и читать сообщения. Он обеспечивает двустороннее, надежное, асинхронное распределенное соединение, допускает взаимодействие с существующими MOM (Message-Oriented Middleware) системами. JMS поддерживает работу как в стиле "отправитель-получатель", так и в режиме "рассылка-подписка".

**Другие средства**, входящие в J2EE:

o JDBC (Java DataBase Connectivity) — стандарт взаимодействия J2EE с реляционными базами данных, позволяющий "соединить" Java-программы и SQL. API JDBC позволяет вызывать SQL-команды из модулей, написанных на языке Java. Основными частями технологии JDBC являются JDBC API (набор классов и методов, к которым обращается прикладной программист) и JDBC-драйверы, которые транслируют эти вызовы в команды API конкретной СУБД. JDBC поддерживает распределенные базы данных и двухфазное завершение транзакций.

o RMI (Remote Method Invocation) — технология обеспечения удаленного взаимодействия объектов в распределенных системах, написанных на Java. Для RMI характерна передача объектов "по значению" — с использованием механизма сериализации Java.

# Обзор Java-технологий. Архитектура J2EE.

**Java SE** — Java Standard Edition, основное издание Java, содержит компиляторы, API, Java Runtime Environment; подходит для создания пользовательских приложений, в первую очередь — для настольных систем.

**Java EE** — Java Enterprise Edition, представляет собой набор спецификаций для создания программного обеспечения уровня предприятия.

**Java ME** — Java Micro Edition, создана для использования в устройствах, ограниченных по вычислительной мощности, например в мобильных телефонах, КПК, встроенных системах;

**Java FX** — технология, являющаяся следующим шагом в эволюции Java как Rich Client Platform; предназначена для создания графических интерфейсов корпоративных приложений и бизнеса.

**Java Card** — технология предоставляет безопасную среду для приложений, работающих на смарт-картах и других устройствах с очень ограниченным объемом памяти и возможностями обработки.

**Архитектура**

Платформа J2EE использует модель многоуровневого распределенного приложения.

* Компоненты клиентского уровня работают на клиентской машине.
* Компоненты Web-уровня работают на J2EE-сервере.
* Компоненты бизнес-уровня работают на J2EE-сервере.
* Программное обеспечение уровня корпоративной информационной системы (EIS) работает на EIS-сервере.

В общем, все как на картинке. Главный аспект, о котором упоминал Сакович, это введение модуля бизнес-логики, которая является буфером между клиентом и сервером. Клиент как бэ ничего не знает о механизмах, он просто говорит бизнес-логике, че делать, а та уже работает.

# Сервлеты. Жизненный цикл. Передача параметров. Интерфейс SingleThreadModel.

Сервлет является Java-интерфейсом, реализация которого расширяет функциональные возможности сервера. Сервлет взаимодействует с клиентами посредством принципа запрос-ответ.

С момента загрузки кода сервлета, сервер создает только один экземпляр класса. И этот единственный экземпляр управляет каждым запросом к сервлету.

Servlet -> GenericServlet -> HttpServlet - > MyServlet

**Жизненный цикл сервлета состоит из следующих шагов**:

1. В случае отсутствия сервлета в контейнере.
   1. Класс сервлета загружается контейнером.
   2. Контейнер создает экземпляр класса сервлета.
   3. Контейнер вызывает метод init(). Этот метод инициализирует сервлет и вызывается в первую очередь, до того, как сервлет сможет обслуживать запросы. За весь жизненный цикл метод init()вызывается только один раз.
2. Обслуживание клиентского запроса. Каждый запрос обрабатывается в своем отдельном потоке. Контейнер вызывает метод service() для каждого запроса. Этот метод определяет тип пришедшего запроса и распределяет его в соответствующий этому типу метод для обработки запроса. Разработчик сервлета должен предоставить реализацию для этих методов. Если поступил запрос, метод для которого не реализован, вызывается метод родительского класса и обычно завершается возвращением ошибки инициатору запроса.
3. Destroy()

**Инициализация и Уничтожение**

Сервлеты имеют пару методов – init() и destroy(). Инициализация происходит в зависимости от конфигурации сервера:

- когда сервер стартует.

- когда сервлет впервые вызван для обработки запроса, непосредственно перед вызовом service().

- по запросу администратора сервиса.

В любом случае, init() гарантированно вызывается ДО обработки сервлетом первого запроса.

Метод destroy()вызывается сервером для удаления сервлета и должен содержать инструкции для освобождения ресурсов, которые не могут быть собраны сборщиком мусора. Также этот метод является последней возможностью для сервлета сохранить каким-либо образом свое состояние до следующего вызова init.

Первым вызовом в переопределенном методе init() должен быть super.init(config).

**Single Thread Model**

Хотя это стандарт – иметь один экземпляр сервлета, однако можно вместо этого работать с пулом (экземпляров одного и того же класса) сервлетов для обработки запросов. Для этого сервлет должен реализовать SingleThreadModel интерфейс. Данный интерфейс **не содержит методов** и является флагом для сервера, чтобы запускать сервлеты альтернативным способом.

В двух словах, SingleThreadModel указывает сервлет-контейнеру, что метод service нельзя одновременно выполнять в нескольких потоках.

Сервер в этом случае (согласно Servlet API документации) обязан гарантировать, что никакие два потока (thread) не будут одновременно выполнять код одного экземпляра сервлета. При таком поведении сервера в код сервлета можно не включать синхронизацию ресурсов.

**Передача параметров – HttpRequest, HttpResponce**

**GET-запрос**

GET-запрос — это самый распространенный вид HTTP-запроса. При помощи него происходит запрос браузером любого файла веб-сервера. **HTTP-запрос типа GET состоит только из HTTP-заголовков, тело у него отсутствует.**

При помощи GET-запроса можно передать веб-серверу параметры — некоторую информацию.

В первой строке запроса после ключевого слова GET помещается путь к запрашиваемому документу. Если в пути встречается знак вопроса, то принято считать, что в этом месте путь заканчивается, а за ним начинаются GET-параметры этого запроса:

*GET /examples/test.html?параметры HTTP/1.1*

GET-параметр имеет формат имя\_параметра=значение\_параметра, сами параметры разделяются знаком &. Пример GET-запроса с двумя параметрами:

*GET /enter?login=admin&password=qwerty HTTP/1.1*

GET-параметры можно вручную дописать в браузере в конце запрашиваемого URL в адресной строке.

Основным преимуществом GET-параметров является их размещение непосредственно в URL, что дает возможность сформировать гиперссылку на документ с определенными параметрами.

**POST-запрос**

Если необходимо передать на веб-сервер большой объем данных, например, текст сообщения или файл, используют POST-запрос. В этом типе запроса параметры помещаются в тело HTTP-запроса, а размер передаваемых данных в байтах указывается в заголовке Content-Length:

*POST /enter HTTP/1.1*

*<Различные заголовки>*

*Content-Length: 27*

*<Различные заголовки>*

*login=admin&password=qwerty*

Таким образом, в URL передаваемые параметры не видны. Простым способом сформировать POST-запрос не получится, они в основном генерируются с помощью HTML-форм. Метод – getParameter().

При получ запроса сервлет:

1. Читает информацию от пользователя
2. Читает прочую информацию
3. Обрабатывает
4. Формирует HTTP-ответа
5. Генерирует прочую информацию (непосредств не видна)

web.xml – параметры инициализации

<web-app>

<servlet>

<servlet-name>…</>

<servlet-class>…</>

<init-param>

<param-name>…</>

<param-name>…</>

</>

</servlet>

ServletConfig sc = get…();

String s = sc.getInitParameter(“…”);

# Структура HTTP-запроса. Заголовки.

*javax.servlet.http.HttpServletRequest*

**Строка запроса** – указывает метод передачи, URL-адрес, к которому нужно обратиться и версию протокола HTTP.

**Заголовки** – передают различные параметры и др. сведения и информациасю.

**Пустая строка**.

**Тело** – может отсутствовать.

Когда мы получаем ответный запрос от сервера, тело сообщения, чаще всего представляет собой содержимое веб-страницы. Но, при запросах к серверу, оно тоже может иногда присутствовать, например, когда мы передаем данные, которые заполнили в форме обратной связи на сервер – см. **Post**-запрос.

GET / HTTP/1.1 - запрос

Host: webgyry.info - заголовок

Все заголовки разделяются на четыре основных группы:

**Accept –** поддерживаемые форматы

**Accept-Encoding –** поддерживается ли сжатие

**Host –** узел, с которого идёт запрос

**User-Agent –** браузер

**Referer –** с какой страницы пришли

**Method –** метод, которым будет обрабатываться

**Accept-Language** – поддерживаемые языки

**Content-Length** – длина содержимого (для post-запросов)

**Authorization:** содержит информацию об аутентификации.

**From:** с его помощью браузер может посылать полный E-mail адрес пользователя серверу.

Например**:** From: [quake@doom.ru](mailto:quake@doom.ru)

**If-Modified-Since:** Используется только при методе GET ресурс возвращается ,если он был изменен с указаного момента времени, используется при кешировании.

**Например:** If-Modified-Since: Fri, 13 Dec 1998 09:15:45 GMT

setRequestProperty(key, value) – задать заголовок

запрос значения полей:

getHeader(String)

getIntHeader

getDateHeader

getИмя\_header : getMethod getHost

# Классификация кодов состояния. Примеры.

[1xx: Informational](http://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_%D0%BA%D0%BE%D0%B4%D0%BE%D0%B2_%D1%81%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D1%8F_HTTP#1xx) (информационные)

[2xx: Success](http://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_%D0%BA%D0%BE%D0%B4%D0%BE%D0%B2_%D1%81%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D1%8F_HTTP#2xx) (успешно)

[3xx: Redirection](http://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_%D0%BA%D0%BE%D0%B4%D0%BE%D0%B2_%D1%81%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D1%8F_HTTP#3xx) (перенаправление)

[4xx: Client Error](http://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_%D0%BA%D0%BE%D0%B4%D0%BE%D0%B2_%D1%81%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D1%8F_HTTP#4xx) (ошибка клиента)

[5xx: Server Error](http://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_%D0%BA%D0%BE%D0%B4%D0%BE%D0%B2_%D1%81%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D1%8F_HTTP#5xx) (ошибка сервера).

Примеры, которые давал Сакович:

100 – continue;

200 – OK;

201 – создано;

204 – no content – это не плохо, так как состояние все равно ОК. Здесь инф не поменялась, значит, браузер берет инфу из КЭШа

205 – reset content – формы сбрасываются на дефолт.

3xx – перенаправление. При этом будет выставлен спец. Заголовок – **location**

301 – перемещёно навсегда;

302 – перемещено временно;

400 – неверный запрос;

401 – не авторизован;

403 – запрещено;

404 – не найдено;

411 – length required – требование длины запроса Post;

500 – внутренняя ошибка сервера;

502 – bad gateway – проблемы со шлюзом;

503 – server unavailable – например,занят. Здесь header – **Retry-After**- время след. обращения;

504 – gateway timeout – шлюз не отвечает.

Установка кода состояния – метод setStatus()

sendRedirect(String) – метод, на который выполнено перенаправление

# Структура HTTP-ответа. Заголовки.

Этот интерфейс представляет абстракцию HTTP ответа сервера. Через него разработчик может установить cookies, HTTP заголовки, послать редирект на другой URL (например на другой Servlet), а так же сформировать ответ клиентскому браузеру, получив Writer и записав в него HTML страницу.

Каждый ответ состоит из трех частей: строка состояния (содержит три поля: версию HTTP, код состояния и описание), заголовок ответа (информация о сервере и передаваемых данных) и тела сообщения. Первые две части представлены тоже в текстовом виде и выглядит это примерно так:

*Request URL:http://ksayri.info/*

*Request Method:GET*

*Status Code:200 OK - строка состояния*

*Accept-Ranges:bytes - заголовки ответа*

*Connection:close*

*...*

*<html> … - тело сообщения*

ContentEncoding

Location

Retry-After

Content-Type

Cache-Control - Указывает директивы для механизмов кэширования. (max-age is measured in seconds)

Content-Length

Date – дата и время, когда было послано сообщение

Set-Cookie – http-ответ посылает cookie с сервера на браузер

Plain text — это формат данных, предоставляемый в текстовом виде для использования в[рассылке по электронной почте](https://www.unisender.com/ru/email-delivery/).

# Использование cookies.

Ку́ки — небольшой фрагмент данных, по сути, это текстовый файл, отправленный веб-сервером и хранимый на компьютере пользователя. Веб-клиент (обычно веб-браузер) всякий раз при попытке открыть страницу соответствующего сайта пересылает этот фрагмент данных веб-серверу в виде HTTP-запроса. Применяется для сохранения данных на стороне пользователя, на практике обычно используется для:

* аутентификации пользователя;
* хранения персональных предпочтений и настроек пользователя;
* отслеживания состояния сеанса[en] доступа пользователя;
* ведения статистики о пользователях.

*Куки на выполнение не запускаются!!* Поэтому они не вредоносны.

Куки на клиенте хранят информацию:

1. какие-то настройки
2. имя cookie
3. значение cookie
4. срок жизни (следит браузер)
5. домен(адрес сервера) указывается для того, чтобы для запросов брать определенные cookie
6. путь – кому cook нужен (относится к соответствующей части домена)

**Установка куки**

Запрашивая страницу, браузер отправляет веб-серверу короткий текст с HTTP-запросом. Например, для доступа к странице http://www.example.org/index.html, браузер отправляет на сервер www.example.org следующий запрос:

*GET /index.html HTTP/1.1*

*Host: www.example.org*

***браузер → сервер***

Сервер отвечает, отправляя запрашиваемую страницу вместе с текстом, содержащим HTTP-ответ. Там может содержаться указание браузеру сохранить куки:

*HTTP/1.1 200 OK*

*Content-type: text/html*

*Set-Cookie: name=value*

*(содержимое страницы)*

***браузер ← сервер***

Строка Set-cookie отправляется лишь тогда, когда сервер желает, чтобы браузер сохранил куки. В этом случае, если куки поддерживаются браузером и их приём включён, браузер запоминает строку name=value (имя = значение) и отправляет её обратно серверу с каждым последующим запросом. Например, при запросе следующей страницы http://www.example.org/spec.html браузер пошлёт серверу www.example.org следующий запрос:

*GET /spec.html HTTP/1.1*

*Host: www.example.org*

*Cookie: name=value*

*Accept: \*/\**

***браузер → сервер***

**Создание:**

Cookie cookie = new Cookie(“myCookie“, “value”);

response.addCookie(cookie);

**Получение:**

Cookie[] cookies = request.getCookies();

Можно задавать

1. время валидности cookie,
2. шифрование,
3. указать HTTPS протокол как обязательный и пр.

Как только браузер подключается к веб-приложению, он отправляет ему те и только те cookies, которые сохранены на клиентской машине для данного приложения.

**getComment/setComment**

**getDomain/setDomain –** *к какому сайту относится кук*

**getMaxAge/setMaxAge –** *время жизни с секундах после сеанса.*

**getName/setName**

**getPath/setPath –** *доступность кука отдельным разделам сайта. «/» - доступ ко всему сайту*

**getSecure/setSecure -** *как значения cookie файла передаются по сети. По умолчанию cookie не защищен, т. е. передается по обычному незащищенному HTTP соединению. Однако если cookie помечен как защищенный, он передается, только когда обмен между броузером и сервером организован по протоколу HTTPS или другому защищенному протоколу. По умолчанию принимает значение false.*

**getValue/setValue**

**getVersion/setVersion**

**Типы Cookie:**

1. Сессионные куки - такие cookie удаляются при закрытии клиента, то есть существуют только на протяжении текущего сеанса. Однако, если в браузере включено автоматическое восстановление сеанса, что случается очень часто, cookie сеанса может храниться постоянно, как если бы браузер никогда не закрывался.
2. Постоянные куки - удаляются не с закрытием клиента, а при наступлении определенной даты (атрибут Expires) или после определенного интервала времени (атрибут Max-Age).
3. Secure(безопасные) -отсылаются на сервер только если запрос выполняется по протоколу SSL и HTTPS.
4. HttpOnly - не доступны из JavaScript через свойства [Document.cookie](https://developer.mozilla.org/ru/docs/Web/API/Document/cookie) API

**Область видимость cookie:**

Директивы Domain  и Path определяют область видимости куки, то есть те URL'ы, к которым куки могут отсылаться.  
Атрибут Domain указывает хосты, к которым отсылаться куки. Если он не задан, то по умолчанию берется доменная часть документа (но без поддоменов).

Атрибут Path указывает URL, который должен быть в запрашиваемом ресурсе на момент отправки заголовка Cookie.

# Поддержка сессии.

**Сессия хранится на стороне сервера, однозначно идентифицирует клиента и хранит данные.**

Сессия характеризуется:

1. Id
2. Время создания/жизни

Способы поддержки сессии:

1. Передавать id с помощью сессионного cookie
2. В url можно увидеть session-id
3. Через hidden – элемент формы

Способ выбора определяет ApplicationServer.

Объект сессии создается каждый раз при получении запроса от нового клиента и впоследствии идентифицирует его уникальным образом.

Разным пользователям соответствуют различные объекты сессий. Получить доступ из сервлета к объекту сессии, и тем самым идентифицировать клиента, чей запрос находится на обработке, можно с помощью метода *getSession():* true – новая, false - старая

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletRequest {

HttpSession session = request.getSession(true);

В объекте сессии можно хранить персональную информацию о соответствующем клиенте. Как и в случае с cookie, соответствующая информация хранится в виде нескольких пар «название - значение». Однако, если в cookie значение записи - это некая строка (String), то в случае с сессией это некий объект (Object). Доступ к значениям, хранящимся в сессии, осуществляется с помощью методов putValue и getValue.

Integer sum = (Integer) session.getAttribute (SUM\_KEY);

Все классы, объекты которых планируется помещать в экземпляр класса HttpSession, должны реализовывать интерфейс ***Serializable***. Доступ к объектам атрибутов осуществляется посредством методов *setAttribute()* и *getAttribute().* Однако для некоторых атрибутов в объекте сессии может потребоваться дополнительное анонсирование, которое выполняется с помощью методов *valueBound()* и *valueUnbound().*

Объект сессии сохраняется до тех пор, пока продолжают поступать HTTP запросы от соответствующего клиента. Если в течение определенного контрольного интервала времени новых запросов от клиента не поступает, соответствующий объект сессии автоматически уничтожается, а персональная информация о клиенте - теряется.

# Технология JSP (Java Server Pages). Элементы сценариев JSP.

JSP – надстройка над servlet и она превращается в сервлет при первом обращении(происходит компиляция).

При последующих обращениях компиляции не происходит.

Writer printer

Response -> getWiter.

JSP (JavaServer Pages) — технология, позволяющая веб-разработчикам легко создавать содержимое, которое имеет как статические, так и динамические компоненты. По сути, страница JSP является текстовым документом, который содержит текст двух типов: статические исходные данные, которые могут быть оформлены в одном из текстовых форматов HTML, SVG, WML, или XML, и JSP элементы, которые конструируют динамическое содержимое. Кроме этого могут использоваться библиотеки JSP тегов, а также EL (Expression Language), для внедрения Java-кода в статичное содержимое JSP-страниц.

Реализует модель **MVC. M -** Bean**, V –** JSP-страница**, C -** Servlet**.**

Структура таких страниц может состоять из пяти конструкций:

1. HTML,
2. комментарии,
3. скриптовые элементы,
4. директивы
5. действия.

**Комментарии** в документе или программе служат к объяснению содержимого. **Скриптовые элементы** позволяют вам указать код на языке Java, который впоследствии станет частью конечного сервлета, **директивы** дадут вам возможность управлять всей структурой сервлета, а **действия** служат для задания существующих используемых компонентов, а также для контроля над поведением движка JSP. Для упрощения работы со скриптами имеются заранее определённые переменные, такие как *request, response, pageContext, session, out, application, config, page, exception.*

Спецификация JSP различает три типа скриптовых элементов:

* Декларации <%! одна или несколько деклараций %> - объекты, в котрых можно обращаться к выражениям и скриптлетам.
* Выражения <%= одно выражение %>
* Скриплеты <% скриплет %>

JSP страница может послать сообщение соответствующему контейнеру с указаниями какие действия необходимо провести. Эти сообщения называются директивами. Все директивы начинаются с <%@, затем следует название директивы и один или несколько атрибутов со значениями, и заканчиваются %>. Существует три основных типа директив:

* *page*, которая позволяет вам совершать такие операции, как импорт классов, изменение суперкласса сервлета, и т. п.;
* *include*, которая даёт вам возможность вставить файл в класс сервлета при трансляции JSP файла в сервлет; и
* *taglib*, позволяющий расширить множество тегов своими собственными, которые JSP контейнер способен истолковать.

# Директива page. Другие директивы.

**Директива JSP page**

Данная директива предоставляет атрибуты для JSP страницы. Атрибуты, определённые в этой директиве, внедряются в данную JSP страницу и на все её вложенные статические элементы, независимо от того были ли они вставлены с помощью директивы include или с помощью действия jsp:include***.*** *Кстати, Сакович наверняка захочет спросить, в чем разница этих двух форм записи*. *В 2-ом случ все подключается динамически, во 1-ом надо перекомпилировать*.

Атрибуты директивы page:

* ***import="пакет.class1, пакет.class2, ..., пакет.classN"****.* Позволяет вам задать пакеты, которые должны быть импортированы. Это единственный атрибут, который может использоваться несколько раз в одной директиве. В список необходимо включить все классы Java, которые Вы хотите использовать и которые не являются частью исходного набора импортируемых классов. Исходный набор содержит: java.lang.\*, javax.servlet.\*, javax.servlet.jsp.\* и javax.servlet.http.\*. Пример использования атрибута import:

*<%@ page import="java.util.Date, javax.text.SimpleDateFormat, com.myclasses.\*" %>*

* ***language="java".*** Данный атрибут предназначен для задания используемого языка программирования. По умолчанию принимается значение "java". Этот атрибут необязательно использовать, но тем не менее проблема потом может наступить в случае, если поставщик JSP контейнера будет использовать другие языки (например, JavaScript). Пример записи данного атрибута следующий:

*<%@ page language=”java” %>*

* ***extends="пакет.class".*** Задаёт суперкласс (родительский класс) для генерируемого сервлета. Обычно сервлет возникает расширением исходного класса. Опытные программисты могут с помощью этого атрибута создавать собственные суперклассы. Пример использования этого атрибута может выглядеть следующим образом:

*<%@ page extends=”mojPaket.PrimerHttp” %>*

* ***session="true|false".*** Данный атрибут может принимать значение true или false, которые определяют принимает ли участие JSP страница в трансляции HTTP. Значение true («истина», принимается по умолчанию) сигнализирует о том, что заранее определённая переменная session (тип HttpSession) должна быть привязана к существующей сессии, если таковая имеется, в противном случае создаётся новая сессия, к которой и осуществляется привязка. Значение false («ложь») определяет, что сессии не будут использоваться, и попытки обращения к переменной session приведут к возникновению ошибки при трансляции JSP страницы в сервлет. Пример использования данного атрибута может выглядеть следующим образом:

*<%@ page session=”false” %>*

* ***buffer="размерkb|none"***. Данный атрибут задаёт объём буферной памяти, необходимой для объекта JspWriter, на который ссылается заранее определённая переменная out. Значение, принимаемое по умолчанию, зависит от настроек сервера, но должно превышать 8kb. Значение задаётся либо в форме «размерkb» либо «none». Если Вы укажете значение буферной памяти как none, то сервлет не будет ничего сохранять в буферную память и передаст результат, записанный до переменной out, прямо объекту PrintWriter, который предоставляется вместе с объектом ServletResponse. Если Вы зададите значение буферной памяти на конкретную величину, то JspWriter будет в эту память сохранять данные, в результате чего увеличится производительность. В отличие от объекта PrintWriter объект JspWriter может вызывать исключительные ситуации IOExceptions. Исходная величина буферной памяти равна 8kB. Пример записи данного атрибута может выглядеть следующим образом:

*<%@ page buffer=”12kb” %>*

* ***autoflush="true|false".*** Данный атрибут может принимать значения true или false. Значение true («истина», принимаемое по умолчанию) устанавливает, что при переполнении буферной памяти он автоматически очистится. Значение false («ложь»), которое крайне редко используется, устанавливает что переполнение буфера должно приводить к возникновению исключительной ситуации (IOExceptions). Обычно атрибуты buffer и autoflush устанавливаются вместе в одной директиве. При установке значение атрибута buffer="none" установка значения false для атрибута autoflush недопустима. Пример может выглядеть следующим образом:

*<%@ page buffer=”16kb” autoflush=”true” %>*

* ***isThreadSafe="true|false".*** Данный атрибут может принимать значения true или false. Значение true («истина», принимается по умолчанию) задаёт нормальный режим выполнения сервлета, когда множественные запросы обрабатываются одновременно с использованием одного экземпляра сервлета, исходя из соображения что автор синхронизировал доступ к переменным этого экземпляра. Значение false («ложь») сигнализирует о том, что сервлет должен наследовать SingleThreadModel (однопоточную модель), при которой последовательные или одновременные запросы обрабатываются отдельными экземплярами сервлета. Другими словами значение true приведёт к тому, что контейнер может посылать сервлету сразу несколько запросов, в то время как при значении false контейнер посылает запросы по одному. Пример использования выглядит так:

*<%@ page isThreadSafe=”false” %>*

* ***info="информация".*** Задаёт строку, которая может быть получена при использовании метода Servlet.getServletInfo(). Обычно этот метод возвращает информацию о сервлете (например, авторе, версии и авторских правах). Пример записи данного атрибута может выглядеть следующим образом:

*<%@ page info=”Автор: Пётр Иванович; версия:1.0” %>*

* ***errorPage="url"***. Задаёт JSP страницу, которая вызывается в случае возникновения каких-либо событий Throwables, которые не обрабатываются на данной странице. Если на на JSP странице случится исключение, и JSP страница не имеет собственного кода для решения этого исключения, то контейнер автоматически передаст управление на URL, которую Вы задали в качестве значения атрибута errorPage. Пример записи выглядит так:

*<%@ page errorPage=“/myweb/errors/myerror.jsp“ %>*

* ***isErrorPage="true|false".*** Данный атрибут может принимать значения true или false. Сигнализирует о том, может ли эта страница использоваться для обработки ошибок для других JSP страниц или нет. По умолчанию принимается значение false («ложь»). Пример использования данного атрибута может выглядеть следующим образом:

*<%@ page isErrorPage=”true” %>*

* ***contentType="MIME-Тип".*** Данный атрибут задаёт тип MIME для вывода и по желанию можно задать кодировку знаков в ответе (HTML ответе). По умолчанию в качестве значения MIME используется text/html. Для наглядности можем использовать следующий пример:

*<%@ page contentType="text/plain" %>*

Того же результата можно добиться и использованием скриплета:

*<% response.setContentType("text/plain"); %>*

**Директива JSP include**

Эта директива позволяет Вам включать файлы в сервлет в процессе трансляции JSP страницы. Использование директивы выглядит следующим образом:

*<%@ include file="относительный url" %>*

Заданный URL обычно интерпретируется относительно JSP страницы, на которой расположена ссылка, но, как и при использовании любых других относительных URL вы можете задать системе положение интересующего вас ресурса относительно домашнего каталога Web сервера, добавив в начало URL символ «/». Содержимое подключаемого файла обрабатывается как обычный текст JSP и поэтому может включать такие элементы как статический HTML, элементы скриптов, директивы и действия. Например, многие сайты используют небольшую панель навигации на каждой странице. В связи с проблемами использования фреймов HTML часто эта задача решается размещением небольшой таблицы сверху или в левой половине страницы, HTML код которой многократно повторяется для каждой страницы сайта. Директива include наиболее естественный способ решения этой задачи, избавляющий разработчика от кошмара рутины копирования HTML в каждый отдельный файл.

**Директива JSP taglib**

Как уже известно, в JSP страницах элементы записываются с помощью знаков (условных знаков, меток, марок). Множество знаков, которые JSP контейнер способен истолковать, можно расширить с помощью так называемых библиотек знаков. Также к расширенному множеству знаков можно присоединять действия, вследствие чего происходит расширение самого языка JSP. Действия можно разделить на стандартные и собственные. Обобщённый способ записи может выглядеть так:

*<%@ taglib uri=”URI к библиотеке знаков” prefix=“префикс знака“ %>*

Библиотеку знаков необходимо идентифицировать с помощью URI адреса (уникального идентификатора ресурса). URI может быть как абсолютным так и отностительным. Уникальный идентификатор ресурса определяет расположение библиотеки знаков (TLD), который определяет собственные знаки этой библиотеки. Пример записи директивы:

<%@ taglib uri=”http://www.moywebserver.ru/naydiznaki.tld” prefix=”iskat” %>

JSP страница может содержать бесконечное кол-во директив taglib, но для каждой директивы необходимо установить различные префиксы, которые определяют содержимое библиотеки на странице. В качестве префикса можете использовать какой угодно текст, слово. В то время, как директиву taglib можно использовать где угодно на JSP странице, все собственные знаки, которые эти директивы используют, должны использоваться за ними.

**А теперь, все то же самое, но в 100500 раз короче и в коде:**

<%@ page атрибут=“значение“ %>

language=”java”

import="java.util.Date”

extends=”mojPaket.PrimerHttp” – задает суперкласс сервлетаsession="true|false"

buffer="размерkb|none"

autoflush="true|false"

isThreadSafe="true|false"

info="информация"

errorPage="url"

contentType="MIME-Тип"

<jsp:forward page="/utils/errorReporter.jsp" />

<jsp:forward page="<%= какое-нибудьВыражениеНаJava %>" />

<%@ include file="относительный url"%>

<%@ taglib uri=”**URI** к библиотеке знаков” prefix=“префикс знака“ %>

# Включение файлов в JSP-документы. Интеграция сервлетов и JSP. Перенаправление запросов.

<jsp:include page="относительный URL" flush="true" /> - Подключает файл в момент запроса страницы;

<%@ include file="относительный url"%> - в процессе трансляции.

<jsp:plugin

type="bean|applet"

code="файл класса"

codebase="объект CodeBase"

align="расположение"

archive="список архивов"

height="высота"

hspace="горизонтальный простор"

jreversion="версия"

name="название компонента"

vspace="вертикальный простор"

width="ширина"

nspluginurl="url"

iepluginurl="url">

<jsp:params>

<jsp:param name="название1" value="значение1" />

<jsp:param name="название2" value="значение2" />

...

<jsp:param name="названиеN" value="значениеN" />

</jsp:params>

<jsp:fallback></jsp:fallback>

</jsp:plugin> - для апплетов.

<jsp:plugin type="applet" code="Blink.class" width=300 height=100>

<jsp:params>

<jsp:param name=lbl value="Да это вкуснее, чем просто нарезанный хлеб!" />

<jsp:param name=speed value="4" />

<jsp:params>

<jsp:fallback>Ваш браузер по непонятным причинам не может запустить этот аплет </fallback>

</jsp:plugin>

На клиентской стороне не <applet>, a <object>

В приложении сложной структуры сервлет может выполнять предварительную обработку запроса, устанавливать компоненты bean, a затем перенаправлять запрос одному из документов JSP.

Для перенаправления запросов или включения внешнего содержимого используется класс RequestDispatcher. Для получения объекта RequestDispatcher используется метод getRequestDispatcher класса ServletContext, которому передается URL относительно корневого каталога сервера. Получив RequestDispatcher, вы можете использовать метод forward() для перенаправления запроса по указанному URL либо метод include() для вывода содержимого указанного документа. В обоих случаях методу необходимо передавать в качестве параметров объекты HttpServletRequest и HttpServletResponse.

# JavaBeans в JSP.

Это действие позволяет загружать [JavaBean](https://ru.wikipedia.org/wiki/JavaBean) для последующего использования на JSP странице. Эта возможность позволяет многократно использовать классы Java, не отказываясь при этом от преимуществ, предоставляемых сервлетами JSP. Помимо того, это один из способов исключения большей части обработки Java из JSP страницы. Если перенести обработку Java из JSP страницы до JavaBean, то потом эти функции можно использовать и в остальных JSP страницах. Простейший синтаксис для указания используемого bean:

<jsp:useBean id="имя"**class**="пакет.class" />

...

<jsp:setProperty name="myName"

property="someProperty" ... />

<jsp:useBean id="myName" ... >

...

<jsp:setProperty name="myName" property="someProperty" ... /> get-возвращает значения соответствующего свойства

</jsp:useBean>

Данные берутся из запроса

Property = “\*” – все установится в параметры запросы

Параметры: id, class, scope, type, и beanName.

* ***id*** — дает имя переменной, которая ссылается на bean. Если удается найти bean с теми же самыми значениями id и scope, то вместо создания нового экземпляра используется ранее созданный объект;
* ***class*** — задает полное имя пакета bean;
* ***scope*** — задает область, в которой bean должен быть доступен. Может принимать четыре допустимых значения: **page**, **request**, **session** и **application**. По умолчанию принимает значение *page*, означающее что bean доступен только на текущей странице (размещается в PageContext текущей страницы). Значение *request* означает что bean доступен только для текущего запроса клиента (размещается в объекте ServletRequest). Значение *session* означает что объект доступен всем страницам на протяжении жизни текущей HttpSession. И, наконец, значение *application* означает что он доступен всем страницам, использующим тот же самый ServletContext. Причина необходимости этого атрибута заключается в том что jsp:useBean приводит к созданию нового экземпляра объекта в том случае, если нет уже существующего объекта с тем же id и scope. Иначе используется уже существующий объект, и все элементы jsp:setParameter или любые другие между тегами jsp:useBean игнорируются.
* ***type*** — указывает тип переменной, которая ссылается на объект. Должен совпадать с именем класса, суперкласса или реализуемого интерфейса. Запомните что имя переменной задается через атрибут id.
* ***beanName*** — дает имя bean, которое будет использовано методом instantiate(). Можно задать type и beanName, и опустить атрибут class.

***ограничения на bean клас*** – private, get/set, public конструктор по умолчанию, пакеты. исп jsp.getProperty/setProperty name=”” property=”” value=””– вызывается get/set

# Определение новых JSP-дескрипторов. Обработка тела JSP-дескриптора.

# Определение новых JSP-дескрипторов. Вложенные JSP-дескрипторы.

Tag parent = (Tag)*findAncestorWithClass*(**this**, LetterOuterTag.**class**);

**Для создания нужно:**

* 1. класс поддержки дескриптора
  2. файл описания
  3. jsp-файл, использующий дескрипторы

**Класс поддержки дескриптора**

должен implements javax.servlet.jsp.tagext.Tag

какправило, extends TagSupport либоBodyTagSupport

метод doStartTag генерирует открывающие теги,

метод doEndTag генерирует закрывающие теги,

Для обработки тела метод doAfterBody

Для обнаружения внешнего findAncestorWithClass, параметры this и Class искомого

для тегов без атрибутов метод doStartTag

используется JspWriter

если нет тела – SKIP\_BODY

если есть атрибуты – есть соответствующие setter

Как и в обычных тегах, между открывающим и закрывающим пользовательскими тегами может находиться тело тега, или **body**. Пользовательские теги могут использовать содержимое элемента **body**–**content**. На данный момент поддерживаются следующие значения для **body–content**:

· **empty** – пустое тело;

· **jsp** – тело состоит из всего того, что может находиться в JSP-файле. Используется для расширения функциональности JSP-страницы;

· **tagdependent** – тело интерпретируется классом, реализующим данный тег. Используется в очень частных случаях.

Когда разрабатывается пользовательский тег с телом, то лучше наследовать класс тега от класса **BodyTagSupport**, реализующего в свою очередь интерфейс **BodyTag**. Кроме методов класса **TagSupport** (суперкласс для **BodyTagSupport**), он имеет методы, среди которых следует выделить:

**void doInitBody()** – вызывается один раз перед первой обработкой тела, после вызова метода **doStartTag()** и перед вызовом **doAfterBody()**;

**int doAfterBody()** – вызывается после каждой обработки тела. Если вернуть в нем константу **EVAL\_BODY\_AGAIN**, то **doAfterBody()** будет вызван еще раз. Если**SKIP\_BODY**, то обработка тела будет завершена;

**int doEndTag()** – вызывается один раз, когда отработаны все остальные методы.

Для того чтобы тело было обработано, метод **doStartTag()** должен вернуть**EVAL\_BODY\_INCLUDE** или **EVAL\_BODY\_BUFFERED**; если будет возвращено**SKIP\_BODY**, то метод **doInitBody()** не вызывается.

<**tlib-version**>1.0</**tlib-version**>  
<**short-name**>ctg</**short-name**>  
<**uri**>customtags</**uri**>  
  
<**tag**>  
 <**name**>letter</**name**>  
 <**tag-class**>by.talstaya.taskadd1.tag.LetterOuterTag</**tag-class**>  
 <**body-content**>scriptless</**body-content**>  
  
 <**attribute**>  
 <**name**>type</**name**>  
 <**required**>true</**required**>  
 <**rtexprvalue**>true</**rtexprvalue**> *<!--the attribute can support scriptlet values.--* </**attribute**>  
</**tag**>

# Библиотека JSTL. Обзор возможностей.

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

JavaServer Pages Standard Tag Library.

Она расширяет спецификацию JSP, добавляя библиотеку JSP тегов для общих нужд, таких как разбор XML данных, условная обработка, создание циклов и поддержка интернационализации.

Библиотека тегов JSTL состоит из четырёх групп тегов: основные теги – **core**, теги форматирования – **fomatting**, теги для работы с SQL – **sql**, теги для обработки XML –**xml**.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Area** | **Subfunction** | **Prefix** | | **Core** | Variable support | c | | Flow control | | URL management | | Miscellaneous | | **XML** | Core | x | | Flow control | | Transformation | | **I18N** | Locale | fmt | | Message formatting | | Number and date formatting | | **Database** | SQL | sql | | **Functions** | Collection length | fn | | String manipulation | | |  |  |  |  | | --- | --- | --- | --- | | **Area** | **Function** | **Tags** | **Prefix** | | **Core** | Variable support | remove  set | c | | Flow control | choose  when  otherwise  forEach  forTokens  if | | URL management | import  param  redirect  param  url  param | | Miscellaneous | catch  out | |

Теги общего назначения:

**<c:out />**– вычисляет и выводит значение выражения;

<**c:set />** – устанавливает переменную в указанную область видимости;

**<c:remove />** – удаляет переменную из указанной области видимости;

**<c:catch />** – перехватывает обработку исключения.

Теги условного перехода:

**<c:if />** – тело тега вычисляется только в том случае, если значение выражения **true**;

**<c:choose />** (**<c:when />**,**<c:otherwise />**) – то же что и **<c:if />** с поддержкой нескольких условий и действия, производимого по умолчанию.

Итераторы:

**<c:forEach />** – выполняет тело тега для каждого элемента коллекции;

**<c:forTokens />** – выполняет тело тега для каждой лексемы в строке.

Теги обработки URL:

**<c:redirect />** – перенаправляет запрос на указанный **URL**;

**<c:import/>** – добавляет на JSP содержимое указанного WEB-ресурса;

**<c:url />** – формирует адрес с учётом контекста приложения**request.getContextPath()**.

**<c:param />** – добавляет параметр к запросу, сформированному при помощи **<c:url/>**.

Для вызова JSP по относительному пути применяется метод forward(), для обращения к JSP по абсолютному пути используется метод sendRedirect(). Отличие этих методов состоит в том, что с методом forward() передается уже существующий объект запроса request, а при вызове метода sendRedirect() формируется новый запрос. Информацию в последнем случае следует передавать с другими объектами. К тому же метод forward() срабатывает быстрее

Forward:

* выполняется непосредственно сервлетом
* браузер абсолютно не в курсе, что происходит, и его исходный URL не меняется
* перезагрузка страницы в браузере инициирует запрос на оригинальный URL

Redirect:

* состоит из двух шагов, в которых Ваше приложение говорит браузеру получить контент с другого URL, отличного от оригинального URL
* перезагрузка страницы не инициирует запрос по оригинальному URL, а пойдет по URL из redirect
* немного медленнее, т.к. приходится делать 2 запроса вместо одного
* данные оригинального запроса (первого), будут недоступны второму запросу

Проще говоря, через forward вы можете вернуть контент с другого ресурса, другую jsp. При этом исходный URL не изменится.

1. Какие из 4-ёх jstl евских блоков (core, fmt, xml, sql) нежелательно использовать?   
Ответ: xml, sql - т.к нарушается паттерн MVC.