

Веб- программирование

Автор курса — Цопа
Е.А.
2024/25 уч. год

1. Введение

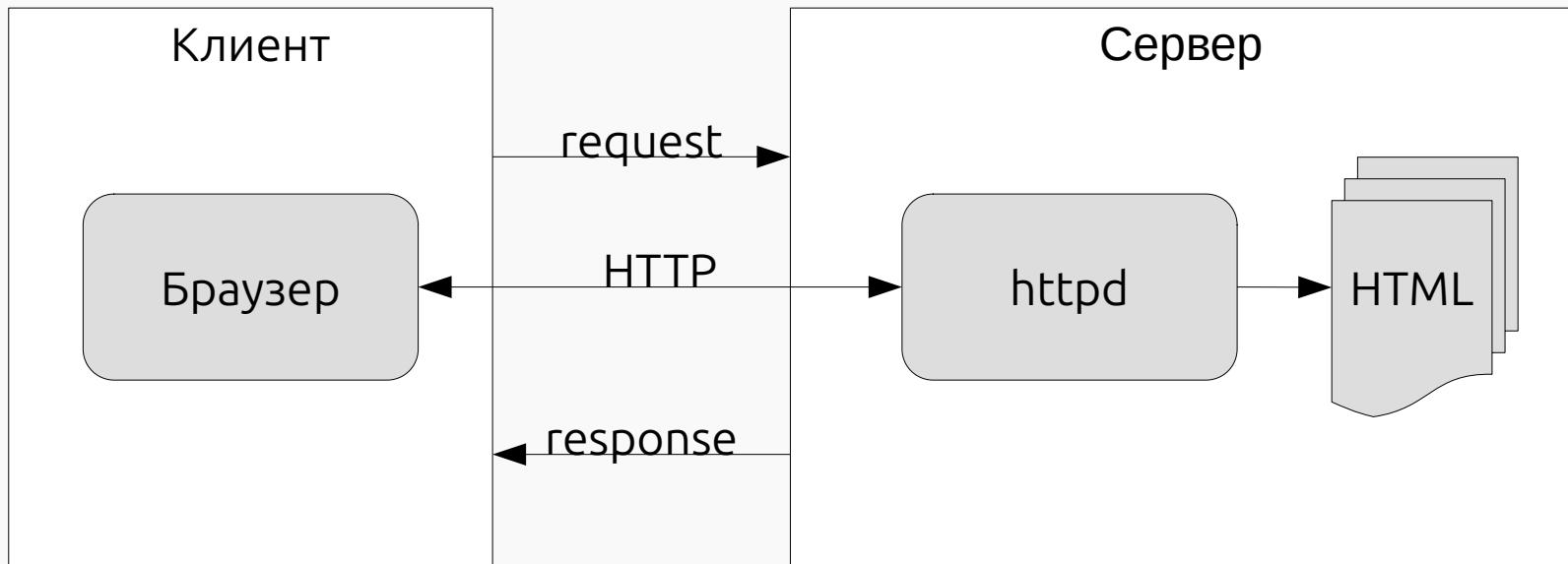
Мир веб-программирования

- Большая часть современных приложений взаимодействует с внешним миром через сеть Интернет.
- Даже локальные приложения часто пишутся по «канонам» веб-программирования.
- Практически все современные веб-сайты – полноценные информационные системы.

Архитектура интернет-приложений

- Много подходов и их реализаций — как на клиентской, так и на серверной стороне.
- Целесообразность подхода определяется сложностью разрабатываемого приложения и его областью использования.
- Независимо от выбранного подхода, «снизу» всё равно используются одни и те же стандарты и протоколы.

Сферическое интернет-приложение в вакууме



План курса (1)

Раздел		Технологии
1	Введение	
2	Общие стандарты и протоколы сети Интернет	
2.1	Клиент-серверное взаимодействие	HTTP
2.2	Разметка страниц	HTML
2.3	Стилизация страниц	CSS, SCSS
2.4	Динамические сценарии на стороне клиента	JavaScript
2.5	Асинхронное клиент-серверное взаимодействие	DHTML, Long Polling, WebSocket

План курса (2)

Раздел		Технологии
3	«Классические» интернет-приложения	
3.1	Серверные сценарии	CGI, FastCGI, Servlets
3.2	Шаблоны проектирования и архитектурные шаблоны	
3.3	Шаблонизация страниц	JSP, FreeMarker
4	Rich Internet Applications	Java Server Faces
5	Архитектура корпоративных приложений	Java / Jakarta EE
6	REST Backend & Frontend SPA	Туман войны

2. Общие стандарты и протоколы сети Интернет

Стандарты и протоколы сети Интернет

- «Основа основ»:
 - Hypertext Transfer Protocol (HTTP) — предназначен для передачи гипертекста между клиентом и сервером.
 - Hypertext Markup Language (HTML) — язык разметки гипертекста.
- «Дополнения» к HTML:
 - Cascade StyleSheets (CSS) – язык описания внешнего вида HTML-документа.
 - JavaScript – язык для написания динамических сценариев, выполняемых на стороне клиента.

2.1. Клиент-серверное взаимодействие. Протокол HTTP.

Протокол HTTP

- Протокол прикладного уровня
- Основа — технология «клиент-сервер»
- Может быть использован в качестве «транспорта» для других протоколов прикладного уровня
- Основной объект манипуляции — *ресурс*, на который указывает *URI*
- Обмен сообщениями идёт по схеме «запрос-ответ»
- Stateless-протокол (состояние не сохраняется). Для реализации сессий используются cookies.

URI, URL и URN

- URI (Uniform Resource Identifier) — уникальный идентификатор ресурса — символьная строка, позволяющая идентифицировать ресурс.
- URL (Uniform Resource Locator) — URI, позволяющий определить местонахождение ресурса.
- URN (Uniform Resource Name) — URI, содержащий единообразное имя ресурса (не указывает на его местонахождение).

URI, URL и URN (продолжение)

- URI:
`<схема>:<идентификатор - в зависимости от схемы>`
- URL:
`https://se.ifmo.ru/courses/web..../task.shtml`
`mailto:Joe.Bloggs@somedomain.com`
- URN:
`urn:isbn:5170224575`
`urn:sha1:YNCKHTQCWBTRNJIV4WNAE52SJUQCZ05C`

REST

- *Representational State Transfer* (передача состояния представления) – подход к архитектуре сетевых протоколов, обеспечивающих доступ к информационным ресурсам.
- Основные концепции:
 - Данные должны передаваться в виде небольшого числа стандартных форматов (HTML, XML, JSON).
 - Сетевой протокол должен поддерживать кэширование, не должен зависеть от сетевого слоя, не должен сохранять информацию о состоянии между парами «запрос-ответ».
- Антипод REST – подход, основанный на вызове удаленных процедур (*Remote Procedure Call – RPC*).

Структура запроса HTTP

- Стартовая строка:
Метод URI HTTP/Версия
GET /spip.html HTTP/1.1
- Заголовки:
Host: se.ifmo.ru
User-Agent: Mozilla/5.0 (X11; U; Linux i686; ru;
rv:1.9b5) Gecko/2008050509 Firefox/3.6
Accept: text/html
Connection: close
- Тело сообщения

Структура ответа HTTP

- Стартовая строка:
HTTP/Версия КодСостояния Пояснение
HTTP/1.1 200 Ok
- Заголовки:
Server: Apache/2.2.11 (Win32) PHP/5.3.0
Last-Modified: Sat, 16 Jan 2010 21:16:42 GMT
Content-Type: text/plain; charset=windows-1251
Content-Language: ru
- Тело сообщения

Методы HTTP

- OPTIONS — определение возможностей сервера.
- GET — запрос содержимого ресурса.
- HEAD — аналог GET, но в ответе отсутствует тело.
- POST — передача данных ресурсу.
- PUT — загрузка содержимого запроса на указанный URI.

Коды состояния

- Состоят из 3-х цифр.
- Первая цифра — *класс состояния*:
«1» — Informational — информационный;
«2» — Success — успешно;
«3» — Redirection — перенаправление;
«4» — Client error — ошибка клиента;
«5» — Server error — ошибка сервера.
- Примеры:
201 Webpage Created
403 Access allowed only for registered users
507 Insufficient Storage

Заголовки HTTP

- Формат:
ключ:значение
- 4 группы:
 - General Headers — могут включаться в любое сообщение клиента и сервера. Пример — Cache-Control.
 - Request Headers — используются только в запросах клиента. Пример — Referer.
 - Response Headers — используются только в запросах сервера. Пример — Allow.
 - Entity Headers — сопровождают любую сущность сообщения. Пример — Content-Language.

Примеры сообщений HTTP

- Запрос клиента:

```
GET /iaps/labs HTTP/1.1
Host: cs.ifmo.ru
User-Agent: Mozilla/5.0 (X11; U; Linux i686; ru;
rv:1.9b5)
Gecko/2008050509 Firefox/3.6.14
Accept: text/html
Connection: close
```

- Ответ сервера:

```
HTTP/1.0 200 OK
Date: Wed, 02 Mar 2011 11:11:11 GMT
Server: Apache
X-Powered-By: PHP/5.2.4-2ubuntu5wmp1
Last-Modified: Wed, 02 Mar 2011 11:11:11 GMT
Content-Language: ru
Content-Type: text/html; charset=utf-8
Content-Length: 1234
Connection: close
...HTML-код запрашиваемой страницы...
```

2.2. Разметка страниц. Язык HTML.

Что такое HTML

- Стандартный язык разметки документов в Интернете.
- Интерпретируется *браузером* и отображается в виде документа.
- Разработан в 1989-91 годах *Тимом Бернерсом-Ли*.
- Является частным случаем *SGML* (стандартного обобщённого языка разметки).
- Существует нотация *XHTML*, являющаяся частным случаем языка *XML*.

Браузеры

- *Браузер* — программа, отображающая HTML-документ в его отформатированном виде.
- Популярные браузеры:
 - Google Chrome
 - Mozilla Firefox
 - M\$ Internet Explorer / Edge
 - Apple Safari
 - Opera

Структура HTML-документа

- Документ состоит из *элементов*.
- Начало и конец элемента обозначаются *тегами*:
`текст`
- Теги могут быть пустыми:
`
`
- Теги могут иметь *атрибуты*:
`Здесь элемент содержит атрибут href.`
- Элементы могут быть вложенными:
`
 Этот текст будет полужирным,
 <i>а этот - ещё и курсивным</i>
`

Структура HTML-документа (продолжение)

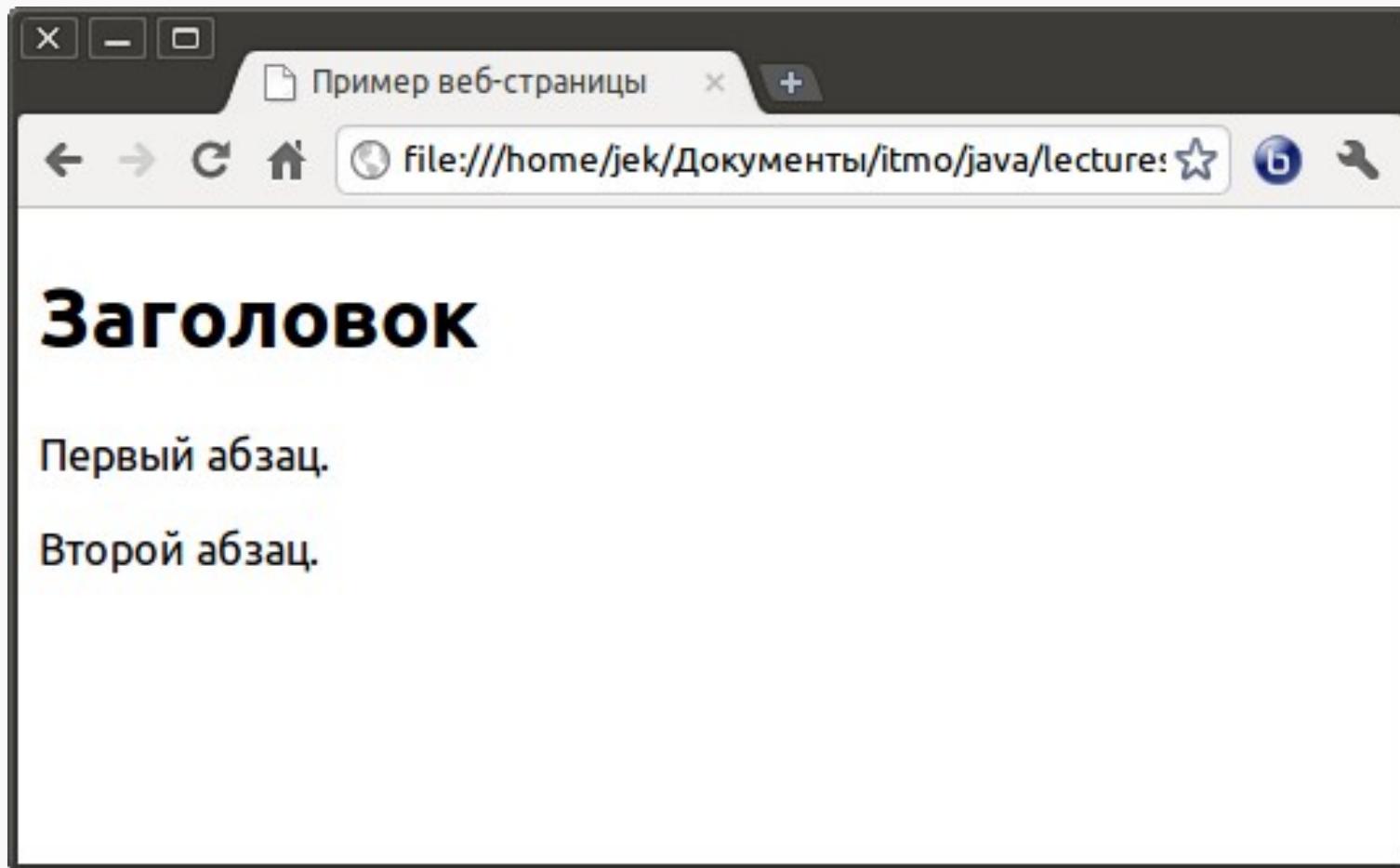
- Документ должен начинаться со строки объявления версии HTML:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML  
4.01//EN"  
"http://www.w3.org/TR/html4/strict.dtd">
```
- Начало и конец документа обозначаются тегами `<html>` и `</html>`.
- Внутри этих тегов должны находиться заголовок (`<head>...</head>`) и тело документа (`<body>...</body>`).

Пример HTML-документа

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"  
"http://www.w3.org/TR/html4/strict.dtd">  
<html>  
  <head>  
    <meta http-equiv="Content-Type"  
content="text/html;  
      charset=utf-8">  
    <title>Пример веб-страницы</title>  
  </head>  
  <body>  
    <h1>Заголовок</h1>  
    <!-- Комментарий -->  
    <p>Первый абзац.</p>  
    <p>Второй абзац.</p>  
  </body>  
</html>
```

Пример HTML-документа (продолжение)



HTML-формы

- Предназначены для обмена данными между пользователем и сервером.
- Документ может содержать любое число форм, но одновременно на сервер может быть отправлена только одна из них.
- Вложенные формы запрещены.
- Границы формы задаются тегами `<form>...</form>`.
- Метод HTTP задаётся атрибутом `method` тега `<form>`:
`<form method="GET" action="URL">...</form>`

Пример HTML-формы

```
<form method="POST" action="handler.php">

<p><b>Как по вашему мнению расшифровывается  
аббревиатура "ОС"?</b></p>

<p><input type="radio" name="answer"  
value="a1">Офицерский состав<br>

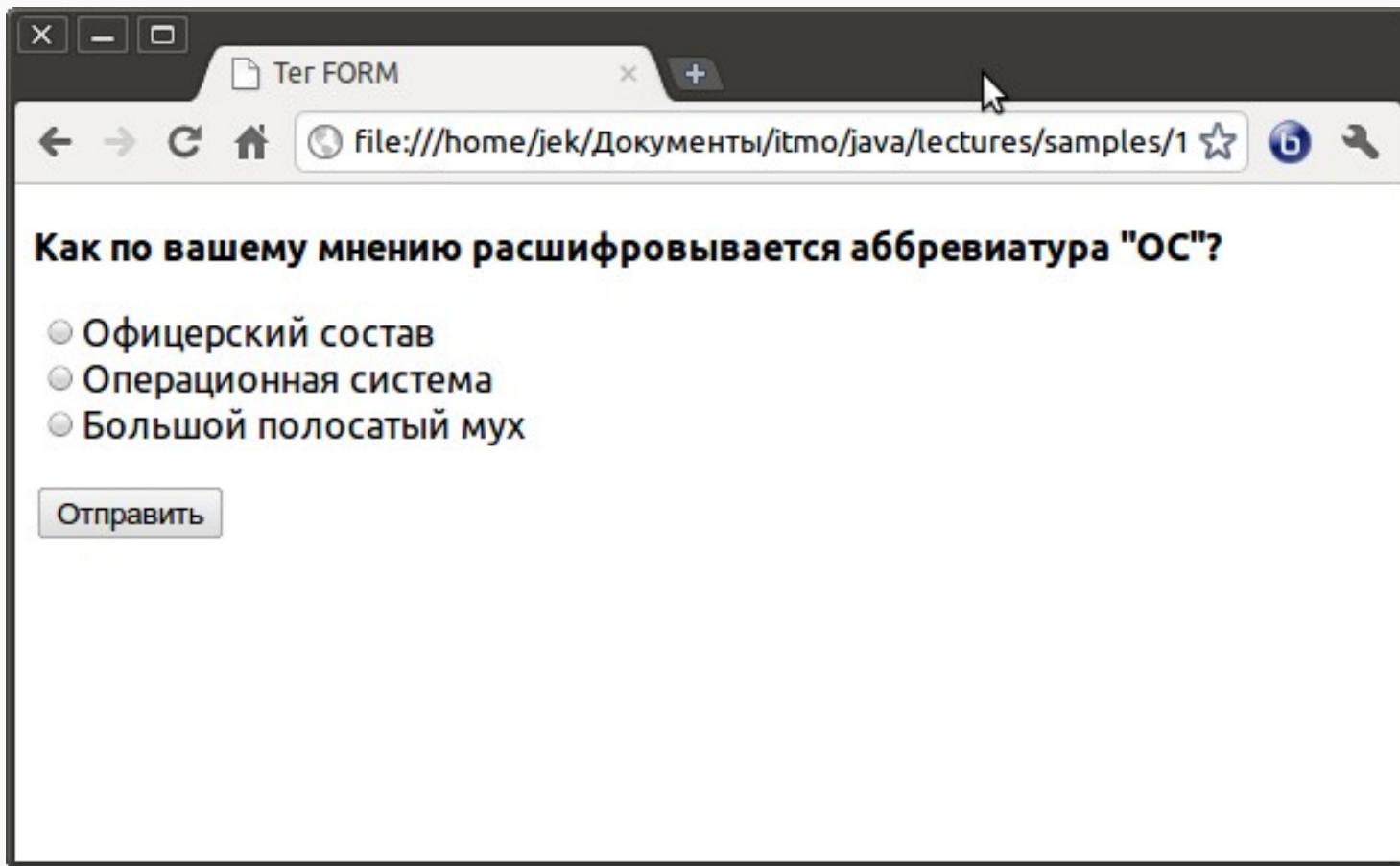
<input type="radio" name="answer"  
value="a2">Операционная система<br>

<input type="radio" name="answer"  
value="a3">Большой полосатый мух</p>

<p><input type="submit"></p>

</form>
```

Пример HTML-формы (продолжение)



The screenshot shows a web browser window with the title "Ter FORM". The address bar displays the URL "file:///home/jek/Документы/itmo/java/lectures/samples/1". The main content area contains the following text and form elements:

Как по вашему мнению расшифровывается аббревиатура "ОС"?

- Офицерский состав
- Операционная система
- Большой полосатый мух

Объектная модель документа (DOM)

- DOM — это платформо-независимый интерфейс, позволяющий программам и скриптам получить доступ к содержимому HTML-документов.
- Стандартизована W3C.
- Документ в DOM представляет собой *дерево узлов*.
- Узлы связаны между собой отношением «родитель-потомок».
- Используется для динамического изменения страниц HTML.

Объектная модель документа (DOM, продолжение)

специализация 220111
дисциплины документация студенты

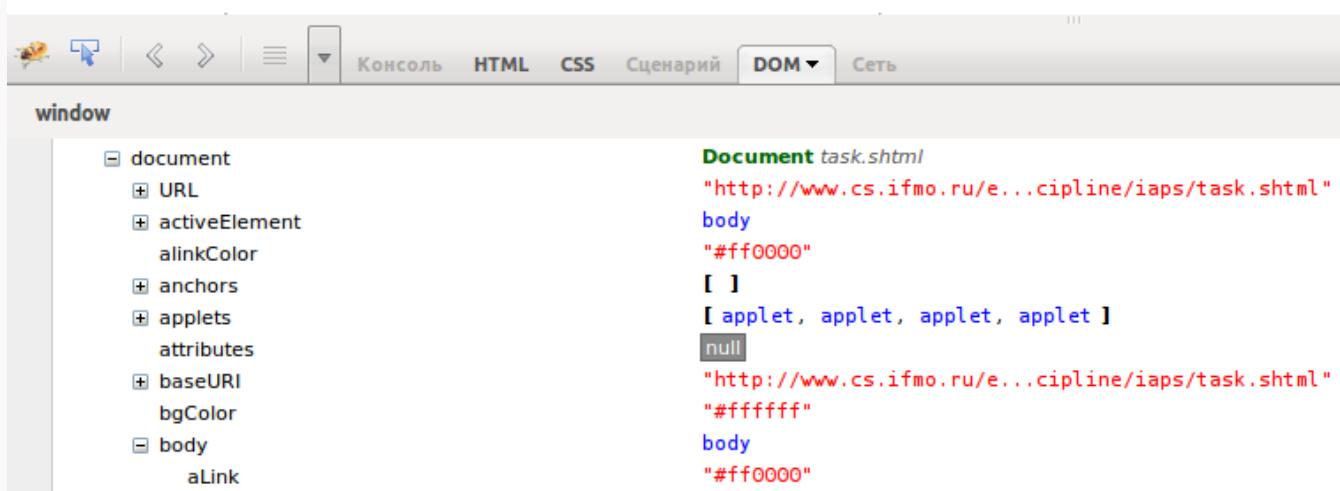
кафедра В рамках лабораторных работ по дисциплине "Системы программирования I" следующие задания:

карта

СПЕЦ-И

Лабораторная работа #1

На языке Java написать консольную программу, которая определяет, какие элементы массива A входят в заданную область S. Программа должна запрашивать у пользователя и выводить на экран координаты точек, входящих в область. Для координат и параметра R использовать типы данных с плавающей точкой. Для использовать стандартный поток ввода System.in.



The screenshot shows the DOM Inspector interface with the 'window' object selected. On the left, a tree view shows properties like document, URL, activeElement, anchors, applets, attributes, baseURI, bgColor, body, and aLink. On the right, the properties of the selected 'body' element are displayed in a code-like format:

```
Document task.shtml
"http://www.cs.ifmo.ru/e...cipline/iaps/task.shtml"
body
"#ff0000"
[ ]
[ applet, applet, applet, applet ]
null
"http://www.cs.ifmo.ru/e...cipline/iaps/task.shtml"
"#ffffff"
body
"#ff0000"
```

HTML5

- Пятая версия спецификации языка HTML.
- Стандарт принят в 2014 г. (HTML4 – в 1999 г.).
- Много новых синтаксических особенностей, в первую очередь – для более удобного управления мультимедийным содержимым страницы.



HTML5: изменения в синтаксисе

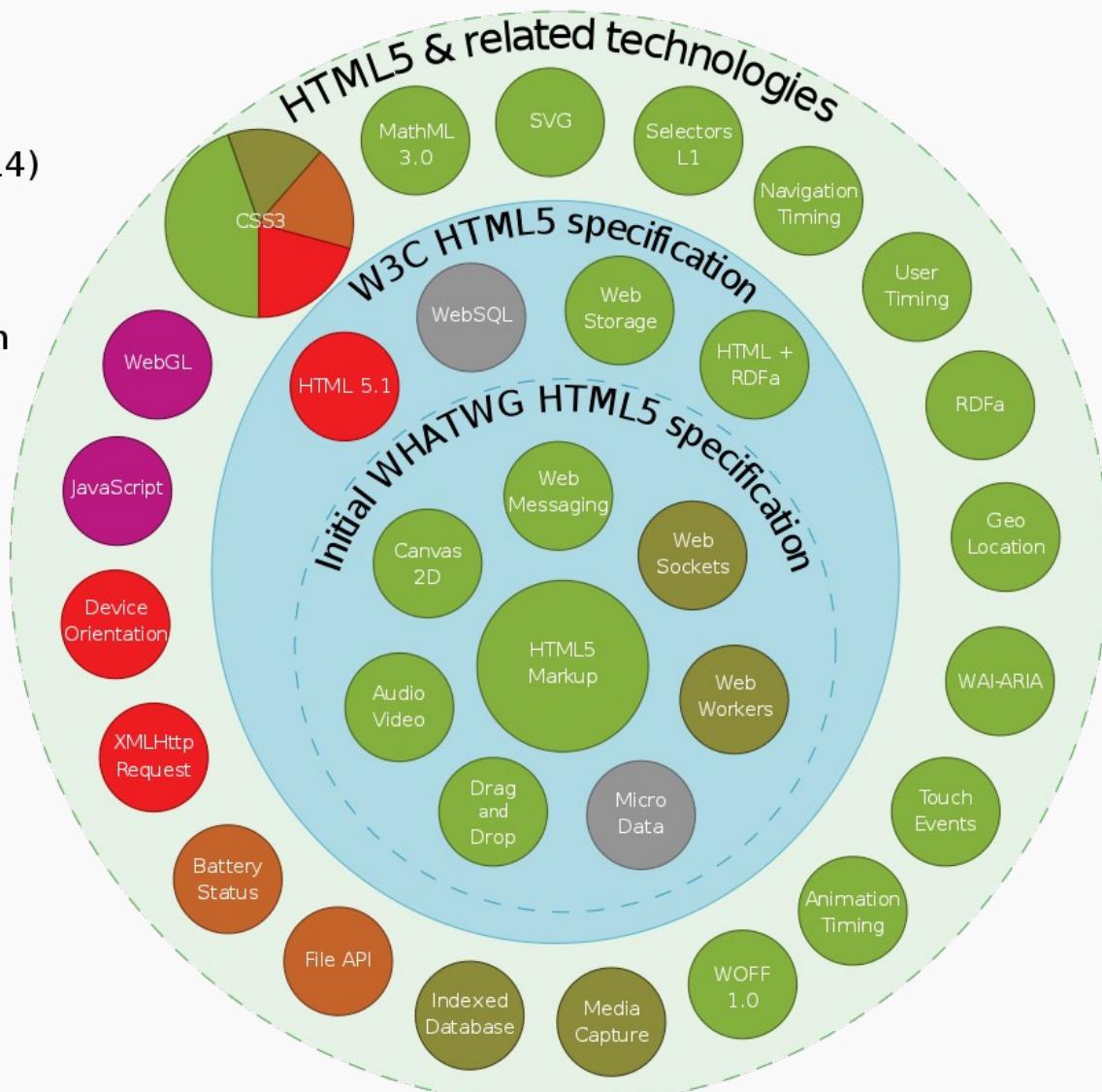
- Больше не базируется на SGML (хотя всё ещё обратно совместим с HTML4).
- Новая вводная строка `<!DOCTYPE html>`.
- Новые мультимедийные теги `<audio>` и `<video>`.
- Семантические замены для универсальных блочных (`<div>`) и строчных (``) элементов – `<nav>`, `<footer>` и т. д.
- Поддержка Web Forms 2.0 – новые поля ввода `date/time`, `email`, новые атрибуты и т. д.

HTML5: изменения в API

HTML5

Taxonomy & Status (October 2014)

-  Recommendation/Proposed
-  Candidate Recommendation
-  Last Call
-  Working Draft
-  Non-W3C Specifications
-  Deprecated or inactive



HTML5: пример страницы

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Img Width Attribute</title>
  </head>
  <body>
    
  </body>
</html>
```

HTML5: пример формы

```
<form>

    <label for="username">Username:</label>
    <input type="text" name="username" id="username" />
    <label for="password">Password:</label>
    <input type="password" name="password" id="password" />
    <input type="radio" name="gender" value="male" />Male<br />
    <input type="radio" name="gender" value="female" />Female<br />
    <input type="radio" name="gender" value="other" />Other
    <input list="Options" />
    <datalist id="Options">
        <option value="Option1"></option>
        <option value="Option2"></option>
        <option value="Option3"></option>
    </datalist>

    <input type="submit" value="Submit" />
    <input type="color" />
    <input type="checkbox" name="correct" value="correct" />Correct
</form>
```

2.3. Стилизация страниц. Основы CSS.

Что такое CSS

- CSS — технология описания внешнего вида документа, написанного языком разметки.
- Используется для задания цветов, шрифтов и других аспектов представления документа.
- Основная цель — разделение содержимого документа и его представления.
- Позволяет представлять один и тот же документ в различных методах вывода (например, обычная версия и версия для печати).

Источники CSS

- Авторские стили (информация стилей, предоставляемая автором страницы) в виде:
 - Inline-стилей — стиль элемента указывается в его атрибуте `style`.
 - Встроенных стилей — блоков CSS внутри самого HTML-документа.
 - Внешних таблиц стилей — отдельного файла `.css`.
- Пользовательские стили:
 - Локальный CSS-файл, указанный пользователем в настройках браузера, переопределяющий авторские стили.
- Стиль браузера:
 - Стандартный стиль, используемый браузером по умолчанию для представления элементов.

Структура CSS

- Таблица стилей состоит из набора *правил*.
- Каждое правило состоит из набора *селекторов* и блока *определений*:

```
селектор, селектор {  
    свойство: значение;  
    свойство: значение;  
    свойство: значение;  
}
```

- Пример:

```
div, td {  
    background-color: red;  
}
```

Приоритеты стилей

- Если к одному элементу «подходит» сразу несколько стилей, применён будет наиболее приоритетный.
- Приоритеты рассчитываются таким образом (от большего к меньшему):
 1. свойство задано при помощи `!important`;
 2. стиль прописан напрямую в теге;
 3. наличие идентификаторов (`#id`) в селекторе;
 4. количество классов (`.class`) и псевдоклассов (`:pseudoclass`) в селекторе;
 5. количество имён тегов в селекторе.
- Имеет значение относительный порядок расположения свойств — свойство, указанное позже, имеет приоритет.

Пример CSS

```
p {  
    font-family: "Garamond", serif;  
}  
  
h2 {  
    font-size: 110 %;  
    color: red;  
    background: white;  
}  
  
.note {  
    color: red;  
    background: yellow;  
    font-weight: bold;  
}  
  
p#paragraph1 {  
    margin: 0;  
}  
  
a:hover {  
    text-decoration: none;  
}  
  
#news p {  
    color: blue;  
}
```



Пример страницы с CSS

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-
8">
    <title>Заголовки</title>
    <style type="text/css">
      h1 { color: #a6780a; font-weight: normal; }
      h2 {
        color: olive;
        border-bottom: 2px solid black;
      }
    </style>
  </head>
  <body>
    <h1>Заголовок 1</h1>
    <h2>Заголовок 2</h2>
  </body>
</html>
```

LESS & Sass / SCSS

Языки стилей, позволяющие повысить уровень абстракции CSS-кода и упростить структуру таблиц стилей.

По сравнению с «обычным» CSS, имеются следующие особенности:

- Можно использовать переменные (константы и примеси).
- Можно использовать вложенные правила.
- Более мощные возможности по импорту, наследованию стилей.
- Поддержка математических операторов.

Браузеры могут не поддерживать LESS & Sass / SCSS-таблицы стилей — нужен специальный транслятор, который преобразует эти правила в «обычный» CSS.

LESS & Sass / SCSS (продолжение)

- LESS — CSS-like синтаксис:

```
.box-1 {  
    color: #BADA55;  
    .set-bg-color(#BADA55);  
}
```

- SASS — Ruby-like синтаксис:

```
.my-element  
    color= !primary-color  
    width= 100%  
    overflow= hidden
```

- SCSS — диалект SASS с CSS-like синтаксисом:

```
.my-element {  
    color: $primary-color;  
    width: 100%;  
    overflow: hidden;  
}
```



Константы

(SCSS)

```
//-- Font size -----  
$md-font-size-h1: 24px;  
$md-font-size-h2: 20px;  
$md-font-size-h3: 16px;  
$md-font-size-h4: 13px;  
$md-font-size-h5: 12px;  
$md-font-size-h6: 10px;  
  
//-- Line height -----  
$md-line-height-h1: 32px;  
$md-line-height-h2: 28px;  
$md-line-height-h3: 24px;  
$md-line-height-h4: 24px;  
$md-line-height-h5: 20px;  
$md-line-height-h6: 20px;  
  
//-- Font weight -----  
$md-font-weight-regular: 400;  
  
h1, h2, h3, h4, h5, h6 {  
    font-weight: $md-font-weight-regular;  
    margin: 0;  
}  
  
h1 {  
    font-size: $md-font-size-h1;  
    line-height: $md-line-height-h1;  
}
```

```
h2 {  
    font-size: $md-font-size-h2;  
    line-height: $md-line-height-h2;  
}  
  
h3 {  
    font-size: $md-font-size-h3;  
    line-height: $md-line-height-h3;  
}  
  
h4 {  
    font-size: $md-font-size-h4;  
    line-height: $md-line-height-h4;  
}  
  
h5 {  
    font-size: $md-font-size-h5;  
    line-height: $md-line-height-h5;  
}  
  
h6 {  
    font-size: $md-font-size-h6;  
    line-height: $md-line-height-h6;  
}
```

Примеси

Код на SCSS:

```
@mixin border-radius($radius,  
$border,$color) {  
    -webkit-border-radius: $radius;  
    -moz-border-radius: $radius;  
    -ms-border-radius: $radius;  
    border-radius: $radius;  
    border:$border solid $color  
}  
  
.box {  
    @include border-  
    radius(10px,1px,red);  
}
```

«Обычный» CSS:

```
.box {  
    -webkit-border-radius:  
    10px;  
    -moz-border-radius: 10px;  
    -ms-border-radius: 10px;  
    border-radius: 10px;  
    border: 1px solid red;  
}
```

Вложенные стили

Код на SCSS:

```
#header {  
    background: #FFFFFF;  
    .error {  
        color: #FF0000;  
    }  
    a {  
        text-decoration: none;  
        &:hover {  
            text-decoration: underline;  
        }  
    }  
}
```

Может быть преобразован в «обычный» CSS:

```
#header {  
    background: #FFFFFF;  
}  
#header .error {  
    color: #FF0000;  
}  
#header a {  
    text-decoration: none;  
}  
#header a:hover {  
    text-decoration: underline;  
}
```

Импорт стилей

```
// _reset.scss          // base.scss

html,                  @import 'reset';

body,                 body {
ul,                   font: 100% Helvetica, sans-
ol {                  serif;
margin: 0;           background-color: #efefef;
padding: 0;           }
}
```

Наследование

```
.message {  
    border: 1px solid  
#ccc;  
    padding: 10px;  
    color: #333;  
}  
  
.success {  
    @extend .message;  
    border-color: green;  
}  
  
.error {  
    @extend .message;  
    border-color: red;  
}  
  
.warning {  
    @extend .message;  
    border-color:  
yellow;  
}
```

Математика

```
.container { width: 100%; }

article[role="main"] {
    float: left;
    width: 600px / 960px * 100%;
}

aside[role="complementary"] {
    float: right;
    width: 300px / 960px * 100%;
}
```

Компиляция в CSS

Пример для Maven:

```
<!-- Sass compiler -->
<plugin>
    <groupId>org.jasig.maven</groupId>
    <artifactId>sass-maven-plugin</artifactId>
    <version>2.25</version>
    <executions>
        <execution>
            <phase>prepare-package</phase>
            <goals>
                <goal>update-stylesheets</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <resources>
            <resource>
                <!-- Set source and destination dirs -->
                <source>
                    <directory>${project.basedir}/src/main/webapp/sass</directory>
                </source>

                <destination>${project.basedir}/src/main/webapp/sass_compiled</destination>
            </resource>
        </resources>
    </configuration>
</plugin>
```

2.4. Динамические сценарии на стороне клиента. Язык JavaScript.

JavaScript и клиентские сценарии

- JavaScript — объектно-ориентированный скриптовый язык программирования.
- Используется для придания интерактивности веб-страницам.
- Основные архитектурные черты:
 - динамическая типизация;
 - слабая типизация;
 - автоматическое управление памятью;
 - прототипное программирование;
 - функции как объекты первого класса.

Особенности синтаксиса

- Все идентификаторы регистрозависимы.
- В названиях переменных можно использовать буквы, подчёркивание, символ доллара, арабские цифры.
- Названия переменных не могут начинаться с цифры,
- Для оформления однострочных комментариев используются //, многострочные и внутристрочные комментарии начинаются с /* и заканчиваются */.

Структура языка

- Ядро (ECMAScript);
- Объектная модель браузера (Browser Object Model);
- Объектная модель документа (Document Object Model).

Особенности ECMAScript

- Встраиваемый расширяемый не имеющий средств ввода/вывода язык программирования.
- 5 примитивных типов данных — Number, String, Boolean, Null и Undefined.
- Объектный тип данных — Object.
- 15 различных видов инструкций.

Особенности ECMAScript (продолжение)

Блок не ограничивает область видимости переменной:

```
function foo() {  
    var sum = 0;  
    for (var i = 0; i < 42; i += 2) {  
        var tmp = i + 2;  
        sum += i * tmp;  
    }  
    for (var i = 1; i < 42; i += 2) {  
        sum += i*i;  
    }  
    alert(tmp);  
    return sum;  
}  
foo();
```

Особенности ECMAScript (продолжение)

Если переменная объявляется вне функции, то она попадает в глобальную область видимости:

```
var a = 42;  
  
function foo() {  
    alert(a);  
}  
  
foo();
```

Особенности ECMAScript (продолжение)

Функция — это тоже объект:

```
// объявление функции
function sum(arg1, arg2) {
    return arg1 + arg2;
}

// задание функции с помощью инструкции
var sum2 = function(arg1, arg2) {
    return arg1 + arg2;
};

// задание функции с использованием
// объектной формы записи
var sum3 = new Function("arg1", "arg2",
    "return arg1 + arg2;");
```

Объектная модель браузера

- ВОМ — прослойка между ядром и DOM.
- Основное предназначение — управление окнами браузера и обеспечение их взаимодействия.
- Специфична для каждого браузера.
- Каждое из окон браузера представляется объектом `window`:

```
var contentsWindow;  
contentsWindow =
```

```
window.open("http://cs.ifmo.ru","contents")  
;
```

Объектная модель браузера (продолжение)

- Возможности ВОМ:
 - управление фреймами;
 - поддержка задержки в исполнении кода и зацикливания с задержкой;
 - системные диалоги;
 - управление адресом открытой страницы;
 - управление информацией о браузере;
 - управление информацией о параметрах монитора;
 - ограниченное управление историей просмотра страниц;
 - поддержка работы с HTTP cookie.

Объектная модель документа

- С помощью JavaScript можно производить следующие манипуляции:
 - получение узлов:
`document.all("image1").outerHTML;`
 - изменение узлов;
 - изменение связей между узлами;
 - удаление узлов.

Встраивание в веб-страницы

- Внутри страницы:

```
<script type="text/javascript">
    alert('Hello, World!');
</script>
```

- Внутри тега:

```
<a href="delete.php" onclick="return confirm('Вы
уверены?');" >Удалить</a>
```

- Отделение от разметки (используется DOM):

```
window.onload = function() {
    var linkWithAlert = document.getElementById("alertLink");
    linkWithAlert.onclick = function() {
        return confirm('Вы уверены?');
    };
};
...
<a href="delete.php" id="alertLink">Удалить</a>
```

- В отдельном файле:

```
<script type="text/javascript"
src="http://Путь_к_файлу_со_скриптом"></script>
```



ES6 / ES2015+



- ES6 — новая версия языка ECMAScript, выпущенная в 2015 г.
- Добавляет в синтаксис языка множество новых возможностей.
- Поддерживается практически всеми современными браузерами (хотя с этим всё ещё могут быть нюансы).
- Для работы в старых браузерах может потребоваться специальная программа — *транспилер (transpiler)*.



Поддержка ES6 браузерами

Block Scope Variables

- Два новых ключевых слова — let и const.
- ES5:

```
var x = 'outer';
function test(inner) {
  if (inner) {
    var x = 'inner'; // scope whole function
    return x;
  }
  return x; // gets redefined on line 4
}

test(false); // undefined
test(true); // inner
```

Ключевое слово let

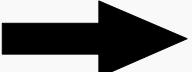
- Позволяет объявить переменную, областью видимости которой является блок.
- ES6:

```
let x = 'outer';
function test(inner) {
  if (inner) {
    let x = 'inner'; // scope whole function
    return x;
  }
  return x; // gets redefined on line 4
}

test(false); // outer
test(true); // inner
```

IIFE (Immediately Invoked Function Expression)

ES5:

```
{  
  var private = 1; //   
}  
  
// 1  
console.log(private);
```

ES5 & IIFE («костыли»):

```
(function(){  
  var private2 = 1;  
})();  
  
// Uncaught ReferenceError  
console.log(private2);
```

ES6:

```
{  
  let private3 = 1;  
}  
  
// Uncaught ReferenceError  
console.log(private3);
```

Ключевое слово const

Позволяет объявить константу:

```
// define MY_FAV as a constant and give it the value 7
const MY_FAV = 7;

// this will throw an error
MY_FAV = 20;

// will print 7
console.log('my favorite number is: ' + MY_FAV);

// trying to redeclare a constant throws an error
const MY_FAV = 20;

// the name MY_FAV is reserved for constant above,
// so this will fail too
var MY_FAV = 20;

// this throws an error too
let MY_FAV = 20;
```

Template Literals

ES5:

```
var first = 'Adrian';
var last = 'Mejia';
console.log('Your name is ' + first
+ ' ' + last + '.');
```

ES6:

```
const first = 'Adrian';
const last = 'Mejia';
console.log(`Your name is ${first} ${last}.`);
```

Деструктуризация

Деструктуризация (*destructuring assignment*) – особый синтаксис присваивания, при котором можно присвоить массив или объект сразу нескольким переменным, разбив его на части.

Пример — получение элемента из массива:

ES5:

```
var array = [1, 2, 3,  
4];
```

```
var first = array[0];  
var third = array[2];
```

```
console.log(first,  
third); // 1 3
```

ES6:

```
const array = [1, 2, 3,  
4];
```

```
const [first, ,third] =  
array;
```

```
console.log(first, third);  
// 1 3
```

Деструктуризация — обмен значениями

ES5:

```
var a = 1;  
var b = 2;
```

```
var tmp = a;  
a = b;  
b = tmp;
```

```
// 2 1  
console.log(a, b);
```

ES6:

```
let a = 1;  
let b = 2;
```

```
[a, b] = [b, a];  
console.log(a, b); // 2  
1
```

Деструктуризация нескольких возвращаемых значений

ES5:

```
function margin() {  
    var left=1, right=2, top=3, bottom=4;  
    return { left: left, right: right, top: top, bottom: bottom };  
}  
  
var data = margin();  
var left = data.left;  
var bottom = data.bottom;  
  
console.log(left, bottom); // 1 4
```

ES6:

```
function margin() {  
    const left=1, right=2, top=3, bottom=4;  
    return { left, right, top, bottom };  
}  
  
const { left, bottom } = margin();  
  
console.log(left, bottom); // 1 4
```

Деструктуризация и сопоставление параметров

ES5:

```
var user = {firstName: 'Adrian', lastName: 'Mejia'];

function getFullName(user) {
    var firstName = user.firstName;
    var lastName = user.lastName;

    return firstName + ' ' + lastName;
}

console.log(getFullName(user)); // Adrian Mejia
```

ES6:

```
const user = {firstName: 'Adrian', lastName: 'Mejia';

function getFullName({ firstName, lastName }) {
    return `${firstName} ${lastName}`;
}

console.log(getFullName(user)); // Adrian Mejia
```

Деструктуризация объекта

ES5:

```
function settings() {  
  return { display: { color: 'red' }, keyboard: { layout: 'qwert' } };  
}  
  
var tmp = settings();  
var displayColor = tmp.display.color;  
var keyboardLayout = tmp.keyboard.layout;  
  
console.log(displayColor, keyboardLayout); // red qwert
```

ES6:

```
function settings() {  
  return { display: { color: 'red' }, keyboard: { layout: 'qwert' } };  
}  
  
const { display: { color: displayColor }, keyboard: { layout: keyboardLayout } } = settings();  
  
console.log(displayColor, keyboardLayout); // red qwert
```

Классы и объекты

В ES6 появился новый синтаксис описания и инициализации объектов:

ES5:

```
var Animal = (function () {
    function MyConstructor(name) {
        this.name = name;
    }
    MyConstructor.prototype.speak =
        function speak() {
            console.log(this.name +
                ' makes a noise.');
    };
    return MyConstructor;
})();

var animal =
    new Animal('animal');
// animal makes a noise.
animal.speak();
```

ES6:

```
class Animal {
    constructor(name) {
        this.name = name;
    }
    speak() {
        console.log(this.name
            + ' makes a noise.');
    }
}

const animal =
    new Animal('animal');
// animal makes a noise.
animal.speak();
```

Наследование

Новые ключевые слова extends и super.

ES5:

```
var Lion = (function () {
  function MyConstructor(name){
    Animal.call(this, name);
  }
  // prototypal inheritance
  MyConstructor.prototype =
    Object.create(Animal.prototype);
  MyConstructor.prototype.constructor =
  Animal;
  MyConstructor.prototype.speak =
  function speak() {
    Animal.prototype.speak.call(this);
    console.log(this.name + ' roars');
  };
  return MyConstructor;
})();

var lion = new Lion('Simba');
lion.speak(); // Simba makes a noise.
// Simba roars.
```

ES6:

```
class Lion extends Animal {
  speak() {
    super.speak();
    console.log(this.name + ' roars');
  }
}

const lion = new Lion('Simba');
lion.speak(); // Simba makes a noise.
// Simba roars.
```

Промисы (promises) вместо коллбэков

ES5:

```
function printAfterTimeout(string,  
timeout, done){  
    setTimeout(function(){  
        done(string);  
    }, timeout);  
}  
  
printAfterTimeout('Hello ', 2e3,  
function(result){  
    console.log(result);  
  
    // nested callback  
    printAfterTimeout(result +  
        ' Reader', 2e3,  
        function(result){  
            console.log(result);  
        });  
});
```

ES6:

```
function printAfterTimeout(string,  
timeout){  
    return new Promise(  
        (resolve, reject) => {  
            setTimeout(function(){  
                resolve(string);  
            }, timeout);  
        });  
}  
  
printAfterTimeout('Hello ', 2e3)  
    .then((result) => {  
        console.log(result);  
        return printAfterTimeout(result  
            + ' Reader', 2e3);  
    }).then((result) => {  
        console.log(result);  
    });
```

Больше промисов

```
let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("готово!"), 1000)
});

promise
    .then((result) => console.log(`Успешный промис: ${result}`))
    .catch((result) => console.log(`\:|`));
```

С then-синтаксисом неудобно — есть async\await, по сути «синтаксический сахар».

```
async function f() {
    try {
        let result = await promise;
        console.log(`Успешный промис: ${result}`);
    } catch(err) {
        console.log(`\:|`);
    }
}
```

Стрелочные функции

ES5:

```
var _this = this; // need to hold a reference

$('.btn').click(function(event){
  _this.sendData(); // reference outer this
});

$('.input').on('change',function(event){
  this.sendData(); // reference outer this
}.bind(this)); // bind to outer this
```

ES6:

```
// this will reference the outer one
$('.btn').click((event) => this.sendData());

// implicit returns
const ids = [291, 288, 984];
const messages = ids.map(value => `ID is ${value}`);
```

Цикл с итератором

ES5:

```
// for
var array = ['a', 'b', 'c', 'd'];
for (var i = 0; i < array.length; i++) {
    var element = array[i];
    console.log(element);
}
```

```
// forEach
array.forEach(function (element) {
    console.log(element);
});
```

ES6:

```
// for ...of
const array = ['a', 'b', 'c', 'd'];
for (const element of array) {
    console.log(element);
}
```

Параметры по умолчанию

ES5:

```
function point(x, y, isFlag){  
    x = x || 0;  
    y = typeof(y) ===  
        'undefined' ? -1 : y;  
    isFlag =  
        typeof(isFlag) ===  
            'undefined' ?  
                true : isFlag;  
    console.log(x,y, isFlag);  
}
```

```
point(0, 0) // 0 0 true  
point(0, 0, false) // 0 0  
false  
point(1) // 1 -1 true  
point() // 0 -1 true
```

ES6:

```
function point(x = 0, y = -1,  
isFlag = true){  
    console.log(x,y,isFlag);  
}  
  
point(0, 0) // 0 0 true  
point(0, 0, false) // 0 0  
false  
point(1) // 1 -1 true  
point() // 0 -1 true
```

Rest-параметры

Аналог vararg в Java.

ES5:

```
function printf(format) {  
    var params = [].slice.call(arguments, 1);  
    console.log('params: ', params);  
    console.log('format: ', format);  
}  
  
printf('%s %d %.2f', 'adrian', 321, Math.PI);
```

ES6:

```
function printf(format, ...params) {  
    console.log('params: ', params);  
    console.log('format: ', format);  
}  
  
printf('%s %d %.2f', 'adrian', 321, Math.PI);
```

Операция spread

ES5:

```
Math.max.apply(Math,  
[2,100,1,6,43]) // 100
```

```
var array1 =  
  [2,100,1,6,43];  
var array2 =  
  ['a', 'b', 'c', 'd'];  
var array3 =  
  [false, true, null,  
   undefined];
```

```
console.log(array1  
  .concat(array2,  
array3));
```

ES6:

```
Math.max(...  
[2,100,1,6,43]) // 100
```

```
const array1 =  
  [2,100,1,6,43];  
const array2 =  
  ['a', 'b', 'c', 'd'];  
const array3 =  
  [false, true, null,  
   undefined];
```

```
console.log([...array1,  
            ...  
array2, ...array3]);
```

2.5. Асинхронное клиент-серверное взаимодействие. DHTML и AJAX

DHTML

- Dynamic HTML — способ создания интерактивного веб-сайта, использующий сочетание:
 - статичного языка разметки HTML;
 - выполняемого на стороне клиента скриптового языка JavaScript;
 - CSS (каскадных таблиц стилей);
 - DOM (объектной модели документа).

Пример страницы DHTML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"  
    "http://www.w3.org/TR/html4/strict.dtd">  
<html>  
  
<head>  
    <title>Заголовок страницы</title>  
    <script type="text/javascript">  
        window.onload= function () {  
            myObj = document.getElementById("navigation");  
            // .... какой-то код  
        }  
    </script>  
</head>  
  
<body>  
    <div id="navigation">  
    </div>  
</body>  
</html>
```

Что такое AJAX

- *AJAX (Asynchronous Javascript and XML)* — подход к построению интерактивных пользовательских интерфейсов веб-приложений.
- Основан на «фоновом» обмене данными браузера с веб-сервером.
- При обмене данными между клиентом и сервером веб-страница не перезагружается полностью.

Основные принципы AJAX

- Использование технологии динамического обращения к серверу «на лету», без перезагрузки всей страницы полностью, например:
 - с использованием XMLHttpRequest;
 - через динамическое создание дочерних фреймов;
 - через динамическое создание тега <script>.
- Использование DHTML для динамического изменения содержания страницы.

XMLHttpRequest

- XMLHttpRequest (XMLHttpRequest, XHR) — набор API, позволяющий осуществлять HTTP-запросы к серверу без необходимости перезагружать страницу.
- Данные можно пересылать в виде XML, JSON, HTML или просто неструктурированным текстом.
- При пересылке используется текстовый протокол HTTP и потому данные должны передаваться в виде текста.

XMLHttpRequest (пример)

```
var req;

function loadXMLDoc(url) {
    req = null;
    if (window.XMLHttpRequest) {
        try {
            req = new XMLHttpRequest();
        } catch (e){}
    } else if (window.ActiveXObject) {
        try {
            req = new ActiveXObject('Msxml2.XMLHTTP');
        } catch (e){
            try {
                req = new ActiveXObject('Microsoft.XMLHTTP');
            } catch (e){}
        }
    }

    if (req) {
        req.open("GET", url, true);
        req.onreadystatechange = processReqChange;
        req.send(null);
    }
}
```

XMLHttpRequest (пример, продолжение)

```
function processReqChange() {  
    try { // Важно!  
        // только при состоянии "complete"  
        if (req.readyState == 4) {  
            // для статуса "OK"  
            if (req.status == 200) {  
                // обработка ответа  
            } else {  
                alert("Не удалось получить данные:\n" +  
                    req.statusText);  
            }  
        }  
    }  
  
    catch( e ) {  
        // alert('Ошибка: ' + e.description);  
        // В связи с багом XMLHttpRequest в Firefox  
        // приходится отлавливать ошибку  
    }  
}
```

Преимущества и недостатки AJAX

- Преимущества:
 - экономия трафика;
 - уменьшение нагрузки на сервер;
 - ускорение реакции интерфейса;
- Недостатки:
 - динамически загружаемое содержимое недоступно поисковикам;
 - сложнее собирать метрики по использованию сайта;
 - усложнение проекта;
 - требуется включенный JavaScript в браузере.

Протокол WebSocket

- *WebSocket* — протокол полнодуплексной связи поверх TCP-соединения, предназначенный для обмена сообщениями между браузером и веб-сервером в режиме реального времени.
- Позволяет серверу отправлять данные браузеру без дополнительного запроса со стороны клиента.
- Обмен данными ведётся через отдельное TCP-соединение.
- Поддерживается всеми современными браузерами (даже IE).
- Альтернатива — AJAX + Long Polling.

Протокол WebSocket (продолжение)

```
<script>

var webSocket = new WebSocket('ws://localhost/echo');

webSocket.onopen = function(event) {
    alert('onopen');
    webSocket.send("Hello Web Socket!");
};

webSocket.onmessage = function(event) {
    alert('onmessage, ' + event.data);
    webSocket.close();
};

webSocket.onclose = function(event) {
    alert('onclose');
};

</script>
```

Библиотека jQuery

- JS-библиотека, предназначенная для разработки DHTML и AJAX-приложений.
- Упрощает доступ к элементам DOM с помощью кучи разных способов.
- Упрощает и унифицирует (для разных браузеров) реализацию AJAX.
- Упрощает добавление визуальных эффектов.
- Ключевым элементом API является функция (объект) \$ и её синоним jQuery.

AJAX на jQuery

Без jQuery (и без кросбраузерности):

```
req = new XMLHttpRequest();
req.open("POST", "some.php", true);
req.onreadystatechange = processReqChange;
req.send(null);
if (req.readyState == 4) {
    if (req.status == 200) {
        alert( "Data Saved" );
    }
}
```

С jQuery:

```
$.ajax({
    type: "POST",
    url: "some.php",
    data: {name: 'John', location: 'Boston'},
    success: function(msg){
        alert( "Data Saved: " + msg );
    }
});
```

jQuery: взаимодействие с DOM

```
$( "div.test" )
    .add( "p.quote" )
    .addClass( "blue" )
    .slideDown( "slow" );
```

```
$( "a" ).click(function() {
    alert("Hello world!");
});
```

```
$( "div.demo-container" )
    .html( "<p>All new content. </p>" );
```

jQuery: работа с событиями

```
$( "#foo" ).bind( "mouseenter mouseleave", function() {  
    $( this ).toggleClass( "entered" );  
});  
  
$( document ).ready(function() {  
    $( "#foo" ).bind( "click", function( event ) {  
        alert( "The mouse cursor is at (" +  
            event.pageX + ", " + event.pageY +  
            ")" );  
    });  
});  
  
$( "#other" ).click(function() {  
    $( ".target" ).change();  
});  
  
$( "#target" ).click(function() {  
    alert( "Handler for .click() called." );  
});
```

jQuery: эффекты и анимация

```
$( "#clickme" ).click(function() {
  $( "#book" ).animate({
    opacity: 0.25,
    left: "+=50",
    height: "toggle"
  }, 5000, function() {
    // Animation complete.
  });
});

$( "#foo" ).slideUp( 300 )
  .delay( 800 ).fadeIn( 400 );

$( "#clickme" ).click(function() {
  $( "#book" ).slideDown( "slow", function() {
    // Animation complete
  });
});
```

SuperAgent

API для реализации AJAX:

```
request
  .post('/api/pet')
  .send({ name: 'Manny', species: 'cat' })
  .set('X-API-Key', 'foobar')
  .set('Accept', 'application/json')
  .end(function(err, res){
    if (err || !res.ok) {
      alert('Oh no! Error');
    } else {
      alert('yay got ' +
JSON.stringify(res.body));
    }
  });
});
```

Fetch API

Современный стандарт на замену XMLHttpRequest для реализации AJAX. Поддерживается всеми современными браузерами.

- В отличие от XMLHttpRequest работает на «промисах»;
- Кросс-браузерность;
- Встроен в браузер => не нужно подключать отдельно => JQuery и SuperAgent «не нужны».

Бородатые браузеры не поддерживают => веб-сайт не будет работать.

Fetch API (пример)

```
async function getData() {  
    const url = "https://example.org/products.json";  
    try {  
        const response = await fetch(url);  
        if (!response.ok) {  
            throw new Error(`Response status: ${  
                response.status}`);  
        }  
  
        const json = await response.json();  
        console.log(json);  
    } catch (error) {  
        console.error(error.message);  
    }  
}
```

3. «Классические» интернет-приложения

Интернет-приложения

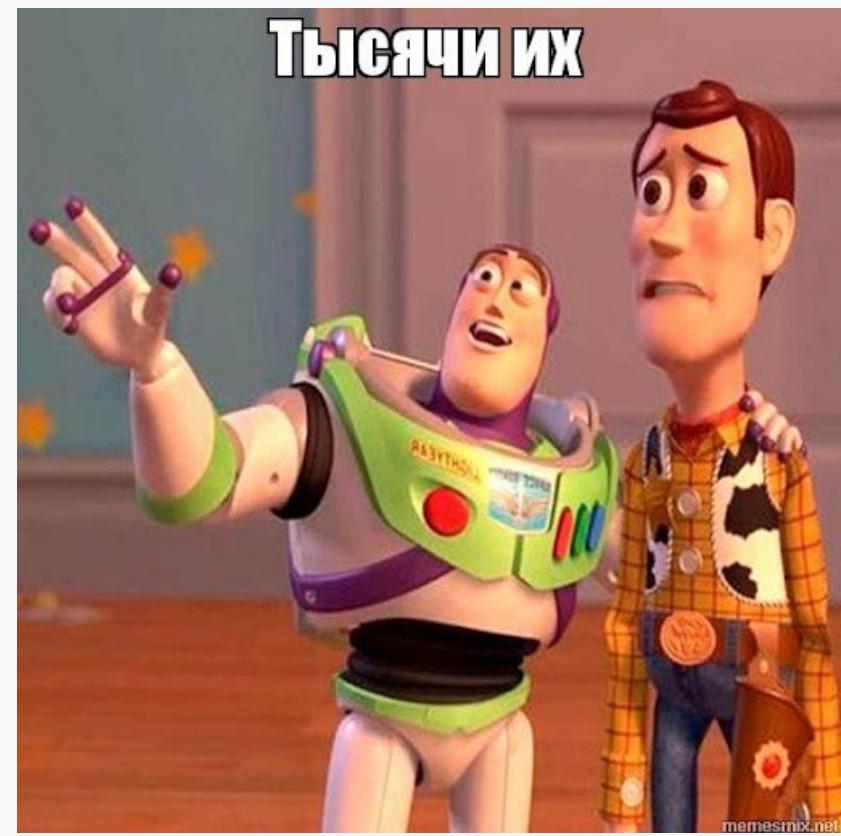
- Интернет-приложение — это сайт с той или иной динамической функциональностью на стороне сервера.
- Интернет-приложение осуществляет вызов программ на стороне сервера, к примеру:
 - Браузер отправляет на веб-сервер запрос на получение HTML-формы.
 - Веб-сервер формирует HTML-форму и возвращает её браузеру.
 - Браузер отправляет на сервер новый запрос с данными из HTML-формы.
 - Веб-сервер делегирует обработку данных из формы какой-либо программе на стороне сервера.

Особенности архитектуры

- Динамический контент формируют серверные сценарии.
- Разметка страниц задаётся с помощью шаблонов.
- Клиент-серверное взаимодействие реализуется либо «вручную», либо с помощью низкоуровневого фреймворка а-ля Fetch API.
- Активно используются архитектурные паттерны, за их реализацию отвечает сам программист.

Технологии для создания веб-приложений

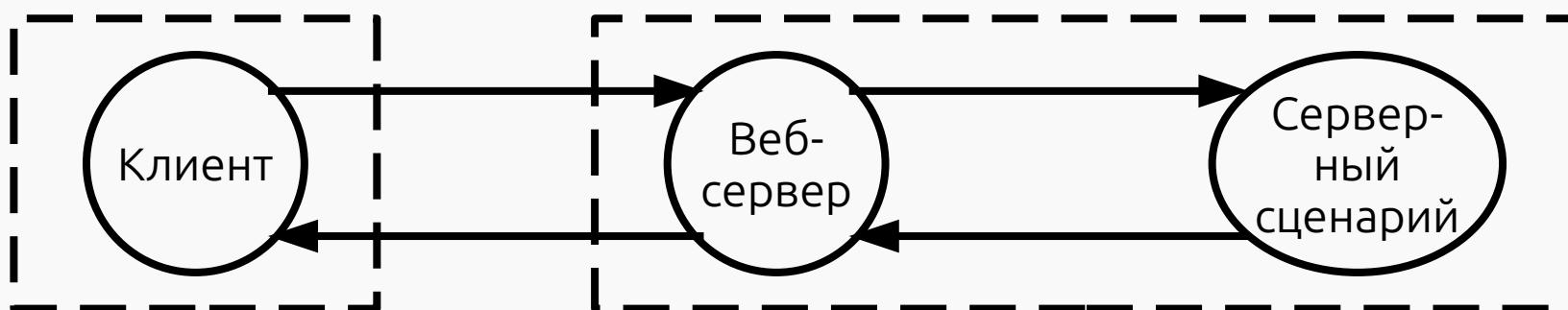
- HTML over HTTP
- Common Gateway Interface (CGI)
- FastCGI
- Servlets
- JavaServer Pages (JSP)
- JavaServer Faces



3.1. Серверные сценарии

Серверные сценарии

- Программы, вызываемые на сервере для формирования динамического контента.
- Веб-сервер «делегирует» им на обработку запрос и «транслирует» сформированный ими ответ клиенту.
- Могут использоваться вместе со «статикой», могут – сами по себе.



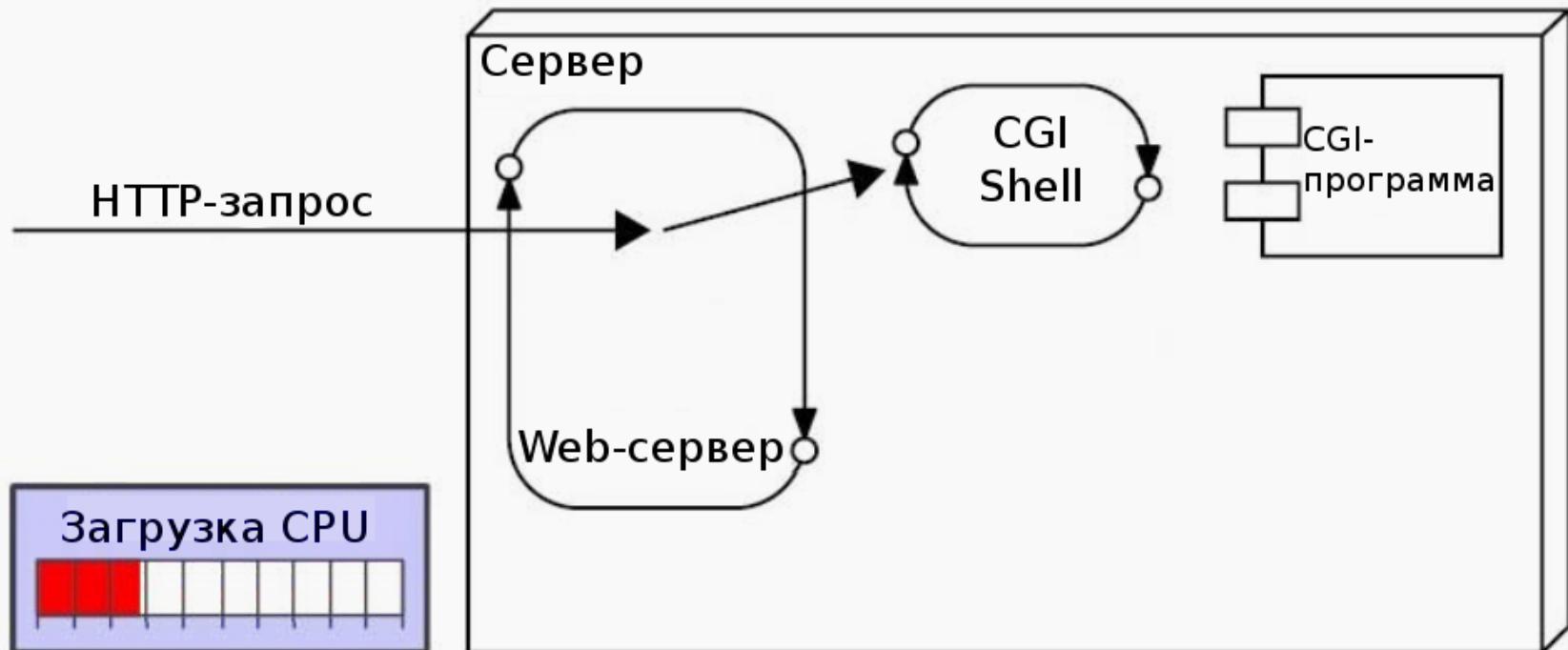
3.1.1. CGI & FastCGI

Common Gateway Interface

- CGI — простейший механизм вызова пользователем программ на стороне сервера.
- Данные отправляются программе посредством HTTP-запроса, формируемого веб-браузером.
- То, какая именно программа будет вызвана, обычно определяется URL запроса.
- Каждый запрос обрабатывается отдельным процессом CGI-программы.
- Взаимодействие программы с веб-сервером осуществляется через `stdin` и `stdout`.

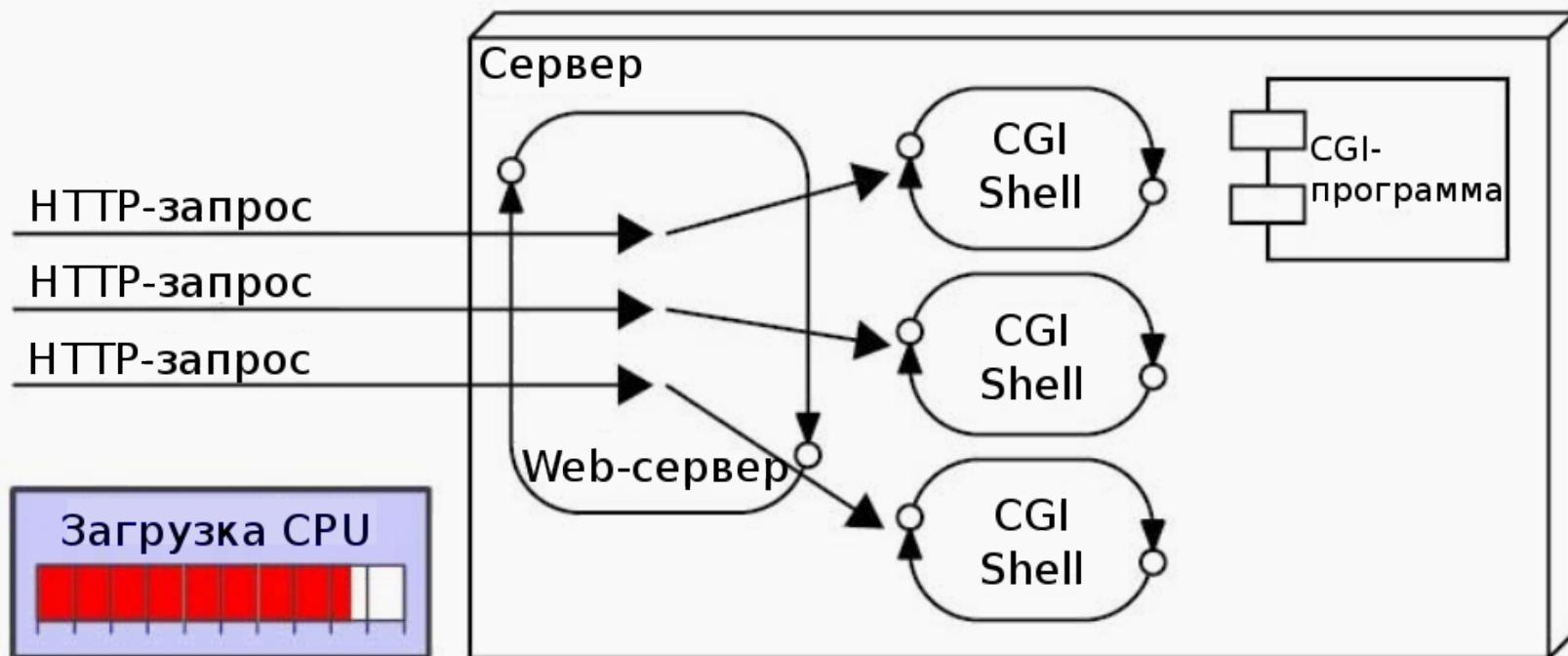
Выполнение CGI-сценариев

Один запрос:



Выполнение CGI-сценариев (продолжение)

Параллельная обработка нескольких запросов:



Пример реализации CGI-сценария

```
#include <stdio.h>

int main(void) {
    printf("Content-Type:
text/html;charset=UTF-8\n\n");

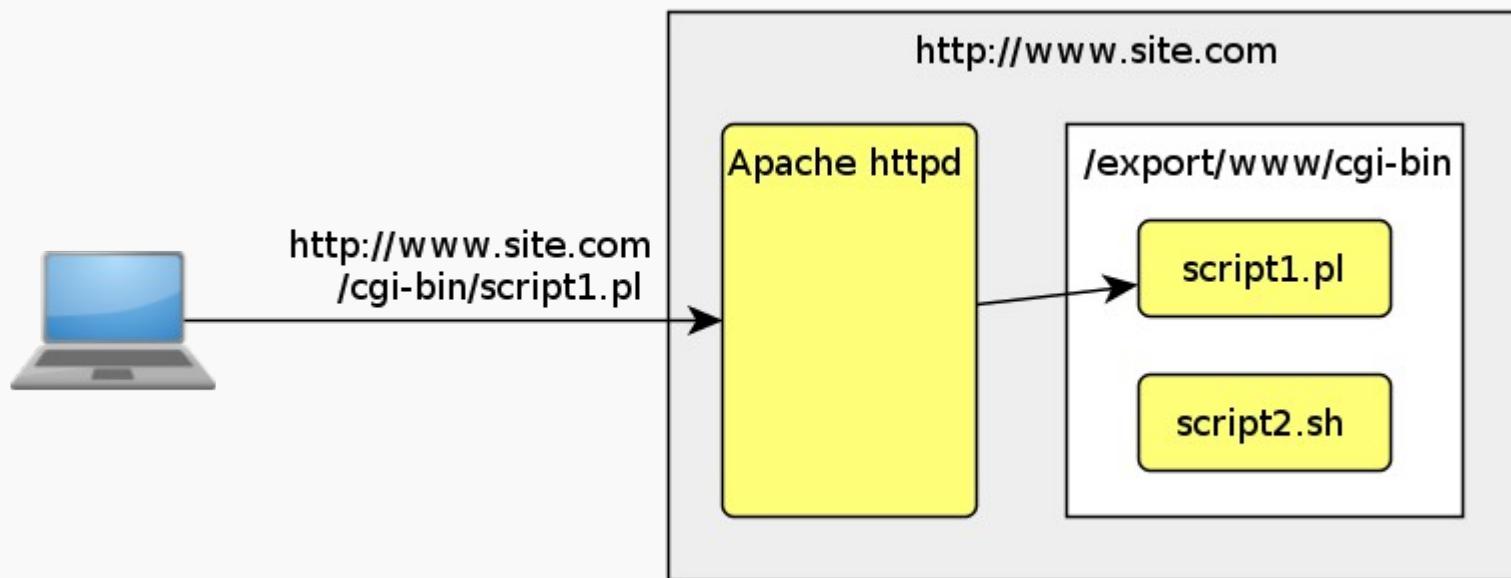
    printf("<HTML>\n");
    printf("<HEAD>\n");
    printf("<TITLE>Hello, World!</TITLE>\n");
    printf("</HEAD>\n");
    printf("<BODY>\n");
    printf("<H1>Hello, World</H1>\n");
    printf("</BODY>\n");
    printf("</HTML>\n");

    return 0;
}
```

Конфигурация веб-сервера

Apache (httpd.conf):

```
ScriptAlias /cgi-bin/ "/opt/www/cgi-bin/"
```



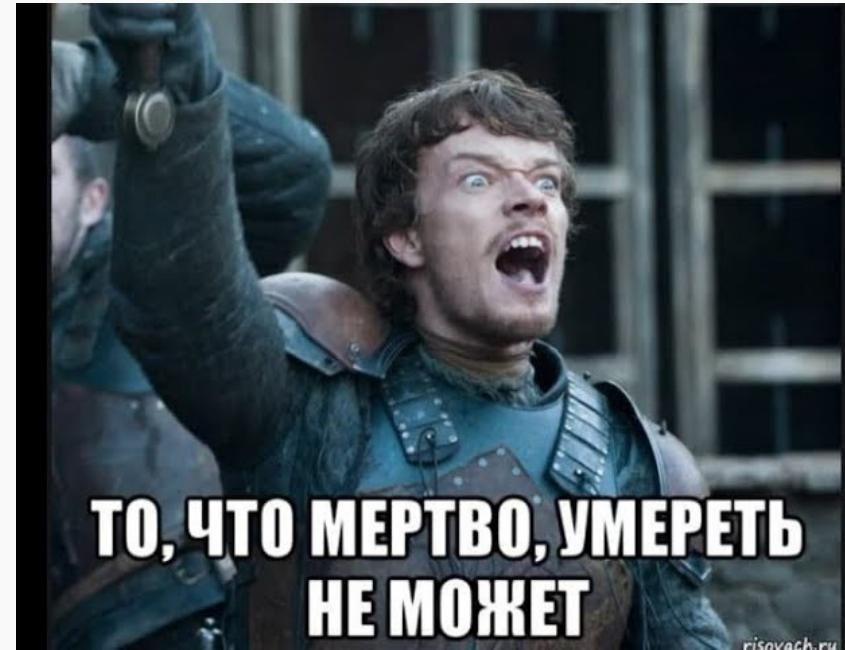
Достоинства и недостатки CGI-сценариев

- Достоинства:
 - Программы могут быть написаны на множестве языков программирования.
 - «Падение» CGI-сценария не приводит к «падению» всего сервера.
 - Исключены конфликты при параллельной обработке нескольких запросов.
 - Хорошая поддержка веб-серверами.
- Недостатки:
 - Высокие накладные расходы на создание нового процесса.
 - Плохая масштабируемость.
 - Слабое разделение уровня представления и бизнес-логики.
 - Могут быть платформо-зависимыми.

FastCGI

- Развитие технологии CGI.
- Все запросы могут обрабатываться одним процессом CGI-программы (фактическая реализация определяется программистом).
- Веб-сервер взаимодействует с процессом через UNIX Domain Sockets или TCP/IP (а не через `stdin` и `stdout`).
- Для общения по TCP/IP используется бинарный протокол, обеспечивающий передачу любых текстовый данных (например HTTP запросов и ответов).

PHP



3.1.2. FastCGI на Java (ЛР 1)

FastCGI на Java

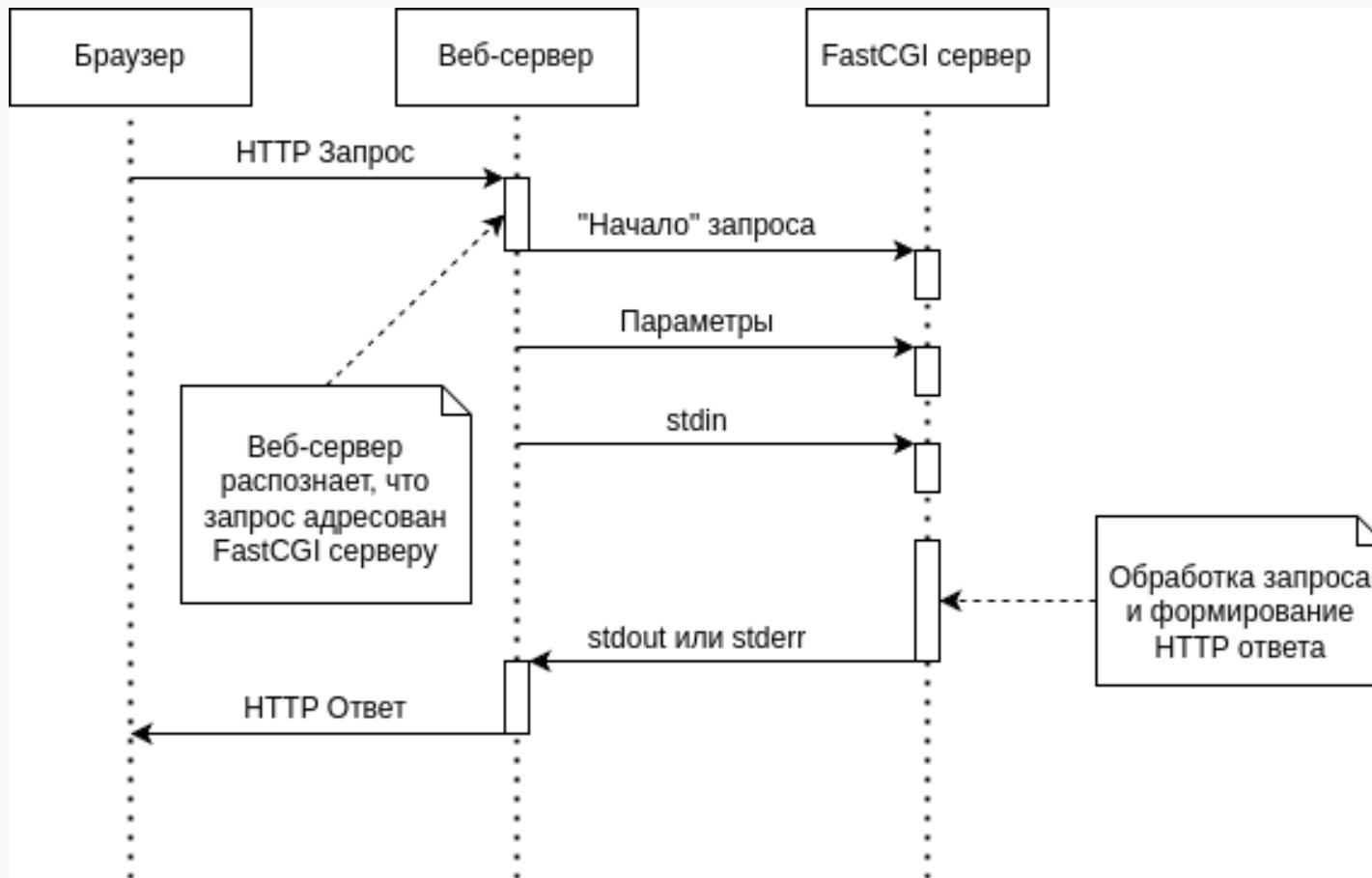
FastCGI: либо веб-сервер поддерживает набор запущенных процессов, либо отдельный FastCGI сервер.

- HTTP «поверх» FastCGI для взаимодействия браузера и сервера;
- Есть «старинные» библиотеки от разработчиков стандарта;
- Есть «более современные» (например, для Netty).

Типы FastCGI пакетов

- «Начало» запроса и «конец» ответа;
- Пакет с параметрами (веб-сервер передает параметры fastcgi-серверу);
- Пакеты «`stdin`», «`stdout`», «`stderr`» - для передачи тела запроса и ответа;
- Служебные (management) пакеты.

Типы FastCGI пакетов



Пример (1)

**Библиотека! Можно
скачать с se**

```
public static void main(String[] args) {  
    var fcgiInterface = new FCGIInterface();  
    while (fcgiInterface.FCGIaccept() >= 0) {  
        var content = """  
            <html>  
                <head><title>Java FastCGI Hello World</title></head>  
                <body><h1>Hello, World!</h1></body>  
            </html>""";  
        var httpResponse = """  
            HTTP/1.1 200 OK  
            Content-Type: text/html  
            Content-Length: %d  
  
            %s  
  
            """.formatted(content.getBytes(StandardCharsets.UTF_8).length, content);  
        System.out.println(httpResponse);  
    }  
}
```

Пример (2)

- **FCGIInterface** — класс из библиотеки от разработчиков FastCGI, реализует в себе «магию» по перенаправлению System.in, System.out, System.err в потоки TCP сокета;
- **FCGIInterface.request.params** — объект **Properties**, через который можно узнать список параметров, переданных веб-сервером fastcgi-серверу;
- Наиболее значимые параметры:
REQUEST_METHOD — GET, POST и т. д.;
QUERY_STRING — параметры запроса после «?»;
CONTENT_TYPE — значение заголовка запроса;
CONTENT_LENGTH — значение заголовка запроса.

Пример (3)

```
private static String readRequestBody() throws IOException {
    FCGIInterface.request.inStream.fill();

    var contentLength = FCGIInterface.request.inStream.available();
    var buffer = ByteBuffer.allocate(contentLength);
    var readBytes =
        FCGIInterface.request.inStream.read(buffer.array(), 0,
        contentLength);

    var requestBodyRaw = new byte[readBytes];
    buffer.get(requestBodyRaw);
    buffer.clear();

    return new String(requestBodyRaw, StandardCharsets.UTF_8);
}
```

Apache httpd + FastCGI

Apache (**httpd.conf**):

Шаблон файла конфигурации для
запуска на helios можно скачать с se

```
LoadModule fastcgi_module libexec/apache24/mod_fastcgi.so
```

```
FastCgiExternalServer
"/home/studs/sXXXXXX/httpd-root/fcgi-bin/hello-world.jar" -host
localhost:24001 -nph
Alias /fcgi-bin/ "/home/studs/sXXXXXX/httpd-root/fcgi-bin/"
<Directory "/home/studs/sXXXXXX/httpd-root/fcgi-bin">
    AllowOverride None
    Options None
    Require all granted
</Directory>
```

Обратиться к FastCGI-серверу можно так:

<https://helios.cs.ifmo.ru:<PORT>/fcgi-bin/hello-world.jar>

Запуск серверов

```
httpd -f ~/httpd-root/conf/httpd.conf -k start
```

```
java -DFCGI_PORT=24001 -jar server.jar
```

Не забудьте поменять порты!

3.1.3. Сервлеты

Платформа Java EE

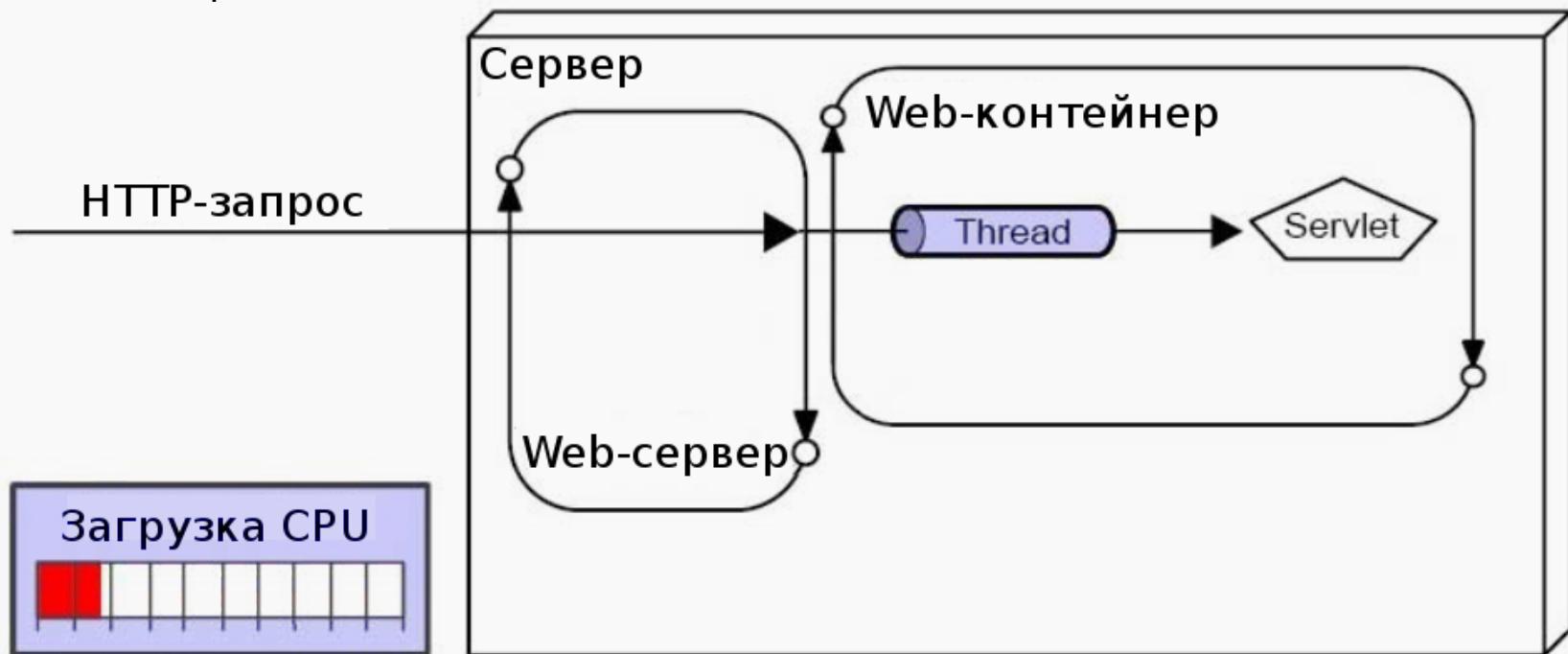
- Набор стандартов и спецификаций для создания корпоративных приложений на Java.
- Спецификации Java EE реализуются серверами приложений:
 - Apache Tomcat
 - Sun / Oracle GlassFish
 - BEA / Oracle WebLogic
 - IBM WebSphere
 - RedHat JBoss

Java Servlets

- Сервлеты — это серверные сценарии, написанные на Java.
- Жизненным циклом сервлетов управляет веб-контейнер (он же контейнер сервлетов).
- В отличие от CGI, запросы обрабатываются в отдельных потоках (а не процессах) на веб-контейнере.

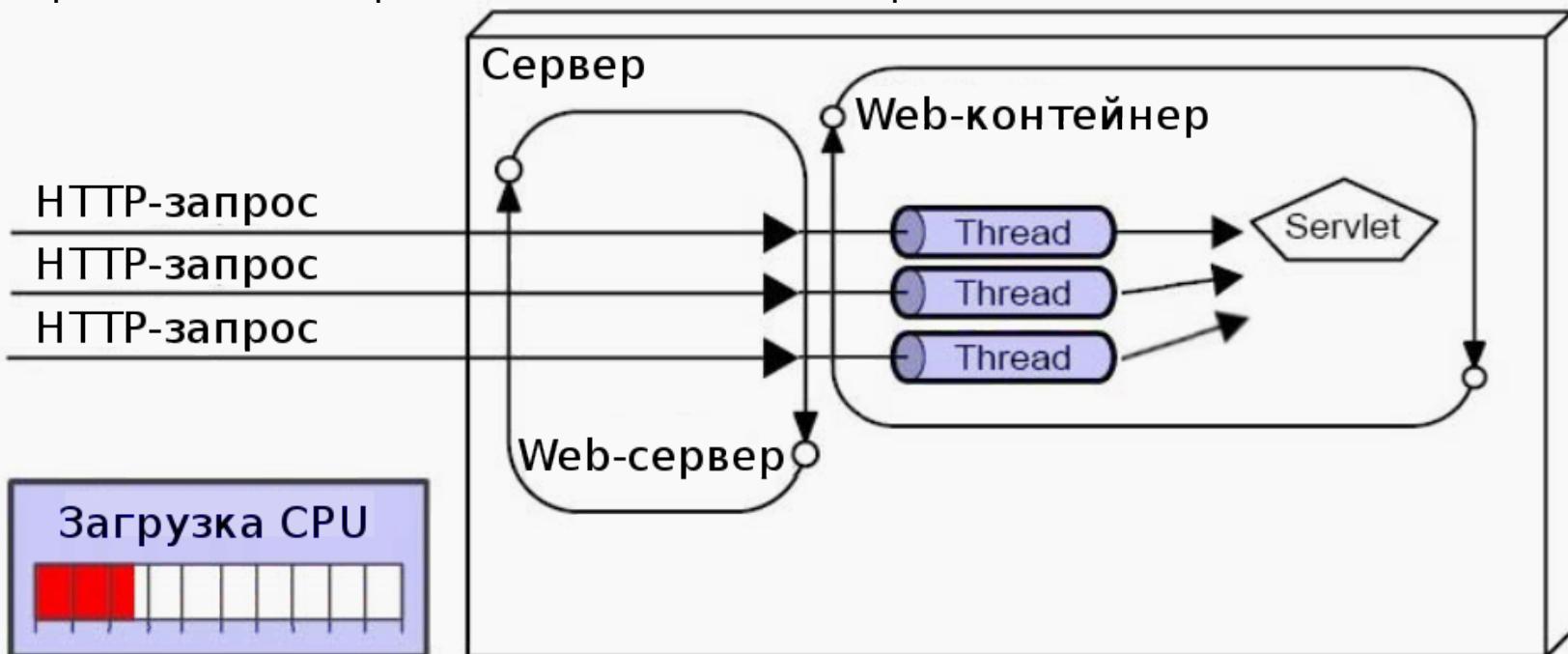
Обработка HTTP-запросов сервлетом

Один запрос:



Обработка HTTP-запросов сервлетом (продолжение)

Параллельная обработка нескольких запросов:

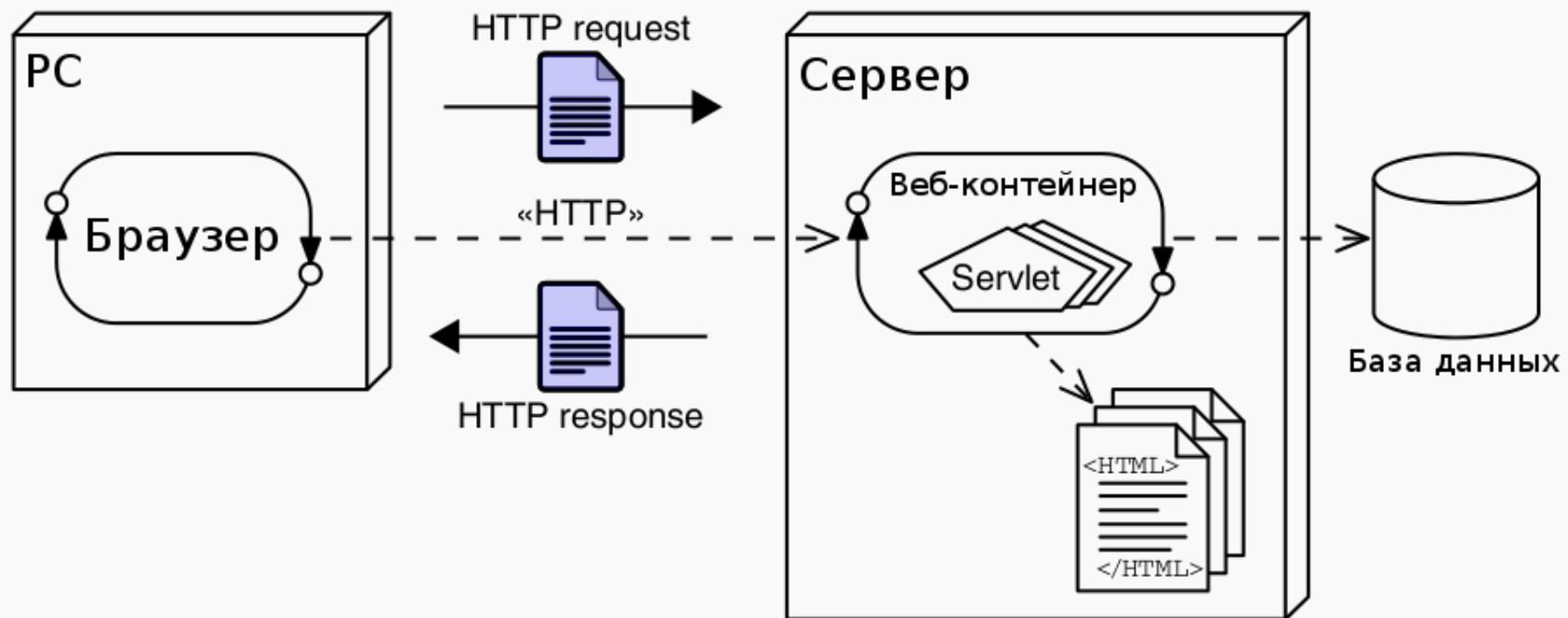


Преимущества и недостатки сервлетов

- Преимущества серверов:
 - Выполняются быстрее, чем CGI-сценарии.
 - Хорошая масштабируемость.
 - Надёжность и безопасность (реализованы на Java).
 - Платформенно-независимы.
 - Множество инструментов мониторинга и отладки.
- Недостатки серверов:
 - Слабое разделение уровня представления и бизнес-логики.
 - Возможны конфликты при параллельной обработке запросов.

3.1.4. Разработка сервлетов

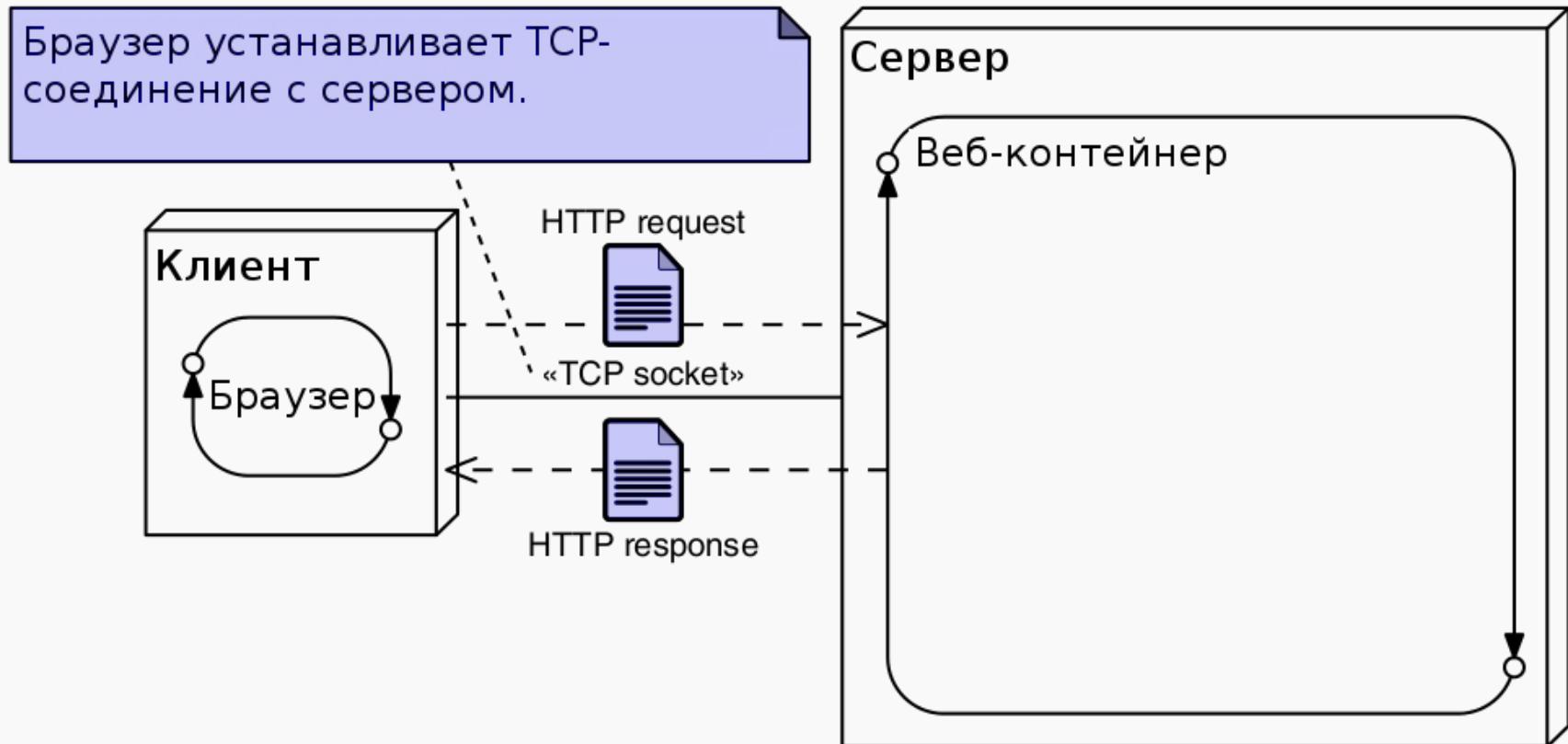
Архитектура веб-контейнера





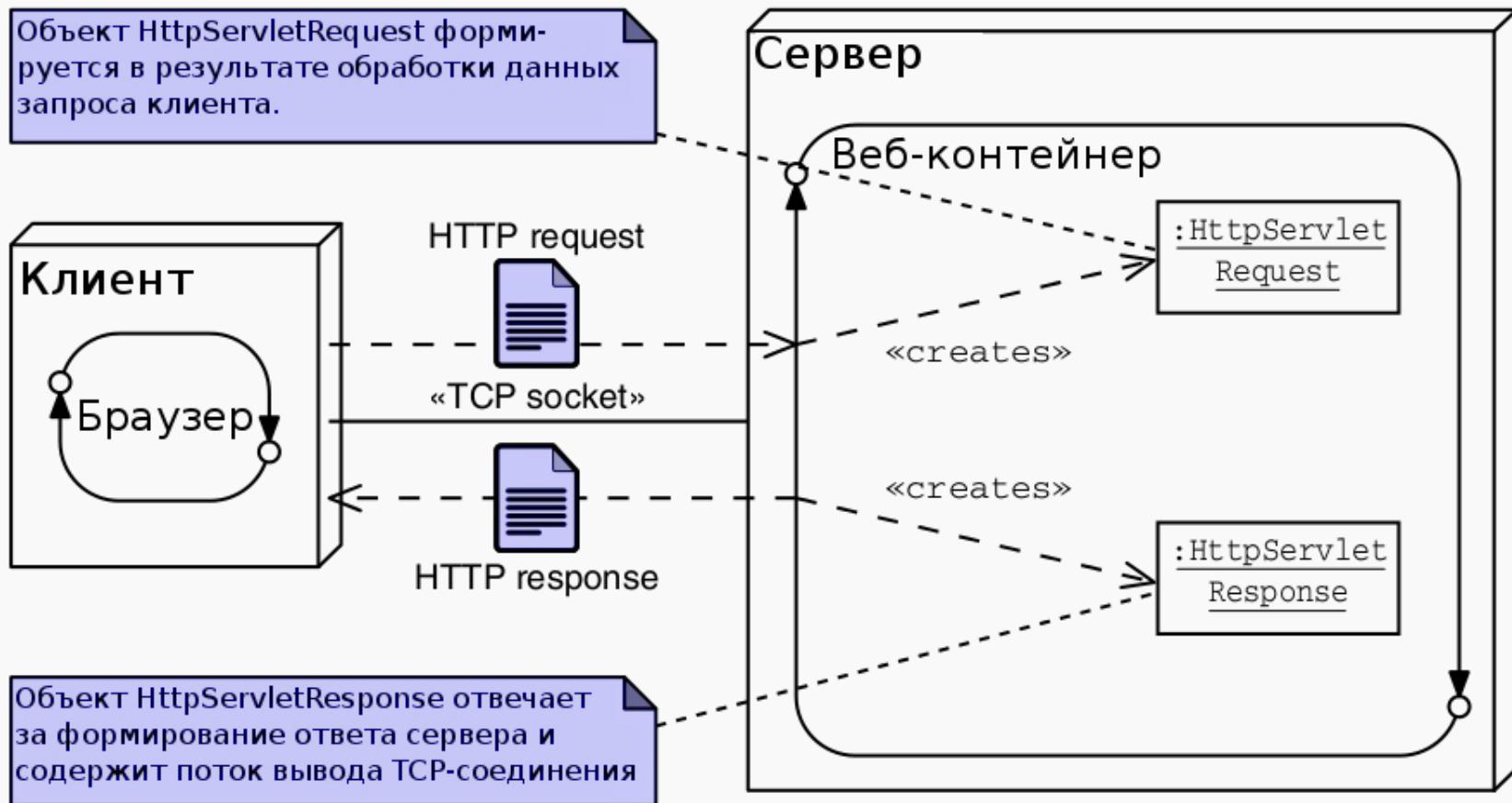
Обработка HTTP-запроса

1. Браузер формирует HTTP-запрос и отправляет его на сервер.



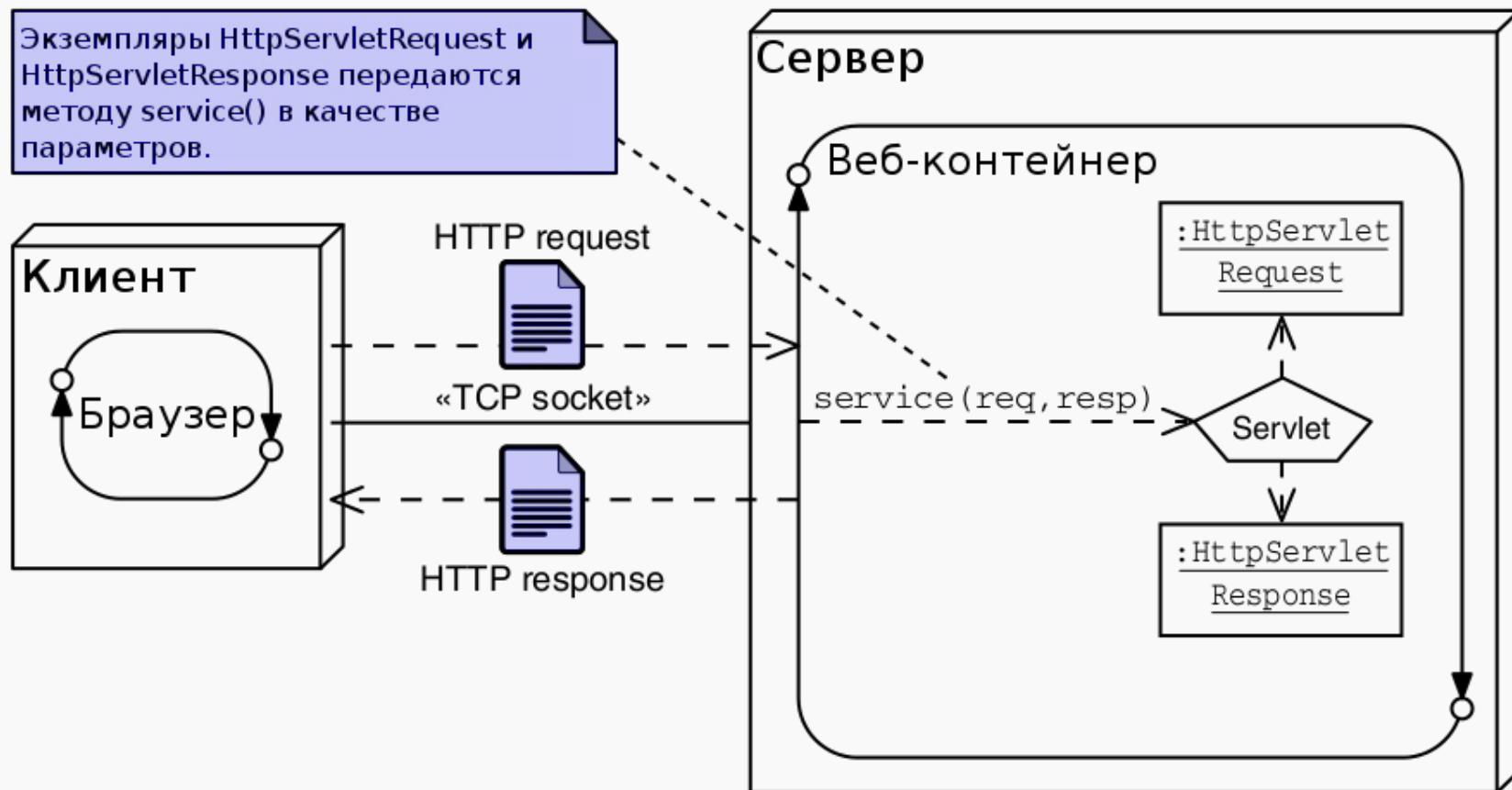
Обработка HTTP-запроса (продолжение)

2. Веб-контейнер создаёт объекты HttpServletRequest и HttpServletResponse.



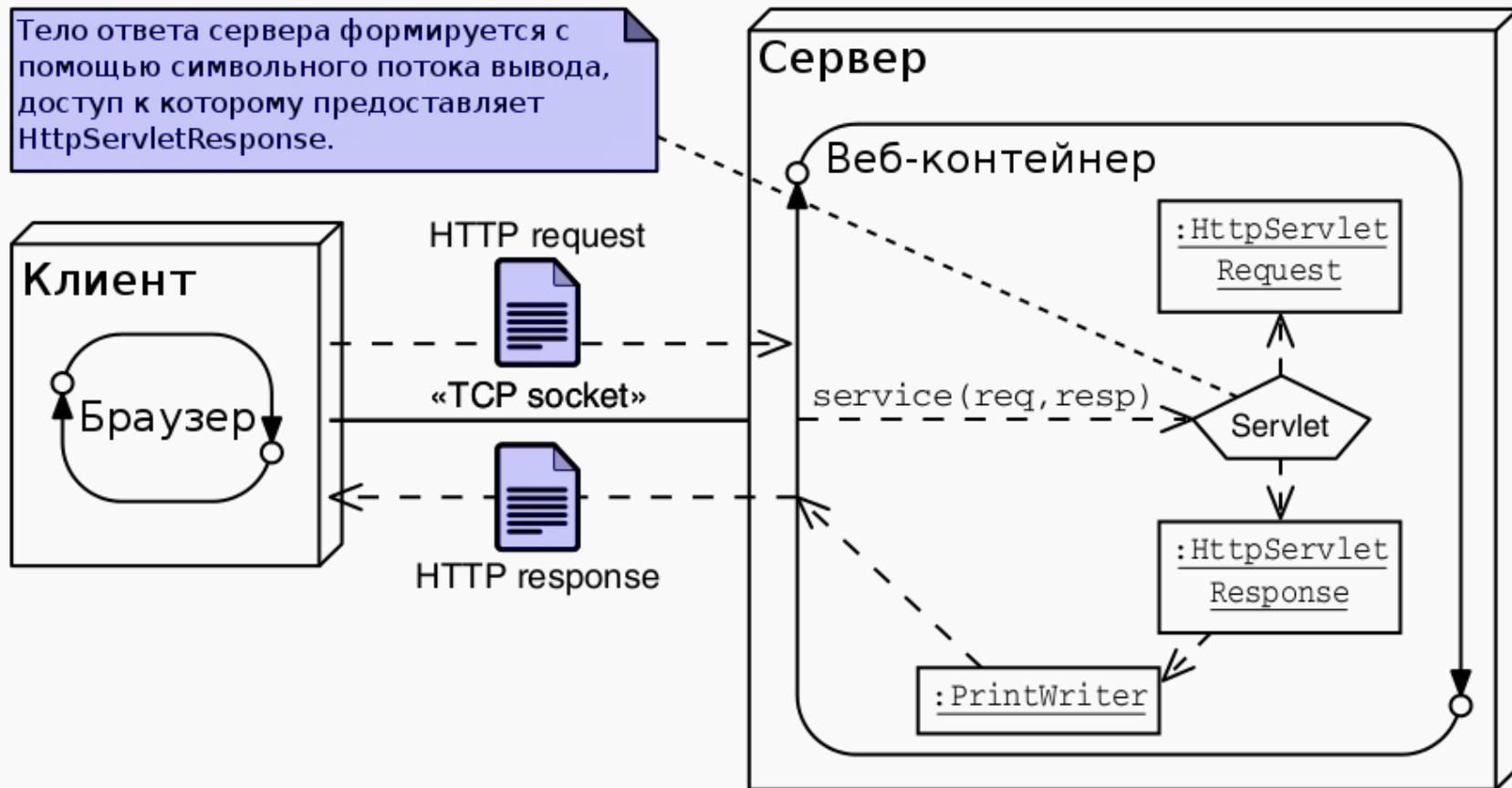
Обработка HTTP-запроса (продолжение)

3. Веб-контейнер вызывает метод `service` сервлета.



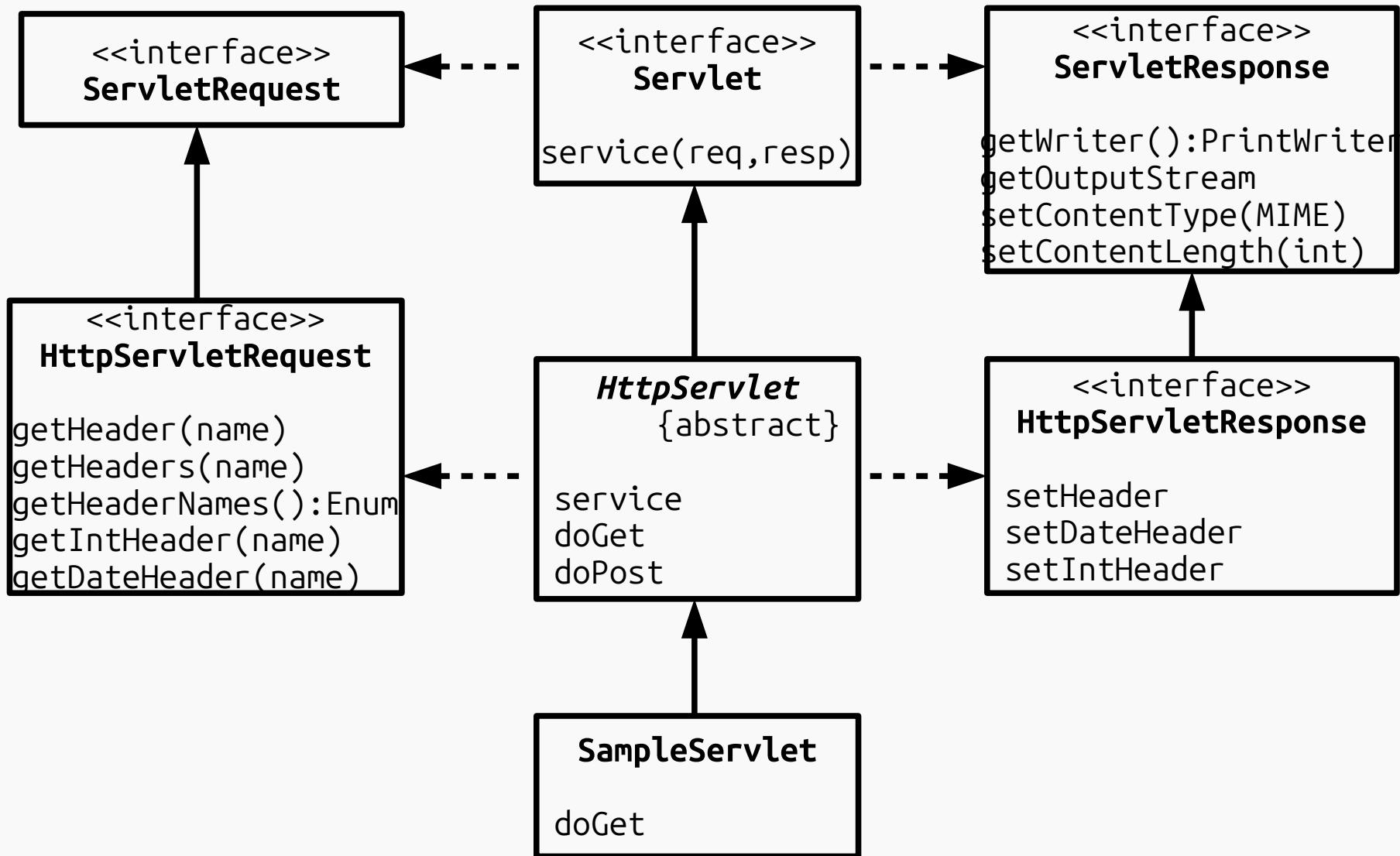
Обработка HTTP-запроса (продолжение)

4. Сервлет формирует ответ и записывает его в поток вывода `HttpServletResponse`.





HttpServlet API



Пример сервлета

```
package sample.servlet;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

// Support classes
import java.io.IOException;
import java.io.PrintWriter;

public class SampleServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException {
        // Заголовок страницы
        String pageTitle = "Пример сервлета";
    }
}
```

Пример сервлета (продолжение)

```
// Content Type
response.setContentType("text/html");

PrintWriter out = response.getWriter();

// Формируем HTML
out.println("<html>");
out.println("<head>");
out.println("<title>" + pageTitle + "</title>");
out.println("</head>");
out.println("<body bgcolor='white'>");
out.println("<h3>" + pageTitle + "</h3>");
out.println("<p>");
out.println("Hello, world!");
out.println("</p>");
out.println("</body>");
out.println("</html>");

}

}
```

Конфигурация сервлета



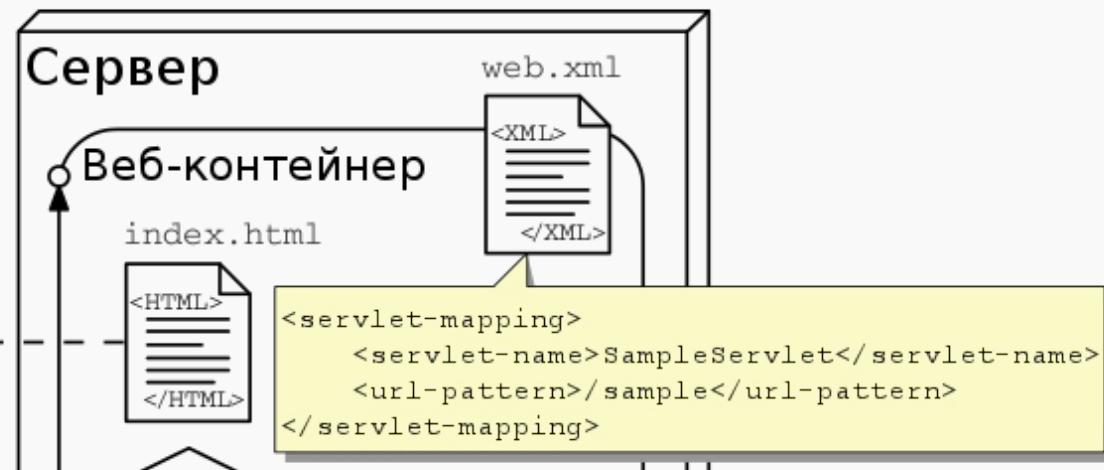
Веб-контейнер создаёт один (строго)
экземпляр сервлета на каждую запись
в дескрипторе.

Конфигурация сервлета (продолжение)



http://localhost:8080/
sampleApp/index.html

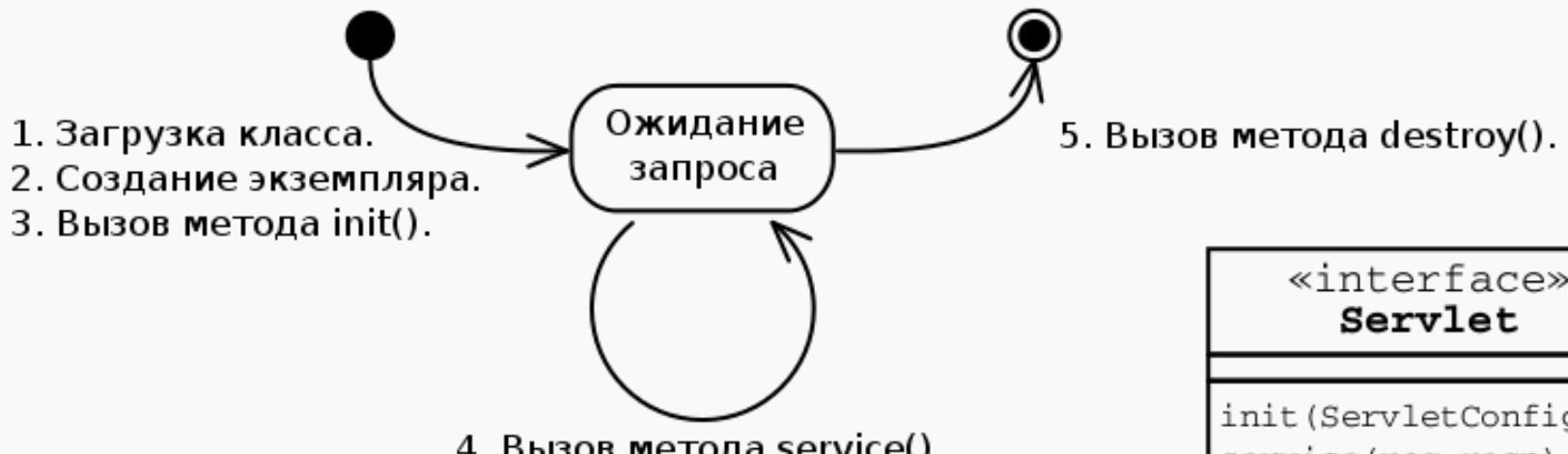
http://localhost:8080/
sampleApp/sample



Веб-контейнер перенаправляет запрос
с URL на конкретный сервле~~т~~т в соответствии с
конфигурацией.

Жизненный цикл сервлета

- Жизненным циклом сервлета управляет веб-контейнер.
- Методы, управляющие жизненным циклом, должен вызывать *только* веб-контейнер.

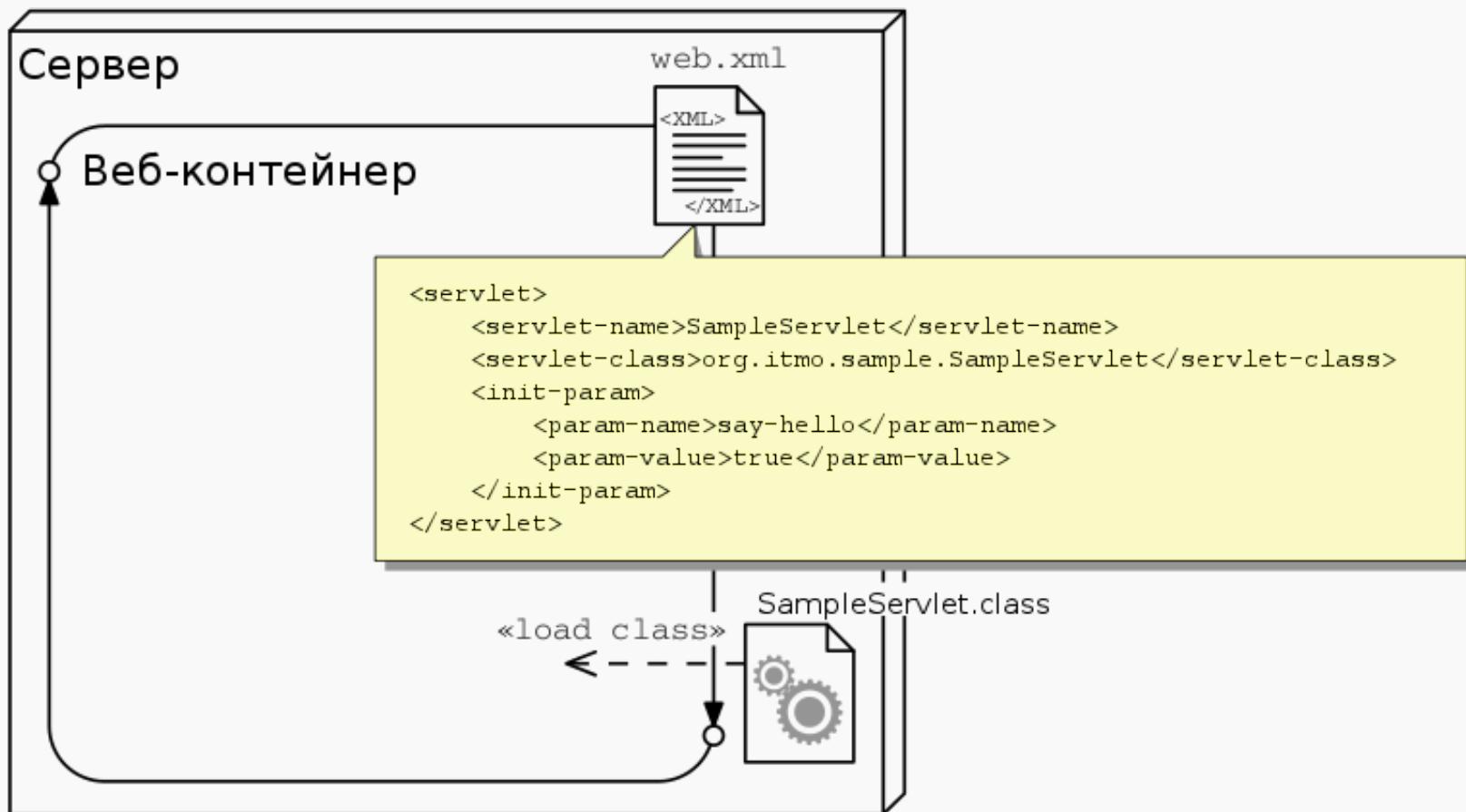


```
«interface»  
Servlet  
-----  
init(ServletConfig)  
service(req, resp)  
destroy()
```

Жизненный цикл сервлета (продолжение)

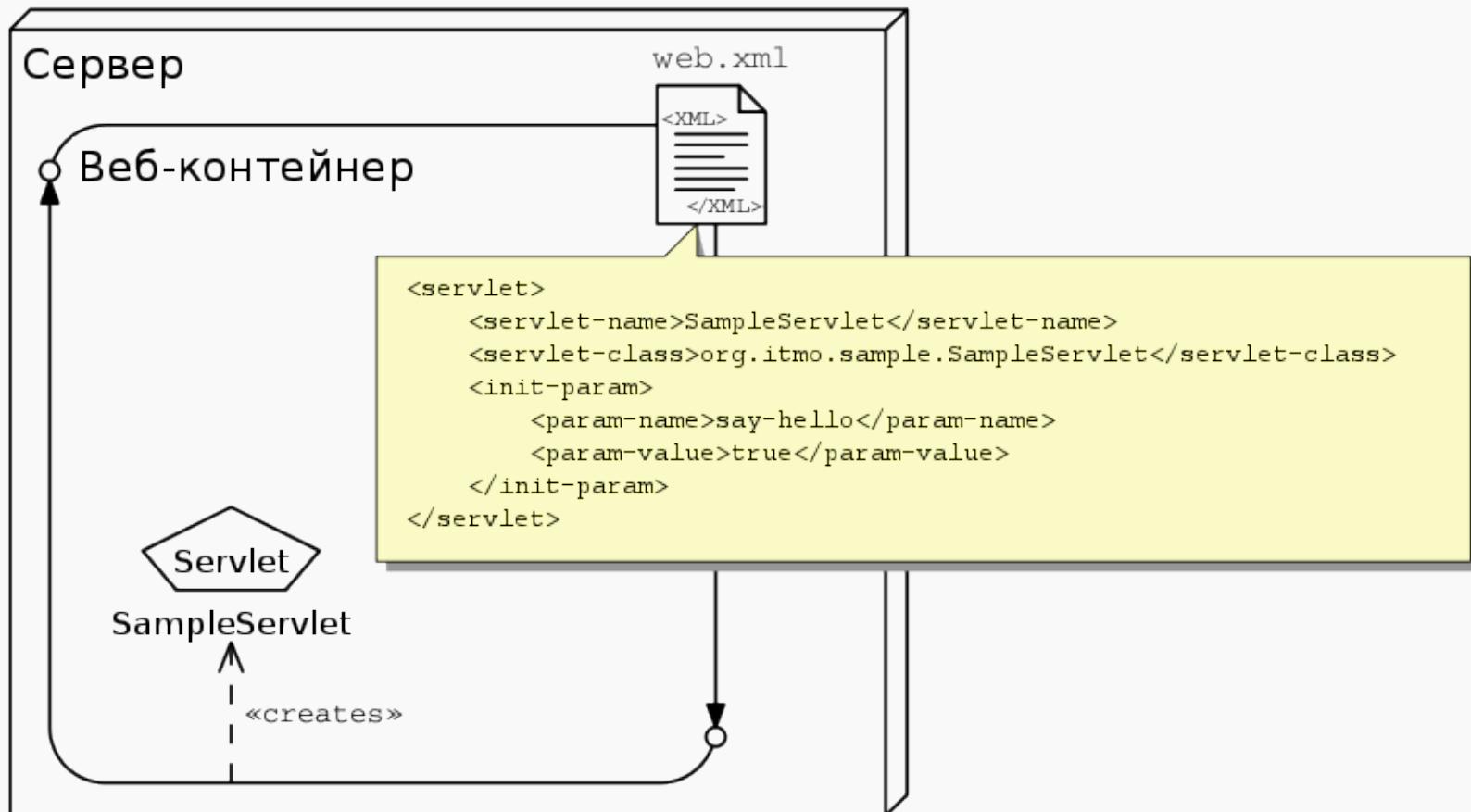
1. Загрузка класса сервлета.

Проверяются пути: /WEB-INF/classes/, WEB-INF/lib/*.jar, стандартные классы Java SE и классы веб-контейнера.



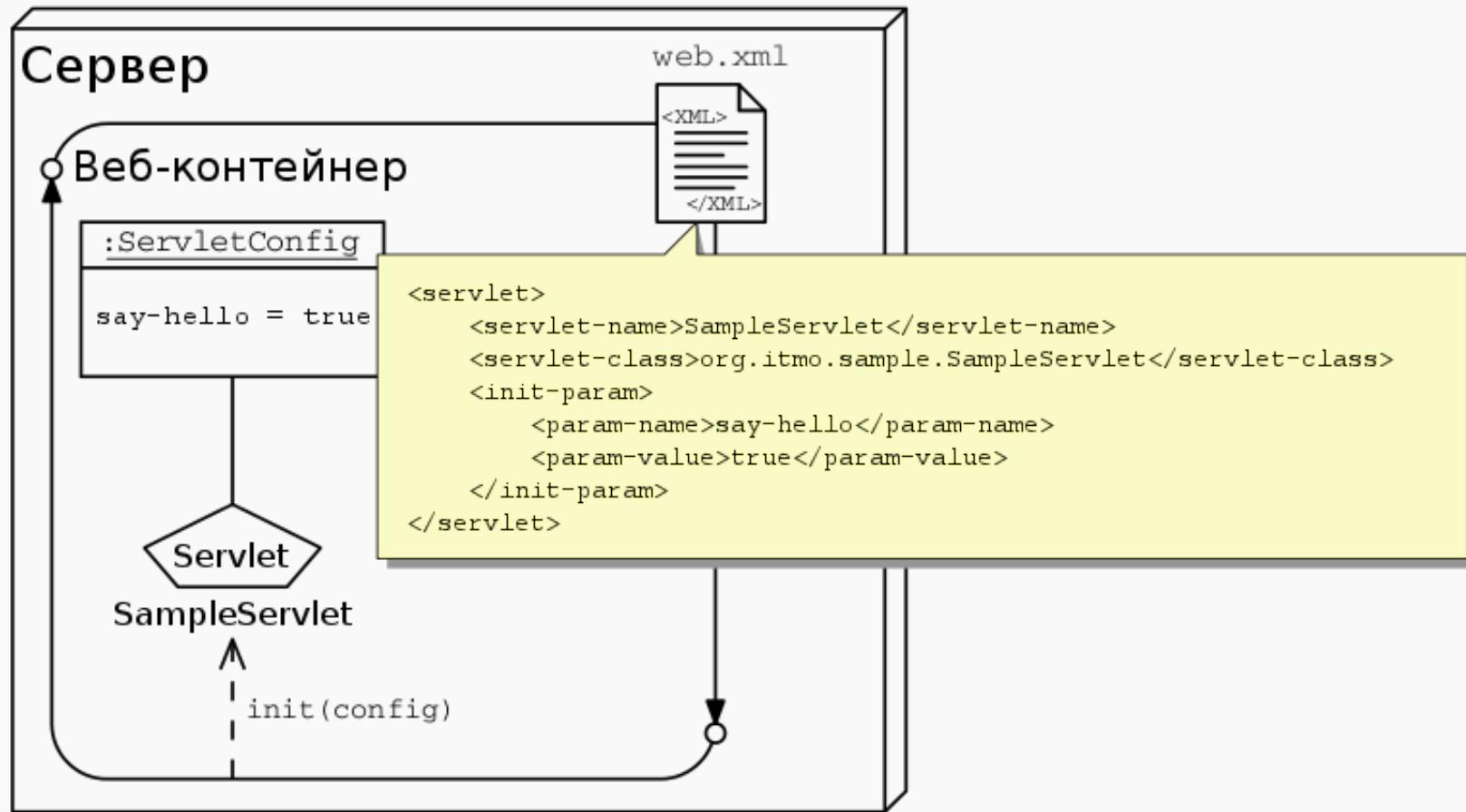
Жизненный цикл сервлета (продолжение)

2. Создание экземпляра сервлета.



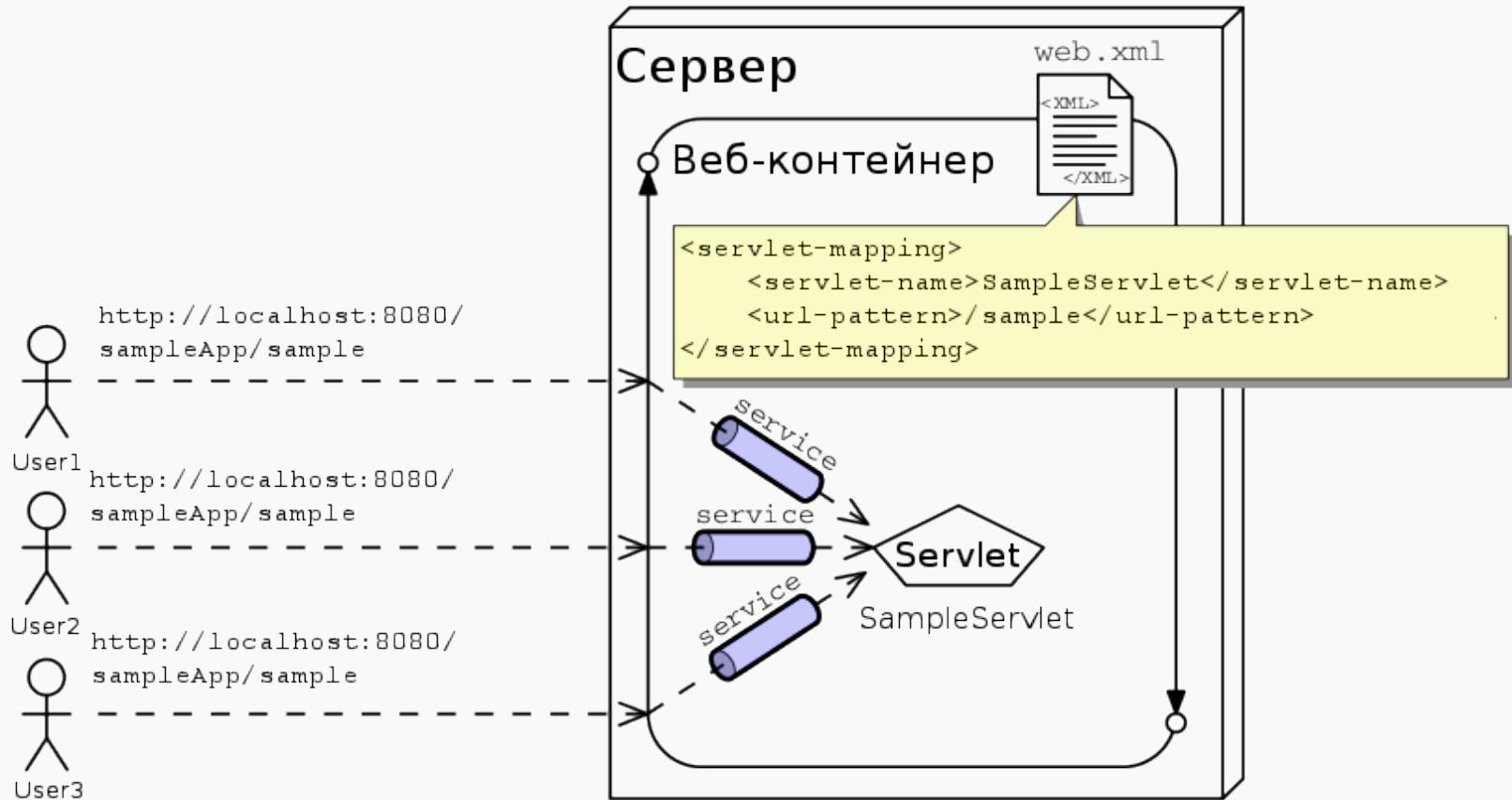
Жизненный цикл сервлета (продолжение)

3. Вызов метода init.



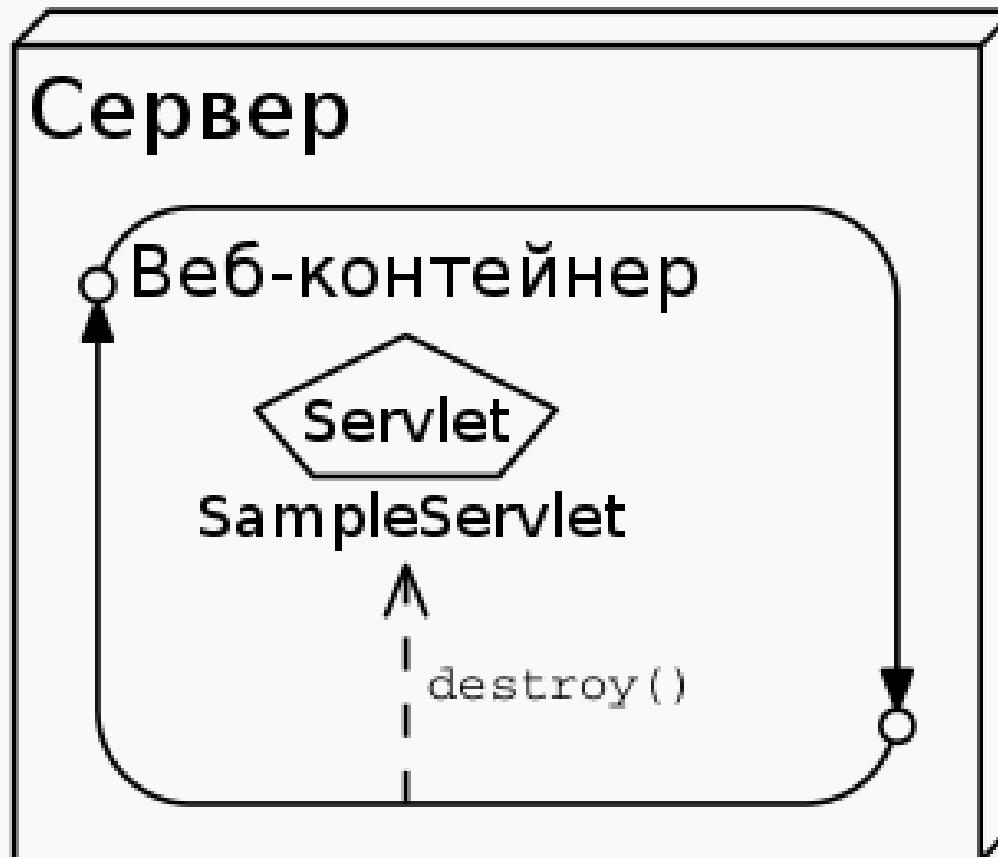
Жизненный цикл сервлета (продолжение)

4. Обработка HTTP-запросов.



Жизненный цикл сервлета (продолжение)

5. Вызов метода `destroy`.



Контекст сервлетов

- API, с помощью которого сервле́т может взаимодействовать со своим контейнером.
- Доступ к методам осуществляется через интерфейс `javax.servlet.ServletContext`.
- У всех сервлетов внутри приложения общий контекст.
- В контекст можно помещать общую для всех сервлетов информацию (методы `getAttribute` и `setAttribute`).
- Если приложение — распределённое, то на каждом экземпляре JVM контейнером создаётся свой контекст.

Контекст сервлетов (продолжение)

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DemoServlet extends HttpServlet{
    public void doGet(HttpServletRequest req,HttpServletResponse
res)
        throws ServletException,IOException {
    res.setContentType("text/html");
    PrintWriter pw=res.getWriter();

    //creating ServletContext object
    ServletContext context=getServletContext();

    //Getting the value of the initialization parameter
    // and printing it
    String driverName=context.getInitParameter("dname");
    pw.println("driver name is="+driverName);

    pw.close();
}
}
```

HTTP-сессии

- HTTP — stateless-протокол.
- `javax.servlet.HttpSession` — интерфейс, позволяющий идентифицировать конкретного клиента (браузер) при обработке множества HTTP-запросов от него.
- Экземпляр `HttpSession` создаётся при первом обращении клиента к приложению и сохраняется некоторое (настраиваемое) время после последнего обращения.
- Идентификатор сессии либо помещается в cookie, либо добавляется к URL. Если удалить этот идентификатор, то сервер не сможет идентифицировать клиента и создаст новую сессию.
- В экземпляр `HttpSession` можно помещать общую для этой сессии информацию (методы `getAttribute` и `setAttribute`).
- Сессия «привязана» к конкретному приложению; у разных приложений — разные сессии.
- В распределённом окружении обеспечивается сохранение целостности данных в HTTP-сессии (независимо от количества экземпляров JVM).

HTTP-сессии (продолжение)

```
public class SimpleSession extends HttpServlet {  
  
    public void doGet(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, java.io.IOException {  
  
        response.setContentType("text/html");  
        java.io.PrintWriter out = response.getWriter();  
        HttpSession session = request.getSession();  
  
        out.println("<html>");  
        out.println("<head>");  
        out.println("<title>Simple Session Tracker</title>");  
        out.println("</head>");  
        out.println("<body>");  
  
        out.println("<h2>Session Info</h2>");  
        out.println("session Id: " + session.getId() + "<br><br>");  
        out.println("The SESSION TIMEOUT period is "  
            + session.getMaxInactiveInterval() + " seconds.<br><br>");  
        out.println("Now changing it to 20 minutes.<br><br>");  
        session.setMaxInactiveInterval(20 * 60);  
        out.println("The SESSION TIMEOUT period is now "  
            + session.getMaxInactiveInterval() + " seconds.");  
  
        out.println("</body>");  
        out.println("</html>");  
    }  
}
```

Диспетчеризация запросов сервлетами

- Сервлеты могут делегировать обработку запросов другим ресурсам (сервлетам, JSP и HTML-страницам).
- Диспетчеризация осуществляется с помощью реализаций интерфейса `javax.servlet.RequestDispatcher`.
- Два способа получения `RequestDispatcher` — через `ServletRequest` (абсолютный или относительный URL) и `ServletContext` (только абсолютный URL).
- Два способа делегирования обработки запроса — `forward` и `include`.

Диспетчеризация запросов сервлетами (продолжение)

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet{
    public void doGet(HttpServletRequest req,
                      HttpServletResponse res)
        throws ServletException, IOException {
        RequestDispatcher dispatcher =
request.getRequestDispatcher("index.jsp");
        dispatcher.forward( request, response );
    }
}
```

Фильтры запросов

- Фильтры позволяют осуществлять пред- и постобработку запросов до и после передачи их ресурсу (сервлету, JSP или HTML-странице).
- Пример предобработки — допуск к странице только авторизованных пользователей.
- Пример постобработки — запись в лог времени обработки запроса.
- Реализуют интерфейс `javax.servlet.Filter`.
- Ключевой метод — `doFilter`.
- Метод `doFilter` класса `FilterChain` передаёт управление следующему фильтру или целевому ресурсу; таким образом, возможна реализация последовательностей фильтров, обрабатывающих один и тот же запрос.

Фильтры запросов (продолжение)

```
import javax.servlet.*;

public class MyFilter implements Filter{

    public void init(FilterConfig arg0) throws ServletException {}

    public void doFilter(ServletRequest req, ServletResponse resp,
        FilterChain chain) throws IOException, ServletException {
        PrintWriter out=resp.getWriter();
        out.print("filter is invoked before");
        chain.doFilter(req, resp);//sends request to next resource

        out.print("filter is invoked after");
    }

    public void destroy() {}
}
```

Конфигурация фильтров

```
<web-app>
```

```
  <servlet>
    <servlet-name>s1</servlet-name>
    <servlet-class>HelloServlet</servlet-class>
  </servlet>
```

```
  <servlet-mapping>
    <servlet-name>s1</servlet-name>
    <url-pattern>/servlet1</url-pattern>
  </servlet-mapping>
```

```
  <filter>
    <filter-name>f1</filter-name>
    <filter-class>MyFilter</filter-class>
  </filter>
```

```
  <filter-mapping>
    <filter-name>f1</filter-name>
    <url-pattern>/servlet1</url-pattern>
  </filter-mapping>
```

```
</web-app>
```

3.2. Шаблоны проектирования в веб- приложениях

Основные понятия

- **Шаблон проектирования или паттерн** — повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста (© Wikipedia).
- Описывает подход к решению типовой задачи.
- Одну и ту же задачу часто можно решить с использованием разных шаблонов.
- Существует много литературы с описанием различных шаблонов проектирования.

Зачем нужны паттерны

- Позволяют избежать «типовых» ошибок при разработке типовых решений.
- Позволяют кратко описать подход к решению задачи — программистам, знающим шаблоны, проще обмениваться информацией.
- Легче поддерживать код — его поведение более предсказуемо.

GoF-паттерны

- Описаны в книге 1994 г. «*Design Patterns: Elements of Reusable Object-Oriented Software*» («Приёмы объектно-ориентированного проектирования. Паттерны проектирования»).
- Авторы — Эрих Гамма (Erich Gamma), Ричард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson), Джон Влиссидс (John Vlissides) — Gang of Four (GoF, «Банда Четырёх»).
- В книге описаны 23 классических шаблона проектирования.

Порождающие GoF-паттерны

- Abstract Factory — Абстрактная фабрика.
- Builder — Строитель.
- Factory Method — Фабричный метод.
- Prototype — Прототип.
- Singleton — Одиночка.

Пример порождающего GoF-паттерна



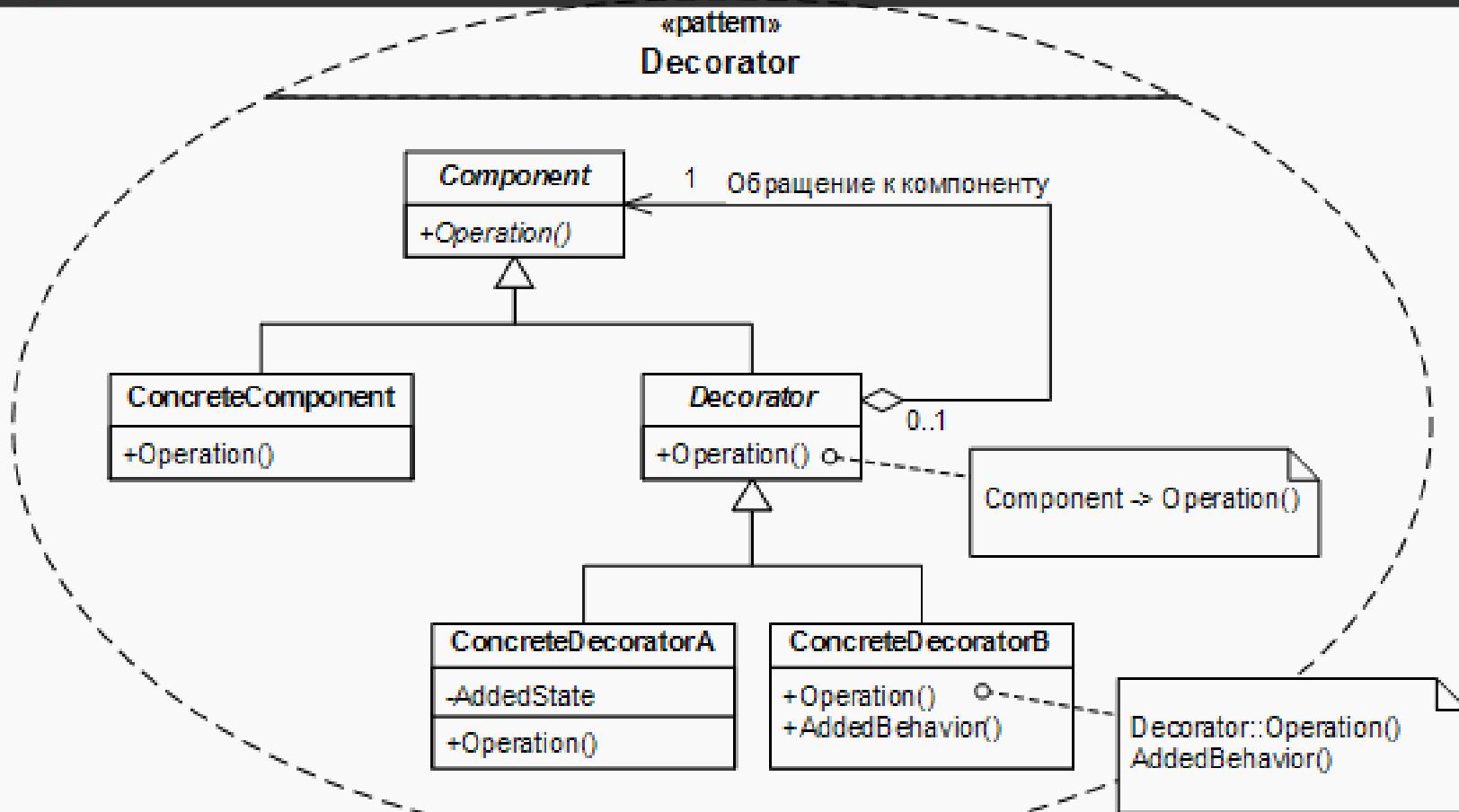
Singleton (одиночка):

- Гарантирует, что у класса есть *только один экземпляр*, и предоставляет к нему глобальную точку доступа.
- Можно пользоваться экземпляром класса (в отличие от статических методов).

Структурные GoF-паттерны

- Adapter — Адаптер.
- Bridge — Мост.
- Composite — Компоновщик.
- Decorator — Декоратор.
- Facade — Фасад.
- Flyweight — Приспособленец.
- Proxy — Заместитель.

Пример структурного GoF-паттерна

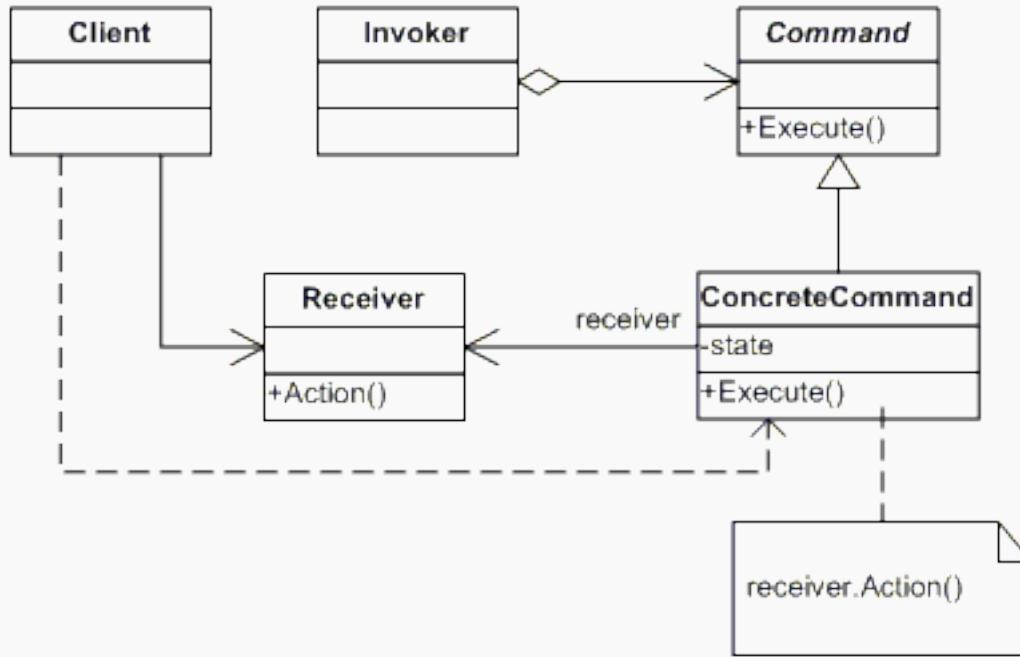


Decorator (декоратор) — позволяет динамически подключать дополнительное поведение к объекту без использования наследования.

Поведенческие GoF-паттерны

- Chain of responsibility — Цепочка обязанностей.
- Command — Команда.
- Interpreter — Интерпретатор.
- Iterator — Итератор.
- Mediator — Посредник.
- Memento — Хранитель.
- Observer — Наблюдатель.
- State — Состояние.
- Strategy — Стратегия.
- Template — Шаблонный метод.
- Visitor — Посетитель.

Пример поведенческого GoF-паттерна



Command (команда) — команда передаётся с помощью специального объекта, который заключает в себе само действие (т. е. логику) и его параметры.

Архитектурные шаблоны

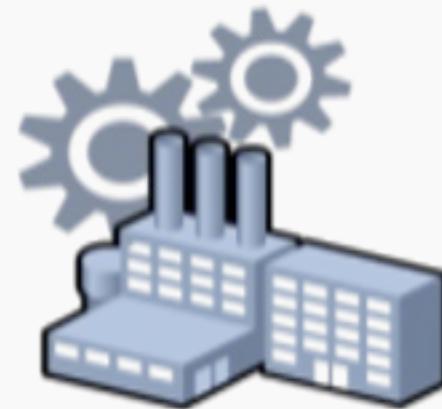
- Более высокий уровень по сравнению с шаблонами проектирования.
- Описывают архитектуру всей системы или приложения.
- Обычно имеют дело не с отдельными классами, а с целыми компонентами или модулями.
- Компоненты и модули могут быть построены с использованием различных шаблонов проектирования.

Архитектура веб-приложений

3 уровня архитектуры:



Клиент



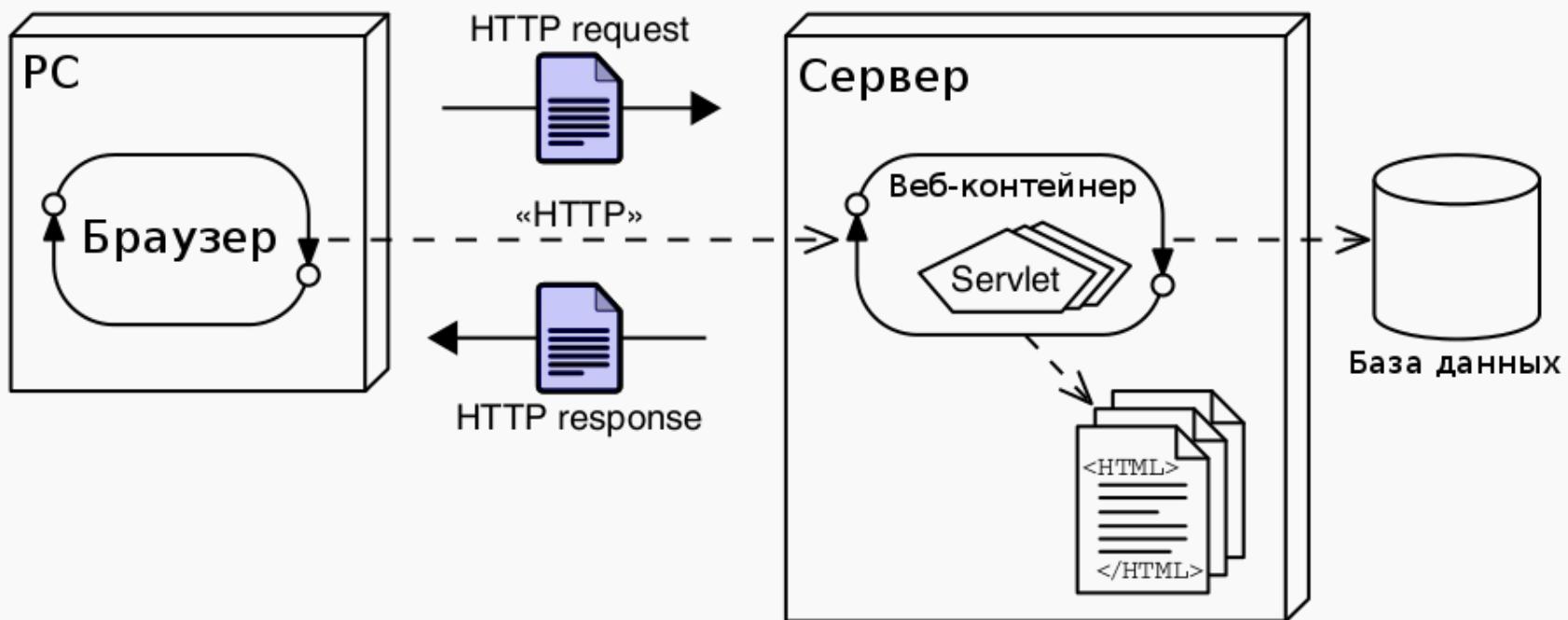
Бизнес-логика



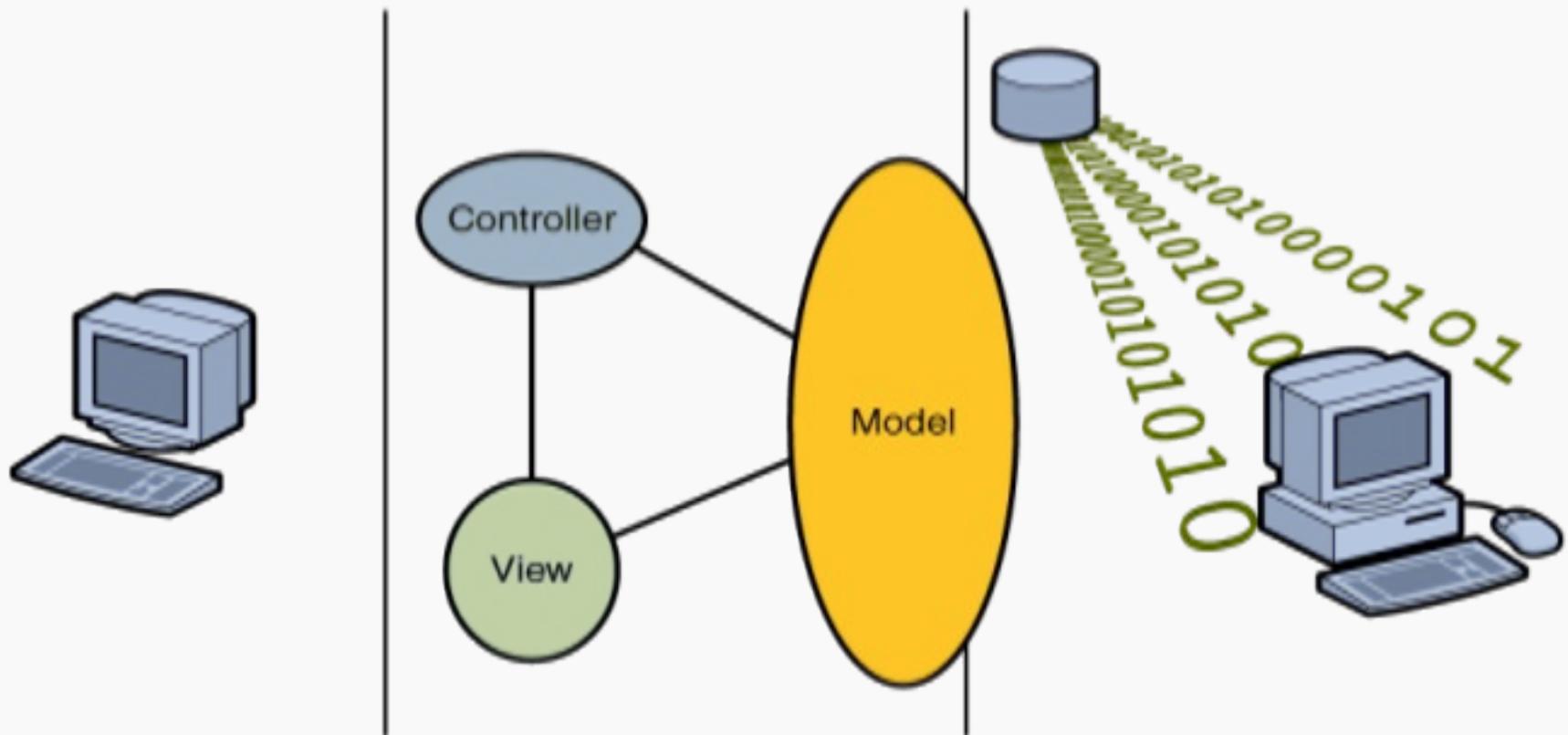
Данные

Архитектура Model 1 (Java)

- Предназначена для проектирования приложений небольшого масштаба и сложности.
- За обработку данных и представления отвечает *один и тот же* компонент (сервлет или JSP).

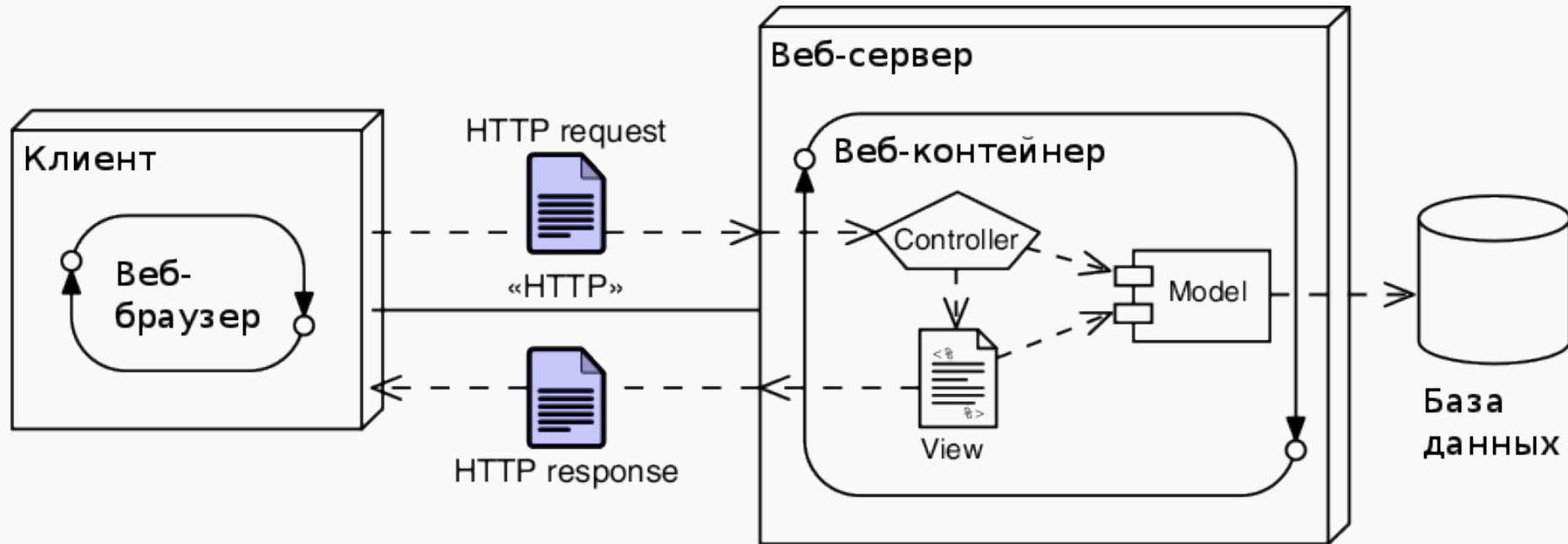


Шаблон MVC



Архитектура Model 2 (Java)

- Предназначена для проектирования достаточно сложных веб-приложений.
- За обработку и представление данных отвечают *разные* компоненты (сервлеты и JSP).



Как реализовать Model 2

- Вручную (сервлеты + какой-нибудь шаблонизатор + какая-нибудь бизнес-логика).
- Использовать фреймворк:
 - Apache Struts.
 - Apache Velocity.
 - JavaServer Faces (в составе Java EE).
 - Spring Web MVC.

3.3. Шаблонизация страниц

Зачем оно нужно

- Делать разметку страницы с помощью серверного сценария неудобно.
- Интерфейс приложения обычно состоит из типовых повторяющихся элементов.
- В больших проектах разработкой логики и интерфейсов обычно занимаются разные люди.

Инструменты шаблонизации страниц

- JavaServer Pages.
- FreeMarker Template Engine (FTL).
- Thymeleaf.
- Velocity.

...тысячи их!

3.3.1. JavaServer Pages

JavaServer Pages

- Страницы JSP — это текстовые файлы, содержащие статический HTML и JSP-элементы.
- JSP-элементы позволяют формировать динамическое содержимое.
- При загрузке в веб-контейнер страницы JSP транслируются компилятором (jasper) в сервлеты.
- Позволяют отделить бизнес-логику от уровня представления (если их комбинировать с сервлетами).

Преимущества и недостатки JSP

- Преимущества:
 - Высокая производительность — транслируются в сервлеты.
 - Не зависят от используемой платформы — код пишется на Java.
 - Позволяют использовать Java API.
 - Простые для понимания — структура похожа на обычный HTML.
- Недостатки:
 - Трудно отлаживать, если приложение целиком основано на JSP.
 - Возможны конфликты при параллельной обработке нескольких запросов.

Сервлеты и JSP

```
public class HelloServlet extends HttpServlet {  
    private static final String DEFAULT_NAME = "World";  
    public void doGet(HttpServletRequest request,  
                      HttpServletResponse response)  
        throws IOException {  
        generateResponse(request, response);  
    }  
    public void doPost(HttpServletRequest request,  
                      HttpServletResponse response)  
        throws IOException {  
        generateResponse(request, response);  
    }  
    public void generateResponse(HttpServletRequest request,  
                                HttpServletResponse response) throws  
IOException {  
    String name = request.getParameter("name");  
}
```

Сервлеты и JSP (продолжение)

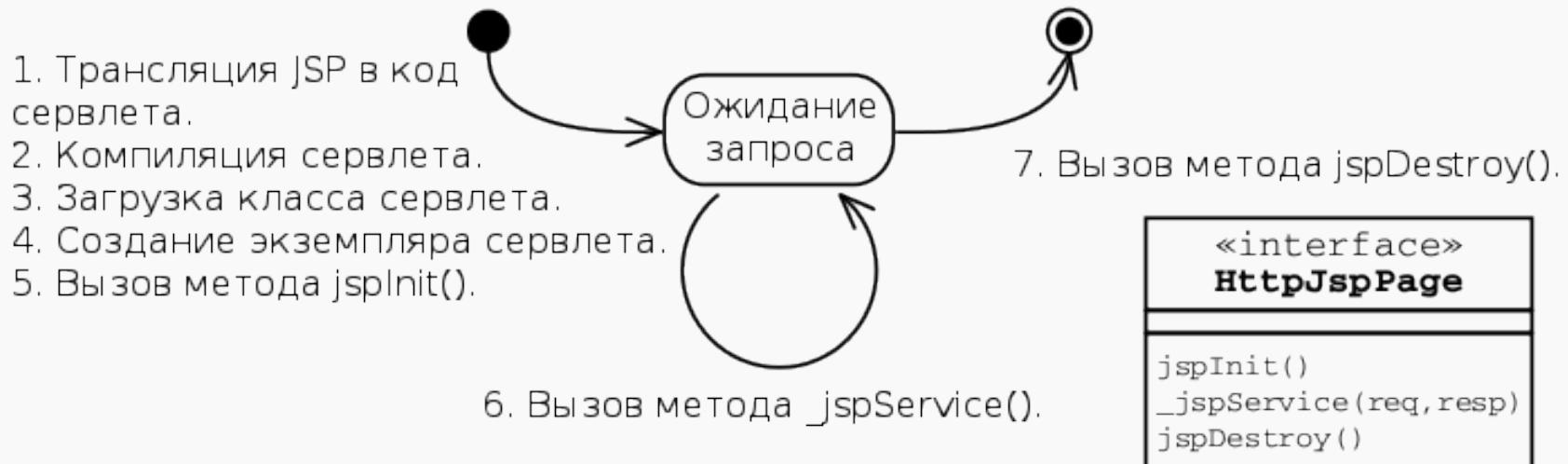
```
        if ( (name == null) || (name.length() ==  
0) ) {  
            name = DEFAULT_NAME;  
        }  
        response.setContentType("text/html");  
        PrintWriter out = response.getWriter();  
        out.println("<HTML>");  
        out.println("<HEAD>");  
        out.println("<TITLE>Hello Servlet</TITLE>");  
        out.println("</HEAD>");  
        out.println("<BODY BGCOLOR='white'>");  
        out.println("<B>Hello, " + name + "</B>");  
        out.println("</BODY>");  
        out.println("</HTML>");  
        out.close();  
    }  
}
```



Сервлеты и JSP (продолжение)

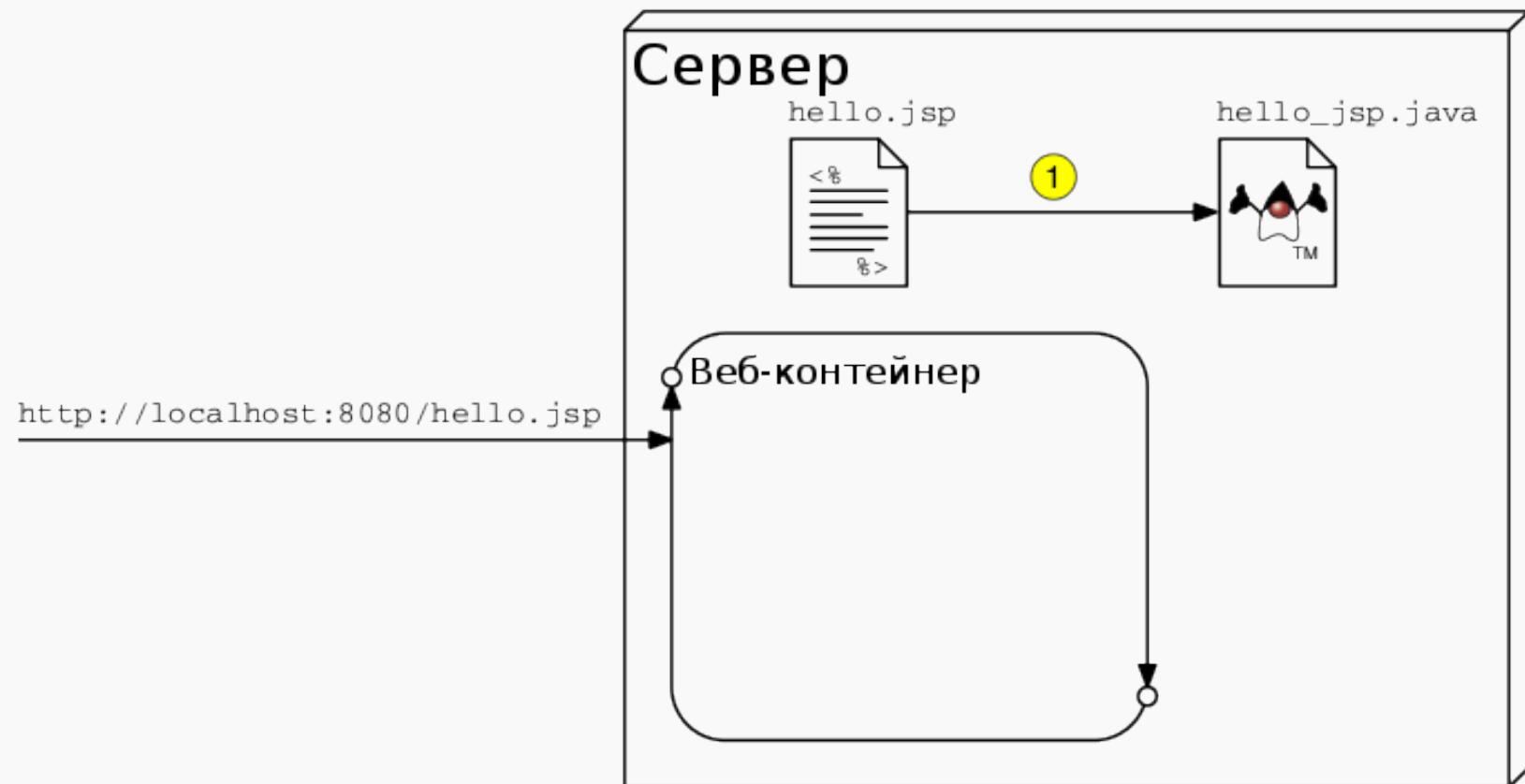
```
<%! private static final String DEFAULT_NAME = "World";%>
<html>
<head>
<title>Hello JavaServer Page</title>
</head>
<%-- Determine the specified name (or use default) --%>
<%
    String name = request.getParameter("name");
    if ( (name == null) || (name.length() == 0) ) {
        name = DEFAULT_NAME;
    }
%>
<body bgcolor='white'>
<b>Hello, <%= name %></b>
</body>
</html>
```

Жизненный цикл JSP



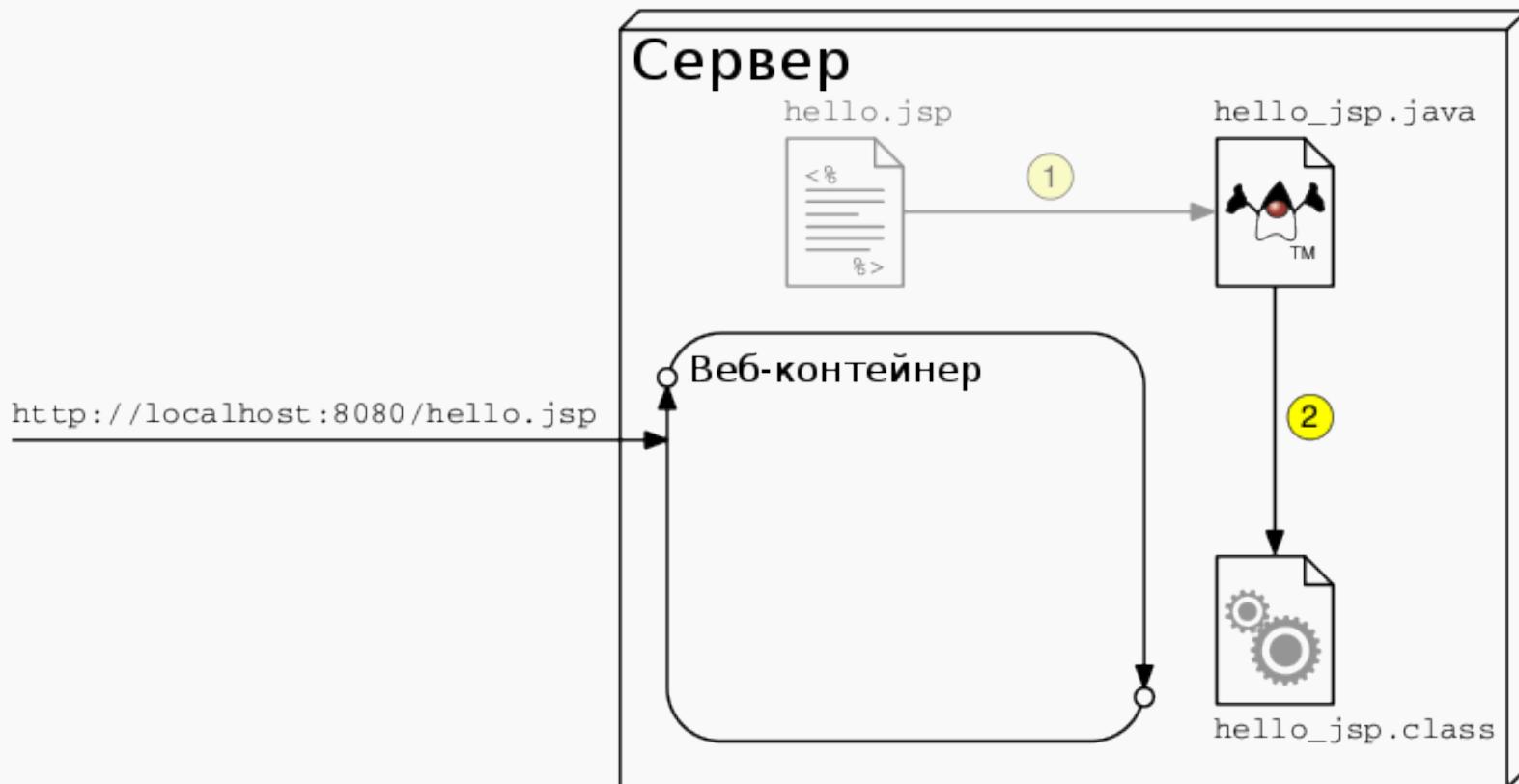
Жизненный цикл JSP (продолжение)

1. Трансляция.



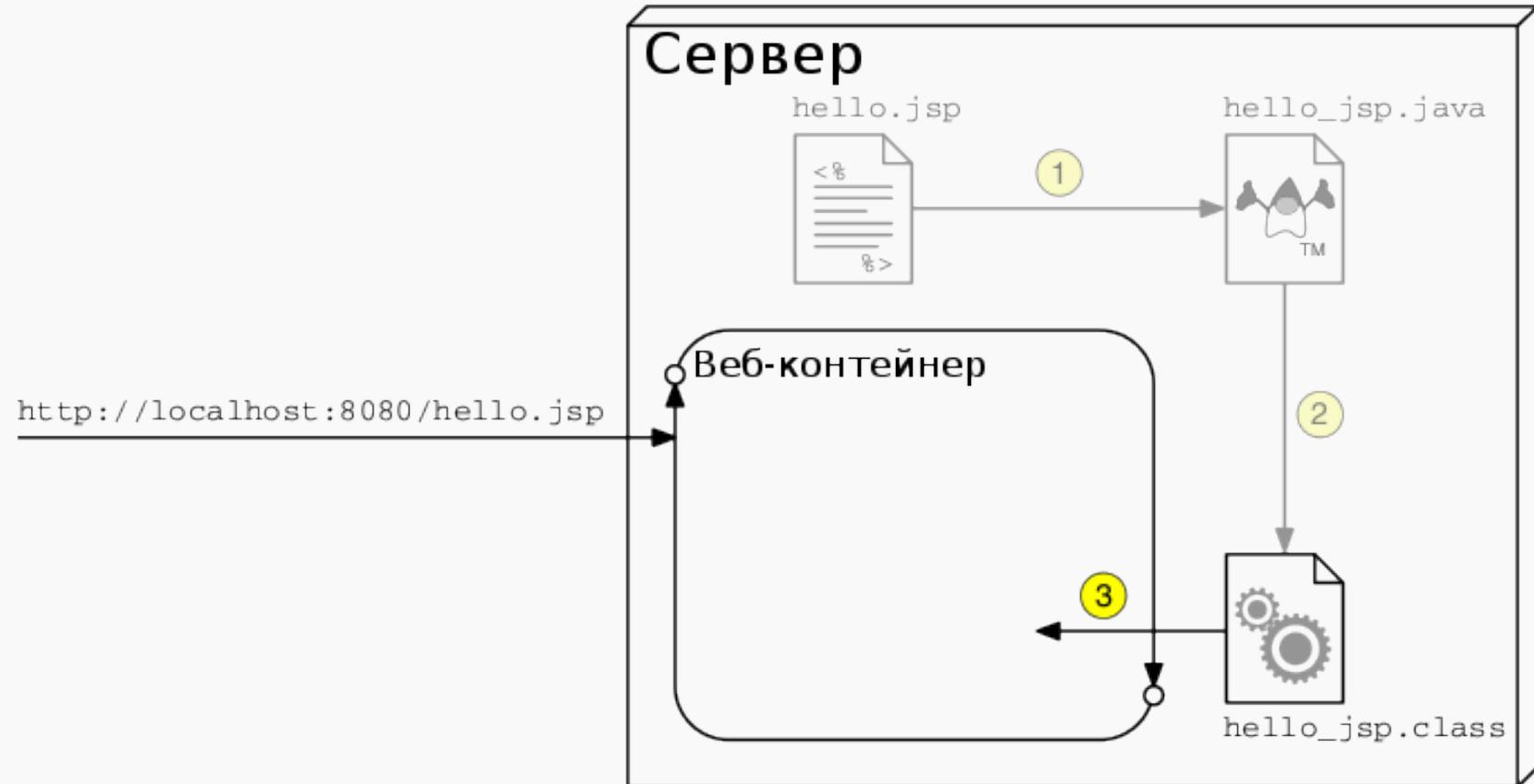
Жизненный цикл JSP (продолжение)

2. Компиляция сервлета.



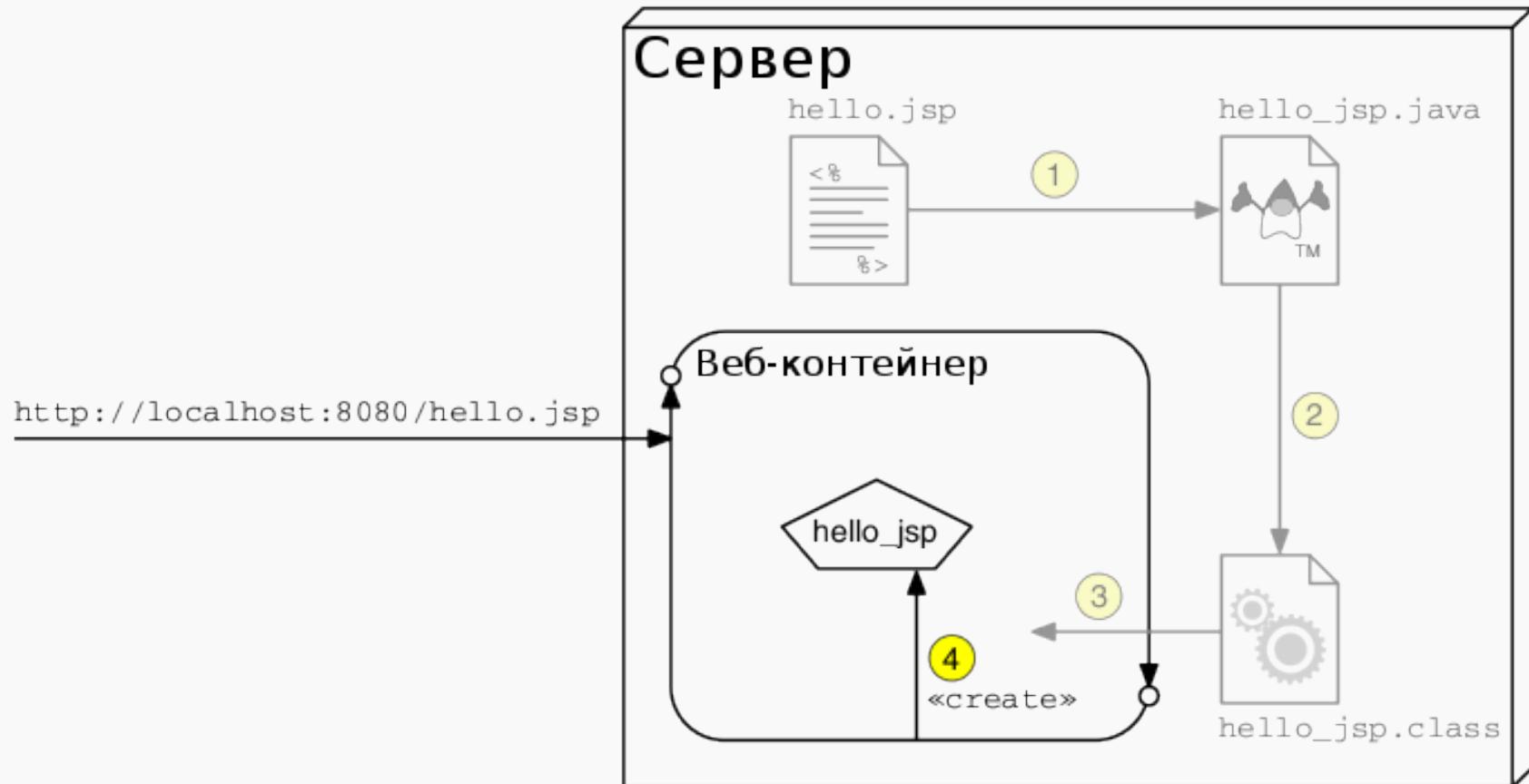
Жизненный цикл JSP (продолжение)

3. Загрузка сервлета веб-контейнером.



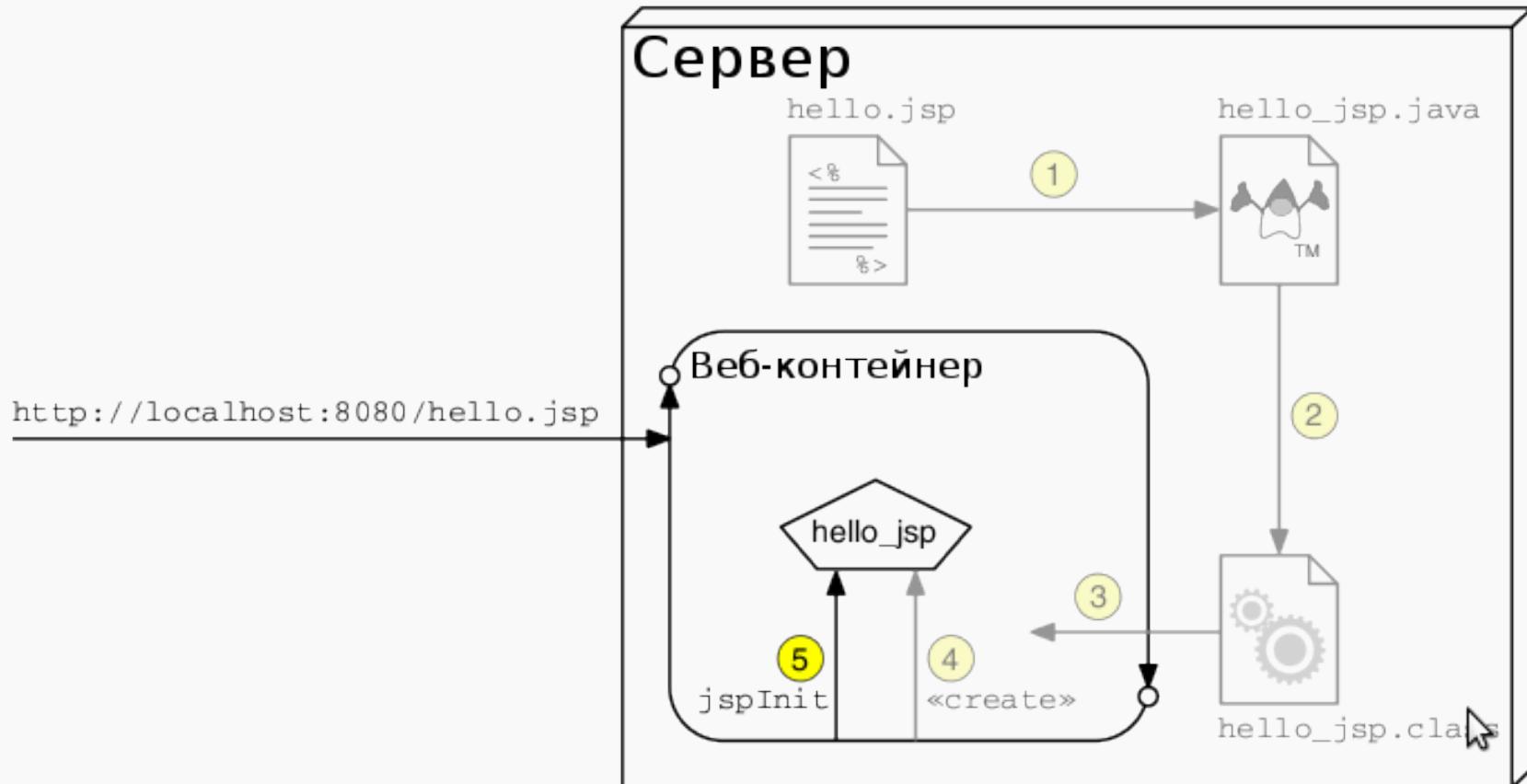
Жизненный цикл JSP (продолжение)

4. Создание веб-контейнером экземпляра сервлета.



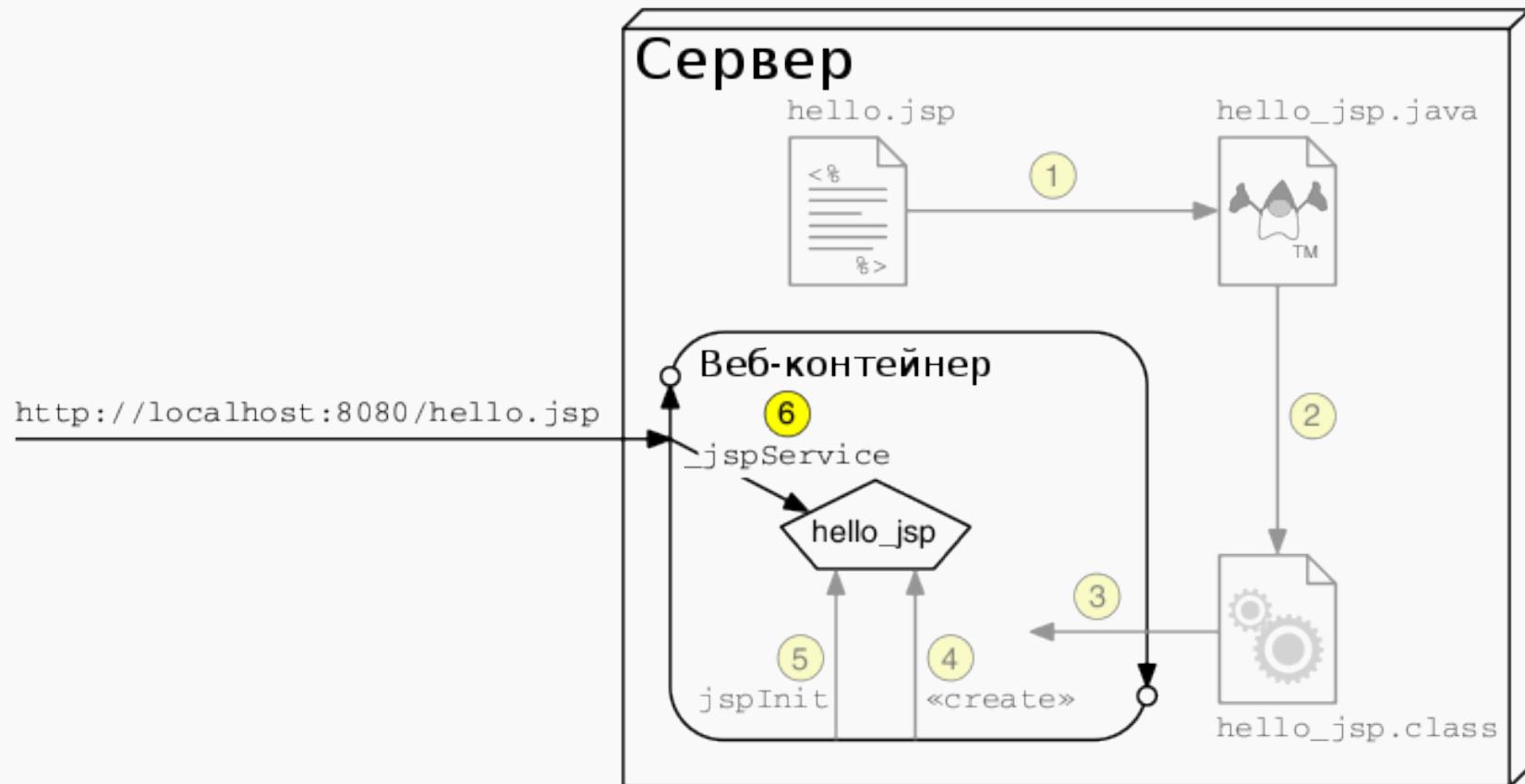
Жизненный цикл JSP (продолжение)

5. Инициализация сервлета.



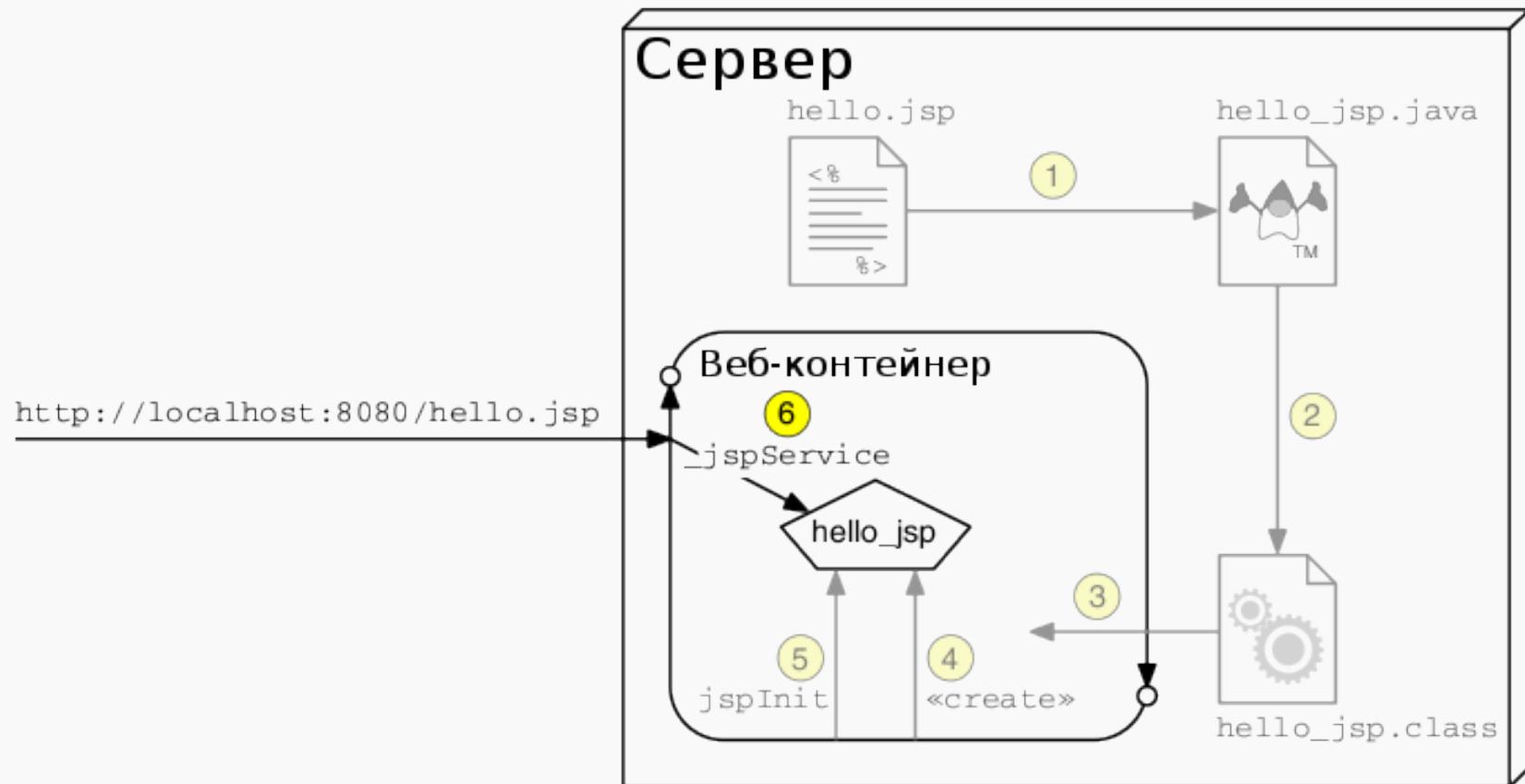
Жизненный цикл JSP (продолжение)

6. Обработка запросов.



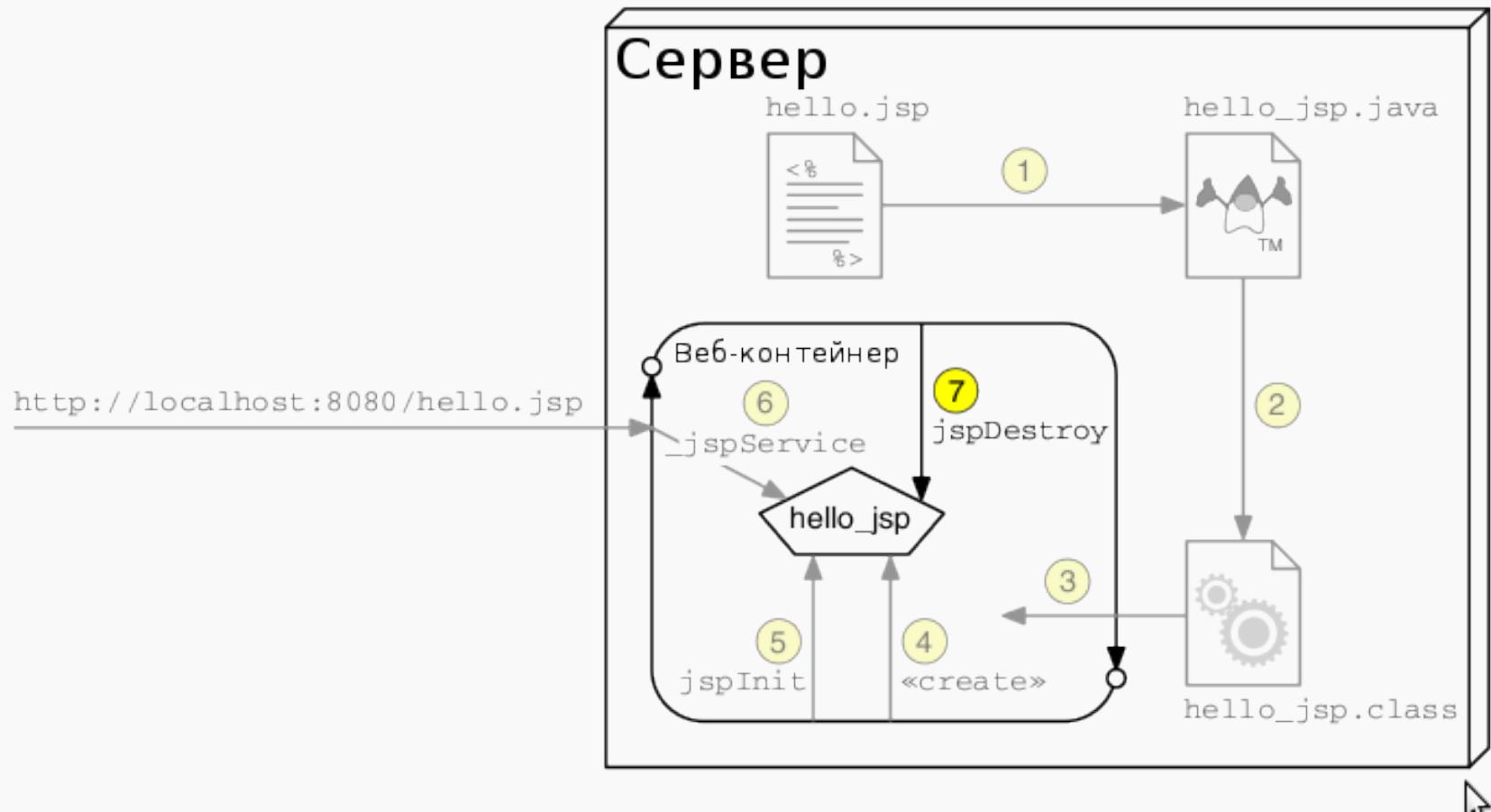
Жизненный цикл JSP (продолжение)

6. Обработка запросов.



Жизненный цикл JSP (продолжение)

7. Вызов метода jspDestroy.



JSP-элементы

- 2 варианта синтаксиса — на базе HTML и XML.
- Обозначаются тегами <% %> (HTML-вариант):
`<html>`
`<%-- scripting element --%>`
`</html>`
- Существует 5 типов JSP-элементов:
 - Комментарий — <%-- Comment --%>;
 - Директива — <%@ directive %>;
 - Объявление — <%! decl %>;
 - Скриптлет — <% code %>;
 - Выражение — <%= expr %>.

Комментарии

Поддерживаются 3 типа комментариев:

- HTML-комментарии:

```
<!-- This is an HTML comment.  
It will show up in the response. -->
```

- JSP-комментарии:

```
<%-- This is a JSP comment.  
It will only be seen in the JSP code.  
It will not show up in either the servlet code  
or the response.  
--%>
```

- Java-комментарии:

```
<%  
/* This is a Java comment.  
It will show up in the servlet code.  
It will not show up in the response. */  
%>
```

Директивы

Управляют процессом трансляции страницы в сервлет.

- Синтаксис:

```
<%@ DirectiveName [attr="value"]* %>
```

- Примеры:

```
<%@ page session="false" %>
```

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

Объявления

Позволяют объявлять поля и методы:

- Синтаксис:

```
<%! JavaClassDeclaration %>
```

- Примеры:

```
<%!
public static final String DEFAULT_NAME = "World";
%>

<%!
public String getName(HttpServletRequest request) {
    return request.getParameter("name");
}
%>

<%! int counter = 0; %>
```

Скриптлеты

Позволяют задать Java-код, который будет выполняться при обработке запросов (при вызове метода `_jspService`).

- Синтаксис:

```
<% JavaCode %>
```

- Примеры:

```
<% int i = 0; %>
```

```
<% if ( i > 10 ) { %>
```

```
    I am a big number
```

```
<% } else { %>
```

```
    I am a small number
```

```
<% } %>
```

Выражения

Позволяют вывести результат вычисления выражения.

- Синтаксис:

```
<%= JavaExpression %>
```

- Примеры:

```
<B>Ten is <%= (2 * 5) %></B>
```

Thank you, <I><%= name %></I>, for
registering for the soccer league.

The current day and time is: <%= new
java.util.Date() %>

Предопределённые переменные

В процессе трансляции контейнер добавляет в метод `_jspService` ряд объектов, которые можно использовать в скриптлетах и выражениях:

Имя переменной	Класс
<code>application</code>	<code>javax.servlet.ServletContext</code>
<code>config</code>	<code>javax.servlet.ServletConfig</code>
<code>exception</code>	<code>java.lang.Throwable</code>
<code>out</code>	<code>javax.servlet.jsp.JspWriter</code>
<code>page</code>	<code>java.lang.Object</code>
<code>PageContext</code>	<code>javax.servlet.jsp.PageContext</code>
<code>request</code>	<code>javax.servlet.ServletRequest</code>
<code>response</code>	<code>javax.servlet.ServletResponse</code>
<code>session</code>	<code>javax.servlet.http.HttpSession</code>

Предопределённые переменные (продолжение)

- `Exception` — используется только на страницах-перенаправлениях с информацией об ошибках (Error Pages).
- `Page` — API для доступа к экземпляру класса сервлета, в который транслируется JSP.
- `PageContext` — контекст JSP-страницы.

Директива *Page*

- Позволяет задавать параметры, используемые контейнером при управлении жизненным циклом страницы.
- Обычно расположена в начале страницы.
- На одной странице может быть задано несколько директив page с разными указаниями контейнеру.
- Синтаксис:
`<%@ page attribute="value" %>`

Атрибуты директивы *Page*

Атрибут	Для чего нужен
buffer	Задаёт параметры буферизации и размер буфера для потока вывода сервлета.
autoFlush	Указывает, автоматически ли выгружается содержимое буфера при его переполнении.
contentType	Позволяет задать Content Type и кодировку страницы.
errorPage	Позволяет задать страницу, на которую будет осуществлено перенаправление при возникновении Runtime Exception.
isErrorPage	Указывает, является ли текущая страница Error Page.
extends	Позволяет задать имя родительского класса, от которого будет наследоваться сервлет.

Атрибуты директивы *Page* (продолжение)

Атрибут	Для чего нужен
<code>import</code>	Импорт классов или пакетов.
<code>info</code>	Задаёт строку, которую будет возвращать метод <code>getServletInfo()</code> .
<code>isThreadSafe</code>	Если <code>isThreadSafe == false</code> , то контейнер блокирует параллельную обработку нескольких запросов страницей.
<code>language</code>	Позволяет задать язык программирования, на котором пишутся скриптовые элементы на странице (по умолчанию — Java).
<code>session</code>	Указывает контейнеру, создавать ли ему предопределённую переменную <code>session</code> .
<code>isELIgnored</code>	Указывает, вычисляются EL-выражения контейнером, или нет.
<code>isScriptingEnabled</code>	Указывает, обрабатываются ли скриптовые элементы.

JSP Actions

- XML-элементы, позволяющие управлять поведением сервлета.
- Синтаксис:
`<jsp:action_name attribute="value" />`

JSP Action	Для чего нужен
<code>jsp:include</code>	Включает в страницу внешний файл <i>во время обработки запроса</i> .
<code>jsp:useBean</code>	Добавляет на страницу экземпляр Java Bean с заданным контекстом.
<code>jsp:getProperty</code> <code>jsp:setProperty</code>	Получение и установка свойств Java Bean.
<code>jsp:forward</code>	Перенаправление на другую страницу.

Конфигурация JSP

- Задаётся в дескрипторе развёртывания (web.xml).
- Находится внутри элемента jsp-config.
- Пример:

```
<jsp-config>
    <property name="initial-capacity"
              value="1024" >
</jsp-config>
```

JSP Standard Tag Library (JSTL)

- Расширение JSP, добавляющее возможность использования дополнительных тегов, решающих типовые задачи.
- Примеры задач:
 - Условная обработка.
 - Создание циклов, вывод массивов / коллекций.
 - Поддержка интернационализации.
- Рекомендуется использовать их вместе с EL вместо скриптлетов.

Теги JSTL

```
// Основные теги создания циклов, определения
условий,
// вывода информации на страницу и т. д.
<%@ taglib prefix="c"
    uri="http://java.sun.com/jsp/jstl/core" %>

// Теги для работы с XML-документами
<%@ taglib prefix="x"
    uri="http://java.sun.com/jsp/jstl/xml" %>

// Теги для работы с базами данных
<%@ taglib prefix="s"
    uri="http://java.sun.com/jsp/jstl/sql" %>

// Теги для форматирования и интернационализации
// информации (i10n и i18n)
<%@ taglib prefix="f"
    uri="http://java.sun.com/jsp/jstl/fmt" %>
```

JSP Expression Language (EL)

- Расширение JSP, позволяющее удобно работать с JavaBeans-компонентами без написания кода на Java.
- Позволяет использовать на страницах арифметические и логические выражения.
- Поддерживается «из коробки», можно отключить в настройках конкретной страницы и / или приложения.
- Пример использования:

```
<jsp:text>
    Box Perimeter is:
    ${2*box.width + 2*box.height}
</jsp:text>
```

Пример использования JSTL + EL

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jsp/jstl/core" %>
<html>
    <head>
        <title>Пример тега <c:if> библиотеки
JSTL</title>
    </head>
    <body>
        <c:set var="salary" scope="session"
            value="${23400*2}" />
        <c:if test="${salary > 45000}">
            <p>Salary = <c:out value="${salary}" /></p>
        </c:if>
    </body>
</html>
```

3.3.2. FreeMarker Template Engine

Apache FreeMarker

- Компилирующий обработчик шаблонов.
- Написан на Java.
- Разработчик – Apache Software Foundation, первая версия вышла в 2000 г.
- Свободное ПО, распространяется по лицензии BSD.

Особенности и возможности

Поддерживает «джентльменский набор» возможностей по созданию шаблонов:

- условия;
- циклы;
- присваивание значений переменным;
- арифметические операции;
- операции со строками;
- инструменты форматирования;
- макросы и функции;
- подключение внешних шаблонов;
- экранирование символов.

Пример шаблона

```
<html>
<head>
    <title>Welcome!</title>
</head>
<body>
    <h1>Welcome ${user}!</h1>
    <p>Our latest product:<br/>
    <a href="${latestProduct.url}">
        ${latestProduct.name}
    </a>!
</body>
</html>
```

Синтаксис шаблонов

Шаблон может содержать следующие элементы:

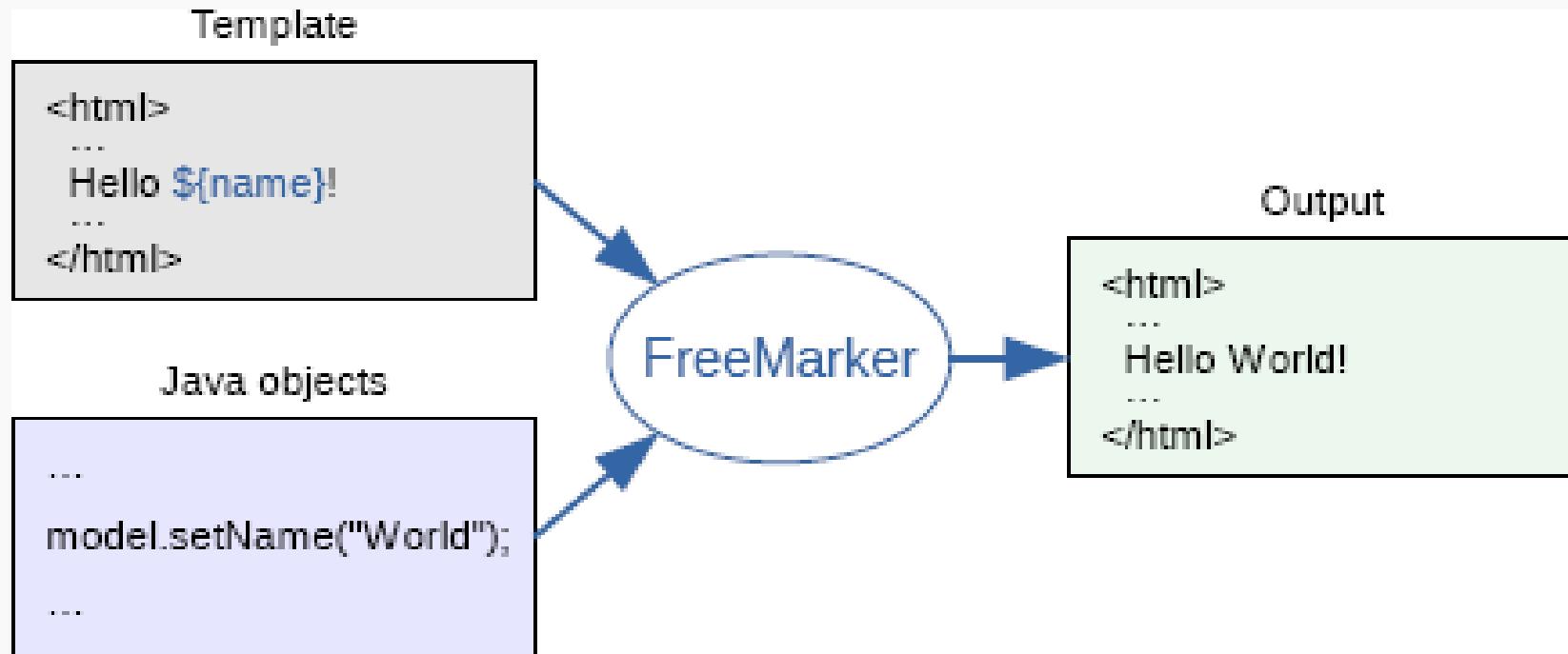
- Статический HTML.
- Обращения к модели данных:

Welcome **\${user}** !

- Директивы:
<#if animals.python.price != 0>
 Pythons are not free today!
</#if>

- Вызовы встроенных функций:
animals?filter(it -> it.protected)

Принцип работы FreeMarker



Модель данных FTL (FTL data-model)

- Древовидная объектная структура, данные из которой шаблон использует при формировании HTML.
- Элементы дерева – Java Beans.
- Сложность иерархии может быть любой.
- При выводе в HTML все объекты преобразуются в строки.

Пример модели данных

```
(root)
|
+- user = "Big Joe"
+- latestProduct
  |
  +- url = "products/greenmouse.html"
  +- name = "green mouse"

// «Корневой» объект. Может быть JavaBean, может быть Map.
Map<String, Object> root = new HashMap<>();

// Кладём строку "user" в «корневой» объект
root.put("user", "Big Joe");

// Создаём объект "latestProduct". Здесь использован Java Bean,
// но Мар'ы можно также вкладывать друг в друга.
Product latest = new Product();
latest.setUrl("products/greenmouse.html");
latest.setName("green mouse");
// Кладём "latestProduct" в «корневой» объект.
root.put("latestProduct", latest);
```

Использование в проектах

- 1) Скачиваем freemarker.jar.
- 2) Создаём конфигурацию.
- 3) Создаём шаблон.
- 4) Создаём модель данных.
- 5) Компилируем шаблон с нужными данными:

```
Writer out =  
    new OutputStreamWriter(System.out);  
temp.process(root, out);
```

3.3.3. Thymeleaf

Thymeleaf

- Компилирующий обработчик шаблонов.
- Универсальный – может использоваться как в веб-приложениях, так и для решения общих задач шаблонизации чего-либо.
- Написан на Java.
- Свободное ПО, распространяется по лицензии Apache 2.0.
- Интеграция «из коробки» со Spring Framework.

Особенности Thymeleaf

- «Выходной» формат – XML, XHTML и HTML5. Также поддерживаются JS, CSS и Plain Text.
- Не привязан к Servlet API – может работать как «онлайн», так и «оффлайн».
- Построен на модульной системе. Модули называются *диалектами (dialects)*.
- Поддержка возможностей i18n и l10n.
- Есть встроенный кеш скомпилированных шаблонов.

Процессоры и диалекты

- Thymeleaf – модульный движок. Модуль Thymeleaf называется *диалектом (dialect)*.
- Диалект состоит из одного или нескольких *процессоров (processor)*.
- Процессор – объект, который применяет некоторую логику к формируемому на основе шаблона артефакту.
- «Из коробки» Thymeleaf содержит *стандартный диалект (Standard Dialect)*, которого достаточно для решения большинства типовых задач.

Стандартный диалект

- Содержит набор процессоров, предназначенных для решения типовых задач.
- Большая часть процессоров стандартного диалекта – *процессоры атрибутов (Attribute Processors)*.
- Процессоры атрибутов обрабатывают дополнительные («нестандартные») атрибуты тегов:

```
<input type="text" name="userName"
       value="James Carrot"
       th:value="${user.name}" />
```
- Благодаря этому шаблоны Thymeleaf обычно можно тестировать в браузере – он просто игнорирует нестандартные атрибуты.

Выражения

- Значения атрибутов присваиваются путём вычисления выражений (*expressions*).
- Выражения, поддерживаемые стандартным диалектом, называются *стандартными выражениями*.
- В стандартном диалекте Thymeleaf реализована поддержка 5 видов выражений:
 - \${...} – Variable expressions.
 - *\${...} – Selection expressions.
 - #\${...} – Message (i18n) expressions.
 - @{...} – Link (URL) expressions.
 - ~\${...} – Fragment expressions.

Пример шаблона

```
<table>
  <thead>
    <tr>
      <th th:text="#{msgs.headers.name}">Name</th>
      <th th:text="#{msgs.headers.price}">Price</th>
    </tr>
  </thead>
  <tbody>
    <tr th:each="prod : ${allProducts}">
      <td th:text="${prod.name}">Oranges</td>
      <td th:text="$
{#numbers.formatDecimal(prod.price,1,2)}">
        0.99
      </td>
    </tr>
  </tbody>
</table>
```

Variable Expressions

- Выражения на языке OGNL (Object Graph Navigation Language).
- Позволяют обращаться к переменной из контекста Thymeleaf.
- В случае использования совместно со Spring – позволяют обращаться к атрибутам модели:

```
<span th:text="${book.author.name}">  
    ( (Book)context.getVariable("book") )  
        .getAuthor()  
        .getName()
```

- Могут использоваться с более сложными конструкциями (условные выражения, перечисления и т. д.):

```
<li th:each="book : ${books}">
```

Selection Expressions

Позволяют обратиться к выбранному ранее объекту вместо обращения к контексту.

```
<div th:object="${book}">  
    ...  
    <span th:text="*{title}">...</span>  
    ...  
</div>  
{  
    final Book selection =  
        (Book) context.getVariable("book");  
    output(selection.getTitle());  
}
```

Message (i18n) Expressions

- Позволяют обращаться к сообщениям из файлов локализации (.properties):

```
#{{main.title}}
```

- Помимо обращения по ключу, можно использовать параметры, в т.ч., вычисленные с помощью Variable Expressions:

```
#{{message.entrycreated(${entryId})}}
```

- Могут использоваться в любых элементах, предполагающих вывод локализованного текста:

```
<table>
```

```
  ...
  <th th:text="{{header.address.city}}">...</th>
  <th th:text="{{header.address.country}}">...</th>
  ...
</table>
```

- В сложных случаях могут целиком вычисляться с помощью Variable Expressions:

```
#${${config.adminWelcomeKey}(${session.user.name})}
```

Link (URL) Expressions

- Позволяют формировать URL внутри контекста приложения:

```
<a th:href="@{/order/list}">...</a>  
  
<a href="/myapp/order/list;jsessionid=23fa31abd41ea093">  
    ...  
</a>
```

- Так же могут принимать аргументы:

```
<a th:href="@{/order/details(id=${orderId}, type=$  
{orderType})}">...</a>  
  
<a href="/myapp/order/details?id=23&type=online">...</a>
```

- Могут быть относительными (внутри контекста приложения), привязанными к контексту сервера, привязанными к протоколу или абсолютными:

```
<a th:href="@{./documents/report}">...</a>  
  
<a th:href="@{~/contents/main}">...</a>  
  
<a th:href="@{//static.mycompany.com/res/initial}">...</a>  
  
<a th:href="@{http://www.mycompany.com/main}">...</a>
```

Fragment Expressions

Позволяют компоновать шаблоны из фрагментов:

```
<div th:insert="~{commons :: main}">  
    ...  
</div>
```

Фрагменты могут использоваться несколько раз, передаваться другим шаблонам в качестве аргументов и т.д.:

```
<div th:with="frag=~{footer :: #main/text()}">  
    <p th:insert="${frag}">  
</div>
```

Использование в приложениях

Необходимо проинициализировать объекты TemplateEngine и TemplateResolver:

```
...
private final TemplateEngine templateEngine;
...

public GTVGAApplication(final ServletContext servletContext) {
    super();

    ServletContextTemplateResolver templateResolver =
        new ServletContextTemplateResolver(servletContext);

    // HTML is the default mode, but we set it anyway for better understanding of code
    templateResolver.setTemplateMode(TemplateMode.HTML);
    // This will convert "home" to "/WEB-INF/templates/home.html"
    templateResolver.setPrefix("/WEB-INF/templates/");
    templateResolver.setSuffix(".html");
    // Template cache TTL=1h. If not set, entries would be cached until expelled
    templateResolver.setCacheTTLMs(Long.valueOf(3600000L));

    // Cache is set to true by default. Set to false if you want templates to
    // be automatically updated when modified.
    templateResolver.setCacheable(true);

    this.templateEngine = new TemplateEngine();
    this.templateEngine.setTemplateResolver(templateResolver);

    ...
}
```

Использование в приложениях на базе Spring

- Есть интеграция «из коробки» – библиотеки `thymeleaf-spring3` и `thymeleaf-spring4`.
- Эти библиотеки позволяют использовать Thymeleaf-шаблоны вместо JSP в приложениях на базе Spring.
- Для интеграции в состав Spring-приложений реализован специальный диалект Thymeleaf – `SpringStandard`.

4. Rich Internet Applications

Концепция RIA

- RIA – интернет-приложения, предназначенные для выполнения задач, обычно выполняемых десктопными приложениями.
- Впервые термин упомянут компанией Macromedia в 2002 г.
- Изначально применялся к «тяжёлым» технологиям а-ля Flash и Silverlight, требовавшим для работы отдельные плагины.
- В настоящее время термин достаточно «размыт», но чаще всего применяется к компонентно-ориентированным server-side веб-фреймворкам (gwt, JSF).

Преимущества и недостатки RIA

Преимущества:

- Приложения пишутся в стиле, похожем на стиль написания десктопных приложений.
- Высокий уровень абстракции => меньше «рутинной» работы программиста.
- Много готовых компонентов.

Недостатки:

- Высокий уровень абстракции => тяжелее «спуститься» на уровень протокола (а это часто бывает надо!)
- Сложность архитектуры фреймворков => сложно сделать что-то, не предусмотренное их разработчиками.
- Идеология «ломает» парадигму веб-страниц.

4.1. JavaServer Faces

JavaServer Faces

- JSF — фреймворк для разработки веб-приложений.
- Входит в состав платформы Java EE.
- Основан на использовании *компонентов*.
- Для отображения данных используются JSP или XML-шаблоны (*facelets*).

Достоинства JSF

- Чёткое разделение бизнес-логики и интерфейса (фреймворк реализует шаблон MVC).
- Управление обменом данными на уровне компонент.
- Простая работа с событиями на стороне сервера.
- Доступность нескольких реализаций от различных компаний-разработчиков.
- Расширяемость (можно использовать дополнительные наборы компонентов).
- Широкая поддержка со стороны интегрированных средств разработки (IDE).

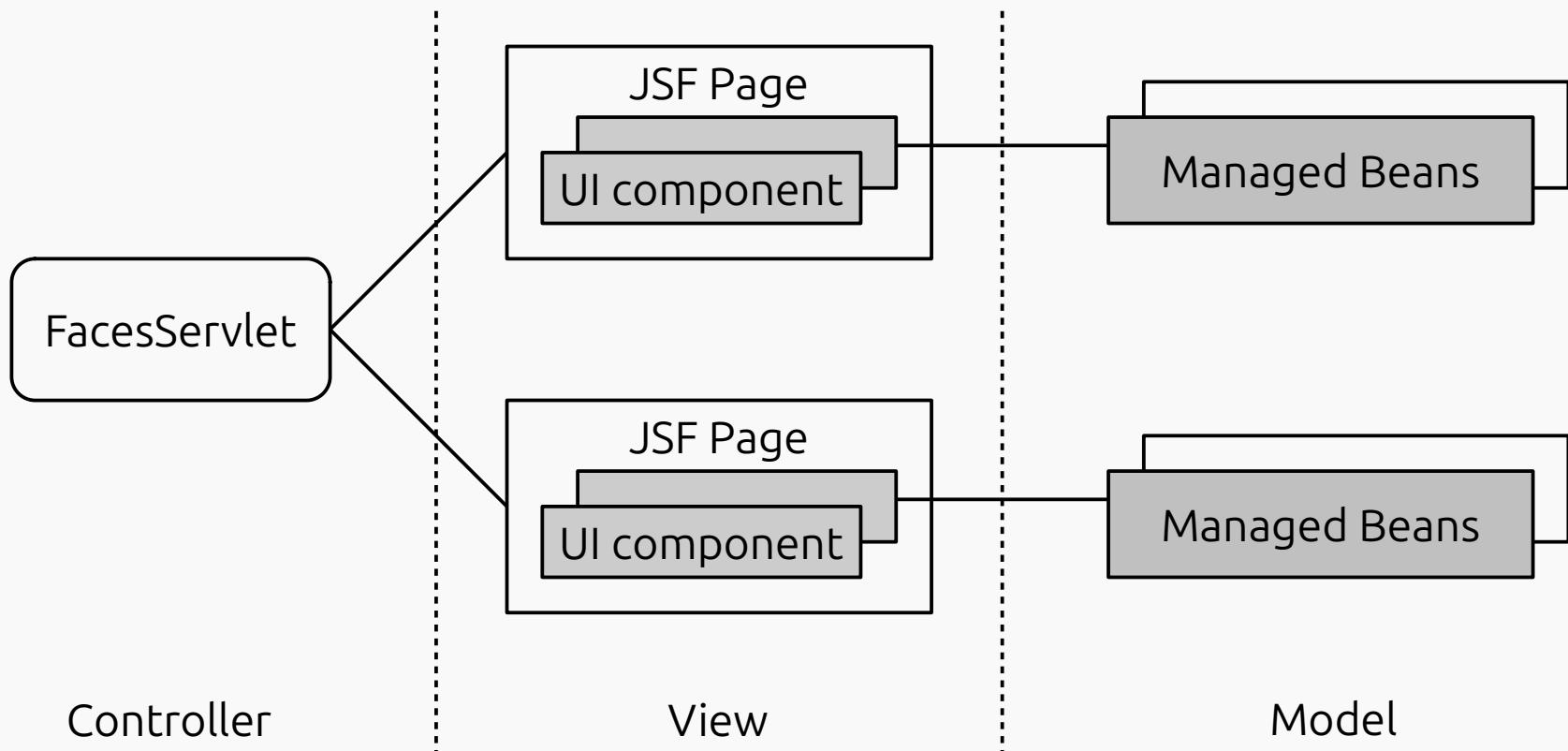
Недостатки JSF

- Высокоуровневый фреймворк — сложно реализовывать не предусмотренную авторами функциональность.
- Сложности с обработкой GET-запросов (устранены в JSF 2.0).
- Сложность разработки собственных компонентов.

Структура JSF-приложения

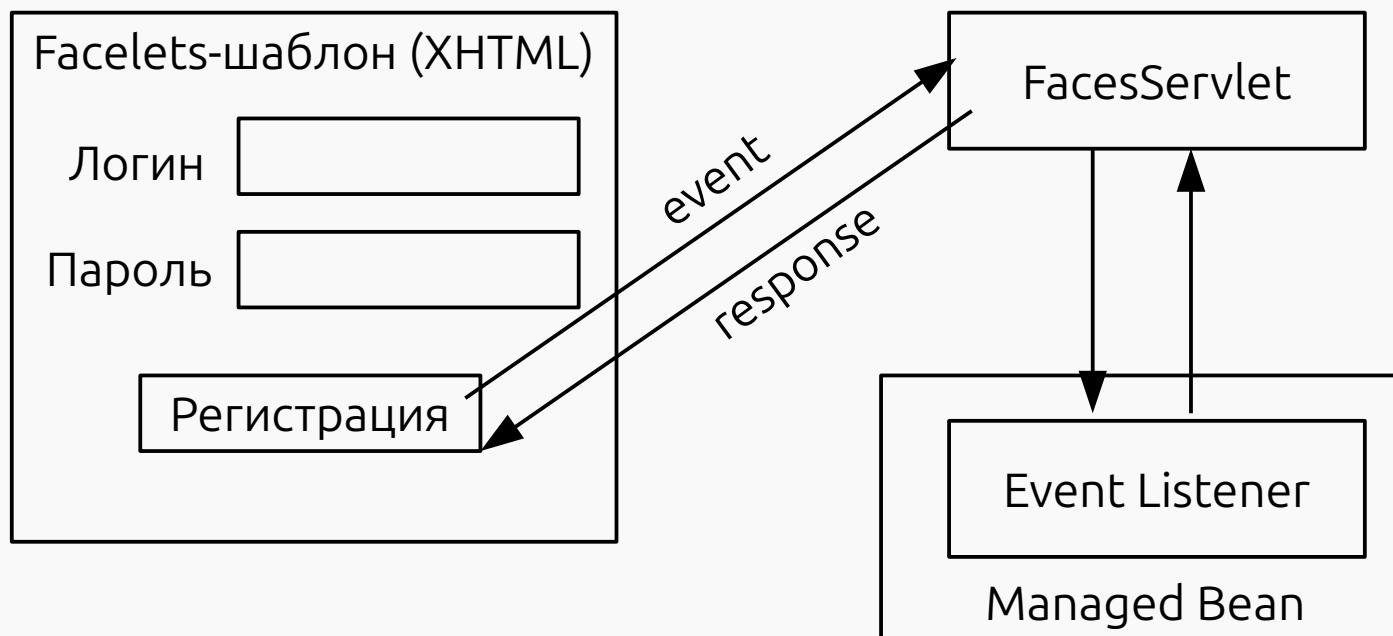
- JSP или XHTML-страницы, содержащие компоненты GUI.
- Библиотеки тегов.
- Управляемые бины.
- Дополнительные объекты (компоненты, конвертеры и валидаторы).
- Дополнительные теги.
- Конфигурация — faces-config.xml (опционально).
- Дескриптор развёртывания — web.xml.

MVC-модель JSF



FacesServlet

- Обрабатывает запросы с браузера.
- Формирует объекты-события и вызывает методы-слушатели.



Конфигурация FacesServlet

Конфигурация задаётся в web.xml:

```
<!-- Faces Servlet -->
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>
        javax.faces.webapp.FacesServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<!-- Faces Servlet Mapping -->
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

Страницы и компоненты UI

- Интерфейс строится из компонентов.
- Компоненты расположены на Facelets-шаблонах или страницах JSP.
- Компоненты реализуют интерфейс `javax.faces.component.UIComponent`.
- Можно создавать собственные компоненты.
- Компоненты на странице объединены в древовидную структуру — *представление*.
- Корневым элементом представления является экземпляр класса `javax.faces.component.UIViewRoot`.

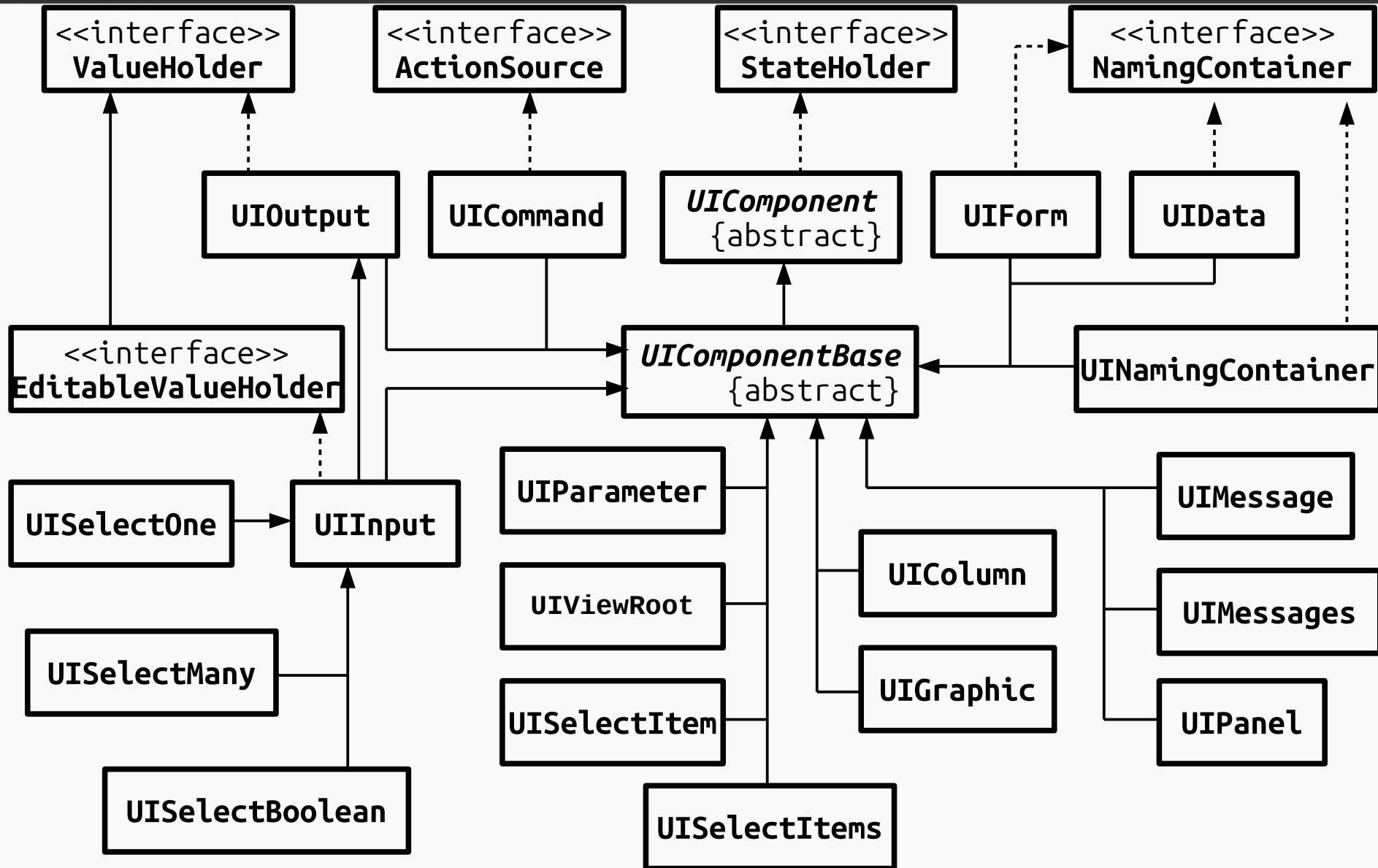


Пример страницы JSF (Facelets)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:body>
    <h3>JSF 2.0 + Ajax Hello World Example</h3>
    <h:form>
      <h:inputText id="name"
value="#{helloBean.name}"></h:inputText>
      <h:commandButton value="Welcome Me">
        <f:ajax execute="name" render="output" />
      </h:commandButton>
      <h2>
        <h:outputText id="output"
value="#{helloBean.sayWelcome}" />
      </h2>
    </h:form>
  </h:body>
</html>
```



Иерархия компонентов JSF



Навигация между страницами JSF

- Реализуется экземплярами класса NavigationHandler.
- Правила задаются в файле faces-config.xml:

```
<navigation-rule>
    <from-view-id>/pages/inputname.xhtml</from-view-
id>
    <navigation-case>
        <from-outcome>sayHello</from-outcome>
        <to-view-id>/pages/greeting.xhtml</to-view-id>
    </navigation-case>
    <navigation-case>
        <to-view-id>/pages/goodbye.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```
- Пример перенаправления на другую страницу:

```
<h:commandButton id="submit"
    action="sayHello" value="Submit" />
```

Управляемые бины

- Содержат параметры и методы для обработки данных с компонентов.
- Используются для обработки событий UI и валидации данных.
- Жизненным циклом управляет JSF Runtime Environment.
- Доступ из JSF-страниц осуществляется с помощью элементов EL.
- Конфигурация задаётся в faces-config.xml (JSF 1.X), либо с помощью аннотаций (JSF 2.0).
- Вместо них могут использоваться CDI-бины, EJB или бины Spring.

Пример управляемого бина

```
package org.itmo.sample;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import java.io.Serializable;

@ManagedBean
@SessionScoped
public class HelloBean implements Serializable {

    private static final long serialVersionUID = 1L;
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSayWelcome(){
        if("".equals(name) || name == null){ //check if null?
            return "";
        }else{
            return "Ajax message : Welcome " + name;
        }
    }
}
```

Контекст (scope) управляемых бинов

- Задаётся через faces-config.xml или с помощью аннотаций.
- 6 вариантов конфигурации:
 - @NoneScoped — контекст не определён, жизненным циклом управляют другие бины.
 - @RequestScoped (применяется по умолчанию) — контекст — запрос.
 - @ViewScoped (JSF 2.0) — контекст — страница.
 - @SessionScoped — контекст — сессия.
 - @ApplicationScoped — контекст — приложение.
 - @CustomScoped (JSF 2.0) — бин сохраняется в Map; программист сам управляет его жизненным циклом.

Конфигурация управляемых бинов

Способ 1 — через faces-config.xml:

```
<managed-bean>
    <managed-bean-name>customer</managed-bean-name>
    <managed-bean-class>CustomerBean</managed-bean-
class>
    <managed-bean-scope>request</managed-bean-scope>
    <managed-property>
        <property-name>areaCode</property-name>
        <value>#{initParam.defaultAreaCode}</value>
    </managed-property>
</managed-bean>
```

Конфигурация управляемых бинов (продолжение)

Способ 2 (JSF 2.0) — с помощью аннотаций:

```
@ManagedBean(name="customer")
@RequestScoped
public class CustomerBean {
    ...
    @ManagedProperty(value="#{initParam.defaultAreaCode}")
    name="areaCode")
    private String areaCode;
    ...
}
```

Доступ к управляемым бинам со страниц приложения

Осуществляется с помощью EL выражений:

```
...
<h:inputText value="#{user.name}"
              validator="#{user.validate}" />
...
<h:inputText binding="#{user.nameField}" />
...
<h:commandButton action="#{user.save}"
                  value="Save" />
...
```

CDI-бины

- Универсальные компоненты уровня бизнес-логики.
- Появились в Java EE 6, копируют концепции, реализованные в Spring.
- Общая идея – «отвязаться» от конкретного фреймворка при создании бизнес-логики внутри приложения.
- В большинстве случаев их можно использовать вместо JSF Managed Beans и EJB.
- По реализации очень похожи на JSF Managed Beans.

Использование CDI-бинов

```
@Named("bb")
@SessionScoped
public class BookBean {
    { ... }
}

@Path("/book")
public class BookEndpoint {
    @Inject
    private @Named("bb") BookBean bookBean;

    @GET
    public List<Book> getAllBooks() { ... }
}
```

Enterprise Java Beans

- Компоненты уровня бизнес-логики для «кровавого энтерпрайза».
- Два основных вида – Session Beans и Message-driven Beans.
- Session Beans похожи на JSF Managed Beans и CDI-бины, но обеспечивают много дополнительных возможностей.
- Session Beans могут быть «прозрачно» использованы в JSF.
- Message-driven Beans предназначены для асинхронного выполнения задач и в JSF не используются.

Пример Stateless Session Bean

```
package converter.ejb;

import java.math.BigDecimal;
import javax.ejb.*;

@Stateless
public class ConverterBean {
    private BigDecimal yenRate = new BigDecimal("83.0602");
    private BigDecimal euroRate = new BigDecimal("0.0093016");

    public BigDecimal dollarToYen(BigDecimal dollars) {
        BigDecimal result = dollars.multiply(yenRate);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }

    public BigDecimal yenToEuro(BigDecimal yen) {
        BigDecimal result = yen.multiply(euroRate);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }
}
```

Конвертеры данных

- Используются для преобразования данных компонента в заданный формат (дата, число и т. д.).
- Реализуют интерфейс `javax.faces.convert.Converter`.
- Существуют стандартные конвертеры для основных типов данных.
- Можно создавать собственные конвертеры.

Назначение конвертеров

- Автоматическое (на основании типа данных):
`<h:inputText value="#{user.age}" />`
- С помощью атрибута converter:
`<h:inputText
converter="#{javax.faces.DateTime}" />`
- С помощью вложенного тега:
`<h:outputText value="#{user.birthDay}">
 <f:converter
 converterId="#{javax.faces.DateTime}" />
</h:outputText>`

Валидация данных JSF-компонентов

- Осуществляется перед обновлением значения компонента на уровне модели.
- Класс, осуществляющий валидацию, должен реализовывать интерфейс `javax.faces.validator.Validator`.
- Существуют стандартные валидаторы для основных типов данных.
- Можно создавать собственные валидаторы.

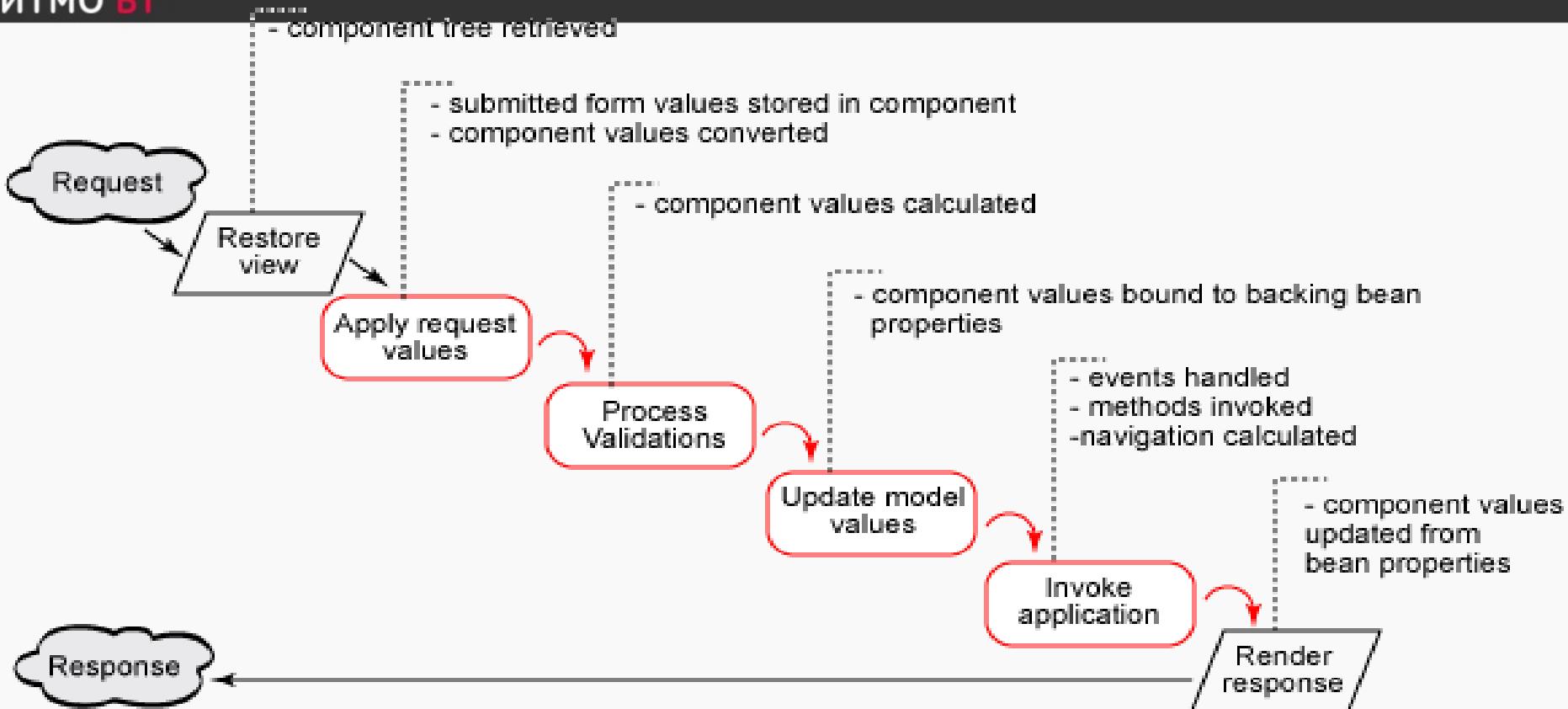
Способы валидации данных

- С помощью параметров компонента:

```
<h:inputText id="zip" size="10"
              value="#{customerBean.zip}"
              required="true">
</h:inputText>
<h:message for="zip"/>
```
- С помощью вложенного тега:

```
<h:inputText id="quantity" size="4"
              value="#{item.quantity}">
  <f:validateLongRange minimum="1"/>
</h:inputText>
<h:message for="quantity"/>
```
- С помощью логики на уровне управляемого бина.

Обработка событий



	Legend
	= System-level phases
	= Application-level phases
	= Process events <ul style="list-style-type: none"> - <code>FacesContext.renderResponse()</code> advances to Render Response phase - <code>FacesContext.responseComplete()</code> ends JSF lifecycle

Фаза формирования представления (Restore View Phase)

- JSF Runtime формирует представление (начиная с UIViewRoot):
 - Создаются объекты компонентов.
 - Назначаются слушатели событий, конвертеры и валидаторы.
 - Все элементы представления помещаются в FacesContext.
- Если это первый запрос пользователя к странице JSF, то формируется пустое представление.
- Если это запрос к уже существующей странице, то JSF Runtime синхронизирует состояние компонентов представления с клиентом.

Фаза получения значений компонентов (Apply Request Values Phase)

- На стороне клиента все значения хранятся в строковом формате — нужна проверка их корректности:
 - Вызывается конвертер в соответствии с типом данных значения.
- Если конвертация заканчивается успешно, значение сохраняется в локальной переменной компонента.
- Если конвертация заканчивается неудачно, создаётся сообщение об ошибке, которое помещается в FacesContext.

Фаза валидации значений компонентов (Process Validations Phase)

- Вызываются валидаторы, зарегистрированные для компонентов представления.
- Если значение компонента не проходит валидацию, формируется сообщение об ошибке, которое сохраняется в FacesContext.

Фаза обновления значений компонентов (Update Model Values Phase)

- Если данные валидны, то значение компонента обновляется.
- Новое значение присваивается *полю* объекта компонента.

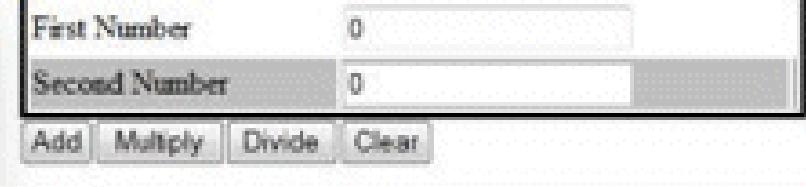
Фаза вызова приложения (Invoke Application Phase)

- Управление передаётся слушателям событий.
- Формируются новые значения компонентов.

Фаза формирования ответа сервера (Render Response Phase)

- JSF Runtime обновляет представление в соответствии с результатами обработки запроса.
- Если это первый запрос к странице, то компоненты помещаются в иерархию представления.
- Формируется ответ сервера на запрос.
- На стороне клиента происходит обновление страницы.

Пример JSF-приложения

Model	Controller	View
Calculator	Calculator controller	<p>Calculator 3rd Example</p> <p>Results cleared</p> 
- JSF-независимый слой - POJO - Бизнес логика	- JSF-зависимый слой - Выбор представления - Управление компонентами - Управление сообщениями	- Никакой логики - Никаких сравнений - Стили + компоненты GUI



Пример JSF-приложения (продолжение)

Calculator 3rd Example

First Number not a number

Second Number required

Add Multiply Divide Clear

Обрабатывает ошибки
и валидирует входные данные.
Выводит сообщения об ошибках
красным шрифтом рядом с полями ввода

Calculator 3rd Example

Added successfully

First Number

Second Number

Add Multiply Divide Clear

Results

First Number 5
Second Number 5
Result 10

/by zero

First Number

Second Number

Add Multiply Divide Clear

Исправляет ошибку деления на ноль,
присваивая делителю значение 1
и выводя соответствующее сообщение.

Позволяет производить операции
сложения, умножения, деления,
а также сбрасывать результат.
Результат выводится только после
выполнения операции

Пример JSF-приложения (продолжение)

Конфигурация web.xml:

```
...
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
...
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
...
...
```

Пример JSF-приложения (продолжение)

```
package org.itmo.sample;

public class Calculator {

    /** Первый operand */
    private int firstNumber = 0;

    /** Результат операции */
    private int result = 0;

    /** Второй operand */
    private int secondNumber = 0;

    /** Сложение operandов */
    public void add() {
        result = firstNumber + secondNumber;
    }

    /** Перемножение operandов */
    public void multiply() {
        result = firstNumber * secondNumber;
    }

    /** Сброс результата */
    public void clear() {
        result = 0;
    }
}
```

Пример JSF-приложения (продолжение)

```
/* ----- свойства ----- */  
  
public int getFirstNumber() {  
    return firstNumber;  
}  
  
public void setFirstNumber(int firstNumber) {  
    this.firstNumber = firstNumber;  
}  
  
public int getResult() {  
    return result;  
}  
  
public void setResult(int result) {  
    this.result = result;  
}  
  
public int getSecondNumber() {  
    return secondNumber;  
}  
  
public void setSecondNumber(int secondNumber) {  
    this.secondNumber = secondNumber;  
}  
}
```

Пример JSF-приложения (продолжение)

Конфигурация faces-config.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config 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-facesconfig_1_2.xsd
    "
        version="1.2">
    <managed-bean>
        <managed-bean-name>calculator</managed-bean-name>
        <managed-bean-class>
            org.itmo.sample.Calculator
        </managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
    </managed-bean>
</faces-config>
```

Пример JSF-приложения (продолжение)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

<head>
    <title>Calculator Application</title>
</head>

<body>
<f:view>
    <h:form id="calcForm">
        <h4>Calculator</h4>
        <table>
            <tr>
                <td><h:outputLabel value="First Number" for="firstNumber" /></td>
                <td><h:inputText id="firstNumber"
                           value="#{calculator.firstNumber}" required="true" /></td>
                <td><h:message for="firstNumber" /></td>
            </tr>
        </table>
    </h:form>
</f:view>
</body>
```

Пример JSF-приложения (продолжение)

```
<tr>
    <td><h:outputLabel value="Second Number"
                        for="secondNumber" />
    </td>
    <td><h:inputText id="secondNumber"
                        value="#{calculator.secondNumber}"
                        required="true" /></td>
    <td><h:message for="secondNumber" /></td>
</tr>
</table>

<div>
    <h:commandButton action="#{calculator.add}"
                      value="Add" />
    <h:commandButton action="#{calculator.multiply}"
                      value="Multiply" />
    <h:commandButton action="#{calculator.clear}"
                      value="Clear" immediate="true"/>
</div>
</h:form>
```

Пример JSF-приложения (продолжение)

```
<h:panelGroup rendered="#{calculator.result != 0}">
    <h4>Results</h4>
    <table>
        <tr><td>
            First Number ${calculator.firstNumber}
        </td></tr>
        <tr><td>
            Second Number ${calculator.secondNumber}
        </td></tr>
        <tr><td>
            Result ${calculator.result}
        </td></tr>
    </table>
</h:panelGroup>
</f:view>
</body>
</html>
```

5. Архитектура корпоративных приложений

Отличия корпоративных приложений от «обычных»

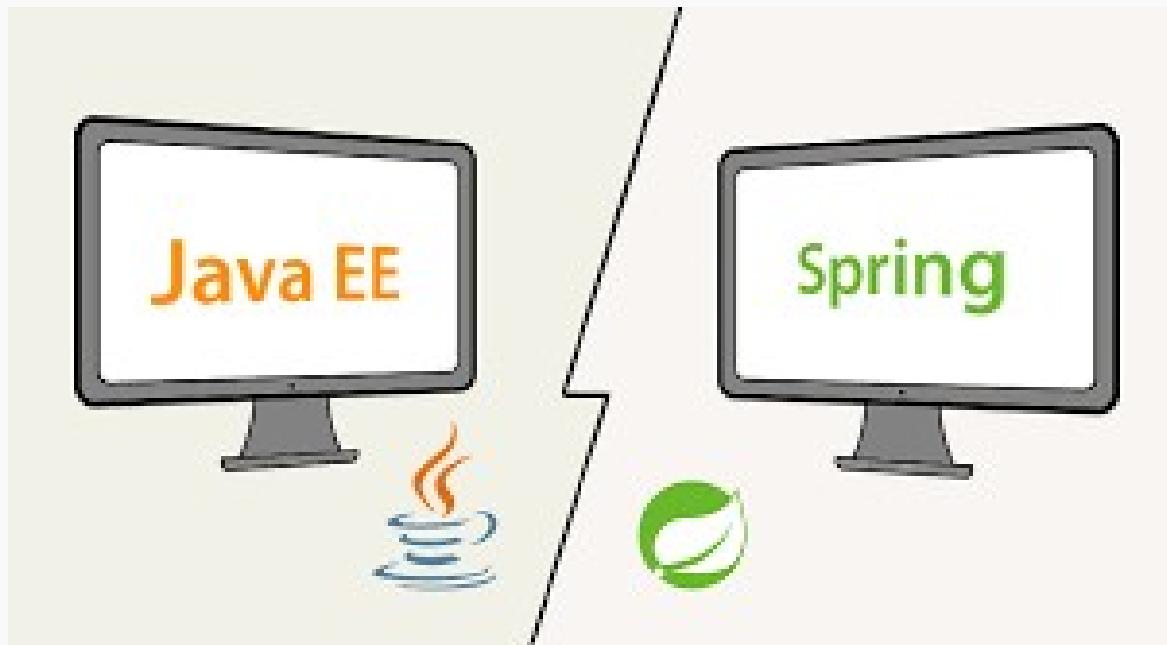
- Зависимость бизнеса от их работоспособности.
- Длительный жизненный цикл.
- Постоянные изменения требований.
- Жёсткие сроки внедрения изменений («эта функция нужна нам вчера!»).
- Сложность и разнородность – обычно состоят из многих подсистем.
- Большие команды разработчиков, часто – узкой специализации.

Требования к архитектуре

- Нужно отделять уровни архитектуры приложения друг от друга.
- Приложение должно строиться из «кубиков» => разные «кубики» смогут разрабатывать разные программисты.
- Нужны высокие масштабируемость и отказоустойчивость => хорошо бы, чтобы «кубики» не взаимодействовали друг с другом напрямую.

Платформы для разработки корпоративных приложений

- Spring: идейно «пляшет» от концепции CDI.
- Java / Jakarta EE: основа – концепции компонентов и контейнеров.



5.1. Java / Jakarta EE Full Profile

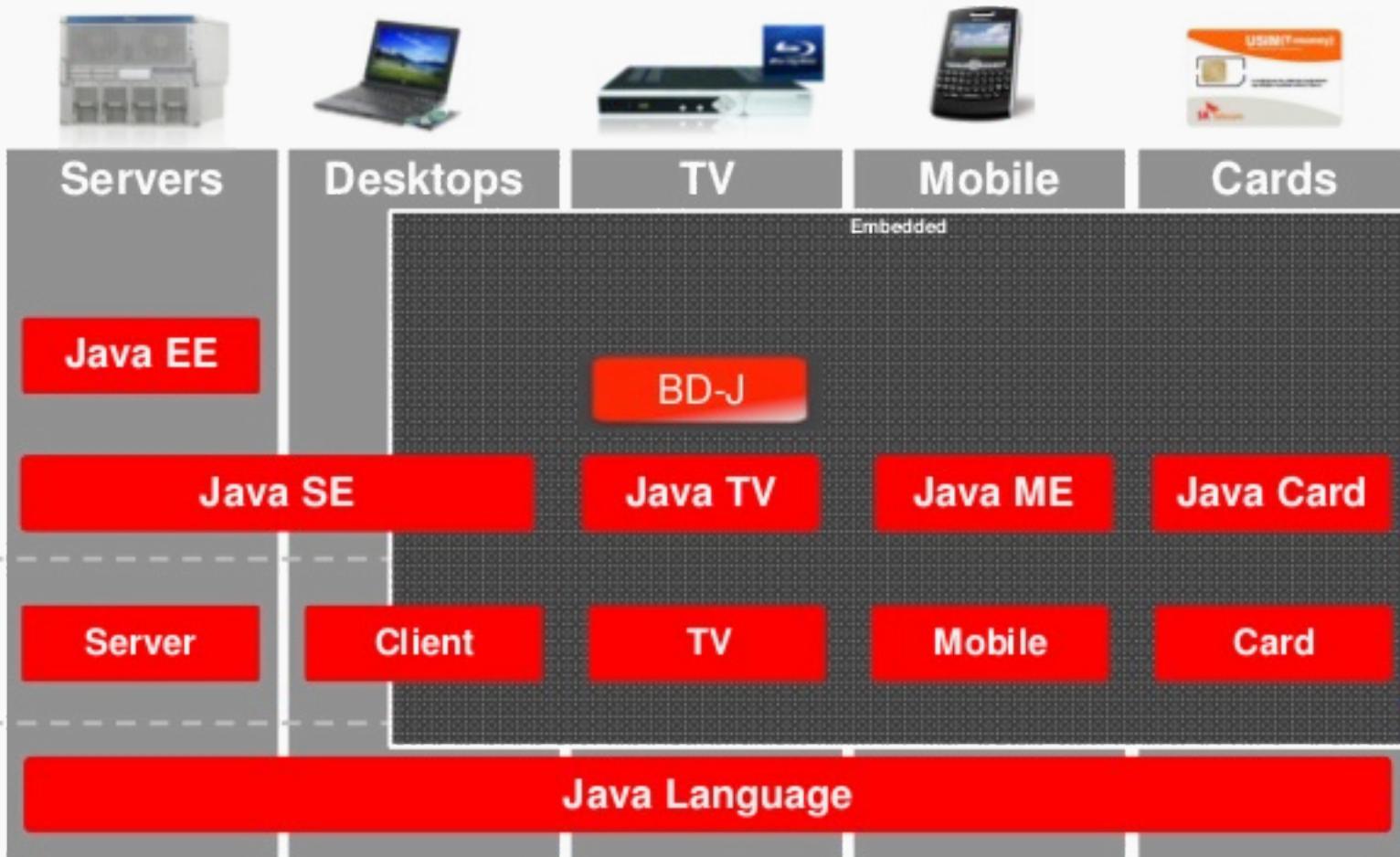
О платформе Jakarta EE

- «Надстройка» над Java SE – те же самые виртуальная машина и язык программирования, но дополнительные API.
- Первая версия вышла в 1999 г., актуальная (9.0) – в 2019 г.
- Другие названия – J2EE (до версии 1.4), Java EE – до версии 8.0.



JAKARTA® EE

Платформы Java



Основные концепции

- Приложения строятся из компонентов, работающих под управлением контейнеров.
- Используются следующие принципы:
 - Inversion of Control (IoC) + Contexts & Dependency Injection (CDI).
 - Location Transparency.

Принципы IoC и CDI

- IoC (применительно к Java EE):
 - Жизненным циклом компонента управляет контейнер (а не программист).
 - За взаимодействие между компонентами отвечает тоже контейнер.
- CDI — позволяет снизить (или совсем убрать) зависимость компонента от контейнера:
 - Не требуется реализации каких-либо интерфейсов.
 - Не нужны прямые вызовы API.
 - Реализуется через аннотации.

Пример CDI (JSF Managed Bean)

```
@ManagedBean(name="message")
@SessionScoped
public class MessageBean implements Serializable {
    //business logic and whatever methods
}
```

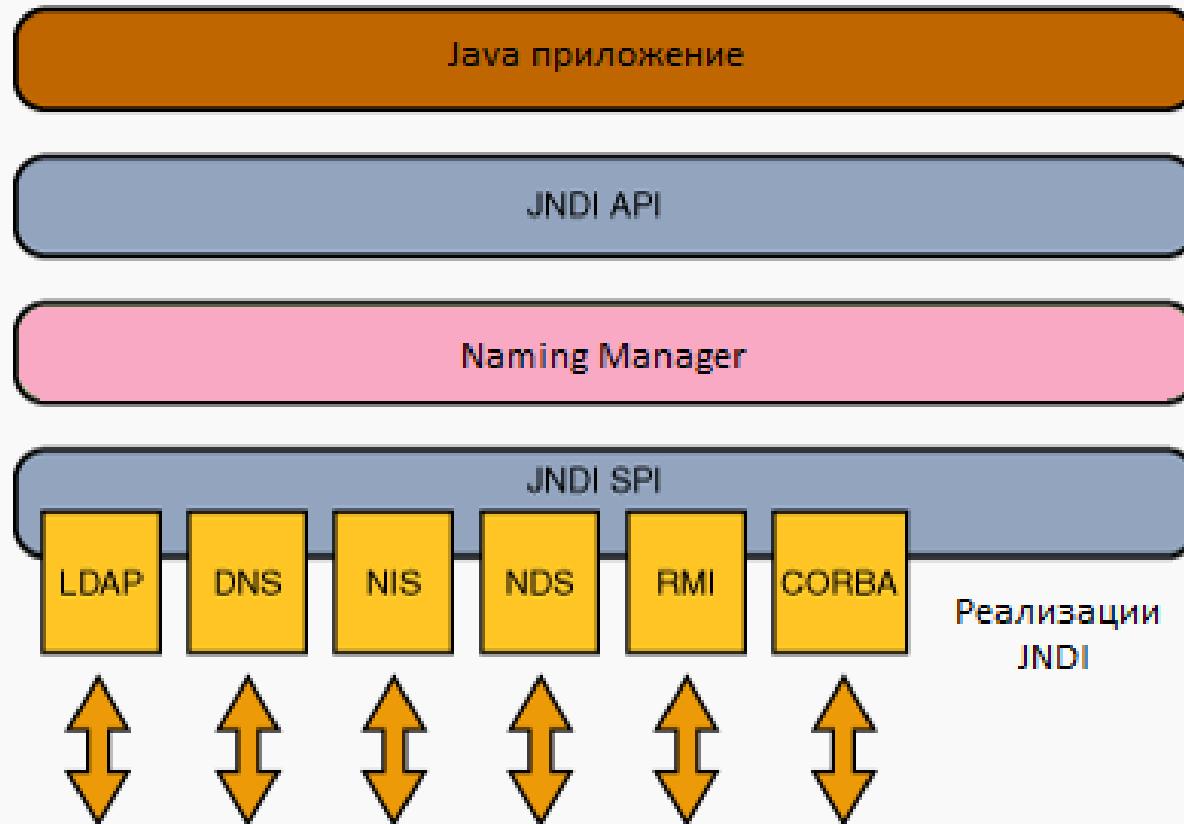
```
@ManagedBean
@SessionScoped
public class HelloBean implements Serializable {

    @ManagedProperty(value="#{message}")
    private MessageBean messageBean;

    //must provide the setter method
    public void setMessageBean(MessageBean messageBean) {
        this.messageBean = messageBean;
    }
}
```

Java Naming & Directory Interface (JNDI)

JNDI — это набор Java API, организованный в виде службы каталогов, который позволяет Java-клиентам открывать и просматривать данные и объекты по их именам (C)
Wikipedia.



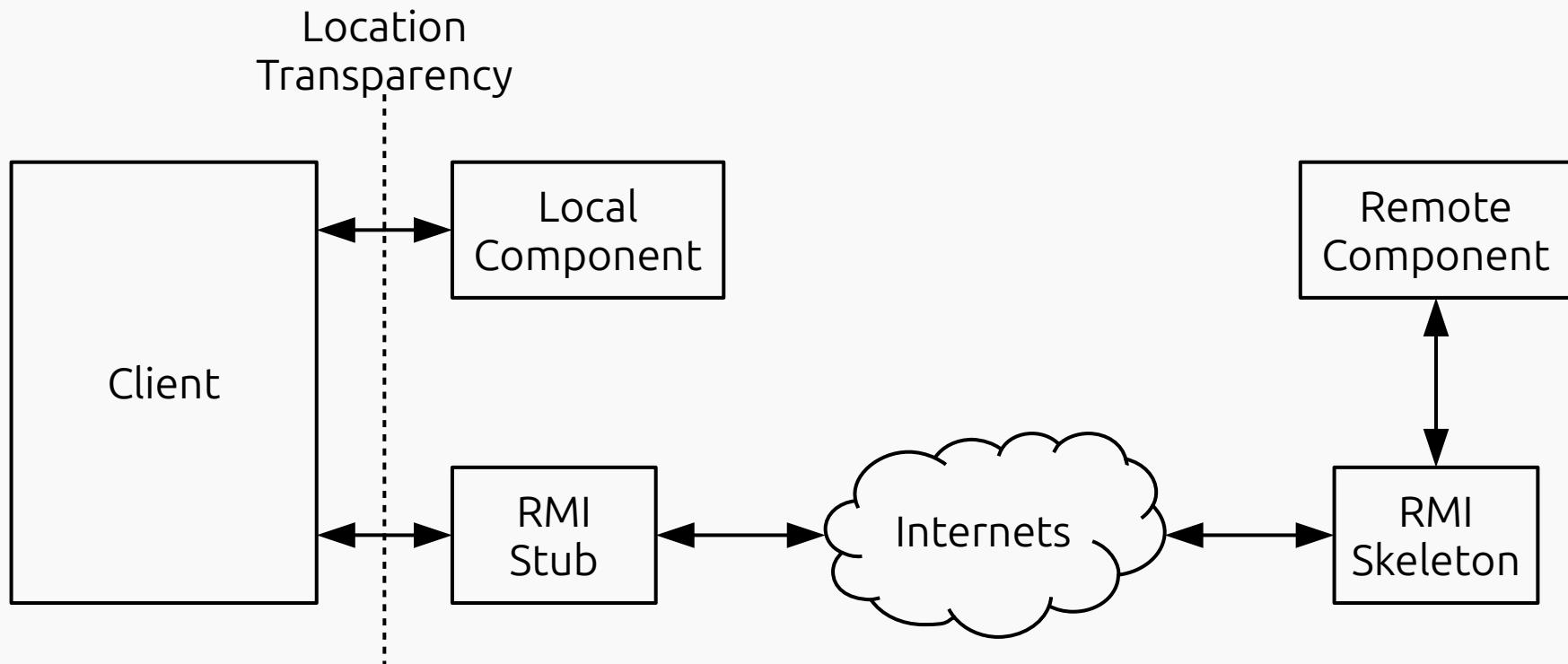
Два варианта использования JNDI:

- CDI (аннотации) — работает только в managed компонентах.
- Прямой вызов API — работает везде.

```
// Пример получения ссылки на JDBC datasource.  
DataSource dataSource = null;  
try {  
    // Инициализируем контекст по умолчанию.  
    Context context = new InitialContext();  
    dataSource =  
        (DataSource) context.lookup("Database");  
} catch (NamingException e) {  
    // Ссылка не найдена  
}
```

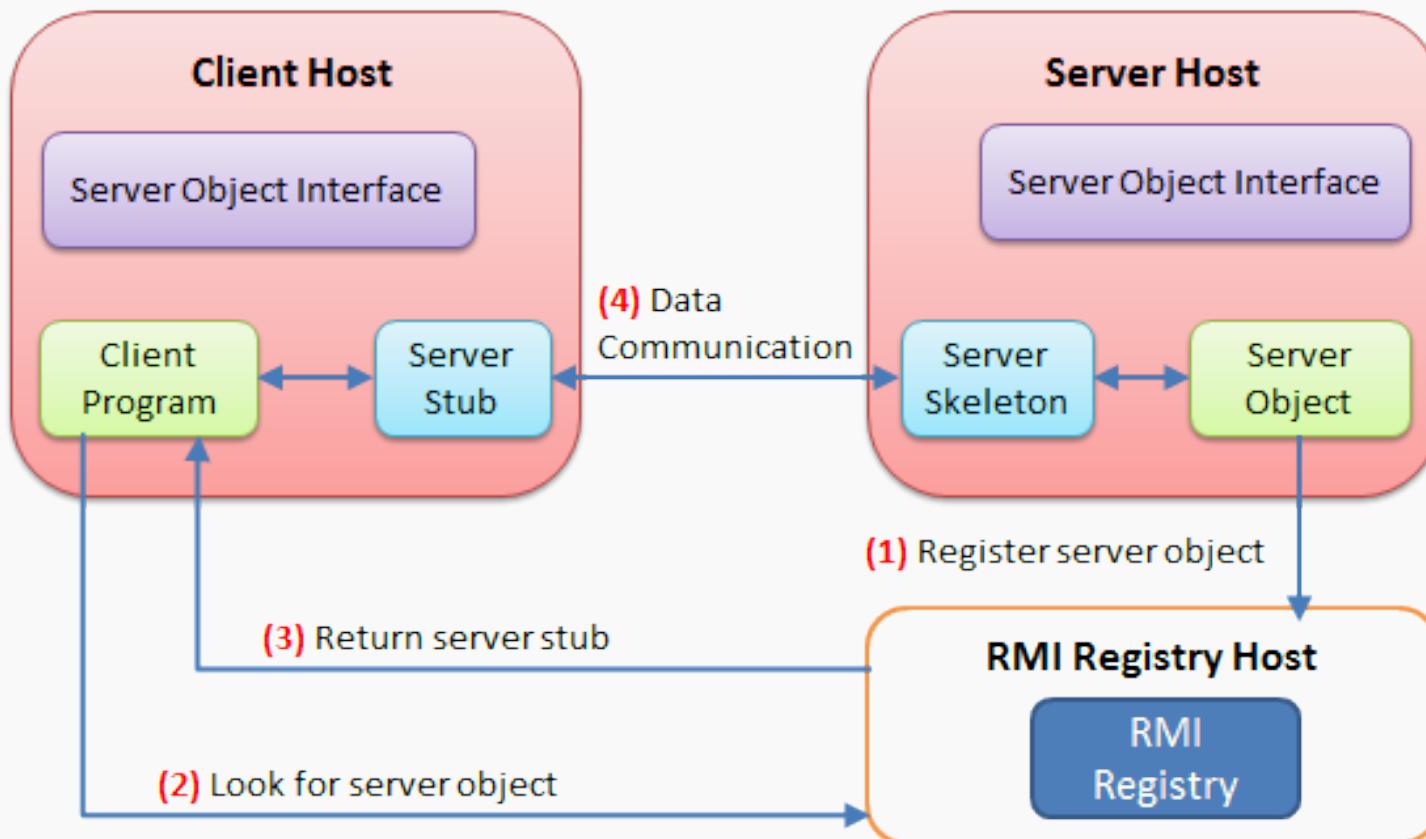
Принцип Location Transparency

Благодаря CDI не важно, где физически расположен вызываемый компонент — за его вызов отвечает контейнер.



Remote Method Invocation

RMI — Java API, позволяющий вызывать методы удалённых объектов.



Особенности RMI

- В общем случае, объекты передаются по значению (копии).
- Передаваемые объекты должны быть Serializable.

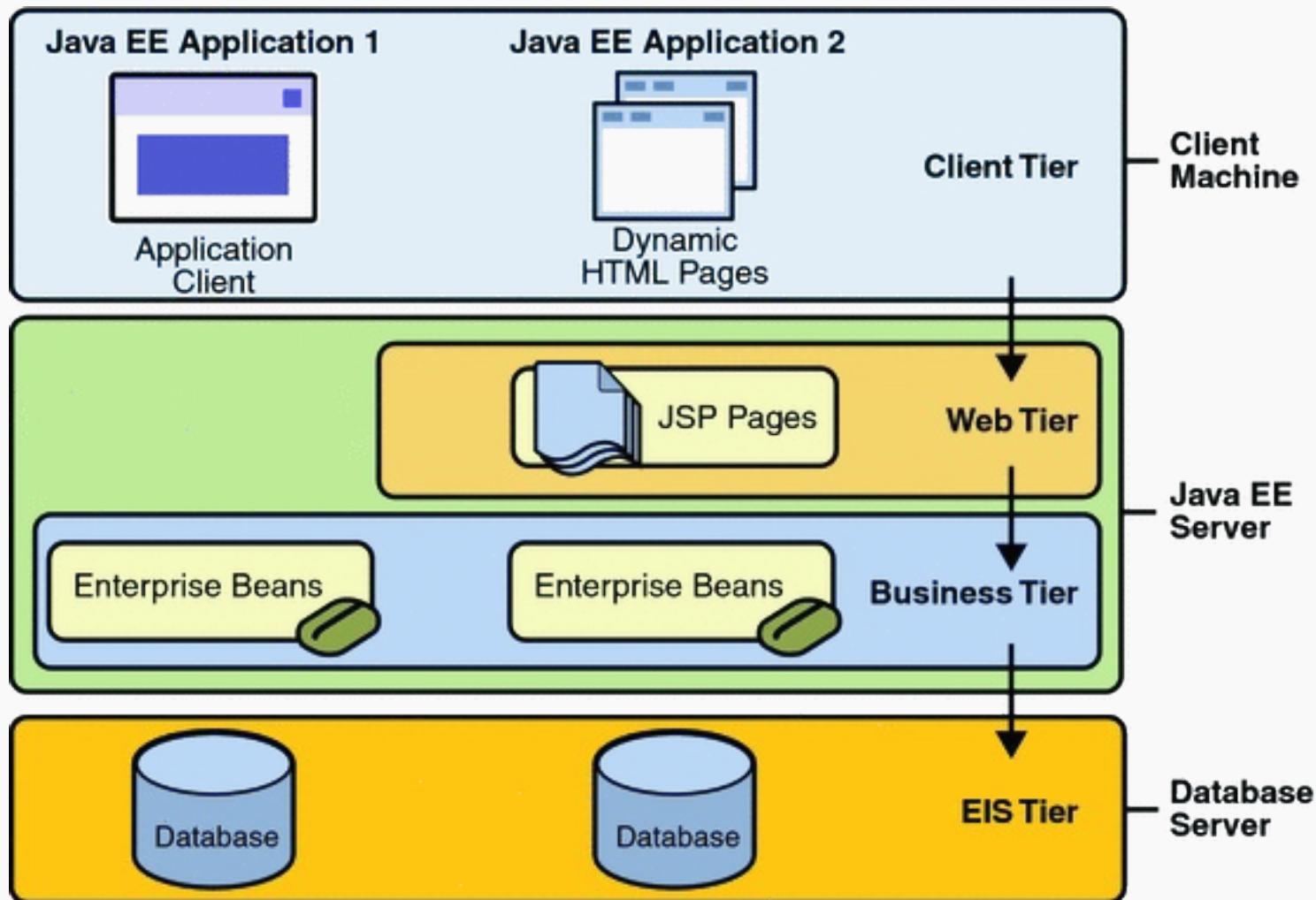
Пример реализации RMI — сервер

```
public class PrimeNumbersSearchServer implements ClientRegister {  
    ...  
  
    public static void main(String[] args) {  
        PrimeNumbersSearchServer server =  
            new PrimeNumbersSearchServer();  
        try {  
            ClientRegister stub = (ClientRegister)  
                UnicastRemoteObject.exportObject(server, 0);  
            Registry registry = LocateRegistry.createRegistry(12345);  
            registry.bind("ClientRegister", stub);  
            server.startSearch();  
        } catch (Exception e) {  
            System.out.println ("Error occurred: " + e.getMessage());  
            System.exit (1);  
        }  
    }  
}
```

Пример реализации RMI — Клиент

```
public class PrimeNumbersSearchClient implements PrimeChecker {  
    ...  
  
    public static void main(String[] args) {  
        PrimeNumbersSearchClient client =  
            new PrimeNumbersSearchClient();  
  
        try {  
            Registry registry = LocateRegistry.getRegistry(null, 12345);  
            ClientRegister server =  
                (ClientRegister) registry.lookup("ClientRegister");  
            PrimeChecker stub = (PrimeChecker)  
                UnicastRemoteObject.exportObject(client, 0);  
            server.register(stub);  
        } catch (Exception e) {  
            System.out.println ("Error occurred: " + e.getMessage());  
            System.exit (1);  
        }  
    }  
}
```

Java EE Application Tiers



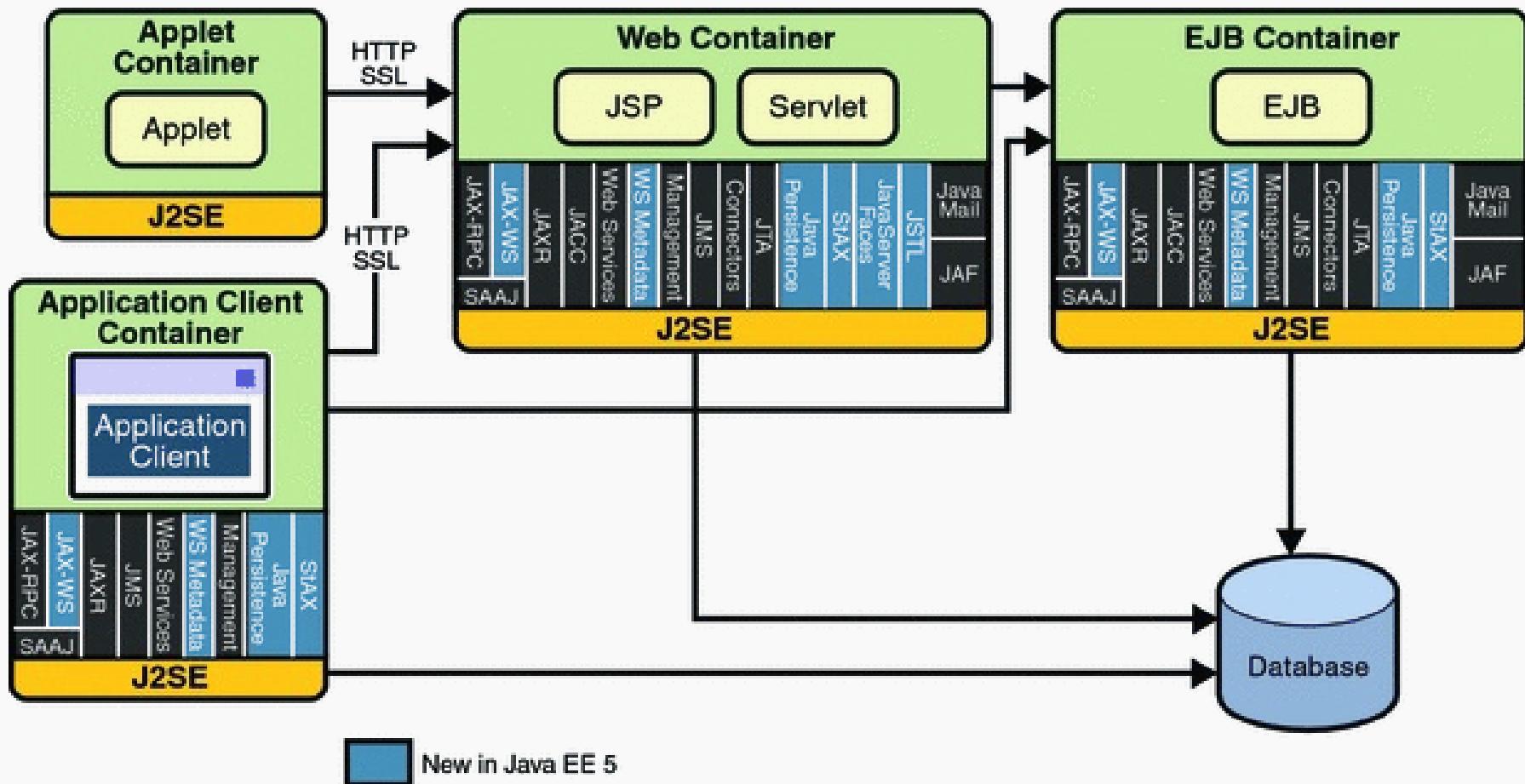
Профили платформы Java EE

- Появились в Java EE 6.
- Позволяют сделать более «лёгкими» приложения, которым не нужен полный стек технологий Java EE.
- Существует только 2 профиля — Full и Web.
- Сервер приложений может реализовывать спецификации не всей платформы, а конкретного профиля.

Java EE Full & Web Profiles

API	Web Profile	Full Profile		API	Web Profile	Full Profile
Servlet	+	+		JTA	+	+
JSP	+	+		JMS		+
JSTL	+	+		JavaMail		+
EL	+	+		JAX-WS		+
JSF	+	+		JAX-RS		+
CDI	+	+		JAXB		+
EJB Lite	+	+		JACC		+
EJB Full		+		JCA		+
JPA	+	+				

Архитектура приложения Java EE Full Profile



5.2. CDI Beans «по-серъёзному»

CDI Beans

- Максимально абстрактная реализация паттерна CDI в Java / Jakarta EE.
- Появились в Java EE 7, «клонируют» бины из Spring.
- Могут использоваться со всеми фреймворками Java / Jakarta EE.
- Конфигурируются аннотациями, основной пакет – javax.enterprise.context.
- Для CDI используется универсальная аннотация @Inject.
- В отличие от EJB, не обеспечивают горизонтальную масштабируемость «сами по себе» (но это часто и не надо!).

Пример инъекции CDI-бина.

@ApplicationScoped

```
public class OrderService { ... }
```

@WebServlet

```
public class OrderServlet extends HttpServlet {
```

@Inject

```
private OrderService service;
```

```
// ...
```

```
}
```

1. При помощи **@ApplicationScoped** указали для CDI-бина время жизни на протяжении ЖЦ приложения.

2. При помощи **@Inject** указали, что поле в `service` необходимо внедрить бин из веб-контейнера.

3. Нужный бин найден по типу — `OrderService`.

CDI Bean Scope

- Аналог контекста (scope) управляемых бинов JSF.
- Определяет жизненный цикл бинов и их видимость друг для друга.
- Задаётся аннотацией.
- В спецификации определены 5 уровней контекста:
 - `@RequestScoped`.
 - `@SessionScoped`.
 - `@ApplicationScoped`.
 - `@ConversationScoped`.
 - `@Dependent`.

Conversation Scope

- Идеально похож на `@CustomScoped` из JSF – жизненным циклом компонента управляет программист.
- Управление осуществляется через инъекцию объекта `javax.enterprise.context.Conversation`.
- Пример использования:

`@ConversationScoped`

```
public class MyBean implements Serializable {
```

`@Inject`

`private Conversation conversation;`

```
public void startConversation() {
```

`conversation.begin();`

`}`

```
public void endConversation() {
```

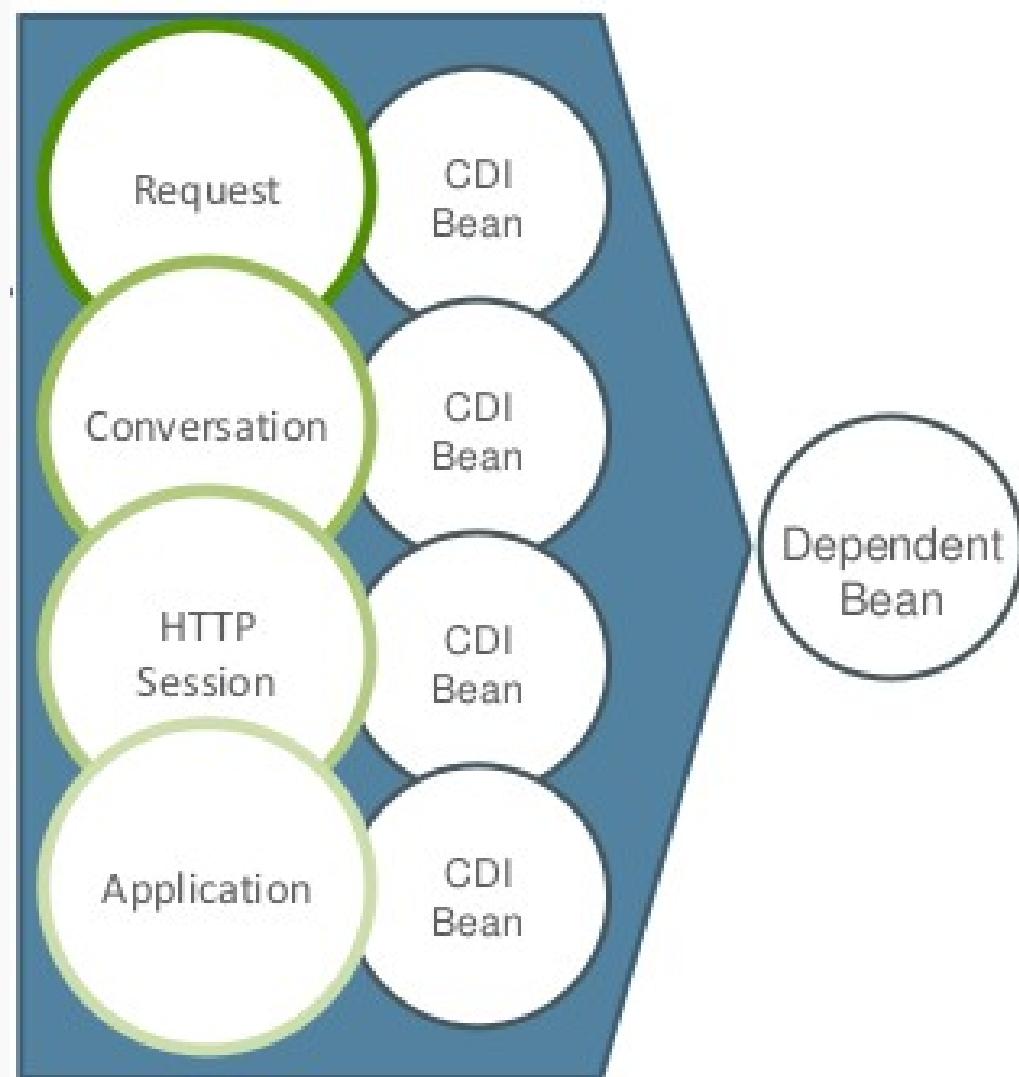
`conversation.end();`

`}`

`}`

Dependent Scope

- Используется по умолчанию, может быть указан «вручную» аннотацией @Dependent.
- Идейно похож на @NoneScoped из JSF – жизненный циклом компонента определяется тем, где он был использован.



Именование бинов

Используется аннотация @Named.

```
@Named("ca")
@RequestScoped
public class CheckingAccount implements Account {...}

@Named("sa")
@RequestScoped
public class SavingsAccount implements Account {...}

@WebServlet
public class AccountServlet extends HttpServlet {
    @Inject
    private @Named("ca") Account account; // Реализацию определяет @Named
}

public class AccountApplication {
    @Inject
    private @Named("sa") Account account; // И здесь тоже
}
```

«Фабрики» (Producer & Disposer Methods)

- Бины, которые управляют созданием новых экземпляров других бинов.
- Используют аннотации `@Produces` и `@Disposes`.
- Позволяют использовать в качестве CDI-бинов классы, не соответствующие паттерну Java Beans (например, имеющие только конструкторы с параметрами).
- `@Produces` определяет метод-фабрику бинов.
- (опционально) `@Disposes` определяет метод,ываемый перед удалением бина, полученного из метода-фабрики.

Пример фабрики

```
@RequestScoped
public class OrderFactory {
    @Produces
    public Order getStoreOrder() {
        return new StoreOrder(...);
    }

    public void disposeOrder(@Disposes Order order) {
        order.close();
    }
}
```

Перехватчики (Interceptors)

- Классы, реагирующие на определённые события жизненного цикла бинов.
- Чем-то похожи на фильтры запросов в сервлетах.
- Конкретный тип события задаётся аннотацией:
 - `@AroundInvoke` (используется чаще всего);
 - `@PostConstruct`;
 - `@PreDestroy`;
 - `@PrePassivate`; // не рассматриваем в курсе
 - `@PostActivate`. // не рассматриваем в курсе



Пример перехватчика 1

```
// Создаём свою аннотацию
@Inherited
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD, TYPE})
public @interface Logging {}

// Реализуем логику перехвата событий заданного типа
@Logging
@Interceptor
@Priority(Interceptor.Priority.APPLICATION)
public class OrderInterceptor {
    @AroundInvoke // Перехватываем вызов метода
    public Object writeLog(InvocationContext ctx) throws Exception {
        logger.log(...);
        Object obj = ctx.proceed(); // Передаём управление в метод
        logger.log(...);
        return obj;
    }
}

// Используем перехватчик для записи в лог
@RequestScoped
public class OnlineOrder{
    @Logging
    public void addProduct(Product p){...}
}
```

Пример перехватчика 2

```
@ApplicationScoped
public class LogService {
    @Inject
    private LogProperties logProperties;

    private PrintWriter logWriter;

    @PostConstruct
    public void init() {
        // бин logProperties уже внедрен
        String filename = logProperties.getFileName();
        logWriter = // инициализировали PrintWriter (открыли лог-файл)
    }

    @PreDestroy
    public void destruct() {
        logWriter.close(); // закрыли лог-файл
    }
}
```

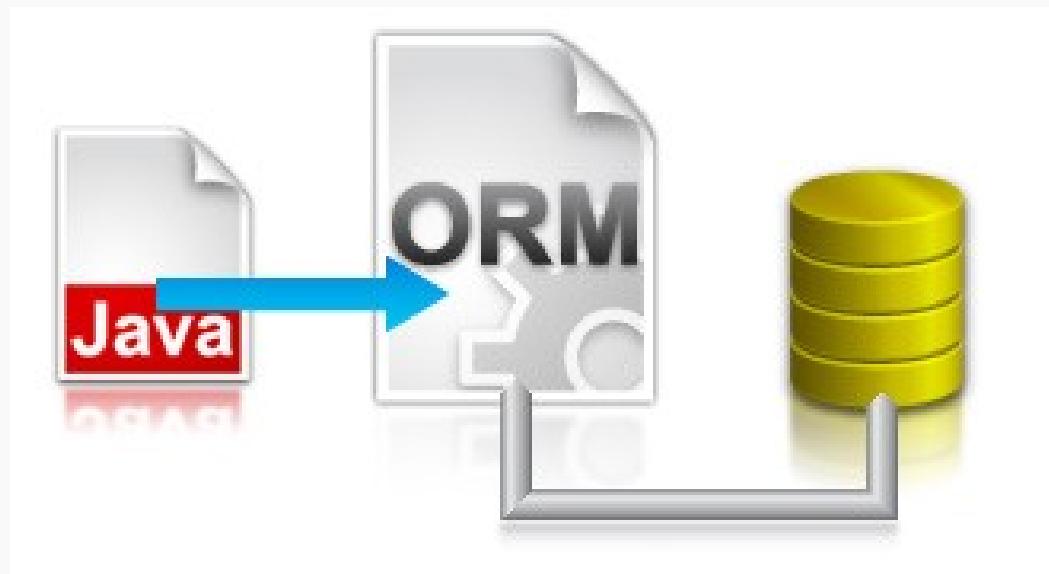
5.3. Введение в ORM

Объектно-реляционное отображение

ORM — Object/Relational Mapping — преобразование данных из объектной формы в реляционную и наоборот.



Как реализовать ORM на Java?



- JDBC;
- ORM-фреймворки (Hibernate, TopLink, ...);
- Java Persistence API (JPA 2.0).

Hibernate ORM

ORM-фреймворк от Red Hat, разрабатывается с 2001 г.

Ключевые особенности:

- Таблицы БД описываются в XML-файле, либо с помощью аннотаций.
- 2 способа написания запросов — HQL и Criteria API.
- Есть возможность написания native SQL запросов.
- Есть возможность интеграции с Apache Lucene для полнотекстового поиска по БД (Hibernate Search).

EclipseLink ORM

ORM-фреймворк от Eclipse Foundation.

Ключевые особенности:

- Основан на кодовой базе Oracle TopLink.
- Является эталонной реализацией (reference implementation) для JPA.

Java Persistence API (JPA)

Java-стандарт (JSR 220, JSR 317), который определяет:

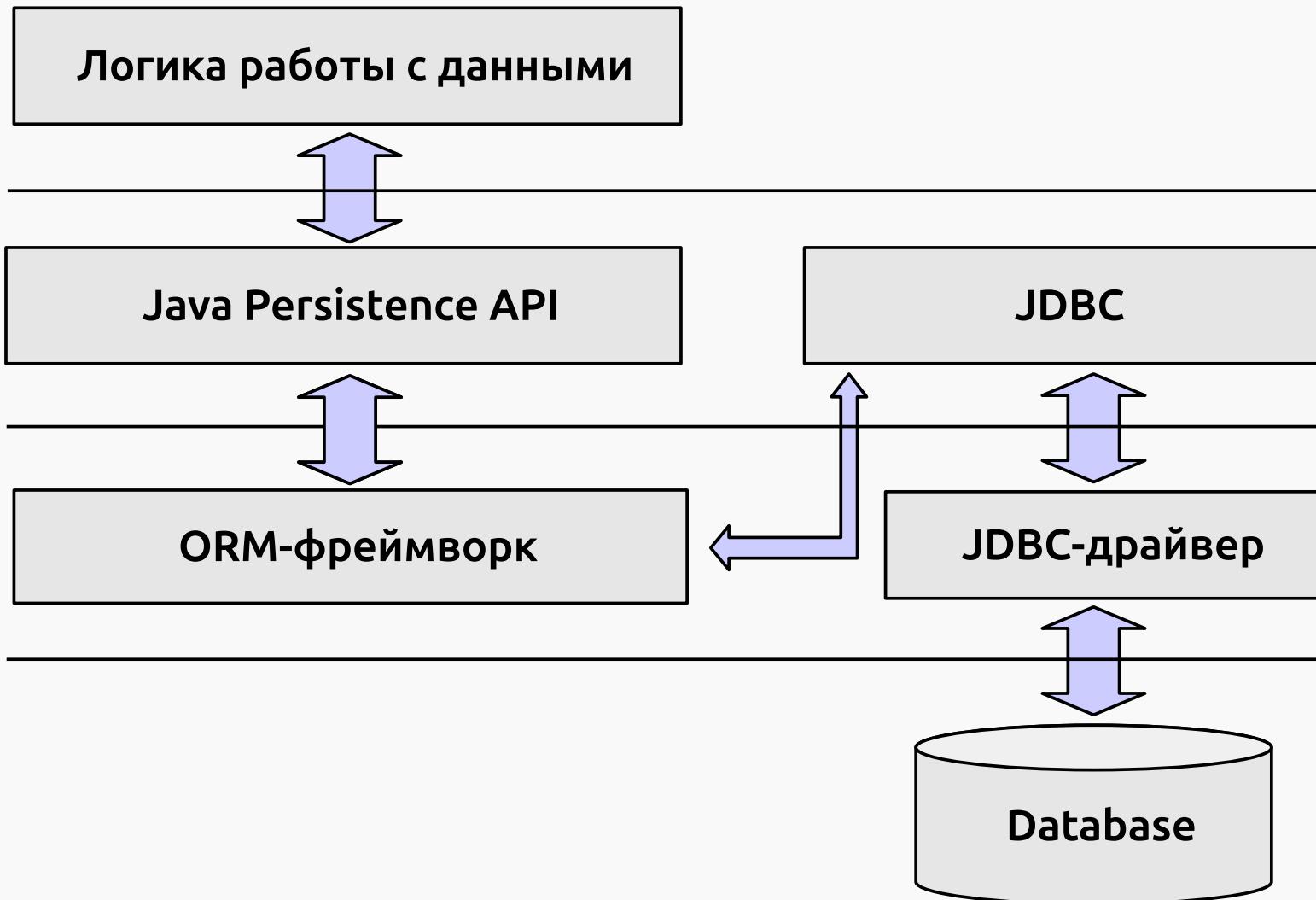
- как Java-объекты хранятся в базе;
- API для работы с хранимыми Java-объектами;
- язык запросов (JPQL);
- возможности использования в различных окружениях.

Что даёт использование JPA?



- Достижение лучшей переносимости.
- Упрощение кода.
- Сокращение времени разработки.
- Независимость от ORM-фреймворков.

Взаимодействие приложения с БД через JPA



Кратко про аннотации JPA

- `@Entity` — помечает Java Bean как JPA Entity;
- `@Table` — задает название таблицы для хранения Entity;
- `@Id` — для каждой Entity обязательно должно быть поле, являющееся идентификатором;
- `@Column` — помечает поле Entity как отображаемое на столбец таблицы, можно задавать атрибуты;
- `@GeneratedValue` — на случай, если идентификатор нужно генерировать автоматически.

Отношения в JPA

Описывают связи между различными Entity. Бывают 4-ех видов: one-to-one, many-to-one, one-to-many, many-to-many.

Вместо отношений можно использовать «простое» поле с идентификатором (отображаемое на внешний ключ).

Подробнее будут рассказывать на 3 курсе.

JPA EntityManager

Основной интерфейс для взаимодействия клиентского кода с ORM механизмом. Основные методы:

- persist() - «сохранить» Entity в БД;
- merge() - «обновить» Entity в БД;
- remove() - «удалить» Entity из БД;
- createQuery(/* JPQL here */) - создание запроса на получение объектов из БД.

Пример использования JPA + Hibernate (1)

Нужно подключить зависимости:

- JPA API — спецификация;
- Hibernate ORM — провайдер спецификации (application сервер может поставлять «из коробки»);
- JDBC драйвер для нужной базы данных.

Пример использования JPA + Hibernate (2)

```
@Entity
@Table(name = «students»)
public class Student {
    @Id
    // @GeneratedValue – если требуется автоматически генерировать id
    private long id;

    @Column(nullable = false) // указываем, что столбец не null
    private String name;

    @Column(name = «student_age») // можно поменять название столбца
    private int age;

    // чтобы использовать TEXT вместо VARCHAR
    @Column(columnDefinition = «TEXT»)
    private String comment;

    // getters & setters & no args constructor
}
```

Пример использования JPA + Hibernate (3.1)

Получение объекта EntityManager (Application Managed) «вручную»:

```
@Produces  
public EntityManager createEntityManager() {  
    var emf = Persistence.createEntityManagerFactory("default");  
    return emf.createEntityManager();  
}
```

WEB-INF/classes/META-INF/persistence.xml:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">  
    <persistence-unit name="default">  
        <class>org.example.demo.Student</class>  
        <exclude-unlisted-classes>true</exclude-unlisted-classes>  
        <properties>  
            <property name="hibernate.hbm2ddl.auto" value="update"/>  
            <property name="jakarta.persistence.jdbc.driver"  
value="org.postgresql.Driver"/>  
            <property name="jakarta.persistence.jdbc.url"  
value="jdbc:postgresql://localhost:5432/postgres"/>  
            <property name="jakarta.persistence.jdbc.user" value="postgres"/>  
            <property name="jakarta.persistence.jdbc.password" value="postgres"/>  
        </properties>  
    </persistence-unit>  
</persistence>
```

Автоматическое
управление схемой

Пример использования JPA + Hibernate (3.2)

Получение объекта EntityManager (Container Managed) «из коробки»:

```
@ApplicationScoped
public class OrderService {
    @PersistenceContext(name = «default»)
    private EntityManager entityManager;
}
```

WEB-INF/classes/META-INF/persistence.xml:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
    <persistence-unit name="default">
        <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
        <properties>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
        </properties>
    </persistence-unit>
</persistence>
```

JNDI Lookup

На стороне application сервера должен быть настроен DataSource, который может быть найден по указанному JNDI имени. Конкретный способ настройки DataSource различается в зависимости от вендора — см. документацию

Пример использования JPA + Hibernate (4.1)

```
@ApplicationScoped
public class StudentService {
    @PersistenceContext(name = «default»)
    private EntityManager entityManager;

    public long createStudent(Student student) {
        entityManager.persist(student);
        return student.getId(); // id был сгенерирован и записан в Entity
    }

    public void updateStudent(long id, String name) {
        entityManager.getTransaction().begin();
        var student = getById(id);
        if (student == null) throw new NoSuchElementException();
        student.setName(name);

        // измененные Entity, находящиеся под управление EM будут
        // автоматически сохранены в бд при завершении транзакции
        entityManager.getTransaction().commit();
    }
}
```

Пример использования JPA + Hibernate (4.2)

```
@ApplicationScoped
public class StudentService {
    @PersistenceContext(name = «default»)
    private EntityManager entityManager;

    public Student getById(long id) {
        return entityManager.createQuery(«SELECT s FROM Student s WHERE
s.id = » + id, Student.class).getSingleResultOrNull();
    }
}
```

JPQL

5.4. REST back-end на JAX-RS

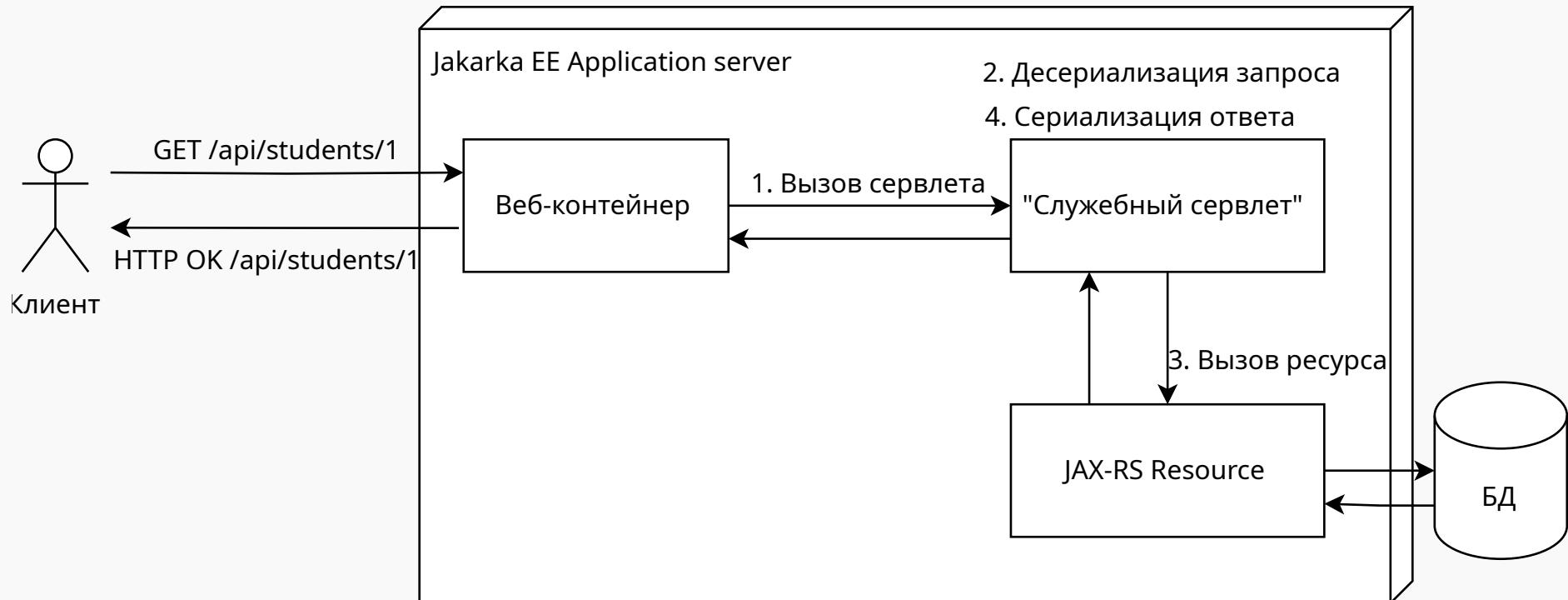
JAX-RS (JSR 370)

Фреймворк для построения RESTful веб-сервисов на базе Jakarta EE.

Ключевые особенности:

- Основан на аннотациях;
- Автоматическая сериализация и десериализация данных;
- Совместим с CDI beans;
- Забирает на себя «рутину» по работе с HTTP;
- «Под капотом» реализуется все еще на сервлетах.

Общий принцип



Основные аннотации

- `@Path` — привязывает класс REST ресурса к URL, а также методы класса;
- `@GET`, `@POST`, `@PUT`, `@DELETE` — определяют методы класса, являющиеся эндпоинтами + определяют HTTP метод доступа к ним;
- `@Produces` и `@Consumes` — определяют типы тела запроса и тела ответа.

Пример

```
@ApplicationScoped
@Path(«/students»)
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class StudentResource {
    @Inject
    private StudentService service;

    @GET
    @Path(«/{id}»)
    public Student getById(@PathParam(«id») long id) {
        return service.getById(id);
    }

    @POST
    public Response create(Student student) {
        return Response
            .status(201)
            .entity(service.create(student))
            .build();
    }
}
```

Десериализация

- `@PathParam` — достаем параметр из пути;
- `@QueryParam` — достаем URL-параметр;
- `@HeaderParam` — достаем значение заголовка;
- `@CookieParam` — достаем значение куки;
- `@FormParam` — достаем параметр из тела запроса при использовании `application/x-www-form-urlencoded` запроса;
- `@Context` — достаем объекты `HttpServletRequest` и `HttpServletResponse`

Настройка API path

Для корректной работы приложения необходимо настроить «entry point» и указать глобальный путь, к которому привязываются все ресурсы.

```
@ApplicationPath(«/api»)  
public class MyApplication extends Application {}
```

Итоговые запросы, которые были написаны в примерах:

GET /api/students/1

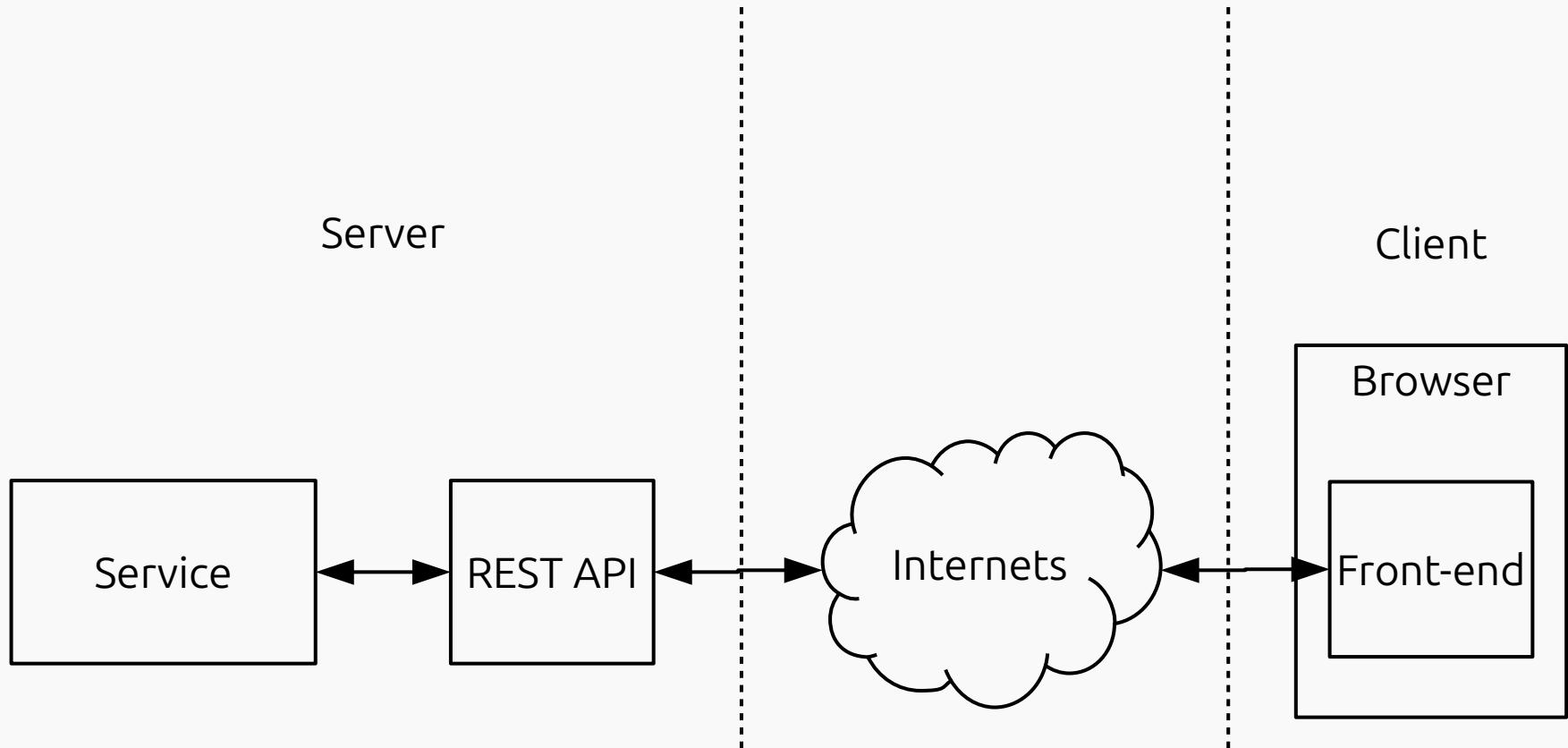
POST /api/students

6. «Современный» фронтиенд

Общие моменты (1)

- Клиент и сервер реализуются независимо, управляют своим состоянием независимо и взаимодействуют по REST.
- Формат передаваемых данных не специфицирован, но обычно это JSON.
- JS (TS) Библиотека\Фреймворк + CSS (SCSS).
- Часто реализуется как набор AJAX-страниц, однако «современный» фронтенд — полноценное веб-приложение (только в браузере).

Common Practice



Общие моменты (2)

- Фреймворков и библиотек очень много (JS-бинго!), знать все – невозможно.
- Базируются на схожих принципах, уровень абстракции различается.
- Наиболее популярные сейчас (осень 2024 г.) – React («библиотека, а не фреймворк»), Angular, Vue.

6.1. Рендеринг

MPA (Multiple Pages Application)

«Раньше» фронтенд представлял собой набор html страничек. Например, несколько JSP-страниц.

- У каждой страницы свой URL => «Честный» роутинг;
- Для каждой страницы — общие *.css, *.js + специфичные для страницы;
- Сложности реализации из-за тесной связи фронтенда и бэкенда.

SPA (Single Page Application)

«Теперь» фронтенд представляет собой одну html страницу, на которой динамически при помощи JS рендерится контент.

- У каждой страницы все еще свой URL, но «виртуальный» => «Виртуальный» роутинг через BOM;
- Для всех страниц — общие *.css, *.js;
- Уменьшение количества трафика между клиентом и сервером;
- Удобство разработки фронтенда;
- Лучше интерактивность.

PWA (Progressive Web Application)

«Модный» принцип, согласно которому веб-приложение реализуется «как будто бы» нативное для целевой платформы.

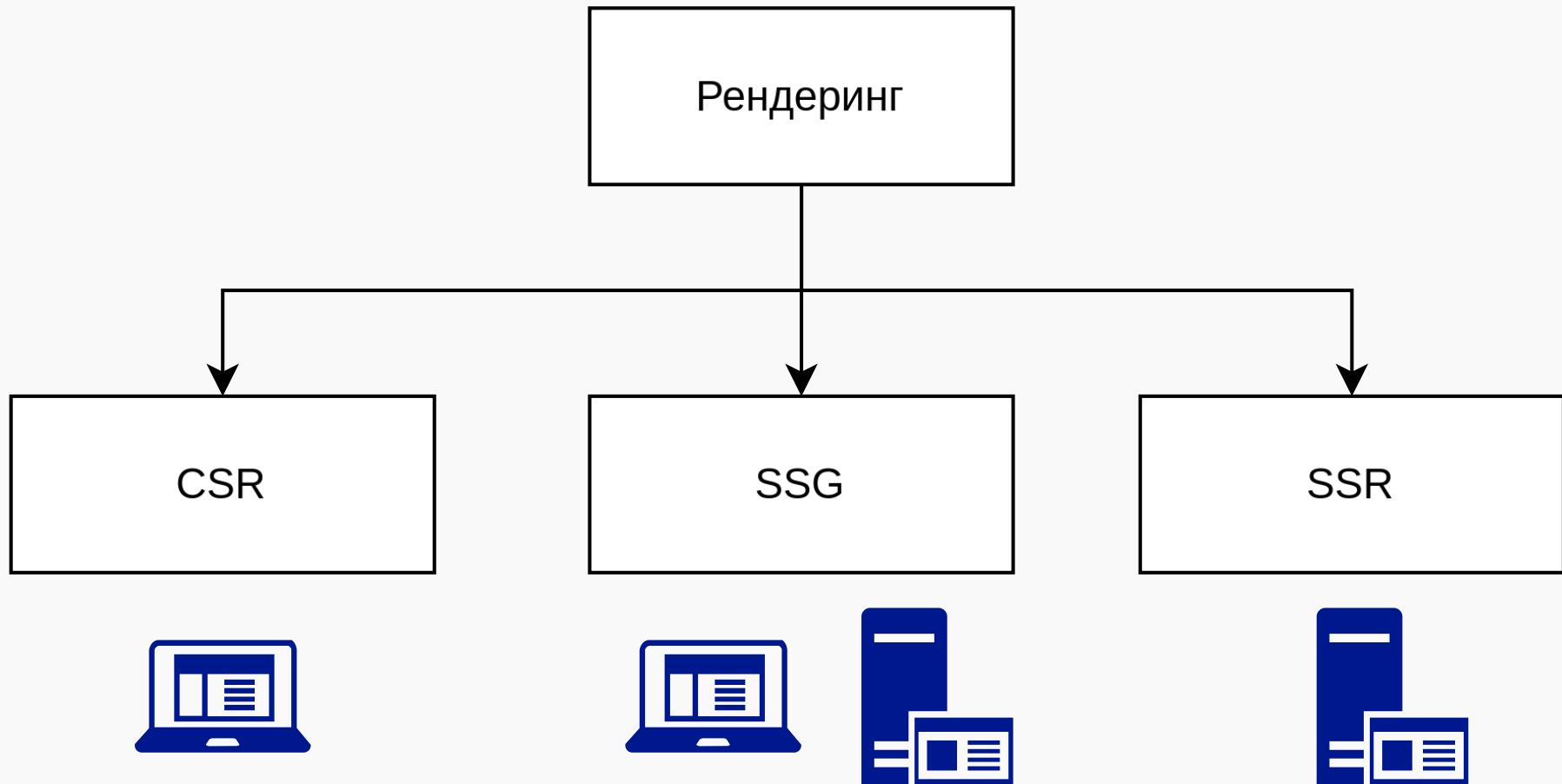
- Актуально для мобильных платформ;
- Возможен offline-режим работы;
- Производительность;
- Сложности реализации вызванные дополнительным технологиями.

Проблема рендеринга

С развитием JavaScript появилась возможность динамически отображать (рендерить) содержимое страниц прямо в браузере.

- Компьютеры пользователей «мощные» => можно переложить ответственность за рендеринг на них;
- Как следствие уменьшаем объем трафика (важно для мобильных устройств!);
- Однако возникает задержка «первого рендеринга».

Паттерны рендеринга



CSR (Client Side Rendering)

Подход, при котором за весь рендеринг отвечает Браузер клиента (используются возможности JavaScript).

Преимущества:

- Снижение объема трафика;
- «Удобство» разработки;
- «Простота» реализации;

Недостатки:

- Поисковая индексация (SEO);
- Высокое время до первой отрисовки;
- Высокое время до первого взаимодействия со страницей;
- Обязателен JS, чтобы увидеть хоть что-то.

SSR (Server Side Rendering)

Подход, при котором за весь рендеринг отвечает сервер (JSP, JSF, Django, Spring + Thymeleaf).

Преимущества:

- У браузера минимум работы => низкие задержки рендеринга;
- Можно без JS;
- Все прекрасно с поисковой индексацией (SEO);

Недостатки:

- Практически всегда больше трафик;
- Требуется больше серверных ресурсов;
- «Технологически сложнее».

SSG (Server Side Generation)

Подход, при котором Браузер все еще выполняет рендеринг на основе динамических данных, однако статические данные рендерятся на сервере.

Преимущества:

- Ниже задержки рендеринга на клиенте, чем при CSR;
- «Распределение» потребностей в ресурсах между клиентами и сервером;
- «Плюс-минус» хорошо с поисковой индексацией (SEO).

Недостатки:

- Все еще больше трафика, чем при CSR;
- Все еще нужны серверные ресурсы для рендеринга;
- «Технологически сложнее».

Вывод по рендерингу

Для каждой задачи — свой инструмент.

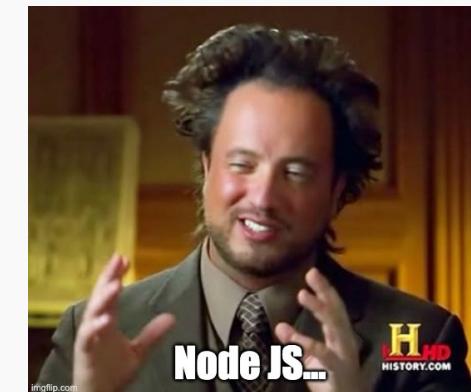
- SSR идеально подходит в случае, если динамически данных нет или мало;
- SSR можно применить для повышения отзывчивости фронта ценой высоких расходов на серверные мощности;
- SSG стоит применять, когда «динамичность» данных невысока (статичность данных между некоторыми пользователями);
- CSR применяется для наибольшей интерактивности ценой первого рендеринга.

6.2. «Вспомогательные» инструменты

JavaScript Runtime

Для исполнения JS нужна виртуальная машина.

- Внутри браузеров — свой JS runtime;
- На серверах - есть Node.js;
- На Node.js (следовательно на JS) можно писать backend;
- Node.js также используется для вспомогательных задач



Node.js

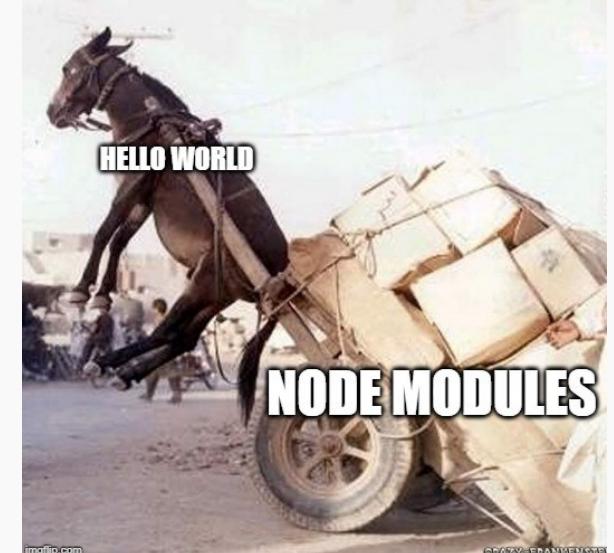
Кросс-платформенная программная среда,
основанная на движке V8. © Википедия

- Поддержка IO;
- Возможность вызова библиотек, написанных на других языках;
- Асинхронная среда;
- Событийно-ориентированная среда;
- Можно писать «простенький» backend;
- Не подходит для «кровавого enterprise».

Пакетные менеджеры

Нужен инструмент управления зависимостями, так как библиотеки пишутся «на каждый чих».

- Runtime — Node.js;
- Наиболее популярные — npm, yarn;
- Структура дескриптора пакета — package.json;
- Реестры пакетов (npm registry) похожи на Maven registry.
- Установленные зависимости — node_modules.



Пример package.json

```
{  
  "name": "test-project",  
  "version": "1.0.0",  
  "main": "src/main.js",  
  "scripts": {  
    "dev": "webpack-dev-server --inline --progress --config  
build/webpack.dev.conf.js",  
    "build": "node build/build.js"  
  },  
  "dependencies": {  
    "vue": "^2.5.2"  
  },  
  "devDependencies": {  
    "babel-core": "^6.22.1",  
    "babel-eslint": "^8.2.1",  
    // и многие другие  
  },  
  "engines": {  
    "node": ">= 6.0.0",  
    "npm": ">= 3.0.0"  
  },  
  "browserslist": [  
    "> 1%",  
    "last 2 versions",  
    "not ie <= 8"  
  ]  
}
```

Примеры

npm install // уагн — установка зависимостей, описанных в package.json

npm install lodash // уагн add lodash — добавить зависимость в package.json и сразу установить

npm uninstall lodash // уагн remove lodash — удалить зависимость из node_modules и package.json

npm run build // уагн build — выполнение скрипта из package.json

TypeScript

Оказалось, что статическая типизация удобна и во фронтенде. В 2012 Microsoft представила TypeScript, как расширение JavaScript.

- Статическая типизация;
- Классы\типы\интерфейсы;
- Generics;
- Компилируется в JavaScript при помощи отдельного компилятора.

Транспиляция и Babel

ECMAScript развивается быстрее, чем браузеры успевают обновляться => нужно обеспечивать поддержку совместимости. При этом хочется писать на свежей версии JS.

Babel — специализированный компилятор, занимающийся транспилированием «нового» JS в «старый» JS.

- Переписывает новые синтаксические конструкции в старые;
- Добавляет вставки кода за место неподдерживаемого API;
- Вообще говоря, можно обойтись и без него.

Системы сборки (бандлеры)

Как и в Java нужные системы сборки, которые позволяют решать задачи:

- Компиляция JSX шаблонов (о них позднее);
- Препроцессинг CSS (SCSS\SASS\LESS);
- Сборка статики (изображения, логотипы, иконки);
- Минификация (сжатие) множества JS (CSS) файлов в один меньшего размера;
- Подстановка env переменных на этапе сборки (в случае CSR нет «честных» env-переменных).
- Сборка зависимостей.

Webpack

Наиболее популярный бандлер в мире JS.

- Имеет широкую систему плагинов.
- Поддерживает интеграцию с Babel.
- Также предоставляет dev-server

```
const path = require('path')

module.exports = {
  entry: './app/index.js',
  module: {
    rules: [
      { test: /\.svg$/, use: 'svg-inline-loader' },
      { test: /\.css$/, use: [ 'style-loader', 'css-loader' ] },
      { test: /\.js$/, use: 'babel-loader' }
    ]
  },
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'index_bundle.js'
  }
}
```

Dev серверы

При разработке хочется иметь возможность изменять файлы проекта и сразу видеть изменения в браузере.

Нужны специализированные инструменты — dev servers.

- Работают на Node.js;
- Осуществляют пересборку проекта «на лету» (hot reload);
- **Небезопасно использовать в production!**

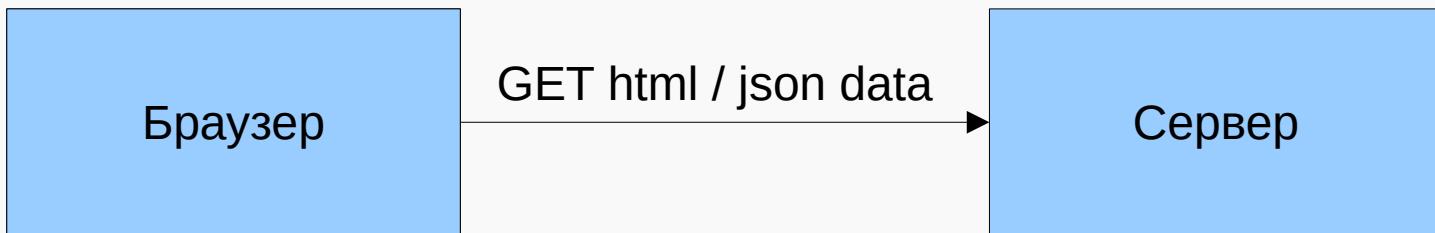
6.3. React

...см. отдельную презентацию.

6.4. Архитектура Frontend

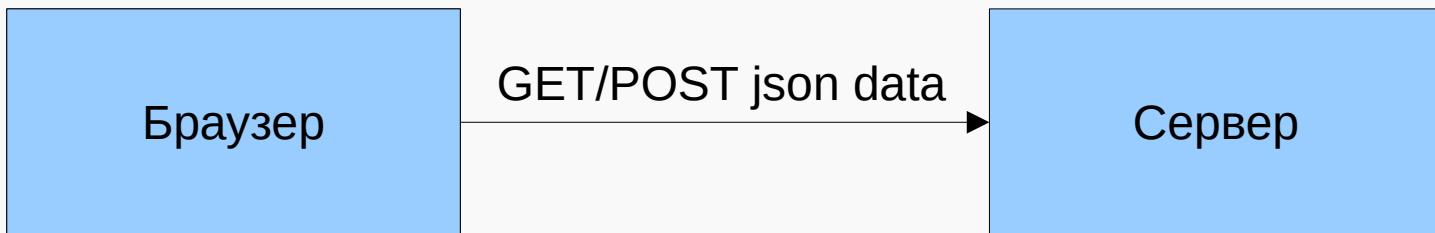
Самостоятельное приложение

- Изначально frontend был нужен для отображения статических данных => «простой HTML»;
- С развитием веба появился JS и AJAX как ответ на потребность в большей отзывчивости сайтов, однако концепт не поменялся.



Самостоятельное приложение

- Сегодня пользователям нужно не только читать, но и обширно манипулировать данными;
- Важен UX => сложные анимации, много вспомогательных UI элементов;
- В идеале хочется взаимодействовать с сайтом так, будто бы сервера не существует.



Хотя принцип общения с сервером не поменялся!

Типовые задачи frontend

«Сложные» формы:

- Валидация формы до отправки на сервер;
- Формы могут видоизменяться по мере заполнения;
- Желательно еще и сохранять состояние формы на случай, если пользователь захочет заполнить позднее.

Логика для
интерактивного UI

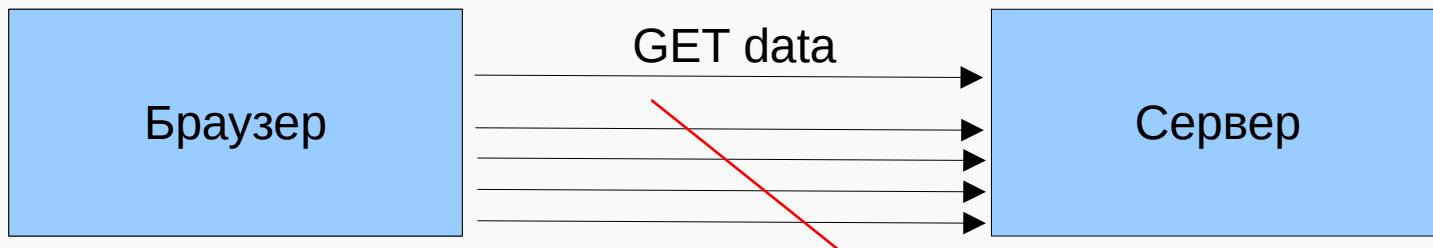
Бизнес-логика

Нужно разделять разную логику

Типовые задачи frontend

«Шаринг» порции данных между разными страницами:

- В качестве примера — хедер в ИСУ (информация о текущем пользователе сразу на всех страницах)
- Хочется логику разных страниц изолировать друг от друга
- Хочется не делать лишних запросов к серверу



Типовые задачи frontend

Знания о ролевой модели:

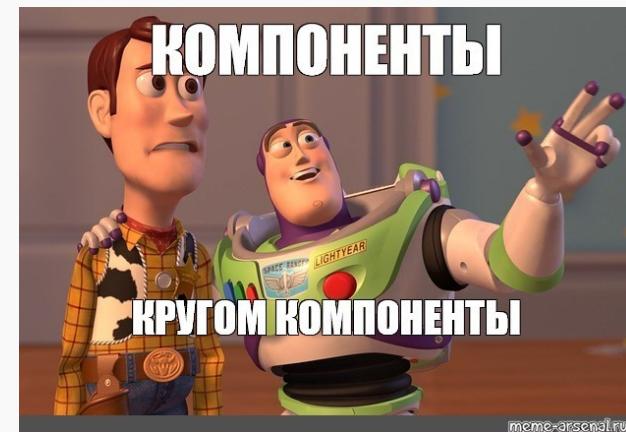
- Раньше о ролевой модели мог знать только сервер (отдавать 403 при запросах «запрещёнки»)
- Теперь нужно frontend приложению скрывать/дизайблить элементы интерфейса в зависимости от роли пользователя
- Также часто требуется более сложная навигация, зависящая о роли пользователя.

И много других типовых задач, на этом список не ограничивается...

Компонентный подход

Появился в мире десктопных приложений «во времена динозавров». Примерно с появлением React стал основополагающим в мире frontend.

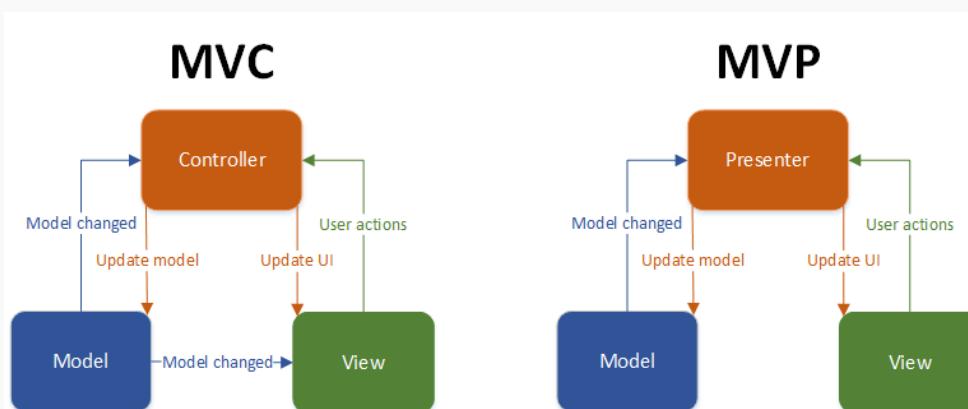
- У компоненты есть HTML представления;
- У компонента есть вспомогательная логика, отвечающая за:
 - анимацию
 - интерактив
 - обработку событий DOM
- Основная идея - **переиспользование**



MVC & MVP & MVVM

Практика показала, что «каноничный» MVC не подходит для frontend. Появились производные шаблоны в виде MVP и MVVM.

ИМХО: MVP и MVVM одно и то же



Ключевое отличие в том, что в MVC View содержит в себе UI-related логику.

В MVP UI-related логика изолирована в Presenter. View отвечает только за рендеринг.



MVP на примере React

```
function ExampleComponent(props) {  
    // RTK для обращения к модели (модель получаем с сервера и  
    // храним в глобальном состоянии)  
    const {data, isLoading} = service.useGetDataQuery();  
  
    // UI-related логика – компонент сам по себе Presenter  
    const [counter, setCounter] = useState(0);  
    const handleUserClick = () => setCounter(counter + 1);  
  
    return (  
        <div>  
            {/* тут какая-то верстка – View */}  
        </div>  
    );  
}
```

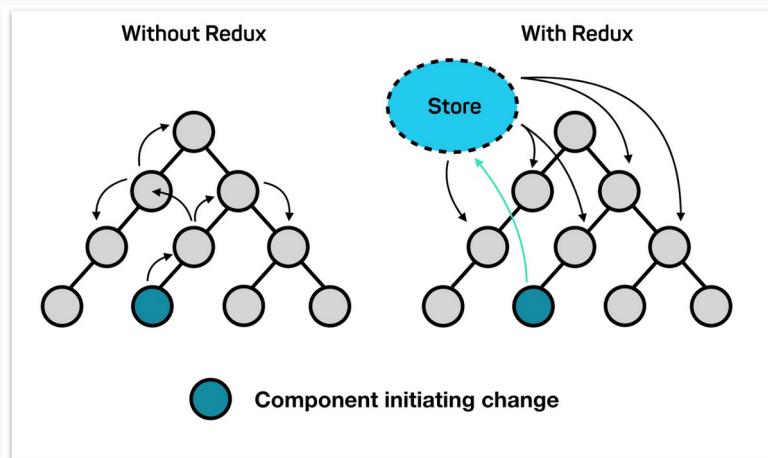


Управление состоянием

- У компонентов есть локальные состояния => видны только компонентам
- Дочернему компоненту можно передать read-only состояние через механизм, аналогичный `props` из React
- Но что если нужно «отдаленным компонентам» использовать одни и те же данные с сервера (запрашивать их дважды плохо)? => нужны глобальные состояния

Общий принцип

- В глобальной области видимости JS создается объект(-ы), где храним состояние
- Получаем доступ из любого компонента к глобальным объектам
- Не стоит «прямо все» хранить в глобальном состоянии — **только то, что нужно**



«Безобразие» библиотек

Как и обычно, в мире frontend-а на любую проблему есть 100500 библиотек. Наиболее популярные:

React:

- Redux
- Jotai
- Recoil
- Zustand

Angular => «стандартное решение» NgRx, но можно прикрутить и другое

Jotai

Redux — сложно, много типового кода (reducers, actions, selectors). Даже с использованием Redux Toolkit все еще хочется проще.

Jotai — легковесная (2кб), гибкая библиотека для управления глобальным состоянием. Базируется на концепте «атома» вместо «стора». Первая версия появилась летом 2021.

```
// создаем atom
const countAtom = atom(0);

function Counter() {
  // используем atom в компонентах
  const [count, setCount] = useAtom(countAtom);
}
```



<https://github.com/pmndrs/jotai>

Глобальное состояние «на максималках»

Вспоминаем проблему с «шарингом» одних и тех же данных с сервера между множеством компонентов.

Существуют библиотеки для взаимодействия с REST API сервера, выполняющие кэширование запрошенных данных при помощи глобального состояния (обычно под капотом Redux).

- SWR
- RTK Query
- и другие...

Общий принцип

- При получении данных размещаем их в глобальном сторе
- Связываем с каждой порцией данных специальную метку
- Если данные изменяются по определенной метке => библиотека сама перезапросит данные с сервера
- Вообще говоря, при получении данных сперва проверяем глобальный стор => если данные уже есть, то используем кэшированные
- Также раз в п-ое время библиотека может инвалидировать кэшированные данные => позволяет увидеть обновление, сделанное другим пользователем

Пример на RTK Query (1)

```
// Объект, представляющий собой весь наш API
export const gatewayApi = createApi({
  reducerPath: 'gatewayServiceApi',
  endpoints: () => ({}),
  tagTypes: ['Chapters'],
})

// Подключаем reducer RTK Query к стору
export const store = configureStore({
  reducer: {
    [gatewayApi.reducerPath]: gatewayApi.reducer,
  }
})
```



Пример на RTK Query (2)

```
// Часть нашего API
export const chapterService = gatewayApi.injectEndpoints({
  endpoints: (builder) => ({
    getChapterList: builder.query<Array<Chapter>, GetParams>({
      query: ({parentId,}) => ({
        url: `${BASE_URL}/${parentId}/chapters`
      }),
      providesTags: ['Chapters'],
    }),
    createChapter: builder.mutation<Chapter, CreateParams>({
      query: ({ parentId, ...body }) => ({
        url: `${BASE_URL}/${parentId}/chapters`,
        method: 'POST',
        body,
      }),
      invalidatesTags: ['Chapters'],
    }),
  })
});
```

```
// в компоненте
const { data: chapters, isLoading } =
  chapterService.useGetChapterListQuery({parentId})
```

6.5. Angular

...см. отдельную презентацию.