Java course

Search

Go to ... ▼ Go to ... ▼

С чего начинается Web

Прежде,чем приступить к практическим вопросам web-программирования, мне бы хотелось немного поговорить о причинах возникновения Internet и intranet систем. Потому что на мой взгляд, это крайне важно — понять, какие задачи хотелось решать и какие инструменты возникли для решения этих задач.

В конце 80-х — начале 90-х годов XX-го столетия практически все системы все еще строились по принципу клиент-сервер. На компьютере работника было установлено графическое (или текстовое) приложение, которое обращалось к базе данных с запросами на изменение или получение данных.

Таким образом было две стороны:
Во-первых — клиент (как сейчас принято говорить, “толстый”), который имел какую-то логику и и давал возможность пользователю что-то делать с данными.
Во-вторых — сервер (обычно СУБД типа SQL), на котором выполнялись запросы на получение данных и их изменение. За счет возможности писать сохраненные процедуры (Stored Procedure) на каком-то расширении стандартного SQL (для Oracle — PL/SQL, для Sybase (и Microsoft) — TransactSQL, для DB2 — SQL PL). Такого рода процедуры позволяли писать достаточно сложную логику обработки данных, не перекладывая эту работу на клиента. Да и изменение логики было удобно сделать — в одном месте поменял и готово.
Клиентское приложение конечно же надо было в случае чего обновлять.

До какого-то момента всем казалось, что это работает замечательно.

Но потом появились ПОТРЕБНОСТЬ. Именно так — с большой буквы. ПОТРЕБНОСТЬ.

Системы клиент-сервер требовали сотрудника. Если клиент хочет сделать заказ — он должен связаться с сотрудником, который занесет его заказ в базу данных с помощью своего клиентского приложения.
Достаточно прямолинейное решение — дать клиентам приложение (пусть и ограниченное) было бы слишком опрометчивым. Во-первых — его надо сделать. В-вторых — его надо менять под потребности клиентов и каждый раз обновлять у этих самых клиентов.
В-третьих — система все чаще становилась зависимой не от одной базы данных, а от многих источников информации.
И что самое главное — БЕЗОПАСНОСТЬ. Клиентское приложение ходит в базу данных. Центральное хранилище “открывать” наружу — это уже не безопасность. Это по сути дела “дать всем ключи от квартиры, где деньги лежат”.

Итак, выросли три задачи. Хотя может и больше, но на мой взгляд именно эти моменты стали определяющими:

1. Безопасность
2. Клиентское приложение с возможностью обновления для сотен тысяч пользователей
3. Возможность принимать запросы от универсального клиента и писать логику для распределенных источников информации

Безопасность
Вполне логичным решением было введение промежуточного звена, которое принимало запросы от клиентов, проверяло бы их и транслировало бы дальше внутрь корпоративной сети. Таким образом все хранилища будут закрыты извне. Будет только один шлюз/вход, через который будут проходить все запросы. Такой вход гораздо легче контролировать.

Универсальный клиент
Самое очевидное решение — нужна программа, которая “на лету” исполняет код, который в нее загружается. И это браузер.
Со временем браузеры совершенствовались, язык отображения HTML (Hyper Text Markup Language — язык разметки гипертекста), становился более сложным и изощренным. Появились javaScript, который мог исполнять браузер и таким образом сделать страницы более интерактивными.В версии HTML 4.0 появились CSS (Cascade Style Sheets — каскадные таблицы стилей). Потом появилась и прижилась технология AJAX. В общем браузер сегодня — это действительно универсальный клиент.
Но даже в очень простой реализации, которая когда-то была, браузер может загрузить данные, отобразить их, позволить что-то ввести в текстовые поля и отправить данные на web-сервер.

Сервер приложений
Он-то и должен предоставить “возможность принимать запросы от универсального клиента и писать логику для распределенных источников информации”.

В простом варианте это была обычная программа, которая, говоря терминами предыдущей статьи о сетевом взаимодействии, открывала серверный сокет и ждала запросы от внешних программ. Причем в ранних версиях у такого сервера не было даже многопоточности — она просто порождала свою копию в виде отдельного процесса и передавала туда обработку запроса.
Также программа предоставляла возможность писать другие программы, которые могли быть ею запущены для исполнения запросов.

Вот такая вот схема выстроилась для решения ПОТРЕБНОСТИ.
Браузер, как универсальный клиент, который обратиться к серверу приложений (по определенному протоколу).
Сервер приложений в свою очередь запустит на выполнение программу, которая написана по определенным правилам.
Программа обработает запрос, что-то поменяет, что-то считает и отдаст результат (какие-то данные) в браузер, который это отобразит.

Как видим, мы пришли к очень неплохим выводам:

1. Нам нужен браузер — и он у нас есть
2. Нам нужна программа для приема сетевых запросов
3. Программа должна предоставлять возможности интегрировать в нее другие программы, которые будут решать специализированные задачи.

Сервер приложений

Программа, которая просто принимает сетевые запросы может быть очень простой. Мы даже написали что-то подобное в разделе “Сетевое взаимодействие”. Наш серверный сокет принимал запросы и отвечал — можно сказать, это наш сервер. Убогонький конечно, но все-таки — СЕРВЕР :).
Я пишу все эти слова с целью донести мысль о том, что сервер приложений в общем-то — это программа. И ее надо запускать. Просто запускать, как обычную программу. Не трепещите перед термином “сервер приложений” — это ОБЫЧНАЯ программа. Возможно, что сложная, с большим количеством возможностей, которые надо изучать. Но это ОБЫЧНАЯ программа.

Конечно же наша простенькая программа мало что может. Она слушает TCP-порт и отвечает достаточно просто. Но ее можно научить отвечать что-то более разумное для какого-то количества входных слов — научить определенному “протоколу общения”.
Помните, мы говорили об этом термине. Клиенту и серверу (да в общем-то любым двум программам) надо научиться разговаривать друг с другом. Сервер должен понимать, что клиент у него спрашивает, а клиент должен понимать, что сервер ему отвечает. Это и есть протокол. Набор слов (байтов), которые можно использовать для общения. Чуть ниже мы подробно поговорим о протоколе для Интернет, а пока продолжим рассуждать о сервере приложений.

Итак, мы уже увидели, что сервер приложений должен уметь слушать TCP-порт. А что еще должен уметь сервер ? На самом деле, если немного пофантазировать, то вы придумаете много чего.
Раз уж только что говорили о протоколе — ну как минимум сервер приложений должен уметь общаться по этому протоколу.
Немного помечтать и вот оно — наверняка многим программам, которые будут “подключаться” к нашему серверу (встраиваться в него, устанавливаться на нем, развертываться на нем) потребуется соединение с базой данных. Почему-бы не научить сервер приложений создавать уже готовые соединения и давать их “в аренду” другим ? А еще организовать механизм распределенных транзакций — чтобы можно было в рамках одной транзакции работать с несколькими базами данных.
Или вот еще — нужна система безопасности. Почему бы серверу приложений не обладать такой системой. Все остальные могут ею пользоваться.

А если встроить систему установки приложений ? А консоль управления параметрами сервера и тех же приложений ? И использовать для консоли браузер.
Можно вспомнить о загадочных web-services, SOAP и ReST. Что-то я увлекся 🙂

В этих рассуждениях вся суть сервера приложений. Эта программа предоставляет широко используемые сервисы. Уже заранее готовые и настроенные.
Если говорить о Java, то упрощенно, это уже готовый набор объектов или классов, которые можно использовать. Эти объекты создает сервер приложений по настройкам и любая программа, которая на сервере установлена, может их использовать.

Теперь не надо писать для каждого приложения кучу кода или подключать кучу библиотек — они уже есть. С любовью отобранные, оттестированные. Мечта, да и только. И мы увидим некоторые из этих возможностей уже достаточно скоро.

Немного забегая вперед, скажу, что список этих классов, интерфейсов и прочая имеет даже свое название — Java Enterprise Edition Specification — Java EE.
На данный момент самая последняя версия — 7.0. Готовится версия 8.0, но пока официально ее не публиковали.

А пока давайте вернемся к земному — погорим о протоколе.

Протокол HTTP

HTTP — HyperText Transfer Protocol — протокол передачи гипертекста. Основа основ на мой взгляд. Если вы не понимаете, что и как он делает, то многое в программировании для Интернет вы будете постигать с большим трудом.
Мы уже с ним встречались с разделе “Сетевое взаимодействие” — в первой программе мы запрашивали файл. И для запроса использовали именно HTTP.
На само деле протокол достаточно простой. Важно помнить, что это обычный текст.
Если говорить упрощенно, то по сути вы создаете TCP-соединение (дозваниваетесь до секретаря — шутка, ха-ха-ха) и начинаете писать в поток вывода слова. В ответ вам приходят тоже слова и иногда байты. Но о байтах будем говорить, когда вы станете опытнее.

Здесь я рассматриваю протокол HTTP версии 1.1, потому что сейчас реализация классов Java предназначена именно для этой версии. Существует HTTP версия 2.0 и для его реализации в Java есть предварительная документация, но судя по ней, принципиальных отличий для Java-разработчика не появится. Это вполне логично, т.к. версия HTTP 2.0 (судя по документации) поддерживает все возможности HTTP 1.1. Дополнительные возможности — вероятно будут. Мне важно, что основная структура и правила останутся.

Для особо любопытствующих могу предложить отправиться в самостоятельное плавание по RFC (Request for Comments).

Протокол HTTP 1.1:
RFC7230: HTTP/1.1, part 1: Message Syntax and Routing
RFC7231: HTTP/1.1, part 2: Semantics and Content
RFC7232: HTTP/1.1, part 4: Conditional Requests
RFC7233: HTTP/1.1, part 5: Range Requests

RFC7234: HTTP/1.1, part 6: Caching
RFC7235: HTTP/1.1, part 7: Authentication

Протокол HTTP 2.0:
Hypertext Transfer Protocol Version 2 (HTTP/2).
HPACK: Header Compression for HTTP/2

Чтение достаточно сложное — надо иметь навык и терпение читать RFC. Но могу сказать, что игра стоит свеч. Научившись читать и понимать RFC, вы получите очень ценный источник информации. Не предлагаю делать это прямо сейчас — просто завяжите себе узелок на память.

Основные части HTTP 1.1
Как я уже говорил, HTTP представляет собой текст, который включает важные группы слов.
Каждая группа используется для решения определенных задач. Задачи настолько важны, что несмотря на отличие текстового вида HTTP 1.1 и HTTP 2.0, эти группы обязательно присутствуют в обоих протоколах. Т.е. текст выглядит по-разному, но эти группы есть в обоих.
Наверно, я больше не буду упоминать о HTTP версии 2.0 — буду все рассматривать на примере HTTP 1.1. Но вы держите в голове, что эти важные группы есть в обоих.

Давайте на них и остановимся. Возьмем для пример ту команду, которую мы посылали в разделе "Сетевое взаимодействие" — там мы посылали строку

```
"GET /network.txt HTTP/1.1\r\n" + "Host:java-course.ru\r\n\r\n"
```

Итак, первая часть — **GET /network.txt HTTP/1.1\r\n**.
Здесь аж три группы:

1. GET — это метод/команда. Вы, как клиент, посылаете серверу команду, которая позволяет ему понять, какое действие вы хотите от него. Список методов мы посмотрим чуть позже — пока просто запомним, что есть важный элемент — МЕТОД
2. /network.txt — имя ресурса. Сервер должен понять, к какому ресурсу вы хотите применить указанный метод. Вы всегда указываете ресурс
3. Версия протокола, по которому вы хотите общаться. Это нужно для корректного взаимодействия сервера и клиента (браузера). Хотя с точки зрения Java-программирования это не столь важно. На мой взгляд.

Обратите внимание, что эта часть заканчивается двумя символами — Возврат Каретки (Carriage Return — CR — \r) и Перевод Строки (Line Feed — LF — \n).

Вторая часть — **Host:java-course.ru\r\n\r\n** — содержит всего одну группу, но эта группа в отличии от предыдущих, может содержать много слов.
Эта группа называется Заголовки (Headers).
Содержит список строк, которые разделяются друг от друга \r\nю В нашем случае у нас всего один заголовок:

```
Host:java-course.ru
```

Но их может быть много (обычно так и есть). Например, вот что отсылает мой браузер (и это не все заголовки)

```
Host: java-course.ru
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Accept-Encoding: gzip, deflate
Accept-Language: ru-RU,ru;q=0.9,en-US;q=0.8,en;q=0.7
```

Сейчас не будем разбирать, что каждый значит. Важно обратить внимание, что каждая строка делиться на две части через двоеточие. Левая сторона — имя заголовка, правая — его значение. Заголовки позволяют послать дополнительную информацию, которая может быть использована для улучшения работы.
Как видите, группа заканчивается двойным набором \r\n.

Есть еще одна группа — это данные ПОСЛЕ заголовков. В методе GET такой группы не предусмотрено. Поэтому сервер, прочитав двойной \r\n, приступит к обработке запроса. Если же такая группа есть — значит сервер ее тоже прочитает и будет как-то использовать.

В случае с Java, сервер разберет по частям весь наш текст, создаст объект специального класса и разложит все указанные группы по его полям. Вот этот момент стоит повторить — сервер РАЗБЕРЕТ текст, СОЗДАСТ объект специального класса и РАЗЛОЖИТ все части нашего запроса по полям объекта. Мы обязательно к этому еще вернемся, но мне бы очень хотелось, чтобы вы об этом помнили — у нас есть объект с данными из запроса.

Кроме метода GET существует еще POST, PUT, DELETE, OPTIONS, HEAD, TRACE.
Вполне достаточно пока помнить о четырех — GET, POST, PUT, DELETE.
Первые два — самые распространенные. Мало того, до недавнего времени браузеры только их и могли отправить. Отличия этих методов мы разберем позже — не хочу, чтобы вы из-за деревьев леса не увидели.

Кстати, о браузерах. Самое время разобрать, что же происходит, когда вы в строке браузера указываете ссылку — например “http://java-course.ru/network.txt”

Во-первых, браузер выделяет адрес сервера, к которому пойдет запрос. В нашем случае — “java-course.ru”. Т.к. мы не указали порт, то по умолчанию используется порт 80. Можно сказать, что на самом деле мы указали адрес вот так:

```
http://java-course.ru:80/network.txt
```

Во-вторых, после того, как удалось соединиться, браузер честно посылает тот текст, который мы разбирали чуть выше. А именно:

```
GET /network.txt HTTP/1.1
Host:java-course.ru
```

В принципе, клиент работу по отправке запроса закончил. Теперь сервер должен ответить на наш запрос.
Потерпите еще немного — я постараюсь больше на мучить вас большим количеством занудной информации, но она действительно очень важна.
Итак, что же нам отвечает сервер. Если вы запускали программу из раздела “Сетевое взаимодействие”, то могли видеть ответ. Он где-то такой:

```
HTTP/1.1 200 OK
Date: Tue, 26 Dec 2017 14:24:34 GMT
Server: Apache
Last-Modified: Sat, 16 Dec 2017 20:43:22 GMT
Accept-Ranges: bytes
Content-Length: 40
Content-Type: text/plain

Congratulation !!!
Socket is working !!!
```

Пока обратим внимание на первую строчку — HTTP/1.1 200 OK. Сервер нам отвечает, что он работает по протоколу HTTP 1.1, далее следует код — 200 значит. Что все замечательно. Кодов немало, их тоже можно найти, но пока просто запомним — возвращается код результата. Далее следуют заголовки наподобие тех, которые мы отправляли серверу. И после двойного перевода строки непосредственно данные.

На этом все. Выдохнули ? Я сам с большим трудом читаю такого рода опусы, но мне было важно их написать — даже если вы сейчас читали не внимательно, уже очень скоро вы начнете возвращаться к этому разделу и ощущать радость от понимания. (Ну я на это надеюсь :))

А пока еще раз подведем итоги.

1. Клиент (браузер) подключается к серверу по TCP
2. Клиент (браузер) отправляет текст с определенным описанием того, что он хочет получить. Текст должен соответствовать протоколу HTTP.
3. Сервер разбирает это текст на части и если это сервер для Java, формирует из этих частей объект Java определенного класса
4. Сервер возвращает ответ клиенту (браузеру)
5. Клиент (браузер) отображает ответ от сервера

Устанавливаем Web-сервер Tomcat

Для того, чтобы начать создавать web-приложения, нам потребуется web-сервер. Т.е. программа, которую мы запустим и она начнет принимать HTTP-запросы. В эту программу мы начнем устанавливать наши приложения.

Для начала есть смысл использовать очень простую программу, которую легко устанавливать и использовать. Я решил остановиться на программе **Tomcat**.

Ранее мы говорили о сервере приложений, чуть выше — о web-сервере. Я также использовал понятие “сервер” без каких-либо добавлений. Чтобы вы не запутались, давайте разберемся, кто есть кто в мире Java. Пока в достаточно упрощенном варианте.

Итак, web-сервер — это программа, которая включает определенный набор классов (библиотек), которые позволяют этой программе принимать HTTP-запросы, разбирать его текст на составные части и на которой можно установить приложения, обрабатывающие такие запросы (наша цель как раз научиться писать такие приложения).
Сервер приложений — это по сути расширенная версия web-сервера. Он не только умеет делать то, что умеет web-сервер, но к тому же обладает более широким набором классов. Эти классы позволяют еще больше.
Когда же я пишу “сервер”, то это просто для краткости. Из контекста (я надеюсь) должно быть понятно, о каком именно сервере (сервере приложений или web-сервере) идет речь.

Что же нам потребуется для запуска Tomcat.? Давайте по шагам:

1. Установить переменную JAVA_HOME. Не делали ? Тогда бегом в раздел Установка JDK и там смотреть как это сделать для Windows. Если вы пользователь Unix-системы, то обычно эта переменная прописывается в файле **.bashrc**
2. Зайти на сайт Tomcat. Слева есть раздел **Download**. Выберите в нем самую последнюю версию.
3. На открывшейся странице найдите раздел **Binary Distributions**. В нем подраздел **Core**, где представлены разные варианты скачиваний. Я рекомендую скачать **ZIP**
4. Скачанный архив вы можете просто распаковать в какую-либо директорию, Как я уже неоднократно писал — я предпочитаю создавать директорию Java и туда ставить все. Tomcat запакован уже с директорией — вот я ее и копирую в Java
5. С установкой все. Осталось проверить. Что Tomcat запускается

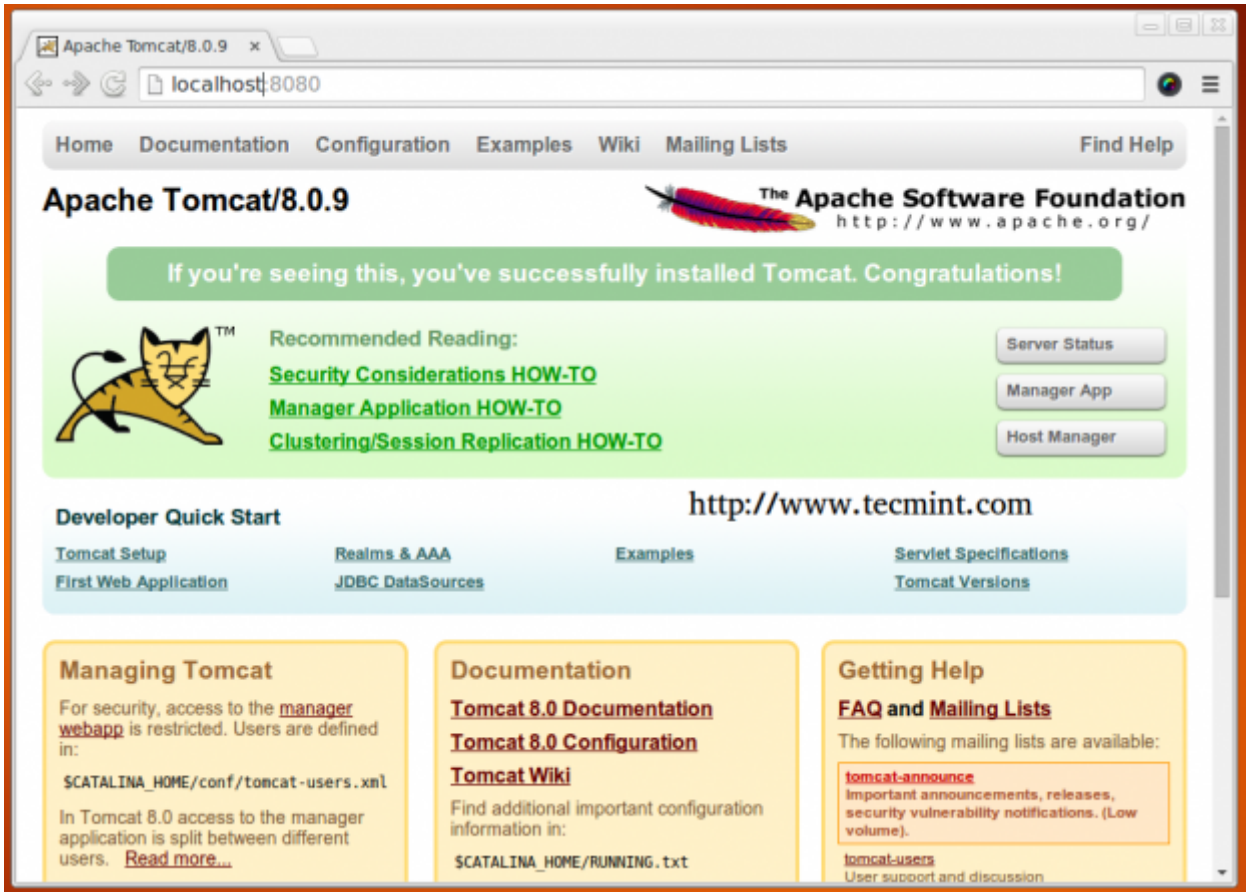
Для запуска зайдите в директорию, в которой установлен Tomcat (в дальнейшем я буду ссылаться на эту директорию как TOMCAT_HOME) и там зайдите в каталог **bin**. Найдите там файл **startup.bat** и запускайте. У вас должно появиться черное командное окошко, в котором будут появляться разные строки. Если все в порядке, то во-первых вы не должны увидеть там стек-трейсов (которые появляются при генерации исключений) и во-вторых — в конце должно появиться что-то подобное

```
INFO: Server startup in 8828 ms
```

Запустите браузер и введите в строке для адреса

```
http://localhost:8080/
```

Если у вас все было сделано правильно, то вы должны увидеть главную страницу Tomcat.



Поздравляю. Вы сделали первый шаг к началу web-программирования.
Но надо научиться останавливать Tomcat. Для этого запустите программу **shutdown.bat** из того же каталога.
Если вдруг программа не завершилась — “убейте” процесс, просто закрыв окно.

Что мне нравится в Tomcat, так это легкость установки. Если совсем накосячили с какими-то установками — удалите всю директорию с Tomcat и распакуйте архив заново. Без проблем.

Первое приложение

У меня был выбор — продолжать дальше рассказывать про идеологию или сразу сделать простое web-приложение и начать объяснять на его основе, что и как.
Наверно идеологии вам пока достаточно — давайте попробуем сразу сделать пример, установить его на Tomcat и проверить его работоспособность.

Проект мы будем делать на Maven и это замечательно — тогда мне не потребуется описывать какие-то пункты меню, нажатие кнопок и использовать поясняющие рисунки.
Если вы еще не читали про Maven — тогда вам обязательно надо смотреть раздел Переезжаем на Maven.

Для тех же, кто прочитал (и сделал самостоятельную работу — вы тогда большие молодцы), представляю код для нашего крохотного проектика. Он состоит из двух файлов:
Первый файл — файл описания проекта — **pom.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>edu.javacourse.web</groupId>
    <artifactId>simple</artifactId>
    <version>1.0</version>
    <packaging>war</packaging>

    <dependencies>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-api</artifactId>
            <version>7.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.7.0</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <version>3.2.0</version>
                <configuration>
```

```
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

И файл с нужным нам классом — мы его подробно разберем чуть позже. Будьте внимательны — он должен находится в каталоге **src/main/java/edu/javacourse/web**. Если вы читали про Maven, то должны догадаться. Итак, второй файл **SimpleServlet.java**

```
package edu.javacourse.web;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet(name = "SimpleServlet", urlPatterns = "/first")
public class SimpleServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        resp.getWriter().append("Hello, world !!!");
    }
}
```

Прежде чем приступить к подробному разбору, что и как устроено, давайте попробуем собрать наш проект, установить на Tomcat и проверить его работоспособность.

Чтобы вы были уверены в правильном наборе файлов и каталога для вашего maven-проекта, вот такая структура каталогов и файлов должна у вас быть:

```
-src
    -main
        -java
            -edu
                -javacourse
```

```
                    -web
                        -SimpleServlet.java
        -resources
-pom.xml
```

Можете вручную создать такую структуру каталогов и потом открыть в том же NetBeans или Intellij IDEA

Команда для сборки нам известна:

```
mvn clean package
```

Если вы не наделали ошибок, то после выполнения сборки, которая должна закочниться словами **BUILD SUCCESS**, в каталоге проекта появится директория **target** и в ней файл **simple-1.0.war**. Расширение говорит, что это web-архив — WebARchive. Наличие этого файла — наша цель, грандиозный результат сборки.
Если вы работаете в какой-либо IDE, то вам удобно будет научиться запускать Maven прямо там. Но лично я достаточно часто предпочитаю запускать Maven из командной строки. Сложилась эта привычка по разным причинам, и вы можете считать меня ретроградом, но пока эта привычка себя оправдывает.

Для установки вам достаточно скопировать этот файла в каталог **TOMCAT_HOME/webapps**. Это не единственный способ установки, но на мой взгляд, для начинающих это то, что надо. Забивать себе голову какими-то административными консолями — да гори оно все синим пламенем.

Скопировали ? Тогда запускайте Tomcat и ждите сообщения о старте сервера. Если Tomcat уже запущен — подождите несколько секунд — по умолчанию Tomcat должен автоматически “подхватить” ваше приложение.
Надеюсь, что у вас в сообщениях Tomcat нет ошибок. Если есть — попробуйте разобраться. Хотя вероятность ошибки крайне мала — наша программа настолько элементарна, что должна установиться и запуститься без каких-либо проблем.
Если все-таки ошибка произошла и вы не понимаете почему — попробуйте повторить все наши шаги. Попробуйте взять более старую версию Tomcat — ребята иногда выкладывают всякие бета-версии и они работают весьма странно. Если все-таки не работает — читайте дальше и может быть чуть позже у вас получится обнаружить причину.
В случае успешной установки в каталоге **TOMCAT_HOME/webapps** должен появиться каталог **simple-1.0**.

Теперь проверим работоспособность нашего приложения. Для этого вам надо запустить браузер и в строке адреса набрать следующее:

```
http://localhost:8080/simple-1.0/first
```

Обратите внимание, что Tomcat использует не стандартный порт “80”, а порт “8080”. Браузер должен отобразить текст:

```
Hello, world !!!
```

Удалось ? Отлично. Ваше приложение работает. Теперь самое время поговорить о том, что же мы там такого написали для того, чтобы это все заработало.

Разбор полетов

Начнем мы с файла **pom.xml**. В нем есть несколько строк, которые требуют объяснения.
Первая часть находится в самом начале — выглядит она так:

```
    <packaging>war</packaging>
```

Предназначена она для достаточно простой цели — результатом сборки нашего проекта должен стать не файл типа **JAR**, а файла **WAR** — веб-архив. Формирование файла — это отдельная фаза для Maven — фаза **package**. В момент выполнения этой фазы используется плагин. В нашем случае это плагин **maven-war-plugin**. И в нашем случае нам даже потребовалось дать ему некоторые указания — в нижней части файла **pom.xml** мы можем увидеть вот такие строчки:

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <version>3.2.0</version>
                <configuration>
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                </configuration>
            </plugin>
```

Зачем же нужны эти строки ? В веб-приложении на Java существует достаточно важный файл — **web.xml**. Когда-то его наличие в архиве было обязательным. По умолчанию плагин проверяет его наличие и выдает ошибку в случае его отсутствия.
С появлением Java EE версии 6 появилась возможность в некоторых случаях обойтись без этого файла. И наша настройка в данном случае говорит плагину: “Эй, приятель. Я тут так все сделал, что файл web.xml не потребуется. Не ругайся пожалуйста”. В общем-то и все.
Что касается фрагмента для другого плагина — **maven-compiler-plugin** — то мы его уже рассматривали в статье про Maven. Но чтобы вам не отвлекаться, напомню — мы хотим использовать Java SE версии 1.8, об этом и сообщаем ему для настройки компиляции.

Осталось разобрать вот этот фрагмент:

```
    <dependencies>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-api</artifactId>
            <version>7.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
```

Здесь мы подключаем библиотеку, которая содержит определение классов, которые существуют в спецификации Java EE версии 7.0. В принципе при рассказе о Maven мы проговаривали эту идею — подключение библиотек. При разборе нашего класса **SimpleServlet** мы рассмотрим используемые классы, а пока сосредоточим внимание на строке:

```
            <scope>provided</scope>
```

Тэг **scope** позволяет настроить правила включения внешних библиотек в веб-архив.
Попробуйте убрать эту строку и собрать проект заново командой

```
mvn clean package
```

Откройте war-файл с помощью любой программы работы с архивированными файлами — на самом деле наш файл является самым обычным ZIP-архивом. Вы должны увидеть две директории:

```
META-INF
WEB-INF
```

Зайдите в **WEB-INF**. Там вы увидите два каталога

```
classes
lib
```

Каталог **classes** содержит наш класс — можете посмотреть. А вот каталог **lib** будет содержать библиотеки, которые вы хотите подключить к вашему веб-приложению. Несомненно, какие-то библиотеки вам надо будет подключать. Но вот те библиотеки, которые вы можете наблюдать в нашем каталоге **lib** в данный момент нам совсем не нужны. Почему ? Потому, что Tomcat, как сервер, который поддерживает функциональность по стандарту (спецификации для web-сервера — мы чуть выше говорили о роли web-сервера) УЖЕ содержит данные библиотеки. Точнее не сами архивные файлы, а набор тех классов, которые в них есть. Он может их разбить на другие архивы, но набор классов должен быть соблюден.
Т.е. нашему приложению в общем-то для работы эти классы (архивы) у себя хранить совсем не надо. Но вот для компиляции они нам нужны. И строка

```
1            <scope>provided</scope>
```

как раз и позволяет подсказать Maven'у — используй данные библиотеки только для компиляции, но в архив их упаковывать не надо. Слово **provided** говорит о том, что нужные классы нашему приложению предоставит сам web-сервер.
Можете вернуть строку со **scope** обратно и снова собрать наш архив. Каталог **lib** должен исчезнуть. Как видите, все достаточно логично и несложно.

Следующим для разбора берем файл **SimpleServlet.java**. Про него нам придется поговорить дольше и сам разговор будет сложнее.

Во-первых, давайте вернемся к протоколу HTTP, а точнее к команде GET. Для обращения к нашему приложению мы в строке браузера набирали вот такую строку

```
1 http://localhost:8080/simple-1.0/first
```

Эта строка очень информативна и позволит нам разобрать несколько важных вопросов, которые приоткроют нам веб-программирование на Java.

В первой части у нас есть вот это: **http://localhost:8080**. Часть **http://** говорит о том, что браузеру надо будет использовать протокол HTTP. Нередко в браузере, если написать просто **localhost:8080**,он по умолчанию будет использовать это протокол.
Далее у нас следует часть **localhost:8080**. Это указание, к какому хосту и к какому порту надо сделать TCP-соединение. Мы об этом тоже говорили совсем недавно.
И наконец последняя часть **/simple-1.0/first**. На мой взгляд, это как раз самое интересное. Во-первых, давайте напишем команду HTTP, которая будет послана нашему серверу на основании этой строки. Для HTTP 1.1 это будет вот так:

```
1 GET /simple-1.0/first HTTP/1.1
2 Host:localhost
```

Возможно, что заголовков будет больше, но то, что я написал — это точно будет. Самое важное сейчас заключено в строке **/simple-1.0/first**. В ней содержится ИМЯ РЕСУРСА. Т.е. мы посылаем нашему серверу запрос с указанием ИМЕНИ. Больше ничего там нет. Как вы уже догадались, имя ресурса — это строка после **localhost:8080**.
Теперь задача сервера — понять, а что за ресурс мы запрашиваем.

Как мы уже говорили, веб-сервер, поддерживающий Java, умеет слушать сообщения, которые приходят на какой-то порт. У нас это порт 8080. Т.е. там открыть серверный сокет. На этот сокет пришло вот такое сообщение, в котором заключено имя ресурса. Что же теперь делать нашему серверу, как искать этот ресурс.

Веб-сервер должен уметь устанавливать (регистрировать, деплоить — deploy) приложения. Наш WAR-файл — это как раз и есть веб-приложение. Если вы успешно запустили наше приложение, то заглянув в каталог **TOMCAT_HOME/webapps** вы увидите каталог **simple-1.0**. Его содержимое соответствует содержимому архива. В такой форме Tomcat устанавливает (регистрирует) приложения у себя. Другие веб-серверы это могут делать иначе. Но в любом случае все они делают это важное дело — они устанавливают (регистрируют) приложения.
Регистрация заключается в том, что приложение проверяется на правильность его содержимого и, что сейчас крайне важно, веб-сервер в упрощенном виде прописывает для каждого приложения специальный параметр — **Context Path**.
Это имя приложения в системе (внутри веб-сервера). Именно по этому пути он узнает из HTTP-запроса, что запрос направлен к конкретному приложению.
В Tomcat значение **ContextPath** в упрощенном случае равен имени каталога в который распаковано приложение. Т.е. наше приложение зарегистрировано под именем **simple-1.0**. И как мы можем видеть — это часть в имени ресурса. Первый шаг сделан. Но у нас осталась еще одна часть — **first**. Что же она значит ?

Когда-то давно было сделано (придумано) так: все запросы в java-приложении будут обрабатывать наследники одного класса или классы, которые реализуют один интерфейс.
Итак, у нас выбор всего из одного класса и одного интерфейса.

- Класс **Servlet**
- Интерфейс **Filter**

В нашем случае мы использовали класс **Servlet**. Точнее, мы использовали наследника этого класса — **HttpServlet**.
Самое время обратиться к нашему файлу **SimpleServlet.java**. Как видите, мы создали класс **SimpleServlet**. Я не буду сейчас приводить весь код — сосредоточимся буквально на двух строчках.

```
1 @WebServlet(name = "SimpleServlet", urlPatterns = "/first")
2 public class SimpleServlet extends HttpServlet
```

Вторая строка уже вам понятна — мы создали класс, который наследуется от **HttpServlet**. Но зачем, спросите вы. Как я выше указал, ВСЕ запросы обрабатываются классами — наследниками от **Servlet**.

Еще раз внимательно проговорите — ЗАПРОС отправляется к классу-наследнику **Servlet**. Т.е. Сам веб-сервер по первой части, а именно по **ContextPath** (у нас это **simple-1.0**) будет пытаться найти ТОЛЬКО ТЕ КЛАССЫ, которые являются наследниками **Servlet**.

С помощью механизма **reflection** это сделать несложно. Веб-сервер при регистрации приложения может спокойно пробежаться по всем классам и понять, какие из них наследники **Servlet**.
Но как вы возможно догадываетесь, в нашем приложении может быть много таких классов. Какой же из них должен обработать конкретный запрос ?
И тут нам на помощь приходит первая строка, а именно АННОТАЦИЯ

```
@WebServlet(name = "SimpleServlet", urlPatterns = "/first")
```

В этой аннотации мы указали два параметра:

1. **name** — он в общем может быть произвольным и пока можете о нем не думать
2. **urlPatterns** — а вот это очень ВАЖНО. Здесь указывается массив строк, которые сервлет регистрирует, как имя, на которое он отзывается. Как видите имен может быть больше одного

Итак, смотрите, что у нас получается. В запросе GET содержится полное имя ресурса (в нашем случае **/simple-1.0/first**). Это имя состоит из двух частей — **ContextPath** приложения (у нас **/simple-1.0**) и **urlPatterns** одного из сервлетов (у нас **/first**), которые есть в этом приложении.

Вот собственно и весь алгоритм, как Tomcat определяет нужный класс — он находит приложение и в нем находит сервлет. Дальше он передает управление нужному сервлету, получает ответ и передает это обратно по сети в браузер (или в другую программу). Как видите, все опять достаточно логично и не сложно.
Но что значит “передать управление и получить ответ” ? Давайте разбираться дальше 🙂

Под капотом сервлета

Итак, Tomcat нашел нужный класс. С этим мы в простом виде разобрались. С этого момента у нас есть еще несколько задач.
Создание объекта типа сервлет — да-да, Tomcat должен создать экземпляр сервлета. Создание осуществляется через рефлекшн и является одним из этапов так называемого **жизненного цикла сервлета.- servlet life cycle**. О нем мы поговорим позже, а пока просто примем к сведению — веб-сервер создает экземпляр сервлета.
Передача управления — здесь все на самом деле несложно — веб-сервер просто вызывает определенный метод, куда передает параметры. Метод называется **service**. В нашем коде мы его сейчас не видим, но он есть 🙂
Если вы хотите использовать стандартные подходы к обработке запросов, то может вам его менять и не придется.
Основная задача этого метода — определить, какая команда содержалась в HTTP-запросе (GET, POST, PUT и т.д.) и вызывать соответствующий метод — **doGet, doPost, doPut** и т.д.

Если мы посмотрим на код нашего сервлета **SimpleServlet**, то мы можем видеть. Что я переопределил метод **doGet**.

```
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        resp.getWriter().append("Hello, world !!!");
    }
```

T.е. теперь цепочка вызова выглядит вот так: Tomcat получает текст, разбирает его на части, определяет приложение и в нем сервлет, делает вызов метода **service**. Внутри метода определяется. Что получена команда GET — занчит управление передается методу **doGet**.

Ну что же — практически добрались до самого конца. Осталось разобраться с непонятными параметрами **HttpServletRequest** и **HttpServletResponse**

Если вы внимательно посмотрите, то обнаружите, что это не классы — это ИНТЕРФЕЙСЫ. Но зачем ?
Хитрость (и гениальность) решения в том, что наше приложение на самом деле НЕ ЗНАЕТ, на какой веб-сервер его будут устанавливать. И как в этом случае договариваться ? Как обычно — использовать интерфейсы.
T.е. разработчики Tomcat (или иного сервера) знают, что они должны “упаковать” пришедший запрос в объект такого класса, который реализует интерфейс **HttpServletRequest**.
Сосредоточьтесь — веб-сервер создает объект, записывает туда все составные части запроса — команду (метод), заголовки, какие-то параметры и передает его в наш сервлет. Но т.к.. наш сервлет не знает, на какой веб-сервер его установят, то он и не собирается знать класс, который будет использовать веб-сервер.
Но зато сервлет занет ИНТЕРФЕЙС. И этот интерфейс — это КОНТРАКТ, который обязаны соблюдать ВСЕ веб-серверы.
С другой стороны — ВСЕ сервлеты знают, что им будут передаваться объекты, которые ВСЕГДА реализуют интерфейсы **HttpServletRequest** и **HttpServletResponse**.
.
Сейчас я не хочу обрушивать на вас все особенности этих параметров — просто отметим, что **HttpServletRequest** содержит информацию о запросе (метод, заголовки и т.д.) а **HttpServletResponse**.позволяет вам “отдать” информацию из сервелат обратно.
В данной программе мы использовали очень простой способ — у объекта **HttpServletResponse** есть вызов **getWriter()**. Это вызов возвращает (вы уже готовы восхищаться ?) ПОТОК ВЫВОДА.- а именно **PrintWriter**. Мы уже с ним встречались при обсуждении сетевых взаимодействий. И у этого потока есть возможность записать строку символов — что мы с успехом и сделали.

Ну что — было сложно ? Надеюсь, что несмотря ни на что, вы выдержали и смогли что-то понять (а если еще запустили пример, то совсем молодцы). Можете собой гордиться — вы успешно начали свое путешествие по веб-программированию для Java.

4 comments to *С чего начинается Web*

- 
  Март 10, 2018 at 00:25
  Иван says:

Небольшая опечатка в файле pom.xml строка 7

du.javacourse.web

du вместо edu

edu.javacourse.web

Reply

- 

  Март 10, 2018 at 13:56
  admin says:

  Спасибо.

  Reply

Март 15, 2018 at 19:19
prim says:

Для запуска Tomcat обязательно надо установить значение для системной переменной CATALINA_HOME. Значение — это директория, в которой находится распакованный Tomcat. У меня, CATALINA_HOME=C:\Java\apache-tomcat-8.5.29

Reply

- 

  Март 17, 2018 at 10:56
  admin says:

  Это не обязательно. Если посмотреть запускающий файл startup.bat, то там можно увидеть проверку на наличие этой переменной и если она не установлена, то ставится текущая.

  Reply

Leave a reply

Comment

You may use these HTML tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code class="" title="" data-url=""> <del datetime=""> <em> <i> <q cite=""> <s> <strike> <strong> <pre class="" title="" data-url=""> <span class="" title="" data-url="">`

Имя *

E-mail *

Сайт

– семь = ноль

Add comment