

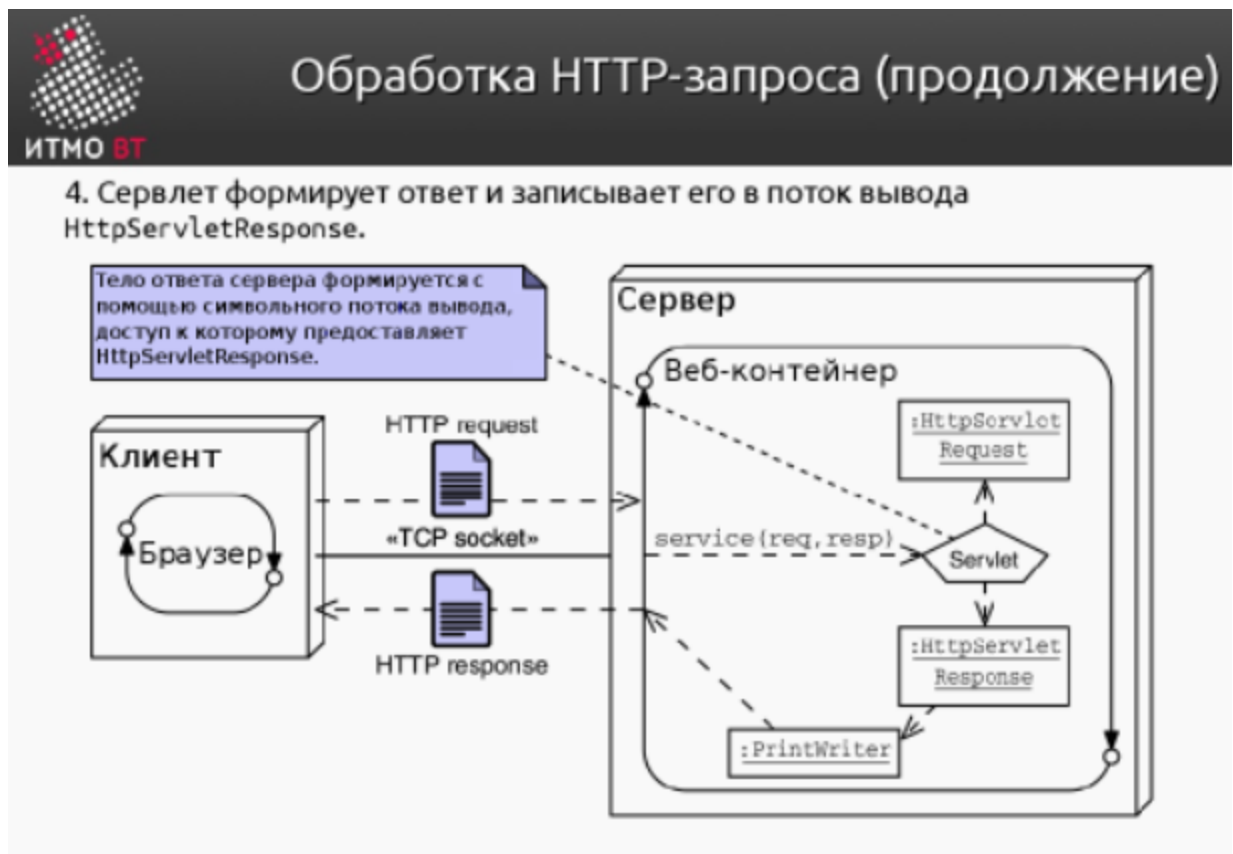
Лекция #10. Разработка сервлетов.

Схема работы сервлетов.

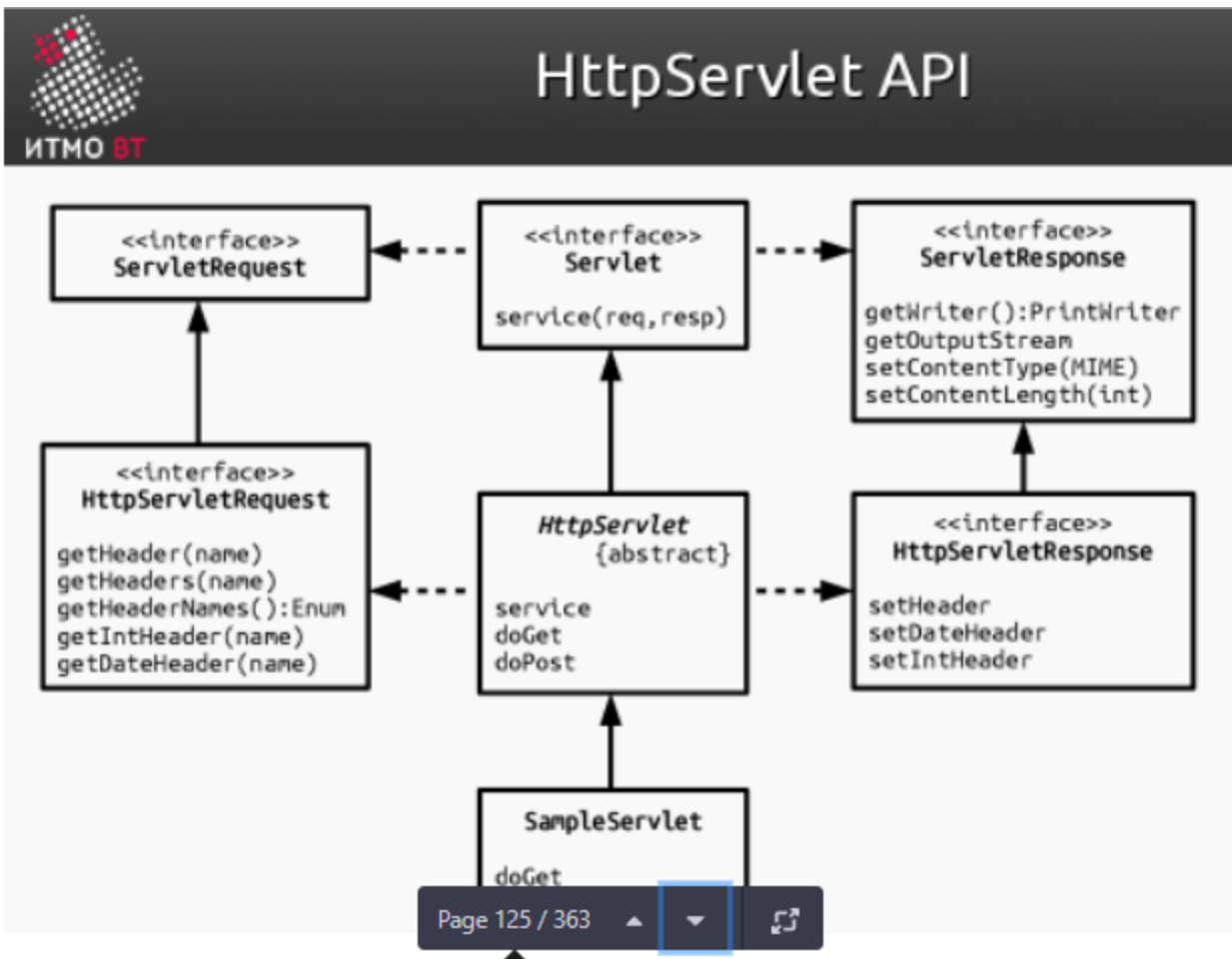


Есть машина пользователя с браузером. Браузер инициирует TCP соединение и отправляет HTTP запрос. Сервер состоит из веб-контейнера (который на самом деле находится в java runtime), и статики (каких-то “заготовленных” HTML страниц. Веб-контейнер открывает TCP сокет и принимает этот запрос. В веб-контейнере есть набор сервлетов. В зависимости от запроса, вызовется нужный сервлет (это определяется по URL, на который пришел запрос). Итак. Браузер формирует HTTP запрос и отправляет его на сервер, установив TCP-соединение с сервером. То есть веб-контейнер открывает TCP сокет. Веб-контейнер формирует два

объекта: `HttpServletRequest` и `HttpServletResponse`. Эти объекты передадутся как аргументы особому методу `service()` от сервлета. Однако тут важно представить, что сервлет (о чудо!) там уже есть и загружен в рантайме. Метод, как уже было сказано ранее, запустится в отдельном потоке. Там сервлет выполняет свою логику (может, например, в базу данных сходить). Внутри сервлета есть особая заранее проинициализированная переменная `out` (по сути, это поток `PrintWriter`). На ее основе формируется страница и отдается обратно клиенту на тот же TCP сокет, с которого пришел запрос. Если сказать точнее, то веб-контейне преобразует `http`-запрос к `HttpRequest` java-объекту, и `HttpResponse` java-объект к `http`-ответу. Схематично это выглядит так:



HttpServlet API.

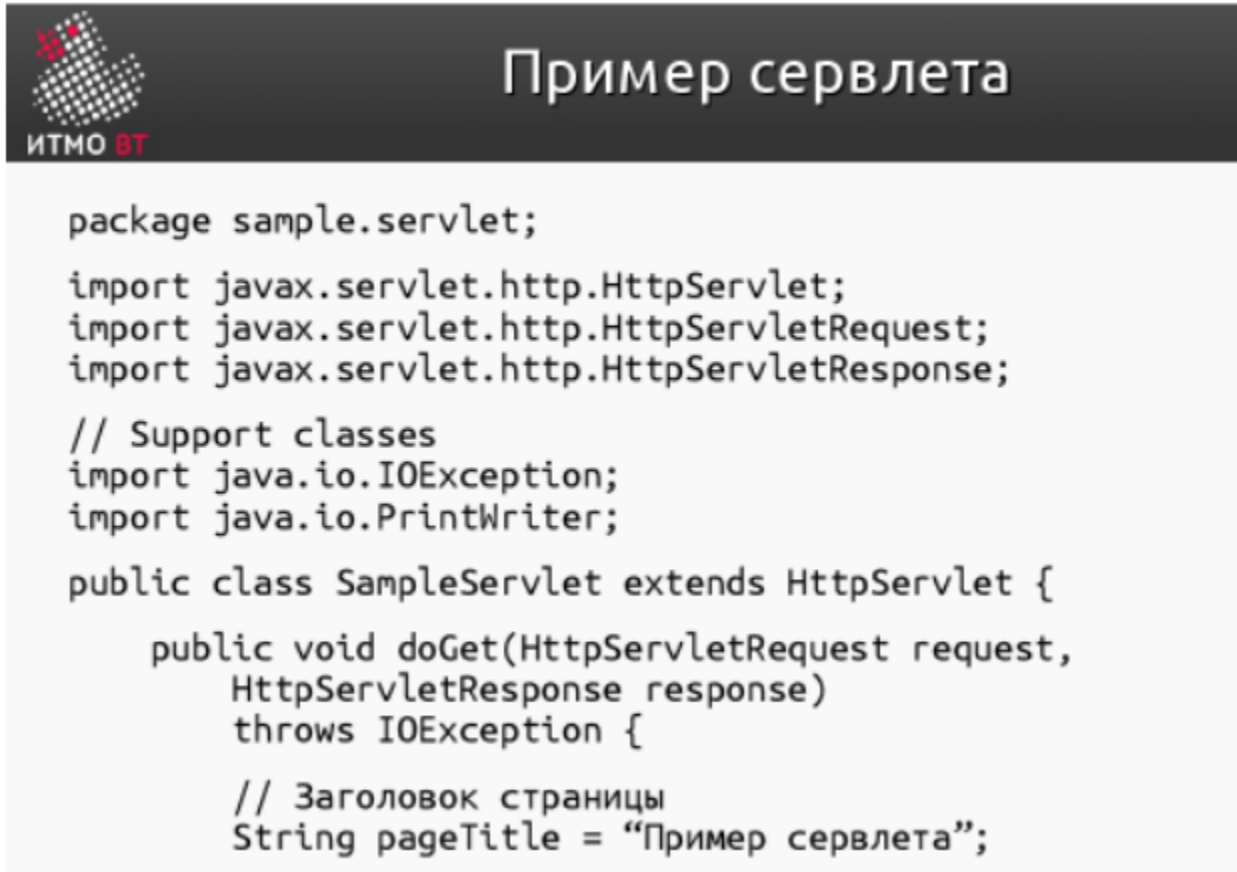


Лично мне схема не кажется картинкой для запугивания. Очень красиво. Верхняя строка - абстрактные интерфейсы. Вторая строчка - чуть менее абстрактные интерфейсы. И в самом низу какой-то конкретно написанный сервлет, и стрелочки наследования и имплементации. Красота. Идея сервлета в том, что он задумывался, как универсальный серверный сценарий. То есть. сервлет- это интерфейс.. Есть `HttpServlet` - абстрактный класс реализации работы с HTTP запросами. Но по идее (на практике в Java EE этого нет) можно написать еще и какой-то другой сервлет, например `FtpServlet`, И классы абстрактных запросов, ответов, интерфейсов к этому всему. Но опять же, в Java EE этого нет. Просто заложена теоретическая возможность расширения.

Метод `service()` реализует парсинг HTTP запроса и запускает нужные методы для каждого типа HTTP запроса, `doGet()`, `doPost()` и так далее. Однако можно переопределить `service()` так, что он будет обрабатывать только запросы одного вида, или, наоборот. любые HTTP запросы.

Возникает еще один вопрос. Все представленное на картинке - интерфейсы и абстрактные классы. Реализации, как таковой, на картинке нет. И в этом как раз заключается назначение веб-контейнера. Он предоставляет эту конкретную реализацию. И каждый веб-контейнер (WildFly из лабы, например) предоставляет свою реализацию интерфейса.

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>");
}
}
```

Данный сервлет реализует обработку только GET запросов. И формирует HTML страницу, которую потом может вернуть. Но такой способ возвращения - слезы верстальщика. Это к словам из предыдущей лекции. Делать так не надо.

Конфигурация веб-контейнера.

Однако просто написанный сервлет сам по себе запрос не получит. Нужно еще каким-то образом объяснить веб-контейнеру, что запрос нужно отправлять на этот сервлет. Веб-контейнер конфигурируется двумя способами: на уровне xml, либо на уровне аннотаций. В xml например, можно прописать несколько десерипторов, чтобы создать несколько объектов-сервлетов. Нюанс в том, что сервлеты многопоточные. Все потоки внутри одного объекта. А можно бы сделать несколько объектов. Это делается, когда нужно чтобы два разных сервлета делали разную логику независимо друг от друга. А количество потоков внутри сервлета определяется конфигурацией application server-а. Важно еще отметить, что

статичные элементы будут общие у обоих сервлетов - так как класс общий. Это прямое следствие из ООП. Примерно то же самое можно сделать с помощью аннотаций.

XML конфигурация первична. Это означает, что в случае, когда XML конфигурация веб-контейнера конфликтует с конфигурацией через аннотации, применится первое из них. Это сделано из тех соображений, что чтобы переписать аннотацию, нужно пересобрать класс, а это долго. Быстрее поправить конфигурацию и указать серверу переписать. Удобства XML так же в том, что при большом количестве сервлетов поддерживать всю систему становится сложно. Удобнее, когда вся конфигурация находится в одном файле, а не раскидана аннотациями по всем пакетам.

Жизненный цикл сервлета.

Жизненным циклом сервлета управляет веб-контейнер. Соответственно, все методы, управляющие этим самым жизненным циклом, должен вызывать только веб-контейнер. Программист определяет, что делает сервлет, а веб-контейнер - когда он это делает. То есть непосредственного взаимодействия с жизненным циклом сервлета обычно не происходит.

1. Класс загружается. Веб-контейнер вызывает class loader, который загружает указанный класс в runtime. Проверяются пути: /WEB-INF/classes/, WEB-INF/lib/*.jar, стандартные классы Java SE и классы веб-контейнера.
2. Создается экземпляр класса. Кстати говоря, при конфигурации можно задать несколько инициализаций одного и того же сервлета и задать разные параметры. Тогда в зависимости от переданных разных параметров будут заданы различные объекты одного и того же класса сервлета. Соответственно, класс один и тот же, а объекты разные и делать будут разные. Полезно, когда сервлет какой-то универсальный и делает много чего.
3. Вызывается метод init(). Этот метод в качестве параметра принимает servletConfig. Это объект, который в себе инкапсулирует конфигурацию сервлета.
4. Вызывается метод service()
5. Вызывается метод destroy(). Сам по себе веб-контейнер этот метод не вызывает после отработки предыдущего пункта. Поэтому чаще это происходит

при отключении application server-а. Можно в метод добавить другую логику завершения, например, закрытие соединения с базой данных.

Взаимодействие между компонентами веб-приложения.

Компонентов в приложении чаще всего несколько. Как организовать взаимодействие между ними? Вопрос: как сделать так, чтобы сервлеты могли обмениваться информацией? Изобретенных вариантов много. Базовых - примерно три.

1. Контекст сервлета.

Использовать контекст сервлетов - объект в результате компиляции веб-приложения целиком. Веб-приложение - способ группировки ресурсов. У него общее пространство имен, то есть элементы видят друг друга. Оно изолировано от других веб-приложений. У него общий контекст. Через него компоненты могут взаимодействовать друг с другом, так как внутри приложения у всех сервлетов этот самый контекст общий. В него можно помещать информацию, которая будет общей для всех сервлетов - через методы `getAttribute()` и `setAttribute()`. Атрибуты - это мапа: есть ключ и значение.

Дополнение. Если приложение распределенное, то на каждом экземпляре JVM контейнером создается свой контекст. Это минус. И это странно. Но вот работает почему-то так.

Контекст грузит приложение. Поэтому активное взаимодействие с ним может замедлять приложение.



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();
    }
}
```

2. HTTP-сессии.

1. HTTP - stateless протокол, поэтому сессию надо делать через костыли, куда-то сохранять информацию о пользователе (браузера). Можно в куки, но куки маленькие и много в них не сложишь. Поэтому хранение каких-то важных данных для реализации сессии осуществляется на сервере. При подключении сервер анализирует, есть ли кука (кука ассоциирована с доменом, поэтому можно посмотреть). Если куки нет - сервер ее создаст и в ответе вернет.




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).

Идентификатор сессии. Если с кукой не получилось поработать (удалена или запрещена), то идентификатор сессии добавится к URL. Также есть методы для помещения общей для сессии информации в объект `HttpSession`: `getAttribute()` и `setAttribute()`. Если на сервере несколько приложений, то каждая сессия привязана к своему приложению, хоть они и могут быть похожи. Кроме того, Java EE должна как-то обеспечивать распределённую сессию для распределённого приложения. То есть сессия не должна дробиться на мини-сессии какие-то. Запросы ассоциированы с сессией. Но ещё сессия своя для каждого веб-приложения, и это в принципе логично. То есть если на одном хосте пользователь пользуется двумя веб-приложениями, то сессии у него будут разные, и сессии эти друг друга не видят, хотя со стороны пользователя, идентификатор пользователя одинаковый.

Внимание. Контекст берется из приложения. А сессия - из запроса, Еще сессия умрет сама через какое-то время, это задается через таймауты.



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 запроса сервлетами.

Иногда хочется, чтобы сервлеты делегировали обработку запроса каким-то другим ресурсам, например, другим сервлетам, или JSP, или HTML странице. Делается это через имплементацию интерфейса `javax.servlet.RequestDispatcher`. Писать ее не надо, можно просто вызвать. Получить ее можно из запроса, можно из контекста, то есть вызывается метод от этих двух сущностей. В контексте при этом можно задать только абсолютный URL, в в сессиии - и абсолютный, и относительный.. У реквеста есть URL. От него можно куда-то пойти, и вызвать соответственно все это от какого-то варианта того, "куда ушли", А контекст

ассоциирован с приложением, URL-а у него нет, поэтому и “идти” неоткуда, ну и то есть путь обязательно относительный должен быть.

После получения ресурса, можно, наконец, делегировать. И есть два метода делегирования - `forward()` и `include()`. Допустим, сервлет что-то делал, а потом делегировал. В случае `forward()` полученные на данном этапе данные не важны. Они затрутсся. Реализация логики полностью делегирована. В случае же `include()` промежуточные результаты как бы тоже делегируются, к ним можно получить доступ, продолжить с этим работать.

Пример форварда - редирект, а пример инклюда - например, генерация шапки и тела страницы разными сервлетами, а нужна-то полностью страница, то есть оба “куска”.