

# #Servlets

## 0. Базовое использование

```
try (Socket socket = new Socket(InetAddress.getByName("google.com"), 80);
    DataOutputStream outputStream = new
DataOutputStream(socket.getOutputStream());
    DataInputStream inputStream = new
DataInputStream(socket.getInputStream())
    ) {
    outputStream.writeUTF("Hello world!");
    byte[] response = inputStream.readAllBytes();
    System.out.println(response.length);
}
```

Тут мы открываем сокет, и преобразовываем ша счет нашего dns сервера целевой хост.

С сокетами мы работаем через стримы. Output - написать внутрь сокета, input - считать с сокета

## 1. Собственный сервер

```
try (ServerSocket serverSocket = new ServerSocket(7777);
//      serverSocket.аccept() - принимает только одно соединение с клиентом,
//      далее это будет решено с помощью многопоточности
//      socket - первый клиент, что подключился
    Socket socket = serverSocket.accept();
    DataOutputStream outputStream = new
DataOutputStream(socket.getOutputStream());
    DataInputStream inputStream = new
DataInputStream(socket.getInputStream());
    Scanner scanner = new Scanner(System.in)
    ) {
    String request = inputStream.readUTF();
    while (!"stop".equals(request)) {
        System.out.println("Client request: " + request);
        String response = scanner.nextLine();
        outputStream.writeUTF(response);
        request = inputStream.readUTF();
    }
}
```

**ServerSocket** - сервер, который мы запускаем у нас на порту **7777**

**Socket** - клиент

В данном примере мы не отключаем контакт пока от клиента не поступит request stop

## 2. UDP соединение

*В UDP соединении сервера общаются с помощью перекидывания датаграм*

### Сервер:

```
public static void main(String[] args) throws IOException {  
    //          Адрес и порт исп только при отправке пакета, тут мы пакеты  
    получаем, прэтому указываем только порт  
        try (DatagramSocket datagramSocket = new DatagramSocket(7777)){  
            byte[] buffer = new byte[80];  
            DatagramPacket datagramPacket = new DatagramPacket(buffer,  
buffer.length);  
            datagramSocket.receive(datagramPacket);  
            //          datagram пакет переписывает тот массив, что мы передали в него,  
            поэтому нам нету смысла даже как-то получать его, просто используем наш buffer  
            System.out.println(new String(buffer));  
        }  
    }  
}
```

### Клиент:

```
public static void main(String[] args) throws IOException {  
    // Создаем DatagramSocket, представляющий собой сокет для отправки и  
    приема DatagramPacket  
        try (DatagramSocket datagramSocket = new DatagramSocket()) {  
            // Создаем массив байтов для хранения данных, которые будут  
            отправлены  
            byte[] bytes = "Hello from UDP client".getBytes();  
            // Создаем DatagramPacket с данными, адресом назначения  
            (localhost) и портом (7777). Адрес и порт исп только при отправке пакета  
            DatagramPacket datagramPacket = new DatagramPacket(bytes,  
bytes.length, InetAddress.getByName("localhost"), 7777);  
            // Отправляем DatagramPacket с использованием DatagramSocket  
            datagramSocket.send(datagramPacket);  
        }  
    }  
}
```

```
}  
}
```

### 3. URL подключение

```
public class UrlExample {  
    public static void main(String[] args) throws URISyntaxException,  
    IOException {  
        //      через url мы можем также подключаться к файлам  
        URL url = (new URI("https://www.google.com")).toURL();  
        URLConnection urlConnection = url.openConnection();  
  
        //      Для того, чтобы мы могли отправлять информацию. по умолчанию мы  
        //      информацию мы получаем, но конечно гугл информацию эту не примет и отправит 405  
        //      ошибку  
  
        urlConnection.setDoOutput(true);  
        String data = "param1=value1&param2=value2";  
        // Получаем OutputStream для записи данных в запрос  
        try (OutputStream outputStream = urlConnection.getOutputStream()){  
            byte[] input = data.getBytes(StandardCharsets.UTF_8);  
            outputStream.write(input, 0, input.length);  
        }  
        System.out.println(urlConnection.getContent());  
    }  
}
```

### 4. HttpClient

```
    public static void main(String[] args) throws IOException,  
    InterruptedException {  
        HttpClient httpClient = HttpClient.newBuilder()  
            .version(HttpClient.Version.HTTP_1_1)  
            .build();  
        //      Либо клиент с дефолтными настройками  
        //      HttpClient.newHttpClient();  
        HttpRequest request1 = HttpRequest.newBuilder(URI.create("https://  
        www.google.com")).GET().build();  
  
        HttpRequest request2 = HttpRequest.newBuilder(URI.create("https://
```

```

www.google.com")).POST(BodyPublishers.ofString("Hello")).build());

        HttpResponse<String> response = httpClient.send(request1,
BodyHandlers.ofString());
        System.out.println(response.body());
        System.out.println(response.headers());
    }

```

## 5. HttpServer однопоточный

```

public class HttpServer {

    private final int port;

    public HttpServer(int port) {
        this.port = port;
    }

    public void run() {
        try {
            ServerSocket serverSocket = new ServerSocket(port);
//            accept блокирует потокк
            Socket socket = serverSocket.accept();
            processSocket(socket);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    private void processSocket(Socket socket) {
//        socket мы передали чтобы в конце его закрыть (лайфхак)
        try (socket;
            DataInputStream inputStream = new
DataInputStream(socket.getInputStream());
            OutputStream outputStream = socket.getOutputStream();) {
//            1 шаг - обработка запроса
            System.out.println("Request: " + new
String(inputStream.readNBytes(400)));
//            2 шаг - обработка ответа
            byte[] body = "Hi client!".getBytes();
            byte[] headers = ""
                HTTP/1.1 200 OK

```

}

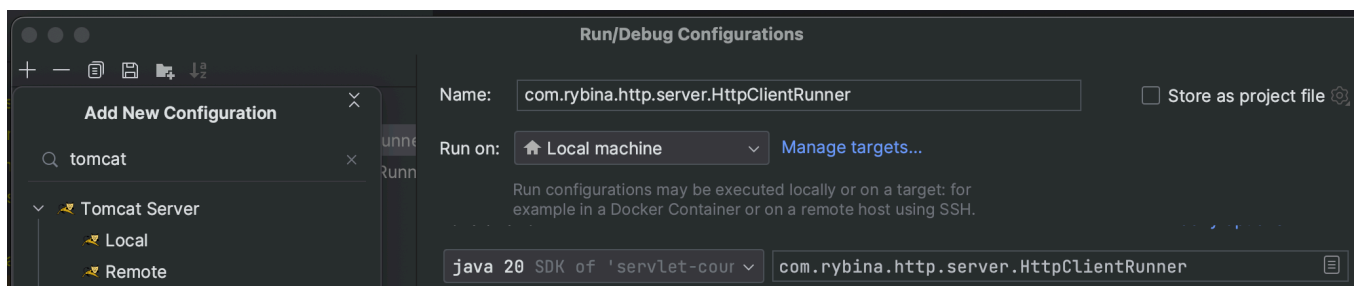
- 
- The screenshot shows the 'Project Settings' dialog in IntelliJ IDEA, specifically the 'Artifacts' tab. The 'Add' button is highlighted, and a dropdown menu is open, listing various artifact types. The 'Web Application: Archive' option is selected. The background shows the 'servlets-course:Web exploded' artifact type.
- Project Settings**
- Project
  - Modules
  - Libraries
  - Facets
  - Artifacts**
- Platform Settings**
- SDKs
  - Global Libraries
- Problems**
- Add**
- JAR
  - Run-time image (JLink)
  - Web Application: Exploded
  - Web Application: Archive**
  - Java EE Application: Exploded
  - Java EE Application: Archive
  - EJB Application: Exploded
  - EJB Application: Archive
  - JavaFx application
  - Platform specific package
  - JavaFx preloader
  - Other
- servlets-course:Web exploded
- Empty
- For 'servlets-course:Web exploded'
- INF
- course' module: 'Web' facet resource:

3. Далее у нас сохранится в папке **out** наш архив, мы де копируем от него абсолютный путь
4. Заходим в папку webapp в apache-tomcat и копируем сюда наш архив

```
cp /Users/rybina/Courses/servlets-course/out/artifacts/servlet-starter/servlets-course_Web.war .
```

5. Далее мы переходим в bin и запускаем наш сервер
6. Готово

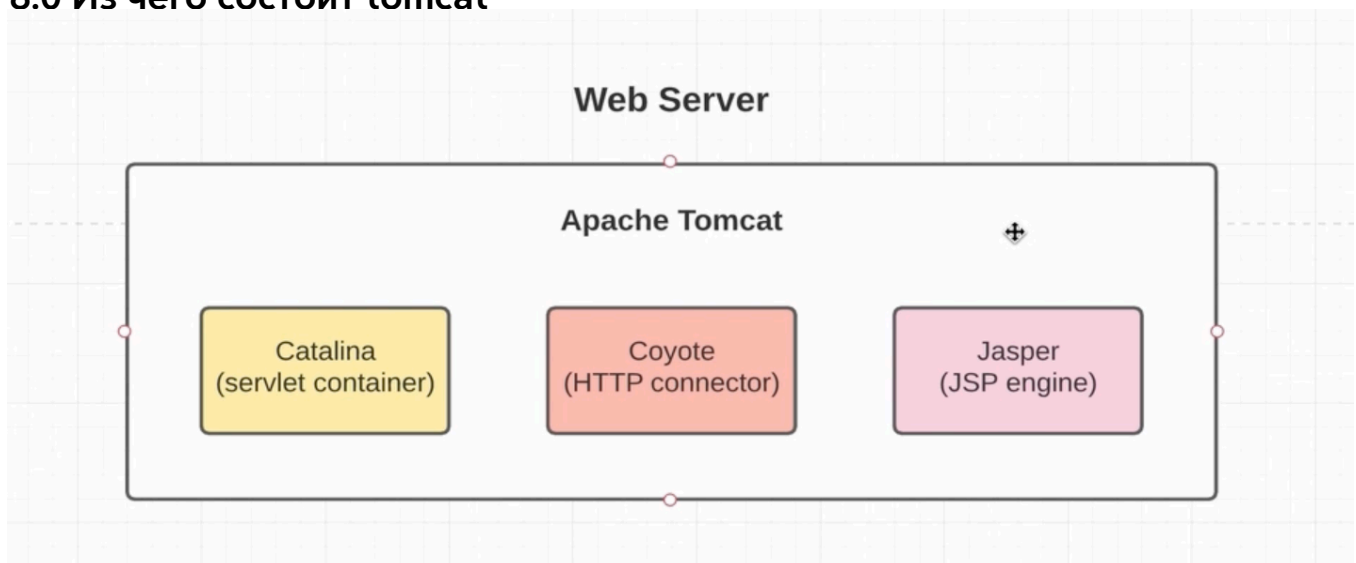
## 7. Добавляем томкат через ИДЕ



— ГОТОВО

## 8. Servlets Lifecycle

### 8.0 Из чего состоит tomcat



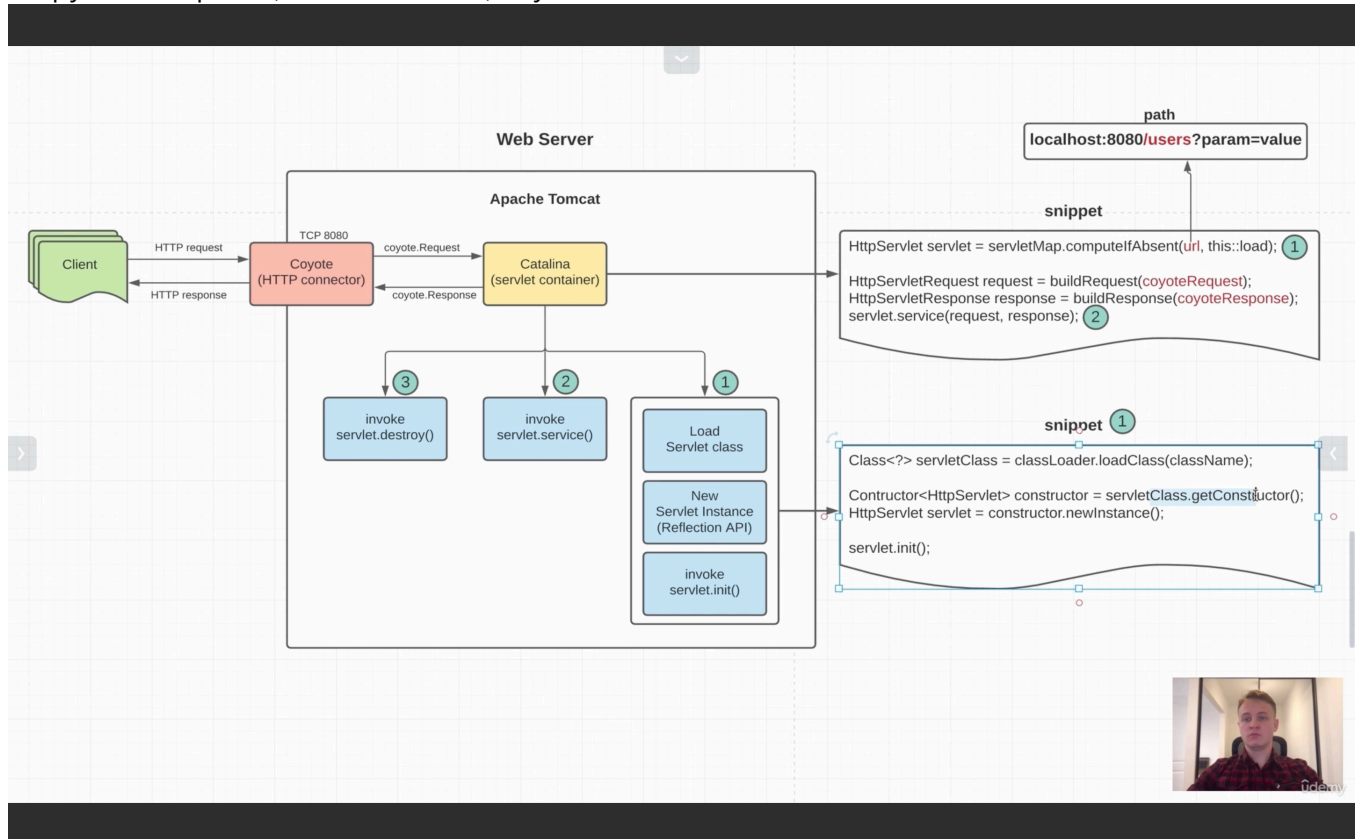
- Каталина - контейнер, по сути мапа из сервлетов

- Кайот - коннектор
- Джаспер - движок

(По сути это все код, который разложен по папкам с вышеперечисленными названиями)

## 8.1 Жизненный цикл

Загрузка → сервис (использование) → удаление



## 9. Первый servlet

Важно учесть, что у серверов только дефолтные конструктор

### 9.1 Создание самого класса

```
public class FirstServlet extends HttpServlet {
    @Override
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }
}

// так делать не грамотно
// @Override
```

```
//    public void service(ServletRequest req, ServletResponse res) throws
ServletException, IOException {
//        super.service(req, res);
//    }

// вместо этого переписывают каждый метод
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
    resp.setContentType("text/html");
    try (PrintWriter writer = resp.getWriter()) {
        writer.write("<h1> hello from first servlet</h1>");
    }
}

@Override
public void destroy() {
    super.destroy();
}
}
```

## 9.2 Маппинг этого класса старый способ

В web-inf/web.xml пишем конфигурацию подобно бинам

```
<servlet>
    <servlet-name>FirstServlet</servlet-name>
    <servlet-class>com.rybina.http.servlet.FirstServlet</servlet-class>
</servlet>
<!--    тут мы пишем по какому ключу мы получим этот сервлет, в нашем случае
localhost:8080/first-->
<servlet-mapping>
    <servlet-name>FirstServlet</servlet-name>
    <url-pattern>/first</url-pattern>
</servlet-mapping>
```

**Однако, сейчас мы уже такой способ не используем**

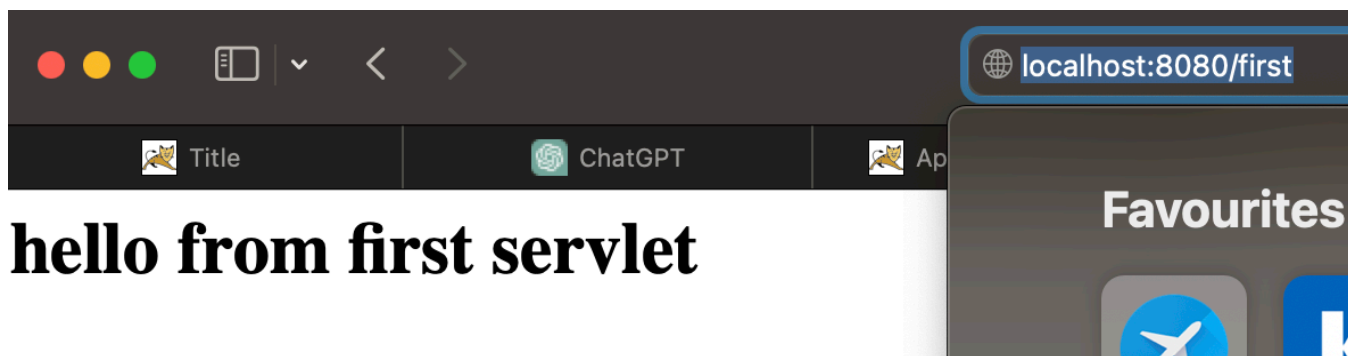
## 9.3 Маппинг этого класса актуальный способ



Теперь нам достаточно поставить **одну аннотацию над классом сервлета**

```
@WebServlet("/first")
public class FirstServlet extends HttpServlet {
```

## Итог



## 10. Servlet, который скачивает файл на диск

Для того, чтобы скачать какой-либо файл (можно бинарный, можно текстовый), мы должны взять *информационный outputStream*, который принадлежит инстанции *response* и **записываем туда необходимые данные**. Далее мы настраиваем **headers с параметром "Content-Disposition", "attachment; filename=\"название\""** и **content-type="text/plain"**. По итогу мы будем иметь файл, с названием, что мы указали, в котором будут записи, сделанные нами

```
@WebServlet("/download")
public class DownloadServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        resp.setHeader("Content-Disposition", "attachment;
filename=\"filename.txt\"");
        resp.setContentType("text/plain");
        resp.setCharacterEncoding(StandardCharsets.UTF_8.name());

        try (PrintWriter writer = resp.getWriter()) {
            writer.write("Hello from servlet");
        }
    }
}
```

```
}
```

## Закачиваем существующий файл

```
@WebServlet("/download")
public class DownloadServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        resp.setHeader("Content-Disposition", "attachment;
filename=\"filename.txt\"");
        resp.setContentType("application/json");
        resp.setCharacterEncoding(StandardCharsets.UTF_8.name());

        try (OutputStream writer = resp.getOutputStream();
            InputStream stream =
DownloadServlet.class.getClassLoader().getResourceAsStream("first.json");
            ) {
            writer.write(stream.readAllBytes());
        }
    }
}
```

Важно отметить, что мы работаем с нашим war-архивом, а то есть с папкой *out*, это значит, что путь к источнику надо писать иначе

```
DownloadServlet.class.getClassLoader().getResourceAsStream("first.json");
```

## 11. Cookies

```
@Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        Cookie[] cookies = req.getCookies();

        if (cookies == null || Arrays.stream(cookies).filter(cookie ->
UNIQUE_ID.equals(cookie.getName())).findFirst().isEmpty()) {
            Cookie cookie = new Cookie(UNIQUE_ID, "1");
            cookie.setPath("/cookies"); // если мы не установим путь, то куки
будет видно по домену сервера (localhost)
            // cookie.setMaxAge(-1); кука будет доступна пока браузер не
```

закроется

```
        cookie.setMaxAge(15 * 60); // кука будет жить 15 минут
        cookie.setHttpOnly(true); // кукинедоступен java script

        resp.addCookie(cookie);

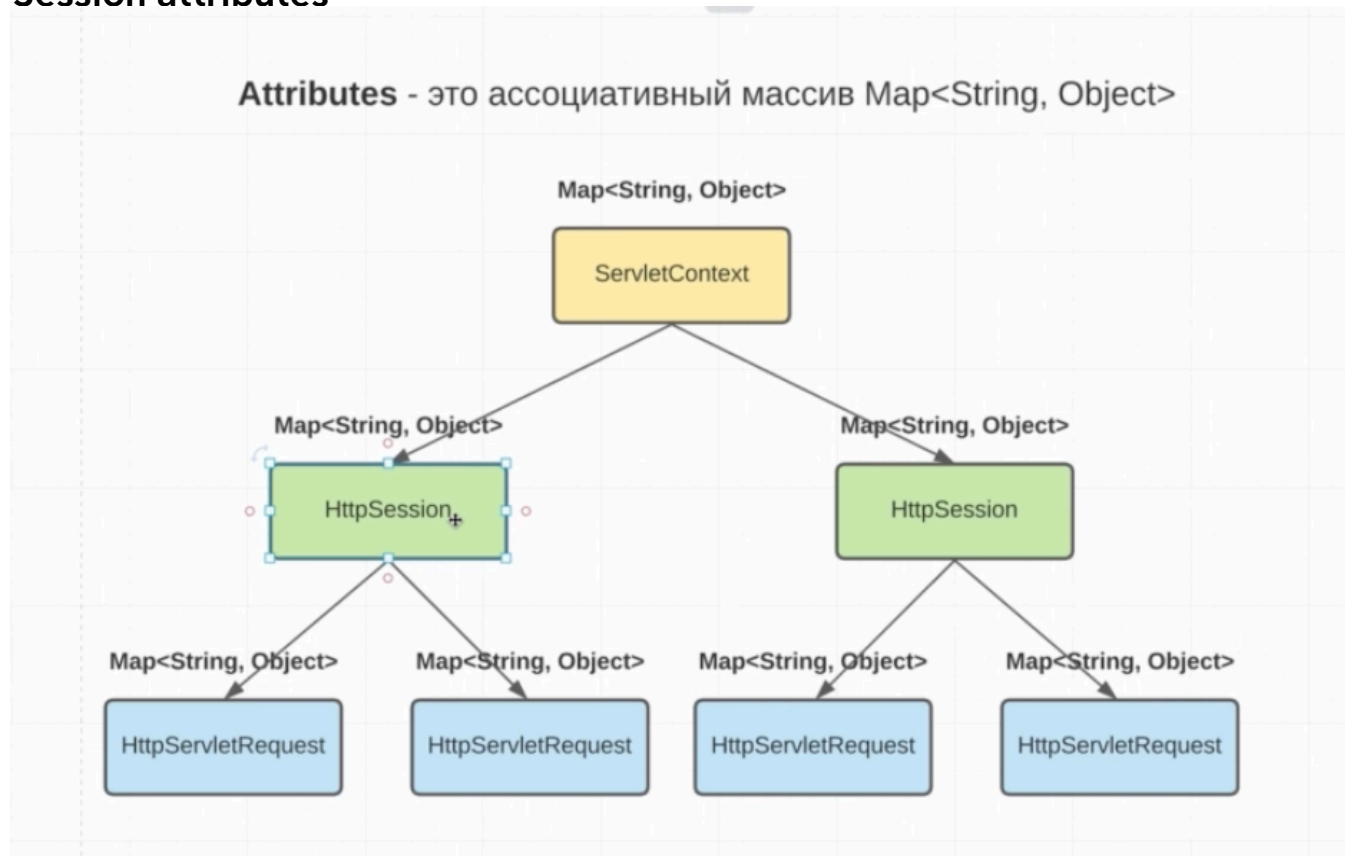
        counter.incrementAndGet();
    }

    resp.setContentType("text/html");
    try (PrintWriter writer = resp.getWriter()) {
        writer.write(counter.get());
    }
}
```

## 12. Sessions

Сессии хранятся на сервере, в нашем случае на **каталине** в качестве мапы. Мапа хранит в себе как ключ **sessionId**, а как значение класс типа **Session**

### Session attributes



Атрибуты хранятся и в глобальном объекте ServletContext

```
... void doGet(HttpServletRequest req, HttpServletResponse resp) ... {  
    req.getServletContext().getAttribute()
```

И в самом request

```
... void doGet(HttpServletRequest req, HttpServletResponse resp) ... {  
    req.getAttribute()
```

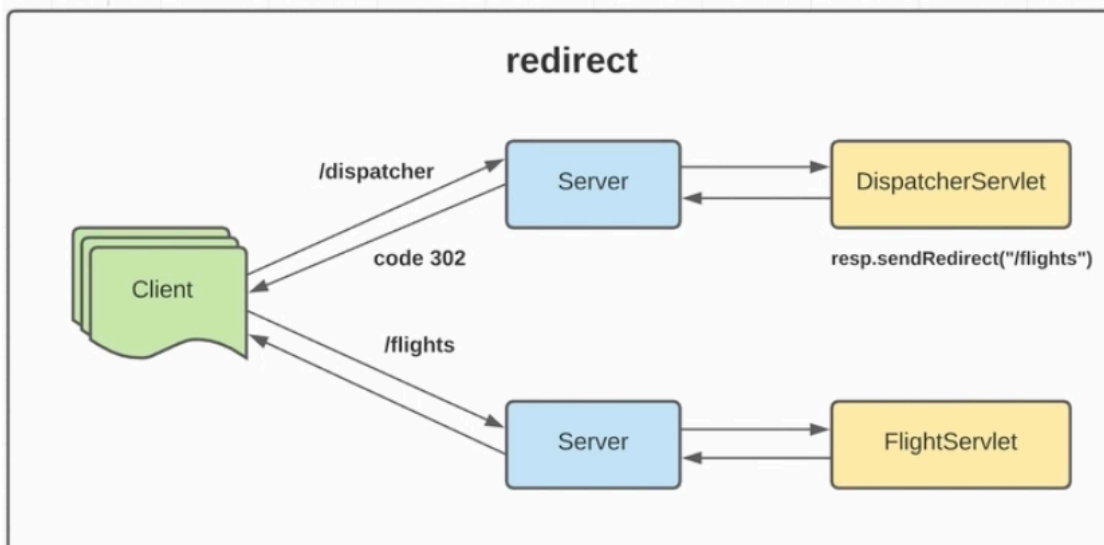
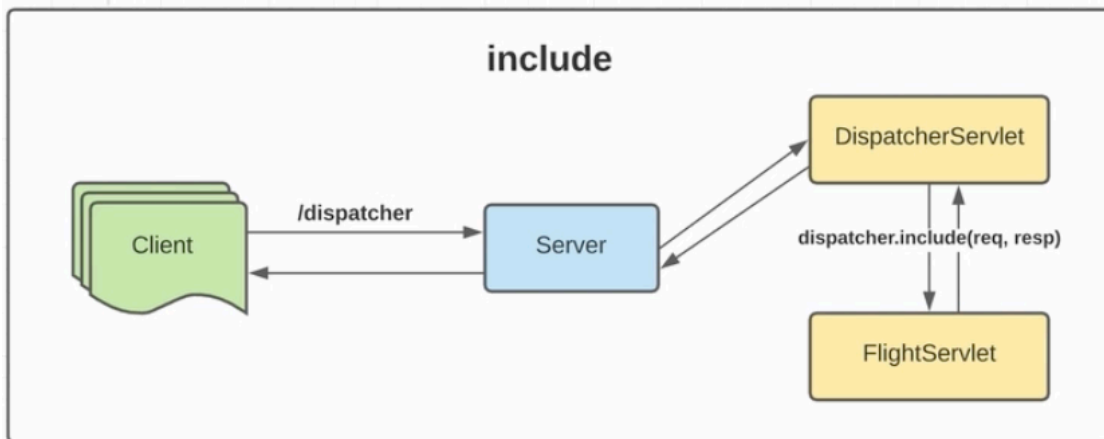
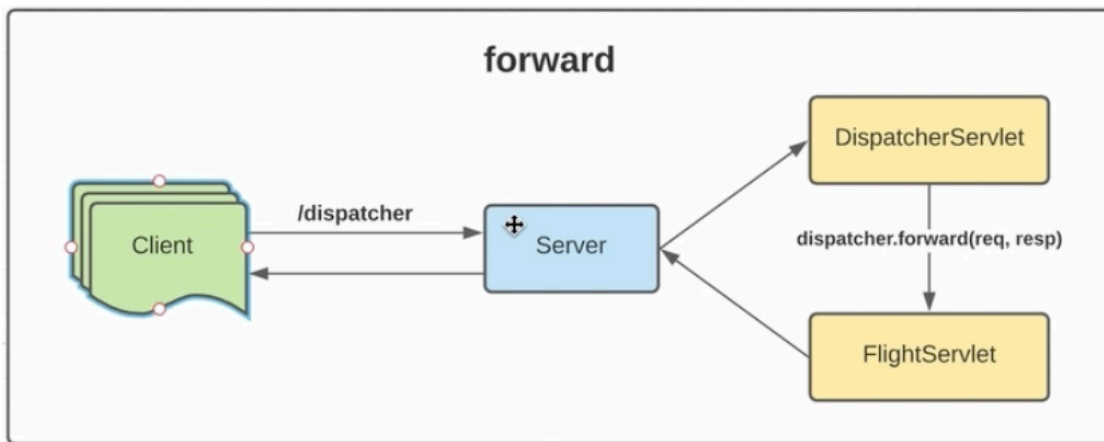
Но мы их берем из **Session**

```
@Override  
protected void doGet(HttpServletRequest req, HttpServletResponse resp)  
throws ServletException, IOException {  
    // req.getAttribute()  
    // req.getServletContext().getAttribute()  
    HttpSession session = req.getSession();  
    UserDto user = (UserDto) session.getAttribute(USER);  
    if (user == null) {  
        user = UserDto.builder().id(25L).mail("test@gmail.com").build();  
        session.setAttribute(USER, user);  
    }  
}
```

## 13. Перенаправление запросов

Реализация 3 методов

Томкат предоставляет 3 способа передачи запросов между сервлетами



```
req.getRequestDispatcher(JspHelper.getPath("tickets")).forward(req, resp);
```

Диспатчер - это глобальный объект, который мы можем взять у **request** и задать туда новый адрес запроса, далее выбираем метод и передаем наш req и resp

## 14. JSP

**Jsp** - это template engine **Джаспер**. Который преобразовывает html код с вставками

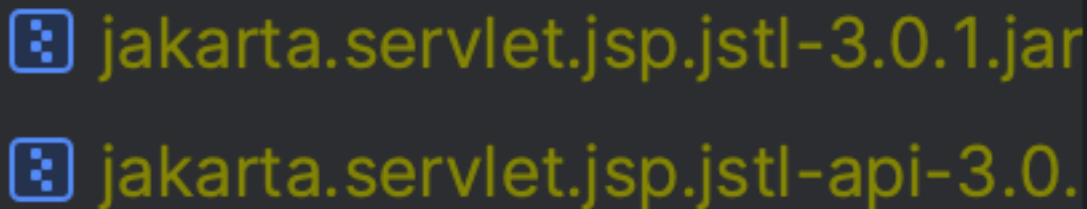
джавовского когда в сервлет и вместо нас вписывает весь уже преобразованных html код с помощью `resp.getWriter.write(...)`

Элементарный пример

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
<h1>Hello world</h1>
</body>
</html>
```

## 15. Jstl

Для jstl необходимы 2 зависимости



```
jakarta.servlet.jsp.jstl-3.0.1.jar
jakarta.servlet.jsp.jstl-api-3.0.1.jar
```

jstl - это стандартная библиотека тегов, которая предоставляет разработчикам набор готовых к использованию, переиспользуемых тегов для общих задач в JSP (JavaServer Pages). Эти теги упрощают разработку JSP-страниц, позволяя избежать использования Java-кода внутри JSP-файлов, что делает код более читаемым и легко поддерживаемым.

Простой пример

```
<ul>
<my_name:forEach var="ticket" items="${requestScope.tickets}">
    <li>
        ${ticket.seatNo}
    </li>
</my_name:forEach>
```

</ul>

*Важно перед использованием подключить нужную библиотеку*

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

Префикс мы можем выбрать любой, но вообще классический это **c**

Также мы можем подключить функции

```
<%@ taglib prefix="my_name" uri="http://java.sun.com/jsp/jstl/functions" %>
```

## 16. Пост запрос в jstl. <form></form>

При отправки формы методом POST, мы получаем данные в качестве атрибутов

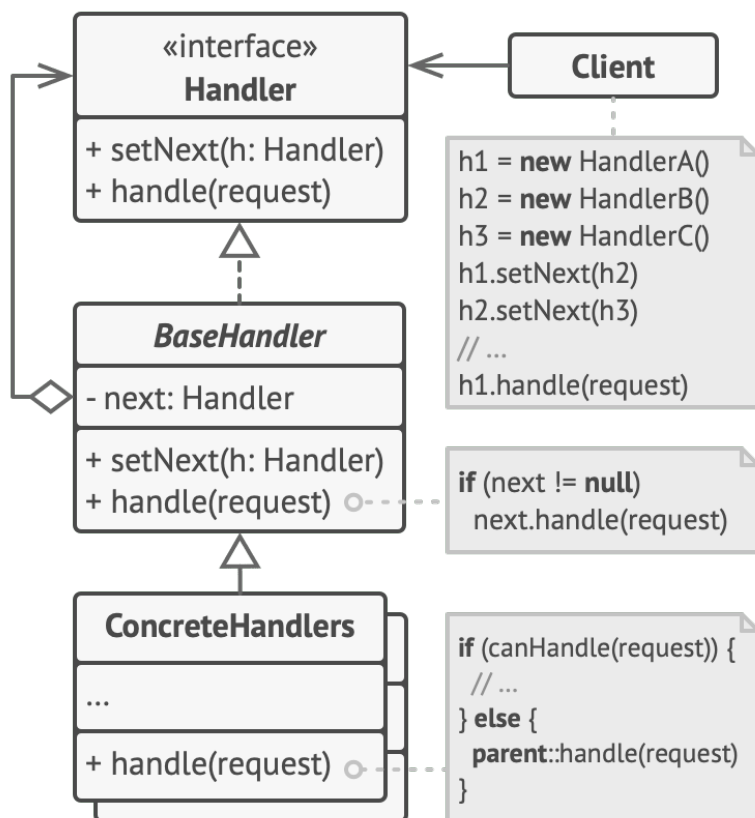
```
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
    String name = req.getParameter("name");
    String email = req.getParameter("email");
    String birthday = req.getParameter("birthday");
    String password = req.getParameter("password");
}
```

## 17. Filtres

В отличии от сервлетов, фильтров может быть несколько на один запрос, они работают по паттерну **chainOfResponsibilities**

Сначала запрос проходит все фильтры и только потом идет на сервлеты

**Вызов фильтров неупорядочен!**



```

@WebFilter(value = "/*", servletNames = {"registrationServlet"},
    initParams = @WebInitParam(name = "param", value = "paramValues"),
    dispatcherTypes = DispatcherType.REQUEST // dispatcherTypes как мы и
обсуждали ранее, это может быть redirect, forward и include
)
public class CharsetFilter implements Filter {

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        servletRequest.setCharacterEncoding(StandardCharsets.UTF_8.name());
        servletResponse.setCharacterEncoding(StandardCharsets.UTF_8.name());

        filterChain.doFilter(servletRequest, servletResponse);
    }
}
  
```

*dispatcherTypes*, как мы и обсуждали ранее, это может быть *redirect*, *forward* и *include* (см. *Аннотации над классом*)

*value* - строка запроса, аналогично сервелатам, которую обрабатывает наш фильтр



## 18. Интернационализация и локализация

Интернационализация (i18n) и локализация (l10n)

**Интернационализация** скорее архитектура для приложения

- Предусмотрение переменных вместо харкодженных значений
- Поддержка разных форматов

**Локализация** - реализация для разных рынков

- Внедрение текста в переменные
- Внедрение нужных форматов

