Chapter 3
Web Services

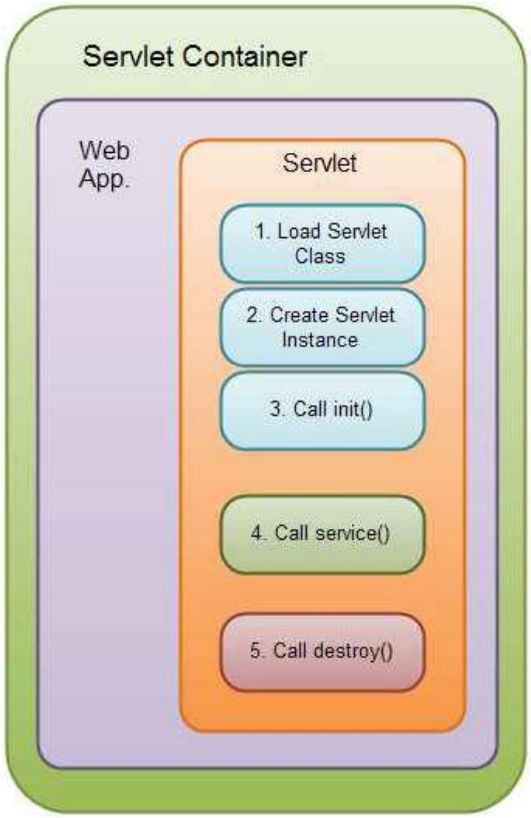
This chapter contains a basic tutorial for following web components enabling the construction of web services:

- Java Servlets (up to 4.0)
- Tomcat
- JavaServer Pages (JSP)
- JavaScript / jQuery / Ajax

3.1 Java Servlets

- Run on a Web or Application server (e.g. Tomcat, Jetty, Glassfish, ...)
  - Can interprete javax.servlet classes
  - Handles packaging of the data, listening on the web socket, mapping of URL requests to specific servlets, etc.
  - Allows to program directly on the HTTP request without worring about the lower levels
- Take the request from a Web browser (or other HTTP clients)
- Process the request (e.g. by using background applications (databases)) and send a response
  - Not only HTML documents, but different response formats possible (e.g. XML, plain text, etc...)

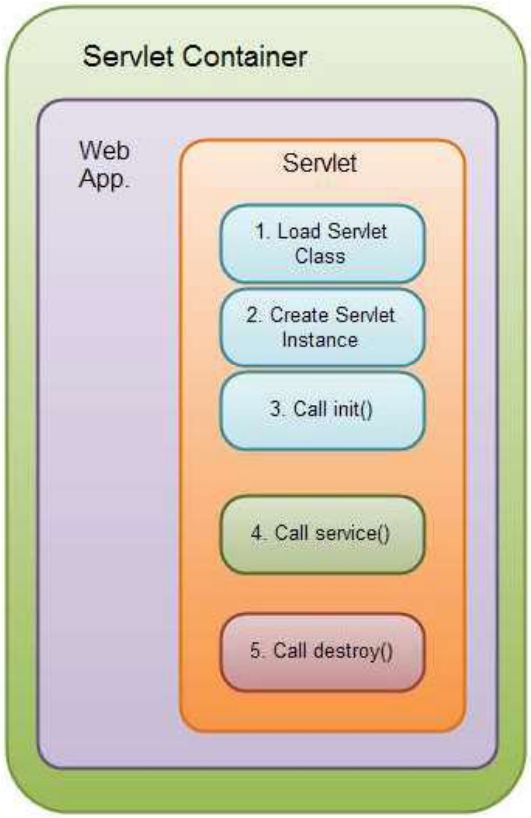
JAVA SERVLET LIFECYCLE



https://o7planning.org/en/10169/cache/images/i/12877.png

- init()
  - Is called when the servlet is loaded into the server
  - By default at the first request directed to this servlet

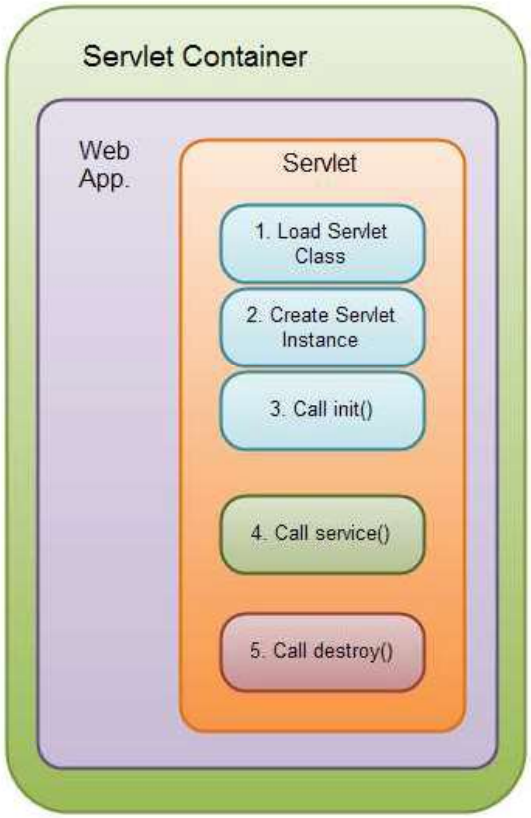
JAVA SERVLET LIFECYCLE



https://o7planning.org/en/10169/cache/images/i/12877.png

- service()
  - Is called for all incoming request to this servlet
    - Each in its own thread
  - Automatically calls specific doXXX()
    - GET, POST, PUT, DELETE

JAVA SERVLET LIFECYCLE



https://o7planning.org/en/10169/cache/images/i/12877.png

- destroy()
  - Is called just before the servlet is unloaded
    - Servlet container shutdown
    - Memory deallocation

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

@WebServlet("/HelloWorld")
public class HelloWorld extends HttpServlet {
   private static final long serialVersionUID = 1L;

   public void init() {
      // TODO Do your initialization here
   }

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                  throws ServletException, IOException {
      response.getWriter().append("Hello World!");
   }

   protected void doPost(HttpServletRequest request, HttpServletResponse response)
                                   throws ServletException, IOException {
      doGet(request, response);
   }

   public void destroy() {
      // TODO Destroy the world here!
   }
}
```

[Filename: Servlet/HelloWorld.java]

HTTPSERVLETREQUEST

- Contains all information that was sent with the client request
  - Header, Cookies, Parameters, etc.
- Outlook to useful getters:
  - .getRequestURI() - The URI the client used in this request
  - .getParameter(*paramName*) - The value of a specific query parameter
  - .getHeaderNames() - The names of all headers in this request
  - .getSession() - The session associated with this request; also automatically creates one if none exists

HTTPSERVLETRESPONSE

- Serves as the connection point to send an answer back to the client
  - Provides a PrintWriter/ServletOutputStream for writing the response message to the client
  - Set the status code of the response (e.g. HTTP 200 = ok)
    - response.setStatus(*HttpServletResponse.SC_OK*);
  - Set the content type (MIME type; Multipurpose Internet Mail Extension) of the response (e.g. text/html)
    - A comprehensive list of MIME types can be found at: https://www.iana.org/assignments/media-types/media-types.xhtml
    - response.setContentType(*"text/plain"*);

IN BETWEEN: WEBAPP STRUCTURE

- Servlets (and other web components) are bundled in a web application
- Webapps are structured in a standardized folder pattern:

- Projectname
  - WebRoot / WebContent
    - META-INF (optional)
      - MANIFEST.MF
    - resource folders: css, js, jsp, images, ... (optional)
    - WEB-INF
      - classes
      - lib
      - web.xml
  - index.html

IN BETWEEN: TOMCAT

From the Apache Tomcat website (http://tomcat.apache.org/):

```
The Apache Tomcat software is an open source implementation of the Java
Servlet, JavaServer Pages, Java Expression Language and Java WebSocket
technologies.
```

As such it is a simple **servlet container** with additional amenities. It supports only a small selection of the Java EE stack in order to be more lightweight.

If more Java EE features (e.g. Enterprise JavaBeans (EJB)) are needed they can be included through respective implementations or by using TomEE (Tomcat + Java EE).
Alternatives: Glassfish, JBoss

Tomcat consists of following parts:

- Catalina
  - Servlet container
  - Managing the lifecycle of servlets
  - Handles the mapping of paths to servlets
- Coyote
  - Connector component
  - Allows Catalina to act as a simple web server
- Jasper
  - JSP engine
  - Manages the preparation of JavaServer pages

How to load a web application into Tomcat?

- Copy the web application into "Tomcat/webapps"
  - Either raw or packaged as a Web ARchive (.war) file.
  - WAR files can be created through the jar tool ("jar -cvf webproject.war *"), Ant/Maven builds or IDEs (e.g. Eclipse)
  - Source files can be optionally included
- Start Tomcat with "Tomcat/bin/startup" & Stop with "Tomcat/bin/startup"

Note: Tomcat brings its own libraries for the supported web components (e.g. servlet-api.jar for javax.servlet.*) located in Tomcat/lib

- Do **not** include your own version in the lib folder of your web application.

SERVLET URL PATTERN

How is a Servlet called through an URL?

- the Web Server (e.g. Apache or Tomcat Coyote) has a base url:
  `http://www.semwebtech.org`
  - in our case it just forwards (most) URLs to the Web Service Container (Tomcat Catalina)
- the Web Service Container maps `relative paths` to projects (web apps)
  - in the default setting the path has the exact name of the project;
    e.g. a project with name "servletdemos" will have the relative path `/servletdemos`
    (or `http://www.semwebtech.org/servletdemos`)
- in each project (featuring multiple servlet classes) the URL path tails are mapped to a specified servlet class

```
@WebServlet("/HelloWorld")
public class HelloWorld extends HttpServlet {
    ...
}
```

The @WebServlet() annotation above the class header tells the servlet container which URL requests should be mapped to this servlet class

- **BEFORE** Servlet 3.0 this mapping had to be done in a separate web.xml file! More on that and other uses of the web.xml file later ...
- Multiple URLs can be mapped to the same servlet:
  @WebServlet(urlPatterns = {"/HelloWorld", "/hello"})
- URL patterns have to be unique! Do not map two servlets to the same URL pattern

- Different kinds of URL pattern:

Pattern	**Matches to ...**
/*	*domain*/*project*/ , *domain*/*project*/HelloWorld , ...
/HelloWorld/*	*domain*/*project*/HelloWorld , *domain*/*project*/HelloWorld/hello , ...
*.png	*domain*/*project*/Welcome.png , ...
/	Default servlet, if no other servlet matches the URL pattern

- If a requested URL maps on two different servlets, the most specific (longest) mapping is chosen
  - E.g. the requested URL http://domain/project/HelloWorld
  - Would be serviced by the servlet mapped to "/HelloWorld/*"
  - Instead of the servlet mapped to "/*"

DIFFERENT PARTS OF THE REQUEST URL

Because a servlet can be accessed through multiple request URLs it is sometimes necessary that the servlet knows what URL the client used to call it:

- Given a servlet mapped to the URL pattern = /HelloWorld/*
- The client uses the request URL:
  http://localhost:8080/project/HelloWorld/hello/howAreYou?a=fine

getRequestURL()	http://localhost:8080/project/HelloWorld/hello/howAreYou
getContextPath	/project
getServletPath	/HelloWorld
getPathInfo	/hello/howAreYou
getQueryString	a=fine
getRequestURI	/project/HelloWorld/hello/howAreYou

FORWARDING

- Client brower keeps the old URL, but content is generated by another page (servlet servicing a different URL)
- Attach information to the request to send it from one page to another by
  - request.setAttribute(*key*,*value*)
- Only forward within the same servlet container

```
@WebServlet("/forward")
public class ForwardServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                  throws ServletException, IOException {

      request.setAttribute("userName", "Bob");
      RequestDispatcher dis = request.getRequestDispatcher("/forwarded");
      dis.forward(request, response);
   }
}
```

[Filename: Servlet/ForwardServlet.java]

REDIRECTING

- Sends the request to another page (even outside of the application)
- Attaching information not possible, because a new request will be created

```
@WebServlet("/redirect")
public class RedirectServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                    throws ServletException, IOException {

      response.sendRedirect(request.getContextpath() + "/HelloWorld");
   }
}
```

[Filename: Servlet/RedirectServlet.java]

GLOBAL VARIABLES

The servlet container provides two global variables that provide access to meta information about the current servlet and the whole web application.

- ServletConfig
  - Created for each servlet
  - Provides access to the servlet name and initialisation parameters
    - Initialisation parameters for servlets can be set directly in the @WebServlet annotation

```
@WebServlet(urlPattern={"/config"}, initParams = { @WebInitParam(name="name", value="Bob") })
public class ConfigServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;
   String name = "";

   @Overwrite
   protected void init() throws ServletException {

      name = getServletConfig().getInitParameter("name");
   }
}
```

[Filename: Servlet/ConfigServlet.java]

- ServletContext
  - Created for the whole web application
  - Provides access to information that should be shared over all servlets
    * Global initialisation parameters can only be set in the web.xml
    * Servlets can attach information to the ServletContext to share them with other servlets

```
@WebServlet(urlPattern={"/context"})
public class ContextServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;
   String name ="";

   @Overwrite
   protected void init() throws ServletException {

      if(getServletContext().getAttribute("name") != null) {
         name = getServletContext.getAttribute("name");
      } else {
         getServletContext.setAttribute("name","Bob");
      }
   }
}
```

[Filename: Servlet/ContextServlet.java]

CONFIGURATIONS IN THE WEB.XML

The web.xml file contains instructions for the servlet container for handling the web components in the web application. These include:

- Registration of web app components (servlets, filters, etc)
- Definition of initialisation parameters before the creation of the ServletConfig & ServletContext variables
- Mapping of URL patterns to web components
- Declaration of welcome files

Since the introduction of Annotations in Servlet 3.0 it is possible to include servlet specific instructions directly in the java files. Web application-wide settings (ServletContext, welcome files, etc.) still have to be declared in the web.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                             http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         id="WebApp_ID" version="3.1">
  <display-name>WebProject</display-name>

  <!-- Registration & URL mapping in the web.xml is no longer necessary! -->
  <!-- It is done automatically in the java file through the @WebServlet annotation
  <servlet>
    <servlet-name>HelloWorld</servlet-name>
    <servlet-class>org.semwebtech.servlet.HelloWorld</servlet-class>
    <init-param>
      <param-name>name</param-name>
      <param-value>Bob</param-value>
    </init-param>
  </servlet>

  <servlet-mapping>
    <servlet-name>HelloWorld</servlet-name>
    <url-pattern>/HelloWorld</url-pattern>
  </servlet-mapping>
  -->

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

[Filename: Servlet/web.xml]

SERVLET SESSIONS

Originally the HTTP protocol was meant to be state-less

- Demand of webservices to have a state (/information) specific to each client

Solution?

- Cookies
- Hidden form fields
- URL rewriting

Which leads to **session management**.

The servlet container automatically handles the organizational aspect of sessions for us!

- Creates a **HttpSession** object for each client that stores information over multiple requests
  - Access through: `HttpSession session = request.getSession();`
    Automatically creates a session if none existed for this client.
  - request.getSession(false) if only already existing sessions should be used.
  - .setAttribute(*key*,*value*) & .getAttribute(*key*) - to attach and retrieve information specific to this session
  - .setMaxInactiveIntercal(*int*) - sets the maximum time between client requests before the session automatically terminates (default 30 min)
  - .invalidate() - manually terminates the session

```
@WebServlet("/session")
public class SessionServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                  throws ServletException, IOException {

      PrintWriter out = response.getWriter();

      HttpSession session = request.getSession();
      Enumeration<String> attr = session.getAttributeNames();

      while( attr.hasMoreElements() ){
         String key = attr.nextElement();
         out.println(session.getAttribute(key).toString());
      }
   }
}
```

[Filename: Servlet/SessionServlet.java]

- Tomcat supports different **session tracking modes**
  - Cookies (default)
    - The server sends the client a *JSESSIONID* cookie, **if the client allows/supports cookies!**
    - The client attaches the *JSESSIONID* cookie to every following HTTP header for identification
  - URL (if no cookies allowed)
    - Rewrites the URL to include the client specific ID
    - Client has to use these special URL requests to maintain the session
- Can be changed at web application start through the web.xml
  - Most of the time URL rewriting is not wanted

```
<session-config>
    <tracking-mode>COOKIE</tracking-mode>
</session-config>
```

MULTI-USER SERVLETS

When programming servlets one should expect that multiple clients use the servlet simultaneously. So it is important to keep in mind which resources are shared on which level:

- **ServletContext** is only created once
  - Shared among all requests in all sessions
- **Web components** (servlet, filter, listener) are instantiated once at load of the web application
  - init() is called when the first request hits the component OR if the <load-on-startup> setting is > 0
  - Shared among all requests in all sessions
- **HttpSessions** are stored on the server and stay active till timeout or termination
  - Shared among all requests in the same sessions
- **HttpServletRequest** / **HttpServletResponse** created for each HTTP request from the client
  - Each as a new thread for the servlet - not shared

Any *attribute* attached to **ServletContext**, **HttpSessions** & **HttpServletRequest** lives as long as the respective object

- This is often referred to as the "scope"

Keep in mind that Java is inherently multithreaded

- Different threads can make use of the same instance

Thread-safety has to be assured through the code!

- Request or session scoped data should **never** be assigned as an instance variable of the servlet

SERVLET FILTER

Servlet filters are special kind of servlet that can/should be mapped on the same URL patterns as the intended servlets. They get called on every request **before** it reaches the servlet and **after** the processing of the servlet just before the response is send to the client. As such they are used for:

- Blocking requests / redirecting them
- Modifying request/response header & data

They are a great way to reduce redundant code which otherwise must have been included in several servlets, e.g. to check if the client sending the request is logged-in or not.

```
@WebFilter("/HelloWorld/*")
public class LoginFilter implements Filter {

    @Override
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
                         throws IOException, ServletException {

        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) res;
        HttpSession session = request.getSession(false);

        if (session == null || session.getAttribute("user") == null) {
            // No logged-in user found, so redirect to login page.
            response.sendRedirect(request.getContextPath() + "/login");
        } else {
            // Logged-in user found, so just continue request.
            chain.doFilter(req, res);
        }
    }

}
```

[Filename: Servlet/LoginFilter.java]

- @WebFilter annotation for automatic registration and URL mapping
- (Optional) init() & destroy() as with servlets
- doFilter() is called for every request
  - `chain.doFilter(req, res)` sends the request to the next filter or to the target servlet

SERVLET LISTENER

Servlet listener are the third and final web component that can be registered to the servlet container. There are three kinds of listener that react to different kinds of events:

- ServletContextListener - Webapp lifecycle
  - Before any filter / servlet / etc. are initialized
  - After all filter / servlet / etc. are destroyed
- HttpSessionListener - Session lifecycle
  - After a session is created - request.getSession()
  - Before a session is destroyed - session.invalidate()
- ServletRequestListener - Request lifecycle
  - Before a request enters the first servlet/filter
  - After the request exits the last servlet

```
@WebListener
public class OpenSessionListener implements HttpSessionListener {

   private static final AtomicInteger sessionCount = new AtomicInteger(0);

   @Override
   public void sessionCreated(HttpSessionEvent event) {
      sessionCount.incrementAndGet();
   }

   @Override
   public void sessionDestroyed(HttpSessionEvent event) {
      sessionCount.decrementAndGet();
   }

   public static int getTotalSessionCount() {
      return sessionCount.get();
   }

}
```

[Filename: Servlet/OpenSessionListener.java]

From the Servlet 3.0 specification:

The container is not required to synchronize the resulting notifications to attribute listener classes. Listener classes that maintain state are responsible for the integrity of the data and should handle this case explicitly.

- A primitiv Integer would not be thread safe. Use AtomicInteger or synchronize the access.

MORE COMPLEX EXAMPLES: DOWNLOAD FILE

Allow the client to download files from the server

- The attribute **Content-Disposition** in the header tells the browser how to handle the accessed file
  - Inline : default; for webpage content
  - Attachment : for downloadable files
    - Can be given a file name that will be displayed in the browser

```
@WebServlet("/download")
public class DownloadServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
                  throws ServletException, IOException {

      response.setContentType("text/plain");
      response.setHeader("Content-disposition", "attachment; filename=test.txt");

      OutputStream out = response.getOutputStream();
      InputStream in = getServletContext().getResourceAsStream("/WEB-INF/test.txt");
      int bytesRead;
      byte[] bytes = new byte[1024];
      while((bytesRead = in.read(bytes)) > 0){
         out.write(bytes, 0, bytesRead);
      }
      in.close();
      out.flush();
      out.close();
   }
}
```

[Filename: Servlet/DownloadServlet.java]

MORE COMPLEX EXAMPLES: UPLOAD FILE

Allow the client to upload files to the server

- First create a HTML page that allows the user to input a file
  - The **enctype** in the HTML form must be set to "multipart/formdata"
  - Input type is "file" ; multiple can be set to true to allow more than one file

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "www.w3.org/TR/html4/sgml/dtd.html">
<html>
   <head>
      <title>Upload file</title>
   </head>

   <body>
      <h1>Upload a file!</h1>
      <form action="upload" method="post" enctype="multipart/form-data">
         <input type="file" multiple="true" name="file" />
         <input type="submit" />
      </form>
   </body>
</html>
```

[Filename: Servlet/UploadFile.html]

On the servlet side it depends on the used servlet version

- **BEFORE** Servlet 3.0 : Use an auxiliary package like org.apache.commons.fileupload
- **WITH** Servlet 3.0 or higher: Use the native API
  - Annotate with @MultipartConfig to recognize and support `multipart/form-data` and get access to `getParts()`

Decide where to store the uploaded file

- Anywhere except in the server's deploy folder
  - The folder of your web application is not a safe place to store new files, because it is not guaranteed how the servlet container handles them.
  - This means especially: do not use `getRealPath()` for the definition of the files location

```
@WebServlet("/upload")
@MultipartConfig
public class UploadServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;

   protected void doPost(HttpServletRequest request, HttpServletResponse response)
                        throws ServletException, IOException {

      List<Part> fileParts = request.getParts().stream()
         .filter(part -> "file".equals(part.getName())).collect(Collectors.toList());
      File upload = new File(File.seperator + "path");

      for(Part filePart : fileParts){
         String fileName = Paths.get(filePart.getSubmittedFileName()).getFileName().toString();
         File file = new File(upload,fileName);
         try(InputStream fileContent = filePart.getInputStream()){
            Files.copy(fileContent, file.toPath());
         }

      }

   }
}
```

[Filename: Servlet/UploadServlet.java]

MORE COMPLEX EXAMPLES: JDBC & LOG4J INITIALIZER

Use the ServletContextListener to initialize global settings and background programs

- Create a JDBC connection and make it available to all servlets
- Initialize log4j so that every servlet logs with the correct settings
- etc...

FEATURES OF SERVLET 4.0

Servlet 4.0 is a core update of Java EE 8 released in September 2017. It includes:

- Usage of HTTP/2 (with that the "server push" feature)
- HttpServletMapping interface
- HTTP trailer
- GenericFilter & HttpFilter classes

HTTP/2

- Built upon the SPDY protocol
- Is not a complete rewrite of HTTP/1.x
  - HTTP methods, status codes are the same
  - Backwards compatible with HTTP/1.1
- Focus is performance
  - percieved browser performance
  - network & server resource usage
- Header compression through a server and client side HPACK
  - (Stream compression (like GZIP) inside encryption would be abusable through the CRIME exploit)

Problems with HTTP/1.x

- Each client has a limited number of TCP connections to a server
- A new request on one of the connections has to wait for the completion of the previous request. This leads to the "head-of-line blocking" problem.
- Thus it is more efficient with HTTP/1.1 to send one big request/response than multiple smaller ones. Even if some of the data is not needed.

Goal of HTTP/2: Use of a single connection from browser to the website

- Fully multiplexed to remove the "head-of-line blocking" problem
  - Allowing multiple requests and response messages to be on the flight at the same time
  - Removes the need of previous "best practices"
    - Sprite files (multiple images in one file)
    - Concatenating CSS & Javascript files
- Serve the user only what he needs

HTTP/2 Server push

- HTTP/1.x : One request -> one response
  - If the response includes references to other resources on the server (images, JavaScript files, etc.), then a new request is sent to retrieve these resources (so far automatically handled by the server)
- HTTP/2 Server push:
  - Server can populate the browser cache before sending the actual response
    - This is done under the assumption that the server should know which additional data the browser will need (e.g. CSS, JavaScript files)
  - Clients can decline the push promise
    - e.g. when the pushed files are already in the cache

- Server push done through the `PushBuilder` obtained from the `HttpServletRequest`
  - Returns null if the client does not support it
- TLS must be enabled for the connection
  - Not by design, but no browser so far supports HTTP/2 without encryption

```
@WebServlet("/http2")
public class PushServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
                    throws ServletException, IOException {

      PushBuilder pushBuilder = request.newPushBuilder();
      if(pushBuilder != null){
         pushBuilder.path("images/test.png");
         pushBuilder.addHeader("content-type",  "image/png");
         pushBuilder.push();
      }
      //Here send some HTML page which includes <img src='images/test.png'>
   }
}
```

[Filename: Servlet/PushServlet.java]

HTTPSERVLETMAPPING

- Interface for runtime discovery of URL mappings
  - .getMappingMatch() - type of the match (e.g. DEFAULT, EXACT, ...)
  - .getPattern() - URL pattern that matched this request
  - .getMatchValue() - the String that was matched
  - .getServletName() - fully qualified name of the servlet class that was activated

```
@WebServlet(name = "MappingServlet", urlPatterns = {"/mapping"})
public class MappingServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
                    throws ServletException, IOException {

      response.setContentType("text/plain");

      HttpServletMapping mapping = request.getHttpServletMapping();
      String mapName = mapping.getMappingMatch().name();
      String value = mapping.getMatchValue();
      String pattern = mapping.getPattern();
      String servletName = mapping.getServletName();

      PrintWriter out = response.getWriter();
      out.println("Mapping type: " + mapName);
      out.println("Match value: " + value);
      out.println("Pattern: " + pattern);
      out.println("Servlet name: " + servletName);

   }
}
```

[Filename: Servlet/MappingServlet.java]

3.2 JavaServer Pages (JSP)

Create dynamically generated web pages based on HTML, XML, etc ...

- Java Servlets - Generate dynamic web pages (HTML, XML) within Java
- JavaServer Pages - Use Java code within HTML to generate dynamic content
  - Translated into Java Servlets at runtime by the container
    (found in Tomcat/work)
    - First time the JSP is accessed
      (Some containers can be configured to recompile the JSP after a specified time)
    - Produces HTML page according to the instructions in the JSP file
  - Not a replacement of Java Servlets, but to compliment it
    - More convenient for the presentation
      (separation of Model-View-Controller (MVC); see "Model 2")
  - (Similar to Active Server Pages (ASP) for Microsoft Windows Server)

Java code in the JSP file is indicated by different **delimiters** depending on the purpose of the Java code:

- JSP Scriptlet: `<% ... %>`
  - Contains a fragment of Java code (loops, local variables, operations)
- JSP Expression: `<%= ... %>`
  - In the resulting HTML page these will be replaced with the result of the contained evaluation
- JSP Directive: `<%@ ... %>`
  - Contains directives on the global document or the JSP engine (e.g. import java packages)
- JSP Declaration: `<%! ... %>`
  - Contains the declaration of global variables & methods
- JSP Action: `<jsp: ... />`
  - Calls predefined functions already included in JSP (e.g. forwarding to another page)
- JSP Comment: `<%- ... -%>`
  - Contains comments in the JSP file; these will not be viewable in the resulting HTML page!

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
      pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
      "http://www.w3.org/TR/html4/loose.dtd">
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
      <title>Hello World!</title>
   </head>
   <body>
      <h1>Hello World in HTML!</h1>
      <% out.println("Hello " + request.getParameter("name") + " in JSP!"); %>
      <p>Counting to three:</p>
      <% for (int i=1; i<4; i++) { %>
         <p>This number is <%= i %>.</p>
      <% } %>
   </body>
</html>
```

[Filename: Servlet/HelloWorld.jsp]

- Java code does not have to be selfcontained in a single scriptlet; can be interrupted by HTML
- Access to implicit objects that can be used without declaring them:
  - request/response (HttpServletRequest/-Response), out (JspWriter), session (HttpSession), application (ServletContext), config (ServletConfig), page (Java "this")

```
@WebServlet("/HelloWorldJSP/*")
public class CallJSP extends HttpServlet {
   private static final long serialVersionUID = 1L;

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
                    throws ServletException, IOException {

      request.getRequestDispatcher("/HelloWorld.jsp").forward(request, response);
   }
}
```

[Filename: Servlet/CallJSP.java]

- Call JSP file from a Java Servlet
- GET or POST are not differentiated for JSP; if necessary the request method can be inferred through `request.getMethod()`

UNIFIED EXPRESSION LANGUAGE

The unified expression language (unified EL) is a scripting language that allows among other things easier access to Java components (Java Beans) without scriptlets. Invoked by **${*expr*}**.

Comes with a lot of implicit objects that are already loaded into the scope, e.g.:

- pageContext: (gives access to) servletContext, session, request, response
- param, header, cookie

The following table shows example replacements of common uses of scriptlets:

${param.name}	request.getParameter("name")
${pageContext.request.contextPath}	request.getContextPath()
${header["Host"]}	request.getHeader("Host")

Arithmetic (+ - * / % div mod), relational (== $<$ $>$ ...) and logical (&& | | not) operators are also available.
To check if the accessed value is null or an empty string, etc. use `empty` $< object >$.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                  "http://www.w3.org/TR/html4/loose.dtd">
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
      <title>Hello User!</title>
   </head>

   <body>
      <h1>Welcome ${not empty param.name ? param["name"] : "Guest"}</h1>
      <a href="${pageContext.request.contextPath}/download"}>Download link</a>
   </body>
</html>
```

[Filename: Servlet/DownloadJSP.jsp]

JSP Standard Tag Library (JSTL)

JSTL is a library to simplify the most common scriptlets. It consists of six components:

- core : conditionals, loops, variables, URL specifics
- xml : XML and XML transformation
- sql : database connections, queries, updating
- fmt : formating and internationalization
- functions : standard functions for string manipulation
- tlv : enforce restrictions regarding the use of scripting elements and permitted tag libraries

The tag libs must be included in the beginning of the JSP file and assigned a prefix to be available in the document.

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

- Tomcat does not include a JSTL implementation (`jstl.jar` Version 1.2 for Servlet 2.5 and higher)
  - Can be downloaded from: https://tomcat.apache.org/taglibs/standard/
  - Include it to the WEB-INF/lib (and add to the build path depending on IDE)

Some examplary tags from the `core` tag lib:

- `<c:out >` - Like <%= ... >, but for expressions
- `<c:set >` - Set the evaluation of an expression to variable in a scope
- `<c:forEach >` - Basic iteration
- `<c:if >` - Conditional

Some examplary tags from the `function` tag lib:

- `<fn:contains(String, String) >` - Tests if an input string contains the specified substring
- `<fn:length(Object) >` - Returns the number of items in a collection, or the number of characters in a string
- `<fn:substring(String, int, int) >` - Returns a subset of a string
- `<fn:trim(String) >` - Removes white spaces from both ends of a string

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<c:set var="max" value="6"></c:set>

<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
      <title>Hello World!</title>
   </head>
   <body>
      <h1>Hello World in HTML!</h1>
      <p>Hello <c:out value="${param.name}" default="Guest"/> in JSTL!<p>
      <p>Counting to three:</p>
      <c:forEach begin="1" end="${max}" var="i" step="2">
         <p>This number is ${i}</p>
      </c:forEach>
   </body>
</html>
```

[Filename: Servlet/HelloWorldJSTL.jsp]

MODEL-VIEW-CONTROLLER (MVC)

A architectural design which separates

- the internal data and business logic (**model**)
- from the UI components that are presented to the user (**view**)
- through which they can make changes to the data (**controller**).

This pattern is often used for GUIs and web applications. A more specific approach in the context of Servlets, JSP, ... is referred to as "Model 2":

- Model: JavaBeans, Databases, etc.
- View: JSP pages
- Controller: Servlet, Filter

Note: It is considered best practice to keep these web components to their respective scope as indicated by the Model 2 design. This means

- a Servlet recieves a request and interacts with server-side components to fulfill it
- then the request is forwarded to a JSP page, which **only** presents the response
- the response page gives the user the option to send new requests

Problem: Through the use of scriptlets it is possible that a JSP page mimicks the role of a Controller. Thus

- it is frowned upon to use any kind of scriptlets in JSP so far that they get actively disabled.
- all programming work can be done via tag libraries, which also have the benefit of included security features.

3.3 JavaScript (Overview)

A *dynamic* programming language (that has nothing to do with Java!) for frontend and backend development.

- Operations otherwise done at compile-time can be done at run-time
  - Change type of a variable
  - Add new properties or methods to an object

Can provide dynamic (client-side) interactivity on websites when applied to an HTML document.

- Imported from seperate files or inline
- Enclosed by <script> </script> element
- Implementation included in most browsers (if not disabled)

- Variables are not type-bound

```
let myVar = [1,'Bob',2,'Alice']
```

  * `const` for constants
  * `var` is the old version of `let`
  * Variable types: Number, String, Boolean, Array, Object

  – Function return type is not specified

```
function test(v1, v2) {    return v1 + v2;
}
```

- Basic functions:

  `document.querySelector(`*`value`*`)`

  * Returns the DOM elements that match the *value*

  `element``.getAttribute(`*`attr`*`)`

  * Returns the value of the attribute *attr* of the DOM *element*

  `alert(`*`value`*`)`

  * Creates a pop-up window in the browser displaying the *value*

  `prompt(`*`text`*`)`

  * Displays *text* and returns the user input

  `localStorage.setItem(`*`key`*`, `*`value`*`)`

  * Calls the Web Storage API
  * Store data in the users browser (as a cookie)

  `localStorage.getitem(`*`key`*`)`

  * Retrieve previously set data

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="ISO-8859-1">
      <title>Hello World in JS</title>
   </head>
   <body>
      <p>Before Hello World!</p>
      <script>
         alert('Hello World in JavaScript!');
      </script>
      <p>After Hello World</p>
   </body>
</html>
```

[Filename: Servlet/HelloWorldJS.html]

We want to run code in response to user behaviour or browser events. –> **Event**-handler

- Different kinds of events: resource, focus, mouse, ...
- Complete set of possible events found at https://developer.mozilla.org/en-US/docs/Web/Events

Different ways to add event handlers:

- Programmatically search for the wanted DOM element and add a EventListener
  - `document.querySelector('button').onclick = function() {`
    `alert('You clicked me!  Ouch!');`
    `};`
  - `document.querySelector('button').addEventListener('click', function() )`
- Inline JavaScript handlers directly in the element
  - `<button onclick="function()">Test</button>`
  - Considered bad practice to mix JavaScript directly into the HTML

Attention regarding HTML content / JavaScript loading!

- Browser loads the HTML page from top to bottom
- If JavaScript should affect some HTML elements, it has to appear after it in the document

=> Old way: Write JS directly before the </body> tag

  - But that slows down the page if there is a lot of JS

Attention regarding HTML content / JavaScript loading!

=> For **internal** JS: Force the JS to be run after the HTML body is completly loaded

- The browser throws a *DOMContentLoaded* event after it finished loading the HTML content

```
document.addEventListener("DOMContentLoaded", function() {
...  PUT YOUR JS CODE HERE ...
} );
```

=> For **external** JS: Tell the JS to be loaded simultaneously to the HTML with **async**

```
<script async src="...  .js"></script>
```

- If multiple JS files are loaded & one depends on the other, tell them to wait with **defer** (includes async)

```
<script defer src="js/script1.js"></script>
<script defer src="js/script2.js"></script>
```

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="ISO-8859-1">
      <title>JS Button Test</title>
      <script>
          document.addEventListener("DOMContentLoaded", function() {
             document.querySelector('button').onclick = function() {
                alert('Stop poking me!');
                let headline = document.getElementById('head1');
                headline.innerHTML = "You pressed it...";
             };
          });
      </script>
   </head>
   <body>
      <h1 id="head1">Do not press the button!</h1>
      <button>Button</button>
   </body>
</html>
```

[Filename: Servlet/JSButton.html]

AJAX

Asynchronous JavaScript and XML (Ajax) for partially updating the web page with new data from the server without doing a page refresh.

- Provoked by GUI elements or events
- Uses the *XMLHttpRequest* for asynchronous communication which is built-in within JavaScript

It is not one technology, but multiple used together:

- HTML
- CSS
- DOM
- JSON / XML
- JavaScript

Often used in interactive web pages, e.g. Google maps.

An "Ajax-Engine" handles JavaScript calls from the interacting user

- If information from the webserver is necassery, only data that is needed can be requested
- Update the DOM tree based on the response
- No browser plugin necessary (except enabled JavaScript)

!The implementation specifics are different for each browser

-> Better to use JavaScript libraries that offer Ajax support and handle the browser differences internally

- jQuery, Prototype, Mootools, ...

3.4 jQuery

A JavaScript library mainly for manipulating the DOM

- High-level functionality for searching and interacting
- Easier access to Ajax functionality
  - Still uses the XMLHttpRequest under the hood

Available by adding a jQuery implementation in the head or body of the HTML document

- Download a specific jQuery implementation from https://jquery.com/download/
- Automatically include the latest version

```
<script src="https://code.jquery.com/jquery-latest.js"></script>
```

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="ISO-8859-1">
      <title>jQuery Hello World</title>
      <script src="https://code.jquery.com/jquery-latest.js"></script>
      <script>
          $(document).ready(function(){
             alert("The headline will change after this");
             $("h1").html("Hello jQuery!");
          });
      </script>
   </head>
   <body>
      <h1>"Goodbye" JavaScript...</h1>
   </body>
</html>
```

[Filename: Servlet/HelloWorldjQuery.html]

- jQuery functionality accessed via `$()` or `jQuery()`
- `$(document).ready(function() )` instead of `document.addEventListener("DOMContentLoaded'', function() )`

DOM navigation: `$( Selector )`

- The *Selector* is an expression that matches DOM elements and consists of CSS and custom additions.
  A complete overview of *Selector* possibilites can be found at
  http://api.jquery.com/category/selectors/
- Some basic *Selector* examples:
  - `$("button[type="submit"]")` - Selects all <button> elements with attribute "type", which has a value "submit"
  - `$("#line1")` - Selects the element with id value "line1"
  - `$(".red")` - Select all elements belonging to class "red"
- The property `.length` can be used to find out how many elements were matched (even 0)

Event handler in jQuery work just like in JavaScript

- `$(`*Selector*`).click(function(){ ... } );`
- `$(document).on("click", `*Selector*`, function() { ... } );`

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="ISO-8859-1">
      <title>jQuery Link Test</title>
      <script src="https://code.jquery.com/jquery-latest.js"></script>
      <script>
          $(document).ready(function(){
             $( "a" ).click(function(event) {
                alert("This link is under construction!");
                event.preventDefault();
             });
          });
      </script>
   </head>
   <body>
      <h1>Hyperlink construction site!</h1>
      <a href="https://www.dbis.informatik.uni-goettingen.de/Teaching">Work in progress</a>
   </body>
</html>
```

[Filename: Servlet/jQueryLink.html]

Ajax functionality is available in many ways through the jQuery library. The standard syntax is:

```
$.ajax( {
    url:  request url,
    data:  data to be sent to the server as key:value pair,
    success:  success function,
    dataType:  expected return data type,
    method:  HTTP method
});
```

With many more possible parameters. An actual request would then be:

```
$.ajax( {
    url:  'HelloWorld',
    data:  { userName :  $('#userName').val()},
    success:  function(responseText){$('#ajaxResponse').text(responseText;)},
    dataType:  'text',
    method:  'GET'
});
```

A more modern syntax uses the `jqXHR` object returned by `$.ajax()` and its implementation of the *Promise* interface:

```
$.ajax( {
    url:  'HelloWorld',
    data:  { userName :  $('#userName').val()},
    success:  function(responseText){$('#ajaxResponse').text(responseText);},
    dataType:  'text',
    method:  'GET'
}).done(function(responseText){$('#ajaxResponse').text(responseText);)};
```

There are also `.fail()` and `.always()` available to assign multiple *callbacks* on a single request.

Another possibility is the use of the shorthand syntax of `$.get()` and `$.post()`:

```
$.get( { 'HelloWorld', { userName :  $('#userName').val()},
    function(responseText){$('#ajaxResponse').text(responseText);},
});
```

Do not forget that to every Ajax request there must be a Servlet which returns the requested information!
It is also important to set the correct content type of the response in the servlet:

- `response.setContentType()`
  - e.g. *text/plain*, *text/html*, *application/json*, *application/html*
- Then the response is automatically parsed by jQuery into the corresponding object and made available through *responseText*, *responseHTML*, ...

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="ISO-8859-1">
      <title>Ajax Test</title>
      <script src="https://code.jquery.com/jquery-latest.js"></script>
      <script>
          $(document).ready(function(){
             $( "button" ).click(function() {
                $.ajax({
                   url:'ajaxTest',
                   data:{user : $('#userName').val()},
                   dataType: 'text',
                   method: 'GET'
                }).done(function(responseText){
                   $('#ajaxResponse').text(responseText);
                }).fail(function(){
                   $('#ajaxResponse').text("Not that thing! The other thing!");
                })
             });
          });
      </script>
   </head>
   <body>
      <input type="text" id="userName">
      <button>Do the thing!</button>
      <p id="ajaxResponse"></p>
   </body>
</html>
```

[Filename: Servlet/AjaxTest.html]

```
@WebServlet("/ajaxTest")
public class AjaxServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
                   throws ServletException, IOException {

      String name = request.getParameter("userName");
      if(name != null && name != ""){
         response.getWriter().append("Hello there, ").append(name);
      } else {
         response.getWriter().append("No name submitted!");
      }
   }
}
```

[Filename: Servlet/AjaxServlet.java]

3.5 HTTP clients

...

3.6 TODO...

- HTTPS
- Authentification
  - Secure password storage in DB
  - DB security risks
- REST & SOAP web architectural design
- Web service frameworks:
  - Jax-RS / Jax-WS
  - JavaServer Faces (JSF)
  - Spring?
  - Angular / React?