CHAPTER 2

JAKARTA SERVLETS

https://jakarta.ee/specifications/servlet/6.0/jakarta-servlet-spec-6.0

Chapter 2: Jakarta Servlets

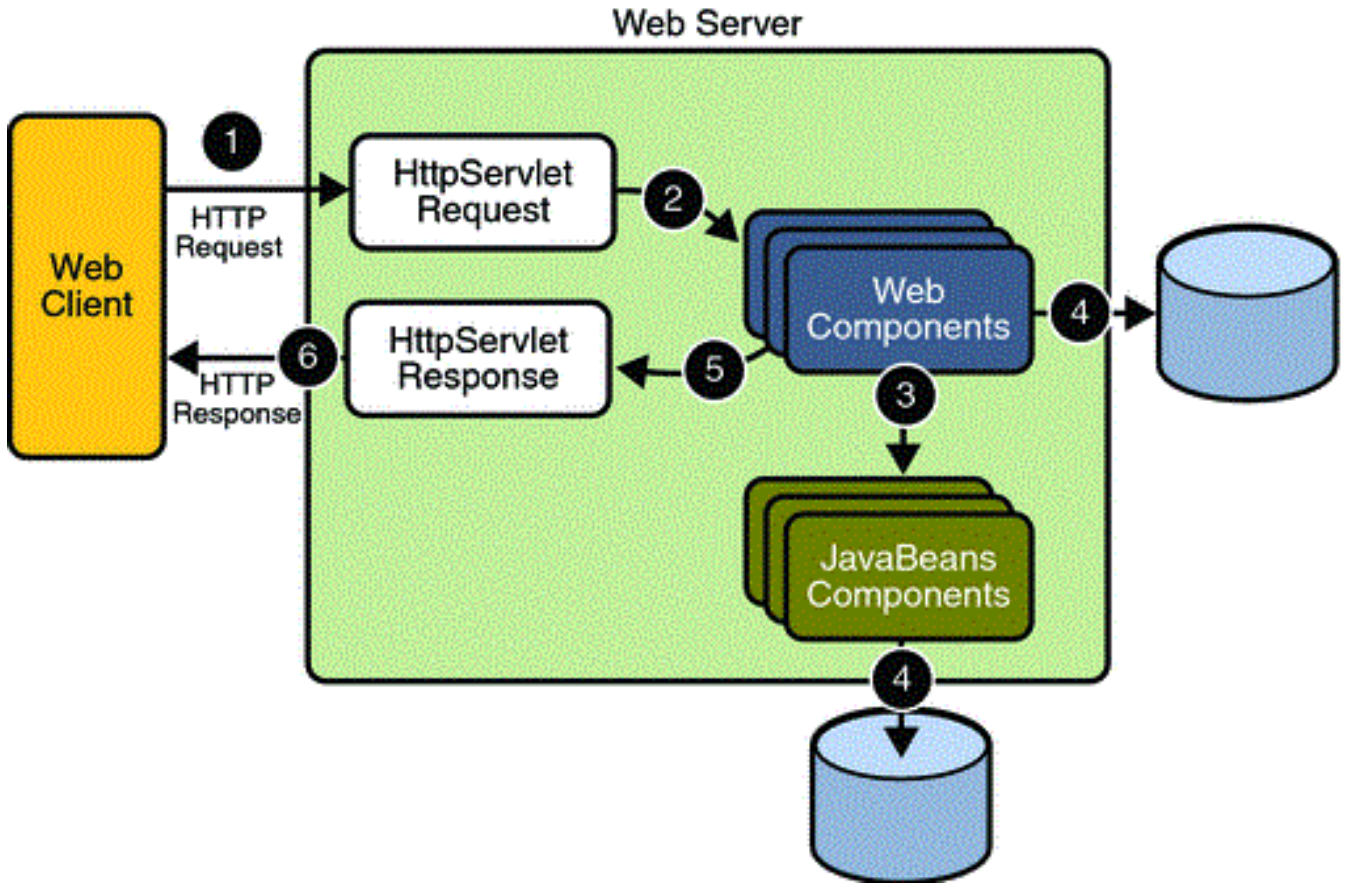
1. What is a Servlet?
2. Servlets Architecture
3. Servlet Lifecycle
4. Asynchronous Processing
5. Contexts and Dependency Injection (CDI)
6. REST API
7. WebSocket

What is a Servlet?

- Java Servlet is a java object file developed as a component an runs on the server.
- Servlets are programs that run on a Web or Application server and act as a middle layer between a request coming from a Web browser or other HTTP client and databases or application on the HTTP server.
- Servlets is a component can be invoked from HTML.
- Servlet do not display a graphical interface to the user.
- A servlet's work is done at the server and only the results of the servlet's processing are returned to the client in the form of HTML
- A Servlet is a Java technology-based Web component, managed by a container that generates dynamic content.
- Like other Java technology-based components, Servlets are platform-independent Java classes that are compiled to platform-neutral byte code that can be loaded dynamic into and run by a Java technology-enabled Web server.

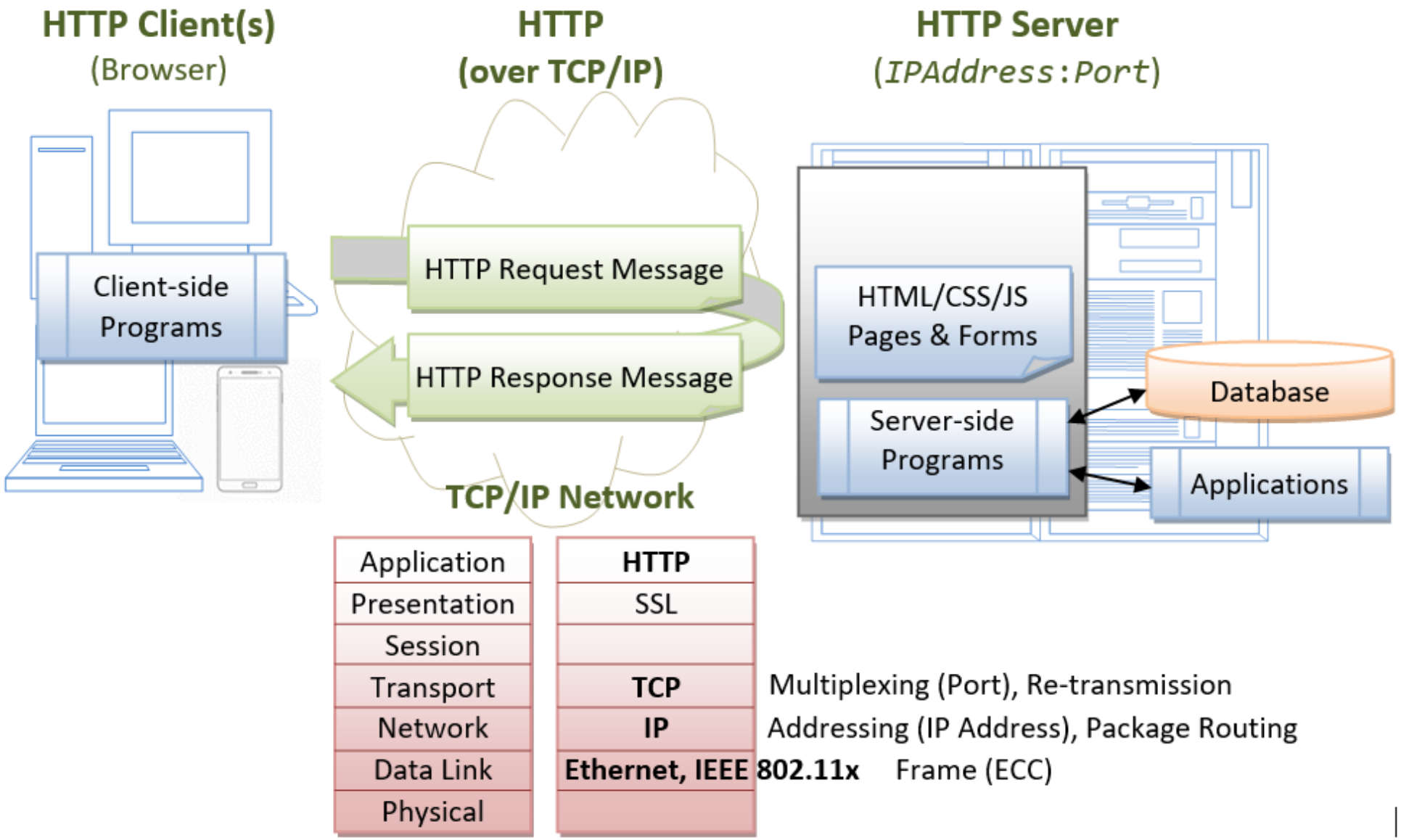
What is Servlet?

- Containers, sometimes called servlet engines, are web server extensions that provide servlet functionality. Servlets interact with web clients via a request/response paradigm implemented by the servlet container.



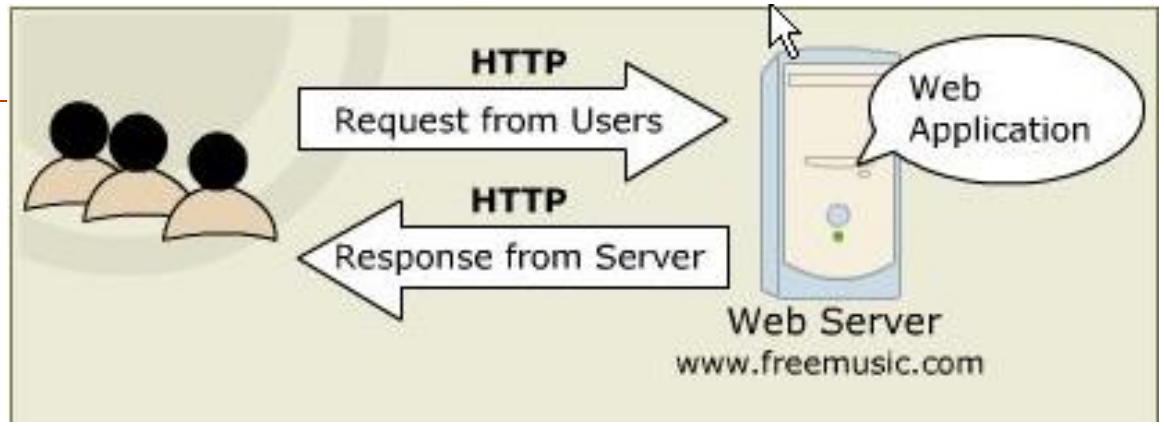
Servlet Request/Response Handling

Java Web Application Request Handling



Servlet Communication & Protocols

HTTP Message



HTTP over TCP/IP

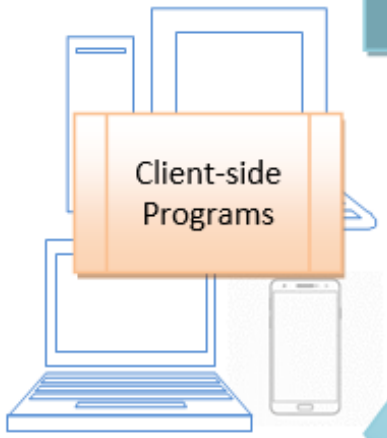
Request Message

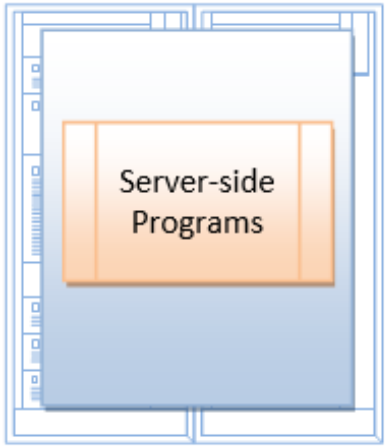
HTTP Client(s)

http://xyz.com/home.html

```
GET /home.html HTTP/1.1
Host: xyz.com
Connection: Keep-Alive
User-Agent: Mozilla/4.0
Accept: image/gif, image/jpeg
----- Blank line ------
(Empty body)
```

HTTP Server





Response Message

```
HTTP/1.1 200 OK
Date: ...
Server: Apache/2.0.45
Last-Modified: ...
Content-Length: 105
Content-Type: text/html
----- Blank line ------
<html>
<head><title>My Home</title></head>
<body><h1>This is my Home Page</h1>
</body></html>
```

Execution of Servlets

1. The clients send the request to the Web server.
2. The Web server receives the request.
3. The Web server passes the request to the corresponding Servlet.
4. The Servlet read:
   - The explicit data, this includes an HTML form on a Web page or a custom HTTP client program.
   - The implicit HTTP request data, this includes cookies, media types and compression schemes the browser understands.
5. The Servlet processes the data and generate the results (the response on the form of output).
6. The Servlet send the response back to the Web server.
7. The Web server sends the response back to the Client and the Client Browser displays it on the screen.

Servlet Packages

1. **jakarta.servlet**: contains a number of classes and interfaces that describe and define the contracts between a **servlet class** and the **runtime environment** provided for an instance of such a class by a conforming **servlet container**.
2. **jakarta.servlet.http**: contains a number of classes and interfaces that describe and define the contracts between a **servlet class** running under the **HTTP protocol** and **the runtime environment** provided for an instance of such a class by a conforming servlet container.
3. **jakarta.servlet.annotation:** contains annotations to annotate servlets, filters, and listeners. It also specifies metadata for annotated components.
4. **jakarta.servlet.descriptor:** contains types that provide programmatic access to a web application's configuration information.

jakarta.servlet Packages

Interfaces	Classes
Servlet	GenericServlet
ServletConfig	ServletRequestWrapper
ServletContext	ServletResponseWrapper
ServletRequest	ServletInputStream
ServletResponse	ServletOutputStream
RequestDipatcher	ServletContextEvent
Filter	ServletContextAttributeEvent
FilterChain	ServletRequestEvent
FilterConfig	ServletRequestAttributeEvent
ServletRequestListener	ServletException
ServlerRequestAttributeListener	UnavailableException
ServlerContextListener	
ServletContextAttributeListener	

jakarta.servlet.http Packages

Interfaces	Classes
HttpServletRequest HttpServletResponse HttpSession HttpSessionAttributeLisrener HttpSessionAttributeListenner HttpSessionBindingListener HttpSessionActivationListener	Cookie HttpServlet HttpServletRequestWrapper HttpServletResponseWrapper HttpSessionEvent HttpSessionBindingEvent

Servlet vs GenericServlet vs HttpServlet

1. **jakarta.servlet.Servlet**: is the top level interface in the hierarchy of Java servlets which defines all the necessary methods to be implemented by the servlets.

2. **jakarta.servlet.GenericServlet:** is an abstract class which implements jakarta.servlet.Servlet interface and provides methods to write protocol-independent servlets.

3. **jakarta.servlet.http.HttpServlet:** is also an abstract class which extends jakarta.servlet.GenericServlet and provides methods to write HTTP-specific servlets.

	Servlet	GenericServlet	HttpServlet
What is it?	Interface	Abstract Class	Abstract Class
Package	jakarta.servlet	jakarta.servlet	jakarta.servlet.http
Hierarchy	Top level interface	Implements Servlet Interface	Extends GenerticServlet
Methods	init() service() destroy() getServletConfig() getServletInfo()	init() service() destroy() getServletConfig() getServletInfo() log() getInitParameter() getInitParameterNames() getServletContext() getServletName()	doGet() doPost() doPut() doDelete() doHead() doOptions() doTrace() getLastModified() service()
Abstract Methods	All methods are abstract	Only service() method is abstract	No abstract method
When to use	Use it when we want to develop own Servlet container	Use to write protocal independent servlets	Use to write HTTP-specific servlets

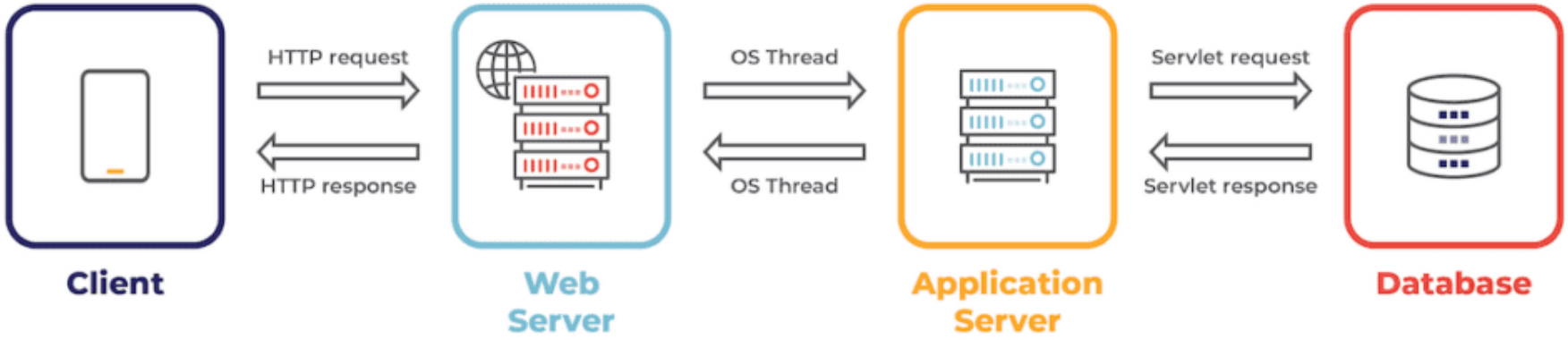
HTTP methods and Servlet method

HTTP – Servlet methods mapping

Http method	Description	Servlet Method
GET	The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.	doGet
POST	The POST method submits an entity to the specified resource, often causing a change in state or side effects on the server.	doPost
PUT	The PUT method replaces all current representations of the target resource with the request payload.	doPut
DELETE	The DELETE method deletes the specified resource.	doDelete
HEAD	The HEAD method asks for a response identical to a GET request, but without the response body.	doHead
OPTIONS	The OPTIONS method describes the communication options for the target resource.	doOptions
TRACE	The TRACE method performs a message loop-back test along the path to the target resource.	doTrace
CONNECT	The CONNECT method establishes a tunnel to the server identified by the target resource.	
PATCH	The PATCH method applies partial modifications to a resource.	

Java Web Server vs. Application Server

- Web Server is a computer program that accepts the request for data and sends the specified documents. Web server may be a computer where the online content is kept. Essentially internet server is employed to host sites however there exist different web servers conjointly like recreation, storage, FTP, email, etc.
- Application Server encompasses Web container as well as EJB container. Application servers organize the run atmosphere for enterprises applications. Application server may be a reasonably server that mean how to put operating system, hosting the applications and services for users, IT services and organizations. In this, user interface similarly as protocol and RPC/RMI protocols are used.

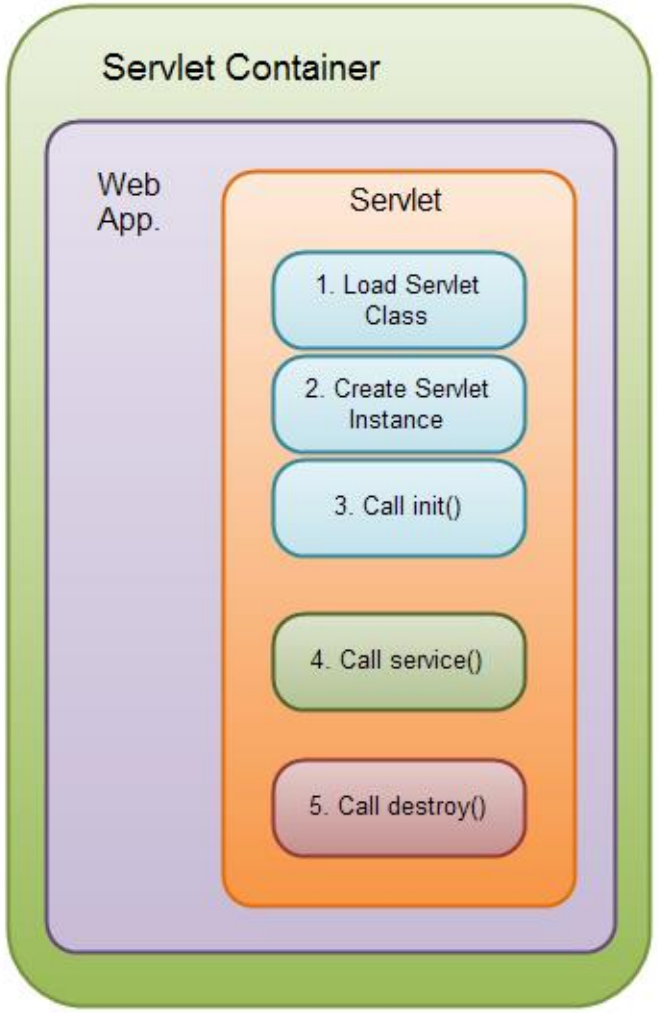


https://aws.amazon.com/compare/the-difference-between-web-server-and-application-server/

Java Web Server vs. Application Server

	Web Server	Application Server
1.	Web server encompasses web container only.	While application server encompasses Web container as well as EJB container.
2.	Web server is useful or fitted for static content.	Whereas application server is fitted for dynamic content.
3.	Web server consumes or utilizes less resources.	While application server utilize more resources.
4.	Web servers arrange the run environment for web applications.	While application servers arrange the run environment for enterprises applications.
5.	In web servers, multithreading is supported.	While in application server, multithreading is not supported.
6.	Web server's capacity is lower than application server.	While application server's capacity is higher than web server.
7.	In web server, HTML and HTTP protocols are used.	While in this, GUI as well as HTTP and RPC/RMI protocols are used.
8.	Processes that are not resource-intensive are supported.	Processes that are resource-intensive are supported.
9.	Transactions and connection pooling is not supported.	Transactions and connection pooling is supported.
10.	The capacity of fault tolerance is low as compared to application servers.	It has high fault tolerance.
11.	Web Server examples are Apache HTTP Server , Tomcat, Jetty, Nginx,...	Application Servers example are JBoss EAP, WildFly, Glassfish, Apache TomEE, IBM Websphere,….

Servlet Lifecycle



There are 5 steps:

1. Load **Servlet** Class.
2. Create Instance of **Servlet**.
3. Call the servlets **init()** method.
4. Call the servlets **service()** method.
5. Call the servlets **destroy()** method.

Step 1, 2 and 3 are executed **only once**, when the servlet is initially loaded

Step 4 is executed **multiple times** - once for every HTTP request to the servlet.

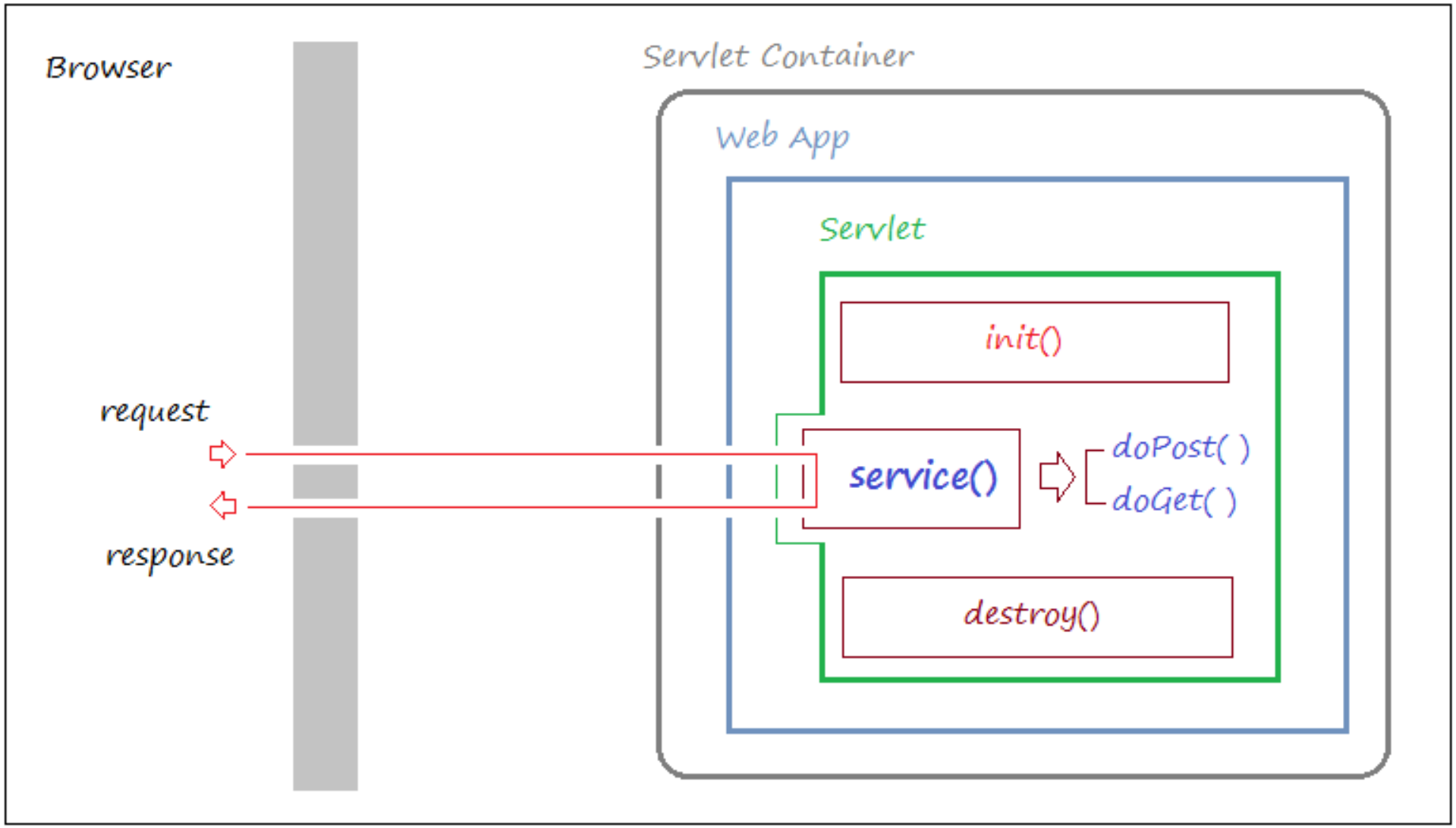
Step 5 is executed when the servlet container **unloads** the servlet.

Servlet Lifecycle (cont.)

init, service, and destroy are **lifecycle methods**

1) void init(ServletConfig config): called when servlet is instantiated; return before any other methods will be called.
2) void service(ServletRequest req, ServletResponse res): method called directly by server when an HTTP request is received.
3) void destroy(): called when server shuts down. Cleans up whatever resources are being held (e.g., memory, file handles, threads) and makes sure that any persistent state is synchronized with the servlet's current in-memory state.
4) ServletConfig getServletConfig(): returns a servlet config object, which contains any initialization parameters and startup configuration for this servlet.
5) String getServletInfo(): returns a string containing information about the servlet, such as its author, version and copyright.

Servlet Lifecycle (cont.)

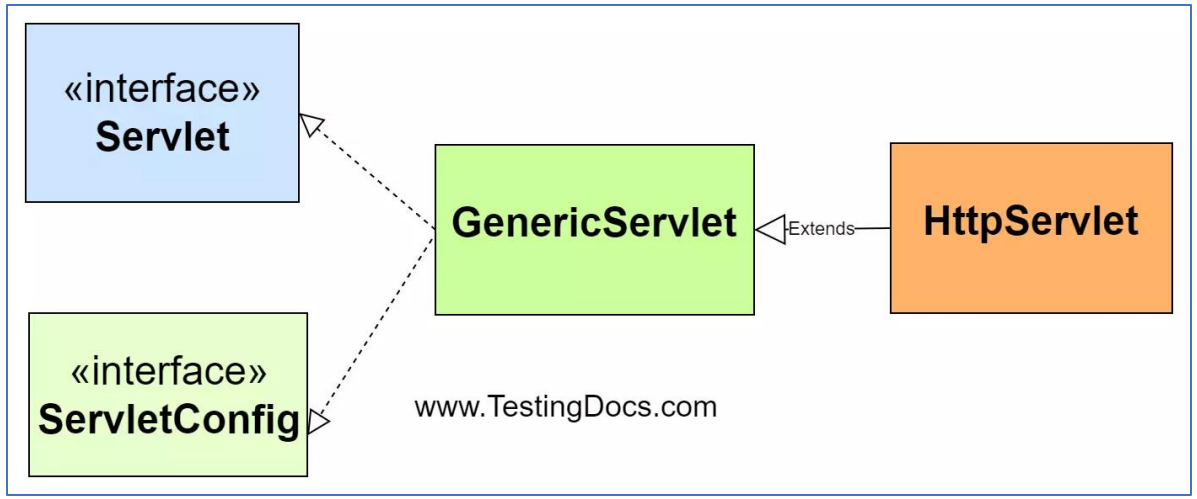


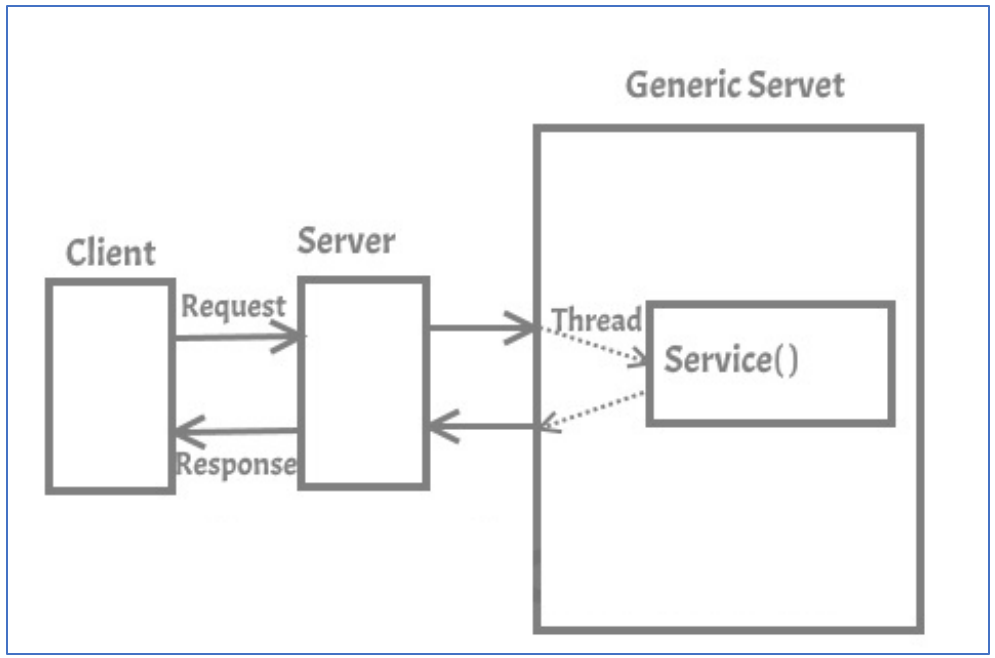
Anatomy of a Servlet

1. HTTPServletRequest object
   - Information about an HTTP request
     - ✓ Headers
     - ✓ Query String
     - ✓ Session
     - ✓ Cookies
2. HTTPServletResponse object
   - Used for formatting an HTTP response
     - ✓ Headers
     - ✓ Status codes
     - ✓ Cookies

Types of Servlet

The jakarta.servlet.GenericServlet





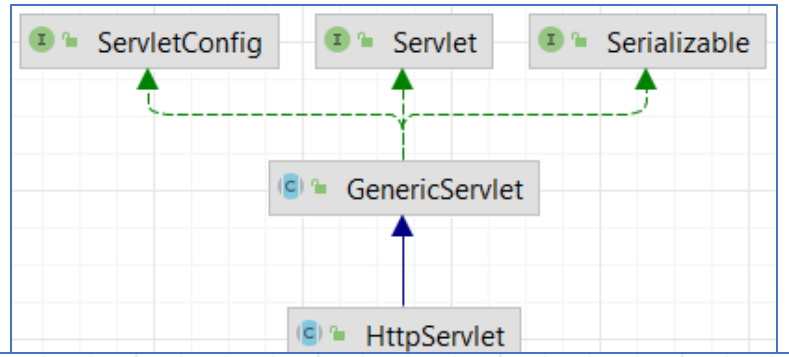

```xml
<servlet>
  <description></description>
  <display-name>myGenericServlet</display-name>
  <servlet-name>myGenericServlet</servlet-name>
  <servlet-class>iuh.edu.fit.servletdemo.MyGenericServlet</servlet-class>
  <init-param>
      <param-name>greeting</param-name>
      <param-value>Hello</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>myGenericServlet</servlet-name>
  <url-pattern>/myGenericServlet</url-pattern>
</servlet-mapping>
<context-param>
  <param-name>name</param-name>
  <param-value>Student Name</param-value>
</context-param>
```

```java
// @WebServlet(name = "myGenericServlet", value="/myGenericServlet")
public class MyGenericServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    private String initParam;

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        initParam = config.getInitParameter("greeting");
    }

    protected void service(HttpServletRequest request, HttpServletResponse resp
        ServletContext ctx = this.getServletContext();
        String name = ctx.getInitParameter("name");
        response.getWriter().println(initParam + " GenericServlet by " + name);
    }

}
```

```
Hello GenericServlet by Student Name
```

Types of Servlet

The jakarta.servlet.http.HttpServlet



HttpServlet

getLastModified(HttpServletRequest)	long
doTrace(HttpServletRequest, HttpServletResponse)	void
maybeSetLastModified(HttpServletResponse, long)	void
init(ServletConfig)	void
getMethodNotSupportedCode(String)	int
doPost(HttpServletRequest, HttpServletResponse)	void
doOptions(HttpServletRequest, HttpServletResponse)	void
getAllDeclaredMethods(Class<HttpServlet>)	Method[]
doHead(HttpServletRequest, HttpServletResponse)	void
doPut(HttpServletRequest, HttpServletResponse)	void
doGet(HttpServletRequest, HttpServletResponse)	void
doDelete(HttpServletRequest, HttpServletResponse)	void
service(HttpServletRequest, HttpServletResponse)	void
service(ServletRequest, ServletResponse)	void

```java
@WebServlet(name = "helloServlet", value = "/hello-servlet")
public class HelloServlet extends HttpServlet {
    private String message;
    public void init() { message = "Hello World!"; }

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response) throws IOException {...}

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {...}
}
```

```jsp
<body>
    <h1><%= "Hello World!" %>
    </h1>
    <br/>
    <a href="hello-servlet">Hello Servlet</a>
    <br/>
    <a href="myGenericServlet">My Generic Servlet</a>
</body>
```

Hello World!

Hello Servlet
My Generic Servlet

GenericServlet class - Methods

1) **public void init(ServletConfig config):** is used to initialize the servlet.

2) **public abstract void service(ServletRequest request, ServletResponse response):** provides service for the incoming request. It is invoked at each time when user requests for a servlet.

3) **public void destroy():** is invoked only once throughout the life cycle and indicates that servlet is being destroyed.

4) **public ServletConfig getServletConfig():** returns the object of ServletConfig.

5) **public String getServletInfo():** returns information about servlet such as writer, copyright, version etc.

GenericServlet class – Methods (cont.)

7) **public ServletContext getServletContext():** returns the object of ServletContext.

8) **public String getInitParameter(String name):** returns the parameter value for the given parameter name.

9) **public Enumeration getInitParameterNames():** returns all the parameters defined in the web.xml file.

10) **public String getServletName():** returns the name of the servlet object.

11) **public void log(String msg):** writes the given message in the servlet log file.

12) **public void log(String msg,Throwable t):** writes the explanatory message in the servlet log file and a stack trace.

ServletRequest interface

1) For every HTTP request, the servlet container creates an instance of ServletRequest and passes it to the servlet's service method . An object of ServletRequest is used to provide the client request information to a servlet such as content type, content length, parameter names and values, header informations, attributes etc.

2) Some of the methods in the **ServletRequest**:
   - ✓ **int getContentLength():** returns the number of bytes in the request body. If the length is not known, this method returns -1.
   - ✓ **java.lang.String getContentType():** returns the MIME type of the request body or null if the type is not known
   - ✓ **java.lang.String getParameter(java.lang.String *name*):** returns the value of the specified request parameter.

ServletResponse interface

1) Prior to invoking a servlet's **service method**, the **servlet container** creates a **ServletResponse** and pass it as the **second argument** to the **service method**

2) Some of the methods in the **ServletResponse**:

   ✓ **java.io.PrintWriter getWriter ():** returns a PrintWriter that can send text to the client. Before sending any HTML tag, you should set the content type of the response by calling the setContentType method, passing "text/html" as an argument

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

   ✓ **ServletOutputStream getOutputStream():** returns a suitable for writing binary data in the response.

ServletConfig interface

1) The **servlet container** passes a **ServletConfig** to the **servlet's init method** when the servlet container **initializes the servlet**
2) The **ServletConfig** encapsulates configuration information that you can pass to a servlet through **@WebServlet** or the deployment descriptor (in ***web.xml***)
3) Some of the methods in the **ServletConfig:**
   - **public String getInitParameter(String name):** returns the parameter value for the specified parameter name.
   - **public Enumeration getInitParameterNames():** returns an enumeration of all the initialization parameter names.
   - **public String getServletName():** returns the name of the servlet.
   - **public ServletContext getServletContext():** returns an object of ServletContext.

ServletContext interface - Usage

1) The **ServletContext** represents the **servlet application**. There is **only one context per web application**

2) We can obtain the **ServletContext** by calling the **getServletContext** method on the **ServletConfig** or **getServletContext** method in **ServletRequest**

3) We can share information that can be accessed **from all resources** in the **application** and to **enable dynamic registration of web objects. Objects** stored in **ServletContext** are called **attributes**

ServletContext interface - Usage (cont.)

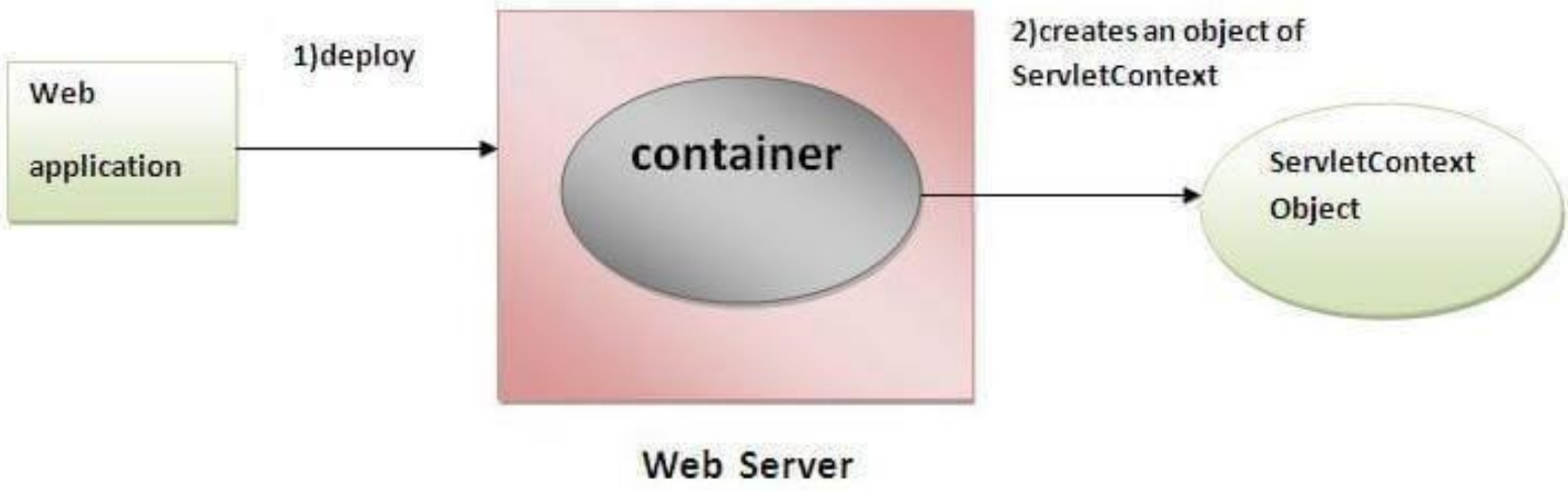
1. The object of ServletContext provides an interface between the container and servlet.

2. The ServletContext object can be used to get configuration information from the web.xml file.

3. The ServletContext object can be used to set, get or remove attribute from the web.xml file.

4. The ServletContext object can be used to provide inter-application communication.

ServletContext interface (cont.)

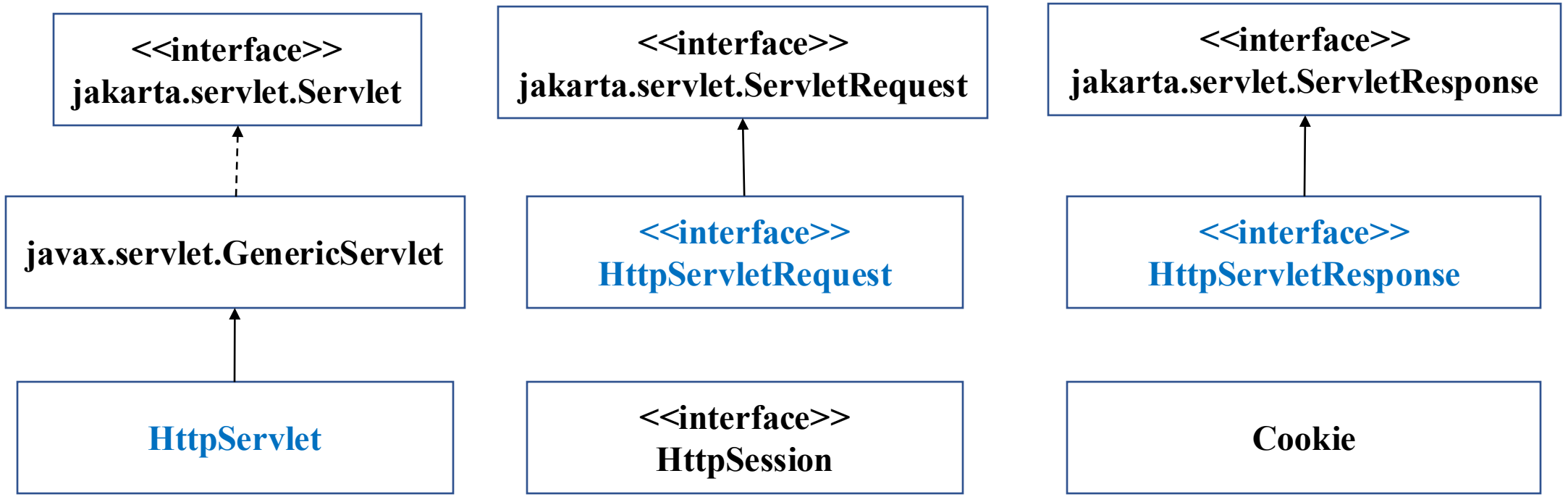


ServletContext interface - Methods

- **public String getInitParameter(String name):** returns the parameter value for the specified parameter name.
- **public Enumeration getInitParameterNames():** returns the names of the context's initialization parameters.
- **public void setAttribute(String name,Object object):** sets the given object in the application scope.
- **public Object getAttribute(String name):** returns the attribute for the specified name.
- **public Enumeration getInitParameterNames():** returns the names of the context's initialization parameters as an Enumeration of String objects.
- **public void removeAttribute(String name):** removes the attribute with the given name from the servlet context.

HTTP Servlets

- Most, servlet applications we write will work with HTTP. This means, we can make use of the features offered by HTTP.
- The jakarta.servlet.http package is the second package in the Servlet API that contains classes and interfaces for writing servlet applications.



HTTPServlet

- When using **HttpServlet**, you will work with the **HttpServletRequest** and **HttpServletResponse** objects that represent the **servlet request** and the **servlet response**, respectively
- **HttpServlet** overrides the service method in **GenericServlet** and adds **another service** method
- The **new service** method **in HttpServlet** then examines the **HTTP method used to send the request** (by calling request.getMethod) and call one of the following methods: **doGet**, **doPost**, **doHead**, **doPut**, **doTrace**, **doOptions**, and **doDelete**
- We rarely need to override the **service methods** anymore. Instead, you override **doGet** or **doPost** or both **doGet and doPost**

HTTPServlet - Methods

1) void **doGet** (HttpServletRequest request, HttpServletResponse response): handles **GET** requests
2) void **doPost** (HttpServletRequest request, HttpServletResponse response): handles **POST** requests
3) void **doPut** (HttpServletRequest request, HttpServletResponse response): handles **PUT** requests
4) void **doDelete** (HttpServletRequest request, HttpServletResponse response): handles **DELETE** requests
5) void **doHead** (HttpServletRequest request, HttpServletResponse response) handles the HEAD request
6) void **doOptions** (HttpServletRequest request, HttpServletResponse response) handles the OPTIONS request

HTTPServlet – Methods (cont.)

7) **void doTrace** (HttpServletRequest request, HttpServletResponse response) handles the TRACE request.

8) **void service** (ServletRequest request, ServletResponse response) dispatches the request to the protected service method by converting the request and response object into http type.

9) **void service** (HttpServletRequest request, HttpServletResponse response) receives the request from the service method, and dispatches the request to the doXXX() method depending on the incoming http request type.

HTTPServletRequest

HttpServletRequest represents the servlet request in the **HTTP environment**

1) **java.lang.String getContextPath():** returns the portion of the request URI that indicates the context of the request.
2) **Cookie[] getCookies(): r**eturns an array of **Cookie** objects.
3) **java.lang.String getHeader(java.lang.String name):** returns the value of the specified HTTP header.
4) **java.lang.String getMethod():** returns the name of the HTTP method with which this request was made.
5) **java.lang.String getQueryString():** returns the query string in the request URL.

HTTPServletRequest (cont.)

6) **HttpSession getSession():** returns the session object associated with this request. If none is found, creates a new session object.
7) **HttpSession getSession(boolean create):** returns the current session object associated with this request. If none is found and the create argument is **true**, create a new session object.

HTTPServletResponse

HttpServletResponse represents the **servlet response** in the **HTTP environment.** Here are some of the methods defined in it:

1) **void addCookie(Cookie *cookie*):** adds a cookie to this response object.
2) **void addHeader(java.lang.String name, java.lang.String value):** adds a header to this response object.
3) **void sendRedirect(java.lang.String location):** sends a response code that redirects the browser to the specified location

HTML Forms

- A web application almost always contains **one or more HTML forms** to **take user input**. We can easily send an HTML form from a servlet to the browser. When the user submits the form, **values entered in the form are sent to the server as request**
- The value of an **HTML input field** (a text field, a hidden field, or a password field) or **text area** is sent to the server **as a string**. An **empty input field or text area** sends an **empty string**. As such, **ServletRequest.getParameter** that takes an **input field name never returns null**
- An HTML **select element** also sends a string to the server. If none of the options in the select element is selected, the **value of the option that is displayed is sent**
- A **multiple-value select element** sends a **string array** and has to be handled by **ServletRequest.getParameterValues**

HTML Forms (cont.)

- A **checked checkbox** sends the string **“on”** to the server. **An unchecked checkbox** sends **nothing to the server** and ServletRequest.getParameter(fieldName) **returns null**
- **Radio buttons** send the value of **the selected button** to the server. If none of the buttons is selected, **nothing is sent to the server** and ServletRequest.getParameter(fieldName) **returns null**
- If a form contains **multiple input elements with the same name**, **all values will besubmitted** and we have to use **ServletRequest.getParameterValues** to retrieve them. **ServletRequest.getParameter** will only return the **last value**

Servlet Mapping

1. Map a servlet with an annotation - **@WebServlet**
2. Map a servlet with **Deployment Descriptor** (**web.xml**)

Map a servlet with an annotation - @WebServlet

With **@WebServlet** we can specify:

- **Name**: defines the name of servlet
- **urlPatterns:** maps the servlet to the pattern
- **loadOnStartup:** loads the servlet at the time of deployment or server start if value is positive
- **Description:** description about servlet
- **initParams:** takes multiple **@WebInitParam** annotation

Map a servlet with an annotation - @WebServlet (cont.)

```
@WebServlet(name="/WelcomeServlet",urlPatterns="/WelcomeServlet"
,loadOnStartup=1, description="Welcome Servlet")


@WebServlet(name="/WelcomeServlet",urlPatterns={"/WelcomeServlet",
"/HelloServlet"}, description="Welcome Servlet")



@WebServlet (urlPatterns= {"/emailList", "/email/*"})
```

Map a servlet with an annotation - @WebInitParam

@WebInitParam enables us to configure one init param and

provides name , value and description attribute.

- **name**: name of init param
- **value**: value of init param
- **description**: description of init param

Map a servlet with an annotation - @WebInitParam (cont.)

```
@WebServlet(name="/WelcomeServlet",urlPatterns={"/WelcomeServlet,

initParams={

@WebInitParam(name="Param1",value="Param 1 Value",description="param1"),

@WebInitParam(name="Param2",value="Param 2 Value",description="param1")
})
```

Map a servlet with Deployment Descriptor (web.xml)

XML elements for working with **servlet mapping**:

- **<servlet-name>:** Specifies a unique name for the servlet that's used to identify the servlet within the web.xml file. This element is required for both the **servlet element** and the **servlet mapping element**
- **<servlet-class>:** Specifies the class for the servlet. Note that this element **includes the package and name for the class but not the .class extension.**
- **<url-pattern>:** Specifies the URL or URLs that are mapped to the specified servlet. This pattern **must begin with a front slash.**

Map a servlet with Deployment Descriptor (cont.)

```
<servlet>
      <servlet-name>MyGenericServletDemo</servlet-name>
      <servlet-class>servletapi.MyServletDemo</servlet-class>
</servlet>

<servlet-mapping>
      <servlet-name>MyGenericServletDemo</servlet-name>
      <url-pattern>/my</url-pattern>
</servlet-mapping>
```

Map a servlet with Deployment Descriptor (cont.)

```
<servlet>
      <servlet-name>MyGenericServletDemo</servlet-name>
      <servlet-class>servletapi.MyServletDemo</servlet-class>
      <init-param>
            <description> param1 </description>
            <param-name>Param1</param-name>
            <param-value>Param 1 Value </param-value>
    </init-param>
</servlet>
<servlet-mapping>
      <servlet-name>MyGenericServletDemo</servlet-name>
      <url-pattern>/my</url-pattern>
</servlet-mapping>
```

@WebFilter Annotation

WebFilter is used to define a filter in a web application. Filters are used to perform filtering tasks on either the request to a resource, the response from a resource, or both.

The @WebFilter annotation provides several attributes for configuring filters:

- filterName: The name of the filter.
- urlPatterns: Specifies the URL patterns to which the filter applies.
- servletNames: Specifies the servlet names to which the filter applies.
- dispatcherTypes: Specifies the dispatcher types (REQUEST, FORWARD, INCLUDE, ERROR, ASYNC).
- initParams: Specifies the initialization parameters for the filter.

@WebFilter Annotation (cont.)

```
@WebFilter(urlPatterns = {"/admin/*", "/user/*”, “/login”},
      initParams = {
      @WebInitParam(name = "param1", value = "value1"),
      @WebInitParam(name = "param2", value = "value2")
      }
)
public class AuthenticationFilter implements Filter {
// Filter implementation
}
```

WebFilter – web.xml

```
<filter>
  <display-name>authentication</display-name>
  <filter-name>authentication</filter-name>
  <filter-class>fit.iuh.edu.servlet.AuthenticationFilter</filter-class>
  <init-param>
    <param-name>username</param-name>
    <param-value>test</param-value>
  </init-param>
  <init-param>
    <param-name>password</param-name>
    <param-value>123456</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>authentication</filter-name>
  <url-pattern>/login</url-pattern>
</filter-mapping>
```

@MultipartConfig Annotation

MultipartConfig annotation, which is used to indicate that the servlet on which it is declared expects requests to be made using the multipart/form-data MIME type. Servlets that are annotated with @MultipartConfig can retrieve the Part components of a given multipart/form-data request by calling the request.getPart(String name) or request.getParts() method.

The @MultipartConfig annotation provides several attributes for configuring file uploads:

- location: Specifies the directory location where files will be stored.
- fileSizeThreshold: Specifies the size threshold after which the file will be written to disk.
- maxFileSize: Specifies the maximum size allowed for uploaded files.
- maxRequestSize: Specifies the maximum size allowed for multipart/form-data requests.

@MultipartConfig Annotation (cont.)

```
@WebServlet("/upload")
@MultipartConfig(
      location = "/tmp",
      fileSizeThreshold = 1024 * 1024 * 1, // 1MB
      maxFileSize = 1024 * 1024 * 10, // 10MB
      maxRequestSize = 1024 * 1024 * 25 // 25MB
)
public class FileUploadServlet extends HttpServlet {
// Servlet implementation
}
```

MultipartConfig – web.xml

```
<servlet>
  <description></description>
  <display-name>fileUpload</display-name>
  <servlet-name>fileUpload</servlet-name>
  <servlet-class>fit.iuh.edu.servlet.FileUploadServlet</servlet-class>
  <multipart-config>
      <location>/tmp</location>
      <max-file-size>20848820</max-file-size>
      <max-request-size>418018841</max-request-size>
      <file-size-threshold>1048576</file-size-threshold>
  </multipart-config>
</servlet>
```

Handling Servlet Lifecycle Events

- You can monitor and react to events in a servlet's lifecycle by defining listener objects whose methods get invoked when lifecycle events occur

```
@WebListener()
public class SimpleServletListener implements ServletContextListener,
        ServletContextAttributeListener, HttpSessionListener {
```

Servlet Lifecycle Events		
Object	**Event**	**Listener Interface and Event Class**
Web context	Initialization and destruction	jakarta.servlet.ServletContextListener and ServletContextEvent
Web context	Attribute added, removed, or replaced	jakarta.servlet.ServletContextAttributeListener and ServletContextAttributeEvent
Session	Creation, invalidation, activation, passivation, and timeout	jakarta.servlet.http.HttpSessionListener, jakarta.servlet.http.HttpSessionActivationListener, and HttpSessionEvent
Session	Attribute added, removed, or replaced	jakarta.servlet.http.HttpSessionAttributeListener and HttpSessionBindingEvent
Request	A servlet request has started being processed by web components	jakarta.servlet.ServletRequestListener and ServletRequestEvent
Request	Attribute added, removed, or replaced	jakarta.servlet.ServletRequestAttributeListener and ServletRequestAttributeEvent

Sharing Information

Using Scope Objects

- Collaborating web components share information by means of objects that are maintained as attributes of four scope objects. You access these attributes by using the getAttribute and setAttribute methods of the class representing the scope.

Scope Object	Class	Accessible From	

Web context	jakarta.servlet.ServletContext	Web components within a web context. See [accessing-the-web-context].	// set/get **application** scoped attribute
			req.getServletContext().setAttribute("name", "application scoped attribute");
			String applicationScope = (String) req.getServletContext().getAttribute("a");

Session	jakarta.servlet.http.HttpSession	Web components handling a request that belongs to the session. See [maintaining-client-state].	// set/get **session** scoped attribute
			HttpSession session = req.getSession();
			session.setAttribute("name", "session scoped attribute");
			String sessionScope = (String) session.getAttribute("b");

Request	Subtype of jakarta.servlet.ServletRequest	Web components handling the request.	// set/get **request** scoped attribute
			req.setAttribute("name", "request scoped attribute");
			String requestScope = (String) req.getAttribute("c");
			// send redirect to other servlet
			req.getRequestDispatcher("get-attributes").forward(req, resp);

Page	jakarta.servlet.jsp.JspContext	The Jakarta Server Pages page that creates the object.	<% page.setAttribute("name","page scoped attribute"); String pageStr = page.getAttribute("name").toString(); %>

Sharing Information

Controlling Concurrent Access to Shared Resources

- In a multithreaded server, shared resources can be accessed concurrently. In addition to scope object attributes, shared resources include in-memory data, such as instance or class variables, and external objects, such as files, database connections, and network connections.
- Concurrent access can arise in several situations.
  - Multiple web components accessing objects stored in the web context.
  - Multiple web components accessing objects stored in a session.
  - Multiple threads within a web component accessing instance variables.
- A web container will typically create a thread to handle each request. To ensure that a servlet instance handles only one request at a time, a servlet can implement the SingleThreadModel interface. If a servlet implements this interface, no two threads will execute concurrently in the servlet's service method. A web container can implement this guarantee by synchronizing access to a single instance of the servlet or by maintaining a pool of web component instances and dispatching each new request to a free instance. This interface does not prevent synchronization problems that result from web components' accessing shared resources, such as static class variables or external objects.

Asynchronous Processing

Introduction

- Web containers in application servers normally use a server thread per client request.
- Under heavy load conditions, containers need a large amount of threads to serve all the client requests.
- Scalability limitations include running out of memory or exhausting the pool of container threads.
- To create scalable web applications, you must ensure that no threads associated with a request are sitting idle, so the container can use them to process new requests.
- There are two common scenarios in which a thread associated with a request can be sitting idle:
  - The thread needs to wait for a resource to become available or process data before building the response. For example, an application may need to query a database or access data from a remote web service before generating the response.
  - The thread needs to wait for an event before generating the response. For example, an application may have to wait for a Jakarta Messaging message, new information from another client, or new data available in a queue before generating the response.

Asynchronous Processing

Asynchronous Processing in Servlets

- To enable asynchronous processing on a servlet, set the parameter asyncSupported to true on the @WebServlet annotation as follows:

```
@WebServlet(urlPatterns={"/asyncservlet"}, asyncSupported=true)
public class AsyncServlet extends HttpServlet { ... }
```

- The jakarta.servlet.AsyncContext class provides the functionality that you need to perform asynchronous processing inside service methods. To obtain an instance of AsyncContext, call the startAsync() method on the request object of your service method;

```
public void doGet(HttpServletRequest req, HttpServletResponse resp) {
   ...
   AsyncContext acontext = req.startAsync();
   ...
}
```

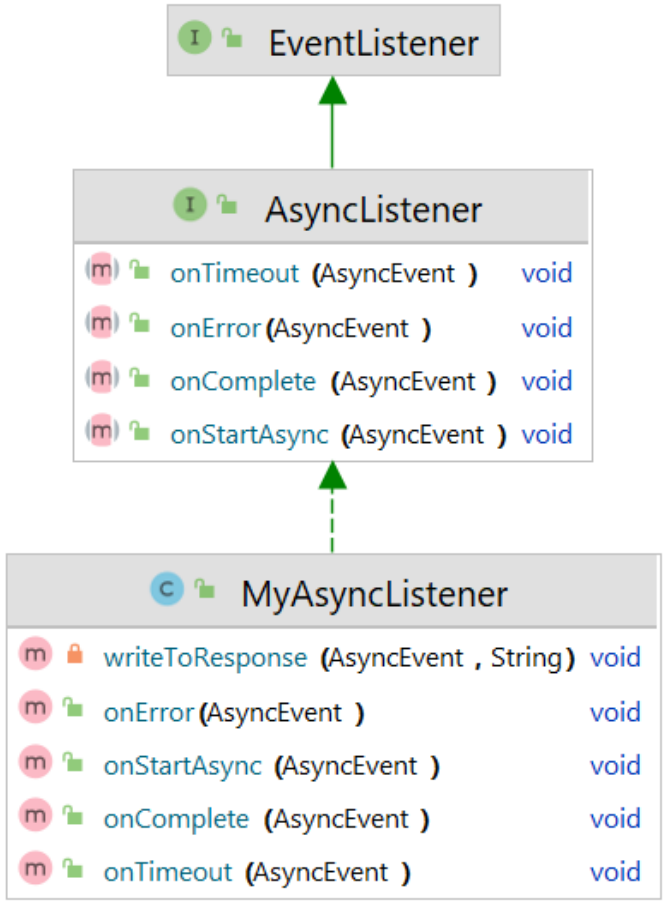
- The AsyncListener class provides the functionality that you can use to listen the states of async thread.

```
public class MyAsyncListener implements AsyncListener {
   //...
}
```

Asynchronous Processing

Example

```java
@WebServlet(name = "asyncServlet", urlPatterns = {"/async-servlet"}, asyncSupported = true)
public class ASyncServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        AsyncContext asyncContext = req.startAsync();

        asyncContext.setTimeout(10000);

        asyncContext.addListener(new MyAsyncListener());

        asyncContext.start(() -> {
            long start = System.nanoTime();

            try {
                Thread.sleep( millis: ThreadLocalRandom.current().nextInt( origin: 1,  bound: 100) * 100);
                PrintWriter out = asyncContext.getResponse().getWriter();
                out.println(Thread.currentThread().getName() + " - Task completed in " + (System.nanoTime() - start) + " ns");
                out.flush();
            } catch (InterruptedException | IOException e) {
                throw new RuntimeException(e);
            }
            finally {
                asyncContext.complete();
            }

            System.out.println("Time to completed long task " + (System.nanoTime() - start));
        });
    }
}
```



Read more: https://github.com/eclipse-ee4j/jakartaee-tutorial/blob/master/src/main/asciidoc/servlets/servlets012.adoc

Contexts and Dependency Injection (CDI)

Introduction

- CDI (Contexts and Dependency Injection) is a standard dependency injection framework included in Java EE 6 and higher.
- It allows us to manage the lifecycle of stateful components via domain-specific lifecycle contexts and inject components (services) into client objects in a type-safe way
- CDI
  - Contexts: The ability to bind the lifecycle and interactions of stateful components to well-defined but extensible lifecycle contexts
  - Dependency injection: The ability to inject components into an application in a type-safe way, including the ability to choose at deployment time which implementation of a particular interface to inject.
- In Tomcat Server, it's needed to add reference to jboss weld dependency

Read more:
https://github.com/eclipse-ee4j/jakartaee-tutorial/tree/master/src/main/asciidoc/cdi-basic

Contexts and Dependency Injection (CDI)

Maven – Weld dependency: pom.xml

```
<!-- https://mvnrepository.com/artifact/org.jboss.weld.servlet/weld-servlet-core -->

<dependency>
  <groupId>org.jboss.weld.servlet</groupId>
  <artifactId>weld-servlet-core</artifactId>
  <version>5.1.2.Final</version>
</dependency>
```

WEB-INF/beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="https://jakarta.ee/xml/ns/jakartaee"

xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee https://jakarta.ee/xml/ns/jakartaee/beans_4_0.xsd"

version="4.0" bean-discovery-mode="none/all/annotated">

</beans>
```

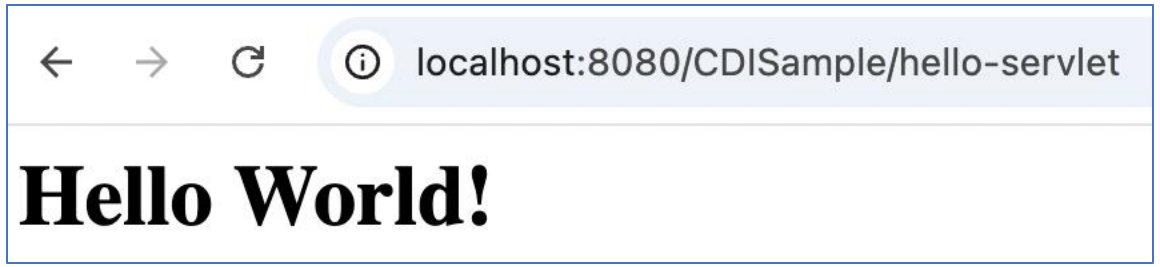
Contexts and Dependency Injection (CDI)

Example

```
@RequestScoped  1 usage
public class World {
    public String world() {  1 usage
        return "World";
    }
}
```

```
@RequestScoped  2 usages
public class Hello {
    @Inject  1 usage
    private World world;
    public String helloWord() {  1 usage
        return "Hello " +  world.world() + "!";
    }
}
```

```
@WebServlet(name = "helloServlet", value = "/hello-servlet")
public class HelloServlet extends HttpServlet {
    @Inject  1 usage
    private Hello hello;
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><body>");
        out.println("<h1>" + hello.helloWord() + "</h1>");
        out.println("</body></html>");
        out.flush();
        out.close();
    }
    public void init() {}
    public void destroy() {}
}
```



REST API

Ref: https://jakarta.ee/learn/docs/jakartaee-tutorial/current/websvcs/jaxrs/jaxrs.html

RESTful Web Services

Introduction

- RESTful web services are loosely coupled, lightweight web services that are particularly well suited for creating APIs for clients spread out across the internet.
- Representational State Transfer (REST) is an architectural style of client-server application centered around the transfer of representations of resources through requests and responses.
- In the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs), typically links on the Web.
- The resources are represented by documents and are acted upon by using a set of simple, well-defined operations.

- URI
- https://domain.com/api/v1/customers
- https://domain.com/api/v1/customers/{customer-id}
- https://domain.com/api/v2/customers/{customer-id}/orders
- https://domain.com/api/v2/customers/{customer-id}/orders/{order-id}

RESTful Root Resource

Developing RESTful Web Services with Jakarta REST

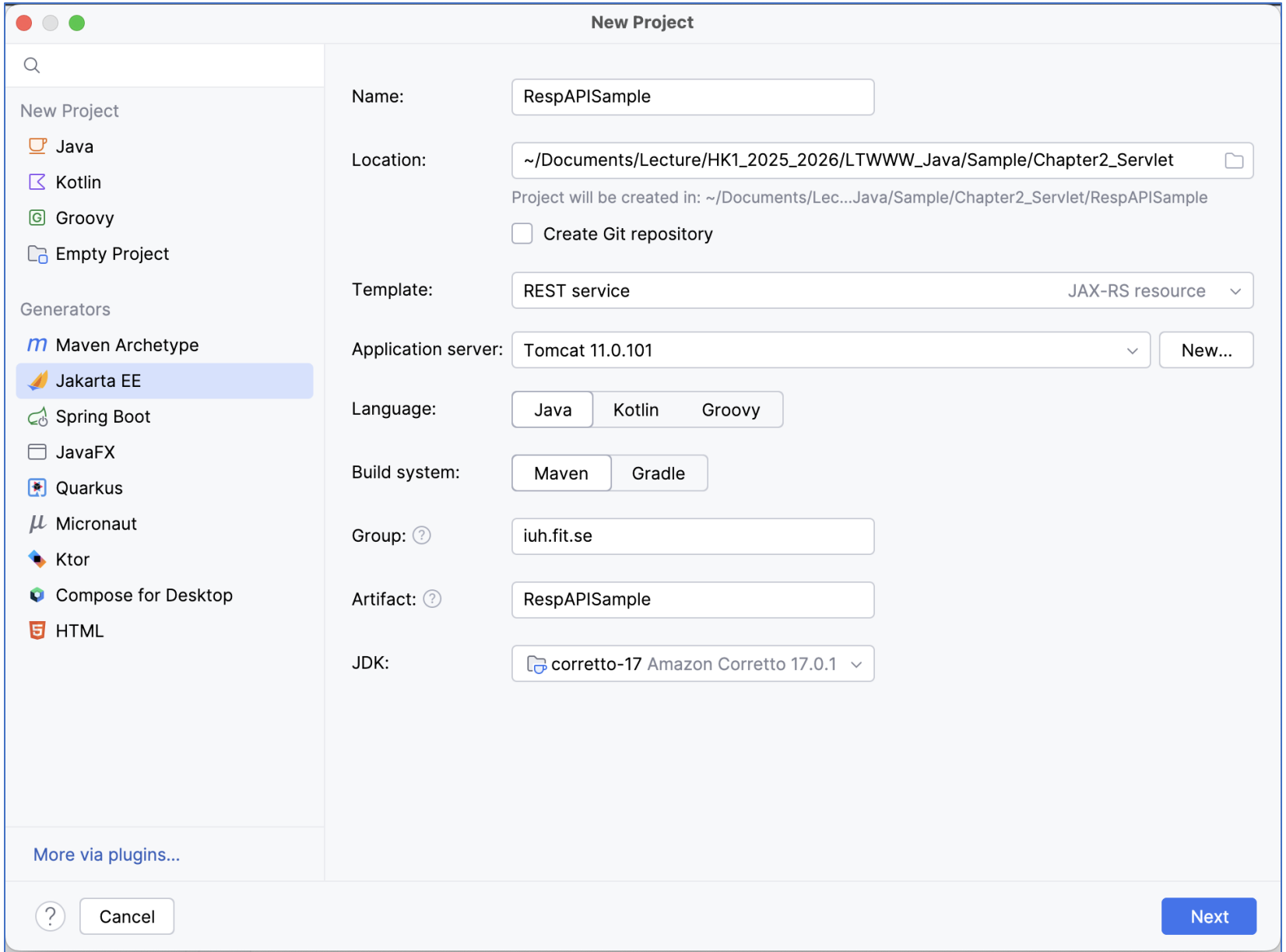
- Jakarta REST is a Java programming language API designed to make it easy to develop applications that use the REST architecture.
- The Jakarta REST API uses Java programming language annotations to simplify the development of RESTful web services.
- Developers decorate Java programming language class files with Jakarta REST annotations to define resources and the actions that can be performed on those resources.

Jakarta REST Annotations		
@Path	@PATCH	@Consumes
@GET	@HEAD	@Produces
@POST	@OPTIONS	@Provider
@PUT	@PathParam	@ApplicationPath
@DELETE	@QueryParam	

https://github.com/eclipse-ee4j/jakartaee-tutorial/blob/master/src/main/asciidoc/jaxrs/jaxrs002.adoc

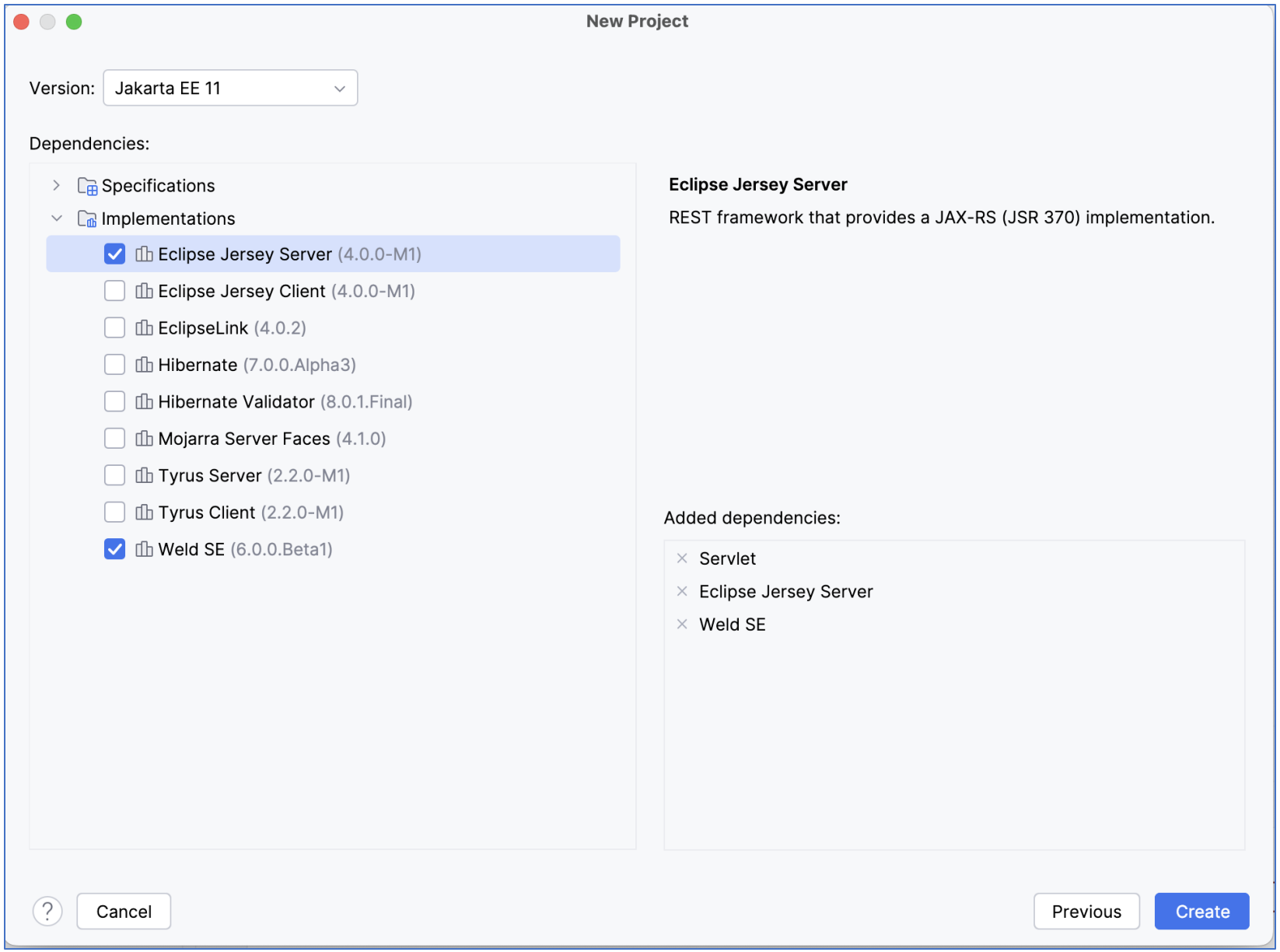
RESTful API - example

Create Project



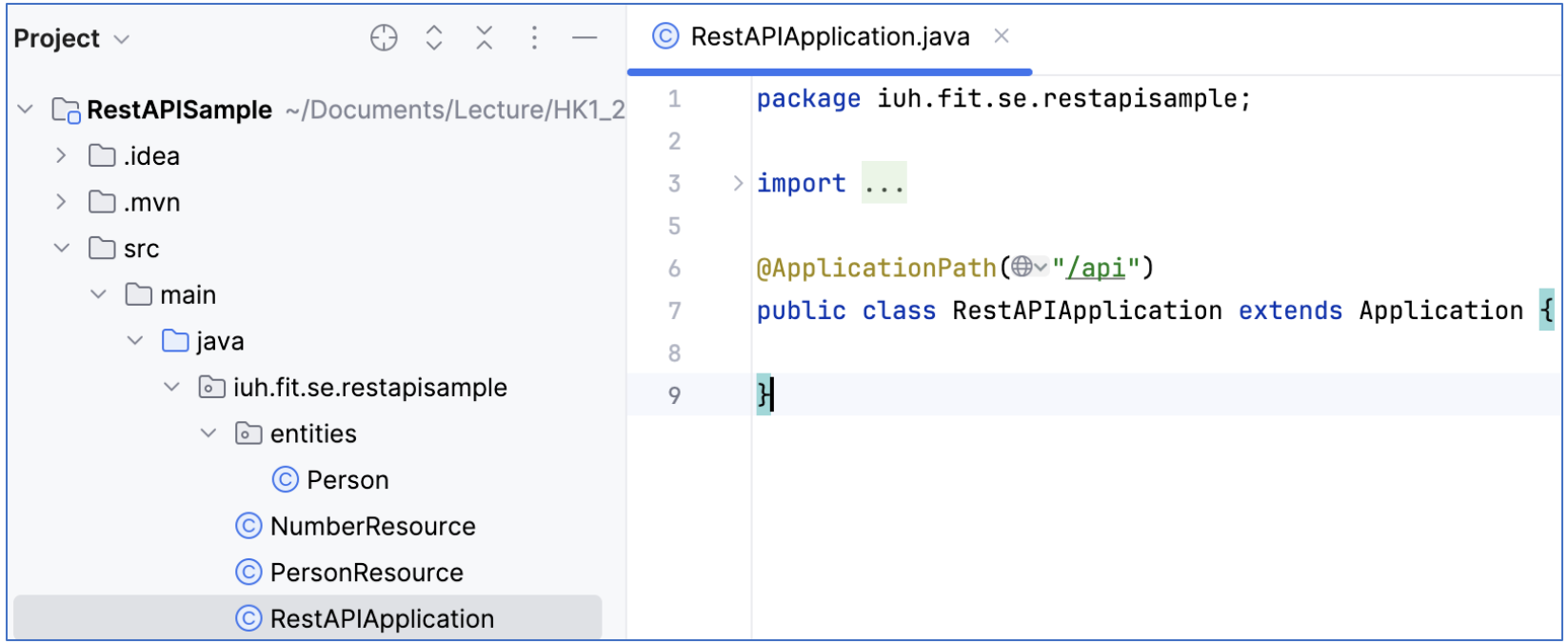
RESTful API – example (cont.)

Create Project (cont.)



RESTful API – example (cont.)

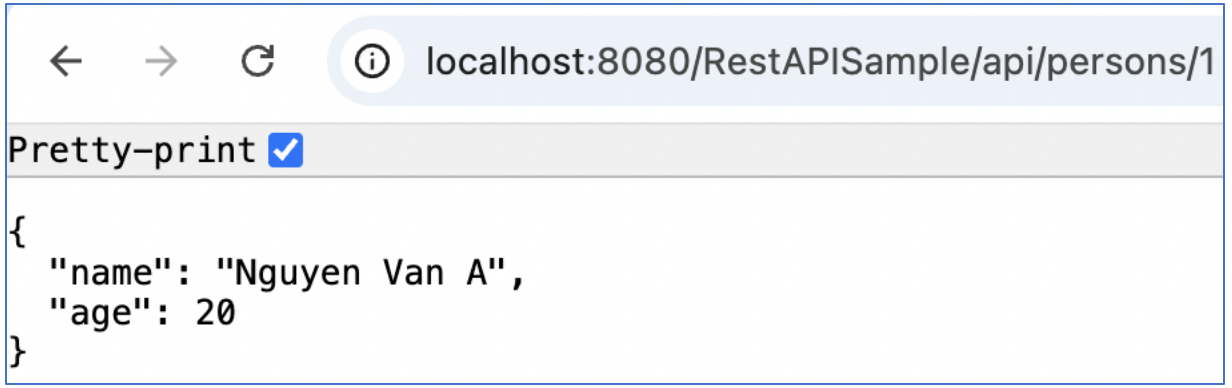
Create Project (cont.)



REST – example (cont.)

```java
@Path("/persons")
public class PersonResource {
    // Endpoint: /api/persons/{person-id} -> Get person information (get by person-id)
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Path("/{person-id}")
    public Person getPerson() {
        Person person = new Person();
        person.setName("Nguyen Van A");
        person.setAge(20);

        return person;
    }

    // Endpoint: /api/persons -> Insert
    @POST
    @Produces(MediaType.TEXT_PLAIN)
    @Consumes(MediaType.APPLICATION_JSON)
    public String handlePersonRequest(Person person) {
        return person.toString();
    }

    // Endpoint: /api/persons -> Get List
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Person> getListPerson() {...}
}
```

REST – example (cont.)




```
thoaha@Has-MacBook-Pro ~ % curl -v http://localhost:8080/RestAPISample/api/persons/1
*   Trying [::1]:8080...
* Connected to localhost (::1) port 8080
> GET /RestAPISample/api/persons/1 HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.4.0
> Accept: */*
>
< HTTP/1.1 200
< Content-Type: application/json
< Content-Length: 32
< Date: Thu, 21 Aug 2025 01:03:48 GMT
<
* Connection #0 to host localhost left intact
{"name":"Nguyen Van A","age":20}%
```

REST – example (cont.)

GET http://localhost:8080/Res

http://localhost:8080/RestAPISample/api/persons/1

GET http://localhost:8080/RestAPISample/api/persons/1

Params Authorization Headers (6) Body Pre-request Script

Query Params

Key

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1  {
2      "name": "Nguyen Van A",
3      "age": 20
4  }
```

REST

The @Path Annotation and URI Path Templates

- The @Path annotation identifies the URI path template to which the resource responds and is specified at the class or method level of a resource.
- The @Path annotation's value is a partial URI path template relative to the base URI of the server on which the resource is deployed, the context root of the application, and the URL pattern to which the Jakarta REST runtime responds.
- URI path templates are URIs with variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI. Variables are denoted by braces ({ and }).

REST

Types Supported for HTTP Request and Response Entity Bodies

- Using Entity Providers to Map HTTP Response and Request Entity Bodies
  - ➢Entity providers supply mapping services between representations and their associated Java types.
  - ➢The two types of entity providers are MessageBodyReader and MessageBodyWriter.

Java Type	**Supported Media Types**
byte[]	All media types (*/*)
java.lang.String	All text media types (text/*)
java.io.InputStream	All media types (*/*)
java.io.Reader	All media types (*/*)
java.io.File	All media types (*/*)
jakarta.activation.DataSource	All media types (*/*)
javax.xml.transform.Source	XML media types (text/xml, application/xml, and application/*+xml)
jakarta.xml.bind.JAXBElement and application-supplied Jakarta XML Binding classes	XML media types (text/xml, application/xml, and application/*+xml)
MultivaluedMap<String, String>	Form content (application/x-www-form-urlencoded)
StreamingOutput	All media types (/), MessageBodyWriter only

REST

Using @Consumes and @Produces to Customize Requests and Responses

- The value of @Produces is an array of String of MIME types or a comma-separated list of MediaType constants. For example:
  - @Produces({"image/jpeg,image/png"})
  - @Produces(MediaType.APPLICATION_XML)
  - @Produces({"application/xml", "application/json"})
- The @Consumes annotation is used to specify which MIME media types of representations a resource can accept, or consume, from the client.
  - If @Consumes is applied at the class level, all the response methods accept the specified MIME types by default.
  - If applied at the method level, @Consumes overrides any @Consumes annotations applied at the class level.
  - Example:
    - @Consumes({"text/plain,text/html"})
    - @Consumes({MediaType.TEXT_PLAIN,MediaType.TEXT_HTML})
    - @Consumes("multipart/related")
    - @Consumes("application/x-www-form-urlencoded")

REST

Extracting Request Parameters

- Parameters of a resource method may be annotated with parameter-based annotations to extract information from a request. A previous example presented the use of the @PathParam parameter to extract a path parameter from the path component of the request URL that matched the path declared in @Path.
- You can extract the following types of parameters for use in your resource class: Query, URI path, Form, Cookie, Header, Matrix

REST – example (cont.)

```java
@Path("/numbers")
public class NumberResource {
    // Endpoint: /api/v1/numbers/{value} ->Path param
    // Endpoint: /api/v1/numbers/{value}?key1=value1 ->/Path param?query param
    @GET
    @Path("/{value}")
    @Produces(MediaType.TEXT_PLAIN)
    public Response checkEven(@PathParam("value") int value) {
        if (value % 2 == 0) {
            return Response.ok( entity: "Value is even").build();
        }
        return Response.status(Response.Status.BAD_REQUEST).entity("Not an even number").build();
    }

    @GET
    @Path("/sum/{a}/{b}")
    @Produces(MediaType.TEXT_PLAIN)
    public Response sum(@DefaultValue("0") @PathParam("a") int a,
                        @DefaultValue("0") @PathParam("b") int b) {
        return Response.ok(String.valueOf( i: a + b)).build();
    }

    @GET
    @Path("/multiply")
    @Produces(MediaType.TEXT_PLAIN)
    public Response multiply(@DefaultValue("1") @QueryParam("a") int a,
                             @DefaultValue("1") @QueryParam("b") int b) {
        return Response.ok(String.valueOf( i: a * b)).build();
    }
}
```

REST

Configuring Jakarta REST Applications

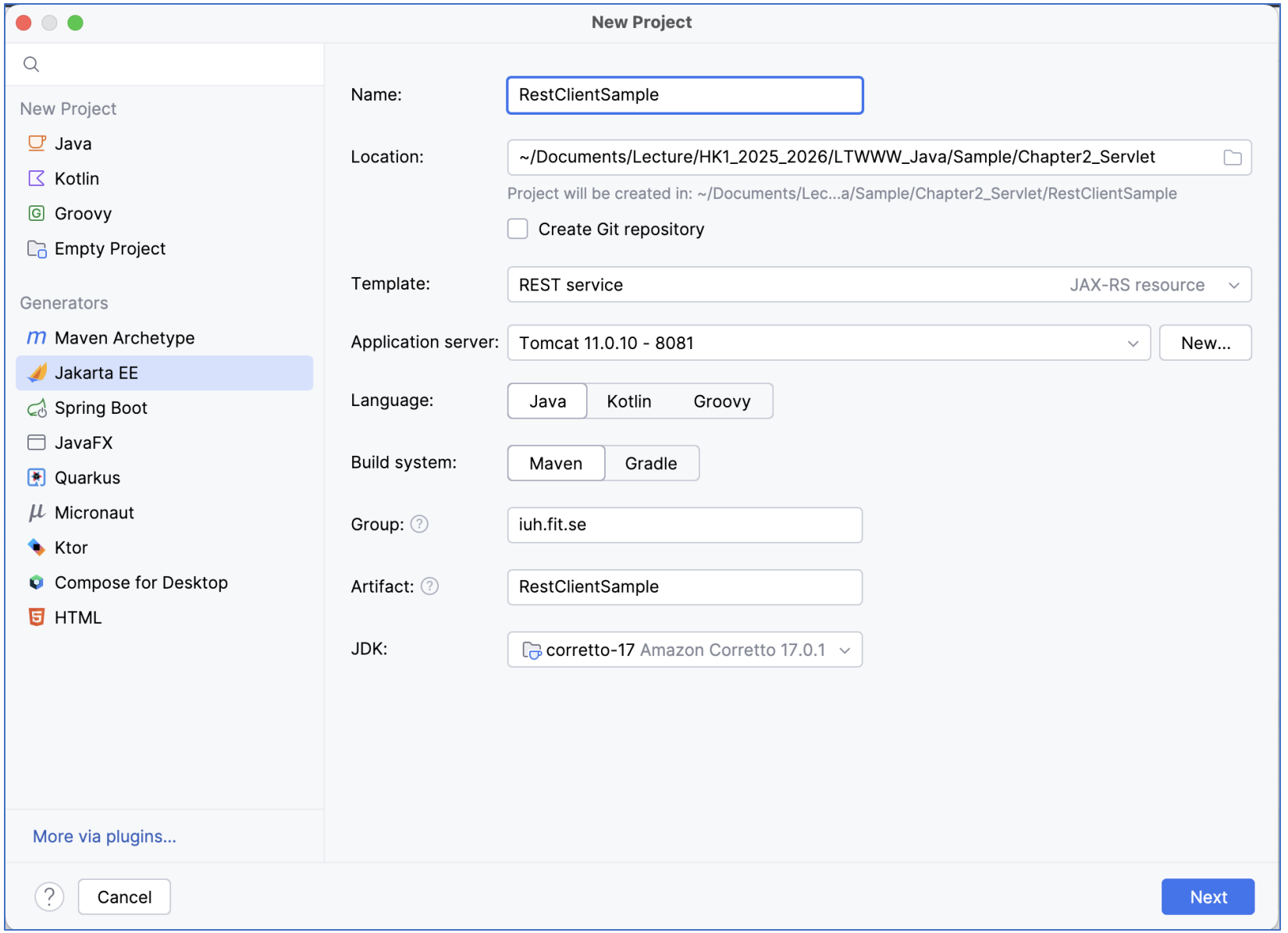
- Create a subclass of jakarta.ws.rs.core.Application to manually configure the environment in which the REST resources defined in your resource classes are run, including the base URI. Add a class-level @ApplicationPath annotation to set the base URI.

```
@ApplicationPath("/api/v1")
public class HelloApplication extends Application {}
```

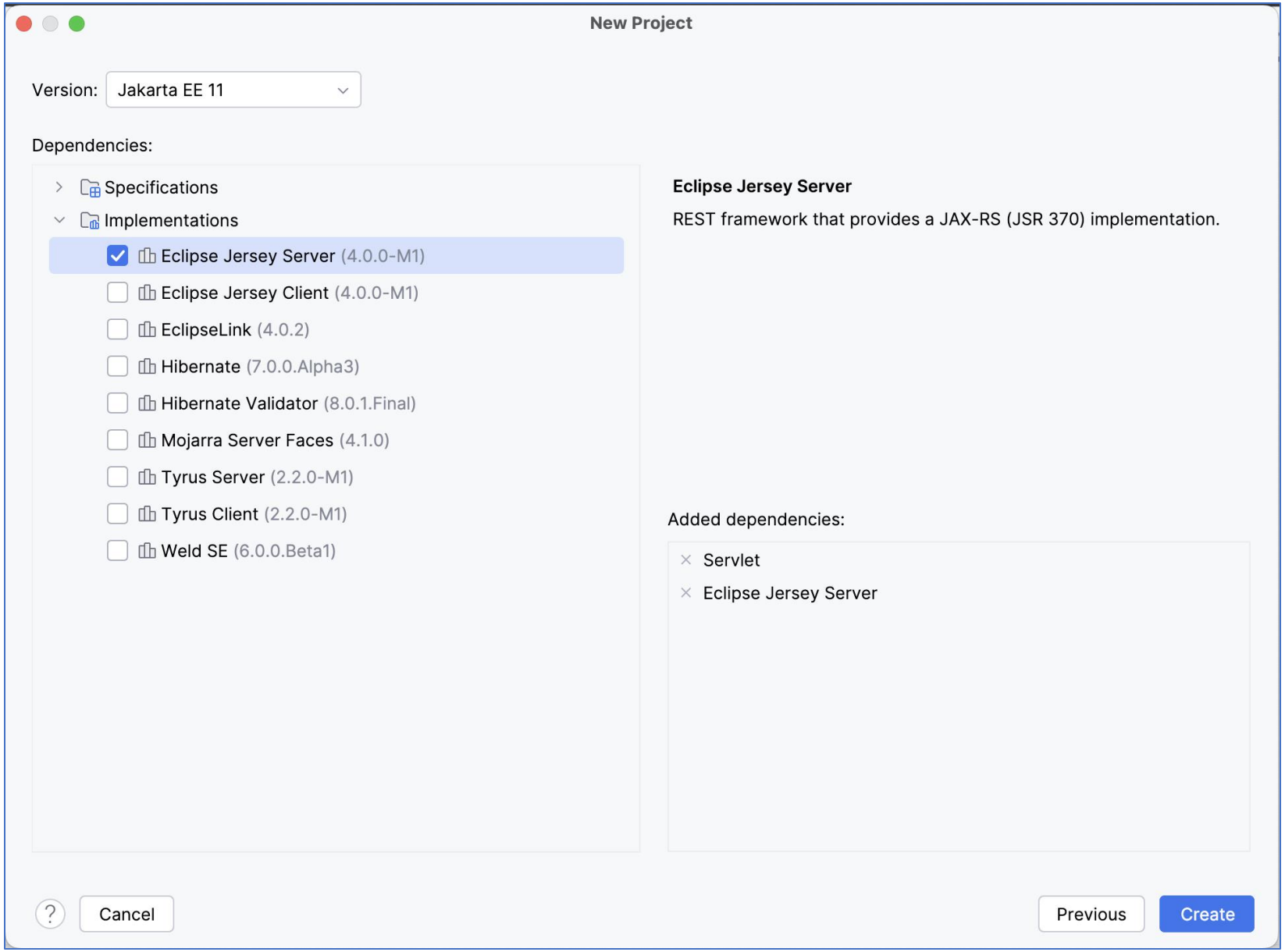
- →all resources defined within the application are relative URI to **/api/v1**
- By default, all the resources in an archive will be processed for resources. Override the getClasses method to manually register the resource classes in the application with the Jakarta REST runtime.

```
@ApplicationPath("/api")
public class HelloApplication extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        final Set<Class<?>> classes = new HashSet<>();
        // register root resource
        classes.add(HelloResource.class);
        return classes;
    }
}
```

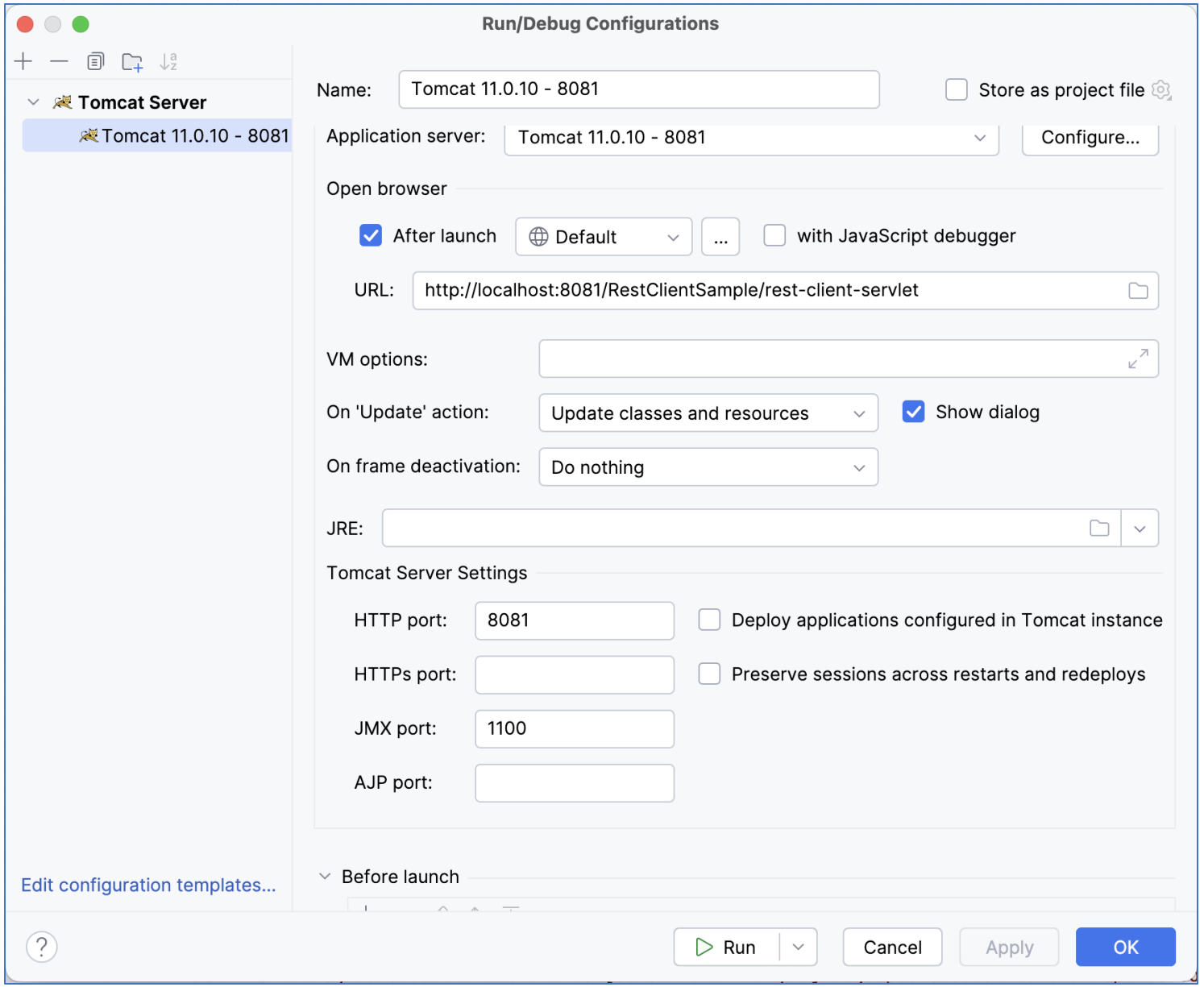
REST Client


New Project
New Project
Java
Kotlin
Groovy
Empty Project
Generators
Maven Archetype
Jakarta EE
Spring Boot
JavaFX
Quarkus
Micronaut
Ktor
Compose for Desktop
HTML
More via plugins...
Name:
RestClientSample
Location:
~/Documents/Lecture/HK1_2025_2026/LTWWW_Java/Sample/Chapter2_Servlet
Project will be created in: ~/Documents/Lec...a/Sample/Chapter2_Servlet/RestClientSample
Create Git repository
Template:
REST service
JAX-RS resource
Application server:
Tomcat 11.0.10 - 8081
New...
Language:
Java
Kotlin
Groovy
Build system:
Maven
Gradle
Group:
iuh.fit.se
Artifact:
RestClientSample
JDK:
corretto-17 Amazon Corretto 17.0.1
Cancel
Next

REST Client (cont.)



REST Client (cont.)



REST Client (cont.)

```java
@WebServlet(name = "restClientServlet", urlPatterns = {"/rest-client-servlet"})
public class RestClientServlet extends HttpServlet {
    @Override  5 usages
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        PrintWriter out = resp.getWriter();
        Client client = ClientBuilder.newClient();

        // WebTarget wt1 = client.target("localhost:8080/RestAPISample/api/numbers/sum/1/2");
        WebTarget wt1 = client.target( uri: "http://localhost:8080/RestAPISample/api/numbers/sum")
                .path("1")
                .path("2");

        Response response1 = wt1.request().accept(MediaType.TEXT_PLAIN).get();
        String s1 = response1.readEntity(String.class);
        out.println("Sum: 1 + 2 = " + s1 + "\n");

        // WebTarget wt2 = client.target("localhost:8080/RestAPISample/api/numbers/multiply?a=1&b=2");
        WebTarget wt2 = client.target( uri: "http://localhost:8080/RestAPISample/api/numbers/multiply")
                .queryParam( name: "a",  ...values: 1)
                .queryParam( name: "b",  ...values: 2);

        Response response2 = wt2.request().accept(MediaType.TEXT_PLAIN).get();
        String s2 = response2.readEntity(String.class);
        out.println("Multiply: 1 x 2 = " + s2);
        out.flush();
        out.close();
    }
}
```

REST Client (cont.)

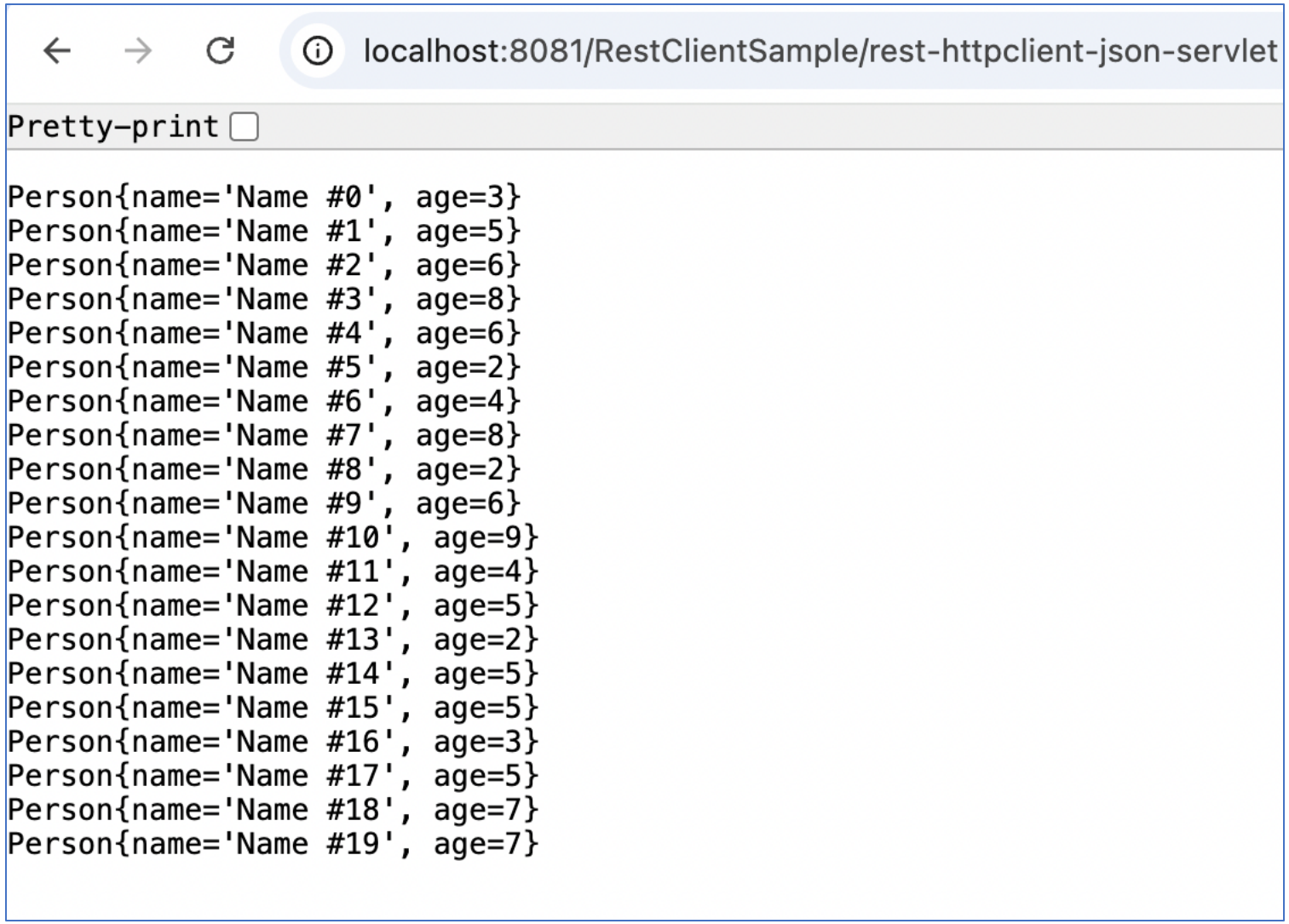
Get a list of Java object

```
@WebServlet(name = "restClientJsonServlet", urlPatterns = {"/rest-client-json-servlet"})
public class RestClientJsonServlet extends HttpServlet {
    @Override  5 usages
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        resp.setContentType("application/json");

        Client client = ClientBuilder.newClient();
        WebTarget wt = client.target( uri: "http://localhost:8080/RestAPISample/api/persons");

        String jsonListString = wt.request(MediaType.APPLICATION_JSON).
                get().readEntity(String.class);
        PrintWriter out = resp.getWriter();

        ObjectMapper objectMapper = new ObjectMapper();
        List<Person> personList = objectMapper.readValue(jsonListString, new TypeReference<List<Person>>() {});
        for(Person person:personList) {
            out.println(person);
        }
        out.flush();
        out.close();
    }
}
```

Rest client using HttpClient

```java
@WebServlet(name = "restClientUsingHttpClientJsonServlet", urlPatterns = {"/rest-httpclient-json-servlet"})
public class RestClientUsingHttpClientJsonServlet extends HttpServlet {
    @Override  5 usages
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        resp.setContentType("application/json");
        PrintWriter writer = resp.getWriter();

        List<Person> personList = null;
        try { personList = getPersons(); }
        catch (URISyntaxException e) { throw new RuntimeException(e); }
        catch (InterruptedException e) { throw new RuntimeException(e);}
        for(Person person:personList) { writer.println(person); }
        writer.flush(); writer.close();
    }

    private List<Person> getPersons() throws IOException, URISyntaxException, InterruptedException {  1 usage
        String url = "http://localhost:8080/RestAPISample/api/persons";

        HttpClient client = HttpClient.newHttpClient();
        HttpRequest request = HttpRequest.newBuilder(new URI(url)).GET().build();
        HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());

        String jsonListString = response.body();
        ObjectMapper objectMapper = new ObjectMapper();
        List<Person> personList = objectMapper.readValue(jsonListString, new TypeReference<List<Person>>() {});

        return personList;
    }
}
```

Rest client using HttpClient (cont.)



WebSocket

WebSocket

Introduction

- In the standard HTTP model, a server cannot initiate a connection with a client nor send an unrequested HTTP response to a client; thus, the server cannot push asynchronous events to clients.
- In a WebSocket application, the server publishes a WebSocket endpoint, and the client uses the endpoint's URI to connect to the server.
  - The WebSocket protocol is symmetrical after the connection has been established; the client and the server can send messages to each other at any time while the connection is open, and they can close the connection at any time.
  - Clients usually connect only to one server, and servers accept connections from multiple clients.
- The WebSocket protocol has two parts: handshake and data transfer.
  - The client initiates the handshake by sending a request to a WebSocket endpoint using its URI.
  - The handshake is compatible with existing HTTP-based infrastructure: web servers interpret it as an HTTP connection upgrade request.

Jakarta WebSocket

Creating WebSocket Applications in the Jakarta EE Platform

- The Jakarta EE platform includes Jakarta WebSocket, which enables you to create, configure, and deploy WebSocket endpoints in web applications. The WebSocket client API specified in Jakarta WebSocket also enables you to access remote WebSocket endpoints from any Java application.
- Jakarta WebSocket consists of the following packages.
  - The jakarta.websocket.server package contains annotations, classes, and interfaces to create and configure server endpoints.
  - The jakarta.websocket package contains annotations, classes, interfaces, and exceptions that are common to client and server endpoints.
- WebSocket endpoints are instances of the jakarta.websocket.Endpoint class.
- Jakarta WebSocket enables you to create two kinds of endpoints: programmatic endpoints and annotated endpoints.
  - To create a programmatic endpoint, you extend the Endpoint class and override its lifecycle methods. (*not-recommended*)
  - To create an annotated endpoint, you decorate a Java class and some of its methods with the annotations provided by the packages mentioned previously.
  - After you have created an endpoint, you deploy it to an specific URI in the application so that remote clients can connect to it.

Jakarta WebSocket

Creating and Deploying a WebSocket Endpoint

- The process for creating and deploying a WebSocket endpoint:

1. Create an endpoint class.
2. Implement the lifecycle methods of the endpoint.
3. Add your business logic to the endpoint.
4. Deploy the endpoint inside a web application.

- The process is slightly different for programmatic endpoints and annotated endpoints, and it is covered in detail in the following sections.

Jakarta WebSocket

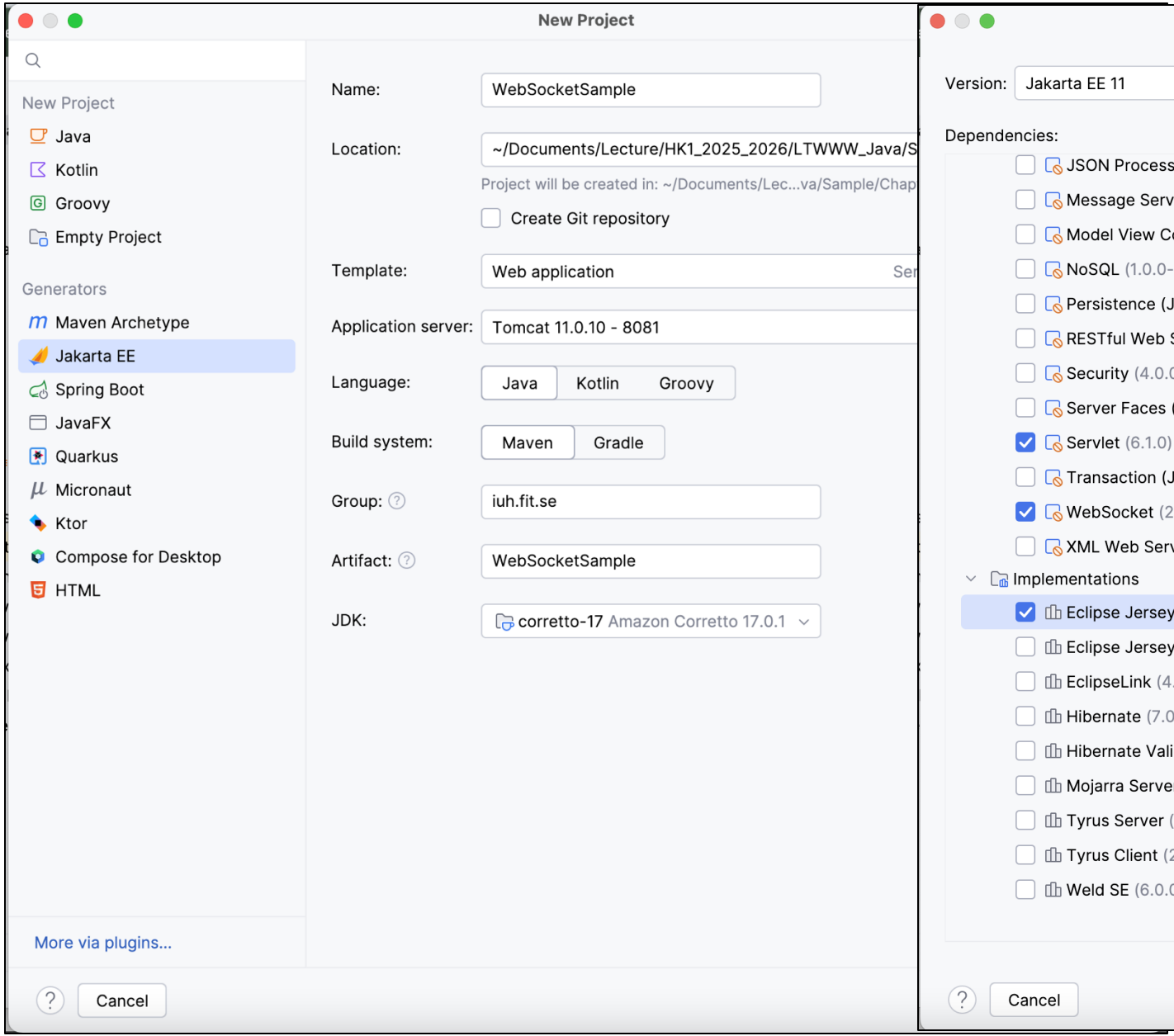
Annotated Endpoints

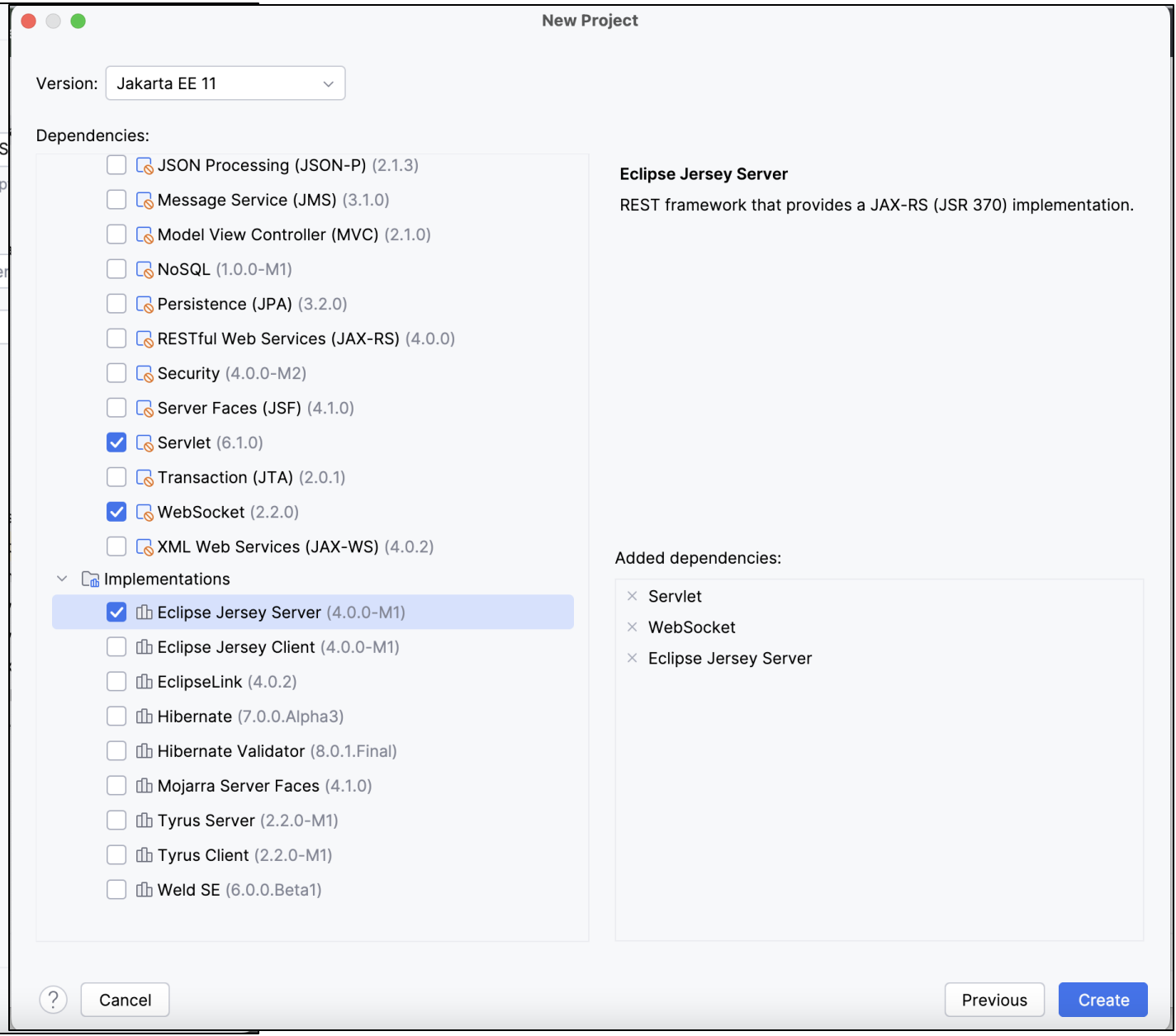
```java
@ServerEndpoint(value = "/sample-endpoint"
        /*,decoders = MessageDecoder.class,
        encoders = MessageEncoder.class*/)
public class SampleEndPoint {
    @OnOpen
    public void onOpen(Session session, String username)throw
        //raise when open a session to endpoint
    }

    @OnMessage
    public void onMessage(Session session, Message message)th
        //raise when any message come
    }
    @OnClose
    public void onClose(Session session) throws Exception{
       //before closed, free resource
    }

    @OnError
    public void onError(Session session, Throwable throwable)
        // Do error handling here
    }
}
```

```java
@ClientEndpoint(
        decoders = MessageDecoder.class,
        encoders = MessageEncoder.class)
public class ChatClientEndPoint {
    private Session session;

    public ChatClientEndPoint() throws Except

    @OnOpen
    public void onOpen(Session session, Endpo

    @OnError
    public void onError(Session session, Thro

    public void sendMessage(Message message)

    @OnClose
    public void onClose(Session session, Clos
}
```

```java
public ChatClientEnpoint(URI endpointURI) {
    try {
        WebSocketContainer container = ContainerProvider.getWebSocketContainer();
        container.connectToServer(this, endpointURI);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Jakarta WebSocket - Demo





Jakarta WebSocket – Demo (cont.)

ServerEndpoint

```
@ServerEndpoint(value = "/chatEndpoint/{username}",  no usages
        encoders = { MessageEncoder.class },
        decoders = { MessageDecoder.class })
public class ChatServerEndpoint {
    ChatSession chatSession = new ChatSession();  4 usages
    private static Session session;  1 usage
    private static Set<Session> chatters = new CopyOnWriteArraySet<>();  3 usages

    @OnOpen  no usages
    public void onOpen(Session session, @PathParam("username") String userName)
            throws IOException, EncodeException {...}

    @OnMessage  no usages
    public void onMessage(Session session, Message message) throws IOException, EncodeException {...}

    @OnClose  no usages
    public void onClose(Session session) {...}

    @OnError  no usages
    public void onError(Session session, Throwable throwable) {...}

    private static void broadcast(Message message) throws IOException, EncodeException {...}
}
```

Jakarta WebSocket – Demo (cont.)

ClientEndpoint

```
@ClientEndpoint(  no usages
        encoders = { MessageEncoder.class },
        decoders = { MessageDecoder.class }
)
public class ChatClientEndpoint {
    private Session session = null;  2 usages
    private MessageHandler handler;  1 usage

    public void ChatClientEndpoint(URI endpointURI) {...}

    @OnOpen  no usages
    public void onOpen(Session session) {...}

    public void addMessageHandler(MessageHandler msgHandler) { this.handler = msgHandler; }

    @OnMessage  no usages
    public void processMessage(String message) { System.out.println("Received message in client: " + message); }

    public void sendMessage(String message) {...}

    public static interface MessageHandler {...}
}
```

Jakarta WebSocket – Demo (cont.)

Client using javascript

ChatServerEndpoint.java | ChatClientEndpoint.java | websocket.js | index.jsp

```jsp
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<html>
    <link href="styles/main.css" rel="stylesheet" type="text/css">
</head>
<body>
<table>
    <tr>
        <td>
            <input type="text" id="userName" placeholder="Username"/>
            <button id="btnConnect" type="button" value="Connect" onclick="connect()">Connect</button>
        </td>
    </tr>
    <tr>
        <td>
            <input type="text" id="message" placeholder="Message"/>
            <button id="btnSendMessage" type="button" value="Send" onclick="sendMessage()">Send</button>
        </td>
    </tr>
    <tr>
        <td>
            <div id="output"></div>
        </td>
    </tr>
</table>
<script type="text/javascript" src="scripts/websocket.js"></script>
</body>
</html>
```

Jakarta WebSocket – Demo (cont.)

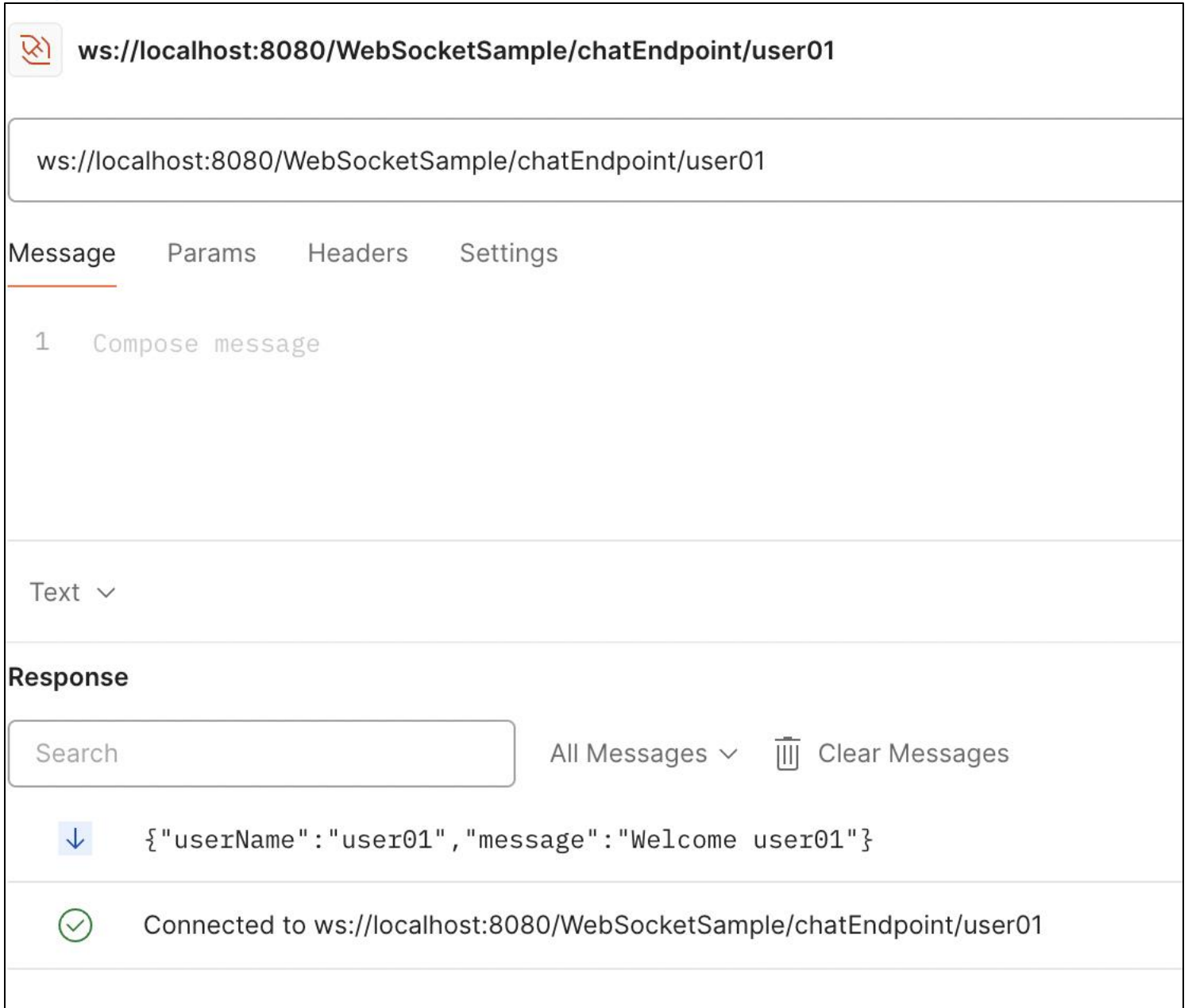
Client using javascript

ChatServerEndpoint.java | ChatClientEndpoint.java | websocket.js

```
let ws;
function connect() : void  {  Show usages
    const userName = document.getElementById( elementId: "userName").value;
    if ("WebSocket" in window) { // Open WebSocket
        ws = new WebSocket( url: "ws://localhost:8080/WebSocketSample/chatEndpoint/" + userName);
        ws.onopen = function() { /*Perform handling when connection is opened */ };
        ws.onmessage = function(evt : MessageEvent ) : void  {
            const json = JSON.parse(evt.data);
            const currentValue : string  = document.getElementById( elementId: 'output').innerHTML;
            document.getElementById( elementId: 'output').innerHTML = currentValue
                + '<br />' + json.userName + ": " + json.message;
        };
        ws.onclose = function() : void  { // websocket is closed.
            alert("Connection is closed...");
        };
    } else { // The browser doesn't support WebSocket
        alert("WebSocket NOT supported by your Browser!");
    }
}

function sendMessage() : boolean  {  Show usages
    const userName = document.getElementById( elementId: 'userName').value;
    const message = document.getElementById( elementId: 'message').value;
    const json : {message: any, userName: any}  = {
        'userName': userName,
        'message': message
    };
    ws.send(JSON.stringify(json));
    return false;
}
```

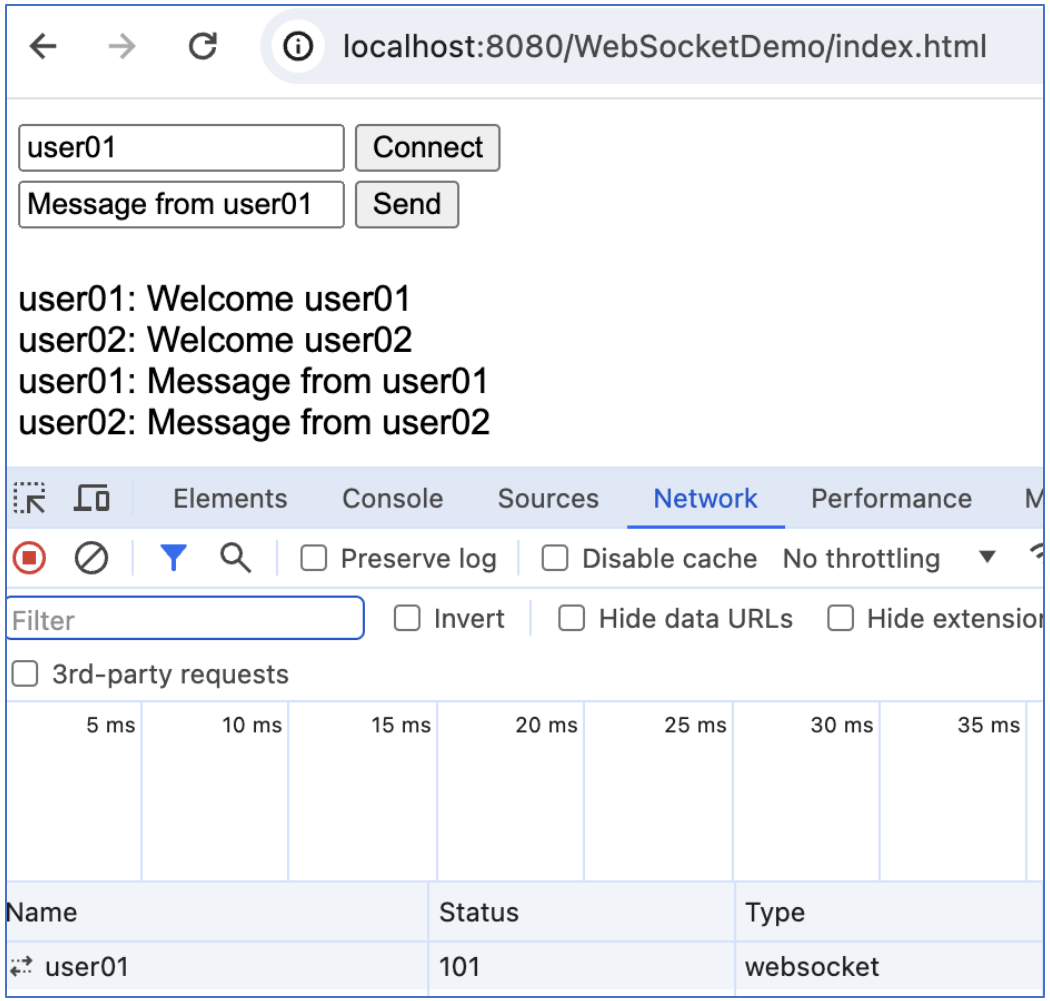
Jakarta WebSocket

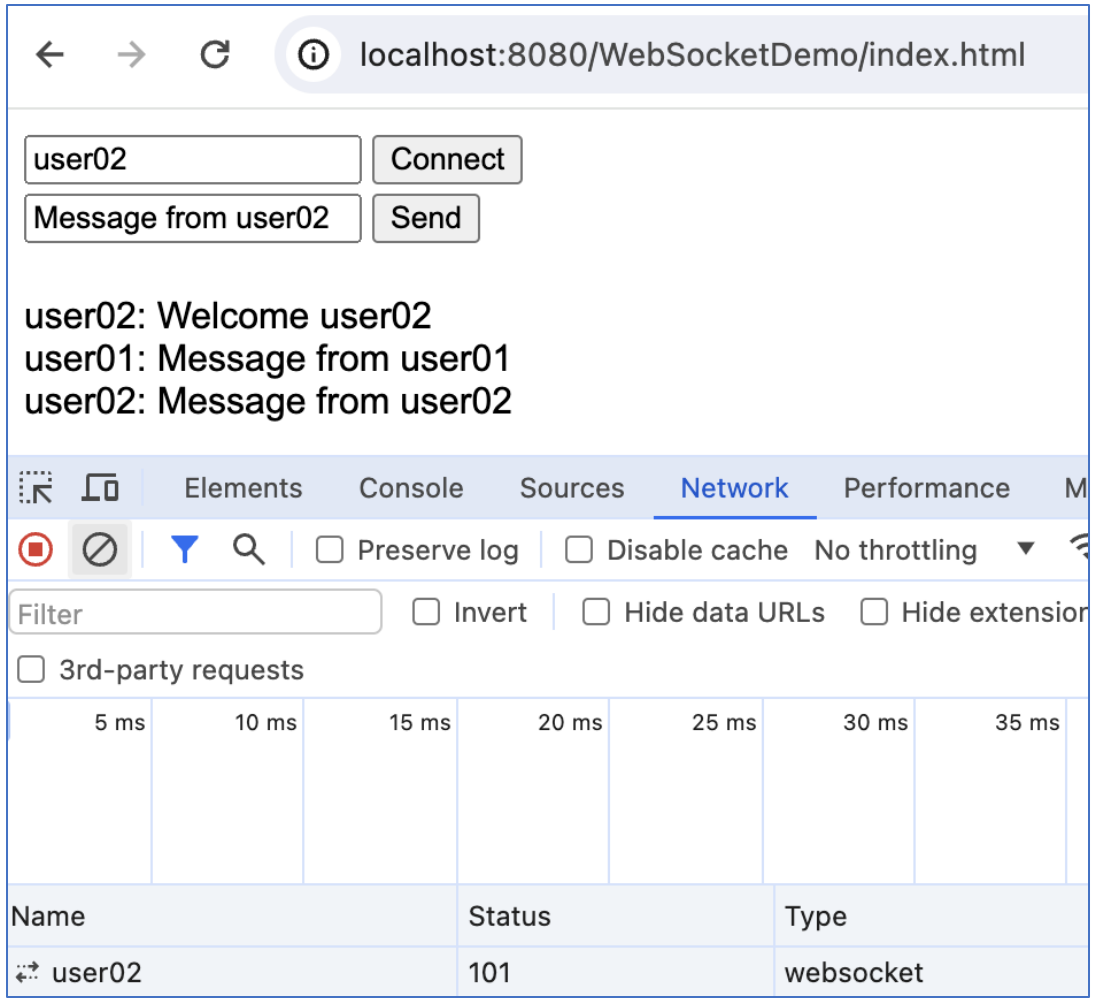
Postman – test Websocket



Jakarta WebSocket (cont.)

Client using javascript





Q&A

Thank you all for your attention and patient !