1

Objectives

- JakartaEE Architecture Overview
- JakartaEE Containers Overview
- JakartaEE APIs Overview
- Jakarta Servlet
  - Servlet Life cycle
  - Sharing Information
  - Asynchronous Processing
  - Contexts and Dependency Injection (CDI)
  - REST API
  - JSP
  - Web Socket
- Java Server Page

2

History

Platform version	Released Date	Specification	Java SE support
Jakarta EE 10	13 September 2022	10	Java SE 17 / 11
Jakarta EE 9.1	25 May 2021	9.1	Java SE 11 / 8
Jakarta EE 9	08 December 2020	9	Java SE 8
Jakarta EE 8	10 September 2019	8	Java SE 8
Java EE 8	31 August 2017	JSR 366	Java SE 8
Java EE 7	28 May 2013	JSR 342	Java SE 7
Java EE 6	10 December 2009	JSR 316	Java SE 6
Java EE 5	11 May 2006	JSR 244	Java SE 5
J2EE 1.4	11 November 2003	JSR 151	J2SE 1.4
J2EE 1.3	24 September 2001	JSR 58	J2SE 1.3
J2EE 1.2	17 December 1997	1.2	J2SE 1.2

3

JavaEE Architecture

Distributed Multitiered Applications

- The Jakarta EE application parts:
  - Client-tier components run on the client machine.
  - Web-tier components run on the Jakarta EE server.
  - Business-tier components run on the Jakarta EE server.
  - Enterprise information system (EIS)-tier software runs on the EIS server.
- Although a Jakarta EE application can consist of all tiers , Jakarta EE multitiered applications are generally considered to be three-tiered applications because they are distributed over three locations: client machines, the Jakarta EE server machine, and the database or legacy machines at the back end.

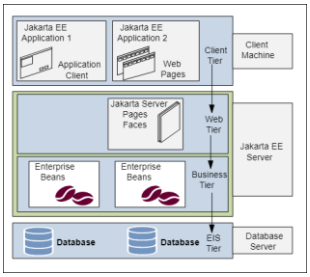


Figure 1-1 Multitiered Applications

4

JavaEE Architecture

Jakarta EE Server Communications

- The various elements that can make up the client tier.
- The client communicates with the business tier running on the Jakarta EE server either directly or, as in the case of a client running in a browser, by going through web pages or servlets running in the web tier.

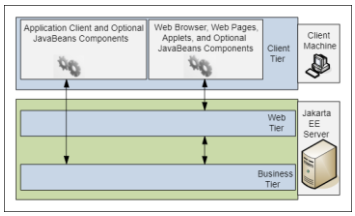


Figure 1-2 Server Communication

5

JavaEE Architecture

Web Components

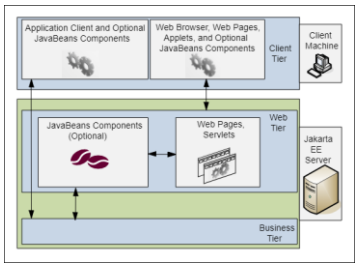


Figure 1-3 Web Tier and Jakarta EE Applications

6

JavaEE Architecture

Business Components

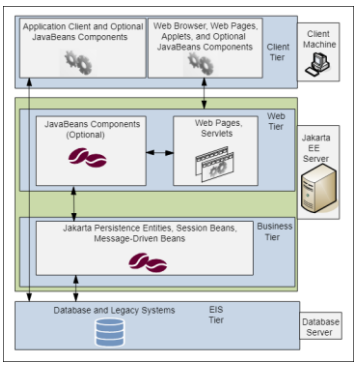


Figure 1-4 Business and EIS Tiers

7

Jakarta EE Containers

Container Services

- Containers are the interface between a component and the low-level, platform-specific functionality that supports the component. Before it can be executed, a web, enterprise bean, or application client component must be assembled into a Jakarta EE module and deployed into its container.
- The assembly process involves specifying container settings for each component in the Jakarta EE application and for the Jakarta EE application itself. Container settings customize the underlying support provided by the Jakarta EE server, including such services as security, transaction management, Java Naming and Directory Interface (JNDI) API lookups, and remote connectivity. Here are some of the highlights.
  - The Jakarta EE security model lets you configure a web component or enterprise bean so that system resources are accessed only by authorized users.
  - The Jakarta EE transaction model lets you specify relationships among methods that make up a single transaction so that all methods in one transaction are treated as a single unit.
  - JNDI lookup services provide a unified interface to multiple naming and directory services in the enterprise so that application components can access these services.
  - The Jakarta EE remote connectivity model manages low-level communications between clients and enterprise beans. After an enterprise bean is created, a client invokes methods on it as if it were in the same virtual machine.

8

Jakarta EE Containers

Container Types

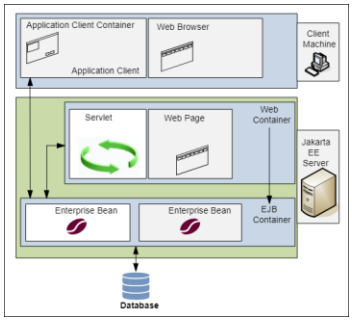


Figure 1-5 Jakarta EE Server and Containers

9

Jakarta EE Containers

Container Types

- The server and containers are as follows:
  - Jakarta EE server: The runtime portion of a Jakarta EE product. A Jakarta EE server provides enterprise and web containers.
  - Jakarta Enterprise Bean container: Manages the execution of enterprise beans for Jakarta EE applications. Jakarta Enterprise Beans and their container run on the Jakarta EE server.
  - Web container: Manages the execution of web pages, servlets, and some enterprise bean components for Jakarta EE applications. Web components and their container run on the Jakarta EE server.
  - Application client container: Manages the execution of application client components. Application clients and their container run on the client.
  - Applet container: Manages the execution of applets. Consists of a web browser and a Java Plug-in running on the client together.

10

Jakarta EE Containers

The relationships among the Jakarta EE containers

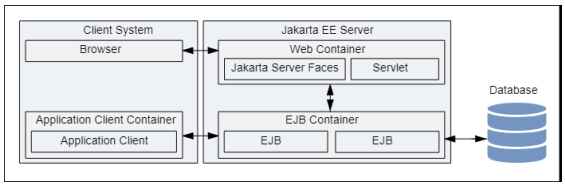


Figure 1-6 The Jakarta EE containers

11

Jakarta EE APIs

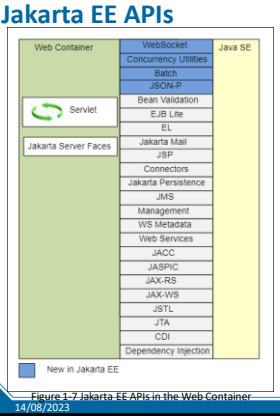


Figure 1-7 Jakarta EE APIs in the Web Container

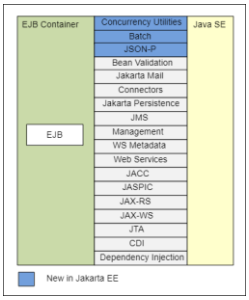


Figure 1-8 Jakarta EE APIs in the enterprise bean Container

12

Jakarta EE APIs

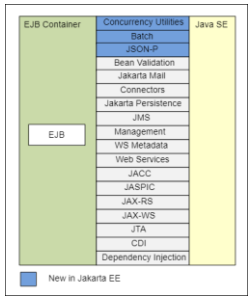


Figure 1-8 Jakarta EE APIs in the enterprise bean Container

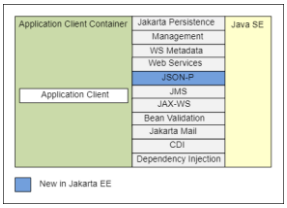


Figure 1-9 Jakarta EE APIs in the Application Client Container

13

Jakarta Enterprise Beans Technologies

Jakarta Enterprise Beans Technologies	
Jakarta Servlet Technology	Jakarta Authorization
Jakarta Faces Technology	Jakarta Authentication
Jakarta Server Pages Technology	Jakarta Security
Jakarta Standard Tag Library	Jakarta WebSocket
Jakarta Persistence	Jakarta JSON Processing
Jakarta Transactions	Jakarta JSON Binding
Jakarta RESTful Web Services	Jakarta Concurrency
Jakarta Managed Beans	Jakarta Batch
Jakarta Contexts and Dependency Injection	Jakarta Activation
Jakarta Dependency Injection	Jakarta XML Binding
Jakarta Bean Validation	Jakarta XML Web Services
Jakarta Messaging	Jakarta SOAP with Attachments
Jakarta Connectors	Jakarta Annotations
Jakarta Mail	

Read more: https://eclipse-ee4j.github.io/jakartaee-tutorial/

14

Jakarta EE API Interoperability

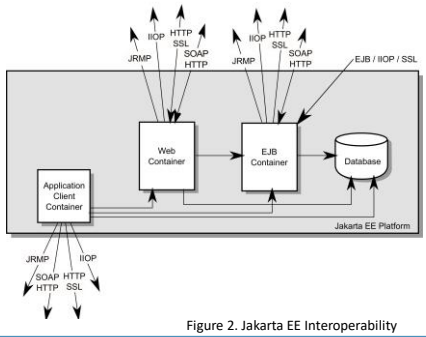
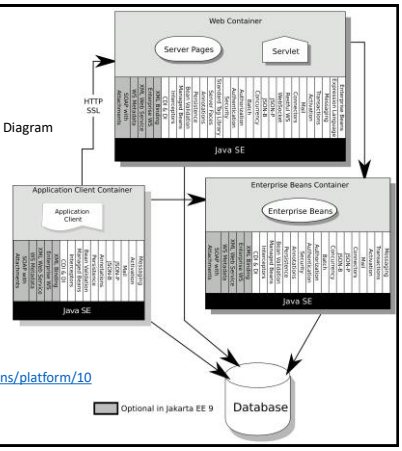


Figure 2. Jakarta EE Interoperability

15

Architecture

Fig1. Jakarta EE Architecture Diagram



https://jakarta.ee/specifications/platform/10/jakarta-platform-spec-10.0

16

Jakarta Servlet 6.0

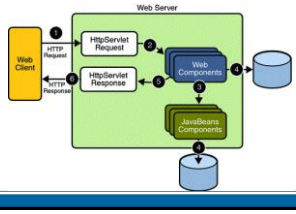
https://jakarta.ee/specifications/servlet/6.0/jakarta-servlet-spec-6.0

17

Terminologies

What is a Servlet?

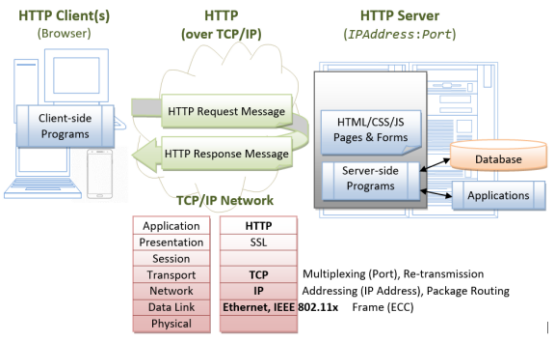
- A servlet is a Jakarta technology-based web component, managed by a container, that generates dynamic content.
- Like other Jakarta technology-based components, servlets are platform-independent Java classes that are compiled to platform-neutral byte code that can be loaded dynamically into and run by a Jakarta technology-enabled web server.
- 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.



18

Servlet Request/Response Handling

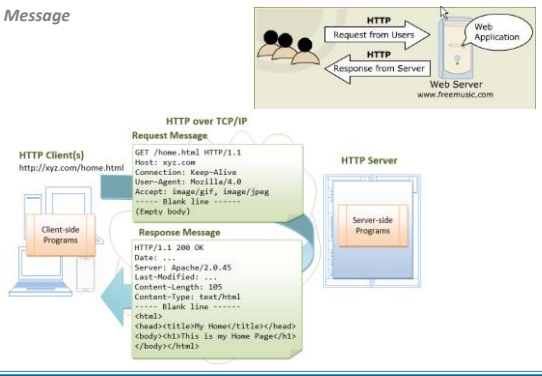
Java Web Application Request Handling



19

Servlet Communication & Protocols

Message



20

Http methods and Servlet methods

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.	

21

Terminologies

What is a Servlet Container?

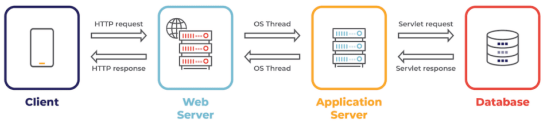
- The servlet container is a part of a web server or application server that provides the network services over which requests and responses are sent, decodes MIME-based requests, and formats MIME-based responses. A servlet container also contains and manages servlets through their lifecycle.
- A servlet container can be built into a host web server or installed as an add-on component to a web server via that server's native extension API. Servlet containers can also be built into or possibly installed into web-enabled application servers.
- All servlet containers must support HTTP as a protocol for requests and responses, but additional request/response-based protocols such as HTTPS (HTTP over SSL) may be supported.

22

Terminologies

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.



23

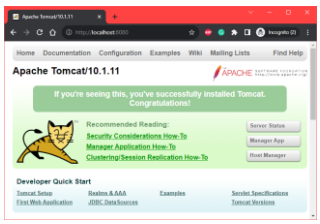
Web Server vs. Application Server

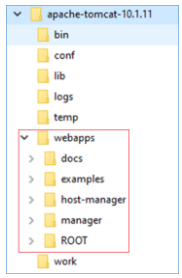
.NO	Web Server	Application Server
1.	Web server encompasses web container only.	While application server encompasses Web container as well as EJB container.
2.	Web server is useful or fitted for static content.	Whereas application server is fitted for dynamic content.
3.	Web server consumes or utilizes 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,....

24

Apache Tomcat Web Server

- Download: https://tomcat.apache.org/download-10.cgi (v 10.1.11 or higher)
- Configuration:
  - Install JDK (8/11/17), add to PATH environment
  - Edit: .\conf\tomcat-user.xml
  - Start Tomcat: .\bin\startup.bat





25

Development

Web Application Directory Structure

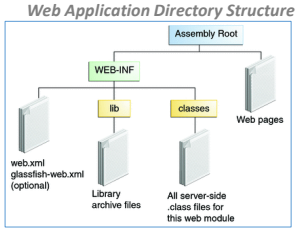
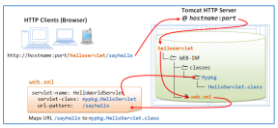


Fig1. Java web-application Structure

1. Create directory structure
2. Put resources into these folders appropriately
3. Copy it to **webapps** folder of Tomcat web server

→ automatically deploy

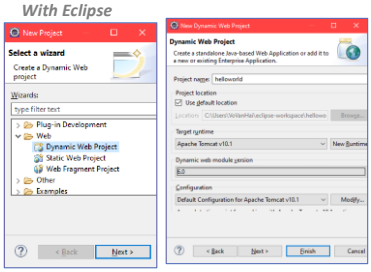
Fig2: Sample application

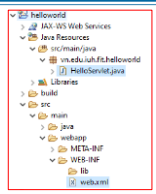


26

Create Application

With Eclipse

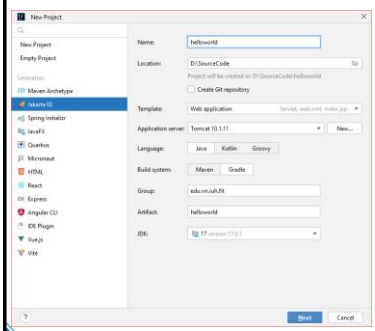


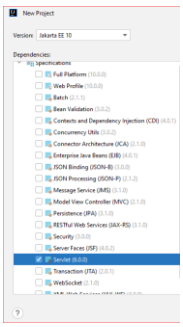


27

Create Application

With IntelliJ IDEA





28

Sample Servlet

HelloServlet.java

```
package edu.vn.iuh.fit.helloworld;

import java.io.*;
import jakarta.servlet.http.*;
import jakarta.servlet.annotation.*;

@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 IO
        response.setContentType("text/html");

        // Hello
        PrintWriter out = response.getWriter();
        out.println("<html><body>");
        out.println("<h1>" + message + "</h1>");
        out.println("</body></html>");
    }

    public void destroy() {
    }
}
```

helloworld

```
.gradle
.idea
build
gradle
src
  main
    java
      edu.vn.iuh.fit.helloworld
        HelloServlet
    resources
    webapp
      WEB-INF
      index.jsp
  test
```

index.jsp

```
<%@ page contentType="text/html; charset=
<!DOCTYPE html>
<html>
<head>
    <title>JSP - Hello World</title>
</head>
<body>
<h1><%= "Hello World!" %>
</h1>
<br/>
<a href="hello-servlet">Hello Servlet</a>
</body>
</html>
```

14/08/2023 29

29

Create Project with Build tools

Maven:

```
<dependency>
<groupId>jakarta.servlet</groupId>
<artifactId>jakarta.servlet-
api</artifactId>
<version>6.0.0</version>
</dependency>
```

Maven

Gradle:
compileOnly('jakarta.servlet:jakarta.servlet-api:6.0.0')

Gradle

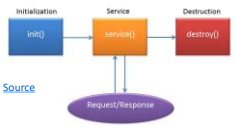
14/08/2023 30

30

Jakarta Servlet Life Cycle

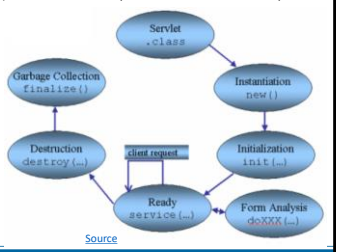
Generic Servlet Life Cycle

- **The life cycle is defined by:**
  - init() – called only one by the server in the first request
  - service() – process the client's request
  - destroy() – called after all requests have been processed or a server-specific number of seconds have passed



Source

Fig: The Servlet Life Cycle



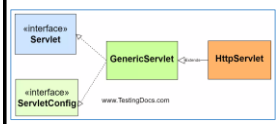
Source

14/08/2023 31

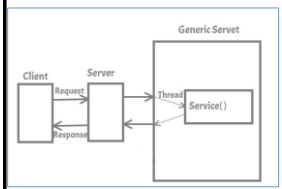
31

Types of Servlet

The jakarta.servlet.GenericServle




```
<servlet>
    <servlet-name>myGenericServlet</servlet-name>
    <servlet-class>edu.vn.iuh.fit.helloworld.MyGenericServlet</servlet-class
    <init-param>
        <param-name>Greetings</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>Vo Van Hai</param-value>
</context-param>
```




```
//@WebServlet(name = "myGenericServlet", value = "/myGenericServlet")
public class MyGenericServlet extends GenericServlet {
    private String initParam;

    @Override
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        initParam = config.getInitParameter( name: "Greetings");
    }

    @Override
    public void service(ServletRequest request, ServletResponse response) throw
        ServletContext ctx = this.getServletContext();
        String name = ctx.getInitParameter( name: "name");
        response.getWriter().println(initParam + " GenericServlet by " + name);
    }
}
```

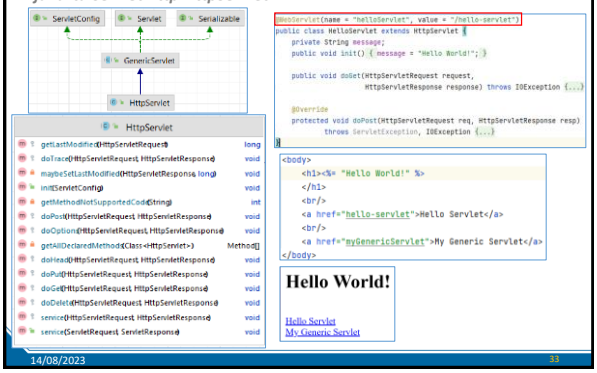
Hello GenericServlet by Vo Van Hai

14/08/2023 32

32

Types of Servlet

jakarta.servlet.http.HttpServlet



33

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

34

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 Objects		
Scope Object	Class	Accessible From
Web context	jakarta.servlet.ServletContext	Web components within a web context. See [accessing-the-web-context].
Session	jakarta.servlet.http.HttpSession	Web components handling a request that belongs to the session. See [maintaining-client-state].
Request	Subtype of jakarta.servlet.ServletRequest	Web components handling the request.
Page	jakarta.servlet.jsp.JspContext	The Jakarta Server Pages page that creates the object.

35

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.

36

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.

37

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 {
   //...
}
```


38

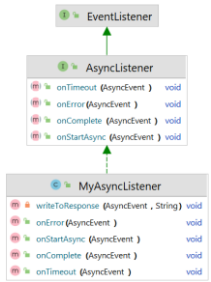
Asynchronous Processing

Example

```
@WebServlet(name = "myAsyncServlet", urlPatterns = {"/async-test"},
            asyncSupported = true)
public class MyAsyncServlet extends HttpServlet {
    @Override
    protected void doGet (HttpServletRequest req,
                          HttpServletResponse resp) throws ServletE

        PrintWriter writer = resp.getWriter();
        AsyncContext asyncContext = req.startAsync();
        asyncContext.addListener(new MyAsyncListener());

        asyncContext.start(new Runnable() {
            @Override
            public void run () {
                String msg = task();
                writer.println(msg);
                asyncContext.complete();
            }
        });
    }
    private String task () {...}
}
```



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

39

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

```
dependencies {
   compileOnly('jakarta.servlet:jakarta.servlet-api:6.0.0')
   implementation 'org.jboss.weld.servlet:weld-servlet-core:5.1.1.Final'
}
```

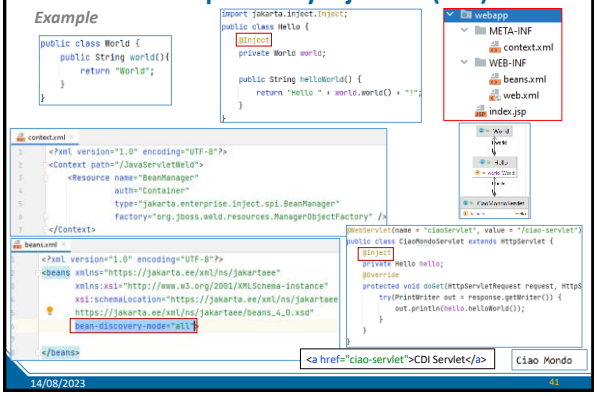
```
<dependency>
    <groupId>org.jboss.weld.servlet</groupId>
    <artifactId>weld-servlet-core</artifactId>
    <version>5.1.1.Final</version>
</dependency>
```

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

40

Contexts and Dependency Injection (CDI)

Example



41

REST API

42

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.

43

RESTful Root Resource

Developing RESTful Web Services with Jakarta REST

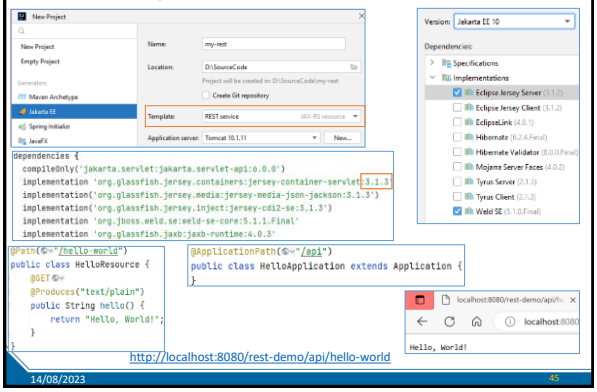
- Jakarta REST is a Java programming language API designed to make it easy to develop applications that use the REST architecture.
- The Jakarta REST API uses Java programming language annotations to simplify the development of RESTful web services.
- Developers decorate Java programming language class 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

44

REST - example



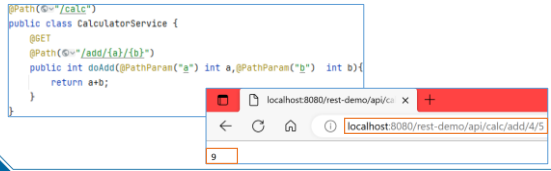
http://localhost:8080/rest-demo/api/hello-world

45

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 }).



46

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

47

REST

Using @Consumes and @Produces to Customize Requests and Responses

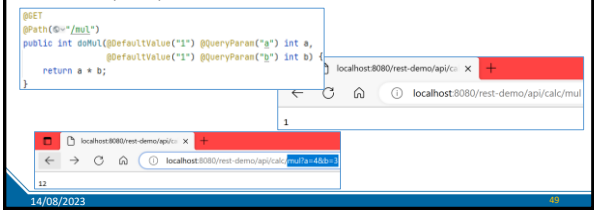
- The value of @Produces is an array of String of MIME types or a comma-separated list of MediaType constants. For example:
  - @Produces({"image/jpeg,image/png"})
  - @Produces(MediaType.APPLICATION_XML)
  - @Produces({"application/xml", "application/json"})
- The @Consumes annotation is used to specify 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")

48

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



49

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")
public class HelloApplication extends Application {}
```

- →all resources defined within the application are relative to /api
- 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;
    }
}
```


50

REST Client

```
public static void main(String[] args) {
    Client client = ClientBuilder.newClient();

//      WebTarget wt1 = client.target("http://localhost:8080/rest-demo/api/calc/mul?a=4&b=3");
    WebTarget wt1 = client.target( uri: "http://localhost:8080/rest-demo/api/calc/mul")
            .queryParam( name: "a", ...values: 4)
            .queryParam( name: "b", ...values: 3);
    Response response1 = wt1.request().accept(MediaType.TEXT_PLAIN).get();
    String s1 = response1.readEntity(String.class);

//      WebTarget wt2 = client.target("http://localhost:8080/rest-demo/api/calc/add/3/4");
    WebTarget wt2 = client.target( uri: "http://localhost:8080/rest-demo/api/calc/add")
            .path("3")
            .path("4");
    Response response2 = wt2.request().accept(MediaType.TEXT_PLAIN).get();
    String s2 = response2.readEntity(String.class);

    System.out.println("--------:" + s1);
    System.out.println("--------:" + s2);
  }
}
```


51

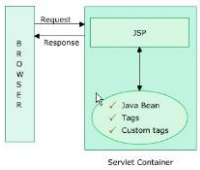
Java Server Pages

52

Java Server Pages

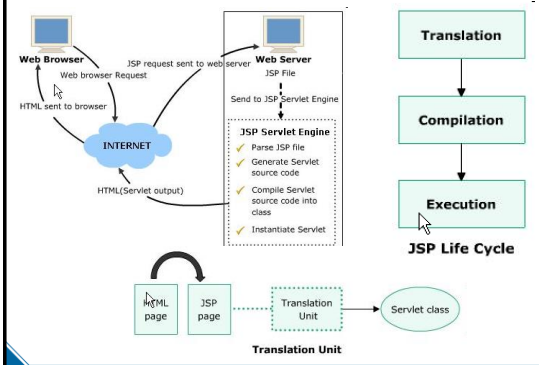
Introduction

- Java Server Page (JSP) is a server side script language
- Saved with .jsp extension
- A simple, yet powerful Java technology for creating and maintaining dynamic-content webs pages
- JSP page are converted by the web container into a Servlet instance
- It focus on the presentation logic of the web application



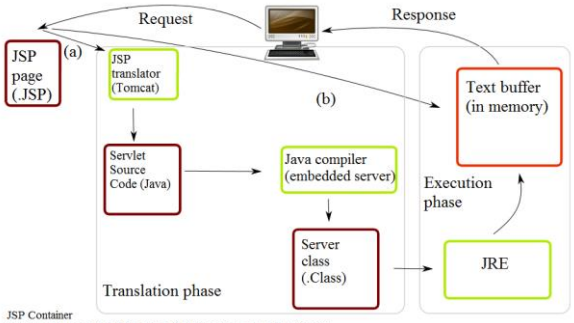
53

JSP Architecture



54

JSP Execution



55

Self-Study

Read more

https://github.com/eclipse-ee4j/jakartaee-tutorial

56

Thanks for your Listening

57