Web Applications Developments

ITWT413

LECTURE 8: JAKARTA EE WEB PROFILE

DR. HALA SHAARI

Jakarta EE Tutorial- Structure

Basic Platform	Jakarta EE Core Profile	Jakarta EE Web Profile	Jakarta EE Platform
Resource Creation	Jakarta CDI Lite	Jakarta CDI Full	Jakarta Mail
Injection	Jakarta REST	Jakarta Validation	Jakarta Messaging
Packaging	Jakarta JSON	Jakarta Security	Jakarta Batch
		Jakarta Servlets	
		Jakarta Faces	
		Jakarta WebSocket	
		Jakarta Persistence	
		Jakarta Enterprise Beans Lite	

Jakarta EE 10

Lecture Agenda

- Getting Started with Web Applications
- Jakarta Servlet
- Servlet Lifecycle
- Creating and Initializing a Servlet

```java
import jakarta.inject.Qualifier;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER, ElementType.TYPE})
public @interface GreetingType {
    String value();
}
```

CasualGreetingService.java

```java
@GreetingType("casual")
public class CasualGreetingService implements GreetingService {
    @Override
    public String greet(String name) {
        return "Hey " + name + ", what's up?";
    }
}
```

FormalGreetingService.java

```java
@GreetingType("formal")
public class FormalGreetingService implements GreetingService {
    @Override
    public String greet(String name) {
        return "Good day, " + name + ". How do you do?";
    }
}
```

```java
public interface GreetingService {
    String greet(String name);
}
```

GreetingController.java

```java
import jakarta.inject.Inject;

public class GreetingController {

    @Inject
    @GreetingType("casual")
    private GreetingService casualGreetingService;

    @Inject
    @GreetingType("formal")
    private GreetingService formalGreetingService;

    public void sendGreetings(String name) {
        System.out.println(casualGreetingService.greet(name));  // Casual Greeting
        System.out.println(formalGreetingService.greet(name)); // Formal Greeting
    }
}
```

What is the output when the sendGreetings method is called with John as the input?

Output

When the `sendGreetings` method is called with `John` as the input:

rust

Copy code

```
Hey John, what's up?
Good day, John. How do you do?
```

Getting Started with Web Applications

- Web Applications:

  A web application, also known as a web app, is a software application that runs on one or more web servers. It is typically accessed through a web browser over a network, such as the Internet. The advantage over desktop and mobile applications is being platform-independent, as it can be accessed and used on different devices.

Web applications types

- Presentation-oriented

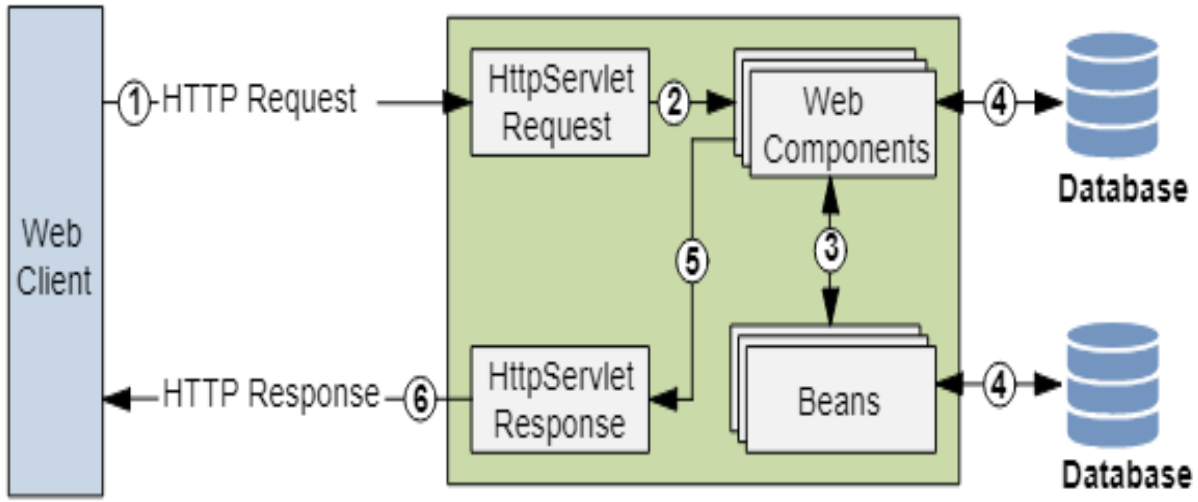
A presentation-oriented web application (also called a "website") generates dynamic web pages in response to HTTP requests. The response is usually represented as a HTML document along with assets, such as Cascading Style Sheets (CSS), JavaScript (JS) and images. Presentation-oriented applications are often directly used by humans.

- Service-oriented

A service-oriented web application (also called a "service") generates dynamic data structures in response to HTTP requests. The response is usually represented as a JSON object or as a XML document or even as plain text. Service-oriented applications are often directly used by presentation-oriented web applications or other service-oriented applications.

Web applications In the Jakarta EE platform

- In the Jakarta EE platform, web applications are represented by web components. A web component can be represented by Jakarta Servlet, Jakarta Faces or Jakarta REST.
- Jakarta Servlet can be used to build both presentation and service-oriented web applications.
- Jakarta Faces is a component-based MVC framework that can be executed on a servlet container and act as a presentation-oriented web application.



The Web Application Archive

A Jakarta Servlet-based web application can contain one or more of the following parts:

- One or more web components, which can be represented by Jakarta Servlet, Jakarta Faces or Jakarta REST.
- Assets (also called static resource files), such as Cascading Style Sheets (CSS), JavaScript (JS) and images.
- Dependencies (also called helper libraries, third party libraries, "JARs").
- Deployment descriptor files.

Build a Jakarta Servlet-based web application

The process for creating, deploying, and executing a Jakarta Servlet-based web application is different from that of Java classes which are packaged and executed as a Java application archive (JAR). It can be summarized as follows:

1. Develop the web component code.
2. Develop the deployment descriptor files, if necessary.
3. Compile the web component code against the libraries of the servlet container and the helper libraries, if any.
4. Package the compiled code along with helper libraries, assets and deployment descriptor files, if any, into a deployable unit, called a web application archive (WAR).
5. Deploy the WAR into a servlet container.
6. Run the web application by accessing a URL that references the web component.

Jakarta Servlet

- Jakarta Servlet is a corner stone web framework that can act as a presentation-oriented as well as a service-oriented web application. Jakarta Servlet intends to reduce the boilerplate code needed to convert the HTTP request into a Java object and to offer a Java object as an HTTP response, and to manage all the lifecycle around them.
- What Is a Servlet?
  - A servlet is a Java programming language class that directly or indirectly implements the jakarta.servlet.Servlet interface.
  - The jakarta.servlet and jakarta.servlet.http packages provide interfaces and classes for writing servlets.
  - All servlets must implement the jakarta.servlet.Servlet interface, which defines lifecycle methods such as init, service, and destroy.
  - When implementing a generic service, you can extend the jakarta.servlet.GenericServlet class which already implements the Servlet interface.
  - When implementing an HTTP service, you can extend the jakarta.servlet.http.HttpServlet class which already extends the GenericServlet class.
  - In a typical Jakarta Servlet based web application, the class must extend jakarta.servlet.http.HttpServlet and override one of the doXxx methods where Xxx represents the HTTP method of interest.

Servlet Lifecycle

- The lifecycle of a servlet is controlled by the servlet container in which the servlet has been deployed. When a request is mapped to a servlet, the servlet container performs the following steps:

1. If an instance of the servlet does not exist, the servlet container:
   - Loads the servlet class
   - Creates an instance of the servlet class
   - Initializes the servlet instance by calling the init method
2. The servlet container invokes the service method, passing request and response objects.

- If it needs to remove the servlet, the servlet container finalizes the servlet by calling the servlet's destroy method.

Sharing Information

- Web components, like most objects, usually work with data to accomplish their tasks. Web components can store this information in a data store or in a scoped bean, among others.

1. Using CDI Managed Beans.
2. Using Scope Objects Directly.
3. Controlling Concurrent Access to Shared Resources

Creating and Initializing a Servlet

- Use the @WebServlet annotation to define a servlet component in a web application.
- This annotation is specified on a class and contains metadata about the servlet being declared.
- The annotated servlet must specify at least one URL pattern. This is done by using the urlPatterns or value attribute on the annotation. All other attributes are optional, with default settings.
- Use the value attribute when the only attribute on the annotation is the URL pattern; otherwise, use the urlPatterns attribute when other attributes are also used.
- Classes annotated with @WebServlet must

extend the jakarta.servlet.http.HttpServlet class.

```
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;

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

Writing Service Methods

- The service provided by a servlet is implemented in the service method of a GenericServlet, in the doMethod methods (where Method can take the value Get, Delete, Options, Post, Put, or Trace) of an HttpServlet object, or in any other protocol-specific methods defined by a class that implements the Servlet interface.
- The term service method is used for any method in a servlet class that provides a service to a client.
- The general pattern for a service method is to extract information from the request, access external resources, and then populate the response, based on that information.
- For HTTP servlets, the correct procedure for populating the response is to do the following:

1. Retrieve an output stream from the response.
2. Fill in the response headers.
3. Write any body content to the output stream.

Getting Information from Requests

- A request contains data passed between a client and the servlet.
- All requests implement the ServletRequest interface. This interface defines methods for accessing the following information:
  - Parameters, which are typically used to convey information between clients and servlets
  - Object-valued attributes, which are typically used to pass information between the web container and a servlet or between collaborating servlets
  - Information about the protocol used to communicate the request and about the client and server involved in the request
  - Information relevant to localization

```
http://[host]:[port][request-path]?[query-string]
```

Constructing Responses

- A response contains data passed between a server and the client. All responses implement the ServletResponse interface.
- This interface defines methods that allow you to do the following.
  - Retrieve an output stream to use to send data to the client. To send character data, use the PrintWriter returned by the response's getWriter method. To send binary data in a Multipurpose Internet Mail Extensions (MIME) body response, use the ServletOutputStream returned by getOutputStream. To mix binary and text data, as in a multipart response, use a ServletOutputStream and manage the character sections manually.
  - Indicate the content type (for example, text/html) being returned by the response with the setContentType(String) method. This method must be called before the response is committed.
  - Indicate whether to buffer output with the setBufferSize(int) method. By default, any content written to the output stream is immediately sent to the client. Buffering allows content to be written before anything is sent back to the client, thus providing the servlet with more time to set appropriate status codes and headers or forward to another web resource. The method must be called before any content is written or before the response is committed.
  - Set localization information, such as locale and character encoding.

Handling Servlet Lifecycle Events (1)

- You can monitor and react to events in a servlet's lifecycle by defining listener objects whose methods get invoked when lifecycle events occur. To use these listener objects, you must define and specify the listener class.
- Defining the Listener Class: You define a listener class as an implementation of a listener interface

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`

Handling Servlet Lifecycle Events (2)

- Use the @WebListener annotation to define a listener to get events for various operations on the particular web application context.
- Classes annotated with @WebListener must implement one of the following interfaces:

```
jakarta.servlet.ServletContextListener
jakarta.servlet.ServletContextAttributeListener
jakarta.servlet.ServletRequestListener
jakarta.servlet.ServletRequestAttributeListener
jakarta.servlet..http.HttpSessionListener
jakarta.servlet..http.HttpSessionAttributeListener
```

```
import jakarta.servlet.ServletContextAttributeListener;
import jakarta.servlet.ServletContextListener;
import jakarta.servlet.annotation.WebListener;

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

Handling Servlet Errors

- Any number of exceptions can occur when a servlet executes. When an exception occurs, the web container generates a default page containing the following message:

```
A Servlet Exception Has Occurred
```

- But you can also specify that the container should return a specific error page for a given exception.

Filtering Requests and Responses

- A filter is an object that can transform the header and content (or both) of a request or response. Filters differ from web components in that filters usually do not themselves create a response. Instead, a filter provides functionality that can be "attached" to any kind of web resource. Consequently, a filter should not have any dependencies on a web resource for which it is acting as a filter; this way, it can be composed with more than one type of web resource.
- The main tasks that a filter can perform are as follows.
  - Query the request and act accordingly.
  - Block the request-and-response pair from passing any further.
  - Modify the request headers and data. You do this by providing a customized version of the request.
  - Modify the response headers and data. You do this by providing a customized version of the response.
  - Interact with external resources.
- Applications of filters include authentication, logging, image conversion, data compression, encryption, tokenizing streams, XML transformations, and so on.
- You can configure a web resource to be filtered by a chain of zero, one, or more filters in a specific order. This chain is specified when the web application containing the component is deployed and is instantiated when a web container loads the component.

Programming Filters

- The filtering API is defined by the Filter, FilterChain, and FilterConfig interfaces in the jakarta.servlet package. You define a filter by implementing the Filter interface.
- Use the @WebFilter annotation to define a filter in a web application. This annotation is specified on a class and contains metadata about the filter being declared. The annotated filter must specify at least one URL pattern. This is done by using the urlPatterns or value attribute on the annotation.
- Classes annotated with the @WebFilter annotation must implement the jakarta.servlet.Filter interface.
- To add configuration data to the filter, specify the initParams attribute of the @WebFilter annotation. The initParams attribute contains a @WebInitParam annotation.

```
@WebFilter(filterName = "TimeOfDayFilter", urlPatterns = {"/*"},
initParams = {@WebInitParam(name = "mood", value = "awake")})
public class TimeOfDayFilter implements Filter {
    ...
}
```

doFilter Method

- The most important method in the Filter interface is doFilter, which is passed request, response, and filter chain objects. This method can perform the following actions.
  - Examine the request headers.
  - Customize the request object if the filter wishes to modify request headers or data.
  - Customize the response object if the filter wishes to modify response headers or data.
  - Invoke the next entity in the filter chain. If the current filter is the last filter in the chain that ends with the target web component or static resource, the next entity is the resource at the end of the chain; otherwise, it is the next filter that was configured in the WAR. The filter invokes the next entity by calling the doFilter method on the chain object, passing in the request and response it was called with or the wrapped versions it may have created. Alternatively, the filter can choose to block the request by not making the call to invoke the next entity. In the latter case, the filter is responsible for filling out the response.
  - Examine response headers after invoking the next filter in the chain.
  - Throw an exception to indicate an error in processing.

Invoking Other Web Resources

- Web components can invoke other web resources both indirectly and directly.
- A web component indirectly invokes another web resource by embedding a URL that points to another web component in content returned to a client.
- While it is executing, a web component directly invokes another resource by either including the content of another resource or forwarding a request to another resource.
- To invoke a resource available on the server that is running a web component, you must first obtain a RequestDispatcher object by using the getRequestDispatcher("URL") method.
- You can get a RequestDispatcher object from either a request or the web context; however, the two methods have slightly different behavior. The method takes the path to the requested resource as an argument.
- A request can take a relative path (that is, one that does not begin with a /), but the web context requires an absolute path. If the resource is not available or if the server has not implemented a RequestDispatcher object for that type of resource, getRequestDispatcher will return null. Your servlet should be prepared to deal with this condition.

Including Other Resources in the Response

- It is often useful to include another web resource, such as banner content or copyright information, in the response returned from a web component.
- To include another resource, invoke the include method of a RequestDispatcher object:

```
include(request, response);
```

- If the resource is static, the include method enables programmatic server-side includes.
- If the resource is a web component, the effect of the method is to send the request to the included web component, execute the web component, and then include the result of the execution in the response from the containing servlet.
- An included web component has access to the request object but is limited in what it can do with the response object.
  - It can write to the body of the response and commit a response.
  - It cannot set headers or call any method, such as setCookie, that affects the headers of the response.

Transferring Control to Another Web Component

- In some applications, you might want to have one web component do preliminary processing of a request and have another component generate the response. For example, you might want to partially process a request and then transfer to another component, depending on the nature of the request.
- To transfer control to another web component, you invoke the forward method of a RequestDispatcher. When a request is forwarded, the request URL is set to the path of the forwarded page.
- The original URI and its constituent parts are saved as the following request attributes:
  - jakarta.servlet.forward.request_uri
  - jakarta.servlet.forward.context_path
  - jakarta.servlet.forward.servlet_path
  - jakarta.servlet.forward.path_info
  - jakarta.servlet.forward.query_string

Accessing the Web Context

- The context in which web components execute is an object that implements the ServletContext interface. You retrieve the web context by using the getServletContext method. The web context provides methods for accessing
  - Initialization parameters
  - Resources associated with the web context
  - Object-valued attributes
  - Logging capabilities
- The counter's access methods are synchronized to prevent incompatible operations by servlets that are running concurrently. A filter retrieves the counter object by using the context's getAttribute method. The incremented value of the counter is recorded in the log.

Maintaining Client State

- Many applications require that a series of requests from a client be associated with one another. For example, a web application can save the state of a user's shopping cart across requests.
- Web-based applications are responsible for maintaining such state, called a session, because HTTP is stateless. To support applications that need to maintain state, Jakarta Servlet technology provides an API for managing sessions and allows several mechanisms for implementing sessions.
  - Accessing a Session
  - Associating Objects with a Session
  - Session Management
  - Session Tracking

Finalizing a Servlet

- The web container may determine that a servlet should be removed from service (for example, when a container wants to reclaim memory resources or when it is being shut down). In such a case, the container calls the destroy method of the Servlet interface. In this method, you release any resources the servlet is using and save any persistent state. The destroy method releases the database object created in the init method.
- A servlet's service methods should all be complete when a servlet is removed. The server tries to ensure this by calling the destroy method only after all service requests have returned or after a server-specific grace period, whichever comes first. If your servlet has operations that may run longer than the server's grace period, the operations could still be running when destroy is called. You must make sure that any threads still handling client requests complete.
  - Keep track of how many threads are currently running the service method.
  - Provide a clean shutdown by having the destroy method notify long-running threads of the shutdown and wait for them to complete.
  - Have the long-running methods poll periodically to check for shutdown and, if necessary, stop working, clean up, and return.

Uploading Files with Jakarta Servlet Technology

- supporting file uploads is a very basic and common requirement for many web applications.
- The Jakarta Servlet specification now helps to provide a viable solution to the problem in a generic and portable way.
- Jakarta Servlet technology now supports file upload out of the box, so any web container that implements the specification can parse multipart requests and make mime attachments available through the HttpServletRequest object.
- A new annotation, jakarta.servlet.annotation.MultipartConfig, 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.

Asynchronous Processing

- 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.
- 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.

Nonblocking I/O (1)

- Web containers in application servers normally use a server thread per client request. To develop scalable web applications, you must ensure that threads associated with client requests are never sitting idle waiting for a blocking operation to complete.
- Asynchronous Processing provides a mechanism to execute application-specific blocking operations in a new thread, returning the thread associated with the request immediately to the container. Even if you use asynchronous processing for all the application-specific blocking operations inside your service methods, threads associated with client requests can be momentarily sitting idle because of input/output considerations.
- For example, if a client is submitting a large HTTP POST request over a slow network connection, the server can read the request faster than the client can provide it. Using traditional I/O, the container thread associated with this request would be sometimes sitting idle waiting for the rest of the request.
- Jakarta EE provides nonblocking I/O support for servlets and filters when processing requests in asynchronous mode.

Nonblocking I/O (2)

- The following steps summarize how to use nonblocking I/O to process requests and write responses inside service methods.
  - Put the request in asynchronous mode as described in Asynchronous Processing.
  - Obtain an input stream and/or an output stream from the request and response objects in the service method.
  - Assign a read listener to the input stream and/or a write listener to the output stream.
  - Process the request and the response inside the listener's callback methods.

Server Push

- Server push is the ability of the server to anticipate what will be needed by the client in advance of the client's request. It lets the server pre-populate the browser's cache in advance of the browser asking for the resource to put in the cache.
- Server push is the most visible of the improvements in HTTP/2 to appear in the servlet API. All of the new features in HTTP/2, including server push, are aimed at improving the performance of the web browsing experience.
- Server push derives its contribution to improved browser performance from the fact that servers know what additional assets (such as images, stylesheets, and scripts) go along with initial requests.
- For example, servers might know that whenever a browser requests index.html, it will shortly thereafter request header.gif, footer.gif, and style.css. Servers can preemptively start sending the bytes of these assets along with the bytes of the index.html.

Protocol Upgrade Processing

- In HTTP/1.1, clients can request to switch to a different protocol on the current connection by using the Upgrade header field. If the server accepts the request to switch to the protocol indicated by the client, it generates an HTTP response with status 101 (switching protocols). After this exchange, the client and the server communicate using the new protocol.
- For example, a client can make an HTTP request to switch to the XYZP protocol as follows (left):

```
GET /xyzpresource HTTP/1.1
Host: localhost:8080
Accept: text/html
Upgrade: XYZP
Connection: Upgrade
OtherHeaderA: Value
```

```
HTTP/1.1 101 Switching Protocols
Upgrade: XYZP
Connection: Upgrade
OtherHeaderB: Value

(XYZP data)
```

- The client can specify parameters for the new protocol using HTTP headers. The server can accept the request and generate a response as up (right):

HTTP Trailer

- HTTP trailer is a collection of a special type of HTTP headers that comes after the response body. The trailer response header allows the sender to include additional fields at the end of chunked messages in order to supply metadata that might be dynamically generated while the message body is sent, such as a message integrity check, digital signature, or post-processing status.
- If trailer headers are ready for reading, isTrailerFieldsReady() will return true. Then a servlet can read trailer headers of the HTTP request using the getTrailerFields method of the HttpServletRequest interface. If trailer headers are not ready for reading, isTrailerFieldsReady() returns false and will cause an IllegalStateException.
- A servlet can write trailer headers to the response by providing a supplier to the setTrailerFields() method of the HttpServletResponse interface. The following headers and types of headers must not be included in the set of keys in the map passed to setTrailerFields(): Transfer-Encoding, Content-Length, Host, controls and conditional headers, authentication headers, Content-Encoding, Content-Type, Content-Range, and Trailer. When sending response trailers, you must include a regular header, called Trailer, whose value is a comma-separated list of all the keys in the map that is supplied to the setTrailerFields() method. The value of the Trailer header lets the client know what trailers to expect.

Questions?