Web Applications Developments

ITWT413

LECTURE 9: JAKARTA EE WEB PROFILE

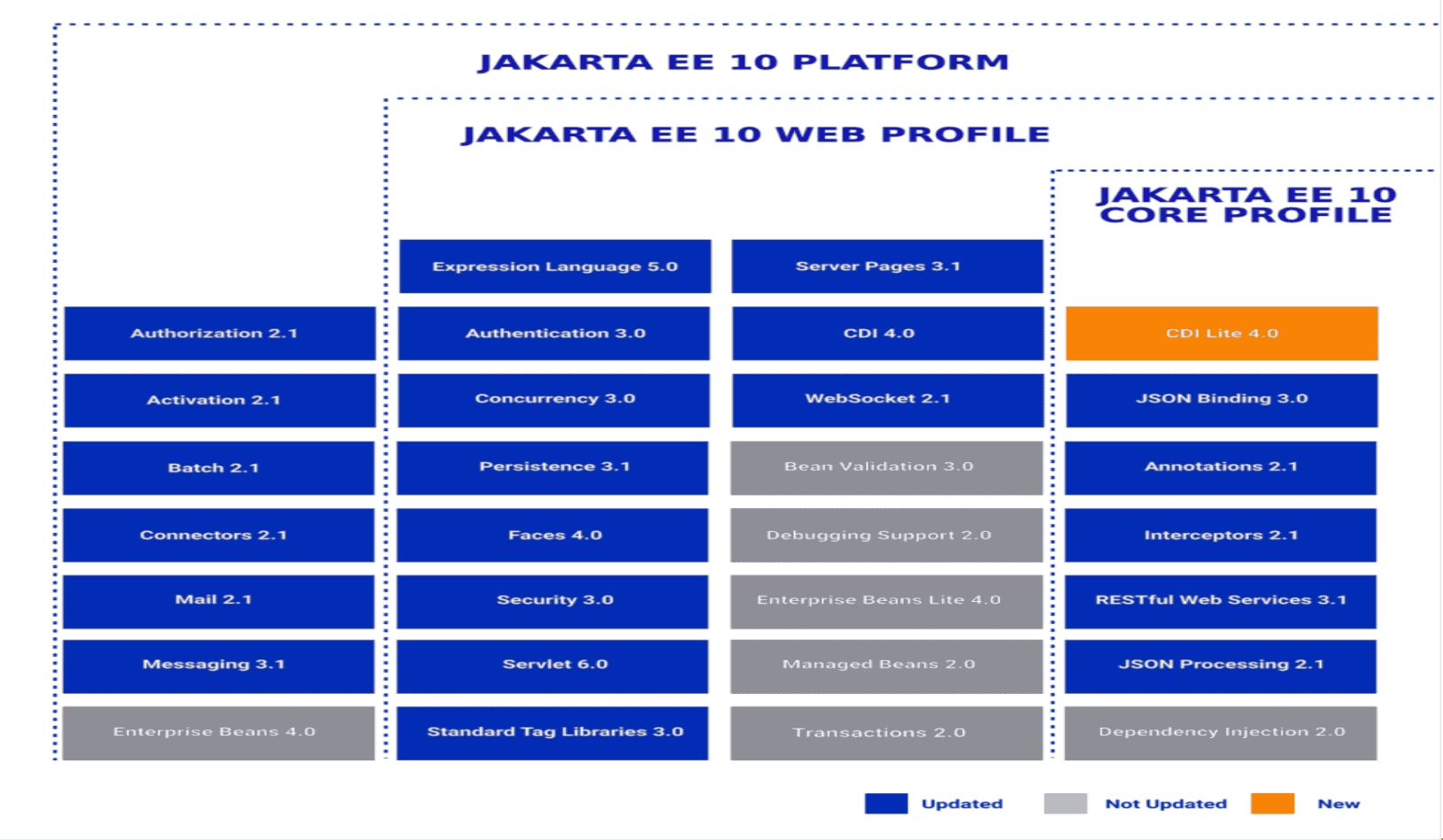
JAKARTA WEBSOCKET

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

- Introduction to WebSocket
  - Creating WebSocket Applications in the Jakarta EE Platform
- Creating and Deploying a WebSocket Endpoint
- Programmatic Endpoints
- Annotated Endpoints
- Sending and Receiving Messages
  - Sending Messages
  - Receiving Messages
- Maintaining Client State
- Using Encoders and Decoders
  - Implementing Encoders to Convert Java Objects into WebSocket Messages
  - Implementing Decoders to Convert WebSocket Messages into Java Objects
- Path Parameters
- Handling Errors
- Specifying an Endpoint Configurator Class

Introduction to WebSocket

- In the traditional request-response model used in HTTP, the client requests resources, and the server provides responses.
- The exchange is always initiated by the client; the server cannot send any data without the client requesting it first.
- This model worked well for the World Wide Web when clients made occasional requests for documents that changed infrequently, but the limitations of this approach are increasingly relevant as content changes quickly and users expect a more interactive experience on the Web.
- The WebSocket protocol addresses these limitations by providing a full-duplex communication channel between the client and the server. Combined with other client technologies, such as JavaScript and HTML5, WebSocket enables web applications to deliver a richer user experience.

Introduction to WebSocket

- 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.

WebSocket Connection

- An example handshake from a client looks like this:

```
GET /path/to/websocket/endpoint HTTP/1.1
Host: localhost
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: xqBt3ImNzJbYqRINxEFlkg==
Origin: http://localhost
Sec-WebSocket-Version: 13
```

- An example handshake from the server in response to the client looks like this

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: K7DJLdLooIwIG/MOpvWFB3y3FE8=
```

How WebSocket Connection is Working?

- The server applies a known operation to the value of the Sec-WebSocket-Key header to generate the value of the Sec-WebSocket-Accept header.
- The client applies the same operation to the value of the Sec-WebSocket-Key header, and the connection is established successfully if the result matches the value received from the server.
- The client and the server can send messages to each other after a successful handshake.
- WebSocket supports text messages (encoded as UTF-8) and binary messages.
- The control frames in WebSocket are close, ping, and pong (a response to a ping frame). Ping and pong frames may also contain application data.

WebSocket endpoints

- WebSocket endpoints are represented by URIs that have the following form:

```
ws://host:port/path?query
wss://host:port/path?query
```

- The ws scheme represents an unencrypted WebSocket connection, and the wss scheme represents an encrypted connection.
- The port component is optional; the default port number is 80 for unencrypted connections and 443 for encrypted connections.
- The path component indicates the location of an endpoint within a server. The query component is optional.
- Modern web browsers implement the WebSocket protocol and provide a JavaScript API to connect to endpoints, send messages, and assign callback methods for WebSocket events (such as opened connections, received messages, and closed connections).

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.

Creating and Deploying a WebSocket Endpoint

- The process for creating and deploying a WebSocket endpoint:
  - Create an endpoint class.
  - Implement the lifecycle methods of the endpoint.
  - Add your business logic to the endpoint.
  - Deploy the endpoint inside a web application.
- As opposed to servlets, WebSocket endpoints are instantiated multiple times.
- The process is slightly different for programmatic endpoints and annotated endpoints.

Programmatic endpoints Vs Annotated endpoints.

Programmatic Endpoints

The following example shows how to create an endpoint by extending the `Endpoint` class:

```
public class EchoEndpoint extends Endpoint {
   @Override
   public void onOpen(final Session session, EndpointConfig config) {
      session.addMessageHandler(new MessageHandler.Whole<String>() {
         @Override
         public void onMessage(String msg) {
            try {
               session.getBasicRemote().sendText(msg);
            } catch (IOException e) { ... }
         }
      });
   }
}
```

Annotated Endpoints

The following example shows how to create the same endpoint from Programmatic Endpoints using annotations instead:

```
@ServerEndpoint("/echo")
public class EchoEndpoint {
   @OnMessage
   public void onMessage(Session session, String msg) {
      try {
         session.getBasicRemote().sendText(msg);
      } catch (IOException e) { ... }
   }
}
```

WebSocket Endpoint Lifecycle Annotations

Annotation	Event	Example
`OnOpen`	Connection opened	`@OnOpen` `public void open(Session session, EndpointConfig conf) { }`
`OnMessage`	Message received	`@OnMessage` `public void message(Session session, String msg) { }`
`OnError`	Connection error	`@OnError` `public void error(Session session, Throwable error) { }`
`OnClose`	Connection closed	`@OnClose` `public void close(Session session, CloseReason reason) { }`

Sending Messages

- WebSocket endpoints can send and receive text and binary messages.
- In addition, they can also send ping frames and receive pong frames.
- This slide describes how to use the Session and RemoteEndpoint interfaces to send messages to the connected peer and how to use the OnMessage annotation to receive messages from it.
- Follow these steps to send messages in an endpoint.

1. Obtain the Session object from the connection.
2. Use the Session object to obtain a RemoteEndpoint object.
3. Use the RemoteEndpoint object to send messages to the peer.

Receiving Messages

- The OnMessage annotation designates methods that handle incoming messages.
- You can have at most three methods annotated with @OnMessage in an endpoint, one for each message type: text, binary, and pong.
- The following example demonstrates how to designate methods to receive all three types of messages:

```
@ServerEndpoint("/receive")
public class ReceiveEndpoint {
    @OnMessage
    public void textMessage(Session session, String msg) {
        System.out.println("Text message: " + msg);
    }
    @OnMessage
    public void binaryMessage(Session session, ByteBuffer msg) {
        System.out.println("Binary message: " + msg.toString());
    }
    @OnMessage
    public void pongMessage(Session session, PongMessage msg) {
        System.out.println("Pong message: " +
                           msg.getApplicationData().toString());
    }
}
```

Maintaining Client State

- Because the container creates an instance of the endpoint class for every connection, you can define and use instance variables to store client state information

```
@ServerEndpoint("/delayedecho")
public class DelayedEchoEndpoint {
   @OnOpen
   public void open(Session session) {
      session.getUserProperties().put("previousMsg", " ");
   }
   @OnMessage
   public void message(Session session, String msg) {
      String prev = (String) session.getUserProperties()
                                    .get("previousMsg");
      session.getUserProperties().put("previousMsg", msg);
      try {
         session.getBasicRemote().sendText(prev);
      } catch (IOException e) { ... }
   }
}
```

Using Encoders and Decoders

- Jakarta WebSocket provides support for converting between WebSocket messages and custom Java types using encoders and decoders.
- An encoder takes a Java object and produces a representation that can be transmitted as a WebSocket message; for example, encoders typically produce JSON, XML, or binary representations.
- A decoder performs the reverse function; it reads a WebSocket message and creates a Java object.

Path Parameters

- The ServerEndpoint annotation enables you to use URI templates to specify parts of an endpoint deployment URI as application parameters.
- For example, consider this endpoint:

```
@ServerEndpoint("/chatrooms/{room-name}")
public class ChatEndpoint {
    ...
}
```

- If the endpoint is deployed inside a web application called chatapp at a local Jakarta EE server in port 8080, clients can connect to the endpoint using any of the following URIs:

```
http://localhost:8080/chatapp/chatrooms/currentnews
http://localhost:8080/chatapp/chatrooms/music
http://localhost:8080/chatapp/chatrooms/cars
http://localhost:8080/chatapp/chatrooms/technology
```

Handling Errors

- To designate a method that handles errors in an annotated WebSocket endpoint, decorate it with @OnError.
- This method is invoked when there are connection problems, runtime errors from message handlers, or conversion errors when decoding messages.

```
@ServerEndpoint("/testendpoint")
public class TestEndpoint {
    ...
    @OnError
    public void error(Session session, Throwable t) {
        t.printStackTrace();
        ...
    }
}
```

Specifying an Endpoint Configurator Class

- Jakarta WebSocket enables you to configure how the container creates server endpoint instances.
- You can provide custom endpoint configuration logic to:
  - Access the details of the initial HTTP request for a WebSocket connection
  - Perform custom checks on the Origin HTTP header
  - Modify the WebSocket handshake response
  - Choose a WebSocket subprotocol from those requested by the client
  - Control the instantiation and initialization of endpoint instances
- To provide custom endpoint configuration logic, you extend the ServerEndpointConfig.Configurator class and override some of its methods.
- In the endpoint class, you specify the configurator class using the configurator parameter of the ServerEndpoint annotation.

Questions?