

ID2212 Network Programming with Java
Lecture 12

Enterprise JavaBeans (EJBs) and
WebSockets

Leif Lindbäck, Vladimir Vlassov,
Paris Carbone
KTH/ICT/SCS
HT 2016

Enterprise JavaBeans (EJB)

[javax.ejb](http://java.sun.com/javax/ejb)

home page:

<http://www.oracle.com/technetwork/java/index-jsp-140203.html>

Outline

- **Enterprise JavaBeans (EJB)**
 - **Session**
 - **Message-Driven**
- **Example: Bank Account Application – to be considered at Exercise 4 (see slides of Exercise 4)**

Enterprise JavaBeans (EJB)

- **An EJB is a server-side components that exposes methods for performing application logic**
- **EJB 3**
 - *Session Beans* (stateless and stateful) represent actions (a single action, a session)
 - Performs a task for a client; optionally may implement a web service
 - *Message-driven Beans*
 - Acts as a listener for a particular messaging type, such as the Java Message Service (JMS)
- **EJB 2**
 - *Session Beans* (stateless and stateful)
 - *Entity Beans* represent persistent stateful entity (e.g. account, person, customer). An entity bean – a row in a database
 - Entity beans have been replaced by Java Persistence API entities.

Session Beans

- **Business process objects, for example, business logic, workflow and similar**
- **A *session bean* represents a single client inside the Application Server.**
 - **Used only by one client at a time**
 - **Lives for the session (or lifetime) of the calling client**
- **Two kind of sessions EJBs**
 - *Stateless session EJBs*
 - *Stateful session EJB*

A Stateful Session Bean

- The bean class is annotated as `@Stateful`
- Spans multiple client calls and retains state on behalf of the individual client (conversational state).
- One bean instance per client.
 - Values of instance variables represent the state of a unique client session.
 - A.k.a. the *conversational state*
 - If the client removes the bean or terminates, the session ends and the state disappears.
- **Example: A shopping cart**

A Stateless Session Bean

- The bean class is annotated as `@Stateless`
- Accommodates only a single request.
- Does not maintain a conversational state with the client.
 - May contain state, but not conversational state related to a specific client.
- A pool of beans
 - All instances of a stateless bean are equivalent
 - Better scalability than stateful beans.
- Example: A currency converter

Business Interface of a Session Bean

- A client uses a session bean through the methods of the bean's *business interface*.
 - The interface includes the business methods exposed by the bean's class
 - The bean class implements the business interface
- A client of a session bean can be a web component, an application client, or another EJB

Business Interface of a Session Bean

(contd)

- The client gets a reference to the session bean's interface either through *injection or JNDI lookup*
- The following example illustrates both options. Note that JNDI lookup is *very seldom* required.

```
@WebServlet(name = "AdderServlet", urlPatterns = {"/AdderServlet"})
public class AdderServlet extends HttpServlet {
    @EJB
    private AdderRemote adder; //injection
    //private AdderRemote adder = lookupAdderBeanRemote(); // through lookup
    private AdderRemote lookupAdderBeanRemote() {
        try {
            Context c = new InitialContext();
            return (AdderRemote) c.lookup(
                "java:global/Adder/Adder-ejb/AdderBean!adder.AdderRemote");
        } catch (NamingException ne) {
            Logger.getLogger(getClass().getName()).log(Level.SEVERE,
                "exception caught", ne);
            throw new RuntimeException(ne);
        }
    }
}
```

Business Interface Design Options

- *Local interface* for local clients on the same JVM
- *Remote interface* for remote clients on different JVM
 - Remote interface obeys the rules of Java RMI.
- *No-interface* view for local clients
 - All public methods of the bean class are automatically exposed to the client
 - Does not require a separate interface; there is no the *implements* clause
 - The same behavior as the local interface view
 - Simplifies programming
- *Web Services interface*
 - Exposing business methods as Web Services
 - Stateless session beans can be invoked over SOAP/HTTP

A Session EJB with a Remote Interface

- An interface for remote access
- Annotate the interface with the `@Remote` annotation:

```
@Remote
public interface InterfaceName {
    ...
}
```

- Annotate the bean class with `@Remote`, specifying the business interface(s):

```
@Remote(InterfaceName.class)
public class BeanName implements InterfaceName
{
    ...
}
```

A Session EJB with a Local Interface

- An interface for local access
- Annotate the interface with the `@Local` annotation:

```
@Local
public interface InterfaceName {
    ...
}
```

- Annotate the bean class with `@Local`, specifying the business interface(s):

```
@Local (InterfaceName.class)
public class BeanName implements InterfaceName
{
    ...
}
```

A Session EJB without Interface

- Annotate the bean class with `@LocalBean` annotation

```
@LocalBean  
public class BeanName {  
    ...  
}
```

Deciding on Remote or Local Access

- **Tight or loose coupling of related beans:**
 - Tightly coupled beans depend on one another, and they are good candidates for *local access*.
- **Type of client:**
 - To enable clients accessing the EJB direct, not through a web interface, the EJB must allow *remote access*,
 - For EJBs accessed by web components or other EJBs, *local access* is almost always preferred.

Deciding on Remote or Local Access

(contd)

- **Semantics:**

Remote calls use pass-by-value semantics, local calls use pass-by-reference.

- **Component distribution:**

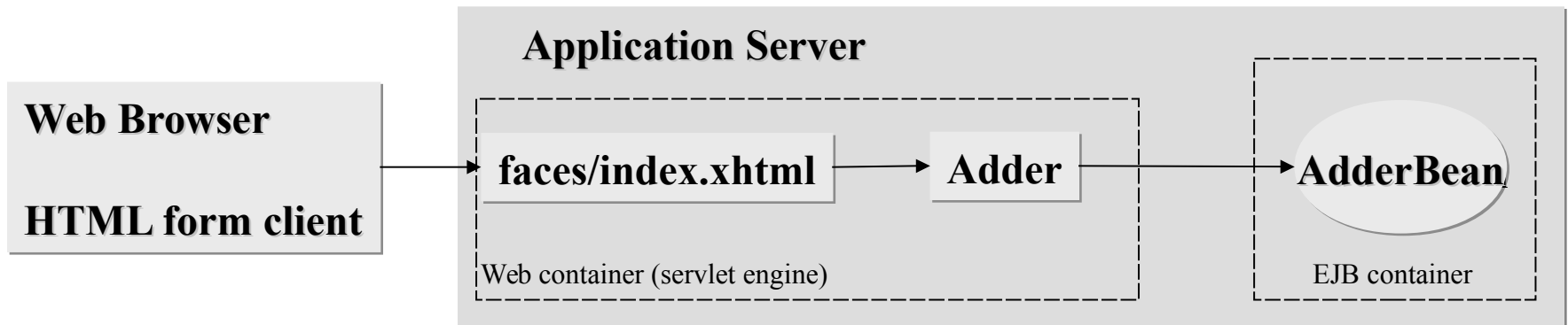
- Server-side components can be distributed among multiple machines for scalability.
- Web components may run on a different server than EJBs, but this is very seldom useful.

- **Performance:**

- Remote calls are slower than local calls.

Example: Adder Service

- **A client-server application**
 - A user enters an integer number via an HTML form and gets a sum accumulated by the Adder application.
- **The Adder application includes:**
 - The index.xhtml JSF page
 - The Adder managed bean
 - The AdderBean EJB (session, stateful) that accumulates a sum of integers entered by a user, and returns the sum on request



Client

- An HTML Form in a Web page (index.xhtml)

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Adder site</title>
  </h:head>

  <h:body>
    <h1>Adder Site</h1>
    <h3>The running total is #{adder.total}</h3>
    <h:form>
      <p><h:outputLabel for="operand" value="Please enter an integer: "/>
        <h:inputText id="operand" value="#{adder.operand}"/>
      </p>
      <p><h:commandButton value="Add"/></p>
    </h:form>
  </h:body>
</html>
```

Adder Managed Bean

- The Adder managed bean is a client of the Adder EJB

```
package se.kth.id2212.lec12.view;

import java.io.Serializable;
import javax.ejb.EJB;
import javax.enterprise.context.SessionScoped;
import javax.inject.Named;
import se.kth.id2212.lec12.model.AdderBean;

@Named(value = "adder")
@SessionScoped
public class Adder implements Serializable {

    private static final long serialVersionUID = -3552965833778221283L;
    @EJB
    private AdderBean adderBean;
```

Adder Managed Bean (cont'd)

```
public int getTotal() {  
    return adderBean.getTotal();  
}
```

```
public void setOperand(Integer operand) {  
    adderBean.add(operand);  
}
```

```
public Integer getOperand() {  
    return null;  
}
```

Adder EJB (Stateful Session Bean)

- **Interface operations:**

```
public void add (int number)
```

```
public int getTotal()
```

- `private int total; // state of the bean`

Adder EJB (contd)

```
package se.kth.id2212.lec12.model;

import javax.ejb.Stateful;

@Stateful
public class AdderBean {
    private int total;

    public void add(int operand) {
        total = total + operand;
    }

    public int getTotal() {
        return total;
    }
}
```

Message-Driven Beans

- A *message-driven bean* is an EJB that allows Java EE applications to process messages *asynchronously*.
 - Normally acts as a *JMS message listener*.
- Defference between m-d beans and session beans
 - Clients do not access m-d beans by method calls through interfaces;
 - M-d bean instead implements the **MessageListener** interface with the **onMessage** method.
- A m-d bean resembles a stateless session bean
 - M-d bean instances retain no conversational state for a specific client;
 - All instances of a m-d bean are equivalent;
 - A single m-d bean can process messages from multiple clients.

Features of Message-Driven EJBs

- **Stateless.**
- **Execute upon receipt of a single client message.**
- **Invoked asynchronously.**
- **Can be transaction-aware.**

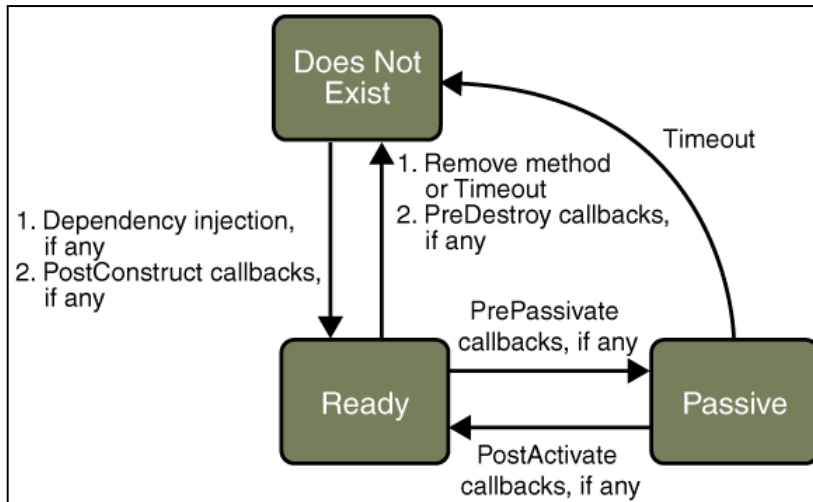
- **An example is a log service**

Client Interaction with a Message-Driven Bean

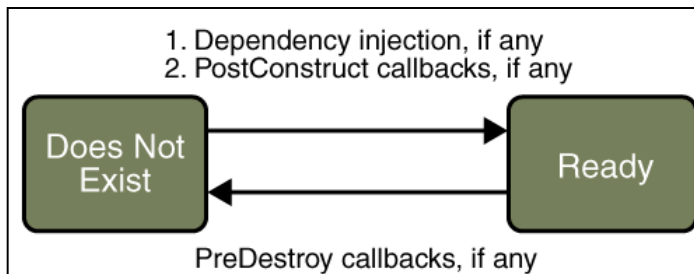
- A client interacts with a m-d bean through JMS by sending messages to the destination for which the message-driven bean class is the JMS message listener.
- An m-d bean's class is annotated `@MessageDriven` with a name of the bean's destination, e.g.
`@MessageDriven(mappedName = "jms/Queue")`
- When a message arrives, the container calls the message-driven bean's `onMessage` method to process the message.

Life Cycles of EJBs

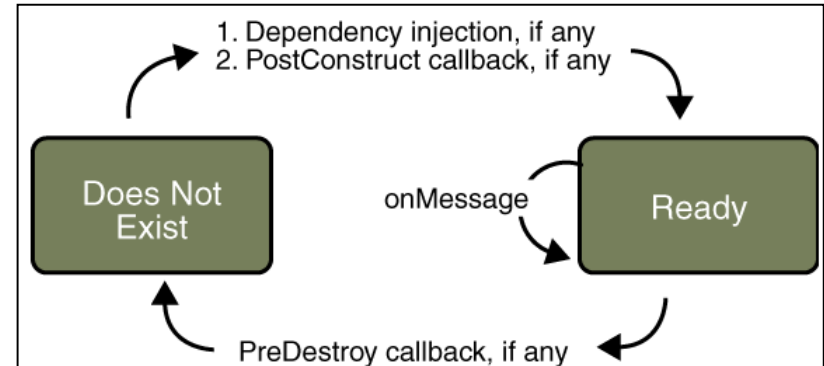
- **Stateful session bean**



- **Stateless session bean**



- **Message-driven bean**



- During the bean's lifetime, the EJB Container invokes methods annotated **@PostConstruct**, **@PrePassivate**, **@PostActivate**, **@PreDestroy**, **@Remove**.
- When a client calls a business method on a bean, which does not exist or is passive, the container constructs the bean or activates the bean to make it ready.

Networking with WebSockets

[javafx.websocket](http://javafx.com/websocket)

home page: <http://www.oracle.com/technetwork/articles/java/jsr356-1937161.html>

Outline

Web Based Communication

- HTTP Limitations

- WebSocket Contribution

The WebSocket API in JavaEE

- Endpoints

- Sending/Receiving Messages

- Encoders/Decoders

- Accessing Sessions

- WebSocket Clients

HTTP Limitations

Half-Duplex (Request / Response based)

A *client* requests a resource and the *server* provides access to it

Verbose (adds communication overhead)

Complicated when resources change *frequently*

eg. *Social networking, gaming, collaborative editing, financial applications etc.*

Polling is **inefficient**

HTTP Limitations

Long Polling (Comet) offered small improvement

- Establishes a persistent connection and waits for the server to push data when it becomes available

- Still requires a new connection (handshake) per update

- Still adds the communication overhead of HTTP

WebSockets Contribution

- **WebSockets** (RFC 6455) build on a **single** TCP connection and offer **bi-directional, full-duplex** communication through the web (HTTP)
- **Message-Based** communication (text, binary and ping-pong messages)
- Offers *addressing* and *protocol naming* mechanisms to support multiple services on one port (eg. 80)

WebSockets Contribution

- WebSocket connection **endpoints** can be represented by URIs:
 - **ws://host:port/path?query** (*plain websocket connection*)
 - **wss://host:port/path?query** (*encrypted websocket connection*)
- They are currently supported by modern browsers through a JavaScript API to allow
 - Endpoints connection
 - Callbacks for Sending/Receiving messages and connection state events

WebSockets Contribution

A websocket connection is established by **upgrading a HTTP connection**. Browser and server exchange the following handshake.

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

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: K7DJLdLooIwIG/MOpvWFB3y3FE8=

Messaging Over WebSockets

- WebSockets is a protocol enabling transport over a network, much like TCP or UDP.
- It is often convenient to use a higher level protocol on top of WebSockets, instead of using plain websockets endpoints.
- There are many such protocols, for example
 - Web Application Messaging Protocol, WAMP
 - Java Message Service, JMS
 - Advanced Message Queuing Protocol, AMQP
 - Extensible Messaging and Presence Protocol, XMPP

WebSockets in Java

- WebSockets were introduced in JavaEE 7.0
 - `javax.websocket` and `javax.websocket.server`
 - Supported by Web Containers (Tomcat 8, Glassfish 4.0, Jetty, WildFly)
- **Endpoints** are instances of the `javax.websocket.Endpoint` class
- There are **two** ways of defining Endpoints:
 - **Programmatic Endpoints:** Extending `javax.websocket.Endpoint` and overriding lifecycle methods
 - **Annotated Endpoints:** Decorating classes with provided annotations.
*This is generally **preferred** to the programmatic approach.*

Endpoints

- WebSocket Endpoints are **not** sockets! Sockets are **stream-based** while Endpoints are **message-based**.
- WebSocket Server Endpoints are **not** servlets! The container creates **one** Endpoint instance **per connection**.
- Any POJO can be converted into an Endpoint simply via using appropriate WebSocket annotations.

Annotated Endpoints

- Easier to Write
- A simple Endpoint that echoes back every text message:

```
@ServerEndpoint("/echo")
public class EchoEndpoint {

    @OnMessage
    public String doEcho(String textMsg) {
        return textMsg;
    }
}
```

Programmatic Endpoints

- More complicated to write.
- A simple Endpoint that echoes back every text message:

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

Annotated Endpoints

Easier to Write

A simple Endpoint that echoes back every text message:

```
@ServerEndpoint("/echo")
public class EchoEndpoint {

    @OnMessage
    public String doEcho(String textMsg) {
        return textMsg;
    }
}
```

Annotated Endpoints

- Annotated Endpoints are simpler and automatically deployed with the Web Application on the relative path defined by the [ServerEndpoint](#) annotation.
- There are annotations for every Endpoint lifecycle event:

Annotation	Lifecycle Event	Expected Parameters
OnOpen	Connection Opened	Session, EndpointConfig, PathParam
OnMessage	Message Received	Session, String ByteBuffer PongMessage POJO, PathParam
OnError	Error Occurred	Session, Throwable, PathParam
OnClose	Connection Closed	Session, CloseReason, PathParam

Receiving Messages

- Currently the following message types are supported:

Message Type	Description
String	A message decoded and wrapped as a character string
ByteBuffer	A message passed as a byte sequence to be manually de-serialized
Pong	Dummy message to be used for checking the connection

Receiving Messages

- An Endpoint can have up to three message handlers

```
@ServerEndpoint("/ireadeverything")
public class ThreeTypeEndpoint {

    @OnMessage
    public void receiveText(Session session, String textMsg) {
    }

    @OnMessage
    public void receiveBinary(Session session, ByteBuffer binaryMsg) {
    }

    @OnMessage
    public void receivePong(Session session, PongMessage pongMsg) {
    }
}
```

Sending Messages

- Each connection's `session` object exposes two remote peer interfaces for sending messages:
 - `RemoteEndpoint.Basic` for blocking communication
 - `RemoteEndpoint.Async` for asynchronous communication

```
@OnMessage
Public void messageBack(Session session, String str)
{
    session.getBasicRemote().sendText(str); //blocks
    session.getAsyncRemote().sendText(str); //returns instantly
}
```

Encoders/Decoders

- For converting messages to/from POJOs the API offers the following interfaces:

Interface	Description
<code>Encoder.Text<T></code>	Encode from T to String
<code>Encoder.Binary<T></code>	Encode from T to binary
<code>Decoder.Text<T></code>	Decode from String to T
<code>Decoder.Binary<T></code>	Decode from binary to T

Encoders/Decoders

- For each server Endpoint we can define **many** encoders and at most **one** decoder per input type (String, Binary) as ServerEndpoint annotation parameters.

```
@ServerEndpoint(  
    value = "/echoserver",  
    encoders = {MyEncoderA.class, MyEncoderB.class},  
    decoders = {MyDecoder.class}  
)  
public class EchoServer{  
    @OnMessage  
    public void message(Session session, MyMessage msg){  
        ...  
    }  
}
```

Encoders/Decoders

- Defining an encoder

```
public class MyEncoderA implements Encoder.Text<MyMessageA>{  
    @Override  
    public void init(EndpointConfig econf){}  
    @Override  
    public void destroy(){}  
    @Override  
    public String encode(MyMessageA myMsg){  
        return myMsg.toJson();  
    }  
}
```

- Same for MyMessageB...

Encoders/Decoders

- Defining the decoder. For multiple type support there should be one **common superclass/interface** defined.

```
public class MyDecoder implements Decoder.Text<MyMessage>{
    @Override
    public void init(EndpointConfig econf){}
    @Override
    public void destroy(){}
    @Override
    public MyMessage decode(String textMsg) throws DecodeException{
        //parse textMsg
        ...
        // MyMessageA and MyMessageB should inherit MyMessage
        return new MyMessageA(textMsg);
        ... //or
        return new MyMessageB(textMsg);
    }
    ...
}
```

Encoders/Decoders

- For sending a custom Message use the `sendObject` method of a `RemoteEndpoint`.

```
MyMessageA myA = new MyMessageA();  
MyMessageB myB = new MyMessageB();  
  
Session.getBasicRemote.sendObject(myA);  
Session.getAsyncRemote.sendObject(myA);
```

Accessing Sessions

- A **Session** object provides access to **all sessions** connected to the same **endpoint**. (useful for web applications where users interact eg. chat, games, collaborative editing etc.)

```
@ServerEndpoint("/myblackboard")
public class Blackboard {
    @OnMessage
    public void onMessage(Session session, String msg){
        try{
            for(Session userSession : session.getOpenSession()) {
                if(userSession.isOpen())
                    userSession.getBasicRemote().sendText(msg);
            }
        }
    }
}
```

<http://docs.oracle.com/javaee/7/tutorial/doc/websocket005.htm>

WebSocket Clients

- Typically WebSocket clients run in [browsers](#), written in JavaScript.

```
var myWebsocket;  
  
function connect() {  
    myWebSocket = new WebSocket("ws://test.kth.se:8080/myblackboard");  
    myWebSocket.onmessage = handleMessage;  
}  
  
var blackboardPanel = document.getElementById("mypanel");  
  
function handleMessage(blackboardMsg){  
    blackboardPanel.innerHTML += blackBoardMsg;  
}
```

More Info

- Javax.websocket API
 - <http://docs.oracle.com/javaee/7/api/javax/websocket/package-summary.html>
- Specification
 - <http://tools.ietf.org/html/rfc6455>