Enterprise Apps and JEE

Alessio Bechini Dept. of Information Engineering, Univ. of Pisa

a.bechini@ing.unipi.it

Outline

Web Applications, Enterprise Applications, and JEE

- General Ideas
- Web Applications
- Servlets and More
- Enterprise Applications
- JNDI
- EJBs
- JMS

General Ideas

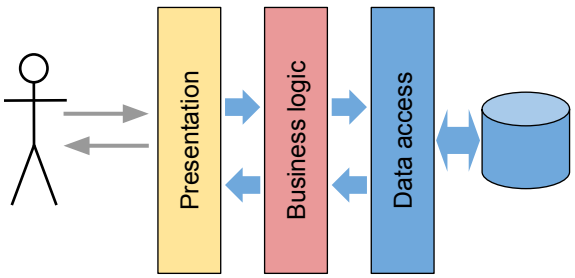
3-Tier Architecture

Typical organization of a (web) application: **3-tier architecture**

➔ a client-server architecture where:

1. user interface,
2. functional process logic,
3. data access/data storage

are developed and maintained as *independent modules*, possibly on separate platforms.

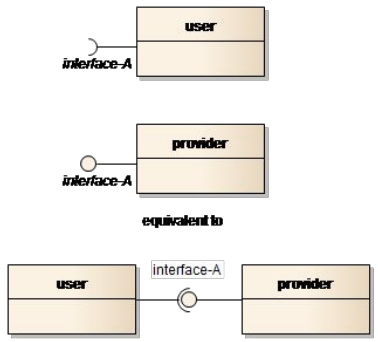


Use of Containers in Middleware

At the middleware level, components can be organized in *containers*, and components managed within containers.

Containers take care of supporting their **managed components**, providing them with the required functionalities.

A container, as well as any component, has both *provided* and *requested interfaces*.



From Now on...

Java EE™

since late 2019...

JAKARTA® EE

Web Applications

Static vs Dynamic Content

HTTP deals with requesting a resource, and getting it.

It's up to the server to provide the resource,
either retrieving it as a file (static content),
or generating it on the fly by means of a program (dynamic content).

The server has to tell apart the type of resource just from its URL.

The possible generation of content must be guided by the server.

Common Gateway Interface

A standard protocol for web servers to execute shell programs that dynamically generate web pages.
CGI scripts are *usually* placed in the special directory `cgi-bin`.
Parameters are passed via environment vars and by std input.

Example of URL:
http://xyz.com/**cgi-bin/myscript.pl**/my/pathinfo?a=1&b=2

To solve process spawning overhead → **FastCGI** approaches

Common Gateway Interface

A standard protocol for web servers to execute shell programs that dynamically generate web pages.
CGI scripts are *usually* placed in the special directory `cgi-bin`.
Parameters are passed via environment vars and by std input.

A Perl script

PATH_INFO

QUERY_STRING

Example of URL:
http://xyz.com/**cgi-bin/myscript.pl**/my/pathinfo?a=1&b=2

To solve process spawning overhead → **FastCGI** approaches

Beyond CGI-Scripts

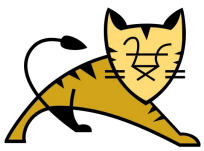
Idea: to improve performance, the code for content generation could be executed **internally** to the web server, i.e. within the same process, possibly using multithreading.

In Microsoft environments: classic **ASP**

In JavaEE: **Servlets**, and related technologies

MANNAGGIA ADLO !

The Tomcat Web Server



Basic internal components:

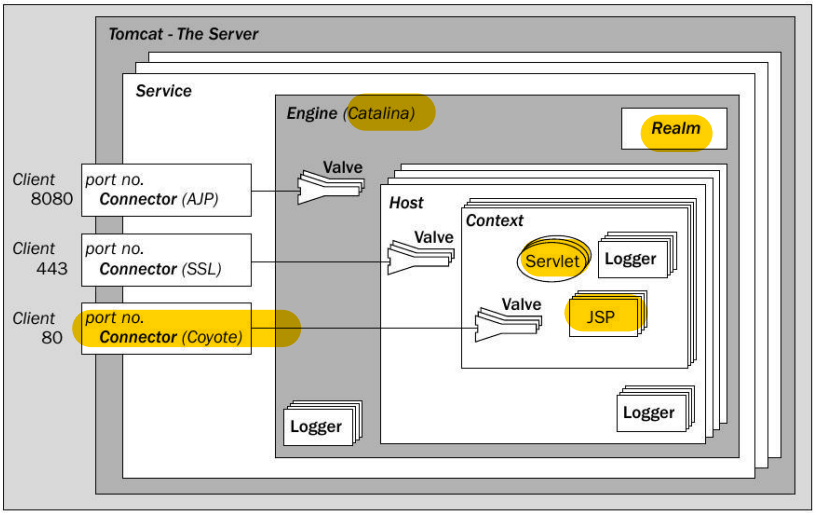
- **Catalina:** "Engine," Servlet container; it refers also to a *Realm* (DB of usernames, passwords, and relative roles).
- **Coyote:** HTTP "Connector," listens on ports and forward requests to the engine
- **Jasper:** JSP container

"Service" element: combination of 1+ Connectors that share a single Engine component for processing incoming requests.

Tomcat Architecture

Valve:
element to be inserted in the REQ processing pipeline

Other nested elems:
(Session) Manager,
(W.apps class) Loader,
Listener(s), etc.



Hosts and Contexts

A “Host” component represents a *virtual host*, i.e. an association of a network name for a server (e.g. "www.mycompany.com") with the particular server Catalina is running on.

A “Context” element represents a web application, which is run within a particular virtual host.

The web app used to process each HTTP request is selected by Catalina based on matching the longest possible prefix of the Request URI against the **context path** of each defined Context.

Context Path and Base File Name

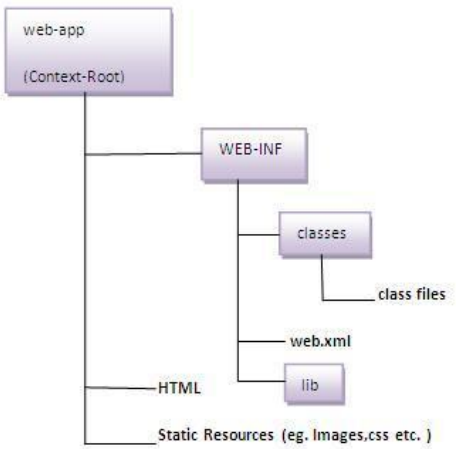
In the filesystem, one directory (*appBase*, default “webapps”) is used to keep the material for web applications.

Each web app corresponds to a base file name.

Rules to obtain the base file name, given the context path:

- If the context path is “” ➔ “ROOT”
- If the context path is not “” ➔ base name = context path with the leading '/' removed, and any remaining '/' replaced with '#'.

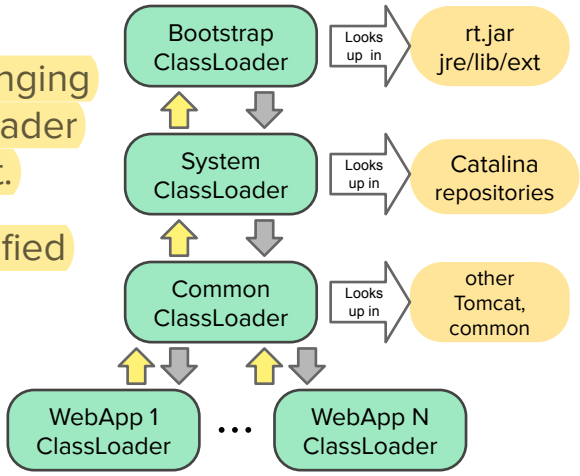
Structure of a WebApp Directory



Not-interfering Contexts

To avoid interference of classes belonging to different contexts, different classloader hierarchies are used for each context.

Remember: any loaded class is identified by its name AND the classloader that has loaded it, so possibly the same class used in two web apps is loaded twice.



Servlets and Their Container

Servlets: Java classes/objects that respond to a request, aimed at producing the content for the response.
The thread-per-request model is generally applied → ***synch issues!***

Servlets are kept in a container.
Their methods are meant to be invoked by the container, also as the main step of a *request processing pipeline*.

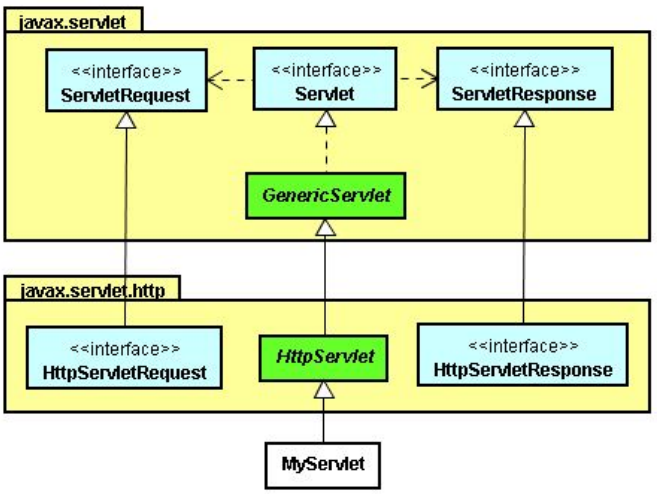
The container is responsible for managing the lifecycle of servlets, and mapping a URL to a particular servlet.

Servlets and More

What's a Servlet?

A class that implements the `javax.servlet.Servlet` Interface...

In practice: our custom servlets have to extend either `GenericServlet` or, usually, `HttpServlet`

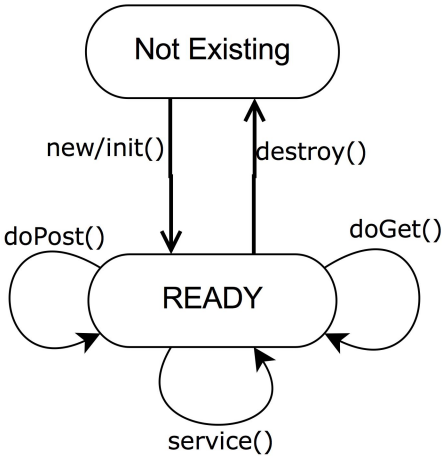


Servlet Life-cycle

The entire lifecycle of a servlet *is managed by the container.*

Upon a REQ, `service()` is invoked; possibly, de-multiplexing to `doPost()`, `doGet()`, etc.

BTW: How many instances of a Servlet class (in a single webapp)?



Handling of REQ/RESPONSE

For a simple handling of HTTP requests/responses, they are represented by corresponding objects, so that containers can easily take care of them throughout their processing pipeline.

According to Servlets specification, such objects must implement:

1. `javax.servlet.http.`**`HttpServletRequest`**
2. `javax.servlet.http.`**`HttpServletResponse`**

This approach is much more convenient that CGI!

Parameters in REQ

Regardless of GET/POST mode, parameters can be accessed within the request object using the methods (`ServletRequest` interface):

- `getParameterNames()` provides the names of the parameters
- `getParameter(name)` returns the value of a named parameter
- `getParameterValues()` returns an array of all values of a parameter if it has more than one values.

Information in HTTP headers is retrieved in the same way.

Acting on Response

Two ways of inserting data in the body:

- **`getWriter()`** returns a `PrintWriter` for sending text data
- **`getOutputStream()`** returns a `ServletOutputStream` to send binary data

Both need to be closed after use!

Setting headers' content: specific methods, e.g. **`setContentType(<MIME_type>)`**

Mapping URLs onto Servlets

Specified in the webapp *deployment descriptor* **`web.xml`** in two steps:

servlet-name ⟶ servlet-class + servlet-name ⟶ url-pattern

Alternative way: directly in the code, using *annotations*:

```
@WebServlet(
    name = "MyAnnotatedServlet",
    urlPatterns = {"/foo", "/bar", "/pippo*"}
)
public class MyServlet extends HttpServlet {
    // servlet code
}
```


Session Tracking

A mechanism to maintain state information *for a series of requests*,

- along a period of time,
- from the same client.

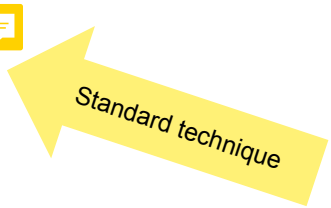
Session are represented through specific objects (interf. `HttpSession`) shared across all the servlets accessed by the same client.

Programmatically, session objects can be obtained from requests:

```
HttpSession mySess = req.getSession(boolean create);
```

Session Tracking Techniques

- Using session cookies
- Hidden fields
- URL rewriting



Sessions: Keeping State Information

Session objects are ordinarily used to *keep state information* throughout the session.

- `public void setAttribute(String name, Object obj)`
- `public Object getAttribute(String name)`

Optionally, sessions can be invalidated:

- `mySess.invalidate()` i.e., immediately
- `mySess.setMaxInactiveInterval(int interval)` i.e., max interval between successive requests (default value defined in web.xml)

Servlet Example

```
import ...

public class HelloWorld extends HttpServlet {
   private String msg;

   public void init() throws ServletException {
      msg = "Hello World";
   }

   public void doGet(HttpServletRequest req,HttpServletResponse res)
                        throws ServletException, IOException {
      res.setContentType("text/html");
      PrintWriter out = res.getWriter();
      out.println("<h1>" + message + "</h1>");
   }
}
```


REQ Processing: Servlet Filters

Upon receiving a REQ, often some typical actions have to be undertaken.

E.g. recording, IP logging, input validation, authentication check, encryption/decryption, etc.

Each action can be carried out by a *pluggable* server-managed component named *servlet filter*; its application depends on the relative *url pattern*.

Adv: separation of concerns (different pieces of SW/different tasks), easy maintenance

APIs: in javax.servlet, interfaces **`Filter`**, **`FilterChain`**, **`FilterConfig`**

Implementation of a Servlet Filter

A custom servlet filter must extend **`Filter`** → implement

- init(), destroy()
- doFilter(ServletRequest req, ServletResponse resp, FilterChain chain)

The body of doFilter() implements the action to be undertaken.

To pass the REQ to the next component → chain.doFilter(req, resp);

Mapping of filters: as for servlets, in web.xml or using annotations, e.g.

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


Structuring Servlet Apps

Servlets are a basic technology that imposes no constraint on how a whole application should be structured.

Risk: "Magic Servlet" antipattern, where issues about different aspects of the application are dealt within the same method.

Solution: define specific roles, so that a specific task/concern refers to a distinct software component.

Consequence: introduction of further levels of abstraction for a well-structured development of web apps.

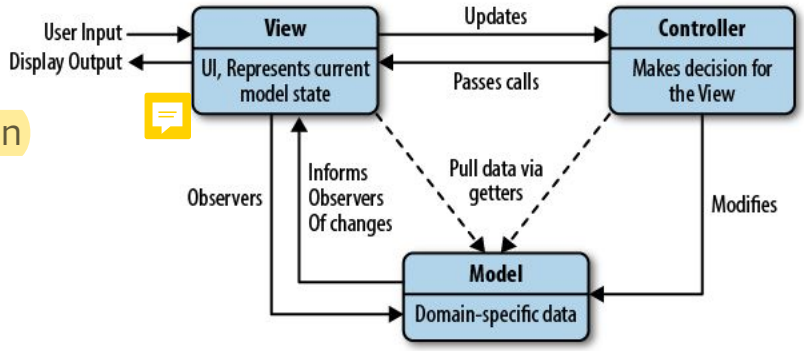
MVC - Model View Controller

Originally conceived for GUIs

Model: manages data

View: handles interaction with the user

Controller: coordinates interactions



Template Systems & Engines

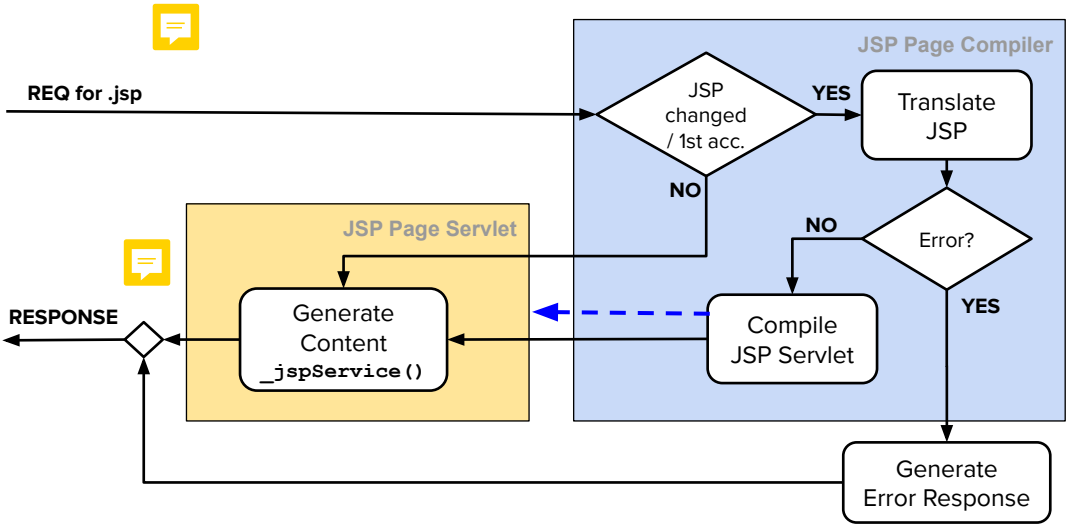
Servlets cannot be easily maintained because of the tight coupling of presentation (HTML) and business logic (in Java).

Proposed solution: adoption of a *server-side template system*, with templates in HTML, and logic embedded by using special tags.

Such templates have to be processed by a *template engine*, getting to the actual dynamic content.

Examples: ASP, **JavaServer Pages**, PHP, etc.

Java Server Pages - Processing



JSP - Basic Scripting

Template: an HTML document. Scripting can be added in several ways; the most direct one is through *scriptlets*, i.e. java code inside the delimiters `<%` ... `%>`. Other possibility: expressions, `<%=` `an_expression` `%>`, etc.

The code within the scriptlet tags goes into the `_jspService()` method.

```
<p>Listing of the first natural numbers:</p>
<% for (int i=1; i<4; i++) { %>
    <p>This number is <%= i %>.</p>
<% } %>
<p>OK.</p>
```

JSP - Implicit Objects

Some objects are made available to the JSP by the environment:

- **request**, **response**
- **out** - PrintWriter obj to write in the response body
- **session** - to access the session obj
- **application** - to access ServletContext obj (info sharing across JSPs)
- **page** (a synonym for this)
- others…

Implicit objects can directly be used in the JSP scripts in developing business logic.

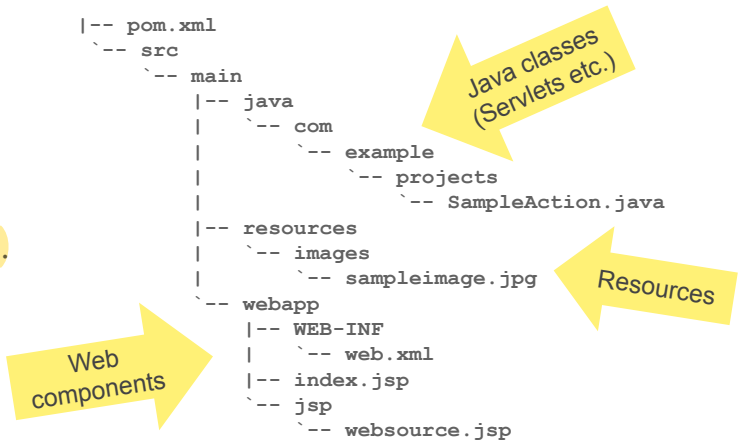
Better Structuring of JSPs

To apply the “separation of concerns principle”, other mechanisms can be used inside JSPs:

- **Java Beans** - basic Java components (classes)
- **JSTL** - library of standard tags to support typical control flow
- **EL (Expression Language)** - makes it possible to easily access application data stored in JavaBeans

Developing WebApps: Project directories

In Maven, a project
for a web application
is structured according to
a standard directory layout,
so to automatize
all the packaging operations.



JSP and MVC

JSPs are typically used to support the “view” part of the MVC pattern.

The abstraction provided by JSP is sufficient
only for relatively small web applications.

As the complexity grows up, it can be controlled by making use
of more integrated approaches that make transparent
the underlying used technologies, like Servlets and JSPs.

The resulting systems are known as “web frameworks,”
and most of them are designed taking MVC as the reference pattern.

MVC Web Frameworks

Out of the most popular frameworks in the community of developers of Java Web/Enterprise Applications, we can recall:

- **JavaServer Faces (JSF)**, part of JEE 
- **Spring**  , **Struts** 

MVC frameworks are popular also with other languages, e.g.:

- **ASP.NET (MVC)** - successor of ASP, with C# and CLI languages
- **Django** - with Python, for complex web apps
- **Play** - with Scala (Akka)

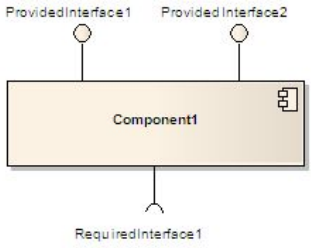
Enterprise Applications

Problems with Plain Objects

Use of plain objects in an enterprise app shows some problems:

- No deployment transparency
- Implicit dependencies (must be made explicit!)

Solution: from concept of object ➔ **component**



Provided/required interfaces:
contract between different components

Need for supporting exploitation of components!

Application Servers

Need to simplify the programming model:
the programmer must focus on business logic,
without spending time on distributed computing issues.

Architectural support to separation of concerns ➔
Container pattern to manage components

Middleware solution: **Application server**

Addressed services: component lifecycle management, resource management, persistence, transactions, concurrency, security, etc.

Java Application Servers - Examples

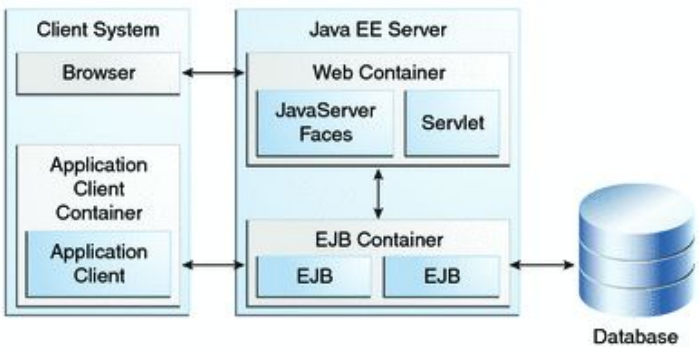
- **Glassfish** (JEE reference implementation)

→ payara

- **JBoss** EAP and subsequently **WildFly**

- IBM **WebSphere** - historical AS
- Oracle **WebLogic** - other historical AS

Java Application Servers - Overview



Enterprise Java Beans

What's an EJB?

Answer: "A *server-side component* that encapsulates the business logic of an application."

The EJB container is responsible for managing them, and it provides system-level services to enterprise beans.

Using EJBs, client modules become thinner.

Reusability: new applications can be built from existing EJBs.

Types of EJBs

Session Beans - accessible either by a *local* or a *remote* interface.

1. **Session Beans** (not persistent)
   - 1.1. Stateful Session Beans
   - 1.2. Stateless Session Beans
   - 1.3. Singleton Session Beans
2. **Message Driven Beans**

Perform work for their clients, encapsulating business logic

Message Driven Beans - business objects whose execution is triggered by messages instead of by method calls.

Stateful Session Beans

State: values for its instance variables.

Client: *code that holds the (remote) reference to a single instance.*
The bean session does not necessarily corresponds to the "web session," if it is present in the enterprise app.

State typically depends on the client-EJB interaction.

The state is retained for the duration of the client-bean session. If the client removes the bean, the session ends and the state disappears.

Stateless Session Beans

Basic idea: No support to conversational state with the client.

The client may change the bean state, but it is not guaranteed to be retrieved on the next invocation - these beans are pooled!

Offer better scalability for apps with a large number of clients.

Typically, an app requires fewer stateless session beans than stateful session beans to support the same number of clients.

A stateless session bean *can implement a web service*, but a stateful session bean cannot (because of idempotence issues).

Singleton Session Beans

State: unique, shared across the application (not *conversational*); a singleton session bean is accessed *concurrently* by clients.

Singleton session beans maintain their state between client invocations, but are not required to maintain their state across server crashes or shutdowns.

Singleton session beans can implement web service endpoints.

EJB Interfaces

Session EJBs may implement a local/remote interface.

The interface represents the contract with the client, and it is usually defined independently of the EJB implementation.

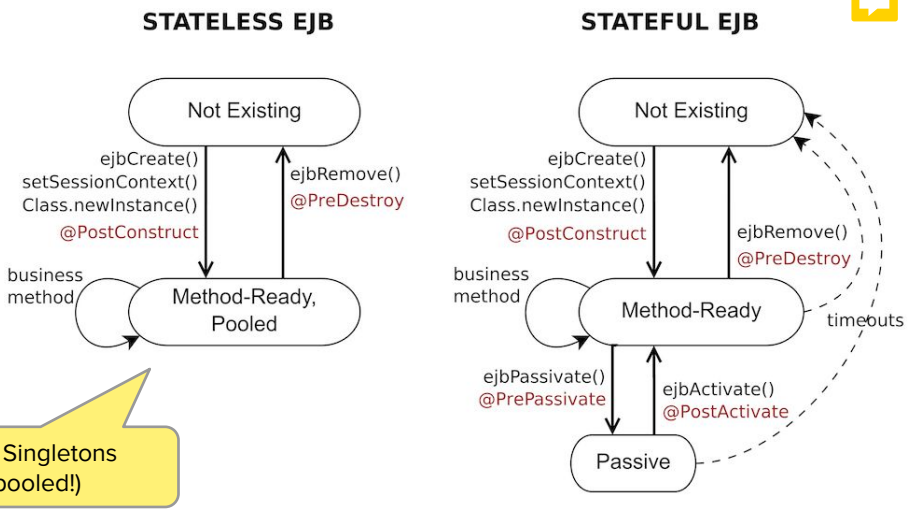
For the sake of making coding simpler, annotations (in `javax.ejb`) are provided to 1) specify the nature of an interface,

`@Local` `@Remote`

and 2) the EJB type for an implementation class:

`@Stateless` `@Stateful`

Lifecycles of Session EJBs

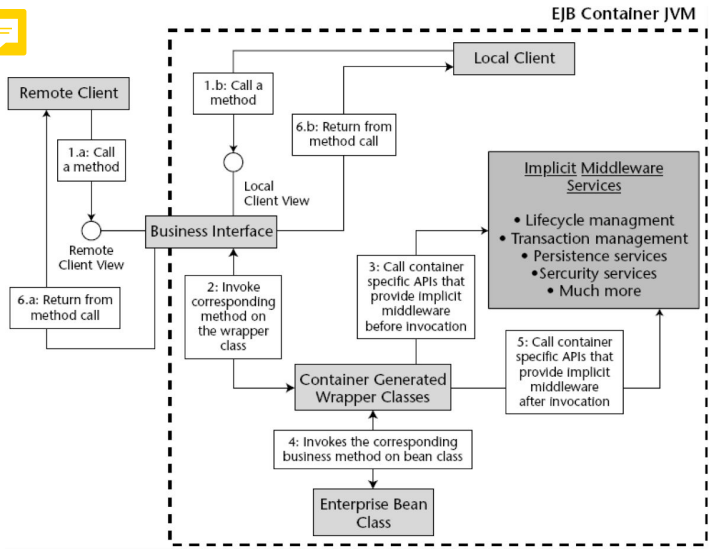


Ditto for Singletons
(not pooled!)

Call of an EJB

The client obtains a reference to an EJB instance through either *dependency injection*, using Java annotations, or *JNDI lookup*.

Other possibility:
Not a business interface, but using a no-interface view.



JNDI

Directory Services

To make easier the access to important resources over a network, *directory services* keep matches between names and network addresses (of different kinds);
moreover, other information can be associated with each match.

Popular standard: LDAP (Lightweight Directory Access Protocol, see RFC 4511), often used also as a repository to keep username/password pairs.

Differently by RDBMS, directory services are specialized in handling the typical structure of network resources.

Directory Services

To make easier the ac ss to important resources over a network, directory services keep hing between names and network addresses (of different kin
moreover, other infor e match.

Popular standard: LD Protocol), often used also as a sword pairs.

Differently by RDBMS, directory services are specialized in handling the structuring typical of network resources.

Essential components for Distributed Operating Systems

What's JNDI?

The Java Naming and Directory Interface (JNDI) is a Java API for a directory service to discover/lookup data/objects via a name.

A JNDI service is typically implemented in any JEE AS.

Names are organized into a hierarchy: e.g., *com.mysite.ejb.MyBean*
A name is bound to an obj either directly or via a reference.

The JNDI API defines a *context* that specifies where to look for an object. The **initial context** is typically used as a starting point.

Basic JNDI Lookup

First step: obtain the initial context (ideal root of the hierarchy)

```
Hashtable contextArgs = new Hashtable();
contextArgs.put( … ) // insert all req. params to locate the service
Context myContext = new InitialContext(contextArgs);
```

Then: lookup via InitialContext

```
MyBean myBean = (MyBean)  myContext.lookup("com.mysite.ejb.MyBean");
```

ATTENTION: take care of standard naming conventions for EJBs!

JNDI Names for Session EJBs

Scope	Name Pattern
Global	java:global[/<app-name>]/<module-name>/<bean-name>[!<fully-qualified-interface-name>]
Application	java:app/<module-name>/<bean-name>[!<fully-qualified-interface-name>]
Module	java:module/<bean-name>[!<fully-qualified-interface-name>]

Context/Dependency Injection

CDI - Context & Dependency Injection

Fosters a more effective interaction between web/business tiers. It promotes loose coupling and strong typing.

Contexts: Ability to bind the lifecycle and interactions of stateful components to well-defined but extensible lifecycle contexts.

Dependency injection: Ability to inject components into an application in a typesafe way, including the ability to choose *at deployment time* which implementation of a particular interface to inject

CDI - Context & Dependency Injection

General design principle:
Inversion of Control.

Binding is performed at runtime by the container,
substituting explicit lookup

© A.Bechini 2020

A. Bechini - UniPi

CDI for EJBs by Annotations

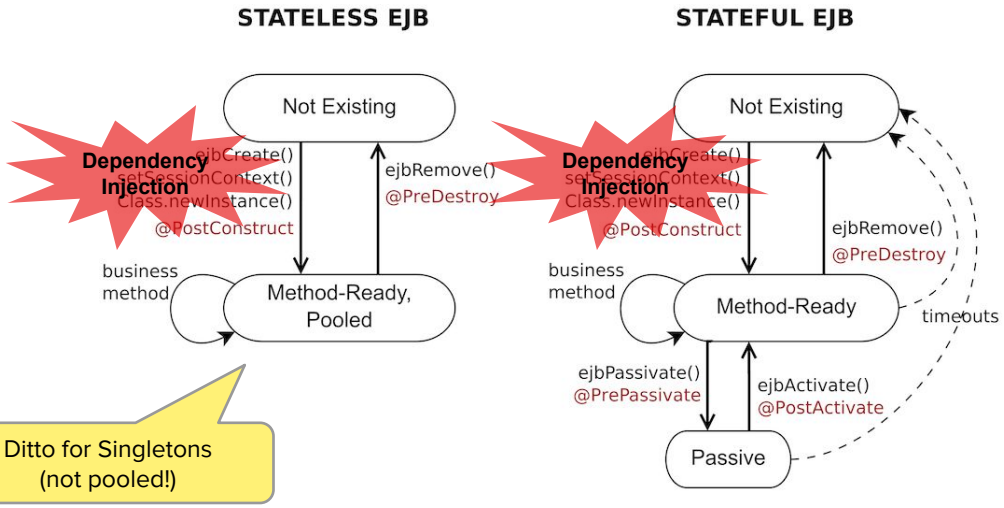
Example:

```
@WebServlet
public class MyServlet extends HttpServlet {
   @EJB
   MyBean myBeanInstance;
   ...
}
```

It's up to the AS to put here the correct reference to the corresponding managed EJB

How to find the correct matching? Apply the *Convention Over Configuration* principle

CDI and EJB Lifecycle



Ditto for Singletons (not pooled!)

Annotations vs. Deployment Descriptor

Up to EJB 3.0, a *deployment descriptor* for EJBs was mandatory: it had to specify bean characteristics in `ejb-jar.xml`

Since EJB 3.0 such characteristics can be specified via annotations, so the EJB deployment descriptor is not required any more.

Both can co-exist, possibly providing complementary information; in case of conflicts, priority is given to indications in the deployment descriptor.

Finally: Message-Driven Beans

A msg-driven EJB is designed to perform a task **asynchronously**.

It is invoked by the EJB container upon receiving a message from queue or topic: typically it acts as a *listener* for JMS messages.

All instances of an MDB are equivalent: the container can assign a msg to any instance ➔ *MDB pooling for concurrent processing*.

Notes:

- One single MDB can process msgs from 1+ clients.
- MDB are stateless.

Finally: Message-Driven Beans

An EJB designed to perform a task **asynchronously**.

It is invoked by the EJB container upon receiving a message from queue or topic: typically it acts as a *listener* for JMS messages.

All instances of an MDB are equivalent: the container can assign a msg to any instance ➜ *MDB pooling for concurrent processing*.

Notes:

- One single MDB can process msgs from 1+ clients.
- MDB are stateless.

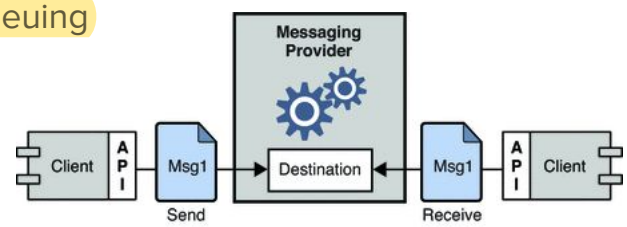
Java Message System

MOM - Message Oriented Middleware

MOM is an infrastructure to support **indirect and asynchronous** communication across components of a distributed application.

It makes transparent networking and communication protocol details, addressing HW/SW heterogeneity → use of **msg brokers**

- Asynchronicity - via message queuing
- [Intelligent][Routing] - different routing specifications
- [Msg Transformation] - via specific tools



Images from official Oracle GlassFish docs

JMS - Basics

Message-oriented technologies rely on an *intermediary component*: senders may ignore many details about receivers (even identities!).

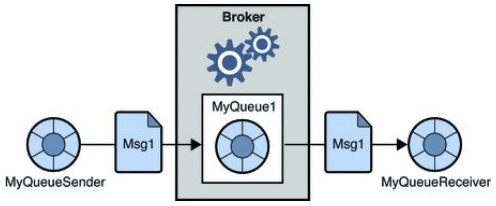
This is an approach suitable to integrate heterogeneous systems.

Group communication (multi/broad-cast), membership management.

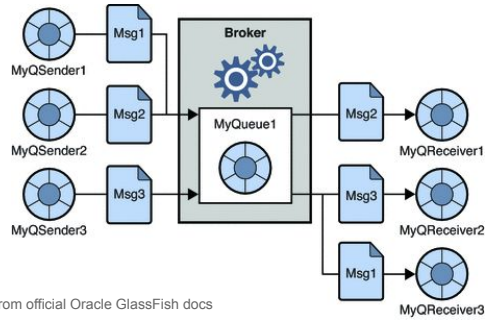
Models:

- **Point-to-point** (message **queues**)
- **Publish-subscribe** (**topics**)

JMS Point-to-Point Messaging



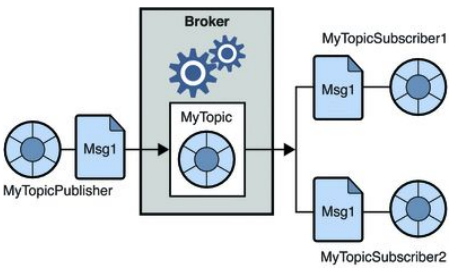
Simple scenario:
1 sender, 1 receiver

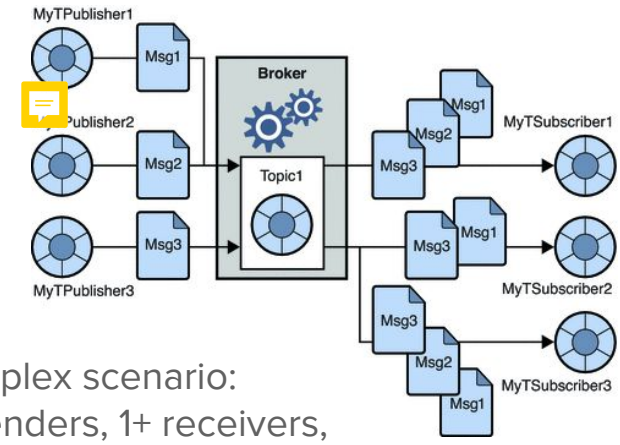


Complex scenario:
1+ senders, 1+ receivers,
with possibly shared connections.
No hypotheses on the order
msgs are consumed.

Images from official Oracle GlassFish docs

JMS Publish/Subscribe Messaging





Simple scenario:
1 sender, 1+ receivers

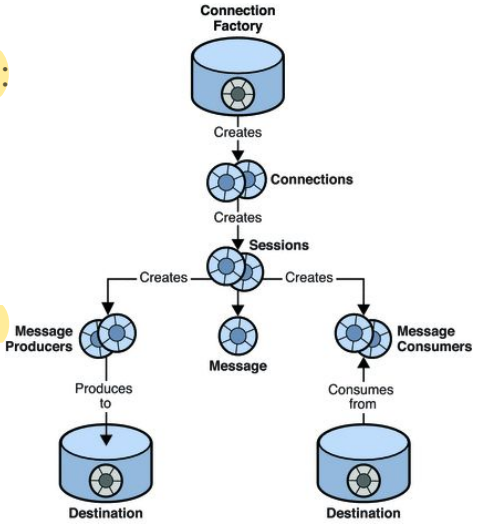
Complex scenario:
1+ senders, 1+ receivers,
with possibly shared connections.

Images from official Oracle GlassFish docs

JMS Programming Objects

Objects used to implement JMS messaging:

- connection factory → connections
- connection - comm. channel to the broker
- session - client/broker conversation
- producer - obj to send msgs to a destination
- consumer - obj to get msgs from a destination
- message - made of header, properties, body
- destination - represents phys. dest.
- connection factory → connections



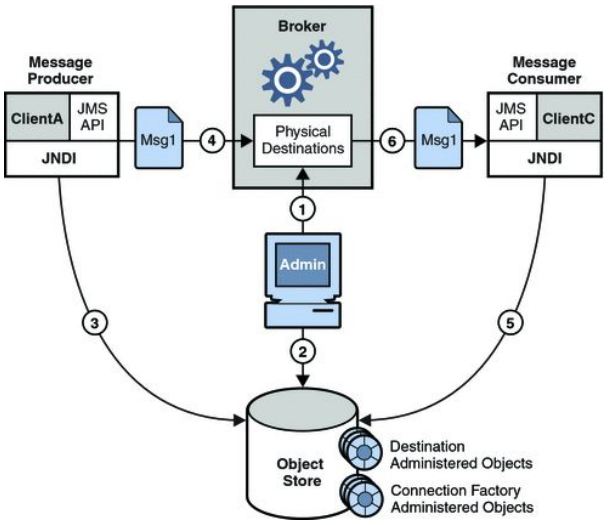
Images from official Oracle GlassFish docs

JMS - Administered Objects

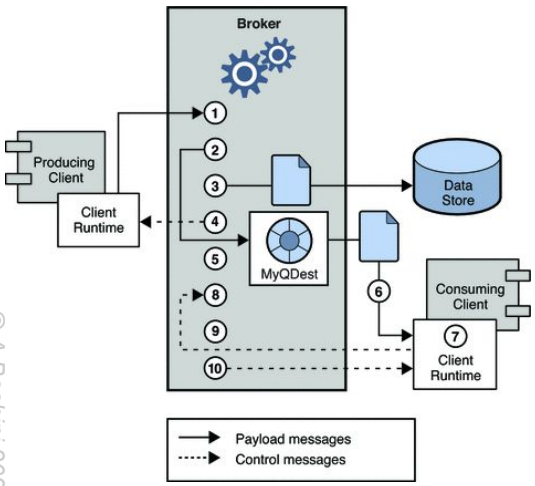
- connection factories
- destinations

are typically created by adm tools on the Application Server;

their use in JMS is possible through JNDI lookup of the corresponding "administered objects."



Message Delivery Steps



1. msg delivery over conn. to the broker
2. broker reads msg and place it in its dest.
3. (persistent msg in data store)
4. broker acks back client
5. broker determines routing
6. broker writes out msg to dest. conn.
7. client's runtime delivers msg
8. client's runtime aks back
9. (broker deletes pers. msg)
10. broker to cl. runtime: ack processed

Not Only JMS Deserves Mention...

Message Brokers:

- Apache ActiveMQ
- RabbitMQ
- ZeroMQ (no broker, only library)

Standards:

- AMQP (broker: StormMQ)
- HLA IEEE 1516
- MQTT (MQ Telemetry Transport), used for IoT