Enterprise Apps and Jakarta EE (JEE)

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

alessio.bechini@unipi.it

Outline

Web Applications, Enterprise Applications, and Jakarta EE (formerly JEE)

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

General Ideas

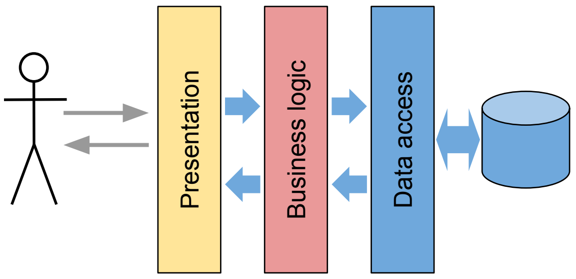
3-Tier Architecture

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.

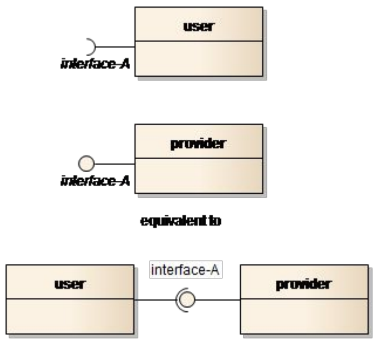


Use of Containers in Middleware

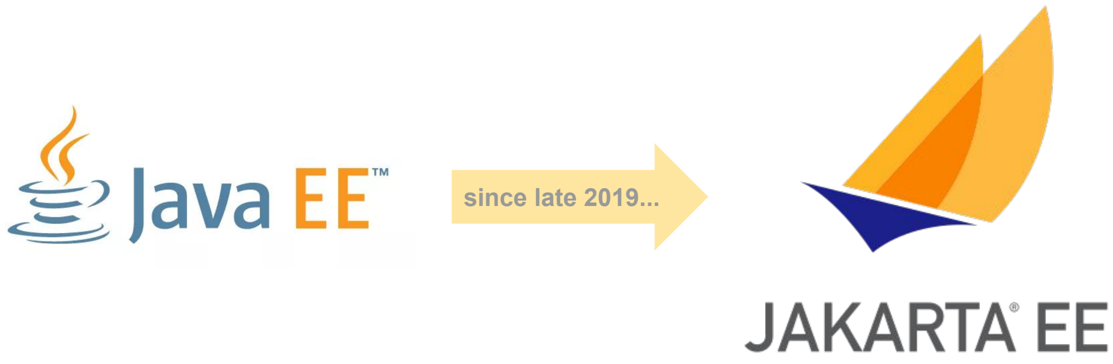
At the middleware level, components can be organized in *containers*, and components managed within containers.

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...



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.
C *lly* plac l dir
Parameters ar ssed via environme vars ar

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

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

A Perl script

`PATH_INFO`

`QUERY_STRING`

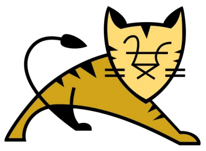
Beyond CGI-Scripts

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 Jakarta EE: **Servlets**, and related technologies

The Tomcat Web Server



Basic internal components:

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

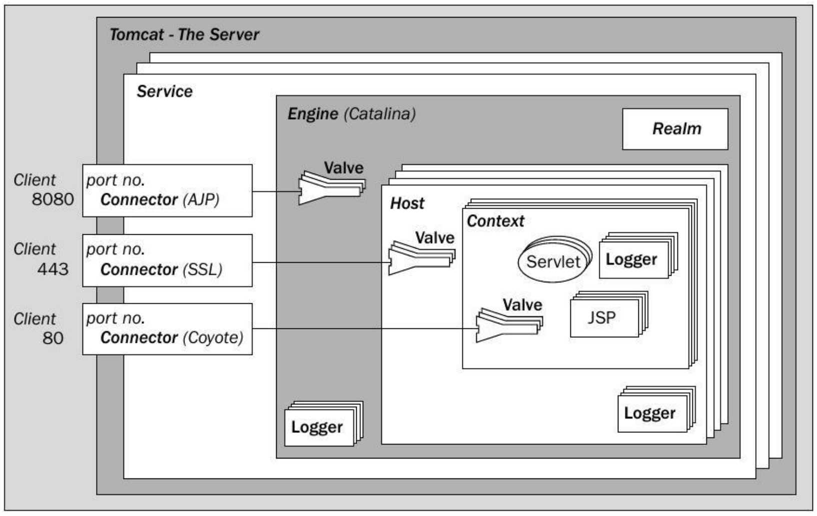
"Connector" - component to communicate w/ clients

"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 specific 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

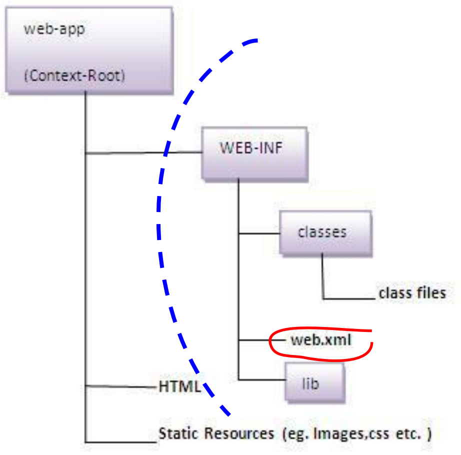
In the filesystem, one directory (*appBase* - by default, “webapps”) is used to keep all the material for web applications.

Each single 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

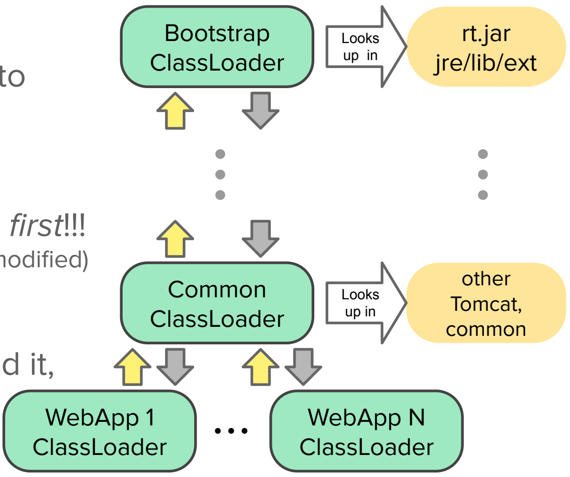


Not-interfering Contexts

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

!!! *A WebApp Classloader doesn't delegate first*!!!
(only for Java core libraries... but such a behavior can be modified)

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



ClassLoaders, In Depth

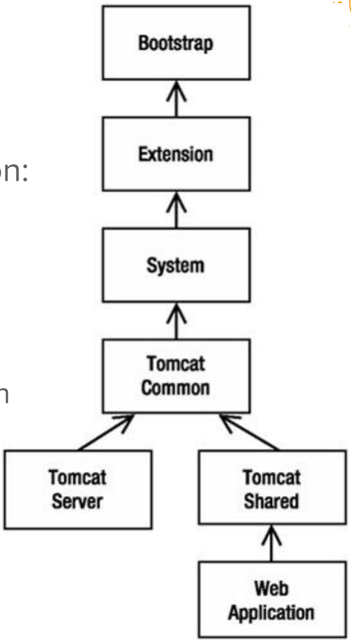
It is possible to specify a more advanced configuration:

Tomcat Common: looks in
CATALINA_HOME/common/**lib**/ — For .jar files
CATALINA_HOME/common/**classes**/ — For .class files

Tomcat Server: for server-specific classes

Tomcat Shared: for classes shared across webapps; looks in
CATALINA_HOME/shared/lib
CATALINA_HOME/shared/classes

Tomcat Webapp: looks in
CATALINA_HOME/webapps/[WAName]/WEB-INF/lib
CATALINA_HOME/webapps/[WAName]/WEB-INF/classes



Servlets and More

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!***

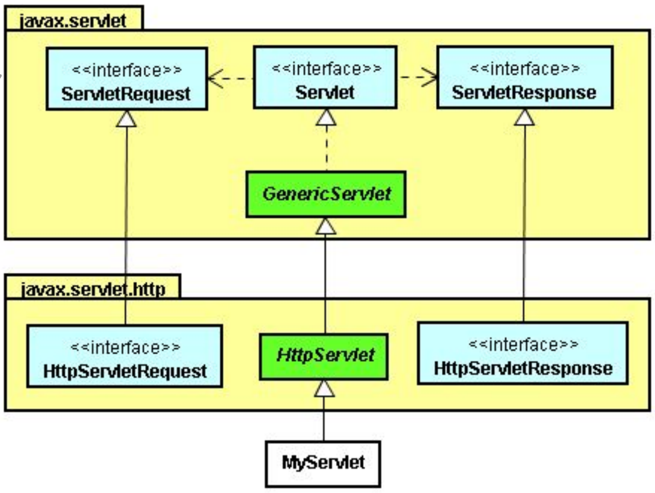
Servlets are kept in a container.
Their methods are meant to be invoked by the container,
also as the main step of a *request processing pipeline*.

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

What's a Servlet?

A class that implements `jakarta.servlet.Servlet` (an interface)…

Recently, `javax` ➜ `jakarta`


javax.servlet
<<interface>> ServletRequest
<<interface>> Servlet
<<interface>> ServletResponse
GenericServlet
javax.servlet.http
<<interface>> HttpServletRequest
HttpServlet
<<interface>> HttpServletResponse
MyServlet

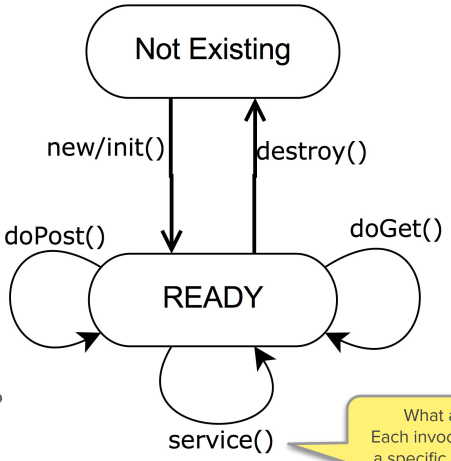
In practice: our custom servlets have to extend either `GenericServlet` or, usually, `HttpServlet`

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)?



What arguments? Each invocations refers to a specific client request…

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. `jakarta.servlet.http.`**`HttpServletRequest`**
2. `jakarta.servlet.http.`**`HttpServletResponse`**

Previously, `javax`

This approach is much more convenient that CGI!

request/response objects are passed **by the container** to the servlet service method!

Reading REQ Parameters

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 of the response message:

- **`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.

Sessions are represented through specific objects (interf. `HttpSession`) that *have to be shared* across all the servlets accessed by the same client.

Programmatically, session objects can be obtained from requests:

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

NOTE: the type is an interface; What about the actual class?

Internal Handling of Session Objects

A specific `HttpSession` object has to be used for every ongoing session.

The webserver marks each new session with a unique “SessionID”;
session objects can be kept in an associative array, with SessionIDs as keys.

SessID1	SessOBJ1
SessID2	SessOBJ2
SessID3	SessOBJ3
	⋮

Any REQ message that belongs to a certain session,
must be associated to the relative SessionID:
this makes the container able to retrieve
the correct session object from any servlet.

In Tomcat, each distinct app (context) has a “manager” object in charge of handling session objects

Session Tracking Techniques

How to associate a REQ with the relative session?
This info must be present in REQs from the client.

i.e. via
its SessionID

Possible techniques for *session tracking*:

- Using session cookies
- Hidden fields
- URL rewriting

Standard technique

Sessions: Keeping State Information

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

"Conversational"
state info

- `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.

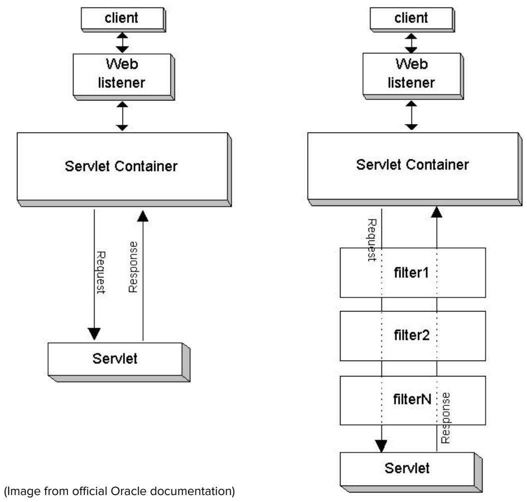
E.g. recording, IP logging, input validation, authentication check, encryption/decryption, etc.

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

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

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

Servlet Filters: Scenario



(Image from official Oracle documentation)

Pay attention: whether applying or not a filter to a REQ depends on the match between the REQ URL and the pattern specified for the filter!

Moreover: Filters can be chained!

Implementation of a Servlet Filter

A custom servlet filter must extend **`Filter`** → and implement the methods

- 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 {
    ...
```

Making Servlets Collaborate

It is possible to make servlets/resources collaborate for the construction of the response content via the **`RequestDispatcher`** interface.

A `RequestDispatcher` can be obtained from the `ServletRequest` object by the factory method `request.`**`getRequestDispatcher`**`(<targetResource>)`

The argument identifies the servlet/resource we want to collaborate with.

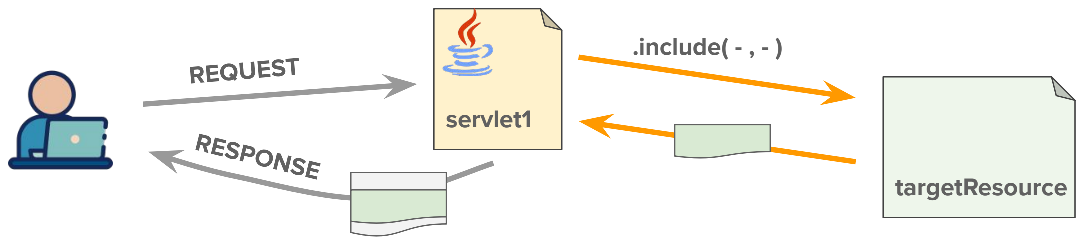
Two possible types of collaborations:

- Obtaining content to include
- Delegation of request handling

Inclusion with Servlets

The outcome produced by another resource (servlet, jsp, html) can be included in the response getting assembled by the current servlet:

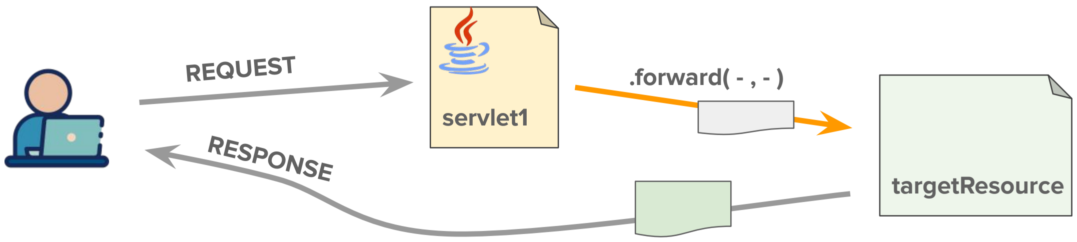
```
rd = request.getRequestDispatcher(<targetResource>);
rd.include(request, response);
```



Forwarding in Servlets

Delegation to another resource (servlet, jsp, html) of finishing the response assembly can be obtained this way:

```
rd = request.getRequestDispatcher(<targetResource>);
rd.forward(request, response);
```



Sharing Info across Servlets

As servlets can possibly collaborate, they may need to share information.

Information sharing can be obtained with standard objects created by the container, and made available to servlets: “scope objects”

They are equipped with the methods `setAttribute/getAttribute`.

Among them:

- Request relative to one http transaction (*request scope*)
- The session object (*session scope*)
- The ServletContext obj, one per hosted web application (*web scope*)

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.

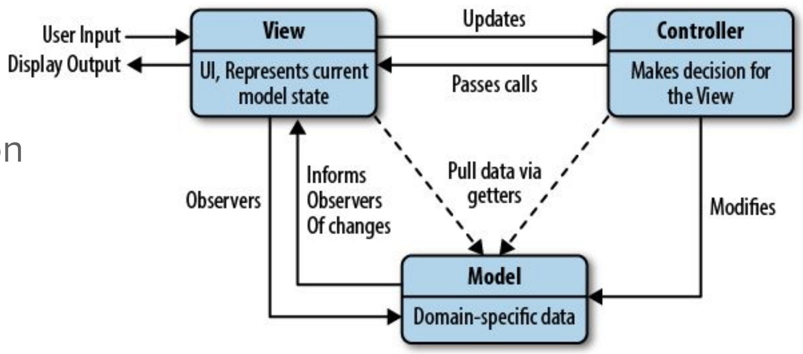
MVC - Model View Controller

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 web 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, **JakartaServer Pages**, PHP, etc.

Formerly **Java**Server Pages

Aside: Web Template Systems

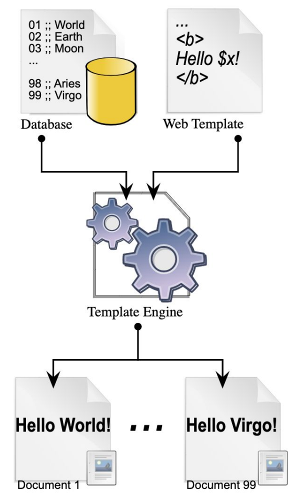


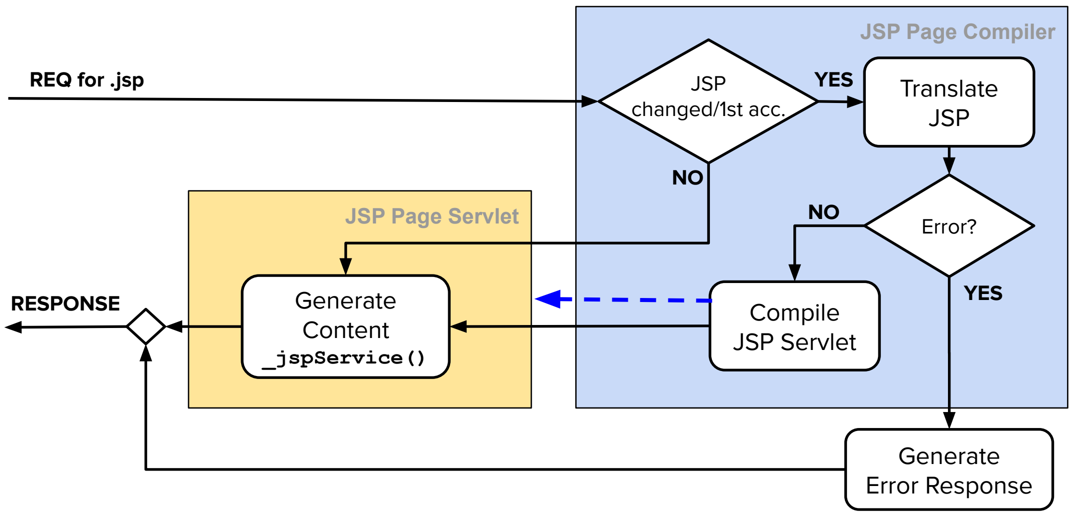
Image from WikiCommons

Many Web Template Systems have been developed to generate custom web pages.

The Template Engine is in charge of analyzing the "placeholder elements" in the template, and getting to the final document(s) on the basis of a collection of elements, possibly in the form of a dataset.

This processing pattern can be applied at different level of the overall web content production system.

Jakarta 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.

```
|-- pom.xml
 `-- src
     `-- main
         |-- java
         |   `-- com
         |       `-- example
         |           `-- projects
         |               `-- SampleAction.java
         |-- resources
         |   `-- images
         |       `-- sampleimage.jpg
         `-- webapp
             |-- WEB-INF
             |   `-- web.xml
             |-- index.jsp
             `-- jsp
                 `-- websource.jsp
```

Java classes (Servlets etc.)

Resources

Web components

Deploying/Packaging WebApps

The standard deployment
of a Jakarta web application
requires the web module
to be organized as shown
aside.

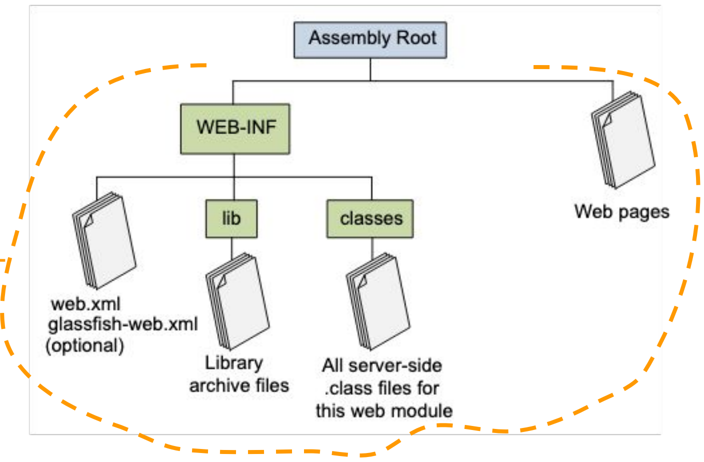


Image from "The Jakarta EE Tutorial"

Possibly packed
in a **.war** file
to make distribution easier

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:

- **JakartaServer 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) **Node.js (express.js)**

Enterprise Applications

Problems with Plain Objects

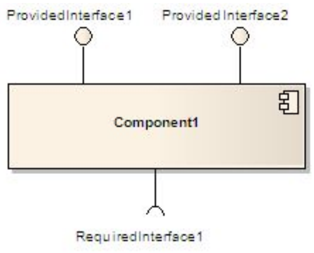
Use of plain objects in an enterprise app shows some problems:

- 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

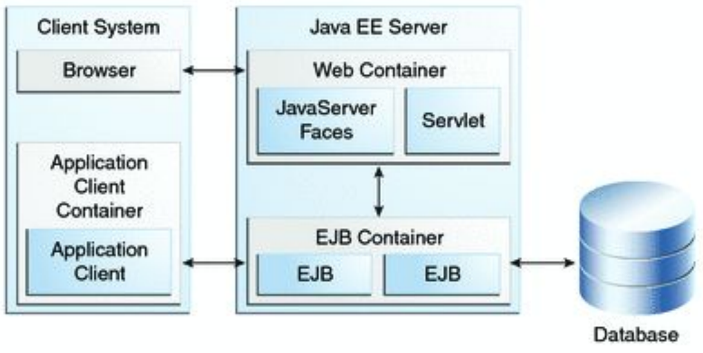
Middleware solution: **Application server**

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.

Mandatory for EJB < v3.0, optional later

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,

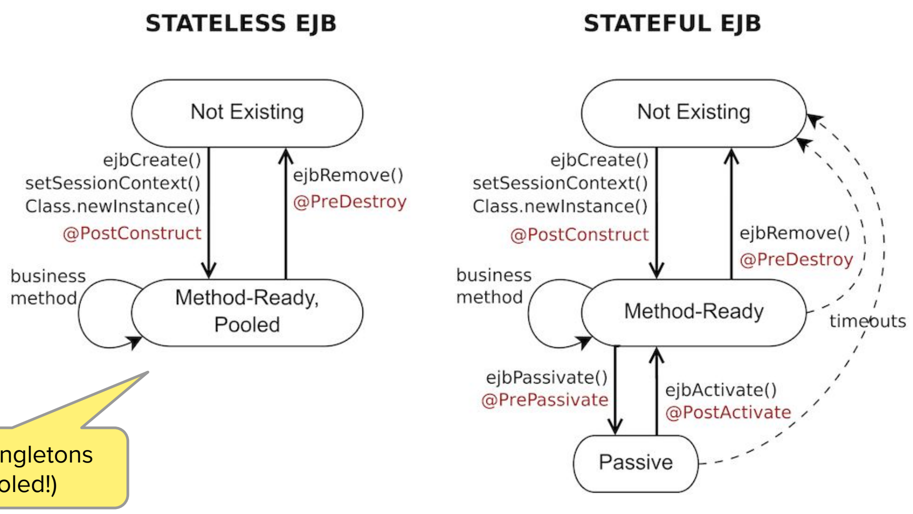
```
@Local   @Remote
```

and 2) the EJB type for an implementation class:

```
@Stateless   @Stateful
```

Recently, `javax` ➔ `jakarta`

Lifecycles of Session EJBs

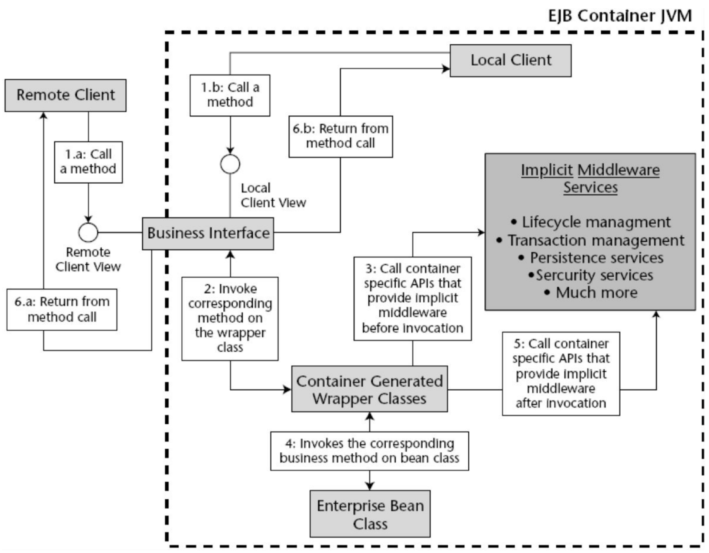


Ditto for Singletons
(not pooled!)

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.


EJB Container JVM
Local Client
Remote Client
1.b: Call a method
6.b: Return from method call
1.a: Call a method
Local Client View
Implicit Middleware Services
• Lifecycle managment
• Transaction management
• Persistence services
•Sercurity services
• Much more
Business Interface
Remote Client View
2: Invoke corresponding method on the wrapper class
3: Call container specific APIs that provide implicit middleware before invocation
6.a: Return from method call
5: Call container specific APIs that provide implicit middleware after invocation
Container Generated Wrapper Classes
4: Invokes the corresponding business method on bean class
Enterprise Bean Class

Session EJBs and Asynchronous Calls

Methods of session beans can implement asynchronous computations, making use of classical `Future<V>` returned values.

To let the system manage this possible behavior, “asynchronous” methods must be annotated with `@Asynchronous`.

The returned `Future<V>` object let us get control over the computation, and retrieve the result by means of the usual Java SE features.

Finally: Message-Driven Beans

EJBs designed to perform tasks **asynchronously**,
i.e. at the occurrence of a given event (an incoming message).

We’ll discuss them later, after introducing other JEE features.

JNDI

Directory Services

To make it 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 it easier the ess to important resources over a network, *directory services* keep hes between names and network addresses (of different kin moreover, other infor ch match.

Essential components for Distributed Operating Systems

Popular standard: LD , see RFC 4511), often used also as a sword pairs.

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

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.

Any JEE Application Server typically includes a JNDI service.

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

Symbolic name

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

(Remote) object

Downcast

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

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

Fosters a more eff   usiness tiers.
It promotes lo

Contexts: Ability   of stateful
components to w   ontexts.

Dependency inj   to an
application in a ty   choose
at deployment time which implementation of a particular interface
to inject

General design principle:
Inversion of Control.

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

CDI for EJBs by Annotations

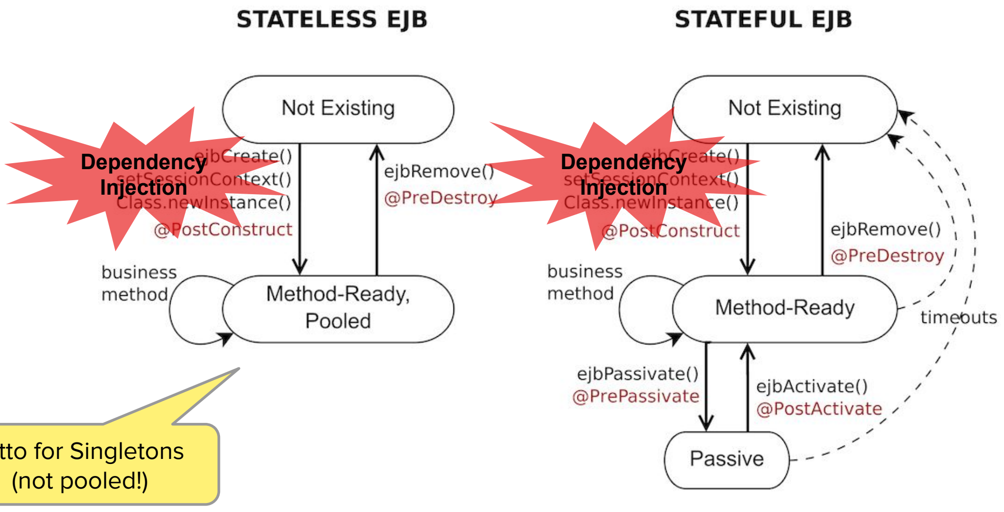
Example:

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

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

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

CDI and EJB Lifecycle



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.

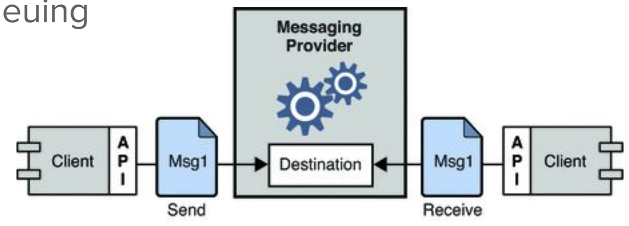
Java Message System

MOM - Message Oriented Middleware

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

It makes transparent the 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!).

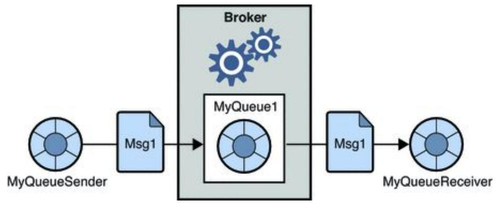
This is an approach suitable to integrate heterogeneous systems.

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

Models:

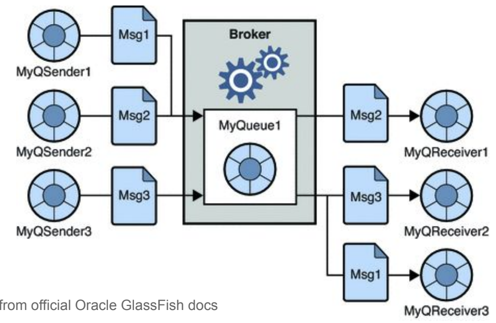
- **Point-to-point** (message **queues**)
- **Publish-subscribe** (**topics**)

JMS Point-to-Point Messaging



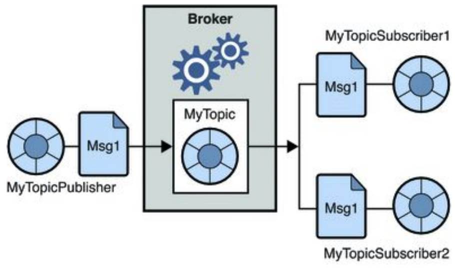
Simple scenario:
1 sender, 1 receiver

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

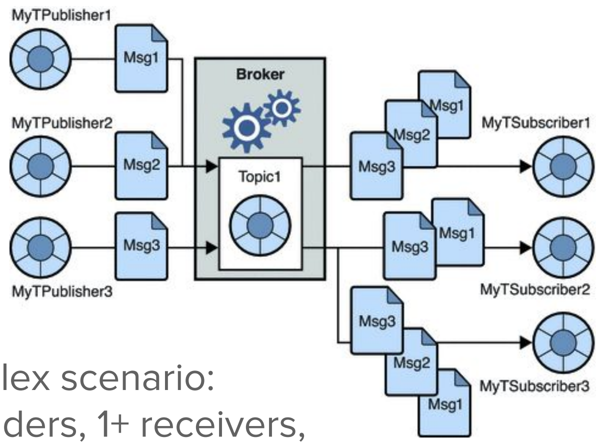


Images from official Oracle GlassFish docs

JMS Publish/Subscribe Messaging



Simple scenario:
1 sender, 1+ receivers



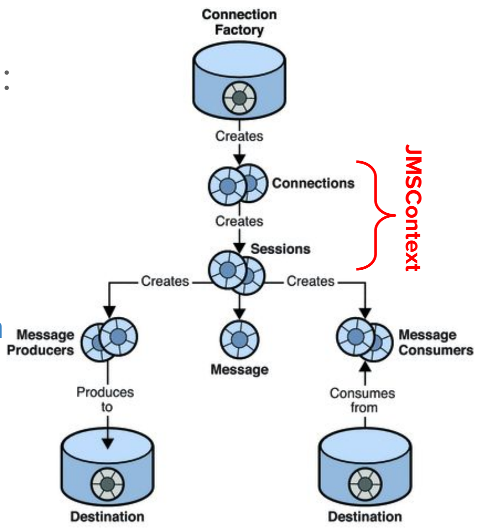
Complex scenario:
1+ senders, 1+ receivers,
with possibly shared connections.

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



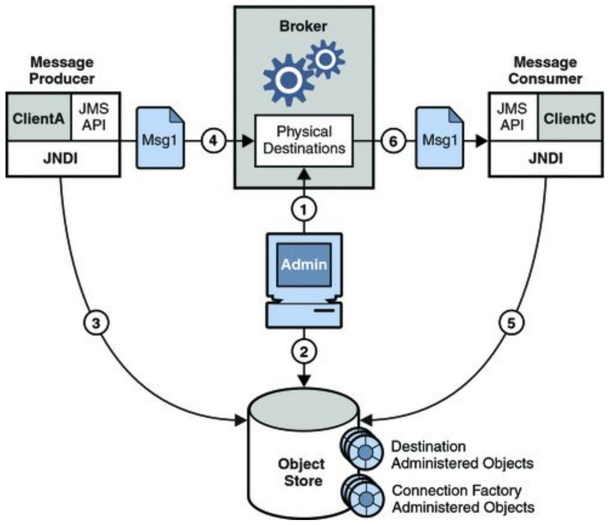
Images from official Oracle GlassFish docs

JMS - Administered Objects

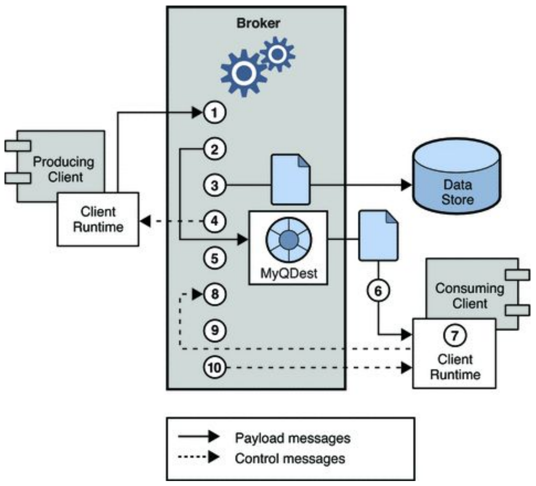
- connection factories
- destinations

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