Enterprise Apps and Jakarta EE (JEE)

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

alessio.bechini@unipi.it

- Servlet, slide 10a
- EJB, slide 29a

Outline

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

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

General Ideas

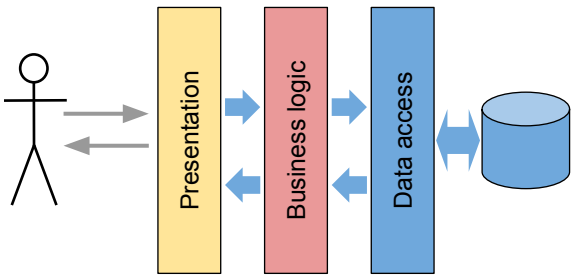
3-Tier Architecture

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.



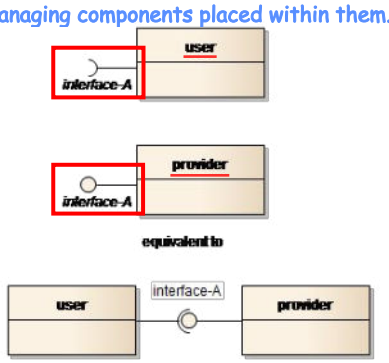
Use of Containers in Middleware

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

"Container" is a general term. Here, we mean that containers are entities capable of managing components placed within them.

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*.



"For example, you may have several components, and it's up to the container to take care of them. So, the components live within the container. When they need to be instantiated, they will be instantiated inside a container. This is a way to achieve proper management of components within a general application."

From Now on...

Jakarta EE rappresenta sia una continuazione che una nuova era per Java EE. Attraverso il trasferimento della gestione da Oracle alla Eclipse Foundation (che per questioni legali non poteva conservare il nome Java EE), Jakarta EE si propone di rinnovare la piattaforma Java EE, mantenendo al contempo la ricchezza e la solidità della sua eredità, ma con un approccio più comunitario, aperto e adattabile alle tendenze tecnologiche moderne.

Java EE™

since late 2019...

JAKARTA® EE

Web Applications

"It is a special and crucial category of enterprise application, particularly important nowadays because more or less any enterprise application has a web interface. Therefore, being enabled to develop part of any significant distributed application as a web one is crucial."

Una web application (o applicazione web) è un'applicazione software che viene eseguita su un web server.
A differenza delle applicazioni tradizionali, che vengono installate e eseguite sul dispositivo dell'utente, le applicazioni web funzionano su un modello client-server, con il browser che funge da client.
Le applicazioni web sono più complesse rispetto ai contenuti statici e spesso richiedono l'interazione con database, autenticazione degli utenti, logica di business, ecc...

Static vs Dynamic Content

contenuto

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.

distinguere

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

La Common Gateway Interface (CGI) è un protocollo che consente ai server web di interagire con programmi esterni, spesso chiamati "script CGI", per generare contenuti dinamici.
CGI è una specifica che definisce come i server web possono eseguire programmi esterni e passare loro informazioni.

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

Svantaggi di CGI: per ogni richiesta viene creato un nuovo processo per eseguire lo script. Questo può causare un sovraccarico di processo e rallentare le prestazioni del server.

Common Gateway Interface

A standard protocol for web servers to execute shell programs
that dynamically generate web pages
C ly pla l dir
Parameters ar ssed via environme vars ar

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

1) http://xyz.com: Questa è l'URL del sito web o del server. Indica il protocollo utilizzato (HTTP) seguito dal nome di dominio "xyz.com".
2) /cgi-bin/myscript.pl: Questa parte dell'URL indica la posizione dello script CGI. Solitamente, gli script CGI vengono collocati nella directory "cgi-bin" del server. In questo caso, si fa riferimento a uno script chiamato "myscript.pl".
3) /my/pathinfo: Questa parte dell'URL rappresenta una parte di percorso aggiuntiva che viene passata allo script come informazioni sul percorso. Gli script CGI possono accedere a queste informazioni tramite variabili d'ambiente o altri meccanismi.
4) ?a=1&b=2: Questa è la parte dei parametri della richiesta. Indica che si stanno passando due parametri, "a" con il valore "1" e "b" con il valore "2". Gli script CGI possono accedere a questi parametri e utilizzarli durante l'esecuzione.

Con CGI, il codice degli script viene eseguito sul server web. Quando una richiesta viene fatta al server per un determinato script CGI, il server crea un nuovo processo per eseguire lo script. Lo script genera quindi il contenuto dinamico che viene restituito al client.
Per migliorare le prestazioni eseguendo il codice internamente al web server, si può utilizzare il multithreading.

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

In ambienti Jakarta EE, l'uso di Servlets è proposto come alternativa. I Servlets possono essere eseguiti all'interno del processo del server web e possono beneficiare del multithreading per gestire più richieste simultaneamente.

"The reference implementation for a web server in a Java program is Tomcat."

The Tomcat Web Server

Il server web Tomcat è composto da diversi componenti interni che svolgono ruoli specifici nella gestione delle richieste HTTP e nell'esecuzione di servlet e JSP.

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.

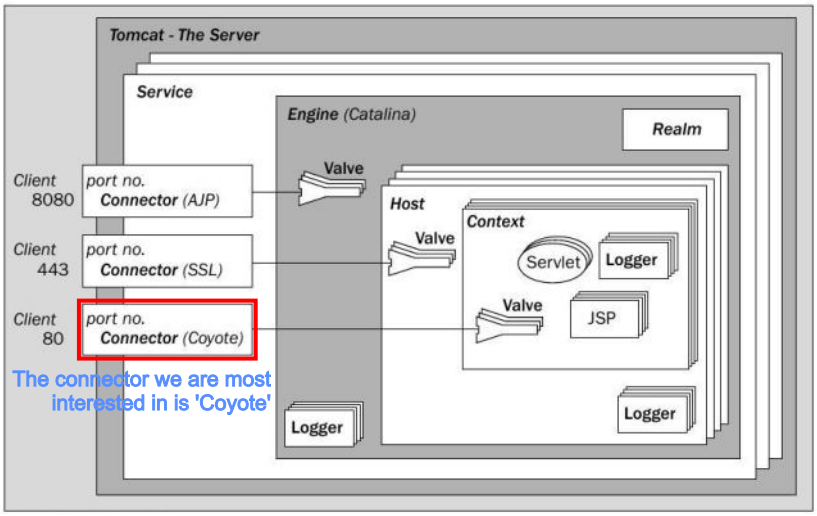
JSP (JavaServer Pages) := sono una tecnologia di programmazione lato server utilizzata per la creazione di pagine web dinamiche. Le JSP consentono di incorporare direttamente codice Java all'interno di pagine HTML, semplificando la creazione di contenuti web dinamici e interattivi.

Tomcat Architecture

Valve:
element to be inserted in the REQ processing pipeline

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

that are not represented in the figure



The connector we are most interested in is 'Coyote'

"The server internally supports diverse services, each equipped with an internal engine. The connector plays a crucial role in facilitating communication with the components housed within each engine. Additionally, each engine is associated with a related Realm. It's worth noting that multiple hosts can exist, corresponding to distinct host names mapped within the web server.

Within a given host, the presence of several independent web applications is possible. To distinguish between these applications, each is associated with a context, a descriptor that typically corresponds to a specific segment of the URL. Every context relies on supporting software, often supplied in Java and managed by servlets within a servlet container, along with JSP.

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.

In Tomcat, un "host" rappresenta un'istanza di un server virtuale all'interno del server Tomcat principale. Questi host virtuali consentono a Tomcat di gestire più domini su un unico server fisico. Ogni host può avere configurazioni e applicazioni web indipendenti.

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.

"webapps" is the main directory. Under webapps, you will have several directories, each associated with one web application.

Each single web app corresponds to a *base file name*.

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

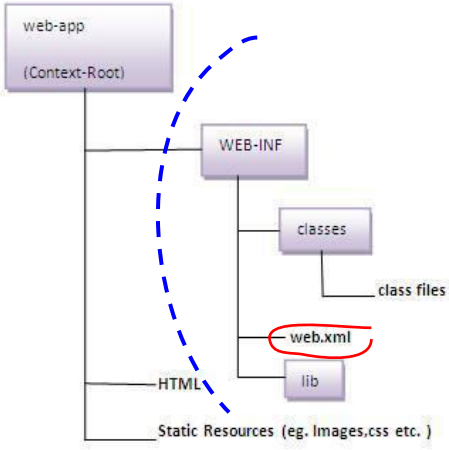
Il "Context Path" rappresenta il percorso specifico a cui si accede per raggiungere un'applicazione web in Tomcat. Ad esempio, se hai un'applicazione con context path "/myapp", il suo URL sarà qualcosa come "http://localhost:8080/myapp".

- 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 '#'.

  Esempio: se il context path è "/myapp", il Base File Name sarà "myapp".

Structure of a WebApp Directory

NB: questa cartella web-app non è la cartella main ("webapps") dove troviamo tutte le cartelle delle varie web applications. Infatti questa è già la cartella di una singola web-app. Questa la troveremo proprio all'interno di "webapps".



Ogni singola applicazione web ha una sua sottodirectory (all'interno della sottodirectory 'webapps' di Tomcat), ognuna dedicata al supporto della relativa applicazione web sotto quell'istanza di Tomcat.

All'interno di una subdirectory di una specifica web-app troviamo:
- Sottodirectory "WEB-INF" (contenuto privato): questa directory contiene configurazioni specifiche dell'applicazione e risorse che non sono direttamente accessibili dall'esterno.
- Contenuto Pubblico: al di fuori di "WEB-INF", è possibile collocare risorse pubbliche accessibili direttamente dall'esterno. Queste risorse possono includere file HTML, CSS, JavaScript, immagini e altro ancora.

"Within WEB-INF, you will also find the 'classes' folder to place the .class files for Java classes, a 'lib' folder where you can place the .jar files necessary for your app (which you may need because they contain third-party files). Additionally, you will find the 'web.xml' file, a special file with a standard name. This is a configuration file where you specify the configuration parameters for this specific application."

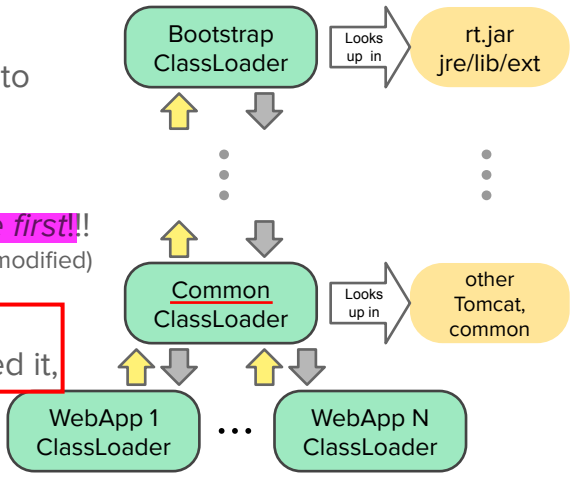
"We can have multiple web applications under Tomcat, and we have to ensure they do not interfere. Interference might occur because different web applications require specific Java classes. In some cases, these classes may belong to different versions of the same library. While they are slightly different, they share the same name. Therefore, we need to find a way to separate the classes needed by one web application from the classes used by another web application. This can be achieved by working at the class loader level."

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.



Un WebApp ClassLoader non delega prima di cercare la classe. Nel contesto delle applicazioni web, la normale politica di delega può essere modificata per dare priorità alle classi specifiche dell'applicazione.

"We have seen that class loaders in Java are organized in a hierarchy, and delegation is used to let one specific class loader deal with sub-portions of Java classes. Now, we want to enable each web application to deal with some specific classes dedicated just to them. The trick is: we will have some classes within Tomcat that have to be shared across the web server (because each web application needs to access them), and this is handled by a dedicated class loader known as the common class loader, which looks up in the common Tomcat directory. Moreover, we will have a specific class loader for each application.
In this way, it is impossible to have the same core class loaded twice, but it is still possible to have the same 'custom' class used in the lower level loaded more than one time."

ClassLoaders, In Depth

It is possible to specify a more advanced configuration:

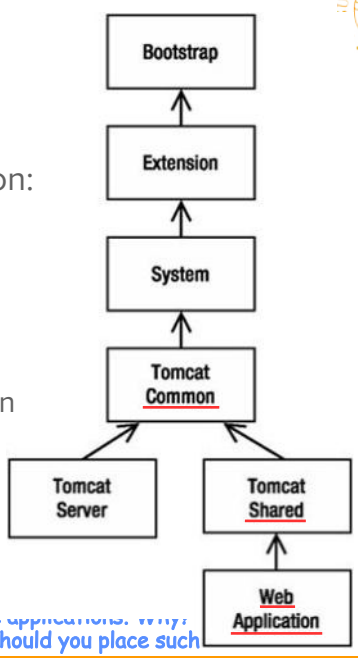
Tomcat Common: looks in
CATALINA_HOME/common/**lib**/ — For .jar files
CATALINA_HOME/common/**classes**/ — For .class files

Tomcat Server: for server-specific classes

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

La variabile d'ambiente "CATALINA_HOME" è utilizzata per indicare la directory di installazione di Apache Tomcat nel sistema.

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



Suppose you need some libraries that are very common. In this scenario, where should you place such classes? You may say, "Well, let's put our classes where the common class loader looks in".
However, in this case, there is the possibility of interference with the classes used by Tomcat itself. Therefore, it is better to have a dedicated place to insert all the libraries used by all the web applications managed by my Tomcat. This place is the 'shared' directory, and it has its own class loader for dealing with such classes."

Servlets and More

Un Servlet è una classe Java che risponde alle richieste HTTP.
Le servlet vengono eseguite sul server, elaborano i dati e inviano una risposta al client (come un browser web). Possono generare contenuti dinamici come pagine HTML dinamiche.
Sono la base per la creazione di applicazioni web dinamiche in Java.

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!***

"Every time a request comes to the web server, a new thread is used to deal with such a request. In any case, as the same code may likely run by multiple threads at the same time, you have to make sure that the threads of our servlets have to be thread-safe."

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.

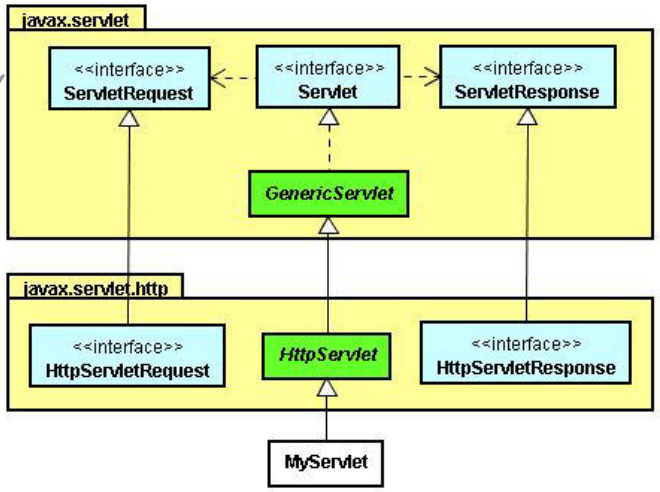
What's a Servlet?

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

Questa interfaccia offre metodi specifici per gestire i diversi tipi di richieste HTTP come GET, POST, etc.

Recently, `javax` ➡ `jakarta`

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



Servlet Life-cycle

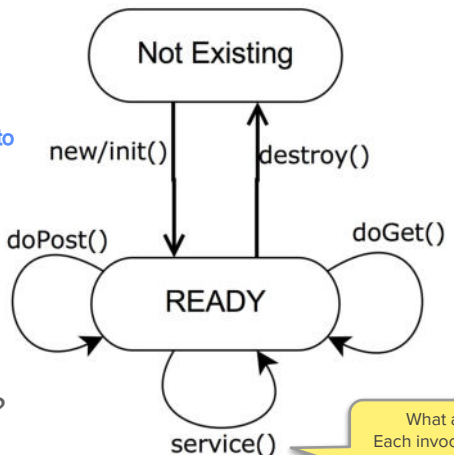
The entire lifecycle of a servlet *is managed by the container.*

"Every time a request comes to the web server, it's up to the container to invoke the method named 'service()'."

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)?

risposta nella pagina seguente



Il metodo service() è fornito dalla classe HttpServlet ed è responsabile della gestione delle richieste HTTP. Questo metodo è invocato dal container servlet ogni volta che arriva una richiesta per il servlet.
La demultiplexazione si riferisce alla capacità del metodo service() di indirizzare la richiesta HTTP ai metodi specifici del servlet in base al tipo di richiesta. Ad esempio, se la richiesta è di tipo GET, il metodo service() può indirizzare la richiesta al metodo doGet(). Se la richiesta è di tipo POST, può essere indirizzata a doPost() e così via.

"We defined a servlet as a class, and we said in practice we will instantiate this class, and we will give this instance to the web container for managing it.

Q: Does it make sense to have a single instance, or is it reasonable to use more instances of a servlet, which are instances of the same class?
A: We may have multiple threads, but we don't need more than one servlet because the code associated with a request will be run by a different thread. So, the problem is not having multiple servlets; the problem is having more available threads to reply to multiple client requests coming at the same time. But we don't have any advantage in keeping more instances of the same servlet. Typically, there is one single servlet instance per servlet class, but multiple threads that make use of such a single instance."

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`

This approach is much more convenient that CGI!

Previously, `javax`

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

Servlet VS CGI

- Efficienza e Prestazioni: le Servlets sono eseguite all'interno dello spazio di memoria del server web (vengono caricate nella memoria del server e rimangono "vive" nel server per gestire molteplici richieste), il che le rende più efficienti rispetto ai CGI, i quali richiedono che il server web avvii un nuovo processo (isolato) per ogni richiesta. L'avvio e la terminazione frequenti di processi possono essere onerosi in termini di prestazioni, specialmente con un numero elevato di richieste.
- Gestione delle Risorse: le Servlets rimangono in memoria e sono gestite dal server delle applicazioni, di conseguenza possono riutilizzare gli oggetti in memoria e le connessioni delle altre Servlets, riducendo l'overhead di creazione e distruzione di queste risorse per ogni richiesta.
- Scalabilità: la capacità delle Servlets di gestire simultaneamente più richieste con un minor overhead le rende più scalabili rispetto ai CGI.

Reading REQ Parameters

indipendentemente

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.

"Some information in the request is also placed in HTTP headers, and you can retrieve the information in the headers exactly in the same way as before."

Acting on Response

"How to write the content of a response? Also in this case, you can put some information in headers or in the response body."

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>)`

l'esempio mostra il metodo setContentType(<MIME_type>) che è utilizzato per impostare il tipo di contenuto (MIME type) che verrà inviato nel corpo della risposta HTTP. Questa informazione viene inserita nell'header della risposta.

Mapping URLs onto Servlets

Questo processo di mappatura è fondamentale per determinare quale Servlet deve essere chiamata in risposta a una particolare richiesta HTTP. Due modi per farlo:

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

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

devo mettere questi due "elementi" nel file web.xml

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

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

array di stringhe che definisce i pattern di URL associati a questa Servlet. Ad esempio, /foo, /bar, e /pippo* sono i pattern di URL per cui questa Servlet sarà invocata.
Il pattern /pippo* indica che qualsiasi URL che inizia con /pippo verrà gestito da questa Servlet.

"The first method is theoretically more correct because it separates configuration from code. However, in practice, the second method is more commonly used."

"It's up to the server to decide where a session starts, as the server is in charge of implementing all the mechanisms for keeping the required information. How to decide the point in time at which the session is over? As is often the case, we can use a timeout. Every time you interact with the same client, a timeout is started. We set a timeout period, and when such a timeout expires, we say that the session is over. Note that the timer is activated at every interaction. If you keep contacting the server, your session will continue over and over."

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:

Nelle Servlets questo si fa tipicamente richiedendo l'oggetto di sessione corrente da un oggetto di richiesta

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

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

un riferimento di tipo interfaccia può essere utilizzato per fare riferimento a qualsiasi oggetto che implementa quell'interfaccia. In questo caso, non ci importa il tipo dell'oggetto restituisto, in quanto .getSession è un factory method.

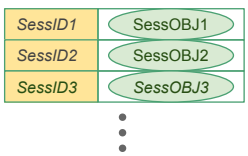
Qui, req è un'istanza di HttpServletRequest. Il metodo getSession() può essere chiamato con un argomento booleano:
- se impostato su true, il metodo crea una nuova sessione per l'utente se non esiste già una sessione.
- se impostato su false, il metodo restituisce l'oggetto di sessione esistente senza crearne uno nuovo, e restituisce null se non c'è una sessione esistente.

Internal Handling of Session Objects

"The session object has to be kept for the whole duration of the session, but the system needs to access it in a very efficient way. Therefore, it is reasonable to keep it in an associative array (a map, a hash table, etc.). So, you will need a specific ID to access this session object. The value for such a key is created by the system at the time the session is first created, and such value will be an ID for the entire session."

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.



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

"The client has to know its session ID and has to store it somewhere, as it will need to include it in the subsequent requests it sends. Where to keep it on the client side? The most natural way to keep it is to make use of cookies. Many web applications ask you if you want to keep the cookies, precisely to be able to use cookies for the purpose of employing this session mechanism"

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

Questa tecnica coinvolge l'inclusione di campi nascosti all'interno del modulo HTML all'interno di una pagina web. Questi campi contengono dati che devono essere inviati al server quando l'utente invia il modulo.

Questa tecnica coinvolge la modifica degli URL delle pagine web in modo da includere informazioni sulla sessione direttamente nell'URL stesso.
Quando un utente fa clic su un link o invia una richiesta, il server può estrarre l'identificatore di sessione dall'URL e utilizzarlo per identificare la sessione utente.

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

tutte le informazioni di sessione associate a quell'utente verranno eliminate.

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

Servlet Example

A first example of a Servlet (we can see that it is a servlet because it extends HttpServlet):

```
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>");
   }
}
```

→ informazione nell'header della risposta

→ informazione nel body

Un PrintWriter è utilizzato per scrivere dati di testo nella risposta HTTP.

Questa risposta verrà inviata al client che ha effettuato la richiesta HTTP di tipo GET e verrà visualizzata come una pagina web nel browser del client con il titolo specificato nel contenuto HTML.

"Who will take care to put in the right value in the parameters req and res?
It's up to the container to handle the servlet, to invoke such methods at the proper time, and to provide values for all the parameters."

REQ Processing: Servlet Filters

"Sometimes it makes sense to use filters for dealing with some operations that should be done every time a request comes towards certain servlets."

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

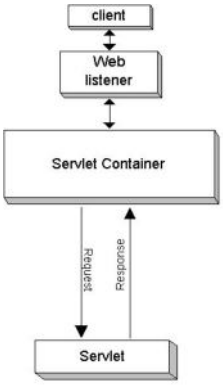
E.g. recording, IP logging, input validation, authentication check, encryption/decryption, etc.

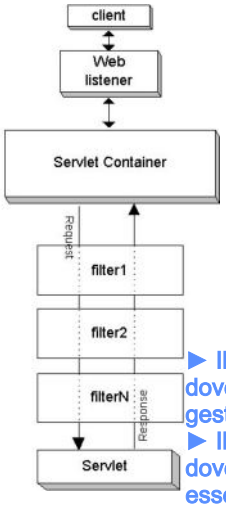
Each action can be carried out by a *pluggable* server-managed component named *servlet filter*; its application depends on the relative *url pattern*.

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!

► Il diagramma a sinistra rappresenta un flusso di richiesta semplice, dove una richiesta del client passa attraverso un listener web e viene gestita direttamente da una Servlet nel container Servlet.
► Il diagramma a destra mostra un flusso di richiesta complesso, dove la richiesta del client passa attraverso una serie di filtri prima di essere gestita dalla Servlet.

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

esempi di "resource": servlet, jsp, html, ...

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

Comprendere come far collaborare le servlet è cruciale per la creazione di robuste applicazioni web che richiedono che più componenti generino una singola risposta. Questo meccanismo è ampiamente utilizzato per comporre pagine web da vari componenti, come intestazioni, piè di pagina o contenuti generati da diverse servlet.

Le servlet per collaborare utilizzando RequestDispatcher, devono trovarsi all'interno dello stesso Servlet Container. Il motivo principale è che RequestDispatcher è una funzionalità del Servlet API, che è specifica per il Servlet Container e offre mezzi per la comunicazione tra servlet all'interno dello stesso container.

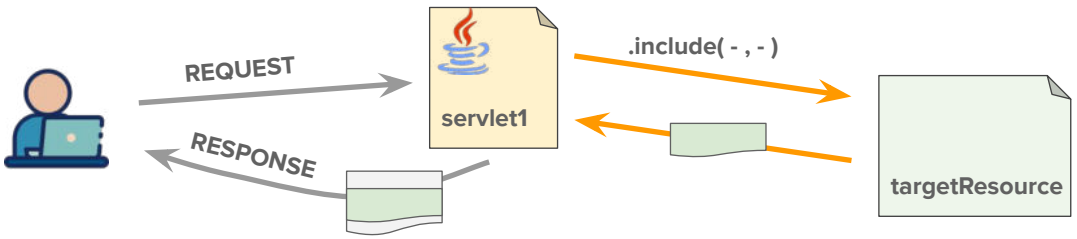
In generale però, le servlet non sono obbligate a trovarsi all'interno dello stesso container per collaborare. Le servlet possono collaborare tra loro all'interno dello stesso container o persino tra diversi container all'interno della stessa applicazione o tra applicazioni diverse su server web separati. Questo è possibile tramite meccanismi di comunicazione inter-applicazione o inter-server, come l'utilizzo di servizi web o la condivisione di dati attraverso il protocollo HTTP.

Inclusion with Servlets

Una servlet può richiedere e includere il contributo di un'altra servlet o risorsa all'interno della sua risposta. Ad esempio, una servlet potrebbe includere il risultato di un'altra servlet all'interno della propria pagina HTML in modo da comporre la risposta finale.

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);
```

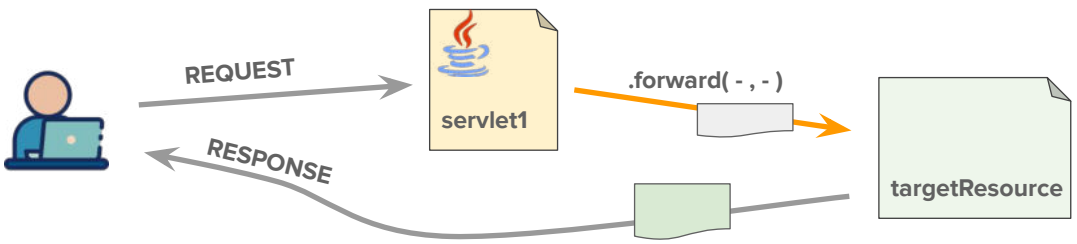


Forwarding in Servlets

Questo tipo di collaborazione comporta il trasferimento della responsabilità della gestione della richiesta da una servlet all'altra. Ad esempio, una servlet può decidere di delegare la gestione di una parte specifica della richiesta a un'altra servlet che è meglio adatta per gestirla. Ciò è utile per organizzare in modo modulare la logica delle servlet e migliorare la manutenibilità dell'applicazione.

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*) informazioni memorizzate accessibili solo durante la vita della richiesta corrente.
- The session object (*session scope*) informazioni persistenti attraverso più interazioni HTTP da parte dello stesso utente, permettendo quindi di mantenere uno stato tra le diverse richieste.
- The ServletContext obj, one per hosted web application (*web scope*)
  informazioni comuni a tutte le richieste e tutti gli utenti dell'applicazione, come configurazioni iniziali o dati condivisi a livello di applicazione.

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.

Antipattern "Magic Servlet": Si verifica quando una singola servlet diventa troppo complessa e inizia a gestire molteplici aspetti dell'applicazione, come l'accesso ai dati, la logica di business e la presentazione dei risultati all'utente. Questo rende la servlet sovraccarica di responsabilità e difficile da gestire. Inoltre, le modifiche diventano complesse e il rischio di errori aumenta.

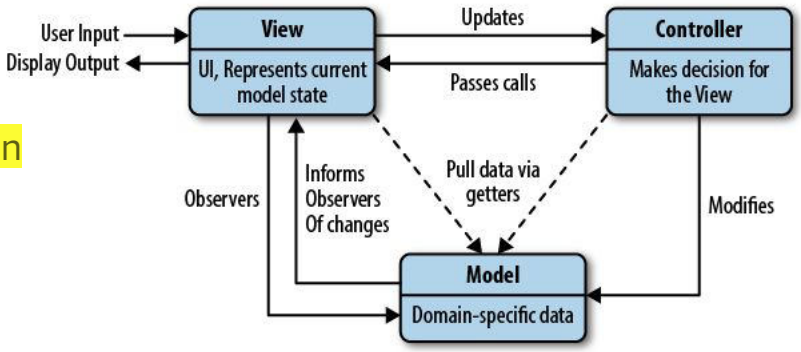
MVC - Model View Controller (pattern)

Originally conceived for GUIs

Model: manages data

View: handles interaction with the user

Controller: coordinates interactions



Flusso di Interazione nel MVC:
1) L'utente interagisce con la Vista (ad esempio, inserendo dati o cliccando un elemento dell'interfaccia).
2) La Vista invia l'input dell'utente al Controllore.
3) Il Controllore interpreta l'input (ad esempio, come un comando per aggiornare i dati) e modifica il Modello se necessario.
4) Il Modello notifica la Vista di eventuali cambiamenti di stato.
5) La Vista si aggiorna di conseguenza, riflettendo i nuovi dati o lo stato del Modello.

Template Systems & Engines

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

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

Un sistema di template web lato server (server-side web template system) è una tecnologia utilizzata per generare dinamicamente pagine web sul lato server prima di inviarle al client. In un'applicazione web, ci sono spesso parti di una pagina che devono essere generate dinamicamente in base a dati o logica specifica. Un sistema di template server-side semplifica questo processo consentendo agli sviluppatori di incorporare facilmente codice dinamico all'interno dei documenti HTML o altri formati di pagina.

Invece di avere tutto il codice HTML mescolato con il codice di logica o di programmazione, il sistema di template permette di separare la presentazione dalla logica. I template contengono segnaposto o tag che vengono sostituiti con dati dinamici quando la pagina viene generata sul server prima di essere inviata al browser del client.

Aside: Web Template Systems

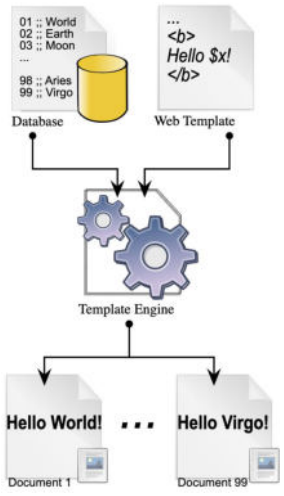


Image from WikiCommons

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

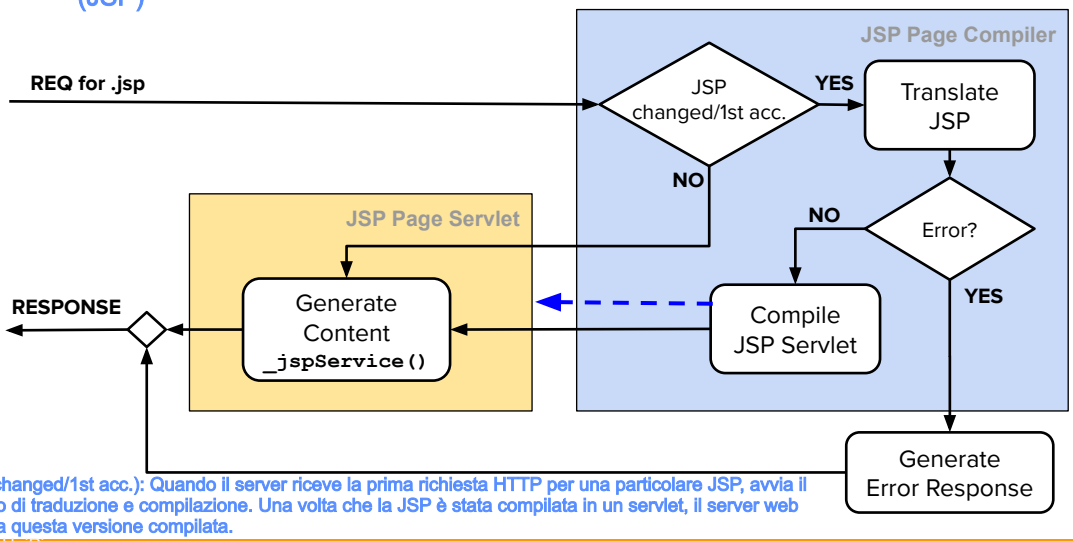
The Template Engine is in charge of analyzing the “placeholder elements” in the template, and getting to the final document(s) on the basis of a collection of elements, possibly in the form of a dataset.

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

ad oggi Thymeleaf può essere considerato un successore di JSP, specialmente nel contesto dello sviluppo web moderno con Java. Infatti, tra i suoi vantaggi, vanta un'eccellente integrazione con Spring Boot, che è diventato lo standard de facto per lo sviluppo di applicazioni Spring moderne.

Jakarta Server Pages - Processing

(JSP)



• (JSP changed/1st acc.): Quando il server riceve la prima richiesta HTTP per una particolare JSP, avvia il processo di traduzione e compilazione. Una volta che la JSP è stata compilata in un servlet, il server web conserva questa versione compilata.

• Translate JSP: la traduzione inizia con l'analisi del codice JSP, che mescola HTML e codice Java. Questo processo è gestito dal traduttore JSP.
Il risultato di questa fase è un file Java puro (.java), che rappresenta il codice sorgente di un servlet. Questo file contiene il codice HTML della JSP originale, trasformato in istruzioni di output (ad esempio, tramite out.write("<html>...")), e inserisce il codice Java degli scriptlet direttamente nel corpo del servlet.
• Compile JSP Servlet: Il file Java generato nella fase di traduzione viene poi compilato in bytecode Java, creando un file .class. Questo è un passaggio cruciale che trasforma il codice sorgente in una forma eseguibile dalla Java Virtual Machine (JVM).
Il servlet risultante è una classe Java che estende HttpServlet, il che significa che eredita la capacità di gestire richieste e risposte HTTP, implementando la logica specifica definita nella JSP originale.
Dopo la compilazione, il servlet è pronto per essere caricato e istanziato dal server ogni volta che la pagina JSP viene richiesta
• _jspService(): il metodo _jspService() è il cuore del servlet generato, poiché è il punto in cui il codice JSP originale viene effettivamente eseguito per servire la richiesta dell'utente. Viene invocato automaticamente dal server ogni volta che il servlet riceve una richiesta HTTP.
All'interno di questo metodo, il codice Java, tradotto e compilato dalla JSP, viene eseguito per produrre il contenuto dinamico della pagina web. Questo include la gestione di input utente, l'interazione con il server e il database, e la generazione di risposte HTML.

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:

possono essere utilizzati direttamente all'interno degli script JSP senza la necessità di essere dichiarati o inizializzati

- **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...

comunemente usato in JSP per scrivere output HTML o testo direttamente nella pagina risultante

sinonimo del riferimento this in Java e rappresenta l'istanza corrente del servlet (che deriva dalla JSP) che sta gestendo la richiesta.

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

- I Java Beans sono componenti Java che seguono una convenzione di programmazione specifica. Sono classi Java con proprietà private, metodi getter e setter pubblici, e un costruttore senza argomenti. Sono utilizzati in JSP per incapsulare e gestire la logica di business e i dati.
- JSTL è una libreria di tag standard che fornisce tag custom per controllare il flusso di esecuzione in una JSP, come cicli, condizioni, manipolazione di XML, internazionalizzazione e accesso ai database. Riduce la necessità di scrivere codice Java scriptlet nelle JSP, favorendo la scrittura di codice più pulito e mantenibile.
- EL è un linguaggio di espressioni utilizzato in JSP per semplificare l'accesso e la manipolazione dei dati dell'applicazione. Permette di accedere facilmente ai dati memorizzati in JavaBeans, variabili di ambito (scope variables), come request, session, e application, e ai parametri di richiesta.

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  anziché il classico .html
            `-- jsp
                `-- websource.jsp
```

Java classes (Servlets etc.)

Resources

Web components

- main: Sotto la directory src, main contiene il codice sorgente effettivo dell'applicazione.
- com/example/projects: Un esempio di struttura di package che potrebbe contenere classi Java, organizzate in modo gerarchico.
- jsp: Una sotto-directory dove possono essere collocate ulteriori pagine JSP come websource.jsp, utilizzate per generare contenuti dinamici.

Deploying/Packaging WebApps

distribuzione confezionamento

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

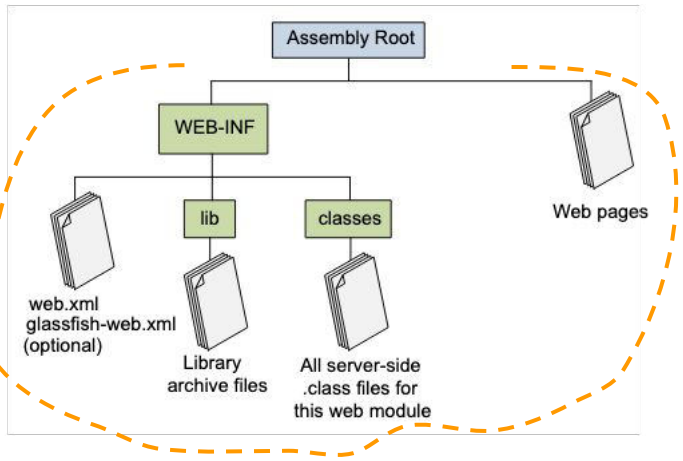


Image from "The Jakarta EE Tutorial"

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

Il file WAR è un archivio che contiene tutti i componenti necessari per eseguire l'applicazione web su un server compatibile con Jakarta EE. È un file compresso che segue lo standard ZIP.

Una volta creato il file WAR, può essere distribuito (deployed) su un server di applicazioni Jakarta EE. Il formato WAR rappresenta quindi una soluzione comoda e efficiente per la distribuzione di applicazioni web su server di applicazioni Jakarta EE.

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*.

all'aumentare della complessità, mantenere tutto il codice in JSP e servlet può diventare difficile da gestire

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.

I framework web forniscono un approccio integrato che nasconde la complessità delle tecnologie sottostanti come servlet e JSP, fornendo astrazioni più elevate e componenti riutilizzabili.

JSP nel Pattern MVC:

- View: è la rappresentazione visuale dei dati, ovvero ciò che l'utente vede e con cui interagisce nel browser. Qui entrano in gioco le JSP, che permettono di creare contenuti dinamici combinando HTML standard con tag JSP e JavaBeans. Il codice JSP utilizza Java Beans o altri componenti Java per recuperare dati dal modello (Model) e visualizzarli.
- Model: Rappresenta la logica di business e i dati dell'applicazione. Non è direttamente parte della JSP, ma la JSP può interagire con il modello per ottenere le informazioni necessarie da visualizzare.
- Controller: Solitamente implementato come un servlet, il Controller gestisce la logica di navigazione e le interazioni dell'utente, determinando quale pagina (view) mostrare e quale modello utilizzare per mostrare i dati richiesti.

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)**

Per PROG Large Scale:

Spring, specialmente con Spring Boot, adotta un approccio più "leggero" rispetto a JSF, mirando a semplificare notevolmente la configurazione iniziale di un progetto. Con Spring Boot, l'obiettivo è ottenere un'applicazione eseguibile con minimo sforzo e configurazione. Mentre la configurazione tradizionale in Spring richiede un setup XML o annotazioni Java, che offre un elevato grado di flessibilità e controllo, può risultare complesso, specialmente per nuovi utenti o progetti di dimensioni ridotte. Spring Boot, al contrario, fornisce configurazioni predefinite per molte funzionalità, che possono essere facilmente sovrascritte se necessario. Questo consente agli sviluppatori di avviare rapidamente il progetto con impostazioni predefinite, ma di personalizzare l'applicazione secondo le esigenze specifiche del progetto.

"Here the focus is not just on showing some content related to some resources through the web interface, but instead, we want to design and develop an entire application made up of different components with the support of specific middleware components. The reason here is to focus on what you want to accomplish in your applications. Typically, this is named the 'business logic part' of the application, which is the part of processing actions to obtain a certain result in the enterprise context.
So, the idea is to avoid caring about all the details required by distributed computing, but we want to understand how to develop the most important components. All the problems related to deployment, communication, synchronization, and so on, will be tackled by means of the functionalities provided by a special kind of middleware."

Enterprise Applications

Le "Enterprise Applications" si riferiscono a applicazioni software progettate per soddisfare le esigenze delle organizzazioni, come aziende, scuole, governi, ecc., piuttosto che dei singoli utenti. Queste applicazioni sono spesso complesse, scalabili, distribuite e orientate ai servizi.

Problems with Plain Objects

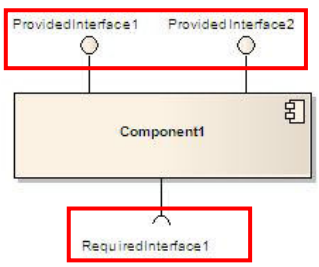
Use of semplici plain objects in an enterprise app shows some problems:

- 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!



In un'architettura basata su componenti, ogni componente dichiara esplicitamente le sue interfacce, cioè i metodi e le funzionalità che fornisce (interfacce fornite) e quelle di cui ha bisogno da altri componenti (interfacce richieste). Questo approccio crea un "contratto" tra i diversi componenti, rendendo le relazioni e le dipendenze tra di essi molto più chiare e gestibili.

Application Servers

"By using application servers, the programming model becomes easier because the programmer must focus just on business logic. They only need to specify what to do to provide the required service, without worrying about managing the object because the management is handled by the application server. By management, we mean keeping it there, providing it with communication facilities, etc."

Need to simplify the programming model:

the programmer must focus on business logic,

without spending time on distributed computing issues.

"So, for this reason, we will use the container pattern to manage the components."

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

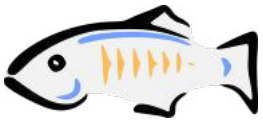
"For example, we have seen how servlets can live in a container. So, in this sense, servlets are special 'components,' but they are specialized ones because they are supposed to deal with requests coming from external clients by means of HTTP."

Middleware solution: **Application server**

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

Java Application Servers - Examples

- **Glassfish** (JEE reference implementation)



We will see it in the laboratory lessons

➔ payara

Payara Server ha le sue radici in GlassFish, ereditando molte delle sue caratteristiche e funzionalità. Payara ha continuato a sviluppare e migliorare il server, fornendo aggiornamenti, correzioni e nuove funzionalità.

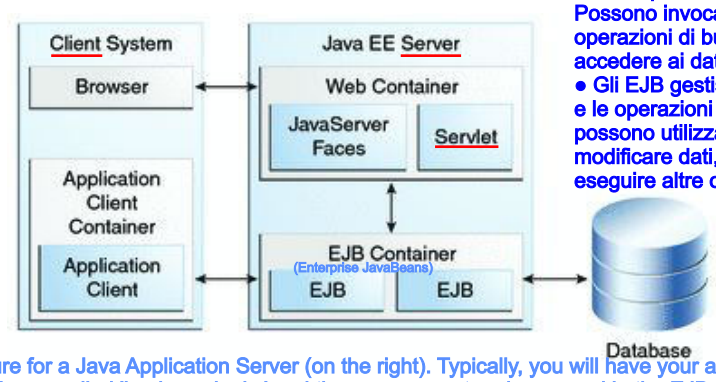
- **JBoss** EAP and subsequently **WildFly**

WildFly

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

Java Application Servers - Overview

Le Servlet e gli EJB spesso lavorano insieme all'interno di un'applicazione web:
- Le Servlet gestiscono le richieste degli utenti e producono risposte dinamiche. Possono invocare gli EJB per eseguire operazioni di business complesse o per accedere ai dati.
- Gli EJB gestiscono la logica di business e le operazioni di persistenza. Le Servlet possono utilizzare gli EJB per ottenere o modificare dati, gestire transazioni, e eseguire altre operazioni di business.



"Let's see a basic architecture for a Java Application Server (on the right). Typically, you will have your application made up of some components implementing the so-called 'business logic,' and they are present and managed in the EJB Container (Enterprise JavaBeans for Java Enterprise applications). Most of the time, an enterprise application also contains the web part, which serves as an interface with users. This part is formed by servlets and other web components. In this case, JavaServer Faces is the component used, just as an example because it is the reference one for the Java Platform.

What about the client side? If your client is just a web client, you don't need anything special other than a browser. But it is also possible to develop a specific application for the client side. In this case, there is the possibility to use a container for the client-side application, even if this way of organizing the application is becoming less and less popular in modern applications.

What about accessing data in a database? Access is mediated by special components, typically EJBs. This is a way to separate concerns within applications. This approach is reasonable because every time you want to modify something in the way you access data or just want to make changes in the data model, you have to work on that specific component, and all the others will remain unchanged."

Enterprise Java Beans

Gli Enterprise Java Beans (EJB) sono una parte fondamentale della specifica Java EE (ora Jakarta EE) e rappresentano un modello di componenti server-side per la realizzazione di applicazioni aziendali distribuite di grandi dimensioni che richiedono scalabilità, transazioni gestite e sicurezza. Gli EJB offrono una serie di servizi che permettono agli sviluppatori di concentrarsi sulla logica di business, delegando ad essi compiti come la gestione delle transazioni, la sicurezza, la gestione della concorrenza e l'accesso ai database.

La "logica di business" di un'applicazione web si riferisce all'insieme di regole, processi e operazioni che definiscono come funziona il core dell'applicazione per risolvere problemi specifici o fornire servizi agli utenti. In altre parole, è la parte dell'applicazione che gestisce la logica applicativa vera e propria, distinguendosi dall'interfaccia utente e dalla gestione delle risorse di sistema (a cui pensano gli EJB).

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.

servizi come: gestione del ciclo di vita, sicurezza, gestione delle risorse, pooling e serializzazione.

Using EJBs, client modules become thinner.

in quanto la logica di business complessa è gestita sul lato server

Reusability: new applications can be built from existing EJBs.

Gli EJB (Enterprise JavaBeans) sono componenti server-side progettati per semplificare la vita degli sviluppatori fornendo molte funzionalità "pronte all'uso" (es. gestione delle transazioni, sicurezza, concorrenza, ecc..) che normalmente richiederebbero molto tempo e sforzo per essere implementate. Questo permette agli sviluppatori di concentrarsi sulla logica di business specifica della loro applicazione, sapendo che l'infrastruttura sottostante si occuperà dei dettagli complessi.

All'interno dell'EJB, gli sviluppatori scrivono la logica di business, cioè le operazioni specifiche dell'applicazione (ad esempio, calcolare un prezzo, processare un ordine, registrare un utente), senza doversi preoccupare di dover scrivere codice per "le funzionalità di infrastruttura", a cui già pensa il codice di base degli EJB.

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** (slide 43a)

Perform work for their clients, by encapsulating the business logic

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

"They are designed to be activated (triggered) by some event occurring in the system, instead of being explicitly called by clients (as was the case for session beans)."

* "They can be accessed directly by other components on our machine or may be accessed by other components that reside somewhere else on some other nodes. So somehow, you may have location transparency for them. But what about issues related to communication in this case? You don't have to care about it because it's up to the application server. In practice, what happens is that the application server puts in place what is required for communication, and the chosen technology is RMI. Why RMI? Because in RMI, we have remote references that let you make use of an object that is not necessarily located on your node, and this is exactly what is needed here. So, you do not necessarily have to know it, but under the hood, RMI will provide whatever you need, and you do not need to know to be able to write code for RMI communication because it's not up to you; it's up to the communication server."

Stateful Session Beans

Una volta che l'istanza del SFbean è stata creata, il client comunica con essa tramite RMI.

State: values for its instance variables.

the state corresponds to the values for the fields of the 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.

In this scenario, you internally utilize remote references similar to what we did in RMI. The client for a particular stateful session bean becomes the software component holding the remote reference for that bean. However, the responsibility of maintaining this remote reference lies with the client. The responsibility for consistently using the same reference to access the specific stateful session bean lies with the holder of the remote reference.

It's important to note that in this situation, the client always has the same instance to communicate with. However, the notion of a "session" in this context doesn't necessarily align with a web session. While it can be defined similarly, considering clients are software components, the session doesn't automatically imply a connection to web applications. For instance, you might decide to associate a stateful session bean with the web session in your web application. In this case, within the session object on your web server, you would store the remote reference to retrieve the corresponding stateful session bean.

As an example, you could use this stateful session bean to manage information like the content of a shopping cart. This choice allows you to associate the web session with the session relative to the connection between your client software and the corresponding stateful session bean. However, this association is not predetermined by the system; it's a decision you make based on your application's requirements.

Stateless Session Beans

Stateless indicates that *ideally* no information is kept by this kind of component

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

"Every time the client performs a request, for the server, it's as if it's the first time it comes into contact with that 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*,

sono più adatti

but a stateful session bean cannot (because of idempotence issues).

non sono adatti

"In a web service, idempotence is a crucial point. Typically, you do not want to have different results from the same call from different clients with the same parameters. So, you expect a web server to exhibit idempotent behavior. Consequently, it is more difficult to guarantee idempotency with SFSB, as operations may have different effects if repeated, based on the bean's internal state.

"These beans are pooled": Il termine "pooled" si riferisce al modo in cui i container EJB gestiscono gli Stateless Session Beans. Invece di creare una nuova istanza del bean ogni volta che un client ne ha bisogno, il container mantiene un pool (un insieme) di istanze. Quando un client richiede un bean, il container assegna al client un'istanza dal pool. Una volta che il client ha finito di utilizzare il bean, l'istanza viene restituita al pool anziché essere distrutta. Questo migliora le prestazioni, poiché la creazione e la distruzione di oggetti sono operazioni costose in termini di risorse.

L'interazione tra un client e un Stateful Session Bean sfruttando le Remote Method Invocation (RMI) avviene attraverso una serie di passaggi che permettono al client di comunicare con il bean su un server remoto. Ecco come funziona il processo:

1. **Localizzazione del Bean**: Inizialmente, il client localizza il Stateful Session Bean utilizzando un servizio di directory, come JNDI (Java Naming and Directory Interface). Questo servizio fornisce al client un riferimento all'oggetto remoto (il bean), che è in realtà uno stub (un oggetto locale che rappresenta il bean remoto).
2. **Creazione dell'Istanza del Bean**: Quando il client ottiene il riferimento al bean, effettua una chiamata iniziale per creare un'istanza del Stateful Session Bean sul server. Questa istanza è dedicata a quel particolare client.
3. **Comunicazione RMI**: Una volta che l'istanza del bean è stata creata, il client comunica con essa tramite RMI. RMI permette al client di invocare metodi sull'oggetto remoto (il bean) come se fosse un oggetto locale. Queste chiamate vengono trasmesse attraverso la rete al server, dove vengono eseguite sull'istanza del bean.
4. **Mantenimento dello Stato**: Durante la sessione, lo stato del client viene mantenuto dall'istanza del Stateful Session Bean. Ogni volta che il client fa una chiamata al bean, quest'ultimo può accedere e modificare il suo stato interno, che è specifico per quel client.
5. **Comunicazione Bidirezionale**: La comunicazione tramite RMI non è solo unidirezionale. Il server, ovvero l'istanza del Stateful Session Bean, può anche ritornare informazioni al client come risposta alle chiamate dei metodi.
6. **Terminazione della Sessione**: Alla fine dell'interazione,

il client deve esplicitamente terminare la sessione con il bean. Questo si fa solitamente invocando un metodo specifico del bean che segnala la fine dell'uso del bean da parte del client. A questo punto, il bean può eseguire operazioni di pulizia e rilasciare le risorse associate allo stato del client.

Singleton Session Beans

Il termine "singleton" si riferisce al Singleton design pattern (una singola istanza)

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.
(as is the case for stateless session beans)

In the case of a singleton session bean, since the same instance is provided to any request by any component, all the components of the application will have access to the state information concurrently. This implies that a singleton session bean is accessed by clients in a concurrent manner. Therefore, the code for a singleton session bean must be thread-safe to handle concurrent access.

EJB Interfaces

Even if it is optional, I would recommend applying an interface-driven approach in the design of EJB. This approach allows you to better specify what you want each individual bean to provide to the overall application.

Session EJBs may implement a local/remote interface.
depending on whether they have to be accessed from clients on the same node or from clients on other nodes.

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.

Even though this method provides a clear design for your application, it is not very agile as it requires a lot of specifications. To simplify the code and make it more manageable, the decision was made to allow programmers to specify much of the information directly in the code using annotations.

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

```
@Local   @Remote
```

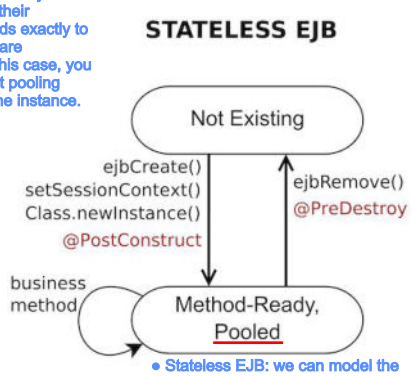
Recently, `javax` → `jakarta`

and 2) the EJB type for an implementation class:

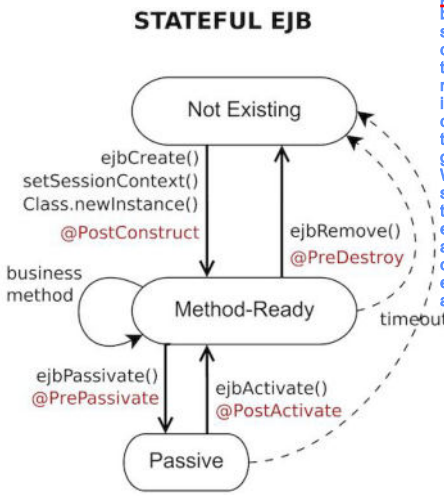
```
@Stateless   @Stateful
```

Lifecycles of Session EJBs

• Singleton session beans: they are not a problem, as there is only one instance per class, and their management corresponds exactly to the way stateless EJBs are managed. However, in this case, you don't have to think about pooling because there is only one instance.



• Stateless EJB: we can model the overall lifecycle with two states: existing and non-existing. Once a business method is called, there is a transition, some actions are performed, some values are returned, and then the EJB returns to exactly the same state it was in.



• Stateful EJB: in this case pooling is not possible because it does not make sense. Every time the same client wants to interact with the system, it needs to retrieve exactly the same instance. Therefore, you cannot have another instance taken out of the pool to be given to that specific client. What you can do is keep a sort of list (perhaps a hash table) of Stateful EJBs, and every time a request towards a specific EJB comes from a client, you will have to pick up exactly the requested one and use it.

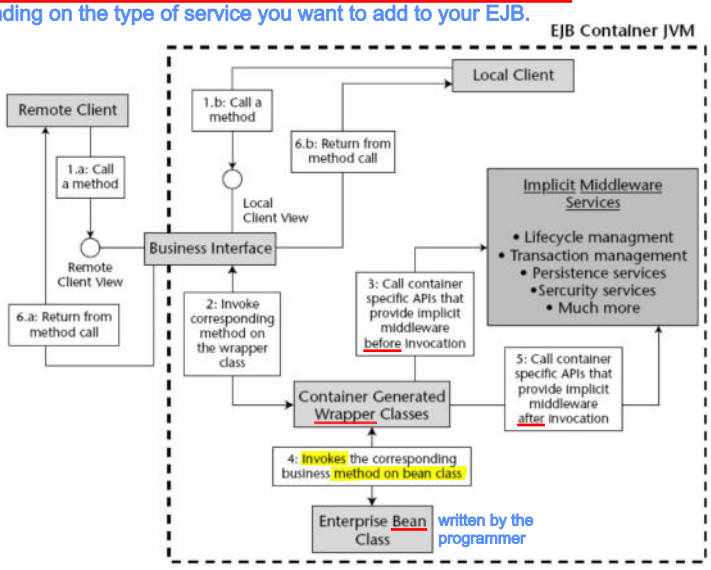
Ditto for Singletons (not pooled!)

Stateful EJBs require significant resources because, if you have 1000 clients at the same time, you will need 1000 Stateful EJBs ready simultaneously. Each of them will likely have state information to keep (like a shopping cart for each client), so you need a lot of memory to have all of them ready for the next call of a business method. However, our resources are limited, so we can think of a trick to limit the necessary resources.
You may notice that, at a certain point, many clients are asking for Stateful EJBs, but it's unlikely that all of them will request exactly the same service at the exact same time. Therefore, you can think that if a certain EJB has not been requested for a certain duration of time, you may save it on your disk. The next time it is called, it's up to the container to retrieve the information and set up the EJB again, so it will be able to respond back to the client.
In this way, you can have many EJBs even if the active ones are really a few. All the others are somehow suspended but are kept on the hard disk. So the overall lifecycle is modified by adding the "passive" state. It's up to the container to decide, in some cases, to move the EJB from the "method-ready" state to the "passive" one. In the passive state, everything is moved to the hard disk, freeing up space in memory. This way, we can serve more clients.

Call of an EJB

We mentioned that the container provides additional services to the EJB, but how is it done? This is accomplished by executing certain actions before or after the invocation of a specific business method, depending on the type of service you want to add to your EJB.

The client obtains a reference to an EJB instance through either *dependency injection*, (using Java annotations) or *JNDI lookup*.
(Java Naming and Directory Interface)
Other possibility:
Not a business interface, but using a no-interface view.



In un ambiente Java EE, c'è un singolo EJB container che gestisce tutti gli EJB all'interno della stessa JVM.

• JNDI lookup: corresponds somehow to what is done in RMI. Remember that RMI is always the underlying technology here. In this case, the client performs an explicit search to find the EJB based on its unique name within a repository where symbolic identifiers are kept. These names are associated with actual references. So, the way to obtain a reference is just to ask the repository, and this can be done by means of a standard technology, which is JNDI.

• Dependency Injection: Instead of explicitly writing the code to retrieve the reference, you insert some annotations directly into the code. These annotations serve as indications to the EJB container, meaning, "Please take care of this variable." I do not write any new statement here to generate one specific instance of a certain class. So, no explicit creation. It will be the server, or more precisely, the application container in the context of Java EE, that takes care of "injecting" a reference to the requested object (an EJB) into the client code when the client component is instantiated. It's up to the container to create the EJB, initialize it, obtain a reference, and make such a variable written in the code be bound to that reference.

(By the term "injection," we mean that instead of the client creating dependencies or objects it needs by itself, these dependencies are provided or "injected" by an external system.)

Il diagramma illustra un contenitore EJB gestito dalla JVM. All'interno, è presente tutta l'infrastruttura necessaria per la gestione degli EJB.
Ora vedremo i passaggi necessari per una chiamata a un EJB:

1 Ottenimento di un Riferimento all'EJB:
• Client Remoto: Il client remoto ottiene un riferimento a un'istanza dell'EJB attraverso l'iniezione di dipendenza (utilizzando annotazioni Java) o attraverso una ricerca JNDI.
• Local Client: Il client locale, che risiede nello stesso ambiente JVM del bean, potrebbe ottenere il riferimento in modo simile o potrebbe utilizzare un riferimento diretto se già noto.

2 Invocazione del Metodo EJB:
Si accede all'EJB attraverso l'interfaccia di business, che può essere sia locale che remota.

• Client Remoto: Il client remoto invoca un metodo sull'EJB utilizzando la vista remota (Remote Client View). Questo comporta una chiamata di rete al server dove risiede l'EJB.
• Local Client: Il client locale invoca un metodo sull'EJB utilizzando la vista locale (Local Client View). Questa invocazione avviene all'interno della stessa JVM e quindi non richiede una chiamata di rete.

3,4,5 Invocazione del Metodo di Business & API
Il contenitore EJB genera automaticamente delle classi wrapper che fungono da intermediari tra il client e l'Enterprise Bean Class. Queste classi non vengono scritte direttamente dallo sviluppatore, ma sono create dal contenitore EJB per semplificare l'interazione tra il client e l'EJB. Oltre a inoltrare le chiamate ai metodi di business appropriati, queste classi si occupano anche di gestire aspetti trasversali come la sicurezza e la gestione delle transazioni.
Inoltre, queste classi wrapper svolgono un ruolo fondamentale nell'invocazione di API specifiche del container prima e/o dopo la chiamata ai metodi di business. Queste API forniscono servizi middleware impliciti come la gestione del ciclo di vita, la gestione delle transazioni, i servizi di persistenza e la sicurezza. Tutto questo processo è gestito internamente dal contenitore ed è trasparente sia per il client remoto che per quello locale.

6 Ritorno dal Metodo Chiamato
• Client Remoto: Il risultato dell'invocazione del metodo (o un'eccezione, se ne viene generata una) viene restituito al client remoto attraverso la rete. Questo passaggio completa il ciclo di chiamata-ritorno per il client remoto.
• Local Client: Analogamente, il risultato o l'eccezione viene restituito al client locale direttamente all'interno della stessa JVM.

Session EJBs and Asynchronous Calls

EJBs have the capability to implement asynchronous behavior, similar to the behavior we discussed when describing the implementation of asynchronous procedures in Java using executors.

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.

Instead of receiving the actual result, you may obtain a Future object, and it's up to the client to query this object about the current state of the progression of the computation. Similar to what we saw for executors, you need to specify everything: how to submit the job, how to get the Future object. However, in this case, using annotations, everything will be automatically managed by the system, and you only need to annotate the asynchronous method with the keyword "asynchronous."

"This implies that to implement an asynchronous behavior in an Enterprise application, you only need to understand what a Future object is and simply annotate the method that will return a future result as asynchronous."

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 (Java Naming and Directory Interface)

è un'API Java fornita dal Java SE e Java EE che fornisce servizi di naming e directory.
Per "naming" si intende l'associazione di un nome a un oggetto, mentre con "directory" si intende una funzionalità estesa di naming che consente non solo di associare nomi agli oggetti, ma anche di cercare oggetti in base a vari attributi associati a essi.

Directory Services (in general)

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. (e.g. file)

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.

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.

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.

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.

This means that it typically contains the context, which corresponds to the JNDI service set up on the local application server, if you desire that one. Otherwise, you can specify to use, as the initial context, a service placed somewhere else. This is the way to locate the repository corresponding to the JNDI service. So, when you want to access such a service, you have to set up the initial context.

Basic JNDI Lookup

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

If you want to refer to the local one, typically, you don't have to specify any configuration parameters. Instead, if you need to refer to some external services or if you want to specify additional configuration, the standard way to do this is by putting all the configuration parameters in a hash table.

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

(costruttore di Context)

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!

Suggerimento: Per trovare il nome di una nostra EJB, ci sono delle regole da seguire, ma per non sbagliare il symbolic name associato alle EJB, possiamo (dopo averla deployata) dare un'occhiata ai log. Troveremo il nome corretto scritto lì.

EJBs will be published to the JNDI repository with a certain name, and this will be done automatically by the container when you deploy the EJB. So, at initialization time, you will instruct the EJB container to manage a certain type of bean, and the container will decide, according to your configuration files, how to name such EJB. It will be deployed, and the name will be made available on the JNDI service. So, upon deployment, you will have a way to access it by name. Now, what you need to do is to look up such an object. This can be done by starting with "myContext" using the lookup method.
Note: We don't know what type of object, in general, will be associated with such a symbolic name, but as a programmer, you know about this. Therefore, you know how to downcast the object returned by the lookup operation so that you can keep it in a variable whose type is the one you decided to use.

JNDI Names for Session EJBs

There are different scopes for JNDI names, and each of them has its own naming pattern to follow:

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>]

- Global Scope: Questo è lo scope più ampio. Un nome globale è unico all'interno dell'intero ambiente JNDI e può essere risolto da qualsiasi parte dell'applicazione, a prescindere da dove si trova. In un'applicazione distribuita su più server o in diverse posizioni, un nome globale punta alla stessa risorsa da qualsiasi punto della rete.
- Application Scope: Un nome definito all'interno dello scope di un'applicazione è valido e accessibile da qualsiasi parte dell'applicazione stessa, ma non oltre. Questo è tipico in applicazioni enterprise, dove diverse applicazioni possono essere distribuite sullo stesso server ma devono mantenere spazi dei nomi separati.
- Module Scope: Questo scope è limitato a un modulo specifico all'interno di un'applicazione. Ad esempio, in un'applicazione Java EE, potrebbe esserci uno scope separato per ogni modulo EJB o web (WAR, EAR, JAR). Risorse come EJB o DataSource definite in uno specifico modulo non sono accessibili al di fuori di quel modulo.

Context/Dependency Injection - CDI

Context and Dependency Injection (CDI) è una parte essenziale della piattaforma Java EE che permette agli sviluppatori di gestire le dipendenze in modo tipo-sicuro e di legare il ciclo di vita degli oggetti ai contesti di esecuzione. CDI si focalizza sulla facilitazione dello sviluppo di componenti enterprise che possono essere facilmente collegati (wired) e scollegati (unwired), migliorando la modularità, la riusabilità e la manutenibilità del codice.

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.

Ad esempio, un bean può avere un ciclo di vita legato a una richiesta HTTP (RequestScoped), a una sessione utente (SessionScoped), all'applicazione nel suo complesso (ApplicationScoped) o a un contesto personalizzato. Questo controllo sul ciclo di vita permette di gestire efficacemente lo stato dei componenti all'interno dell'applicazione.

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.

flessibilità nell'iniezione delle dipendenze: gli sviluppatori possono definire più implementazioni di un'interfaccia e configurare l'applicazione per selezionare quale implementazione iniettare a tempo di distribuzione. Questo si realizza tramite le annotazioni di qualificatori che consentono di distinguere tra diverse implementazioni dello stesso tipo.

- Loose coupling si riferisce alla riduzione delle dipendenze dirette tra diverse parti di un programma. In un sistema con accoppiamento lasso, i moduli o classi sono progettati in modo da essere il più possibile indipendenti gli uni dagli altri.
- Strong typing significa che il tipo di ogni variabile è conosciuto in fase di compilazione, e il linguaggio di programmazione impedisce operazioni non consentite tra tipi incompatibili.

La tipizzazione forte aiuta a prevenire errori di tipo in fase di esecuzione e rende il codice più chiaro e più facile da comprendere. In combinazione con IoC e loose coupling, promuove un design del software in cui le interazioni tra i componenti sono ben definite e sicure dal punto di vista del tipo, riducendo così gli errori a runtime e aumentando la manutenibilità del codice.

CDI - Context & Dependency Injection

Fosters a more effective separation between business tiers.
It promotes loose coupling

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

Dependency injection: Ability to inject components into an application in a typesafe way, and to 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*

Inversion of Control è un principio secondo il quale il controllo del flusso del programma è invertito rispetto a quello che si ha in un programma tradizionale. Invece che il programmatore controllare il flusso e le dipendenze tra gli oggetti, è il framework o il container (come nel caso di un'applicazione Java EE con CDI) a prendere il controllo, gestendo la creazione e la gestione degli oggetti e delle loro dipendenze.
IoC è spesso implementato attraverso l'iniezione delle dipendenze, dove le dipendenze di un oggetto (come altri oggetti o servizi di cui ha bisogno) non sono create dall'oggetto stesso, ma gli vengono fornite (iniettate) da un sistema esterno (container o framework).

CDI for EJBs by Annotations

Example:

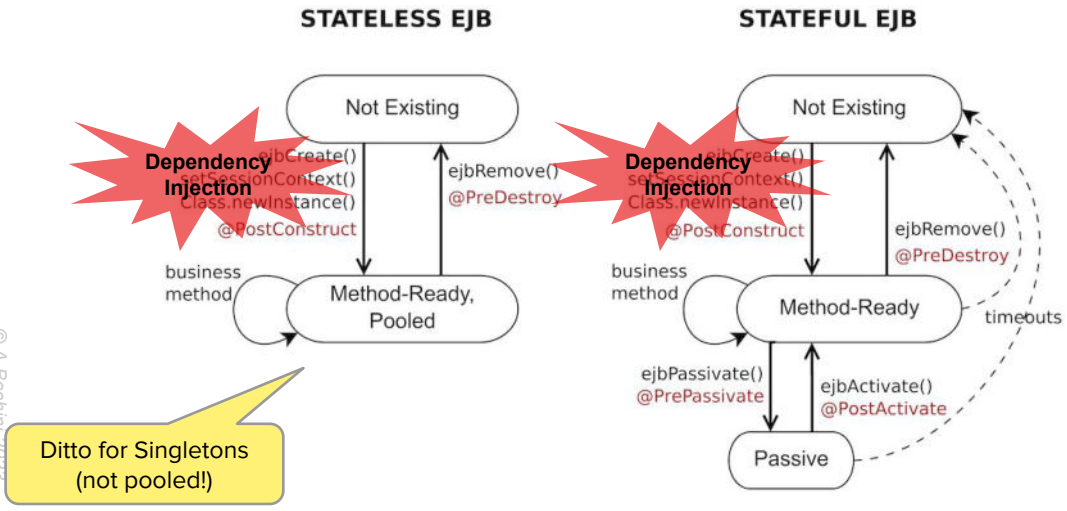
notiamo che è una Servlet

```
@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



"Everything is performed by means of dependency injection."

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.

"In most modern applications, the use of deployment descriptors is no longer common because developers prefer specifying configurations through annotations."

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.

viene attivato automaticamente dal container EJB per elaborare i messaggi in arrivo

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

Notes:

Quando un messaggio arriva in una destinazione JMS (Java Message Service) a cui un MDB è in ascolto, il contenitore seleziona un'istanza dal pool per elaborare il messaggio. Dopo che l'elaborazione è terminata, l'istanza del MDB viene restituita al pool anziché essere distrutta, pronta per essere riutilizzata per un altro messaggio in arrivo.

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

Java Message System

Java Message Service (JMS) è una specifica API della piattaforma Java EE che permette agli sviluppatori di creare, inviare, ricevere e leggere messaggi. Fornisce un modo affidabile e asincrono per integrare diversi sistemi e componenti di un'applicazione enterprise, garantendo che le informazioni possano essere scambiate senza necessità di una connessione immediata e diretta tra le parti coinvolte.

MOM - Message Oriented Middleware

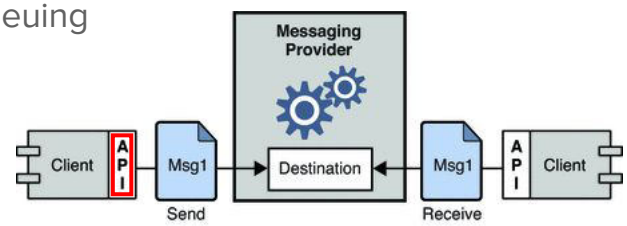
The communication is indirect because you have to refer to a broker system that will take care of obtaining the information you want to transmit. The delivery of the sent information will be done later, and the sender is not immediately informed about the message's delivery.

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

Advantages:

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.

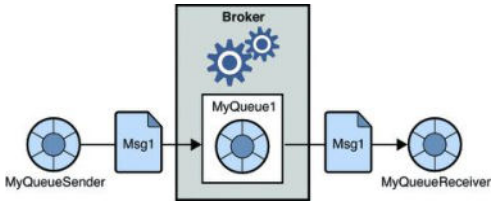
Models:

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

i messaggi vengono pubblicati su argomenti specifici da parte dei mittenti e i destinatari interessati si iscrivono a questi argomenti. I messaggi vengono quindi consegnati a tutti i destinatari che sono sottoscritti a un determinato argomento. (comunicazione uno-a-molti)

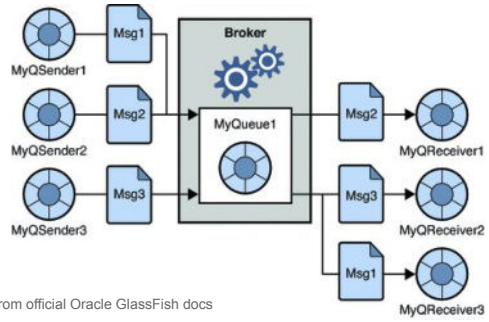
i messaggi vengono inviati da un sender a un dest attraverso una coda di messaggi. Il sender e il dest non sono consapevoli l'uno dell'altro e la coda agisce come intermediario per la consegna del messaggio, infatti il sender specifica solo la coda in cui il messaggio deve essere inviato, ma non specifica nulla sul destinatario finale. (comunicazione uno-a-uno)

JMS Point-to-Point Messaging



Simple scenario:
1 sender, 1 receiver

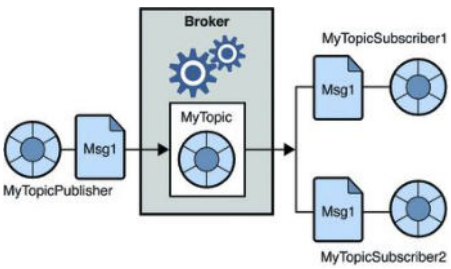
Complex scenario:
1+ senders, 1+ receivers,
with possibly shared connections.
No hypotheses on the order msgs are consumed.

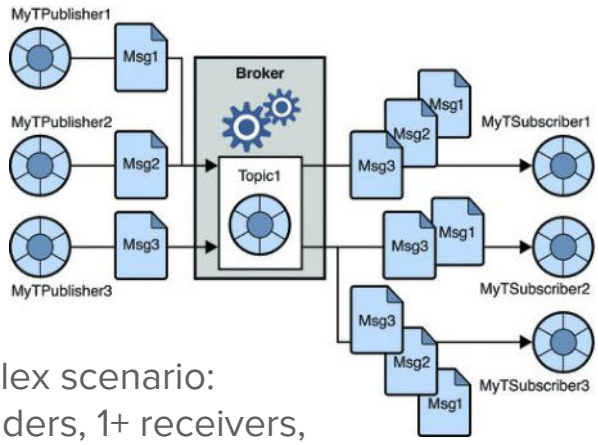


Images from official Oracle GlassFish docs

JMS Publish/Subscribe Messaging

With this model, every message sent will be delivered to ALL the subscribers subscribed to that topic.





Simple scenario:
1 sender, 1+ receivers

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

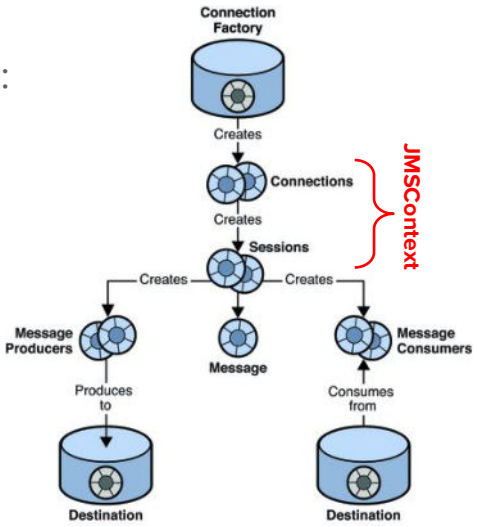
Images from official Oracle GlassFish docs

Why in the second figure message 2 is not delivered to subscriber 2? We don't know. What we can state is that at the time of sending message 2, subscriber 2 was not subscribed to the topic of message 2.

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

- We have a "connection factory" used to generate connections. When you need a connection, you can request it from the connection factory.
- A "connection" represents a communication channel to the broker. Once you have a connection, over that connection, you can set up a session.
- A "session" object is used for implementing the conversation between the client and the broker.
- Producer objects and consumer objects are the two endpoints. They are also objects and are used to send or get messages.
- A "message" is, of course, an object itself, made up of a header, properties, and body.

Out of the connection factory, you will obtain connections. Out of connections, you will obtain sessions. Sessions will be used to create message producers and message consumers, as well as messages. Messages will be sent to destinations.

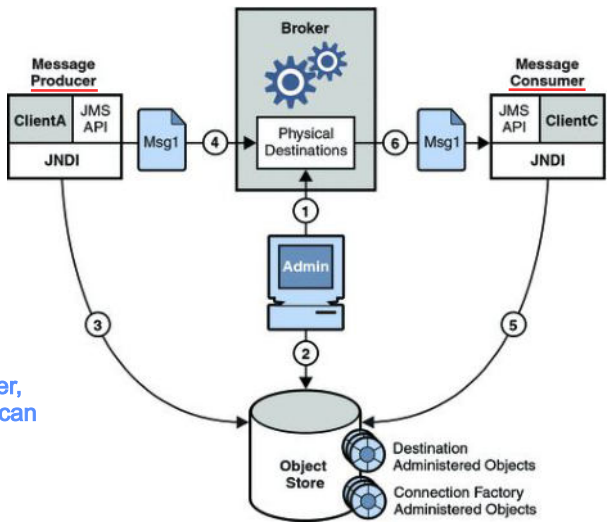
JMS - Administered Objects

- 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."

Once you have these resources in your application server, they are given specific identification names so that they can be looked up by means of usual JNDI methods.



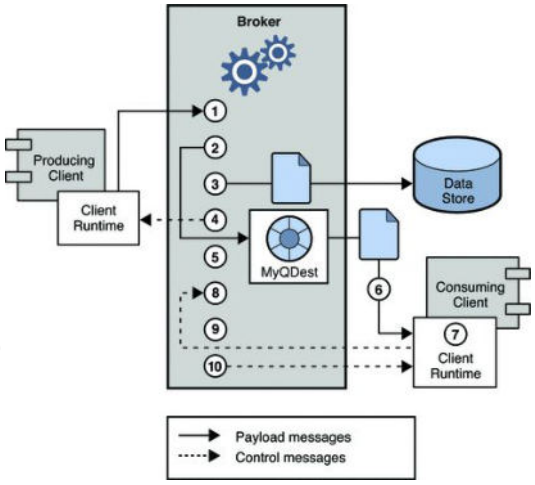
"If you want to use a connection factory or if you have to use a destination, typically, this is not something you can do programmatically. You have to enter the configuration interface and specify them there. So, they are resources belonging to the application server."

Here, by "destination," we don't mean the client ready to receive messages, but rather the queue that is supposed to receive the messages (for example, MyQDest from the next slide).

Once you have these resources in your application servers, you can use them for your program. Again, as these resources belong to the application server, they are given specific identification names, so they can be looked up by means of JNDI. They are dealt with exactly the same way as the other objects we have seen before (like the EJBs that are deployed on the relative container in the application server). Therefore, it's up to the administrator to interact with the broker and specify the physical destination. At that time, you will have the destinations and the connection factories in the object store. Once you have these, client A can refer to these objects to set up communication, and the same can be done by the message consumer.

So the steps are:
1) The admin configures the broker module on the application server. Destinations and the connection factory are stored in the Object Store.
2) The producer can look them up programmatically and can send the message to the physical destination.
3) The message consumer can know, by means of JNDI lookup action, about the physical destination of the broker, and it will be able to connect to it and get the message.

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) if you choose to store them (to prevent loss in case of a crash or when the recipient is currently unavailable)
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

Please note that the sending and receiving of messages are handled by the client runtime part of the system. You should be aware that it's there, but you don't have to write any code for it. The programmer just makes use of the standard API.

Not Only JMS Deserves Mention...

Message Brokers:

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

Standards:

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