

# Arquitectura de componentes JEE

(Java Enterprise Edition)



# Índice

- 
1. Introducción Java Empresarial
  2. Enterprise Java Beans (EJBs) y Context Directory Inyection (CDI)
  3. Java Persistence Api (JPA)
  4. Servlets y jsps
  5. JavaServer Faces
  6. Web Services (SOAP y REST)
  7. Seguridad en JEE

# 1. Introducción

- Vivimos en un mundo globalizado, donde la **eficiencia** y **productividad** de las empresas es un factor crucial para el éxito o fracaso de las mismas.
- Los **Sistemas de Información** juegan un papel fundamental en la mejora y consolidación de las compañías.
- Java permite crear aplicaciones para usuarios de distintos tipos como son clientes de Escritorio, Web y Móviles.

# 1. Introducción

- Las aplicaciones empresariales Java tienen a su cargo establecer las reglas de negocio de la empresa y/o sistema y ofrecer conectividad a los distintos tipos de clientes, con ello se logra ofrecer una solución integral a sus necesidades de sistemas de información a la medida.

# 1. Introducción

- La versión empresarial de Java (Java Enterprise Edition) cuenta con una **enorme comunidad de programadores** alrededor del mundo.
- A su vez, una de las mayores ventajas de Java es su **independencia de plataforma**, de esta manera, el programador puede seleccionar entre utilizar herramientas de pago o de software libre, y esto comienza desde el sistema operativo, hasta las herramientas de desarrollo, el servidor de aplicaciones, la base de datos, etc.

# 1. Introducción

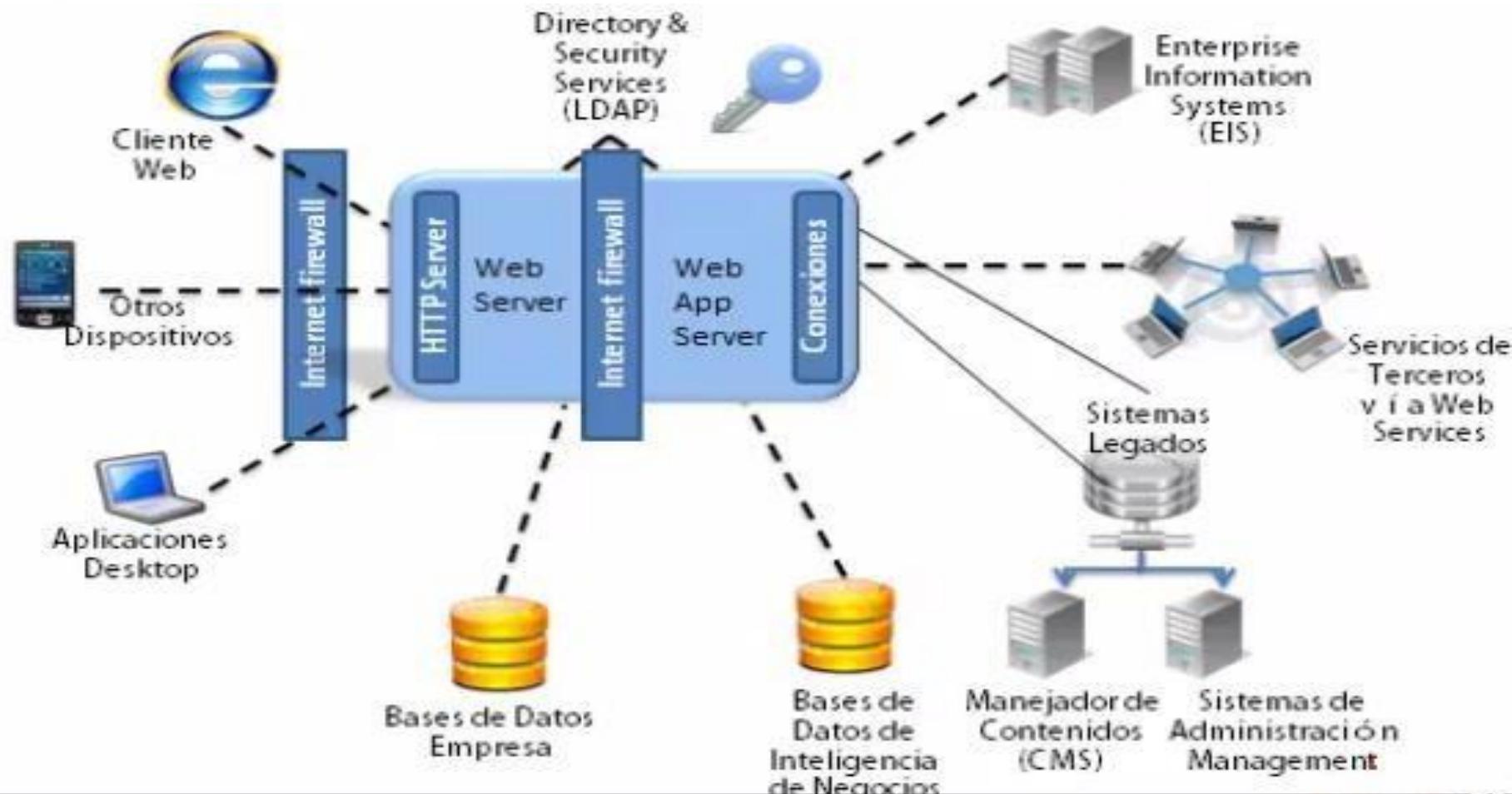
- ¿Qué es una aplicación empresarial?
- La plataforma Java Empresarial (JEE) está diseñada para crear aplicaciones que den servicio a clientes, empleados, proveedores u otras entidades o personas relacionados con el negocio de la empresa.

# 1. Introducción

- **¿Qué es una aplicación empresarial?**
- Este tipo de aplicaciones suelen ser complejas
  - **múltiples fuentes de datos**
  - **distribuyen información a numerosos usuarios** a menudo ubicados en localizaciones diferentes que pueden llegar desde múltiples oficinas en un mismo edificio hasta múltiples oficinas en múltiples edificios en diferentes países si la empresa tiene un alcance internacional.

# 1. Introducción

- ¿Qué es una aplicación empresarial?



# 1. Introducción

- ¿Qué es una aplicación empresarial?
- Además de la complejidad inherente a su naturaleza distribuida, estas aplicaciones tienen que cumplir **requisitos de seguridad**, ya que a menudo mueven datos de negocio que pueden ser confidenciales para la empresa y deben ser **portables** para poder ejecutarse en diferentes plataformas, desde un ordenador Mac o Windows de escritorio hasta una aplicación desplegada en un tablet..

# 1. Introducción

- ¿Qué es una aplicación empresarial?
- La plataforma JEE utiliza el modelo multinivel distribuido para crear aplicaciones empresariales.
- La lógica completa de la aplicación es dividida en diferentes niveles que son desplegados a su vez en diferentes ordenadores conectados en red. Así tenemos:
  - El nivel de cliente, que se ejecuta en los dispositivos u ordenadores clientes
  - El nivel Web, que se ejecuta en el servidor de aplicaciones JEE
  - El nivel de negocio, que se ejecuta en el servidor de aplicaciones JEE
  - El nivel de sistemas de información o datos, que se ejecuta en el servidor de datos

# 1. Introducción

- **¿Qué es una aplicación empresarial?**
- **Ventajas de la separación en niveles**
  - Los requerimientos de procesamiento en cada nivel pueden ser diferentes.
  - Es más fácil escalar en vertical (añadiendo más CPU, memoria o espacio) o en horizontal (añadiendo nuevos nodos a una estructura en cluster)
  - Se pueden tratar diferentes requerimientos de seguridad
  - Es posible compartir lógica de negocios entre distintas aplicaciones
  - Facilita las labores de administración de nuestro centro de datos

# 1. Introducción

## ¿Qué es J2EE?

- Para entender qué es Java EE, comencemos por responder a la pregunta de ¿Qué es un API?. Un API (Application Programming Interface) es un **conjunto de clases que resuelven una necesidad muy particular**.
- Por ejemplo el API de JDBC permite crear código Java para establecer la comunicación con una base de datos.
- Java EE es un conjunto de **API's enfocadas en brindar una serie de servicios que toda aplicación empresarial necesita**, tales como: transaccionalidad, seguridad, interoperabilidad, persistencia, objetos distribuidos, entre muchos servicios más.

# 1. Introducción

- ¿Qué es J2EE?
- Estas APIs se basan en un conjunto de especificaciones, las cuales pueden ser implementadas por empresas orientadas a software libre (Tomcat, Jboss, etc) o software comercial (Oracle, IBM, etc).
- Algunas de estas APIs empresariales son:
  - Manejo de Transacciones: Java Transaction API (JTA)
  - Persistencia: Java Persistence API (JPA)
  - Mensajería: Java Message Service (JMS)
  - Manejo de Servicios Web: Java API for XML Web Services (JAX-WS) y Java

# 1. Introducción

## ¿Qué es J2EE?

- API for RESTful Web Services (JAX-RS)
- Seguridad: Java Authentication and Authorization Service (JAAS)
- Localización de objetos: Java Naming and Directory Interface (JNDI)
- Entre muchas APIs más
- Una de las grandes ventajas de seleccionar estas tecnologías es que **son el estándar propuesto por el grupo JCP** (Java Community Process), el cual se encarga de revisar y liberar las especificaciones Java y las APIs empresariales respectivas.

# 1. Introducción

## ¿Qué es J2EE?

- En resumen, la versión empresarial de Java se puede entender como **una extensión de la versión estándar (JSE)**, pero con la intención de **facilitar el desarrollo de aplicaciones empresariales**, permitiendo agregar de manera muy simple los servicios descritos anteriormente, y así crear aplicaciones Java robustas, poderosas, y de alta disponibilidad.

# 1. Introducción

## Historia JEE

- Uno de los principales requerimientos en su primera versión fue el **manejo de sistemas distribuidos**, los cuales consisten en poder ejecutar componentes en distintos servidores. CORBA era la tecnología utilizada para cubrir este requerimiento.
- Java, por su lado, propuso el protocolo **RMI-IIOP** (Remote Method Invocation–Internet Inter- ORB Protocol) para cubrir este tipo de requerimientos de llamadas distribuidas, y con este concepto en mente fue que liberó la primera versión de los **Enterprise JavaBeans (EJBs)**. Además se introdujeron tecnologías como **Servlets** y **JSPs** para la creación de aplicaciones Web, y temas de mensajería con **JMS**.

# 1. Introducción

## Historia JEE

- En las siguientes versiones se lograron varios objetivos, tales como hacer las **aplicaciones más robustas y escalables**. Posteriormente se incluyó el tema de **Web Services** como parte del estándar.
- La versión Java EE 5 dio un giro sustancial en la **simplificación de la programación** de aplicaciones empresariales. Esto como respuesta a que varios frameworks libres como Struts, Spring y Hibernate simplificaron la programación **promoviendo el uso de clases puras de Java (POJO's)**. Además, el concepto de anotaciones permitió simplificar la configuración de clases como los EJB's, y se hizo opcional la configuración de estas clases y archivos xml.

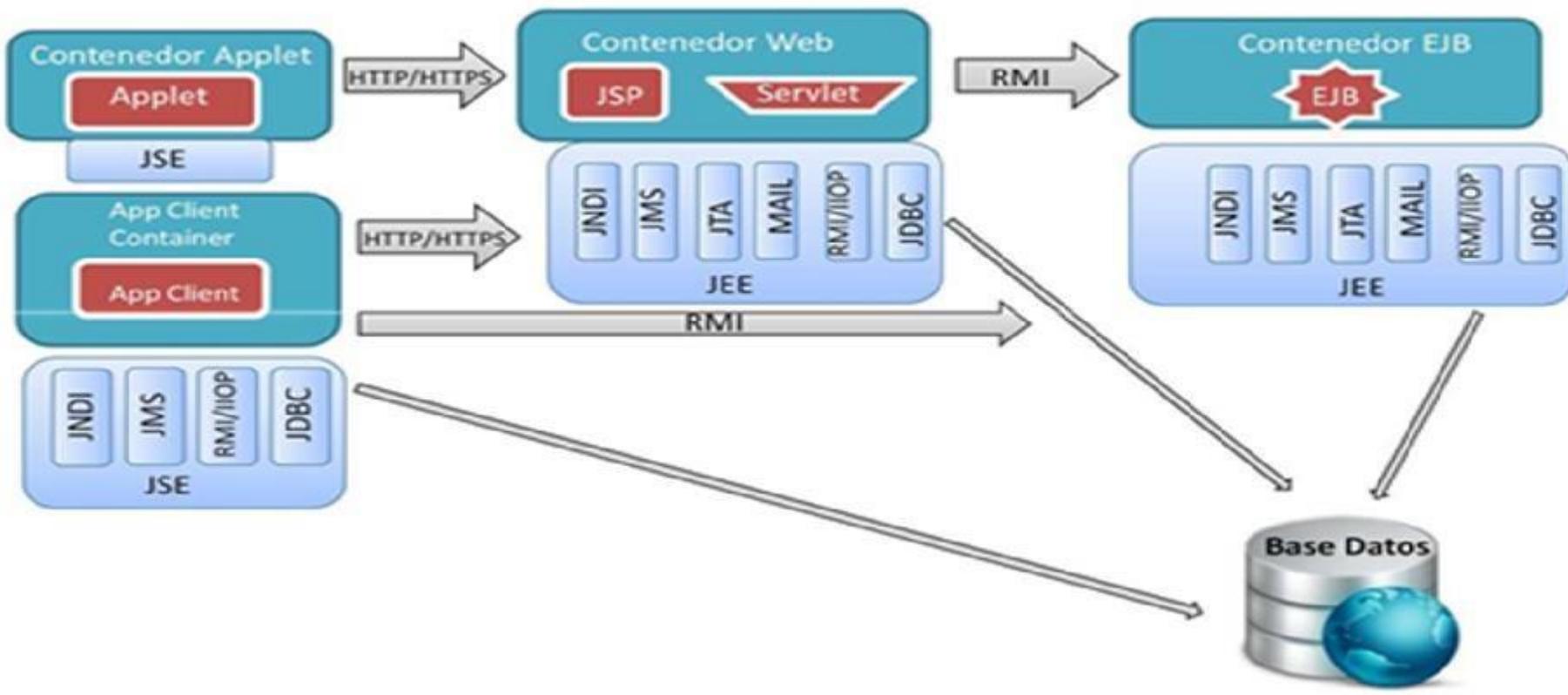
# 1. Introducción

## Historia JEE

- Las diferentes versiones de JEE [https://en.wikipedia.org/wiki/Java\\_EE\\_version\\_history](https://en.wikipedia.org/wiki/Java_EE_version_history) tienen como objetivo **simplificar** la programación de requerimientos empresariales, y facilitar la integración entre las diferentes tecnologías, a través de conceptos como **CDI** (Context and Dependency Injection), creación y **ejecución de pruebas unitarias** a través de **contenedores empresariales embebidos**, selección de perfiles según las tecnologías a utilizar, y muchas mejoras más.

# 1. Introducción

## Tecnologías Empresariales JEE



# 1. Introducción

## Tecnologías empresariales JEE

- La Tecnología Empresarial JEE incluye muchas mejoras en cada una de las tecnologías que la componen, en particular se enfoca en simplificar la integración de varios componentes a través del concepto de CDI (Contexts and Dependency Injection), el uso de anotaciones y el uso de POJOs (Plain Old Java Objects).
- Algunas de las tecnologías más importantes son:
  - Enterprise JavaBeans (EJB)
  - Servlets
  - JavaServer Pages
  - JavaServer Faces (JSF)

# 1. Introducción

## Tecnologías empresariales JEE

- Java Persistence API 2.0
- Java Transaction API
- Java API from XML Web Services (JAX-WS) y Java API for RESTful Web Services
- Context and Dependency Injection (CDI)
- Java Message Service API (JMS)
- JavaMail API
- Java Naming and Directory Interface API (JNDI)
- Java Authentication and Authorization Service (JAAS)

# 1. Introducción

## Java Enterprise Edition

En una aplicación JEE podemos utilizar tecnologías como EJB, JPA, Web Services, entre muchas más.



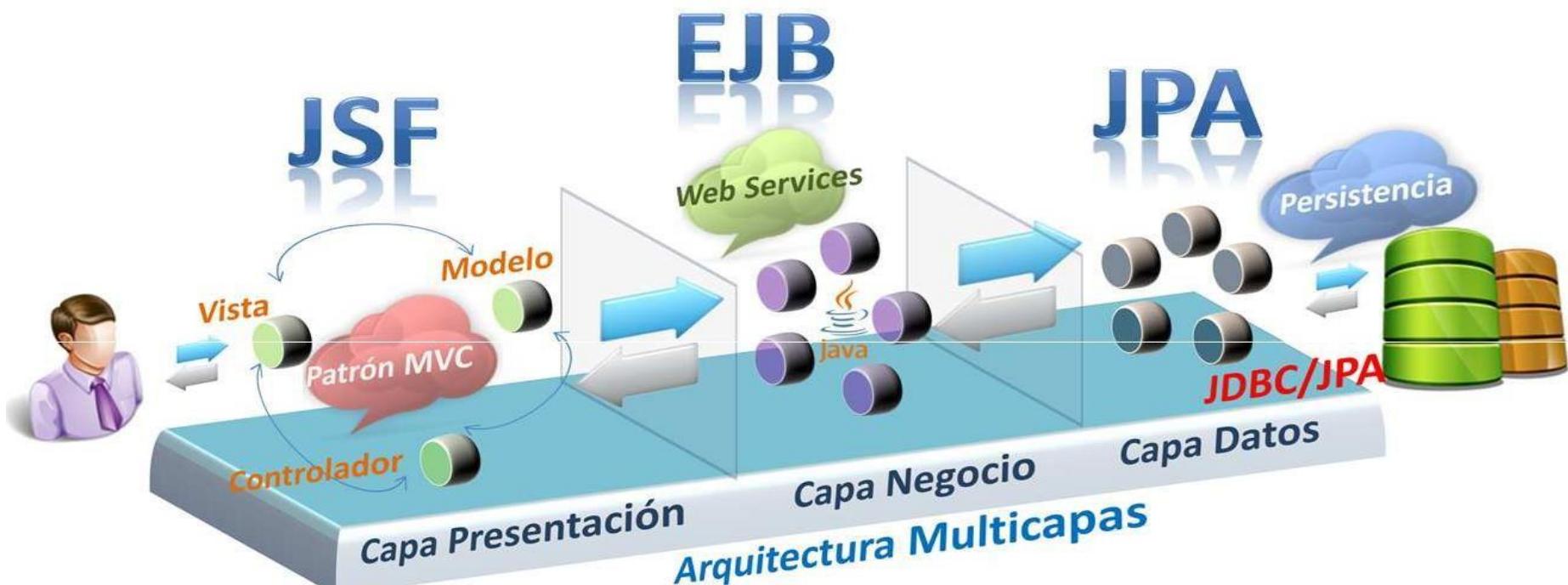
# 1. Introducción

## Arquitectura Multicapa

- Una aplicación empresarial en Java se compone de **distintas capas**, cada capa tiene una función muy específica.
- Dividir una aplicación en capas tiene varias ventajas:
  - como son separación de responsabilidades,
  - un mejor mantenimiento a la aplicación,
  - especialización de los programadores en cada capa,
  - entre muchas más.
- La versión empresarial de Java brinda un API distinta para cada capa de una aplicación empresarial, desde la capa de presentación, la capa de negocio y la capa de datos.

# 1. Introducción

## Arquitectura Multicapas



Servidor de  
Aplicaciones Java

# 1. Introducción

- A continuación mencionaremos cada una de las capas de una aplicación multicapas.
  - **Capa Web:** La capa del Cliente es donde el cliente interactúa por medio de un navegador Web, un cliente móvil, una aplicación de escritorio, entre otros.
  - **Capa Web:** la capa web que puede residir en un servidor web, las tecnologías mas básicas que podemos encontrar en este servidor web son los JSP"s y los Servlets o JavaSever Faces.
  - **Capa de Negocio:** en esta capa podemos encontrar tecnología como son los Enterprise Java Beans (EJBs).
  - **Capa de Datos:** Aquí vamos a encontrar tecnologías como JDBC, o JPA. Este código nos va a permitir comunicarnos con nuestra base de datos para leer y almacenar información en ella.

# Índice

- 
1. Introducción Java Empresarial
  2. **Enterprise Java Beans (EJBs) y Context Directory Inyection (CDI)**
  3. Java Persistence Api (JPA)
  4. Servlets y jsps
  5. JavaServer Faces
  6. Web Services (SOAP y REST)
  7. Seguridad en JEE

## 2. EJBs y CDI

### EJBs

- Los Enterprise Java Beans (EJB) es **código Java del lado del Servidor**. Normalmente tienen la **lógica de negocio** de nuestra aplicación, y por lo tanto **cubren el rol de la capa de servicio de nuestras aplicaciones Java**.
- Los EJB's son clases puras de Java (POJO's) los cuales al ser desplegados en un Servidor de Aplicaciones **permiten reducir la complejidad de programación, agregando robustez, reusabilidad y escalabilidad** a nuestras aplicaciones empresariales de misión crítica.

## 2. EJBs y CDI

### ¿Qué es un Enterprise JavaBean (EJB's)?

- Un Enterprise JavaBean es una clase de Java con características que lo hacen mucho más potente y robusto.
  - Los métodos de un EJB son transaccionales
  - Los métodos pueden ser remotos
  - Facilidad de comunicación con las bases de datos
  - Los métodos pueden ser seguros



## 2. EJBs y CDI

### Características de un EJB

- Cuando un EJB se ejecuta en un Contenedor Java EE que soporta EJB's, el contenedor agrega los siguientes servicios disponibles:



## 2. EJBs y CDI

- Los servidores de aplicaciones Java, también agregan **otras características** tales como:
  - clustering,
  - balance de cargas
  - y tolerancia a fallos.
- Esto permite crear aplicaciones de misión crítica con operaciones 7/24 los 365 días del año.
- Así que **independientemente del tipo de servidor de aplicaciones que utilicemos, tendremos todas estas características disponibles al crear y desplegar nuestros EJBs.**

## 2. EJBs y CDI

- En una arquitectura típica Java EE, los EJB juegan el rol de la capa de Servicio, donde es común encontrar muchas de las reglas de negocio de nuestra aplicación.
- Una regla de negocio son las normas o políticas de la empresa u organización,
  - por ejemplo, si un cliente ha sido leal a un producto por cierta número de años, se le puede aplicar un descuento extra por determinado monto de compra. Este tipo de decisiones se aplican automáticamente por medio de los sistemas, y la capa de negocio es la encargada de ejecutar estas reglas.

## 2. EJBs y CDI

### Configuración y tipos de EJB

- En versiones previas a EJB 3.0, el programador debía crear varias clases e interfaces para hacer funcionar un EJB: una interface local o remota (o ambas), una interface de tipo **home** local o remota (o ambas), y un archivo de configuración xml, conocido como deployment descriptor.
- Los EJB en su versión 3.0 promovió el uso de anotaciones para su configuración.

## 2. EJBs y CDI

### Configuración y tipos de EJB

- En la versión 3.2 continúa agregando y **simplificando** la integración de tecnologías empresariales a través del concepto de **anotaciones**.
- **Este concepto simplificó en gran medida el desarrollo de EJBs, y en general de toda la tecnología Java.**

## 2. EJBs y CDI

### Configuración y Tipos de EJB

Configuración de un Enterprise JavaBeans (EJB):



**Anotación**



**EJB 3**



Tipos de Enterprise JavaBeans:

- **Stateless:** No guardan estado y se utiliza la anotación `@Stateless`
- **Stateful:** Guardan estado y se utiliza la anotación `@Stateful`
- **Singleton:** Solo existe una instancia en memoria y se utiliza la anotación `@Singleton`

## 2. EJBs y CDI

- Existen diferentes **tipos de beans**, dependiendo de la función que se agrega a una arquitectura multicapas Java.
- Esta organización permite entender mejor la configuración de una aplicación empresarial.
- Debido a que las aplicaciones empresariales suelen ser complejas, se han definido los siguientes tipos de EJBs, según los requerimientos a cubrir.

## 2. EJBs y CDI

- **EJB de Sesión:** Un bean de sesión se invoca por el cliente para ejecutar una operación de negocio específica.
  - **Stateless:** Este tipo de EJB **no mantiene ningún estado del usuario**, es decir, no recuerda ningún tipo de información después de terminada una transacción.
  - **Stateful:** Este tipo de EJB, **mantiene un estado de la actividad del cliente**, por ejemplo, si se tiene un carrito de compras. Este estado se puede recordar incluso una vez terminada la transacción, pero si el servidor se reinicia esta información se pierde. Es similar al alcance Session de una aplicación Web.
  - **Singleton:** Este tipo de beans utiliza el patrón de diseño *Singleton*, en el cual **sólo existe una instancia en memoria de esta clase**.

## 2. EJBs y CDI

- Otras clasificaciones que podemos encontrar son:
  - **EJB Timer**: Esta es una característica que se puede agregar a los beans, para que se **ejecuten en un tiempo especificado** (scheduling).
  - **Message-driven beans (MDBs)**: Este tipo de beans se utiliza para **enviar mensajes utilizando la tecnología JMS**.
  - **Entity Beans**: Esta es una clasificación anterior a la versión 3.0 de los EJB. Al día de hoy el estándar JPA (Java Persistence API) ha sustituido a este tipo de beans. Así que, a menos que estemos utilizando una versión anterior a 3.0, se debería utilizar JPA en lugar de los Entity Beans.

## 2. EJBs y CDI

- Los EJBs pueden ser configurados de la siguiente forma, con el objetivo de permitir la comunicación con sus métodos:
  - **Interfaces de Negocio:** Estas interfaces contienen la declaración de los métodos de negocio que son visibles al cliente. Estas interfaces son implementadas por una clase Java.
  - **Una clase Java (bean):** Esta clase implementa los métodos de negocio y puede implementar cero o más **Interfaces de Negocio**. Dependiendo del tipo de EJB, esta clase se debe anotar con **@Stateless**, **@Stateful** o **@Singleton** dependiendo del tipo de EJB que deseemos crear.

## 2. EJBs y CDI

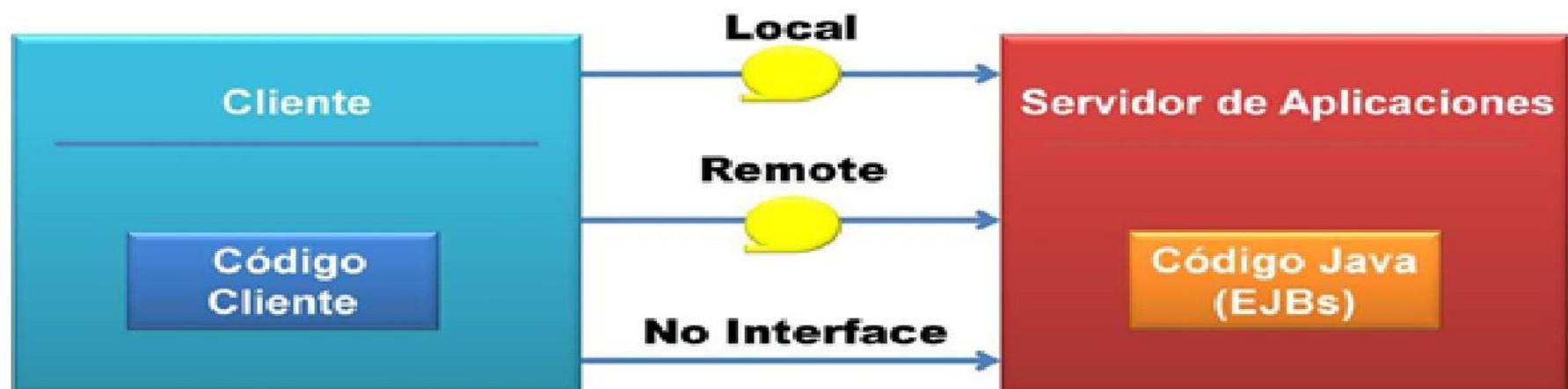
### Formas de comunicarnos con un EJB

- Existen diferentes formas de comunicarnos con nuestro componente EJB.
  - **Interfaz Local:** Se utiliza cuando el cliente se encuentra dentro del mismo servidor Java, de esta manera se evita la sobrecarga de procesamiento al utilizar llamadas remotas vía RMI.
  - **Interfaz Remota:** Se utiliza cuando el código del cliente está fuera del servidor de aplicaciones Java (en una Java Virtual Machine distinta) y por lo tanto debemos hacer llamadas remotas para poder ejecutar los métodos del EJB.
  - **No Interface:** Es una simplificación en la versión 3.1, ya que no se requiere de una interfaz para establecer la comunicación, siempre y cuando las llamadas sean locales, es decir, dentro del mismo servidor de aplicaciones Java.

## 2. EJBs y CDI

### Formas de comunicarnos con un EJB

Existen diferentes formas de comunicarnos con un EJB:



- Interfaz Local: Se utiliza cuando el cliente se encuentra en el mismo servidor Java.
- Interfaz Remota: Se utiliza cuando el cliente se encuentra fuera del servidor Java
- No Interface: Es una variante y simplificación de los EJB locales.

## 2. EJBs y CDI

### Anatomía de un EJB

- En la siguiente figura podemos observar la estructura general de un EJB, el cual puede implementar o no una interface (local o remota), y puede tener uno o más métodos de negocio:

```
1 package beans;
2
3 import javax.ejb.Stateless;
4
5 @Stateless
6 public class HolaMundoEJB {
7
8     public int sumar(int a, int b) {
9         return a + b;
10    }
11
12 }
```

Puede implementar o no una interface

Método de Negocio

## 2. EJBs y CDI

### Anatomía de un EJB

- Previo a la versión J2EE se requería crear varias clases para hacer funcionar a un EJB: una **interfaz local o remota** (o ambas), un interfaz home local o remota (o ambas) y un descriptor de despliegue xml.
- La versión Java EE 5 y EJB 3.0 simplificó dramáticamente esta configuración agregando el concepto de anotaciones, sin embargo todavía se requería agregar una interfaz a los EJB, local o remota.

## 2. EJBs y CDI

### Anatomía de un EJB

- Desde la versión Java EE 6 y EJB 3.1 permite convertir una clase pura de Java (POJO) en un EJB simplemente agregando la anotación del bean correspondiente, por ejemplo @Stateless.
- **Esto hace que esta clase tenga características como métodos transaccionales, métodos con seguridad, y puede acceder al manejador de entidades** (entity manager) y así persistir información en la base de datos, entre muchas características más.

## 2. EJBs y CDI

### Anatomía de un EJB

- Otra forma de configurar un EJB es utilizando el **archivo descriptor ejb-jar.xml**, el cual ya es opcional al día de hoy. Este archivo descriptor **sobrescribe** el comportamiento agregado con las anotaciones en las clases Java.
- Aunque el código mostrado en la figura es muy simple, debemos hacer énfasis y recordar que un **EJB es un componente que se ejecuta en un contenedor Java**.

## 2. EJBs y CDI

- Este ambiente de ejecución es el que permite agregar las características empresariales a nuestras clases Java permitiendo realizar
  - llamadas remotas,
  - inyección de dependencias,
  - manejo de estados y ciclo de vida,
  - pooling de objetos,
  - manejo de mensajería,
  - Manejo de transacciones,
  - seguridad,
  - Soporte de concurrencia,
  - interceptores,
  - manejo de métodos asíncronos

## 2. EJBs y CDI

- Todo esto ocurre simplemente haciendo deploy de esta clase Java al servidor de aplicaciones (sea embebido o no).
- Esto permite que el programador Java se enfoque en los métodos de negocio y delegue todas estas características de **requerimientos no funcionales** a los servidores de aplicaciones Java.

## 2. EJBs y CDI

### Cliente EJB via JNDI

- JNDI (Java Naming and Directory Interface) es un API que nos permite encontrar servicios o recursos en un servidor de aplicaciones Java.
- En un inicio JNDI era la única manera de encontrar los componentes EJB, pero conforme se introdujo el concepto de EJB locales y el manejo de anotaciones existieron otras maneras de ubicar y proporcionar una referencia de los componentes empresariales que se necesitan, a este concepto se le conoce como inyección de dependencias.
- Todo esto simplemente agregando la anotación @EJB!!!

## 2. EJBs y CDI

- Anterior a la versión JEE 6, no existía un nombre estándar para ubicar a los EJB por medio del API JNDI, por lo que cada servidor Java brindaba sintaxis distintas para ubicar a los componentes empresariales.
- Sin embargo, a partir de la versión Java EE 6, se introdujo un nombre global para ubicar a los componentes EJB.

`java:global[/<app-name>]/<module-name>/<bean-name>[!<fully-qualified-interface-name>]`

## 2. EJBs y CDI

### Cliente EJB vía JNDI

- JNDI es un API que permite encontrar servicios o recursos empresariales en un servidor de aplicaciones Java.
- Para encontrar un EJB a partir de la versión 3.1 podemos utilizar la siguiente sintaxis:

java:global[/<app-name>]/<module-name>/<bean-name>[!<fully-qualified-interface-name>]

- Por ejemplo:

```
public void iniciarContenedor() throws Exception {
    System.out.println("----Iniciando EJBContainer...");
    Map<String, Object> map = new HashMap<String, Object>();
    map.put(EJBContainer.APP_NAME, "miApp");
    ec = EJBContainer.createEJBContainer(map);
    ctx = ec.getContext();
    ejb = (HolaMundoEJB) ctx.lookup("java:global/miApp/classes/HolaMundoEJB!beans.HolaMundoEJB");
}
```

Nombre JNDI portable en  
Java EE 6

## 2. EJBs y CDI

- Esto permite ubicar de manera estándar cualquier EJB en cualquier servidor de aplicaciones Java.
- El código básico para encontrar un EJB utilizando JNDI es:

```
HolaMundoEJB ejb = (HolaMundoEJB)  
contexto.lookup("java:global/classes/HolaMundoEJB");
```

- O Incluyendo el nombre del paquete Java:

```
HolaMundoEJB ejb = (HolaMundoEJB)  
contexto.lookup("java:global/classes/HolaMundoEJB!beans.HolaMundoE  
JB");
```

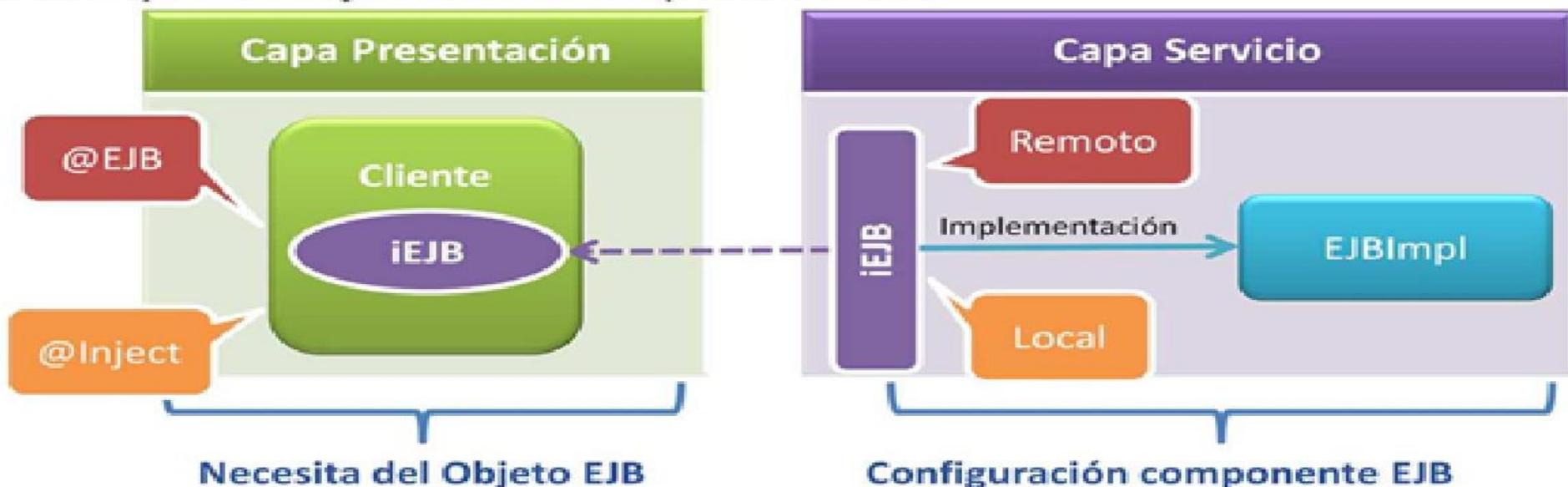
## 2. EJBs y CDI

# Inyección de Dependencias en Java EE

## 2. EJBs y CDI

### Inyección de Dependencias

Concepto de Inyección de Dependencias:



Existen 2 tipos de inyección de dependencias en EJB:

- A) Utilizando la anotación @EJB
- B) Utilizando la anotación @Inject

## 2. EJBs y CDI

- La inyección de dependencias revisa si existe en memoria un EJB ya sea con el mismo tipo o con el mismo nombre, según se especifique, y si existe ese objeto, el servidor de aplicaciones Java regresa una referencia para que pueda ser utilizado.
- En la versión empresarial Java EE 6 existen dos maneras de realizar la inyección de dependencias:

## 2. EJBs y CDI

- Utilizando la anotación **@EJB**: Esta opción está disponible desde la versión Java EE 5, sin embargo es la forma de inyección de dependencias más básica.
- Se recomienda cuando utilizamos **llamadas remotas** a los EJB, injectar un recurso (JDBC DataSource, JPA, Web Service, etc) o si queremos mantener **compatibilidad con Java EE5**.
- Ejemplo de código en el cliente:

**@EJB**

*private PersonaEJBRemote personaEJB;*

## 2. EJBs y CDI

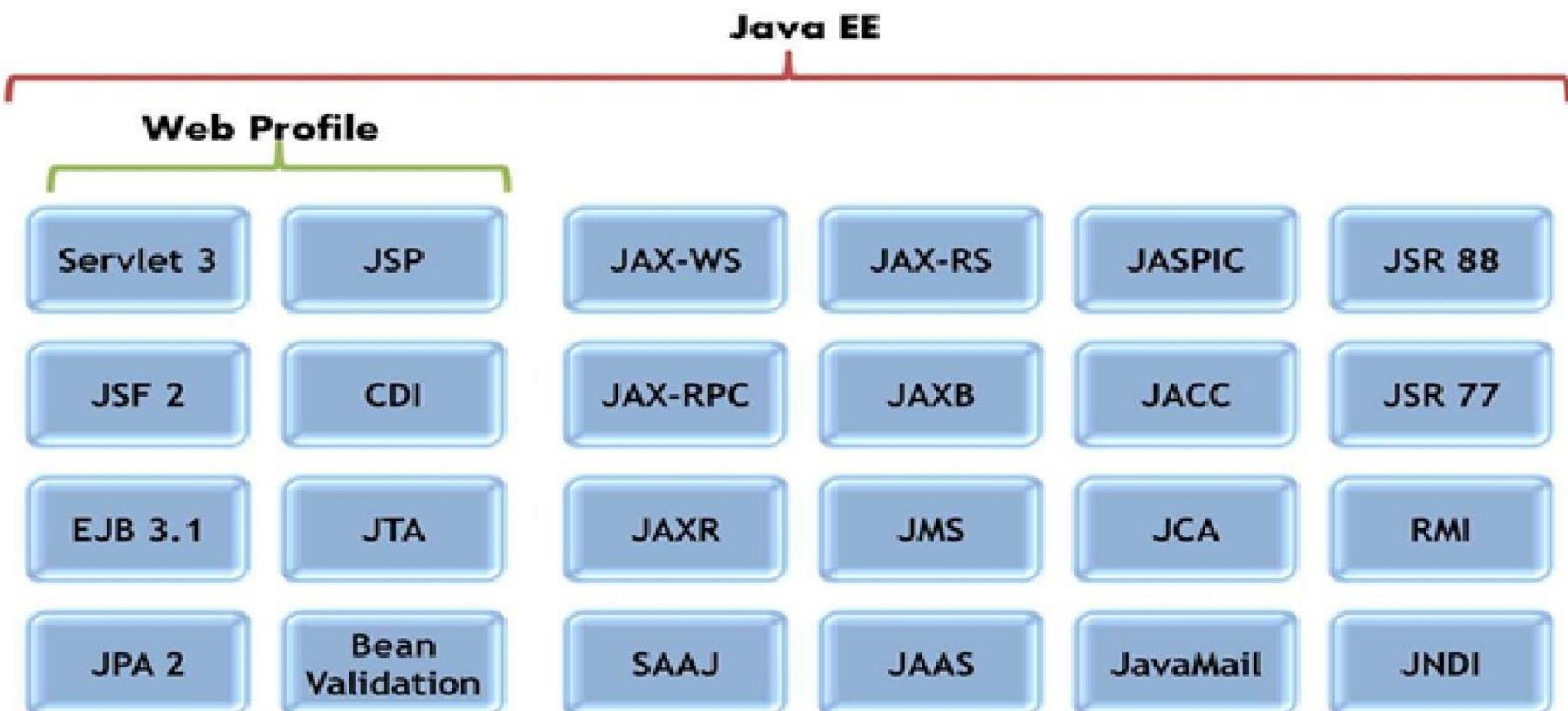
- Utilizando la anotación **@Inject**:
  - Esta forma de inyección de dependencia se apoya del concepto CDI (Context and Dependency Injection), y está disponible a partir de la versión Java EE 6.
  - Esta forma es **más flexible y robusta**, ya que muchos de los conceptos fueron tomados de la experiencia de otros frameworks como Spring, los cuales tienen métodos de inyección de dependencias más poderosos y robustos.
  - Para que el servidor de aplicaciones Java reconozca el concepto de CDI, se debe agregar un archivo llamado **beans.xml**. Se recomienda utilizar la anotación **@Inject** sobre **@EJB** en todos los casos, excepto cuando tenemos EJBs remotos o queremos mantener compatibilidad con Java EE 5. Ejemplo de código en el cliente.

***@Inject***

*private PersonaEJB personaEJB;*

## 2. EJBs y CDI

### Java Web Profile



## 2. EJBs y CDI

- En la figura podemos observar el API Java EE y en particular la relación con el perfil Web, el cual tiene acceso únicamente a ciertas APIs.
- Esto surgió debido a que muchas de las aplicaciones Java EE no necesitaban de todo el poder ni las APIs tan robustas con las que cuenta, por lo tanto únicamente **se agregaron a este perfil Web las APIs más comunes**. Podremos utilizar EJBs 3.1 en nuestras aplicaciones Web sin agregar la complejidad de configuración de los EJBs en versiones anteriores.

## 2. EJBs y CDI

- En la versión Java EE 6 es posible utilizar EJBs locales sin necesidad de empaquetarlos por separado en un archivo .jar, sino únicamente utilizar un archivo .war.
- Sin embargo, lo que debemos resaltar de esta figura es observar que tenemos acceso a los EJB, JPA, JTA, CDI, como las APIs más comunes que utilizaremos en nuestras aplicaciones empresariales.
- Si necesitamos de otras APIs como Java Mail, Web Services, etc, será necesario utilizar un servidor de aplicaciones completo (full).

## 2. EJBs y CDI

- Seleccionar un tipo de perfil dependerá de los requerimientos actuales y futuros de nuestra aplicación empresarial, así que queda a consideración del Arquitecto/Programador la selección del perfil Java EE más adecuado a sus necesidades.
- Podemos utilizar el perfil Web de Java EE y utilizar EJBs.
- La especificación mínima de APIs que podemos utilizar en un perfil Web se conoce como **EJB Lite**.
- Las limitaciones de APIs que tenemos en el perfil Web son las limitantes que tenemos cuando utilizamos EJB, por ello el nombre de **lite**.

## 2. EJBs y CDI

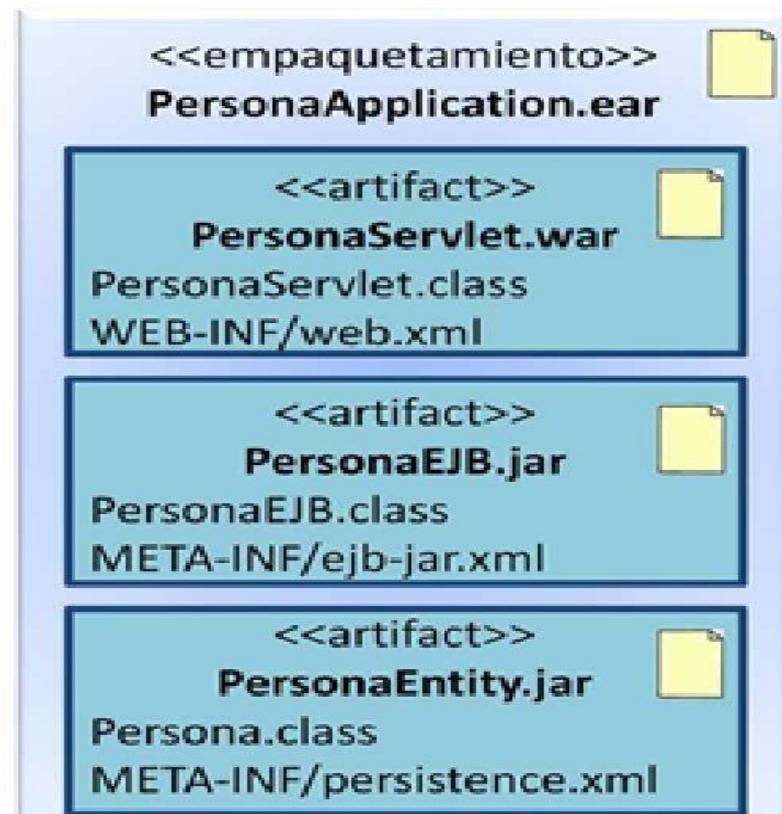
### Comparación de EJB y EJB Lite

API Soportada	EJB Lite	Full EJB 3.1
Stateless beans	✓	✓
Stateful beans	✓	✓
Singleton beans	✓	✓
Message driven beans		✓
No Interfaces	✓	✓
Local Interfaces	✓	✓
Remote Interfaces		✓
Web service Interfaces		✓
Asynchronous Invocation		✓
Interceptors	✓	✓
Declarative security	✓	✓
Declarative transactions	✓	✓
Programmatic transactions	✓	✓
Timer Service		✓
EJB 2.x support		✓
CORBA Interoperability		✓

## 2. EJBs y CDI

### Empaquetamiento de un EJB

- Como cualquier componente empresarial, los EJB también deben empaquetarse para ser desplegados en un servidor Java EE:



## 2. EJBs y CDI

- Debido a que una aplicación Java Empresarial incluye distintos tipos de componentes, tales como: Servlets, páginas JSF, Web Services, EJB, etc, estos componentes deben empaquetarse para ser desplegados en el servidor de aplicaciones Java.
- Los módulos EJB se depositan en META-INF/ejb-jar.xml y en WEB-INF/ejb- jar.xml para los módulos Web. EJB lite puede empaquetarse directamente en un archivo .war (Web Archive File) o .jar (Java Archive File).

## 2. EJBs y CDI

- Si tus requerimientos utilizan la especificación completa de EJBs (llamadas remotas, JMS, llamadas asíncronas, Web Services, etc), entonces se debe empaquetar en un archivo .jar y no en un archivo .war.
- Un archivo .ear (Enterprise Archive File) es utilizado para empaquetar uno o más módulos, ya sean .jar o .war., en un archivo único, el cual es reconocido por el servidor de aplicaciones y éste se encarga de desplegar correctamente cada módulo empaquetado en el archivo .ear.

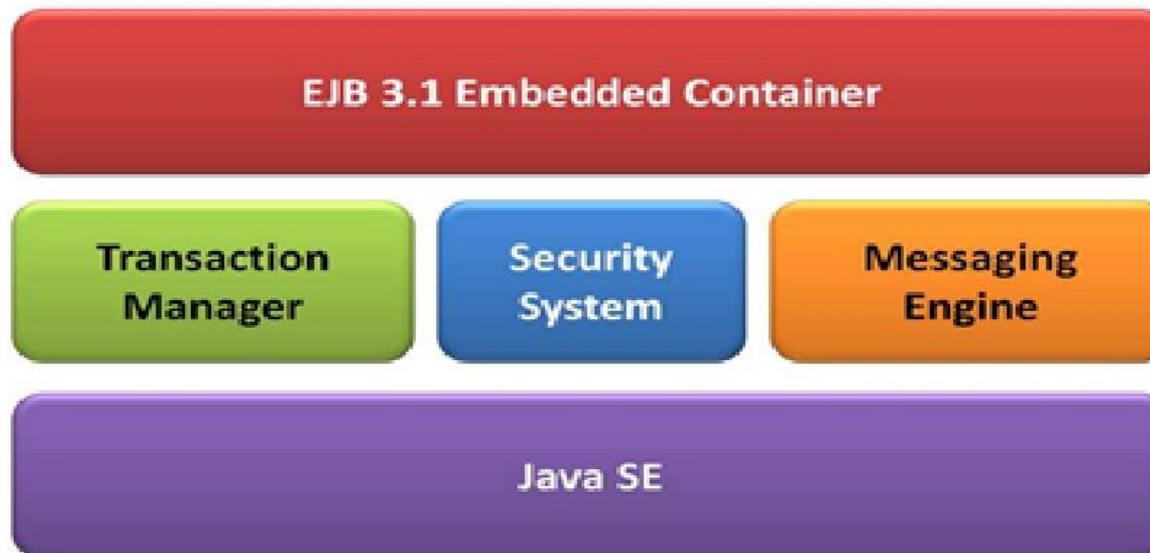
## 2. EJBs y CDI

- Si es necesario desplegar una aplicación Web, podemos empaquetar los EJBs y las clases de Entidad en archivos .jar separados, los Servlets, y páginas JSP o JSF dentro del archivo .war, y estos archivos agregarlos a un archivo .ear, el cual empaqueta todos los componentes en uno solo.
- Desde la especificación EJB 3.1, el concepto de EJB Lite puede empaquetar componentes EJB directamente en un archivo .war, sin necesidad del archivo .jar.

## 2. EJBs y CDI

### Contenedor Embebido Java EE

- Un contenedor embebido tiene como finalidad proveer un ambiente de ejecución Java EE.



```
public void iniciarContenedorEJB() throws Exception {  
    EJBContainer contenedor = EJBContainer.createEJBContainer();  
    Context contexto = contenedor.getInitialContext();  
    HolaMundoEJB ejb = (HolaMundoEJB) contexto.lookup("java:global/classes/HolaMundoEJB");  
    ejb.saluda();  
}
```

## 2. EJBs y CDI

- En sus inicios los EJB para ser probados, debían desplegarse en un contenedor J2EE compatible, y hasta no haber sido desplegados no había forma de saber si un componente funcionaba o no.
- Esto hacía muy lento el desarrollo de aplicaciones ya que el programador pasaba mucho tiempo desplegando su aplicación, únicamente para darse cuenta que debía corregir su código.

## 2. EJBs y CDI

- Esto sin incluir el **tiempo en detener y reiniciar el servidor** de aplicaciones Java. Si una aplicación era de mediana a grande podía demorar varios minutos por cada cambio en un componente sólo para revisar si se había programado correctamente.
- En la versión Java EE 6 y EJB 3.1 contamos con un contenedor embebido, el cual nos permite realizar pruebas unitarias de nuestros componentes empresariales.

## 2. EJBs y CDI

- La idea del **contenedor embebido** es poder ejecutar **componentes EJB** dentro de **aplicaciones Java SE** (aplicaciones estándar), permitiendo utilizar la **misma JVM** (Java Virtual Machine) para ejecutar pruebas (testing), procesos de tipo batch, EJB en aplicaciones de escritorio, entre varias tareas más.
- Un contenedor embebido provee del mismo ambiente de ejecución que un contenedor Java EE y puede manejar los mismos servicios:
  - inyección de dependencias,
  - acceso a componentes empresariales,
  - acceso a CMT (Container- Managed Transactions) para el manejo de transacciones, etc.

## 2. EJBs y CDI

- Podemos observar un ejemplo de cómo ejecutar el contenedor embebido, además utilizar JNDI para encontrar un EJB y ejecutar un método.

```
EJBContainer contenedor = EJBContainer.createEJBContainer();
Context contexto = contenedor.getContext();
HolaMundoEJB ejb = (HolaMundoEJB)
    contexto.lookup("java:global/classes/HolaMundoEJB");
ejb.saluda();
```

## 2. EJBs y CDI

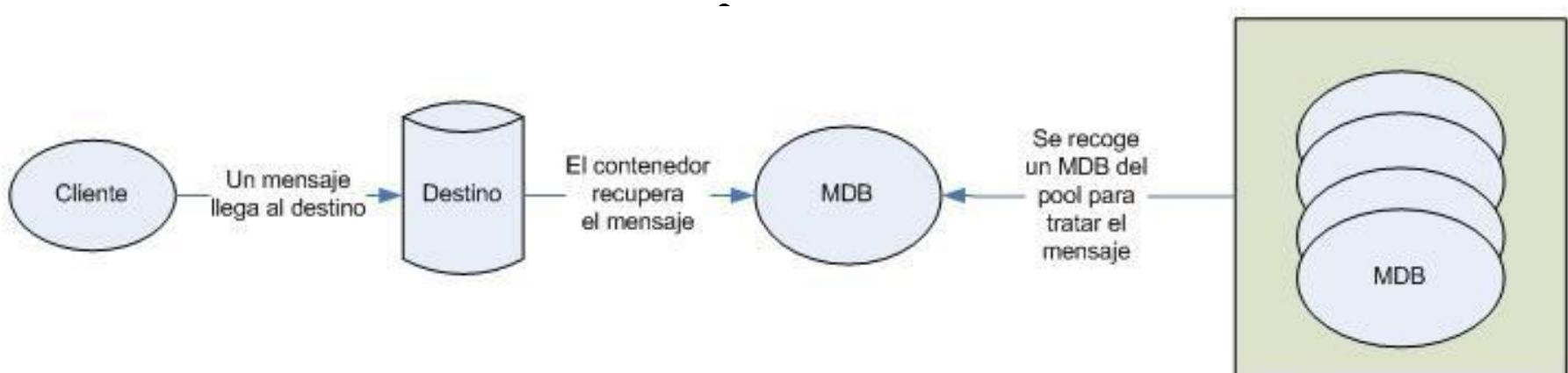
### Message-Driven Beans

## 2. EJBs y CDI

- Los componentes de tipo **Message-Driven Bean** (MDB - Bean Dirigido por Mensajes) son componentes asíncronos de tipo **listener**.
- Un MDB no es más que un **componente que espera a que se le envíe un mensaje**, y realiza cierta acción cuando **finalmente recibe dicho mensaje** (el MDB escucha por si alguien le llama, de ahí su nombre). Algunas propiedades de los componentes MDB son:
  - No mantienen estado
  - Son gestionados por el contenedor (transacciones, seguridad, concurrencia, etc)
  - Son clases puras que no implementan interfaces de negocio

## 2. EJBs y CDI

- Los MDBs **soportan el multihilo** sin necesidad de código adicional. Los MDBs **gestionan los mensajes entrantes mediante múltiples instancias de beans** (dentro de un pool), y tan pronto como un nuevo mensaje llega al destino, una instancia MDB sale del pool para manejar el mensaje.



## 2. EJBs y CDI

- Los componentes MDB forman parte de un sistema de mensajería, el cual se compone de los siguientes subsistemas:

```
Cliente (Productor) --> Broker --> Cliente (Consumidor) --> Message-Driven Bean
```

- El cliente consumidor será el propio contenedor EJB, el cual *distribuirá* los mensajes que consume a los MDB correspondientes.
- Clientes y broker forman parte del servicio de mensajería, que en la especificación EJB es gestionado por defecto mediante JMS.

## 2. EJBs y CDI

### Java Message Service: conceptos básicos

- Java Message Service (JMS - Servicio de Mensajería en Java) es una API neutral que puede ser usada para acceder a sistemas de mensajería.
- Todos los contenedores EJB 3.x deben proporcionar un proveedor de JMS de manera que podamos trabajar con mensajes sin necesidad de añadir librerías externas
- Una aplicación JMS se compone, generalmente, de múltiples clientes JMS y un único proveedor JMS.

## 2. EJBs y CDI

### Java Message Service: conceptos básicos

- Un cliente JMS puede ser de dos tipos:
  - Productor: su misión es enviar mensajes
  - Consumidor: su misión es recibir mensajes
- Por otro lado, la misión del proveedor JMS es dirigir y enviar los mensajes que le llegan a través de un **broker** (**subsistema donde se almacenan los mensajes enviados hasta que son servidos a todos sus consumidores**)
- JMS proporciona un tipo de mensajería asíncrona: los clientes JMS envían mensajes a través del broker sin esperar una respuesta. El broker hacer llegar el mensaje a los clientes JMS que deban consumir el mensaje.

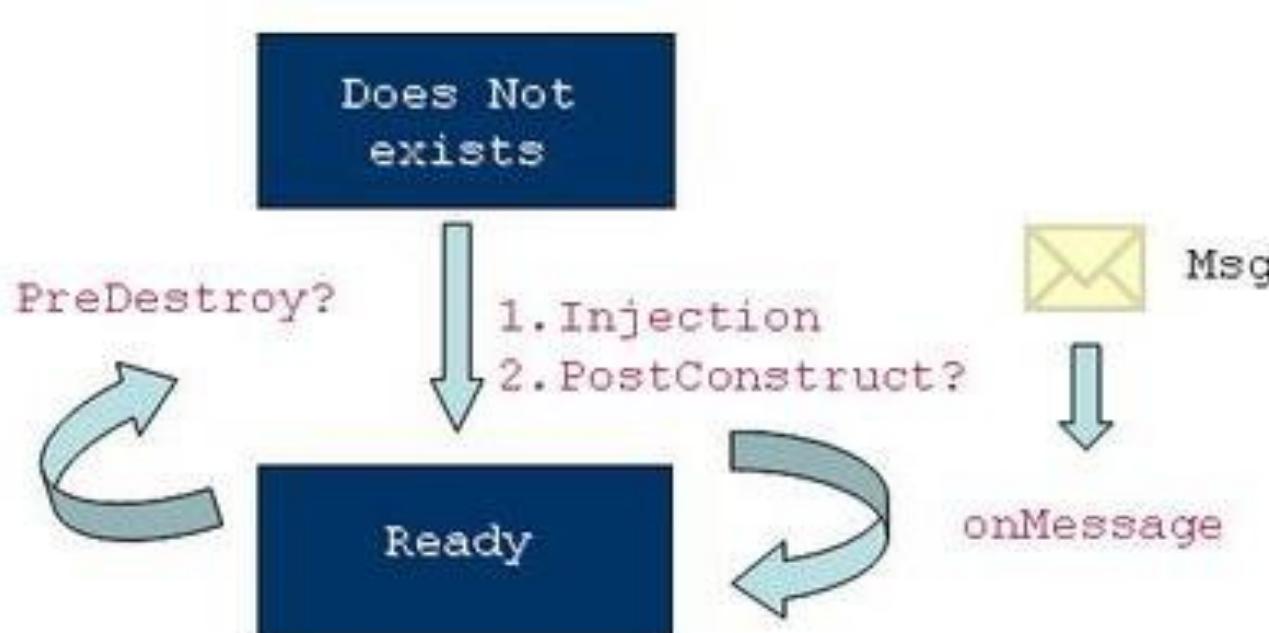
## 2. EJBs y CDI

### Message-Driven Beans: el ciclo de vida

- Los componentes MDB no mantienen estado entre invocaciones, son *stateless*, (pero no confundir con los componentes SLSB).
- Por tanto, su ciclo de vida es similar a los SLSB, constando de dos estados:
  - No existe (Does not exists): Un MDB en el primer estado es aquel que no ha sido creada aún, y por tanto no existe en memoria
  - Preparado en pool (Method-ready pool): representa una instancia que ha sido instanciada e inicializada por el contenedor. Se llega a este estado cuando se inicia el servidor

## 2. EJBs y CDI

### Message-Driven Beans: el ciclo de vida



## 2. EJBs y CDI

- Durante la transición entre el primer estado y el segundo, el contenedor realizará las siguientes tres operaciones en el siguiente orden:
  - Instanciación del MDB (mediante constructor por defecto)
  - Inyección de cualquier recurso necesario y de dependencias
  - Ejecución de un método dentro del MDB marcado con la anotación @PostConstruct, si existe
- Cuando el contenedor no necesita una instancia del MDB (por reducir el número de instancias en el pool, o porque se está produciendo un shutdown del servidor), se realiza una transición en sentido inverso: **del estado preparado en pool al estado no existe.** Durante esta transición se ejecutará, si existe, un método anotado con @PreDestroy

## 2. EJBs y CDI

### Message-Driven Beans: definición

- Todo MDB debe implementar la interface `javax.jms.MessageListener`, la cual define el método `onMessage()`. Es dentro de este método donde se desarrolla toda la acción cuando el MDB procesa un mensaje.
- De manera adicional, debemos indicar al contenedor que nuestra clase es un componente MDB. Para ello, utilizamos la anotación `@MessageDriven`:

## 2. EJBs y CDI

### Message-Driven Beans: definición

```
package es.davidmarco.ejb.mdb;

import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven()
public class PrimerMDB implements MessageListener {
    public void onMessage(Message message) {
        // procesar el mensaje
    }
}
```

## 2. EJBs y CDI

### Message-Driven Beans: definición

- El ejemplo anterior aún no es funcional (producirá un error de despliegue), ya que el MDB necesita saber el tipo de canal virtual (topic/queue) al que debe conectarse, así como el nombre de dicho canal virtual.
- Esta información forma parte de la configuración del MDB, configuración que proporcionamos al componente a través del atributo activationConfig (configuración de activación) de la anteriormente vista anotación @MessageDriven:

## 2. EJBs y CDI

### Message-Driven Beans: definición

```
@MessageDriven(activationConfig={  
    @ActivationConfigProperty(propertyName="destinationType", propertyValue="javax.jms.Topic"),  
    @ActivationConfigProperty(propertyName="destination", propertyValue="topic/MiTopic")})  
public class PrimerMDB implements MessageListener {  
    // ...  
}
```

- Se ha configurado el MDB con las siguientes propiedades:
  - El tipo de canal virtual al que el MDB se conectará (`javax.jms.Topic`)
  - El nombre JNDI del canal virtual al que el MDB se conectará (`topic/MiTopic`).

## 2. EJBs y CDI

### Message-Driven Beans: definición

- Otra opción de configuración bastante interesante (aunque opcional) es la duración de la suscripción:

```
@ActivationConfigProperty(propertyName="subscriptionDurability", propertyValue="Durable")
```

- El componente recibirá todos los mensajes que se envíen a su canal virtual mientras el contenedor EJB (y por tanto el propio componente MDB) esté offline (shutdown del servidor, un problema de conectividad por red, etc.).
- En otras palabras, el mensaje estará disponible hasta que todos sus suscriptores de tipo *durable* lo hayan recibido y procesado.

# Índice

- 
1. Introducción Java Empresarial
  2. Enterprise Java Beans (EJBs) y Context Directory Inyection (CDI)
  - 3. Java Persistence Api (JPA)**
  4. Servlets y jsps
  5. JavaServer Faces
  6. Web Services (SOAP y REST)
  7. Seguridad en JEE

### 3. Java Persistence Api (JPA)

- La mayoría de la información de las aplicaciones empresariales es almacenada en **bases de datos relacionales**.
- La persistencia de datos en Java, y en general en los sistemas de información, ha sido uno de los grandes temas a resolver en el mundo de la programación.

### 3. Java Persistence Api (JPA)

- Al utilizar únicamente **JDBC** tenemos el problema de crear **demasiado código** para poder ejecutar una simple consulta.
- Por lo tanto, para simplificar el proceso de interacción con una base de datos (select, insert, update, delete), se ha utilizado desde hace ya varios años el concepto de frameworks **ORM (Object Relational Mapping)**, tales como **Hibernate**.

# 3. Java Persistence Api (JPA)

## ¿Qué es Java Persistence API?

- Java Persistence API, mejor conocido como JPA, es el estándar de persistencia de Java.
- JPA implementa conceptos de frameworks ORM (Object Relational Mapping)



### 3. Java Persistence Api (JPA)

- Esta tecnología de Java es el **estándar de persistencia en Java**, y la buena noticia es que cuenta con **varias implementaciones**, tales como Hibernate, EclipseLink, OpenJPA, TopLink, entre otras más.
- Si ya conoces Hibernate, o algún framework de persistencia Java, te será muy familiar muchos de los conceptos que estudiaremos en esta lección.

# 3. Java Persistence Api (JPA)

## Características de JPA

### Características de Java Persistence API:

- ✓ Persistencia utilizando POJOs.
- ✓ No Intrusivo.
- ✓ Consultas utilizando Objetos Java.
- ✓ Configuración Simplificada.
- ✓ Integración.
- ✓ Testing.



### 3. Java Persistence Api (JPA)

- La idea del API JPA es **trabajar con objetos Java y no con código SQL**, de tal manera que podamos enfocarnos en el código Java.
- JPA permite abstraer la comunicación con las bases de datos y crea un estándar para ejecutar consultas y manipular la información de una base de datos.

### 3. Java Persistence Api (JPA)

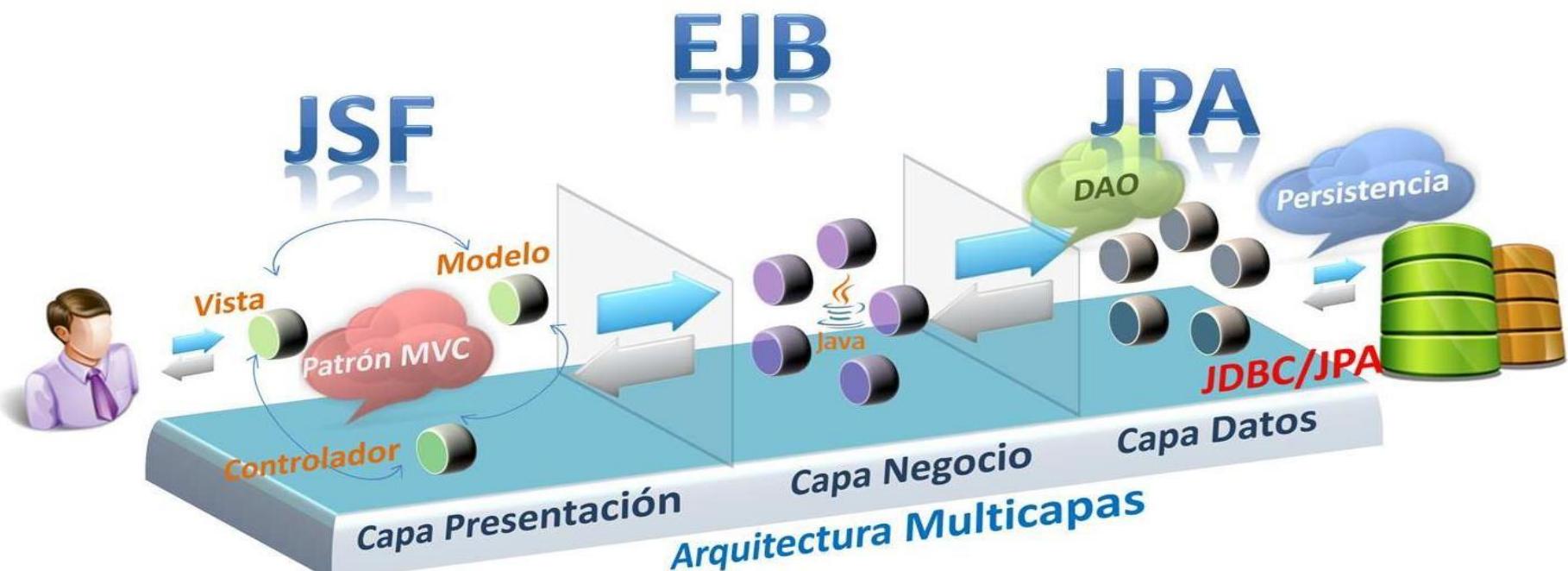
- Características de Java Persistence API:
  - **Persistencia utilizando POJOs:** Este es posiblemente el aspecto más importante de JPA, debido a que cualquier clase de Java podemos convertirla en una clase de entidad, simplemente agregando anotaciones y/o agregando un archivo xml de mapeo.
  - **No Intrusivo:** JPA es una capa separada de los objetos a persistir. Por ello, las clases Java de Entidad no requieren extender ninguna funcionalidad en particular ni saber de la existencia de JPA, por ello es no intrusivo.
  - **Consultas utilizando Objetos Java:** JPA permite ejecutar queries expresadas en términos de objetos Java y sus relaciones, sin necesidad de utilizar el lenguaje SQL. Las queries son traducidos por el API de JPA en el código SQL equivalente.

### 3. Java Persistence Api (JPA)

- Características de Java Persistence API:
  - **Configuración simple:** Muchas de las **opciones** de JPA están configuradas con opciones **por defecto**, sin embargo si queremos personalizarlas, es muy simple, ya sea con **anotaciones** o **a través de archivos xml de configuración**.
  - **Integración:** Debido a que las arquitecturas empresariales Java son por naturaleza mult capas, una integración transparente es muy valiosa para los programadores Java y JPA permite hacer la integración con las demás capas de manera muy simple.
  - **Testing:** Con JPA ahora es posible realizar pruebas unitarias, o utilizar cualquier clase con un método main fuera del servidor, simplemente utilizando la versión estándar de Java. Esto permite reducir los tiempos de desarrollo de las aplicaciones empresariales de manera considerable.

# 3. Java Persistence Api (JPA)

## Arquitectura Empresarial con JPA



Servidor de  
Aplicaciones Java

### 3. Java Persistence Api (JPA)

- Esta tecnología aplica directamente en la capa de datos, la cual se encarga de tareas tales como:
  - Recuperación de información a través de consultas (select)
  - Manejo de información de objetos Java en las tablas de base de datos respectivas (insert, update, delete)
  - Manejo de una unidad de persistencia (Persistance Unit) para la creación y destrucción de conexiones a la base de datos.
  - Manejo de transacciones, respetando el esquema de propagación definido en la capa de negocio en los EJBs de Sesión.
  - Portabilidad hacia otras bases de datos con un impacto menor, así como bajo acoplamiento con las otras capas empresariales.

### 3. Java Persistence Api (JPA)

- Además, para realizar las tareas de persistencia, podemos utilizar patrones de diseño tales como:
  - **DAO** (Data Access Object): Este **patrón de diseño** suele definir una interfaz y una implementación de dicha interfaz, **para realizar las operaciones más comunes con la Entidad respectiva**. Por ejemplo, para la entidad Persona, generaremos la interfaz DaoPersona, y agregaremos los métodos agregarPersona, modificarPersona, eliminarPersona, findAllPersonas, etc.
  - **DTO** (Data Transfer Object): Este patrón de diseño permite definir una **clase, que en ocasiones es muy similar a la clase de entidad**, ya que contiene los mismos atributos, pero con el **objetivo de transmitirla a las siguientes capas**, incluso, hasta la capa Web. Por ello se les conoce como **objetos de valor o de transferencia**.

# 3. Java Persistence Api (JPA)

## Entidades y Persistencia en JPA

- Una clase de entidad es un POJO y puede configurarse por medio de anotaciones o un archivo XML.
- Ejemplo de clase de Entidad con anotaciones:

```
@Entity  
public class Persona {  
  
    @Id  
    @GeneratedValue  
    private Long personaId;  
  
    @Column(nullable = false)  
    private String nombre;  
  
    private String apePaterno;  
  
    private String apeMaterno;  
    private String email;  
    private Integer telefono;  
  
    // Constructores, getters, setters  
}
```

### 3. Java Persistence Api (JPA)

- En las **primeras versiones** de J2EE, existía el concepto de **Entity Beans** para el manejo de persistencia.
- Sin embargo esta tecnología resultaba complicada para sistemas del mundo real, resultando en un bajo rendimiento. Realizar consultas (queries) utilizando los objetos Entity era muy complicado,
  - se necesitaba un servidor de aplicaciones Java para ejecutar un EJB tipo Entity
  - no existían pruebas unitarias, y por lo tanto cualquier cambio en nuestro código implicaba un nuevo deploy

### 3. Java Persistence Api (JPA)

- Una clase conocida como **Entidad** es simplemente un **POJO**, y en combinación con el uso de **anotaciones**, es suficiente para convertirla en una clase de Entidad, la cual representa un registro de una tabla de base de datos.
- Este tipo de conceptos, heredados de frameworks como Hibernate, TopLink, JDO, entre otros, contribuyó en lo que conocemos al día de hoy como el estándar de persistencia Java conocido como JPA.

### 3. Java Persistence Api (JPA)

- El API de JPA se puede utilizar en una aplicación estándar de Java o en un servidor Web o Empresarial Java.
- Ahora ya **es posible realizar pruebas unitarias** sobre nuestras clases de Entidad y Consultas sobre los objetos de Entidad, **disminuyendo así el tiempo** de desarrollo de nuestras clases de entidad, consultas, y en general en la creación de la capa de datos de una aplicación empresarial.

### 3. Java Persistence Api (JPA)

- Para que una clase de Entidad pueda ser persistida, se debe realizar una llamada al API de JPA. De hecho muchas de las operaciones se realizan a través de esta API, la cual está separada de nuestras clases de Entidad.

### 3. Java Persistence Api (JPA)

- El API JPA, la cual tiene como elemento principal al objeto **EntityManager**, siendo este una interfaz.
- Una implementación de esta interfaz es la que realmente ejecuta el trabajo de:
  - persistencia,
  - sincronización con la base de datos,
  - transaccionalidad,
  - validación de mapeo,
  - conversión de código Java a SQL, entre muchas otras tareas.

### 3. Java Persistence Api (JPA)

- Por lo tanto, una clase **Entity**, desde el punto de vista descrito anteriormente, es tan solo una clase Java normal, la cual al vincularse con un **EntityManager**, se persiste en la base de datos.
- El objeto EntityManager se obtiene de una fábrica de objetos conocida como **EntityManagerFactory**, y este objeto se asocia con un proveedor JPA, pudiendo haber seleccionado entre varios proveedores según la implementación de JPA escogida (Hibernate, EclipseLink, OpenJPA, etc).

### 3. Java Persistence Api (JPA)

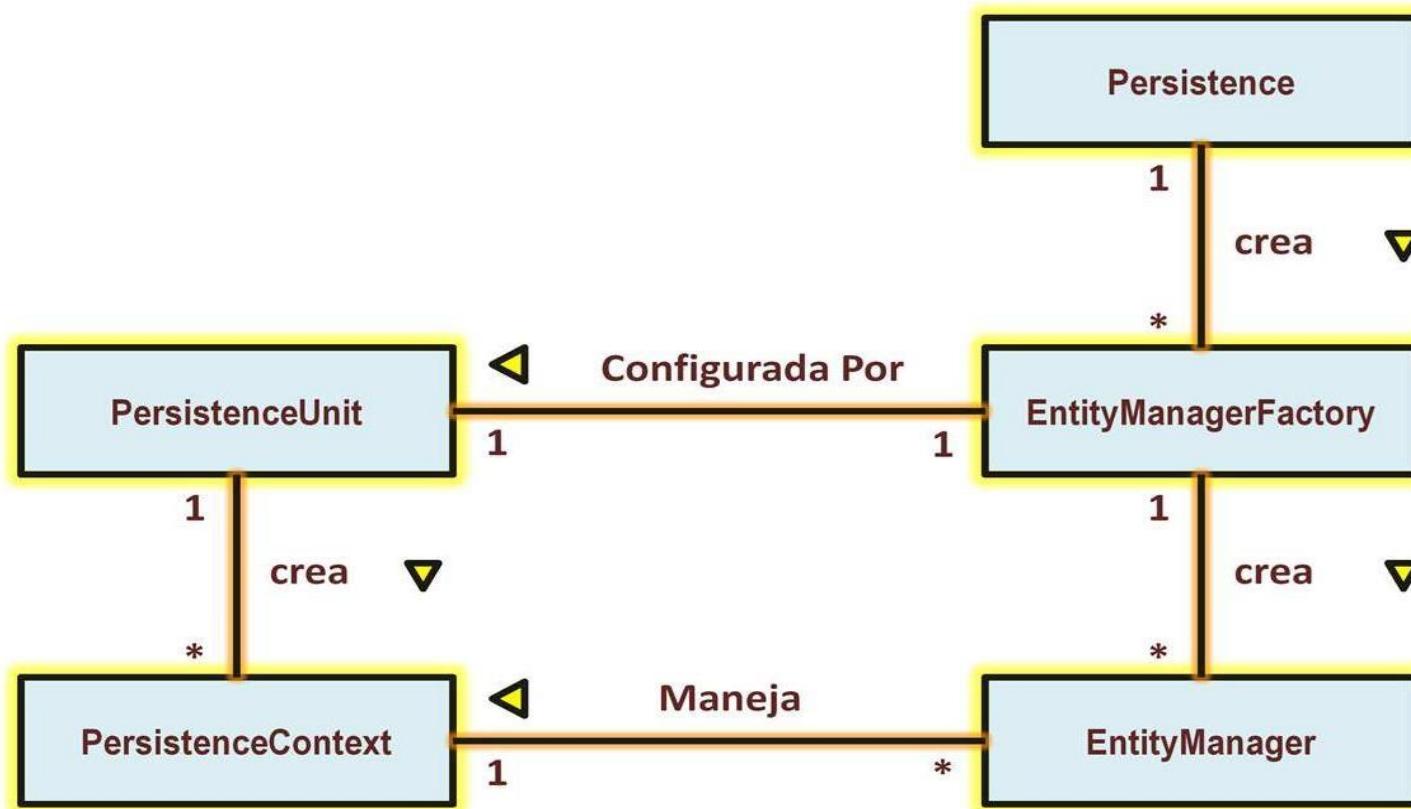
- El objeto Persistance Unit, se encarga de realizar la **configuración del proveedor seleccionado por medio de un archivo xml**, además de definir otros elementos tales como: la forma de comunicarse con la Base de Datos, las clase de Entidad en la aplicación, si se va a utilizar JTA para el manejo transaccional, entre varias características más.

### 3. Java Persistence Api (JPA)

- A su vez, al conjunto de objetos Entity administrados por JPA en un tiempo específico de la aplicación se le conoce como PersisteceContext, de esta manera JPA se asegura que no existan objetos de Entidad duplicados en memoria, entre otras tareas más.

# 3. Java Persistence Api (JPA)

## API de JPA y Entity Manager



### 3. Java Persistence Api (JPA)

- Un ejemplo de cómo utilizar el API JPA para persistir un objeto de Entidad, se muestra a continuación:

```
EntityManagerFactory emf =  
Persistence.createEntityManagerFactory("PersonaService");  
EntityManager em = emf.createEntityManager();  
Persona persona = new Persona(15);  
em.persist( persona );
```

# 3. Java Persistence Api (JPA)

## Configuración de Unidad de Persistencia

Configuración de la Unidad de Persistencia (Persistence Unit):

Ejemplo de contenido del archivo persistence.xml:

```
<persistence>
  <persistence-unit name="PersonaService" transaction-type="RESOURCE_LOCAL">
    <class>domain.Persona</class>
    <properties>
      <property name="javax.persistence.jdbc.driver"
                value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="javax.persistence.jdbc.url"
                value="jdbc:derby://localhost:1527/PersonaServDB;create=true"/>
      <property name="javax.persistence.jdbc.user" value="APP"/>
      <property name="javax.persistence.jdbc.password" value="APP"/>
    </properties>
  </persistence-unit>
</persistence>
```

### 3. Java Persistence Api (JPA)

- Para realizar la configuración de la Unidad de Persistencia se debe utilizar un archivo xml llamado **persistence.xml**.
- El nombre del elemento **persistence-unit** indica el **nombre de la unidad de persistencia**, y es muy importante recordarlo, ya que es el nombre que utilizaremos en nuestro código Java al momento de utilizar el objeto EntityManagerFactory.

### 3. Java Persistence Api (JPA)

- El atributo **transaction-type** especifica el **tipo de transaccionalidad** que se utilizará, pudiendo seleccionar JTA como el proveedor.
- Se pueden especificar también las **clases de Entidad del sistema**, pudiendo definir varias clases. Si la aplicación se despliega en un servidor Java, no es necesario declarar estas clases, sin embargo, para aplicaciones Java SE (Standard Edition) es necesario especificar las clases de Entidad del sistema.

### 3. Java Persistence Api (JPA)

- La sección de propiedades especifica **características del proveedor** a utilizar, así como los datos de conexión a la base de datos.
- Al momento de empaquetar una aplicación Java, el archivo persistence.xml se debe ubicar en la carpeta META-INF/persistence.xml del archivo .jar.

# 3. Java Persistence Api (JPA)

## Referenciando la Unidad de Persistencia

Ejemplo de uso de la unidad de persistencia y del Entity Manager:

```
@Stateless  
public class PersonaServiceBean implements PersonaService {  
  
    @PersistenceContext(unitName="PersonaService")  
    EntityManager em;  
  
    public void agregarPersona(Persona persona) {  
        em.persist(persona);  
    }  
  
    public Persona encontrarPersona(int idPersona) {  
        return em.find(Persona.class, idPersona);  
    }  
  
    public Persona modificarNombrePersona(int idPersona, String nuevoNombre) {  
        Persona persona = em.find(Persona.class, idPersona);  
        if (persona != null) {  
            persona.setNombre(nuevoNombre);  
        }  
        return persona;  
    }  
  
    public void eliminarPersona(int idPersona) {  
        Employee emp = em.find(Employee.class, idPersona);  
        em.remove(emp);  
    }  
}
```

### 3. Java Persistence Api (JPA)

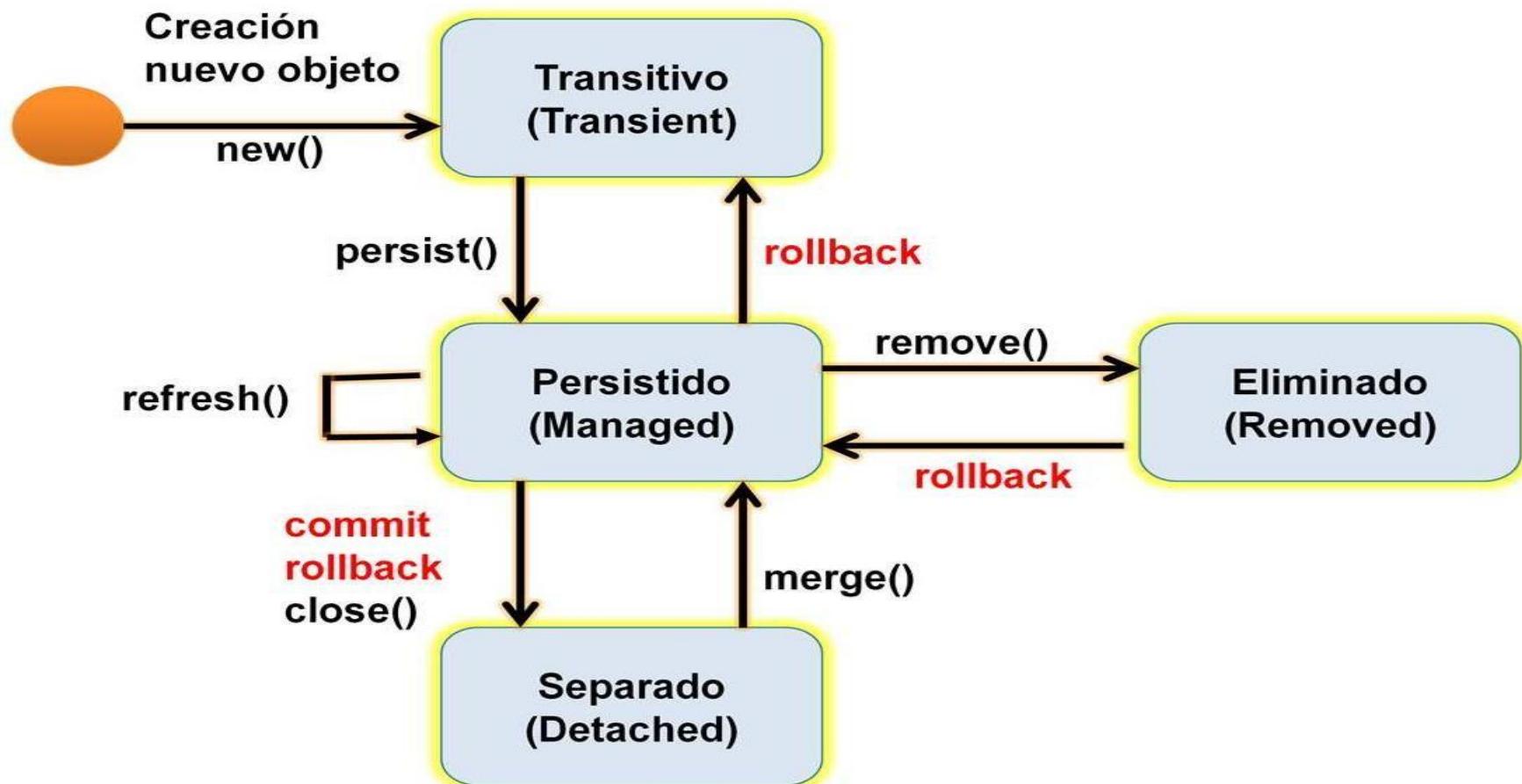
- Por ejemplo, podemos realizar tareas como:
  - **Inserción:** Para persistir una entidad se utiliza el método **persist** del EntityManager. Con este método podemos generar un registro en la base de datos. Este registro no será guardado hasta haber concluido la transacción (commit). **Los métodos de un Session Bean son transaccionales por default**, esto implica que al terminar de ejecutar el método agregarPersona y al ejecutarse en un contenedor empresarial Java, en automático se realizará el commit.
  - **Búsqueda:** Una vez que tenemos un objeto persistido, podemos recuperar la información del registro de la base de datos utilizando el método **find** del objeto EntityManager, y basta con especificar el tipo (clase) y el id (llave primaria) que estamos buscando.

### 3. Java Persistence Api (JPA)

- **Modificación:** La modificación cambia un poco, debido a que JPA necesita primero saber con qué entidad se está trabajando, por ello necesitamos recuperar el objeto de entidad. Una vez recuperado, realizamos las modificaciones necesarias, y si el objeto se encuentra en una transacción activa, JPA revisará en automático si es necesario realizar alguna actualización sobre el registro. Lo interesante es que no es necesario volver a llamar al método **persist**, esta llamada es opcional.
- **Eliminación:** Similar a la modificación, primero se debe recuperar la entidad con el método **find**, y una vez en memoria, llamamos el método **remove**.

# 3. Java Persistence Api (JPA)

## Ciclo de Vida Entidad JPA



### 3. Java Persistence Api (JPA)

- El API de JPA simplifica en gran medida la forma en que interactuamos con una base de datos. JPA agrega un ciclo de vida para la administración de los objetos de entidad.
- Los estados del ciclo de vida son.
  - **Estado Transitivo (Transient):**
    - Los objetos de entidad nuevos
    - NO son guardados directamente en la Base de Datos (BD).
    - No están asociados con un registro de BD.
    - Se consideran NO transaccionales.
  - **Estado Persistente (Managed):**
    - Un objeto persistente tiene asociado un registro en la BD.
    - Los objetos persistentes siempre están asociados con una transacción. Su estado se sincroniza con la BD al terminar la transacción.

# 3. Java Persistence Api (JPA)

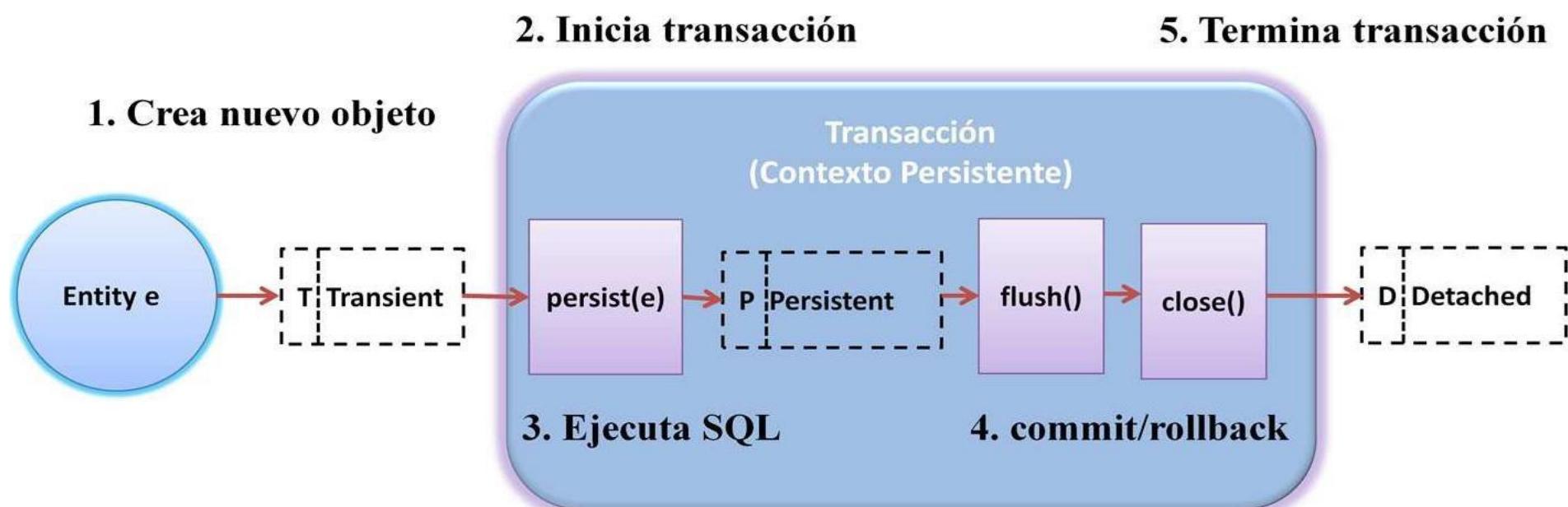
- **Estado Separado (Detached):**
  - Estos objetos tienen asociado un registro de BD, pero su estado no está sincronizado con la BD
  - Todos los objetos recuperados en una transacción se convierten en detached una vez que termina la misma
- **Estado Eliminado (Removed):**
  - Estos objetos ya no tiene una representación con la BD y al terminar la transacción, el registro asociado es eliminado.

### 3. Java Persistence Api (JPA)

- Existen además anotaciones para complementar cualquier acción deseada entre los estados mostrados, tales como `@PrePersist` y `@PostPersist`, las cuales se utilizan al persistir un objeto, `@PreRemove` y `@PostRemove` al eliminar un objeto, `@PreUpdate` y `@PostUpdate` al actualizar un objeto, así como `@PostLoad` al hacer refresh del objeto.

# 3. Java Persistence Api (JPA)

## Persistir un objeto en JPA



# 3. Java Persistence Api (JPA)

- Código ejemplo para persistir un objeto en JPA:

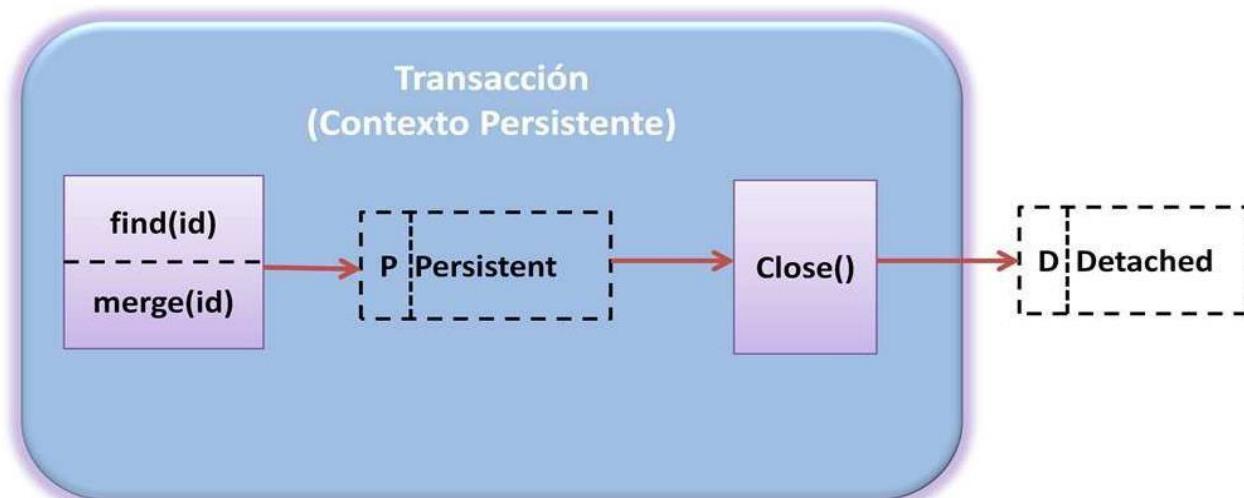
```
public void testPersistirObjeto() {  
    //Paso 1. Crea nuevo objeto  
    //Objeto en estado transitivo Persona persona1 = new  
    Persona("Pedro","Luna",null,"pluna@mail.com","19292943");  
  
    //Paso 2. Inicia transacción  
    EntityTransaction tx = em.getTransaction();  
    tx.begin();  
  
    //Paso 3. Ejecuta SQL  
    em.persist(persona1);  
  
    //Paso 4. Commit/rollback  
    tx.commit();  
  
    //Objeto en estado detached  
    log.debug("Objeto persistido:" + persona1);  
}
```

# 3. Java Persistence Api (JPA)

## Recuperar un objeto de Entidad en JPA

1. Inicia transacción

3. Termina transacción



2. Ejecuta SQL

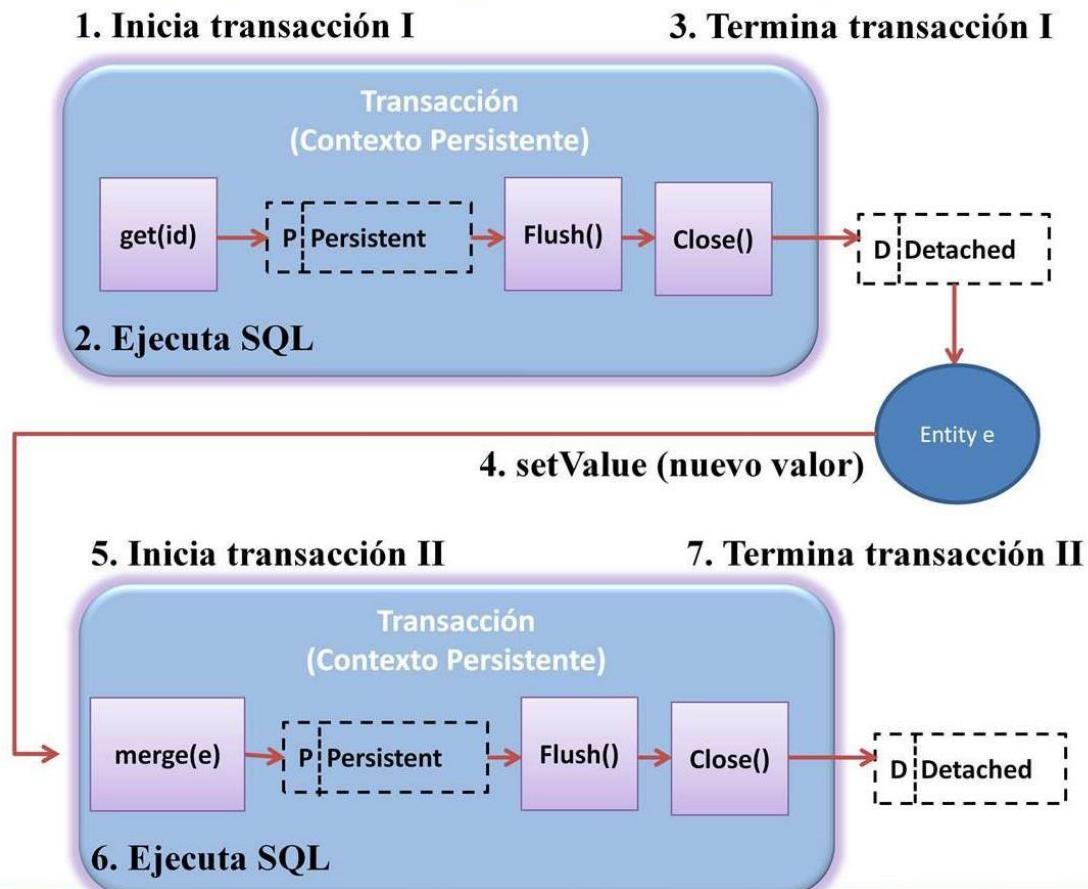
### 3. Java Persistence Api (JPA)

- Código de ejemplo para recuperar un objeto en JPA:

```
public void testEncontrarObjeto() {  
  
    //Paso 1. Inicia transacción  
    EntityTransaction tx = em.getTransaction();  
    tx.begin();  
  
    //Paso 2. Ejecuta SQL de tipo select  
    Persona persona1 = em.find(Persona.class, 23);  
  
    //Paso 3. Termina transacción  
    tx.commit();  
  
    //Objeto en estado detached  
    log.debug("Objeto recuperado:" + persona1);  
}
```

# 3. Java Persistence Api (JPA)

## Actualizar un Objeto Persistente en JPA



### 3. Java Persistence Api (JPA)

- Proceso para actualizar un objeto en JPA:

```
public void testActualizarObjeto() {  
    //Paso 1. Inicia transacción 1  
    EntityTransaction tx1 = em.getTransaction();  
    tx1.begin();  
  
    //Paso 2. Ejecuta SQL de tipo select  
    //El id proporcionado debe existir en la base de datos  
    Persona personal1 = em.find(Persona.class, 23);  
  
    //Paso 3. Termina transacción 1  
    tx1.commit();  
  
    //Objeto en estado detached |  
    log.debug("Objeto recuperado:" + personal1);  
  
    //Paso 4. setValue (nuevoValor)  
    personal1.setApeMaterno("Nava");  
  
    //Paso 5. Inicia transacción 2  
    EntityTransaction tx2 = em.getTransaction();  
    tx2.begin();
```

### 3. Java Persistence Api (JPA)

```
//Paso 6. Ejecuta SQL. Es un select, pero al estar modificado,  
//al terminar la transacción hará un update  
//Como ya tenemos el objeto hacemos solo un merge para resincronizar  
//el objeto a hacer merge, debe contar con el valor de la llave primaria  
em.merge(persona1);  
  
//Paso 7. Termina transacción 2  
//Al momento de hacer commit, se revisan las diferencias  
//entre el objeto de la base de datos  
//y el objeto en memoria, y se aplican los cambios si los hubiese  
tx2.commit();  
  
//Objeto en estado detached ya modificado  
log.debug("Objeto recuperado:" + persona1);  
}
```

# 3. Java Persistence Api (JPA)

## Actualizar un Objeto Persistente con Sesión Larga

1. Inicia transacción

4. Termina transacción



**No hay necesidad de hacer un UPDATE explícito, al terminar la transacción (commit) se ejecuta el update.**

# 3. Java Persistence Api (JPA)

- Persistir un objeto en JPA:

```
public void testActualizarObjetoSesionLarga() {  
  
    //Paso 1. Inicia transacción 1  
    EntityTransaction tx1 = em.getTransaction();  
    tx1.begin();  
  
    //Paso 2. Ejecuta SQL de tipo select  
    //Puede ser un find o un merge si ya tenemos el objeto  
    Persona persona1 = em.find(Persona.class, 23);  
  
    //Paso 3. setValue (nuevoValor)  
    persona1.setApeMaterno("Aguirre");  
  
    persona1.setApeMaterno("Torres");  
  
    //Paso 4. Termina transacción 1  
    //Ejecuta el update, aunque hayamos hecho 2 cambios sobre el  
    //objeto  
    //unicamente persiste el último cambio del objeto  
    //es decir, el valor de apeMaterno=Torres  
    tx1.commit();  
  
    //Objeto en estado detached  
    log.debug("Objeto recuperado:" + persona1);  
}
```

# 3. Java Persistence Api (JPA)

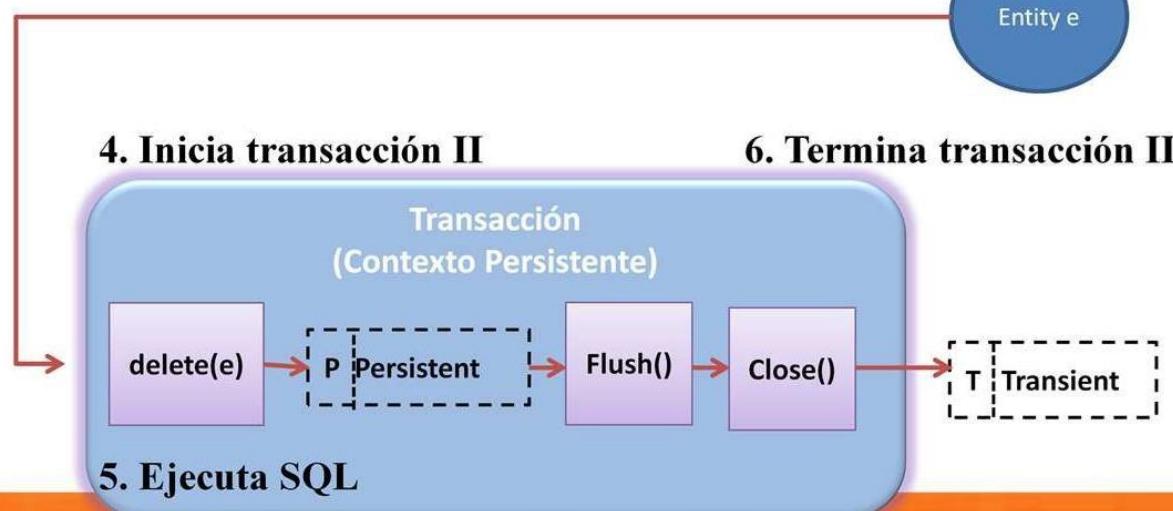
## Eliminar un objeto en JPA

1. Inicia transacción I



2. Ejecuta SQL

3. Termina transacción I



4. Inicia transacción II

6. Termina transacción II

### 3. Java Persistence Api (JPA)

```
public void testActualizarObjetoSesionLarga() {  
    // Paso 1. Inicia transacción 1  
    EntityTransaction tx1 = em.getTransaction();  
    tx1.begin();  
  
    // Paso 2. Ejecuta SQL de tipo select  
    Persona personal1 = em.find(Persona.class, 23);  
  
    // Paso 3. Termina transacción 1  
    tx1.commit();  
  
    // Objeto en estado detached  
    log.debug("Objeto recuperado:" + personal1);  
  
    // Paso 4. Inicia transacción 2  
    EntityTransaction tx2 = em.getTransaction();  
    tx2.begin();  
  
    // Paso 5. Ejecuta SQL (es un delete)  
    em.remove(personal1);  
  
    // Paso 6. Termina transacción 2  
    // Al momento de hacer commit,  
    // se realiza el delete  
    tx2.commit();  
  
    // Objeto en estado detached ya modificado  
    // Ya no es posible resincronizarlo en otra transacción  
    // Solo está en memoria, pero al terminar el método se eliminará  
    log.debug("Objeto eliminado:" + personal1);  
}
```

### 3. Java Persistence Api (JPA)

```
public void testActualizarObjetoSesionLarga() {  
    // Paso 1. Inicia transacción 1  
    EntityTransaction tx1 = em.getTransaction();  
    tx1.begin();  
  
    // Paso 2. Ejecuta SQL de tipo select  
    Persona personal1 = em.find(Persona.class, 23);  
  
    // Paso 3. Termina transacción 1  
    tx1.commit();  
  
    // Objeto en estado detached  
    log.debug("Objeto recuperado:" + personal1);  
  
    // Paso 4. Inicia transacción 2  
    EntityTransaction tx2 = em.getTransaction();  
    tx2.begin();  
  
    // Paso 5. Ejecuta SQL (es un delete)  
    em.remove(personal1);  
  
    // Paso 6. Termina transacción 2  
    // Al momento de hacer commit,  
    // se realiza el delete  
    tx2.commit();  
  
    // Objeto en estado detached ya modificado  
    // Ya no es posible resincronizarlo en otra transacción  
    // Solo está en memoria, pero al terminar el método se eliminará  
    log.debug("Objeto eliminado:" + personal1);  
}
```

# 3. Java Persistence Api (JPA)

## Relaciones en JPA

### Tipos de Relaciones:

- ✓ **Uno a Uno:** @OneToOne
- ✓ **Uno a Muchos:** @OneToMany
- ✓ **Muchos a Uno:** @ManyToOne
- ✓ **Muchos a Muchos:** @ManyToMany

### Direccionalidad en las relaciones:

- ✓ **Unidireccional:** Se define el atributo de relación solo en una clase.
- ✓ **Bidireccional:** Se define los atributos de relación en ambas clases.

### 3. Java Persistence Api (JPA)

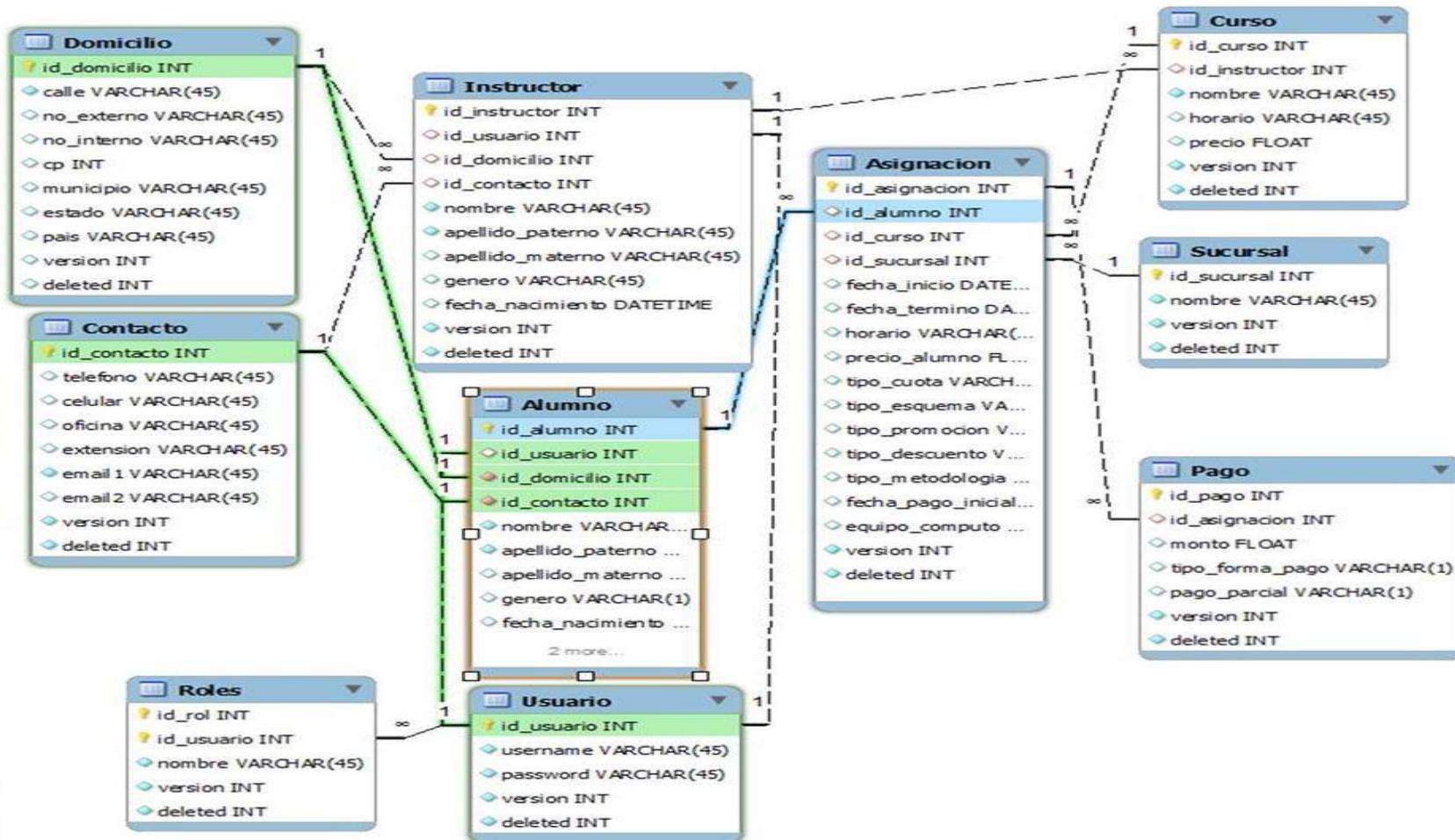
- Normalmente los objetos de entidad, en un sistema con bases de datos relacionales, mantienen asociaciones con uno o más objetos. Los tipos de relaciones en JPA son las mismas que se manejan en la teoría de bases de datos relacionales.
  - 1 a 1
  - 1 a Muchos o Muchos a 1
  - Muchos a Muchos

### 3. Java Persistence Api (JPA)

- Las relaciones también tienen **navegabilidad** (directionalidad), esto quiere decir que podemos acceder a los objetos con los que tenemos relación de manera **unidireccional** o **bidireccional**.
- Esto lo logramos debido a que en los objetos de entidad manejamos un atributo que identifica el objeto(s) de entidad(es) con el que tenemos relación.
- Cuando cada objeto de entidad se apunta uno al otro por medio de este atributo o colección, se dice que es una relación **bidireccional**, y si por solamente una entidad apunta a la otra, la relación se conoce como **unidireccional**.

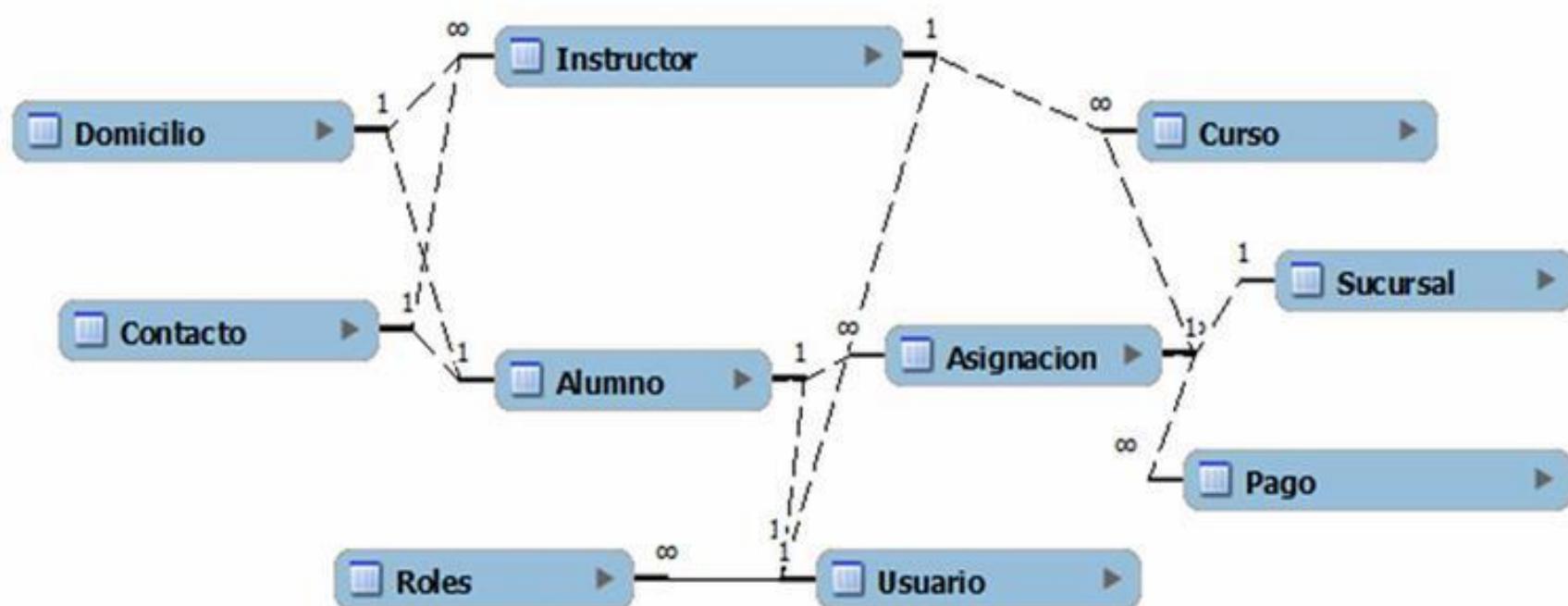
# 3. Java Persistence Api (JPA)

## Ejemplo Diagrama Entidad Relación



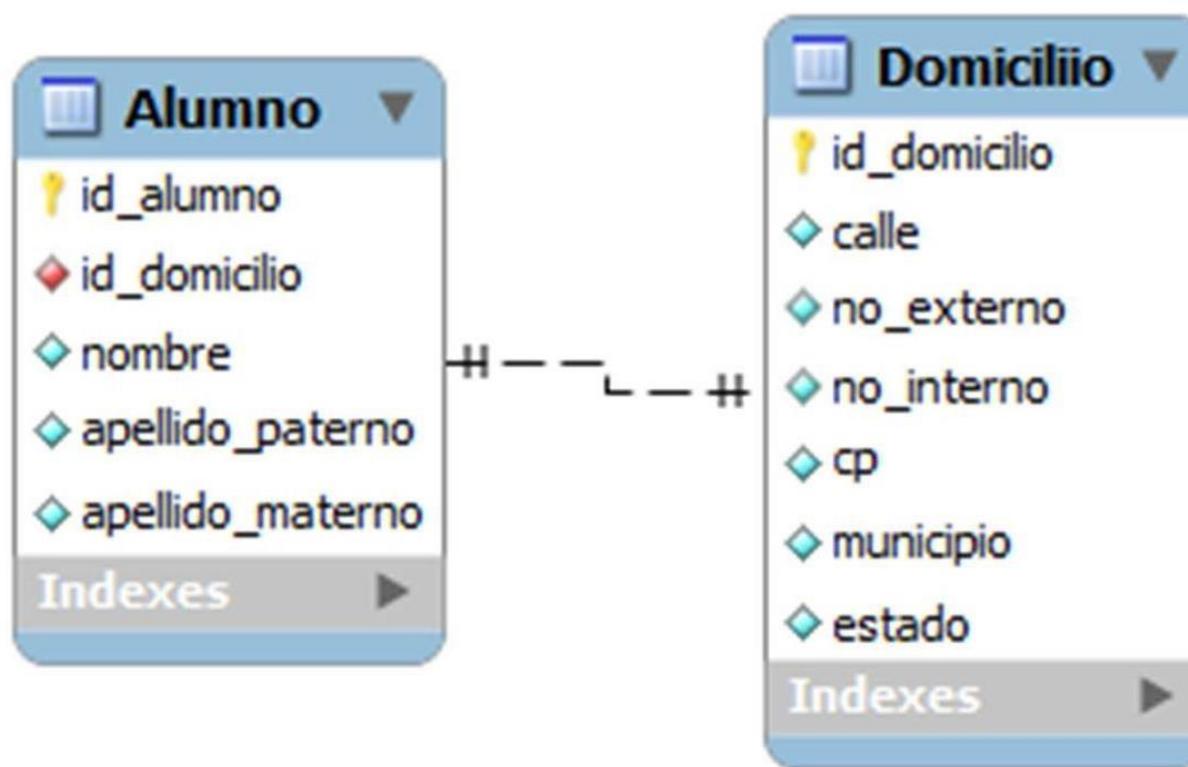
### 3. Java Persistence Api (JPA)

- A continuación realizaremos un análisis de cada relación descrita.



# 3. Java Persistence Api (JPA)

Ejemplo de Relación 1 a 1  
(Un Alumno tiene Un Domicilio)



### 3. Java Persistence Api (JPA)

- En la figura podemos observar una relación de 1 a 1, en la cual una entidad Alumno tiene una relación con sólo un domicilio, y viceversa, esto es, la cardinalidad entre las entidades es de 1 a 1.
- Podemos observar que la clase de Alumno es la que guarda la referencia de un objeto Domicilio, para mantener una navegabilidad unidireccional y que a partir de un objeto Alumno podamos recuperar el objeto Domicilio asociado.
- Es importante destacar que el manejo de relaciones es por medio de objetos, y no atributos aislados, esto nos permitirá ejecutar queries con JPQL (Java Persistence query language) que recuperen objetos completos.

### 3. Java Persistence Api (JPA)

The image shows two Java code editors side-by-side. The left editor is titled 'Alumno.java' and contains the following code:

```
@Entity
public class Alumno implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id_alumno")
    private int idAlumno;

    //Más atributos...

    //relación unidireccional one-to-one con Domicilio
    @OneToOne
    @JoinColumn(name="id_domicilio")
    private Domicilio domicilio;
```

The right editor is titled 'Domicilio.java' and contains the following code:

```
@Entity
public class Domicilio implements Serializable {
    private static final long serialVersionUID = 1L;

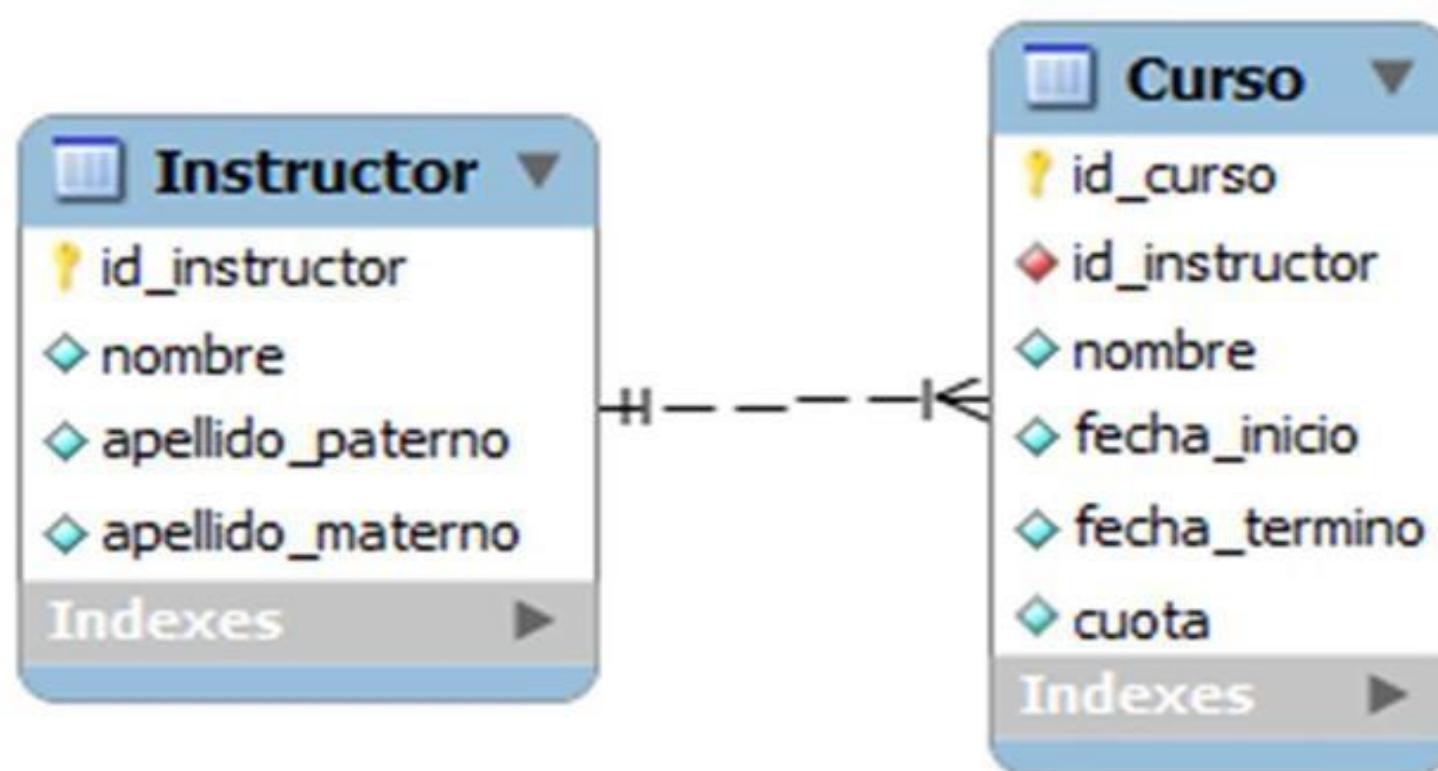
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id_domicilio")
    private int idDomicilio;

    //Mas atributos...
```

A red arrow points from the `@JoinColumn(name="id_domicilio")` annotation in the `Alumno.java` code to the `@Column(name="id_domicilio")` annotation in the `Domicilio.java` code.

### 3. Java Persistence Api (JPA)

Ejemplo de Relación 1 a Muchos  
(Un Instructor imparte Muchos Cursos)



### 3. Java Persistence Api (JPA)

- Cuando un objeto de Entidad está asociado con una colección de otros objetos de Entidad, es más común representar esta relación como una relación de **Uno a Muchos**.
- Si queremos saber desde la clase Curso qué instructor tiene asociado, deberemos agregar el mapeo bidireccional (ManyToOne) hacia Alumno. Y en la clase Alumno, se especifica una relación uno a muchos (OneToMany) hacia una colección de objetos de tipo Curso, el cual puede ser una estructura de datos Set o un List, dependiendo si queremos manejar orden o no, respectivamente.

### 3. Java Persistence Api (JPA)

- Si no queremos manejar una relación bidireccional, basta con eliminar la definición de alguna de las clases y así tendremos una relación unidireccional.

```
Instructor.java
@Entity
public class Instructor implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id_instructor")
    private int idInstructor;

    //Más atributos...

    //bi-directional many-to-one association to Curso
    @OneToMany(mappedBy="instructor")
    private Set<Curso> cursos;
}

Curso.java
@Entity
public class Curso implements Serializable {
    private static final long serialVersionUID = 1L;

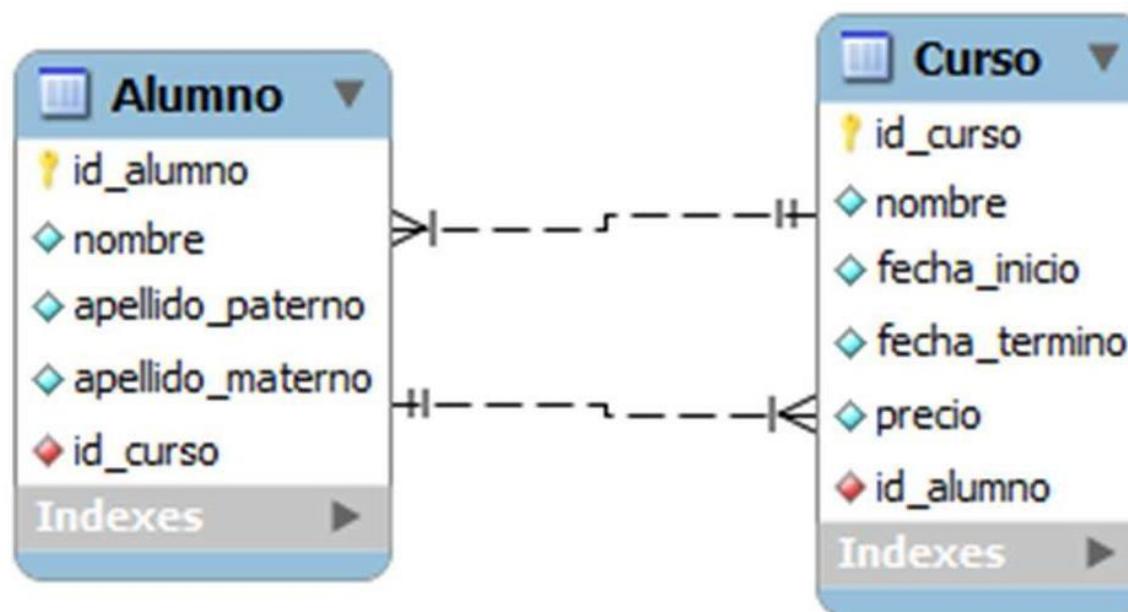
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id_curso")
    private int idCurso;

    //Más atributos..

    //bi-directional many-to-one association to Instructor
    @ManyToOne
    @JoinColumn(name="id_instructor")
    private Instructor instructor;
}
```

### 3. Java Persistence Api (JPA)

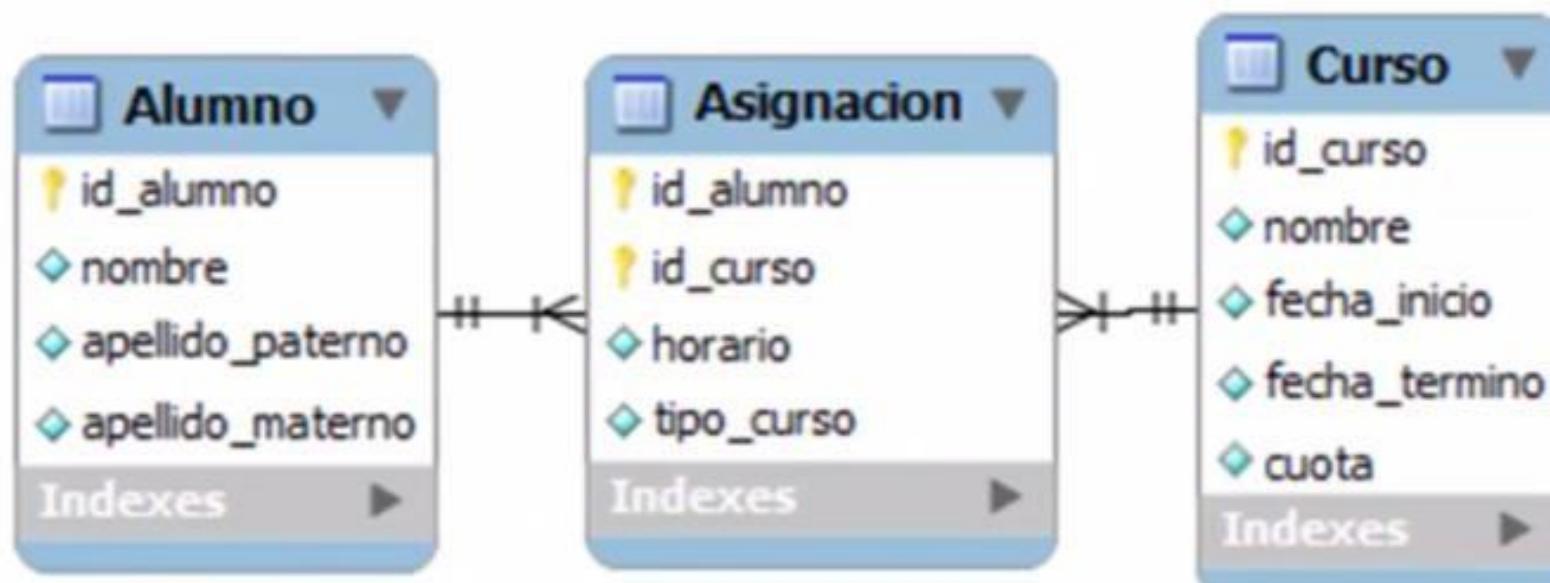
Ejemplo de Relación Muchos a Muchos  
(Un Alumno tiene Muchos Cursos y un Curso tiene  
Muchos Alumnos)



### 3. Java Persistence Api (JPA)

- En la figura podemos observar una relación muchos a muchos @ManyToMany, pero ya está desnormalizada, esto con el objetivo de no utilizar relaciones muchos a muchos directamente, ya que no es recomendable.
- Aplicamos una normalización agregando una tabla transitiva llamada asignación y así convertir la relación de 1 a Muchos.

### 3. Java Persistence Api (JPA)



# 3. Java Persistence Api (JPA)

```

@Entity
public class Alumno implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id_alumno")
    private int idAlumno;

    //Más atributos...
}

@Entity
public class Curso implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id_curso")
    private int idCurso;

    //Más atributos...

    //bi-directional many-to-one association to Asignacion
    @OneToMany(mappedBy="curso")
    private Set<Asignacion> asignaciones;
}

@Entity
public class Asignacion implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id_asignacion")
    private int idAsignacion;

    //Más atributos...

    //bi-directional many-to-one association to Alumno
    @ManyToOne
    @JoinColumn(name="id_alumno")
    private Alumno alumno;

    //bi-directional many-to-one association to Curso
    @ManyToOne
    @JoinColumn(name="id_curso")
    private Curso curso;
}

```

# 3. Java Persistence Api (JPA)

## Fetching en Relaciones

### Lazy Loading: Carga Retardada

```
@Entity  
public class Alumno implements Serializable {  
    private static final long serialVersionUID = 1L;  
  
    //Atributos...  
  
    //bi-directional many-to-one association to Domicilio  
    //relación de tipo Lazy, No se recuperan los datos  
    //del objeto Domicilio, sino hasta que son solicitados  
    @OneToOne(fetch=FetchType.LAZY)  
    @JoinColumn(name="id_domicilio")  
    private Domicilio domicilio;
```

### Eager Loading: Carga Inmediata

```
@Entity  
public class Alumno implements Serializable {  
    private static final long serialVersionUID = 1L;  
  
    //Atributos...  
  
    //bi-directional many-to-one association to Domicilio  
    //relación de tipo Eager, SI se recuperan los datos  
    //del objeto Domicilio desde que se realiza la consulta  
    @OneToOne(fetch=FetchType.EAGER)  
    @JoinColumn(name="id_domicilio")  
    private Domicilio domicilio;
```

### 3. Java Persistence Api (JPA)

- Cuando recuperamos información de diferentes objetos de entidad (información de distintas tablas de base de datos), en ocasiones no hay necesidad de recuperar toda la información y sobrecargar la memoria Java, ya que el usuario NO necesita ver la información completa en ese momento, sino en consultas subsecuentes.
- Por ello JPA maneja el concepto de Lazy Loading (carga retardada), con lo cual logramos **NO cargar** las colecciones de entidades asociadas a cierta clase de Entidad y recuperar sólo la información que sea relevante para esa transacción.

### 3. Java Persistence Api (JPA)

- Un error común que nos podemos encontrar es **Lazy Loading Exception**. Este error se produce cuando ejecutamos un query de tipo Lazy y queremos acceder a objetos que no fueron recuperados en la consulta inicial, por lo que se debe tener una transacción activa para poder recuperar los datos que no fueron cargados (puede ser durante la misma transacción o crear una nueva)
- Ejemplo de consulta Lazy Loading con:  
`SELECT p FROM Persona p JOIN p.domicilio WHERE ...`

### 3. Java Persistence Api (JPA)

- Por otro lado, tenemos el concepto de **Eager Loading**, esto es lo opuesto a Lazy Loading, y se utiliza cuando queremos recuperar todos los objetos de una colección asociada a un bean de Entidad. Con esto ya no necesitaremos volver a realizar consultas posteriores a la base de datos.
- Ejemplo de consulta Eager Loading JPQL (se utiliza la palabra reservada Fetch):

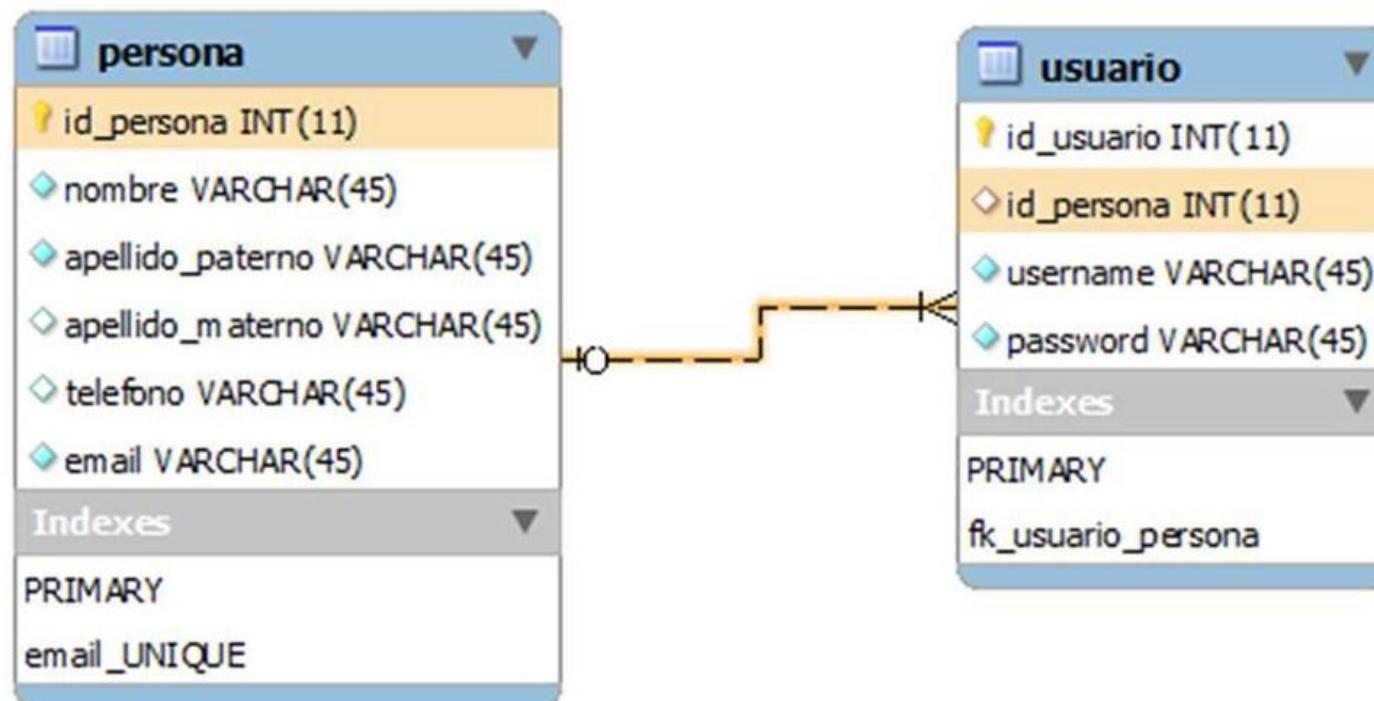
```
SELECT p FROM Persona p JOIN FETCH p.domicilio WHERE ...
```

### 3. Java Persistence Api (JPA)

- Otro concepto muy interesante en JPA es el concepto de **persistencia en Cascada**.
- Por ejemplo, si queremos persistir un objeto de tipo Usuario, pero además queremos insertarlo con sus dependencias (Persona) y que JPA se encargue de generar las llaves primarias respectivas, así como la generación de las sentencias *insert* en el orden adecuado (primero inserta el objeto persona y después el objeto Usuario), podemos apoyarnos de la configuración de persistencia en cascada.

# 3. Java Persistence Api (JPA)

## Guardado en Cascada



### 3. Java Persistence Api (JPA)

- Para configurar nuestra clase de entidad Usuario con persistencia en cascada utilizamos la siguiente sintaxis:

```
@Entity  
public class Usuario {  
  
    //Más atributos...  
  
    //bi-directional many-to-one association to Persona  
    @ManyToOne(cascade={CascadeType.ALL})  
    @JoinColumn(name="id_persona")  
    private Persona persona;
```

- Tenemos los siguientes valores
  - all, persist, merge, remove, refresh
- En todos estos casos es posible aplicar el concepto de persistencia en cascada.

# 3. Java Persistence Api (JPA)

## Java Persistence Query Language (JPQL)

### Java Persistence Query Language ( JPQL):

- ✓ Lenguaje de Consulta, similar a SQL pero utilizando objetos Java.
- ✓ Queries Parametrizables.
- ✓ Consola de Ejecución en IDE's como Eclipse o MyEclipse.
- ✓ Consultas Avanzadas con recuperación de colecciones de datos.

### Características de JPQL:

- ✓ Uso de select, from y where y subselects.
- ✓ Sensible a Mayúsculas/Minúsculas.
- ✓ Asociaciones, uso de joins y fetch.
- ✓ Uso de expresiones y operadores como: +, >, between, upper, etc.
- ✓ Uso de Funciones de agregación, tales como: avg, sum, count, etc.
- ✓ Uso de order by y group by

### 3. Java Persistence Api (JPA)

- Java Persistence Query Language (JPQL) nos permite recuperar información de la Base de Datos. JPQL se enfoca en ejecutar queries que regresen objetos Java, en lugar de datos individuales.
- **¿Qué es Java Persistence Query Language ( JPQL)?:**
  - Es un Lenguaje de Consulta, similar a SQL pero utilizando objetos Java.
  - Permite ejecutar Queries Parametrizables.
  - Cuenta con una Consola de Ejecución en IDE's como Eclipse o MyEclipse.
  - Se pueden ejecutar Consultas Avanzadas para recuperar colecciones de datos.

Ejemplo consulta con JPQL: **SELECT p FROM Persona p**

### 3. Java Persistence Api (JPA)

- JPA permite recuperar los objetos de diferentes maneras, tanto utilizando una sintaxis muy similar a SQL, pero también ofrece otras alternativas, utilizando código Java, conocido como API de Criteria y Query By Example.
- **Tipos de Queries:**
  - **Dynamic queries:** Consultas que reciben parámetros en tiempo de ejecución.
  - **Named queries:** Consultas ya creadas previamente, y que se pueden ejecutar solo utilizando el nombre.
  - **Native queries:** Consultas con SQL nativa, ya que hay casos de uso que lo requieren.
  - **Criteria API queries:** Consultas con una sintaxis con código Java, en lugar de SQL.

# 3. Java Persistence Api (JPA)

## Queries de Tipo Select en JPQL

A continuación revisaremos varios ejemplos utilizando JPQL:

1) Consulta de todas las Personas: `select p from Persona p`

2) Consulta de la Persona con id = 1: `select p from Persona p where p.idPersona = 1`

3) Consulta de la Persona por nombre: `select p from Persona p where p.nombre = 'Juan'`

4) Consulta de datos individuales, se crea un arreglo (tupla) de tipo object de 3 columnas:

```
select p.nombre as Nombre, p.apePaterno as Paterno, p.apeMaterno as Materno from Persona p
```

5) Obtiene el objeto Persona y el id, se crea un arreglo de tipo Object con 2 columnas:

```
select p, p.idPersona from Persona p
```

6) Obtiene la lista de alumnos, utilizando el constructor del idPersona

```
select new com.gm.sga.domain.Persona( p.idPersona) from Persona p
```

7) Regresa el valor mínimo y máximo del idPersona (Scalar results)

```
select min(p.idPersona) as MinId, max(p.idPersona) as MaxId, count(p.idPersona) as Contador from Persona p
```

8) Extrae los nombres de alumnos que son distintos

```
select count(distinct p.nombre) from Persona p
```

# 3. Java Persistence Api (JPA)

9) Concatena y Convierte a mayúsculas el nombre y apellido:

```
select CONCAT (p.nombre, ' ', p.apePaterno) as Nombre FROM Persona p
```

10) Obtiene el objeto alumno con id igual al parámetro:

```
select p from Persona p where p.idPersona = :id
```

11) Obtiene las p

```
select p from Persona p where upper(p.nombre) like upper(:param1)
```

12) Uso de between:

```
select p from Persona p where p.idPersona between 1 and 2
```

13) Uso del ordenamiento:

```
select p from Persona p where p.idPersona, sin importar mayúscula/minúsculas:  
order by p.nombre desc, p.apePaterno desc
```

14) Uso de un subquery (el soporte de esta funcionalidad depende de la base de datos utilizada)

```
select p from Persona p  
where p.idPersona in ( select min(p1.idPersona) from Persona p1)
```

15) Uso de join con lazy loading: `select u from Usuario u join u.persona p`

16) Uso de left join con eager loading: `select u from Usuario u left join fetch u.persona`

# 3. Java Persistence Api (JPA)

## Código Java para ejecutar un query JPQL

El siguiente código Java permite ejecutar los queries JPQL descritos:

```
@Test
public void testActualizarObjeto() {

    String jpql = null;
    List<Persona> personas;

    EntityTransaction tx1 = em.getTransaction();
    tx1.begin();

    //1) Consulta de todas las Personas:
    log.debug("1) Consulta de todas las Personas");

    jpql = "select p from Persona p";

    personas = em.createQuery(jpql).getResultList();
    for(Persona p : personas){
        log.debug(p);
    }

    tx1.commit();
}
```

### 3. Java Persistence Api (JPA)

- Como podemos observar en la figura, utilizando los queries descritos anteriormente podemos ejecutar con ayuda del API de JPA, las sentencias JPQL que hemos construido. Existen distintas formas de procesar las consultas dependiendo de la información a recuperar.

Por ejemplo, si queremos recuperar la lista de resultados:

```
String jpql = "select p from Persona p";
List<Persona> personas = em.createQuery(jpql).getResultList();
```

Si queremos recuperar sólo un registro:

```
String jpql = "select p from Persona p where p.idPersona = 1";
Persona persona = (Persona) em.createQuery(jpql).getSingleResult();
```

### 3. Java Persistence Api (JPA)

Si queremos procesar un conjunto de valores (tupla):

```
String jpql = "select p.nombre as Nombre, p.apePaterno as Paterno from Persona p";
iter = em.createQuery(jpql).getResultList().iterator();
while(iter.hasNext()){
    tupla = (Object[]) iter.next();
    String nombre = (String) tupla[0];
    String apePat = (String) tupla[1];
}
```

Si queremos enviar parámetros a un query:

```
String jpql = "select p from Persona p where p.idPersona = :id";
Query q = em.createQuery(jpql);
q.setParameter("id", 1);
persona = (Persona) q.getSingleResult();
```

# 3. Java Persistence Api (JPA)

## API de Criteria en JPA

### API Criteria en JPA (\* Nuevo en JPA 2.0)

- ✓ El API de Criteria es una alternativa al uso de JPQL o SQL Nativo
- ✓ Permite la combinación de campos de criterio complejos (ej. Una pantalla de búsqueda avanzada)
- ✓ Permite crear queries dinámicos complejos más fácilmente, evitando el concatenado de cadenas.

### Características del API de Criteria

- ✓ Son escritos en código Java
- ✓ Son typesafe (parámetros revisados en tiempo de compilación)
- ✓ Son queries portables (funcionan en cualquier base de datos)
- ✓ Se utilizan clases de Java en lugar de cadenas JPQL o SQL
- ✓ Permite utilizar expresiones, joins, ordenamiento, etc

### 3. Java Persistence Api (JPA)

- La construcción de queries dinámicas conlleva mucha concatenación de cadenas si estamos utilizando JDBC directo, incluso en queries complejas utilizando JPQL podemos tener demasiados parámetros concatenados en la cadena JPQL.
- Para simplificar este proceso, JPA en su versión 2.0 liberó el API de Criteria, el cual está basado en el API de Criteria de Hibernate, entre otros frameworks ORM, por lo tanto si ya se tiene conocimiento de Hibernate y su API de Criteria o alguno similar, será más sencillo aprender esta API.

### 3. Java Persistence Api (JPA)

- Con el API de Criteria, es posible construir consultas dinámicas complejas, permitiendo configurar desde el mismo lenguaje de programación (y no de tipo SQL) los filtros necesarios para dicha consulta.
- El API de Criteria es muy utilizado cuando tenemos pantallas con demasiados filtros de búsqueda, y el usuario tiene la opción de seleccionar uno o más filtros. En cambio las consultas con JPQL funcionan muy bien cuando tenemos un número establecido de parámetros y la mayoría de nuestro query es estático (no existe demasiada concatenación de parámetros).

### 3. Java Persistence Api (JPA)

- La forma más rápida de aprender a utilizar el API de Criteria es comparando las consultas que realizamos con JPQL, ya que con ello podremos ver las diferencias y ver los pros y contras de cada API. Así que a continuación revisaremos y compararemos varias de estas queries utilizando las APIs mencionadas.

# 3. Java Persistence Api (JPA)

## Código Java para ejecutar un query con API Criteria

El siguiente código Java permite ejecutar los queries API Criteria:

```
// Query utilizando el API de Criteria
// 1) Consulta de todas las Personas:

log.debug("\n1) Consulta de todas las Personas");

//1. El objeto EntityManager crea instancia CriteriaBuilder
CriteriaBuilder cb = em.getCriteriaBuilder();

//2. Se crear un objeto CriteriaQuery
CriteriaQuery<Persona> criteriaQuery = cb.createQuery(Persona.class);

//3. Creamos el objeto Raiz del query
Root<Persona> fromPersona = criteriaQuery.from(Persona.class);

//4. Seleccionamos lo necesario del from
criteriaQuery.select(fromPersona);

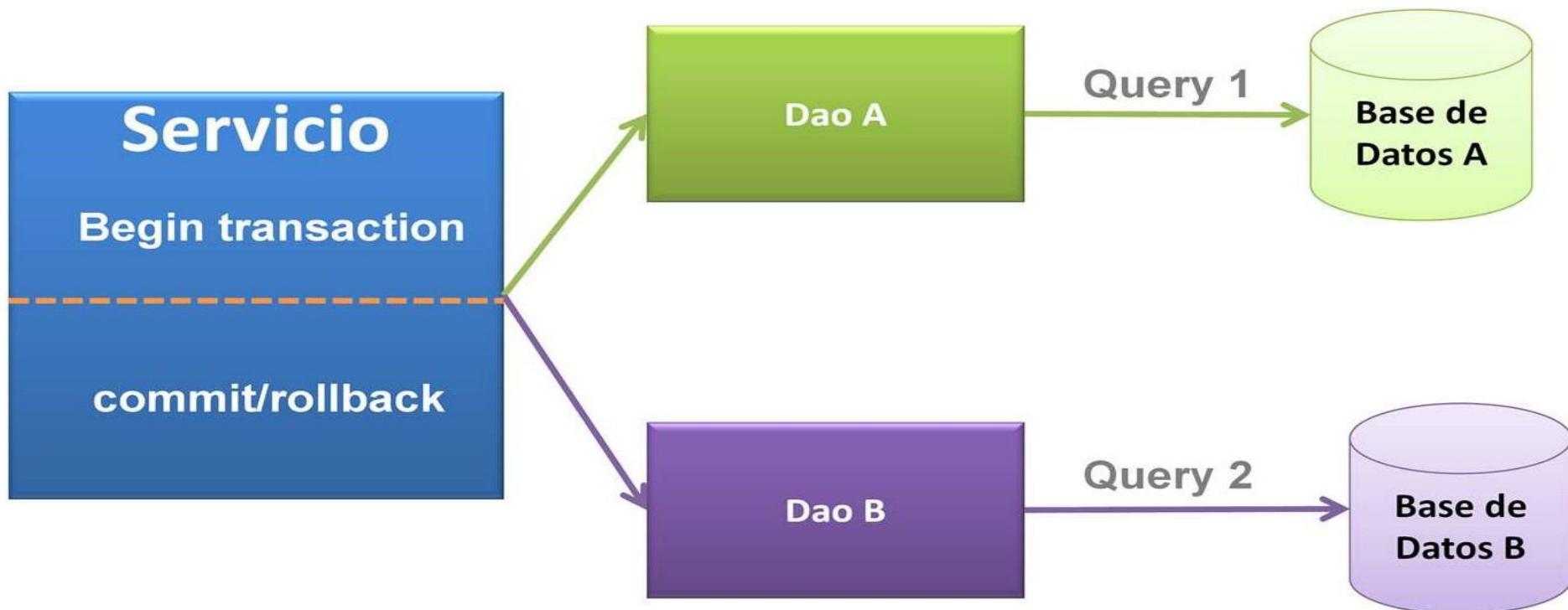
//5. Creamos el query typeSafe
TypedQuery<Persona> query = em.createQuery( criteriaQuery);

//6. Ejecutar la consulta
List<Persona> personas = query.getResultList();
mostrarPersonas(personas);
```

# 3. Java Persistence Api (JPA)

## ¿Qué es una Transacción?

- Una transacción se conoce como una unidad de trabajo atómica, es decir, se realiza toda o nada del método transaccional.



### 3. Java Persistence Api (JPA)

- El Manejo Transaccional es uno de los temas cruciales en cuanto a requerimientos para aplicaciones empresariales. Esta motivación surge debido a que **en todo sistema empresarial nos interesa mantener la integridad de nuestra información**, con esto en mente es que surge el tema de transacciones.
- El objetivo de una transacción es ejecutar todas las líneas de código de nuestro método y guardar finalmente la información en un repositorio, por ejemplo en nuestro caso, una base de datos. Esto se conoce como ***commit*** de nuestra transacción.

### 3. Java Persistence Api (JPA)

- Si por alguna razón algo fallara en nuestro método de Servicio, se daría marcha atrás a los cambios realizados en la base de datos. Esto se conoce como ***rollback***.
- Lo anterior permite que nuestra información, ya sea que se una única base de datos o no, esté íntegra, y no exista posibilidad de datos corruptos por errores o fallos en la ejecución de nuestro código Java.

### 3. Java Persistence Api (JPA)

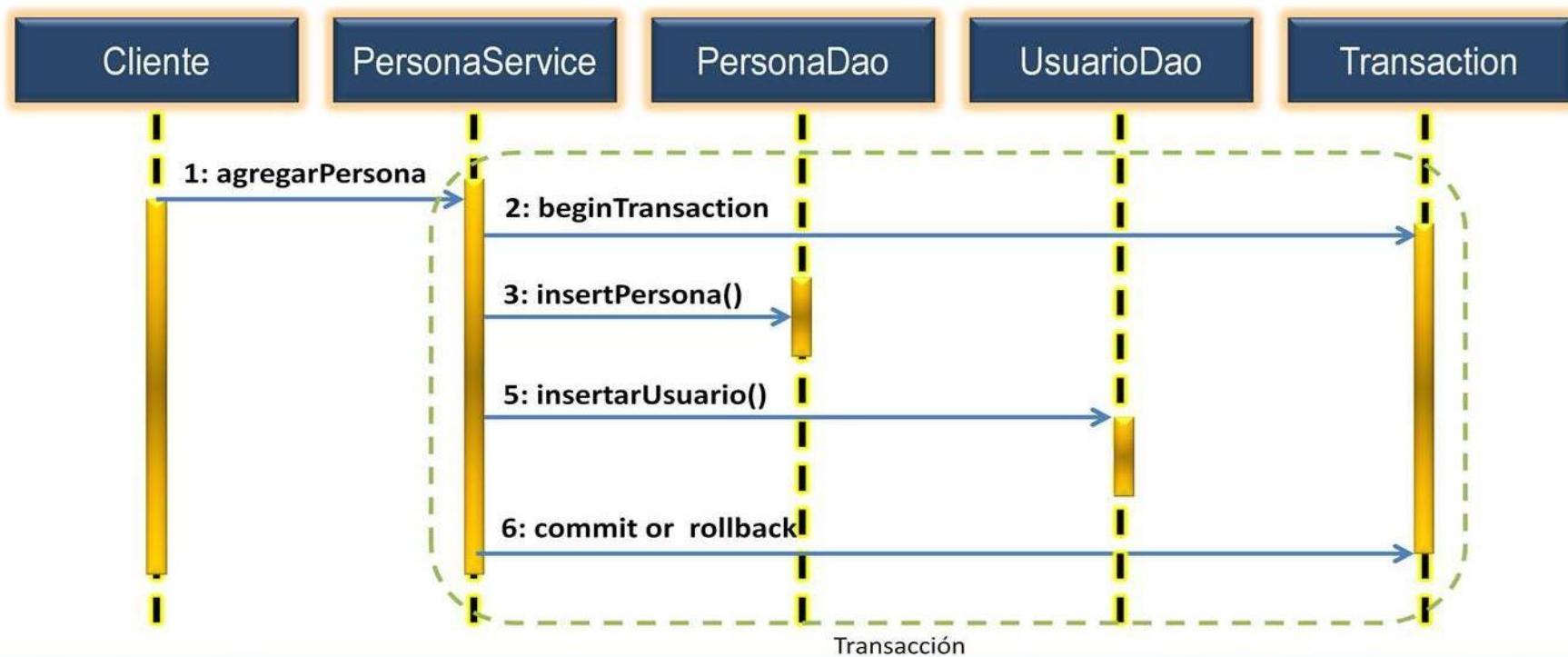
- Las características de una transacción tienen el acrónimo **ACID**:
  - **Atomicidad**: Las actividades de un método se consideran como una unidad de trabajo. Esto se conoce como *Atomicidad*. Este concepto asegura que **todas las operaciones en una transacción se ejecuta todo o nada**.
  - **Consistente**: Una vez que termina una transacción (sin importar si ha sido exitosa o no) la información queda en estado *consistente*, ya que se realizó todo o nada, y por lo tanto **los datos no deben estar corruptos en ningún aspecto**.

### 3. Java Persistence Api (JPA)

- **Aislado**: (Isolated) Múltiples usuarios pueden utilizar los métodos transaccionales, sin afectar el acceso de otros usuarios. Sin embargo debemos prevenir errores por accesos múltiples, *aislando* en la medida de lo posible nuestros métodos transaccionales. **El aislamiento normalmente involucra el bloqueo de registros o tablas de base de datos**, esto se conoce como locking.
- **Durable**: Sin importar si hay una caída del servidor, una transacción exitosa debe guardarse y *perdurar* posterior al término de una transacción.

### 3. Java Persistence Api (JPA)

- Al crear aplicaciones empresariales debemos poner especial atención en la persistencia. Además, una aplicación empresarial, según hemos estudiado, se divide en distintas capas, las cuales tienen responsabilidades bien definidas.



### 3. Java Persistence Api (JPA)

- La capa de datos es la responsable de establecer la comunicación con la base de datos a través del objeto Entity Manager.
- La transacción comienza desde la capa de servicio, y se propaga a la capa de datos. Esto se debe a que la capa de servicio puede tener comunicación con muchas clases Dao, y por lo tanto la transacción termina hasta que el método de negocio haya insertado, modificado, eliminado y seleccionado todos los datos requeridos de la base de datos, aplicando las características del manejo transaccional que hemos estudiado.

### 3. Java Persistence Api (JPA)

- Los métodos de la capa de Servicio, son los que contienen mucha de la lógica de negocio de la aplicación, y por lo tanto es en este nivel donde definimos el manejo transaccional, ya que si lo aplicamos al nivel de la capa de datos, el manejo de commit/rollback por cada operación de un DAO, afectaría las operaciones restantes que tenga un método de negocio.
- Para indicar si un método maneja un método distinto de programación se utiliza el código antes de la definición del método:
  - [@TransactionAttribute](#)(TransactionAttributeType.SUPPORTS)

### 3. Java Persistence Api (JPA)

#### Configuración de la Propagación en Java EE

Demarcación de transacciones por medio de Container-Managed Transactions (CMTs)

Tipo de Propagación	Significado
MANDATORY	El método tiene que ejecutarse dentro de una transacción, de lo contrario se lanzará una excepción.
REQUIRED	El método DEBE ejecutarse dentro de una transacción. Si ya existe una transacción el método la utilizará, de lo contrario creará una nueva.
REQUIRES_NEW	El método DEBE ejecutarse dentro de una transacción. Si ya existe una transacción, se suspende durante la ejecución del método, de lo contrario creará una nueva.
SUPPORTS	Indica que el método no requiere el manejo transaccional, pero puede participar de una transacción si ya hay alguna ejecutándose.
NOT_SUPPORTED	El método NO debe ejecutarse en una transacción. Si ya existe una transacción, se suspenderá hasta la conclusión del método.
NEVER	El método NO debe ejecutarse en una transacción, de lo contrario lanza una excepción.

# 3. Java Persistence Api (JPA)

## Ejemplo Código de Transacciones

Ejemplo de código de transacciones administradas por el contenedor (CMT):

```
@Stateless  
public class PersonaServiceImpl {  
  
    @Resource  
    private SessionContext contexto;  
  
    public void modificarPersona(Persona persona) {  
        try {  
  
            personaDao.updatePersona(persona);  
  
        } catch (Throwable t) {  
  
            contexto.setRollbackOnly();  
        }  
    }  
    //Más métodos...  
}
```

### 3. Java Persistence Api (JPA)

- Según hemos visto, el manejo de transacciones se lleva a cabo en los EJB de la capa de servicio o negocio. Como observamos en el código mostrado, el EJB PersonaService contiene métodos de servicio, los cuales por defecto son transaccionales al ejecutarse en un contenedor empresarial.
- El comportamiento por defecto de un método EJB es de tipo REQUIRED. Sin embargo si queremos modificarlo podemos utilizar el código

TransactionAttribute(TransactionAttributeType.SUPPORTS)

ya sea a nivel de la clase para que afecte a todos los métodos, o al nivel del método que queremos modificar.

### 3. Java Persistence Api (JPA)

- Cualquier código que espere arrojar una excepción, y si queremos aplicar rollback de manera explícita deberemos utilizar setRollbackOnly() del objeto SessionContext, el cual podemos injectar directamente utilizando la anotación [@Resource](#).
- Sin embargo, utilizar el método setRollbackOnly no provocará el rollback de la transacción de manera inmediata, es sólo una indicación de que en cuanto el contenedor termine de ejecutar el método, deberá realizarse el rollback.

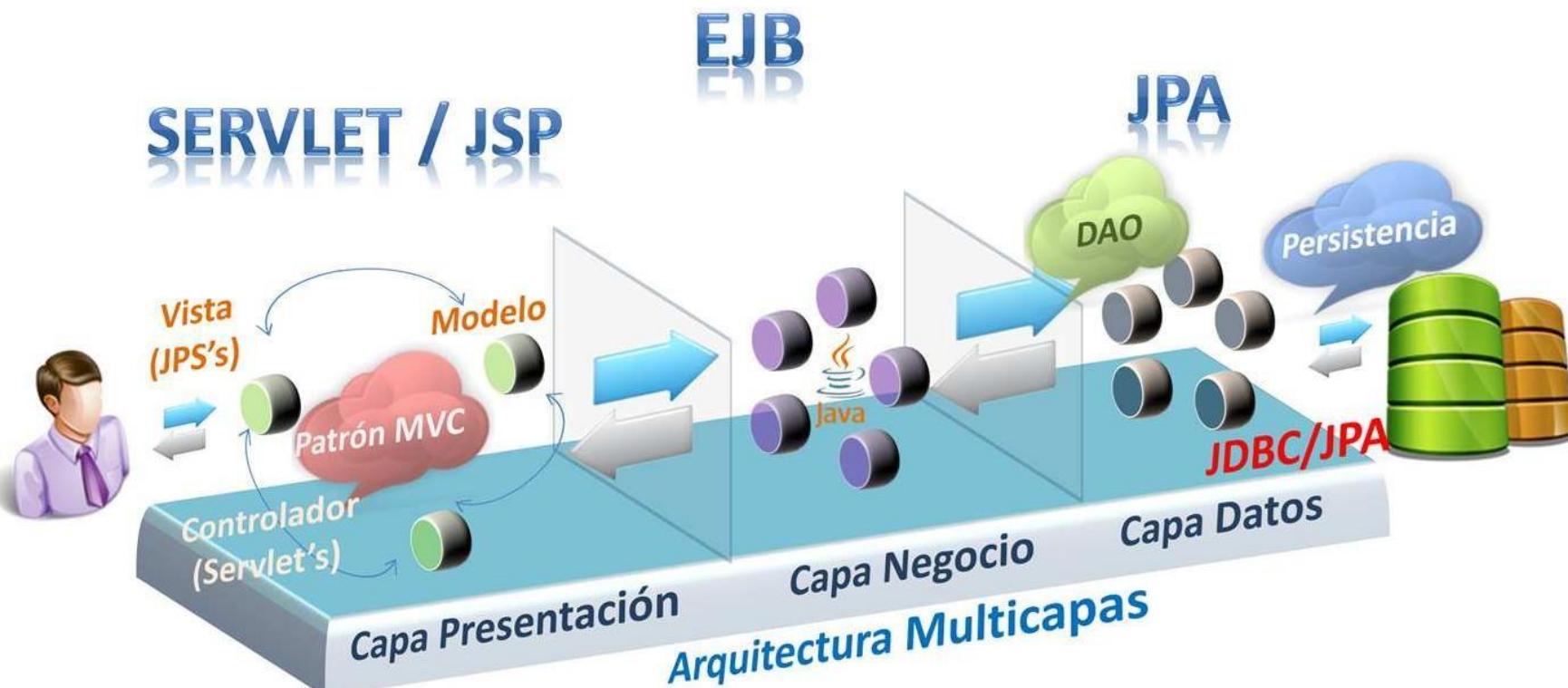
### 3. Java Persistence Api (JPA)

- Cada transacción JTA está asociada con la ejecución de un hilo (Thread), así que **solo se puede ejecutar una transacción a la vez**. De tal manera que si una transacción se encuentra activa, el usuario NO puede iniciar otra dentro del mismo hilo (a menos que dicha transacción sea suspendida).

# Índice

- 
1. Introducción Java Empresarial
  2. Enterprise Java Beans (EJBs) y Context Directory Inyection (CDI)
  3. Java Persistence Api (JPA)
  - 4. Servlets y jsps en Java EE**
  5. JavaServer Faces
  6. Web Services (SOAP y REST)
  7. Seguridad en JEE

## 4. Servlets y jsp's en JEE



Servidor de  
Aplicaciones Java

## 4. Servlets y jsp's en JEE

- Las razones por las cuales creamos arquitecturas empresariales multicapas son muchas, a continuación mencionamos algunas:
  - **Abstracción y Encapsulamiento:** Las capas abstraen los detalles del trabajo que realiza cada capa. Así, cada una será responsable de ciertos procesos, sin tener que involucrar a las demás capas para realizar sus tareas fundamentales.
  - **Funcionalidad claramente definida:** Cada capa define sus tareas, permitiendo delegar adecuadamente el trabajo entre el grupo de programadores.
  - **Alta Cohesión:** Esto significa que una capa realiza solamente la tarea para la cual fue creada, y las demás funciones las delega o se apoya de las demás capas.

## 4. Servlets y jsp's en JEE

- **Bajo Acoplamiento:** Debido a que cada capa tiene aplicado el concepto de abstracción, solamente existen los puntos de comunicación mínimo necesarios entre las distintas capas, permitiendo acoplarse y desacoplarse sin problemas.
- **Reutilizable:** Como resultado del bajo acoplamiento y las demás características, es posible reutilizar capas completas para distintos proyectos.

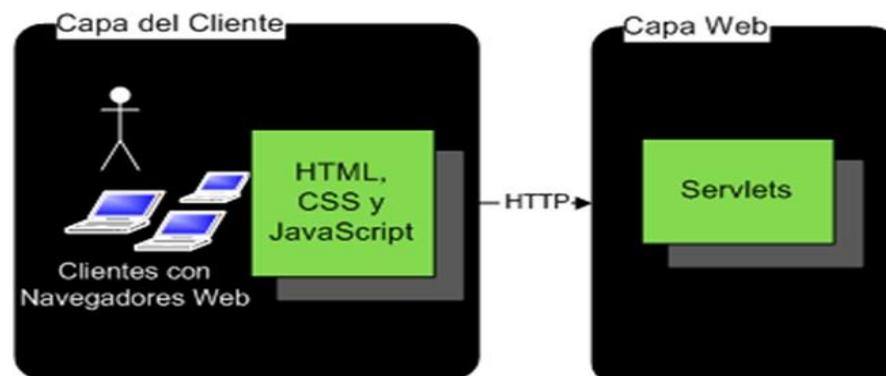
## 4. Servlets y jsp's en JEE

- Además, podemos tener los siguientes beneficios al utilizar capas en una arquitectura empresarial:
  - **Aislamiento:** Permite reducir el impacto para cambiar la tecnología, ya que podemos ir paulatinamente migrando una capa a una nueva tecnología, sin impactar demasiado en el sistema total.
  - **Rendimiento:** Si hay necesidad de distribuir las capas entre diferentes servidores es posible realizarlo, con el objetivo de agregar conceptos como escalabilidad, fiabilidad, tolerancia a fallos y en conclusión mejorar el rendimiento completo del sistema.
  - **Mejoras en pruebas:** Entre más claras sean las responsabilidades de cada capa, podemos realizar pruebas de y descartar fallas de software aislando el problema de manera muy precisa.

# 4. Servlets y jsp's en JEE

## Características de los Servlets

- Un Servlet es una clase de Java que permite procesar peticiones Web.
- Permite leer información del cliente Web (parámetros de petición).
- Permite generar una respuesta para mostrar al cliente (HTML y archivos binarios como PDF, Audio, Video, etc.)



## 4. Servlets y jsp's en JEE

- Un Servlet es el **componente más básico de la tecnología Java** para el desarrollo de aplicaciones Web. Incluso, un JSP, al ser compilado se convierte en un Servlet.
- Esta clase Java permite **procesar una petición basada en el protocolo HTTP** (Hypertext Transfer Protocol). Este es el protocolo que revolucionó en su momento el uso de Internet, ya que a través de él es posible solicitar páginas Web a Servidores de distintas tecnologías como Java, PHP,.Net, entre muchas tecnologías más.

## 4. Servlets y jsp's en JEE

- Los Servlets, son clases Java que tienen un ciclo de vida bien definido, y que son administrador por un contenedor de aplicaciones Web Java, por lo tanto, no basta con un JDK para ser ejecutados, sino que debemos tener como mínimo un servidor Tomcat o, si se requiere de integración de más tecnologías, un servidor completo de aplicaciones Java como Glassfish, WebSphere, Weblogic o JBoss.
- Una clase Servlet permite procesar la petición del cliente y puede generar una respuesta de manera dinámica. El contenido de respuesta puede ser código HTML o archivos binarios, tales como PDF, Audio, Video, etc)

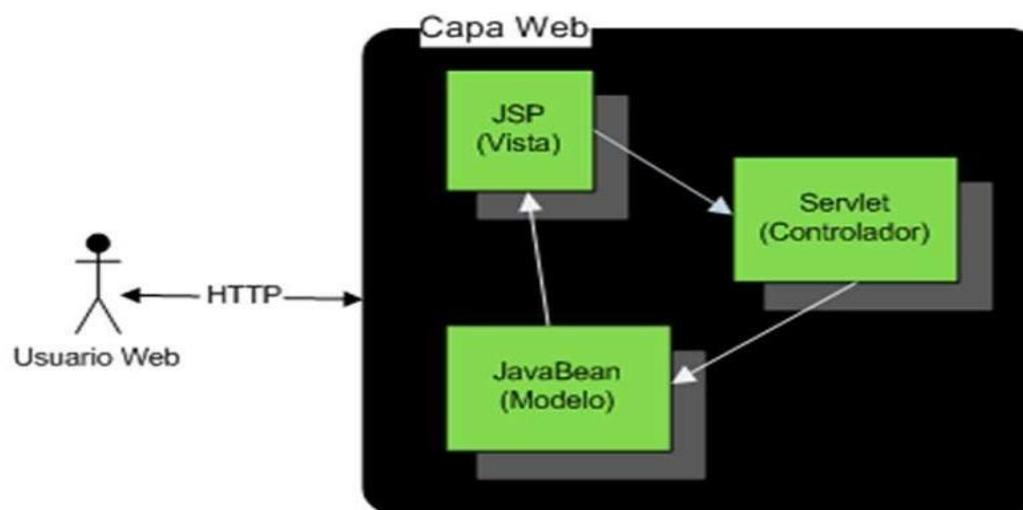
## 4. Servlets y jsp's en JEE

- Los Servlets están orientados a procesar la petición del cliente, y por lo tanto el código que contienen es código Java, pudiendo incrustar código HTML para generar la respuesta al cliente, sin embargo, precisamente para evitar esto último es que vamos a utilizar la tecnología de los JSPs, los cuales sí están orientados a tener principalmente código HTML e incrustar código Java.
- Hace ya más de una década que los Servlets aparecieron. Al día de hoy es la versión 3.0 la que está liberada, y se ha renovado para soportar de manera nativa varias características muy interesantes que se verán más adelante.

# 4. Servlets y jsp's en JEE

## Funciones de un Servlet

- Un Servlet contiene código Java, y puede agregar código HTML.
- Un Servlet se utiliza como una clase que controla el flujo de una aplicación Web (Controlador) en una arquitectura MVC (Model – View - Controller).



## 4. Servlets y jsp's en JEE

- El Servlet es el **controlador de la aplicación** y por lo tanto procesa la petición del cliente, recupera la información del modelo y envía de vuelta la respuesta al cliente, pudiéndose apoyar de los JPS's para esta última tarea, o también lo puede hacer directamente el Servlet (aunque no es recomendable).

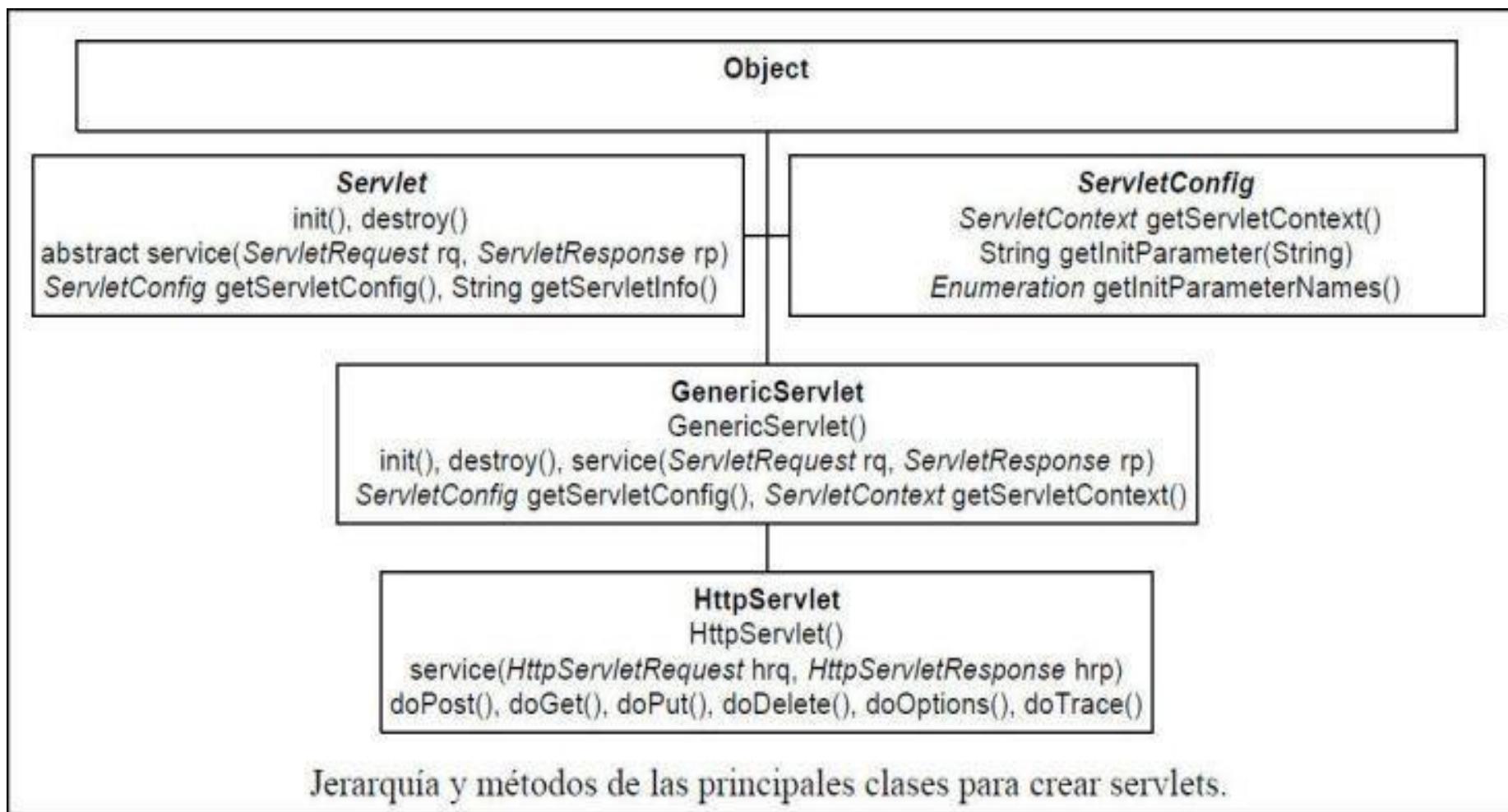
# Servlets

- **El API Servlets y el ciclo de vida.**

Los servlets usan clases e interfaces de dos paquetes:

- **javax.servlet** que contiene clases para servlets genéricos (independientes del protocolo que usen) y,
- **javax.servlet.http** (que añade funcionalidad particular de http).

# Servlets



# Servlets

- Toda clase, para que se considere un servlet, debe implementar el interfaz *javax.servlet.Servlet*.
- Para conseguirlo lo más sencillo es hacer que nuestra clase herede
  - O bien de la clase *javax.servlet.GenericServlet*
  - O *javax.servlet.http.HttpServlet*.
- Con la primera obtendremos un servlet independiente del protocolo, mientras que con la segunda tendremos un servlet HTTP.

# Servlets

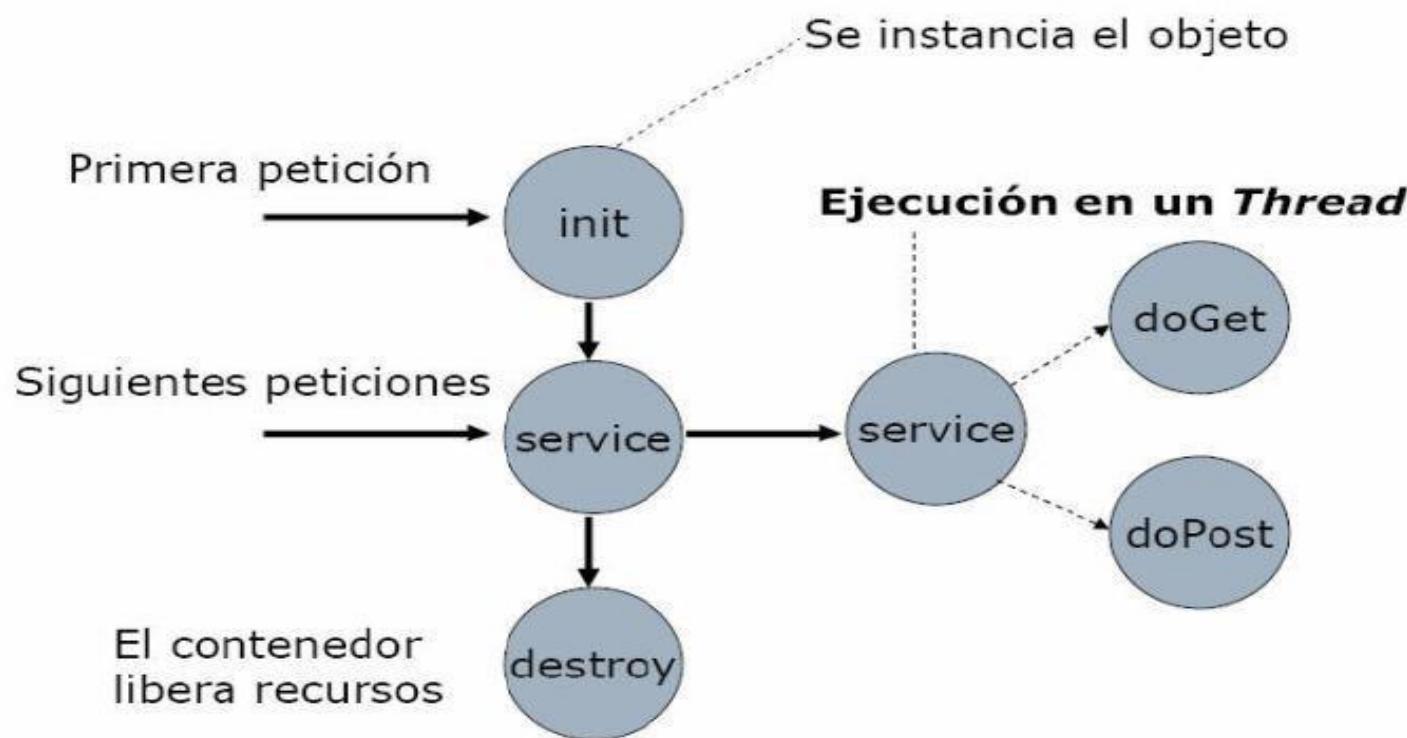
- Los servlets no tienen el método main() como los programas Java, sino que se invocan unos métodos cuando se reciben peticiones.
- A esta metodología se le llama ciclo de vida de un servlet y viene dado por tres métodos: **init, service, destroy:**
  - **INICIALIZACIÓN:** Una única llamada al método “init” por parte del servlet. Incluso se pueden recoger unos parámetros concretos con “getInitParameter” de “ServletConfig” iniciales y que operarán a lo largo de toda la vida del servlet.

# Servlets

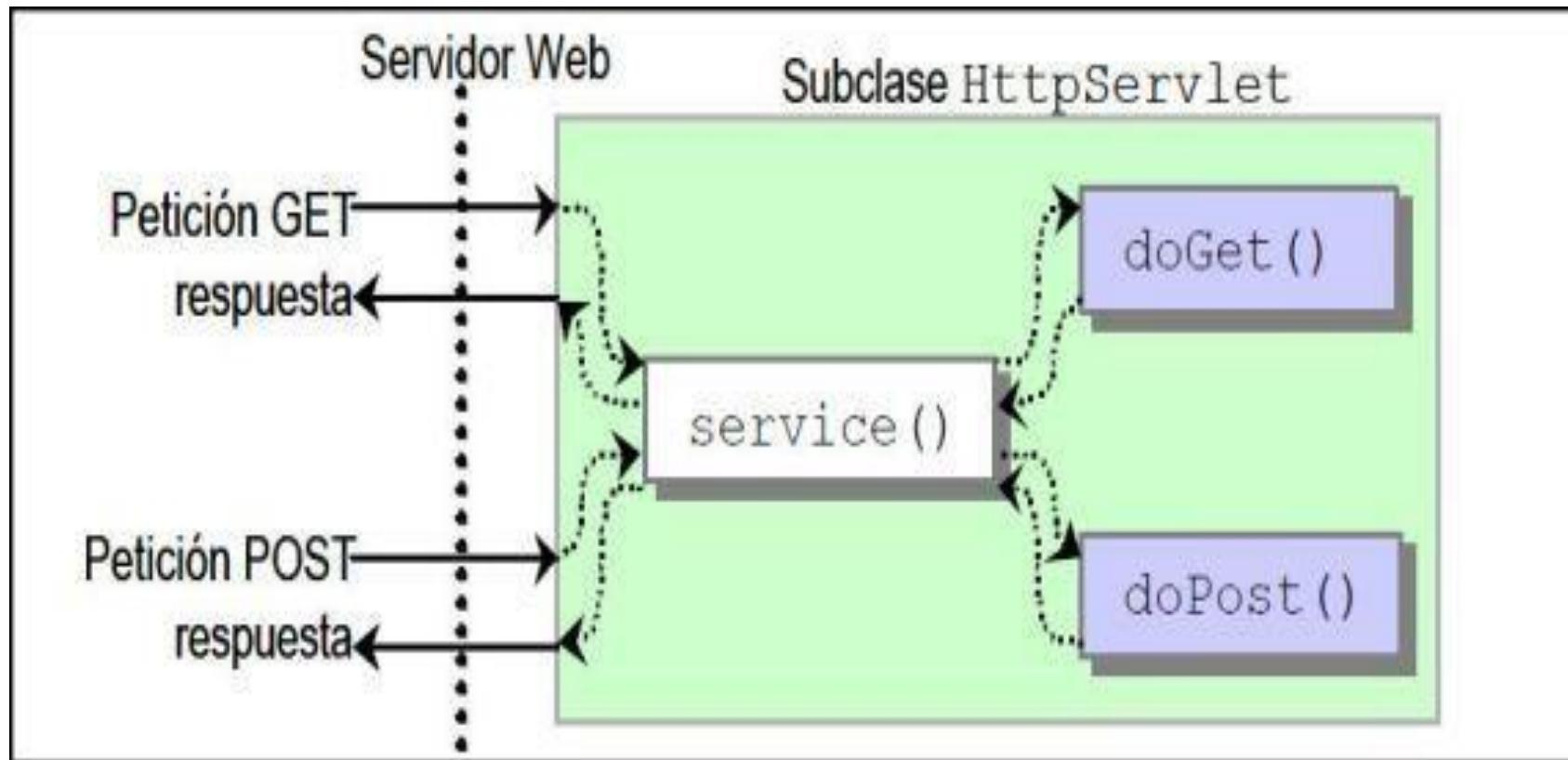
- **SERVICIO:** una llamada a service() por cada invocación al servlet para procesar las peticiones de los clientes web.
- **DESTRUCCIÓN:** Cuando todas las llamadas desde el cliente cesen o un temporizador del servidor así lo indique o el propio administrador así lo decida se destruye el servlet. Se usa el método “destroy” para eliminar al servlet y para “recoger sus restos” (garbage collection).

# Servlets

## Ciclo de Vida



# Servlets



# Servlets

- De esta manera, una vez se carga el servlet, es muy eficiente, pues sólo hay una copia cargada en memoria (se ejecutan uno o varios hilos), no hay que crear nuevos objetos (un solo objeto servlet), y tiene persistencia: puede guardar información entre peticiones, como contadores o conexiones a una base de datos.
- Esto último puede ser muchísimo más eficiente hacerlo en el método init que abrir y cerrar la conexión con la base de datos en cada petición.

# Servlets

- Pasos generales que ejecuta un Servlet Controlador:

- a) Procesamos y validamos los parámetros (si aplica)

*int idPersona = request.getParameter("idPersona");*

- b) Realizamos la lógica de presentación almacenando el resultado en JavaBeans

*Persona persona = personaService.encontrarPersonaPorId( idPersona );*

# Servlets

- c) Compartimos el objeto bean a utilizar en algún alcance (scope)

```
request.setAttribute("persona", persona );
```

- d) Redireccionamos al JSP seleccionado

```
RequestDispatcher dispatcher =request.getRequestDispatcher("listar.jsp");  
dispatcher.forward( request, response );
```

# Servlets

- Pasos generales que ejecuta un Servlet Controlador:
  - **setContextType(String str)** para establecer el tipo de respuesta que vamos a dar. Para indicar que se trata de una página web, como haremos siempre, usamos el tipo "text/html".  
*res.setContentType("text/html");*
  - **PrintWriter getWriter(void)** con el que obtendremos una clase *PrintWriter* en donde iremos escribiendo los datos que queremos que el cliente reciba  
*PrintWriter out = res.getWriter();*

# Servlets

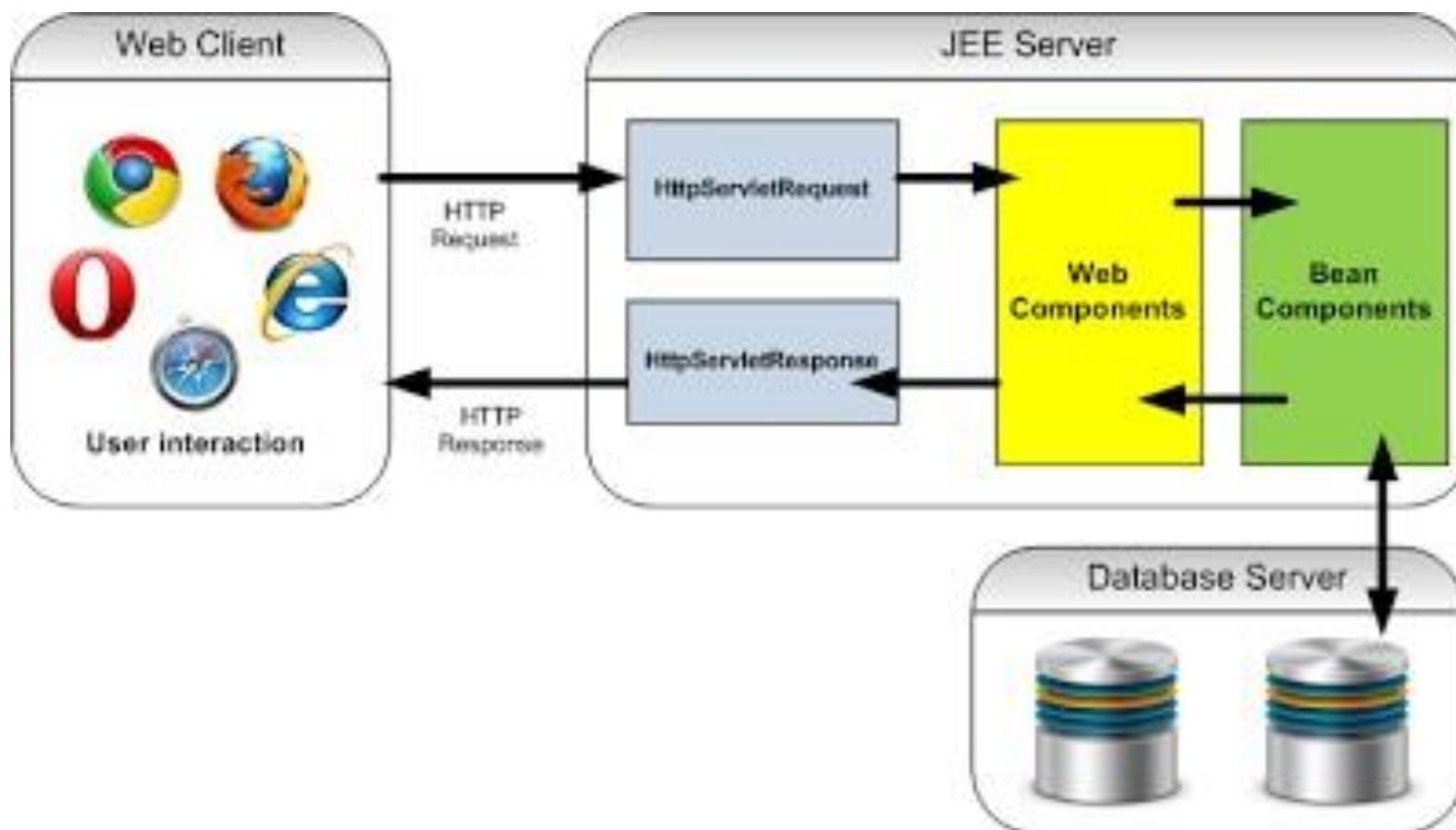
- El API de los Servlets 3.0 tiene las siguientes mejoras:
- **Facilidad de Desarrollo:**
  - Uso de Anotaciones para la declaración de Servlets, Filtros, Listeners, etc.
  - El archivo web.xml es opcional (si es utilizado sobreescribe a las anotaciones)
  - Soporte del uso de tipos genericos en el API, sin romper compatibilidad
  - Mejor soporte por defecto

# Servlets

- **Uso de Anotaciones:**

- [@WebServlet](#) – Define un Servlet
- [@WebFilter](#) – Define un filtro
- [@WebListener](#) – Define un listener
- [@WebInitParam](#) – Define un parametro de inicio
- [@MultipartConfig](#) – Define propiedades para subida de archivos
- [@ServletSecurity](#) – Define una restricción de seguridad

## 4. Servlets y jsp's en JEE



## 4. Servlets y jsp's en JEE

- Pasos generales que ejecuta un Servlet Controlador:

- a) Procesamos y validamos los parámetros (si aplica)

```
int idPersona = request.getParameter("idPersona");
```

- b) Realizamos la lógica de presentación almacenando el resultado en JavaBeans

```
Persona persona = personaService.encontrarPersonaPorId( idPersona );
```

- c) Compartimos el objeto bean a utilizar en algún alcance (scope)

```
request.setAttribute("persona", persona );
```

- d) Redireccionamos al JSP seleccionado

```
RequestDispatcher dispatcher =request.getRequestDispatcher("listar.jsp");  
dispatcher.forward( request, response );
```

# 4. Servlets y jsp's en JEE

## Características de los JSPs

- Los JavaServer Pages (JSP's) son componentes del lado del servidor, especializados en manejar código HTML, e incrustar código Java por medio de etiquetas (tags).
- Los JSP's son utilizados como componentes de presentación, es decir, para mostrar la información obtenida o procesada por los Servlets.
- Un JSP al compilarse se crea un Servlet de manera automática y al vuelo. Por ello el ciclo de vida de un JSP es muy similar al de un Servlet.

## 4. Servlets y jsp's en JEE

- Los JSPs son componentes Java del lado del servidor, los cuales se encargan de desplegar la información de respuesta hacia el cliente, generando de manera dinámica el contenido HTML necesario.
- Estos son algunos de los beneficios de utilizar los JSP"s:
  - Nos podemos enfocar en escribir código HTML, haciendo **más fácil el mantenimiento de la capa de presentación**
  - Podemos **utilizar herramientas de diseño** para crear visualmente las páginas HTML e incrustar las etiquetas dinámicas de los JSP"s
  - Separa el código de presentación del código de Java
  - Cada miembro del equipo de desarrollo se puede enfocar en distintas tareas, al **separar las responsabilidades**.

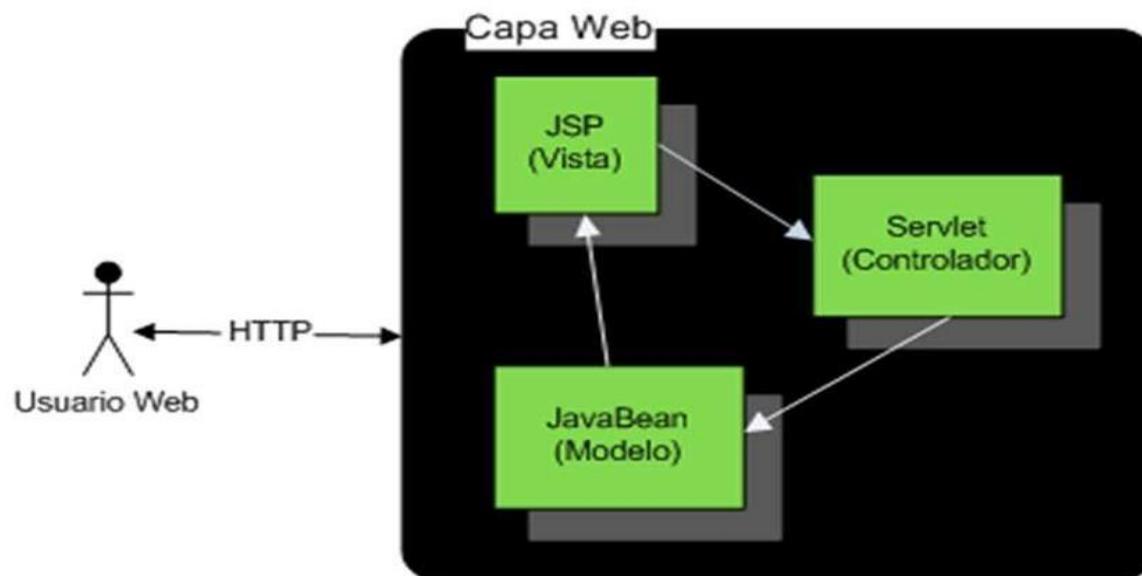
## 4. Servlets y jsp's en JEE

- La tecnología de los JSP's se apoya a su vez de otras tecnologías para simplificar el despliegue de información.
  - **Expression Language:** El lenguaje de expresión (EL) **simplifica el acceso a la información compartida por los Servlets**, según el modelo MVC, y con ello el JSP se enfoca en su tarea primordial, la cual es procesar la información para enviar la respuesta al cliente.
  - **JSTL (JSP Standard Tag Library):** La librería JSTL es un **conjunto de etiquetas que podemos utilizar en los JSPs**, y nos permiten simplificar tareas en las que comúnmente tendríamos que agregar código Java, permitiendo tener un código más limpio y mantenible.

## 4. Servlets y jsp's en JEE

### Funciones de un JSP

- Un JSP contiene código HTML y puede agregar código Java a través de etiquetas (tags)
- Un JSP se utiliza se debe utilizar como un componente de presentación (Vista) en una arquitectura MVC (Model – View - Controller)



## 4. Servlets y jsp's en JEE

- Ejemplo de Código de un JSP listar.jsp que utiliza una lista de objetos Persona como Modelo, el cual fue compartido por un Servlet:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<html>
    <head>
        <title>Listado Personas</title>
    </head>
    <body>

        <h1>Listado de Personas</h1>

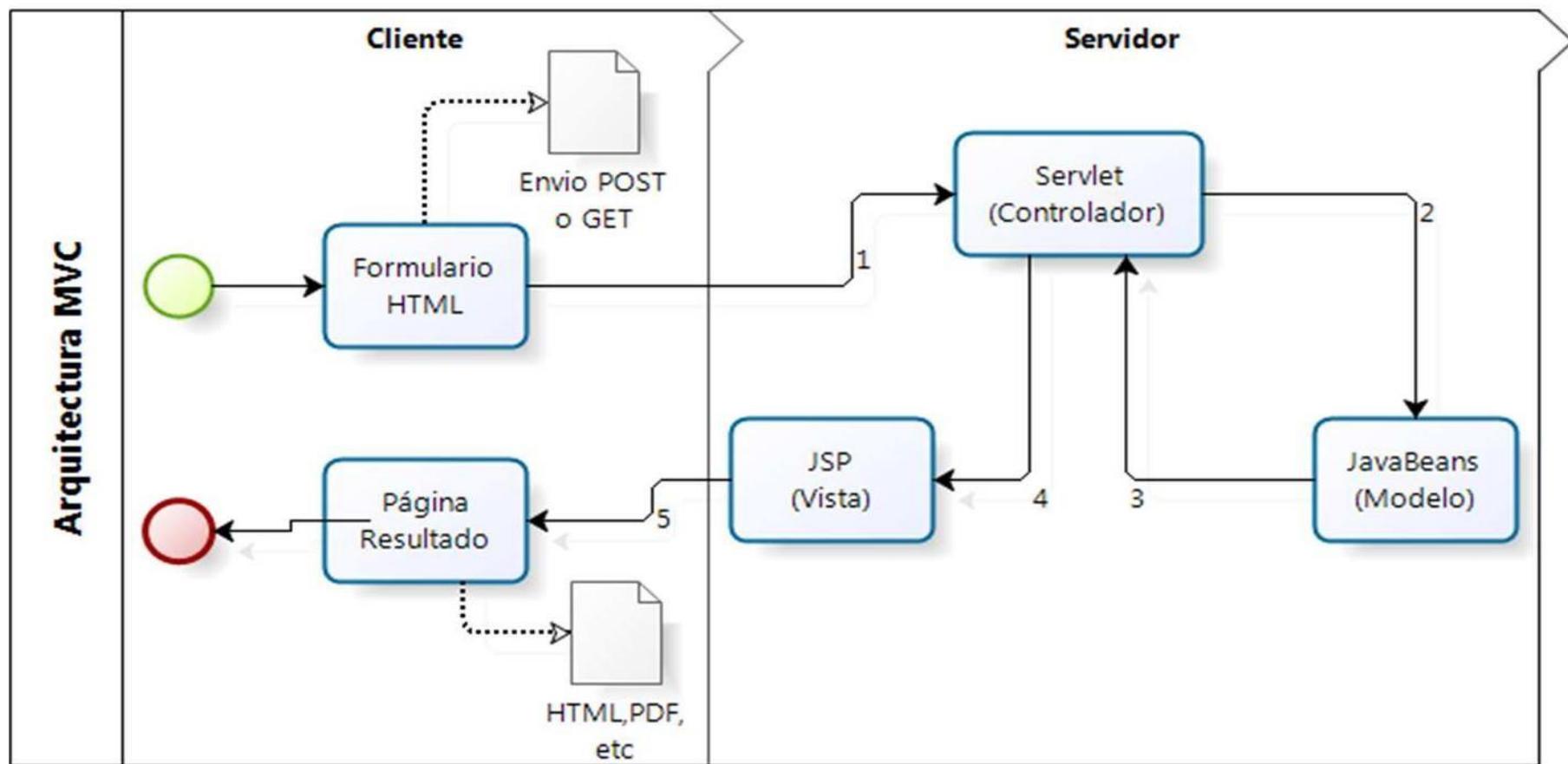
        <table border="1">
            <tr>
                <th>Nombre</th>
                <th>Apellido Paterno</th>
                <th>Email</th>
            </tr>

            <c:forEach var="persona" items="${personas}">
                <tr>
                    <td>${persona.nombre}</td>
                    <td>${persona.apePaterno}</td>
                    <td>${persona.email}</td>
                </tr>
            </c:forEach>
        </table>

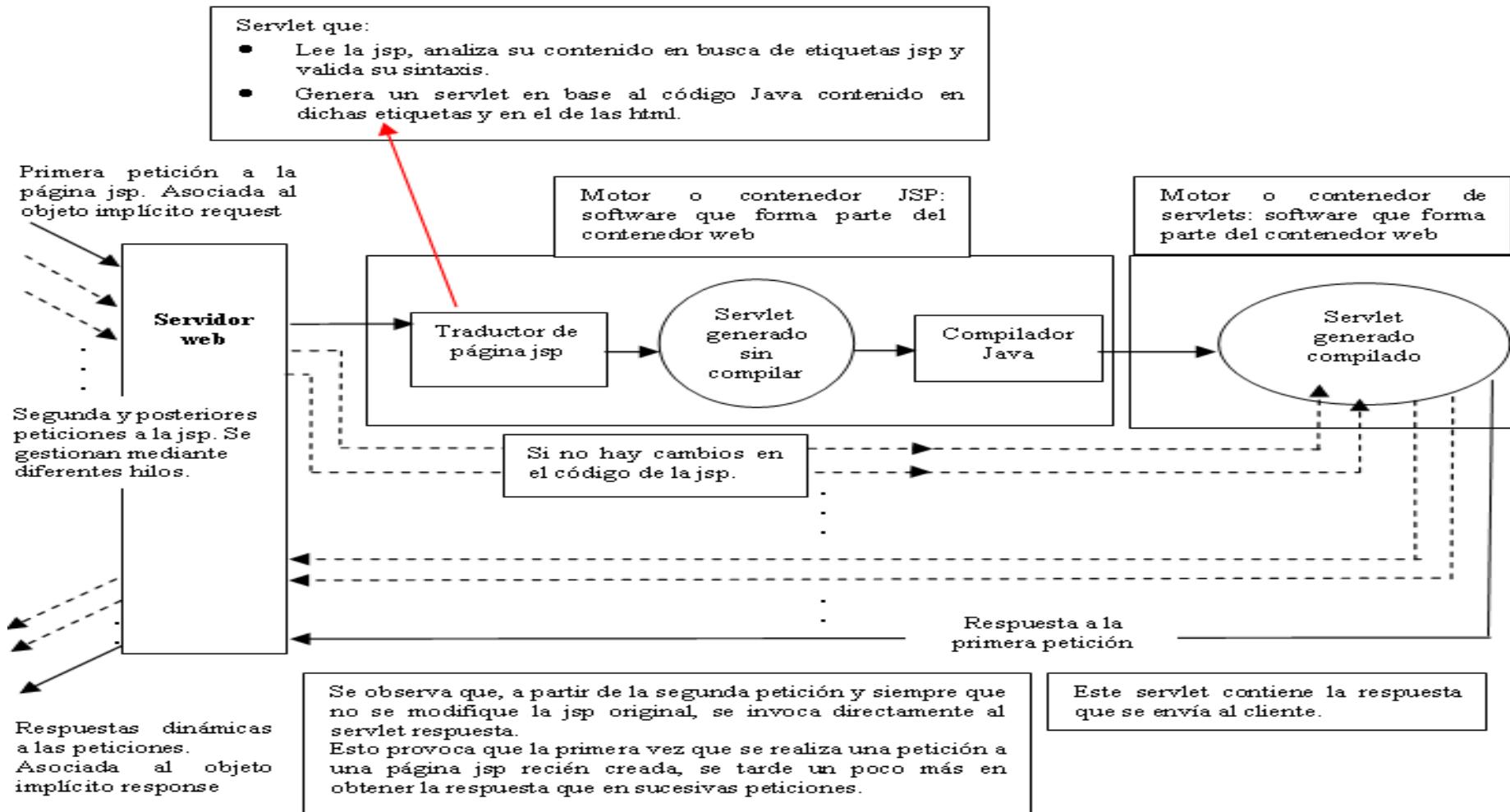
    </body>
</html>
```

# 4. Servlets y jsp's en JEE

## Arquitectura MVC con Servlets y JSPs



# 4. Servlets y jsp's en JEE



## 4. Servlets y jsp's en JEE

- Ejemplo de Código de un Servlet Controlador aplicando el patrón MVC:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    //1. Procesamos los parámetros
    String idPersonaString = request.getParameter("idPersona");
    int idPersona = 0;
    if (idPersonaString != null) {

        //2. Creamos/recuperamos el Modelo
        idPersona = Integer.parseInt(idPersonaString);
        Persona persona = new Persona(idPersona);
        persona = this.personaService.encontrarPersonaPorId(persona);

        //3. Compartimos el Modelo con la Vista
        request.setAttribute("persona", persona);

        //4. Redireccionamos a la Vista que mostrará el Modelo
        request.getRequestDispatcher("editarPersona.jsp").forward(request, response);

    }
}
```

# 4. Servlets y jsp's en JEE

## Nuevas Características Servlets 3.0

### Características el API Servlets 3.0:

- Facilidad de Desarrollo
- Registro Dinámico de Servlets y Filtros
- Compartición de Recursos
- Llamadas Asíncronas
- Mejoras en el API de Seguridad

## 4. Servlets y jsp's en JEE

- El API de los Servlets 3.0 tiene las siguientes mejoras:
- **Facilidad de Desarrollo:**
  - Uso de Anotaciones para la declaración de Servlets, Filtros, Listeners, etc.
  - El archivo web.xml es opcional (si es utilizado sobreescribe a las anotaciones)
  - Soporte del uso de tipos genericos en el API, sin romper compatibilidad
  - Mejor soporte por defecto

# 4. Servlets y jsp's en JEE

- **Uso de Anotaciones:**

- `@WebServlet` – Define un Servlet
- `@WebFilter` – Define un filtro
- `@WebListener` – Define un listener
- `@WebInitParam` – Define un parametro de inicio
- `@MultipartConfig` – Define propiedades para subida de archivos
- `@ServletSecurity` – Define una restricción de seguridad

# 4. Servlets y jsp's en JEE

## Integración entre Servlets y EJB

Ejemplo de un Servlet que accede al Servicio EJB:

```
package web;

import java.io.IOException;

@WebServlet("/ServletControlador")
public class ServletControlador extends HttpServlet {

    @EJB
    private PersonaService personaService;

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) throws ServletException, IOException {
        // Uso de la clase de servicio
        List<Persona> personas = personaService.listarPersonas();
        request.setAttribute("personas", personas);

        request.getRequestDispatcher("listarPersonas.jsp").forward(request,
                                                               response);
    }
}
```

## 4. Servlets y jsp's en JEE

- Al día de hoy, la integración entre las distintas tecnologías empresariales es más simple cada vez.
- Como podemos observar en el siguiente código, integrar un EJB para ser utilizado en un Servlet es tan simple como utilizar la anotación @EJB y especificar el tipo del EJB a utilizar.
- Esto provocará que en automático el servidor de aplicaciones busque una instancia del tipo de EJB especificado, y una vez localizado se realiza una inyección de esta dependencia de manera automática por parte del servidor Java.

## 4. Servlets y jsp's en JEE

```
@WebServlet("/ServletControlador")
public class ServletControlador extends HttpServlet {

    @EJB
    private PersonaService personaService;

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        //Uso de la clase de servicio para recuperar el listado de personas
        List<Persona> personas = personaService.listarPersonas();
        request.setAttribute("personas", personas);

        //Redireccionamos a la vista
        request.getRequestDispatcher("listarPersonas.jsp").forward(request, response);
    }
}
```

# 4. Servlets y jsp's en JEE

## Estructuras JSP de código

- **Directivas:** dentro de las etiquetas <%@ ... %>
  - Controlan la estructura global del servlet respuesta y no generan salida para el cliente.
- La sintaxis de una directiva genérica es la siguiente:  
`<%@ nombreDirectiva atributo1="..." atributo2="..." ... %>`
- Las directivas más usadas son las siguientes:
- **Directiva page:** Empleada para transmitir información de interés de la jsp al contenedor web

# 4. Servlets y jsp's en JEE

Atributo	Contenido	Por defecto
info	Una cadena de texto con información corporativa o de copyright del autor de la jsp. Se captura dentro de un scriptlet mediante el método de la clase javax.servlet.GenericServlet "String getServletInfo()". <b>Suele dejarse el valor por defecto.</b>	Ninguno
language	Lenguaje de script empleado por la jsp. <b>Suele dejarse el valor por defecto.</b>	Java
contentType	Tipo MIME asociado a la respuesta y código de caracteres empleado. <b>Suele dejarse el valor por defecto</b> (contentType="text/html;charset=ISO-8859-1")	text/html y ISO-8859-1
import	Clases o paquetes usados en los scriptlets. Si se necesitan varios se separan con comas. No debe escribirse el punto y coma final característico utilizado en código Java normal. <b>Suele emplearse.</b>	java.lang, javax.servlet, javax.servlet.http y javax.servlet.jsp
extends	Superclase del servlet asociado a la jsp. Suele definirla el contenedor web. <b>Apenas se emplea.</b>	Ninguno
buffer	Tamaño en bytes del buffer de salida. Si no se desea buffer, none. Puede definirse como valor por defecto una cantidad >8 kb. El buffer de salida es como un contenedor donde se almacena la respuesta antes de ser enviada íntegramente al cliente. ¿Quién controla el vaciado del buffer? El contenedor web, vía finalización del procesamiento de la jsp. Hasta que no se haya generado todo el servlet que contiene la respuesta no se vacía. Digamos que la respuesta le llega al cliente de una sola vez. Es obligatorio según la especificación 1.2 de las páginas jsp. <b>Suele dejarse el valor por defecto.</b>	8 kb
session	Valor booleano. Indica si la jsp va a vincularse con una sesión o no. Un valor de true equivale a la línea HttpSession session=request.getSession() en el servlet asociado; un valor false implica no vinculación con ninguna sesión. <b>Suele dejarse el valor por defecto.</b>	true

## 4. Servlets y jsp's en JEE

errorCode	Valor entero que indica el código de error que se ha producido.	
errorPage	URL local que apunta a otra jsp encargada de gestionar cualquier excepción o error producida durante el procesamiento de la jsp en la que se define la directiva. Por ejemplo, cuando la jsp precisa de valores numéricos y se le introduce texto. En el momento de su procesamiento se lanzaría una NumberFormatException. Pues la gestión de esta excepción se va a realizar en la jsp a la que apunta el url. Si no se define este atributo y se produce la excepción, es el contenedor web el encargado de mostrar los mensajes de error. <b>Ver Parametros1.jsp</b> de la aplicación Bean1 (se encuentra en la sección de javabeans) <b>Suele emplearse.</b>	Ninguno
isErrorPage	Valor booleano. Un valor de true indica que la página es una página de gestión de errores. El contenedor web le pasa el objeto implícito <b>exception</b> con el fin de poder obtener información mediante los métodos adecuados (getMessage(), toString(), etc.) de la excepción producida en la jsp que define el atributo errorPage. <b>Ver Errores.jsp</b> de la aplicación Bean1. <b>Suele emplearse.</b>	false
autoFlush	Valor booleano. Un valor de true indica que el buffer de salida se vaciará cuando se llegue a su capacidad máxima o acabe el procesamiento de la jsp. Un valor false indica que se lanzará una excepción cuando ocurra lo anterior <b>Suele dejarse el valor por defecto.</b>	true
isThreadSafe	Valor booleano. Si es false, indica al contenedor web que el servlet asociado a la jsp va a implementar la interface SingleThreadModel con el fin de que sólo pueda gestionar una petición al mismo tiempo y evitar accesos concurrentes no deseados a sus variables de instancia. Hasta que no se procese una petición no se admitirá ninguna otra. <b>Suele dejarse el valor por defecto ya que casi siempre se desea que la jsp gestione múltiples peticiones simultáneamente.</b>	true

# 4. Servlets y jsp's en JEE

- los atributos principales y más habitualmente usados son:
  - import**. Permite importar clases Java en una jsp.
  - session**. Declara que una jsp se vincula a una sesión.
  - errorCode**. Declara que una jsp usa una página de gestión de errores.
  - isErrorHandler**. Declara que una jsp es una página de gestión de errores.

```
1 <%@ page info="Copyright por Jesús Fernández" language="java" contentType="text/html" %>
2 <%@ page import="java.util.*,java.text.*" %>
3 <html>
4 <head><title>Segunda página jsp</title></head>
5 <body>
6 <%
7     Date hoy=new Date();
8     DateFormat df=DateFormat.getDateInstance(DateFormat.MEDIUM,DateFormat.MEDIUM,
9         Locale.getDefault());
10 %>
11 Fecha en castellano y con formato medio: <%=df.format(hoy)%><p>
12 <hr>
13 <%
14     String alumno=request.getParameter("nombre");
15     if(alumno==null)
16         alumno="mundo";
17 %>
18 HOLA <font color="red"><b><%=alumno%></b></font>, cómo te va la vida?<p>
19 <hr>
20 <%
21     out.println(getServletInfo());
22 %>
23 </body>
24 </html>
```

# 4. Servlets y jsp's en JEE

## • Directiva include

- Permite la inclusión estática del contenido de otras jsps, páginas html, ficheros de texto no formateado, ficheros Java, ficheros XML, etc. No admite inclusión de servlets.
- Suele emplearse **para incluir encabezados (headers) o pies de página (footers)** que contendrán todas las jsps de una aplicación web.

```
<%@ include file="urlLocal" %>
```

- Este url puede definirse mediante:
  - Ruta relativa al directorio raíz de la aplicación.
  - Ruta relativa a la jsp donde se encuentra la directiva.
- No hay limitación en el número de directivas include que pueden agregarse y sí es importante el orden en que aparecen.

# 4. Servlets y jsp's en JEE

```
1 <html>
2 <head><title>Estudio de la directiva include ... </title></head>
3 <body>
4 <h1>Ejemplo de uso de la directiva include</h1>
5
6 <font color="red"><b>
7
8 <!-- Empleando ruta relativa al directorio raíz jsp-examples de la
9 aplicación web de ejemplo de Tomcat -->
10 <%@ include file="/Fecha.jsp"%><p>
11
12 <!-- Empleando ruta relativa al directorio donde se encuentra la jsp -->
13 <%--<%@ include file="Fecha.jsp"%><p>--%>
14 </b></font>
15
16 <font color="blue"><b>
17
18 <%@ include file="/Copyright.html"%><p>
19
20 <%--<%@ include file="Copyright.html"%><p>--%>
21 </b></font>
22 <b><i>Cucurrucucu Paloma</i></b>
23 </body>
24 </html>
```

```
1 <%@ page import="java.util.* , java.text.*" %>
2 <% DateFormat df=DateFormat.getDateInstance(DateFormat.MEDIUM,DateFormat.MEDIUM,
3   Locale.getDefault()); %>
4 <b>Fecha de hora:</b> <%=df.format(new Date())%>
```

## 4. Servlets y jsp's en JEE

- **Elementos de scripting:**

- Permiten la inserción de código Java dentro de la página jsp.
- Todos los elementos de scripting tienen acceso a una serie de objetos implícitos suministrados por el contenedor jsp que permiten aumentar la funcionalidad de las páginas jsp

- Los elementos de scripting son los siguientes:

- Declaraciones: dentro de las etiquetas <%! código Java %>
- Expresiones: dentro de las etiquetas <%= código Java %>
- Scriptlets: dentro de las etiquetas <% código Java %>
- Comentarios: se tienen los siguientes tipos:
- De HTML: <!-- comentario -->

## 4. Servlets y jsp's en JEE

- De JSP: <%-- comentario --%>
- Del lenguaje de script Java: <%// comentario línea %> y <%/\* comentario varias líneas \*/%>
- Acciones: Permiten trabajar con componentes complementarios a la página jsp como applets, otras páginas jsp, javabeans, etc.
  - No asociadas a los javabeans:
    - <jsp:include> ... </jsp:include> o <jsp:include ... /> si sólo tiene atributos
    - <jsp:plugin> ... </jsp:plugin>
    - <jsp:forward> ... </jsp:forward> o <jsp:forward ... /> si sólo tiene atributos
  - Asociadas a los javabeans:
    - <jsp:useBean ... /> si sólo tiene atributos
    - o <jsp:useBean ...> ... </jsp:useBean>
    - <jsp:setProperty ... />
    - <jsp:getProperty ... />

## 4. Servlets y jsp's en JEE

### JSTL (JSP Standard Tag Library)

- **JSTL** es una **colección de funciones de uso común**, cuando se desarrolla páginas dinámicas (como ser bucles, bloques condicionales, formateo de textos, internacionalización, etc), rápidas para ser usadas desde páginas JSP, y con una apariencia similar a etiquetas de xml
  - ej: <c:out value="se muestra este texto en la página" />
- El uso de estas funcionalidades (JSTL) permiten que las **páginas sean más fáciles de leer y de mantener**, evitando agregar gran cantidad de código java insertado en las páginas

## 4. Servlets y jsp's en JEE

### JSTL (JSP Standard Tag Library)

- Para poder usar una tag lib en una página, es necesario importarla usando el identificador asociado al tag (la uri ), y dar un nombre de prefijo, el cual es usado para llamar luego a las funciones disponibles.
- Para esto, en la página usamos:
  - <%@ taglib prefix="c" uri="<http://java.sun.com/jsp/jstl/core>" %>
  - <%@ taglib prefix="fmt" uri="<http://java.sun.com/jsp/jstl/fmt>" %>
  - <%@ taglib prefix="x" uri="<http://java.sun.com/jsp/jstl/xml>" %>
  - <%@ taglib prefix="sql" uri="<http://java.sun.com/jsp/jstl/sql>" %>
  - <%@ taglib prefix="fn" uri="<http://java.sun.com/jsp/jstl/functions>" %>

## 4. Servlets y jsp's en JEE

### LENGUAJE de EXPRESION (EL)

- EL (Expression Language) es un lenguaje utilizado en las paginas jsp para interactuar con los datos servidos por parte del servidor, combinado con la librería **JSTL Core** nos permite construir toda la lógica de las jsp de una forma mucho mas amena y eficaz.
- Para obtener el valor de cualquier variable, sea del tipo que sea, lo único que tenemos que hacer es escribir su nombre entre \${}, de tal modo, si nuestra variable “**miVariable**” es un String, Integer, Date()... bastara con escribir <c:out value=”\${miVariable}”/> para acceder a ella.

## 4. Servlets y jsp's en JEE

### LENGUAJE de EXPRESION (EL)

- EL (Expression Language) es un lenguaje utilizado en las paginas jsp para interactuar con los datos servidos por parte del servidor, combinado con la librería **JSTL Core** nos permite construir toda la lógica de las jsp de una forma mucho mas amena y eficaz.
- Para obtener el valor de cualquier variable, sea del tipo que sea, lo único que tenemos que hacer es escribir su nombre entre \${}, de tal modo, si nuestra variable “**miVariable**” es un String, Integer, Date()... bastara con escribir <c:out value=”\${miVariable}”/> para acceder a ella.

## 4. Servlets y jsp's en JEE

- Del mismo modo, para acceder a un javabean con sus propiedades, getters y setters, EL automáticamente trata de obtener la propiedad o insertarla utilizando para ello los estándares de `getNombre` y `setNombre` de un javabean clásico.

```
public class Persona{  
    private String nombre;  
    private int edad;  
  
    public void setNombre(String nombre){  
        this.nombre = nombre;  
    }  
  
    public String getNombre(){  
        return this.nombre;  
    }  
  
    public void setEdad(int edad){  
        this.edad = edad;  
    }  
  
    public int getEdad(){  
        return this.edad;  
    }  
}
```

Todo lo que necesitamos hacer para acceder a sus propiedades con **EL** desde el **JSP** es escribir el nombre del objeto, seguido de punto, y el nombre de la propiedad. `<c:out value="${persona.nombre}">` y `<c:out value="${persona.edad}">`.

## 4. Servlets y jsp's en JEE

- Accediendo a listas con la propiedad `forEach` de JSTL Core.
- Imaginemos una list formada por objetos Persona.

*List<Persona> personas;*

- La forma de acceder a cada uno de los objetos de la list `personas` es iterando en la lista a través del **forEach**, y accediendo dentro del bucle desde el nombre que hallamos dado a cada objeto iterado como si fuera un simple javabean.

```
<c:forEach items="${personas}" var="persona">
    ...<c:out value="${persona.nombre}"/>...
    ...<c:out value="${persona.edad}"/>...
</c:forEach>
```

# Índice

- 
1. Introducción Java Empresarial
  2. Enterprise Java Beans (EJBs) y Context Directory Inyection (CDI)
  3. Java Persistence Api (JPA)
  4. Servlets y jsps en Java EE
  - 5. JavaServer Faces**
  6. Web Services (SOAP y REST)
  7. Seguridad en JEE

# 5. JavaServer Faces

## ¿Qué es JavaServer Faces?

- El objetivo de la tecnología JavaServer Faces es desarrollar aplicaciones web de forma parecida a como se construyen aplicaciones locales con Java Swing, AWT (*Abstract Window Toolkit*), SWT (*Standard Widget Toolkit*) o cualquier otra API similar.
- **JavaServer Faces** pretende facilitar la construcción de estas aplicaciones proporcionando un entorno de trabajo *framework* vía web que gestiona las acciones producidas por el usuario en su página HTML y las traduce a eventos que son enviados al servidor con el objetivo de regenerar la página original y reflejar los cambios pertinentes provocados por dichas acciones.

# 5. JavaServer Faces

## ¿Qué es JavaServer Faces?

- JSF es un framework para crear aplicaciones java J2EE basadas en el patron MVC.
- JSF utiliza páginas JSP para generar las vistas, añadiendo una biblioteca de etiquetas propia para crear los elementos de los formularios HTML.
- Asocia a cada vista con formularios un conjunto de objetos java manejados por el controlador (**managed beans**)
- Introduce una serie de etapas en el procesamiento de la petición, como por ejemplo la de validación, reconstrucción de la vista, recuperación de los valores de los elementos, etc.

# 5. JavaServer Faces

## ¿Qué es JavaServer Faces?

- JavaServer Faces (JSF) es el **marco de aplicaciones web estándar** para Java Enterprise Edition (Java EE).
- Al ser un **estándar** de Java, la tecnología cuenta con el **apoyo** de una industria muy sólida y pueden encontrarse **implementaciones de distintos fabricantes**.
- La tecnología ha crecido en su uso a nivel mundial.
- Se cuenta con un fuerte apoyo de IDEs de Java, así como Servidores de Aplicaciones para su despliegue.
- El número de empresas que extienden la funcionalidad de JSF es muy amplia y muchos proyectos son OpenSource.

# 5. JavaServer Faces

## Características de JSF

- **MVC:** Implementa el patrón de diseño **Modelo-Vista-Controlador**, proporcionando un enfoque orientado a eventos.
- **RAD (rapid application development):** **Desarrollo rápido** de aplicaciones para Web debido al **numero de componentes listos para usarse**.
- **Componentes de interfaz de usuario:** JSF tiene desarrollados componentes reutilizables listos para utilizarse.
- **Render-Kits:** Los componentes pueden desplegarse no solamente en navegadores Web, sino en **dispositivos móviles u otros tipos de clientes**.

# 5. JavaServer Faces

## Características de JSF

- **Extensibilidad:** JSF permite **crear más fácilmente nuevos componentes**, por lo que existen varios frameworks que extienden el poder de JSF y Ajax, tales como richFaces, iceFaces, entre otros.
- **Internacionalización:** Las vistas pueden mostrarse en distintos idiomas.

## 5. JavaServer Faces

La versión JSF 2.0 generó una mejora muy importante para este framework Java. De hecho al día de hoy es el estándar de presentación en una arquitectura JEE.

Algunas de las mejoras son:

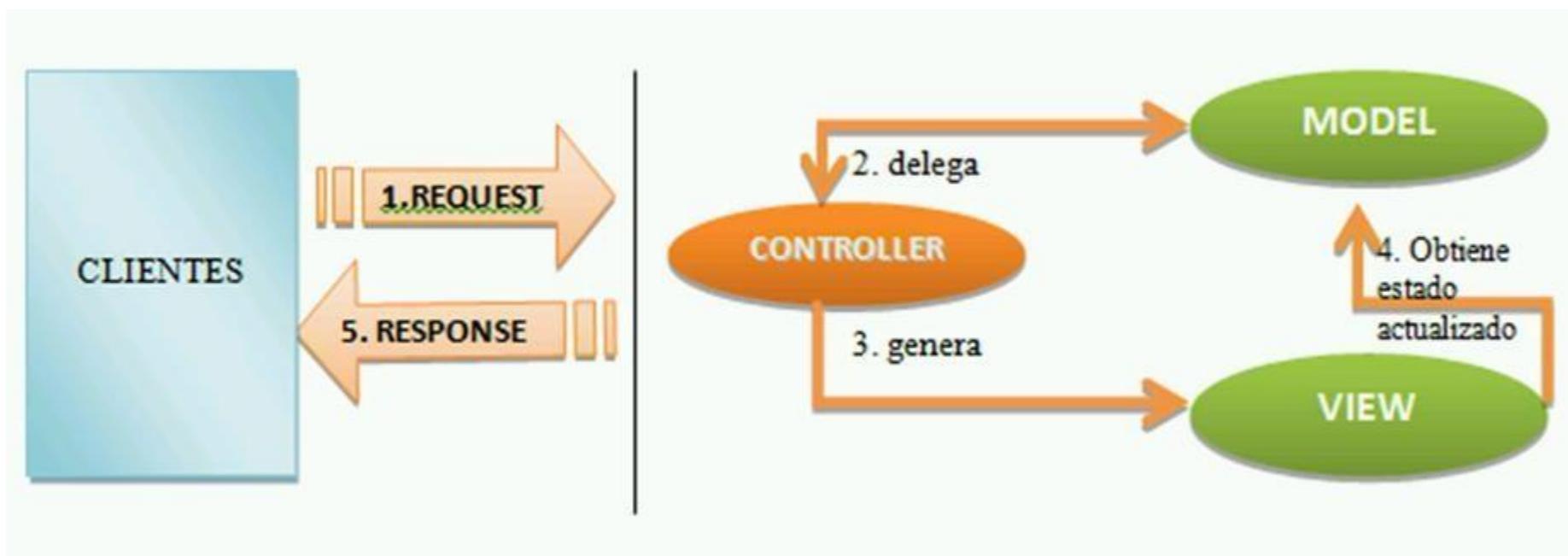
- **Manejo de condiciones por defecto más inteligentes:** Esto aplica en **casos de navegación simples**, los cuales ya **no** se requiere agregarlos al archivo de configuración **faces-config.xml**
- **Manejo de anotaciones para varias configuraciones:** Esto simplifica en gran medida agregar un **Managed Bean** a nuestra aplicación, **evitando** su declaración en el archivo **faces-config.xml**.

## 5. JavaServer Faces

- **Soporte nativo para AJAX:** La tecnología AJAX, *Asynchronous JavaScript And XML* ([JavaScript](#) asíncrono y [XML](#)) ya es parte del ciclo de vida de JSF. Procesar peticiones asíncronas es mas rápido.
- **Soporte por defecto para Facelets:** La tecnología de los Facelets (sistema de código abierto de plantillas [web](#)) toma en cuenta el ciclo de vida de JSF, a diferencia de los JSP"s.
- **Más componentes y validadores:** Se han creado y agregado nuevos componentes a la librería estándar de JSF.

## 5. JavaServer Faces

- Patrón diseño MVC en JSF



## 5. JavaServer Faces

JSF es un framework de aplicaciones web que implementa el patrón de diseño MVC, con una clara separación de responsabilidades:

- **El Modelo:** Contiene datos de la interfaz de usuario y es el responsable de almacenar los datos de la aplicación Web. Se puede implementar con clases puras de Java (POJO: Plain Old Java Object) o con Managed Bean de Modelo (**no contienen lógica de la aplicación ni administran el flujo de la misma**).
- **La Vista:** Define la interfaz de usuario con una jerarquía de componentes, utilizando la librería estándar de JSF, Expression Language (EL), JSTL, entre otras tecnologías para facilitar el despliegue de la información del Modelo. La tecnología utilizada por default en JSF 2.0 son los Facelets.
- **El Controlador:** Maneja las interacciones del usuario y la navegación o flujo de la aplicación. Se implementa con Managed Beans.

# 5. JavaServer Faces

## ¿Qué es una aplicación JavaServer Faces?

- En su mayoría, las aplicaciones JavaServer Faces son como cualquier otra aplicación web Java. **Se ejecutan en un contenedor de servlets de Java** y, típicamente, contienen:
  - Componentes *JavaBeans* (llamados objetos del modelo en tecnología JavaServer Faces) conteniendo datos y funcionalidades específicas de la aplicación.
  - Oyentes de Eventos.
  - Páginas, (principalmente páginas JSP).
  - Clases de utilidad del lado del servidor, como *beans* para acceder a las bases de datos.

## 5. JavaServer Faces

- Además de estos ítems, una aplicación JavaServer Faces también tiene:
  - Una librería de etiquetas personalizadas para dibujar componentes UI en una página.
  - Una librería de etiquetas personalizadas para representar manejadores de eventos, validadores y otras acciones.
  - Componentes UI representados como objetos con estado en el servidor.

# 5. JavaServer Faces

- Configuración básica
  - a. **Web Deployment Descriptor**: web.xml  
Declarar FacesServlet
  - b. **Archivos de Configuración**: faces-config.xml  
Permite configurar todos los elementos usados en una aplicación JSF.
  - c. **Anotaciones**: permiten configurar
    - **Managed Beans**: @ManagedBean
    - **Validators**: @FacesValidator
    - **Converters**: @FacesConverter
    - **Componentes IU**: @FacesComponent
    - **Renderers**: @FacesRenderer
    - **Behavior**: @FacesBehavior

Para todas las demás configuraciones se debe usar el archivo faces-config.xml.

# 5. JavaServer Faces

## Las etiquetas JSF

- JSf dispone de un conjunto básico de etiquetas que permiten crear fácilmente componentes dinámicos en las páginas web.

Estas etiquetas son:

- h:commandButton. Un botón al que podemos asociar una acción.
- h:commandLink. Un enlace hipertexto al que podemos asociar una acción.
- h:dataTable. Crea una tabla de datos dinámica con los elementos de una propiedad de tipo Array o Map del bean.
- h:form. Define el formulario JSF en la página JSP-
- h:graphicImage. Muestra una imagen jpg o similar.
- h:inputHidden. Incluye un campo oculto del formulario.

## 5. JavaServer Faces

- h:inputSecret . Incluye un campo editable de tipo contraseña (no muestra lo que se escribe)
- h:inputText. Incluye un campo de texto normal.
- h:inputTextarea. Incluye un campo de texto multilínea.
- h:message. Imprime un mensaje de error en la página (si se ha producido alguno).
- h:messages. Imprime varios mensajes de error en la página, si se han producido.
- h:outputFormat. Muestra texto parametrizado. Utiliza la clase java.text.MessageFormat de formateo.
- h:outputLabel. Muestra un texto fijo.

## 5. JavaServer Faces

- h:panelGrid. Crea una tabla con los componentes incluidos en el panelGrid.
- h:panelGroup. Agrupa varios componentes para que cierto componente los trate como un único componente (por ejemplo para meter varios componentes en una celda de un panelGrid).
- h:selectBooleanCheckbox. Crea una casilla con dos estados: activado y desactivado.
- h:selectManyCheckbox. Crea un conjunto de casillas activables.

## 5. JavaServer Faces

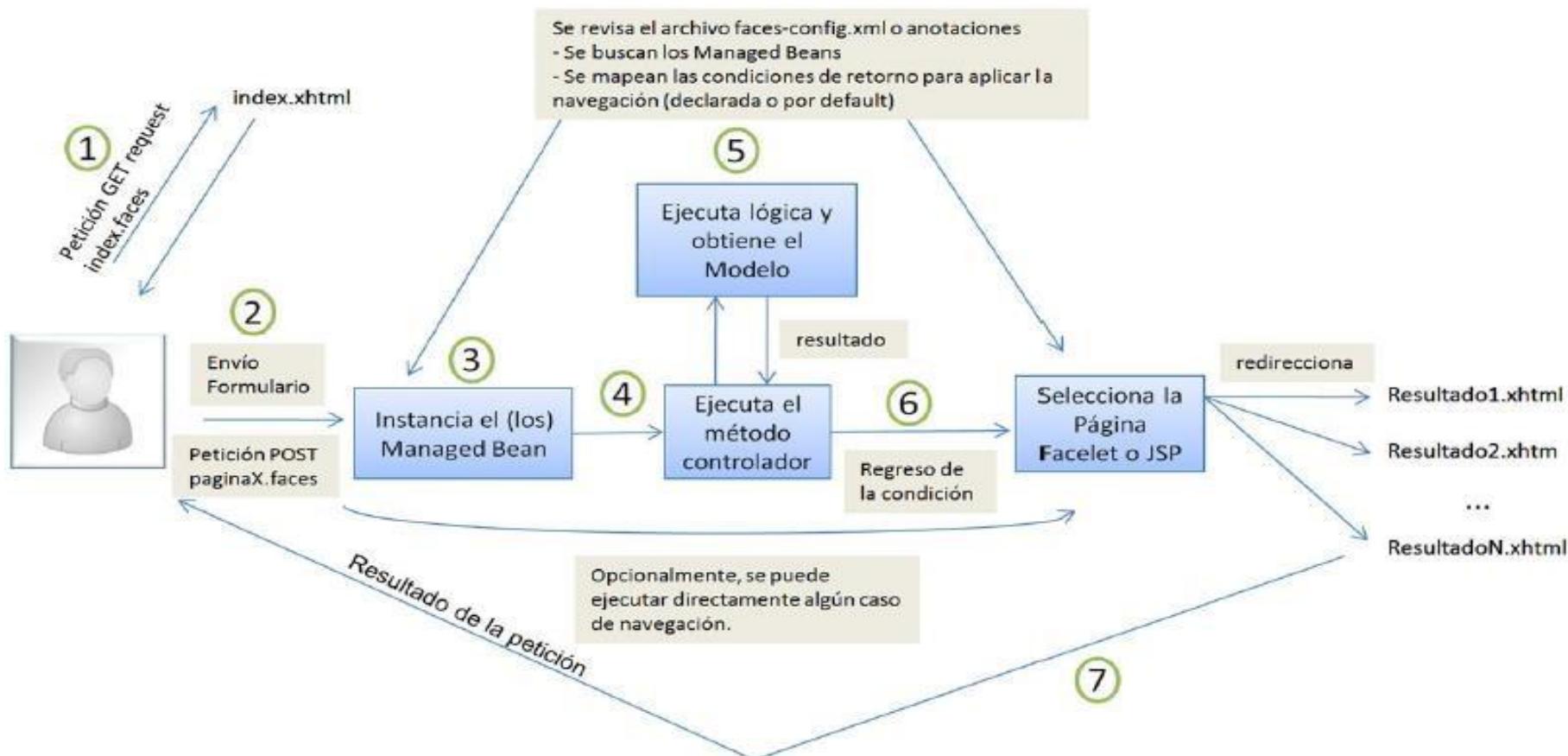
- h:outputLink. Crea un enlace hipertexto.
- h:outputText
- h:selectManyListbox. Crea una lista que permite seleccionar múltiples elementos.
- h:selectManyMenu. Crea una lista desplegable de selección múltiple.
- h:selectOneListbox. Crea una lista en la que se puede seleccionar un único elemento.
- h:selectOneMenu. Crea una lista desplegable de selección.
- h:selectOneRadio. Crea una lista de botones, redondos normalmente, excluyentes.

## 5. JavaServer Faces

- A partir de la versión JSF 2.0. la tecnología de despliegue (Page Description Language – PDL) es por defecto **Facelets**.
- **Los Facelets** tienen la ventaja que **fueron creados tomando en cuenta el ciclo de vida de JSF**, incluyendo las siguientes características:
  - Los Facelets son **documentos XHTML**, por lo que el analizador XML permite que **la depuración de errores sea más sencilla** que en los JSP"s.
  - Al ejecutar un Facelet, **todos los componentes JSF son convertidos a instancias Java** y son administrados por un Component Tree.
  - Es más sencillo **encapsular código** y así crear componentes reutilizables.
  - Permiten crear **plantillas** y así definir más fácilmente nuevas vistas a partir de las plantillas definidas.

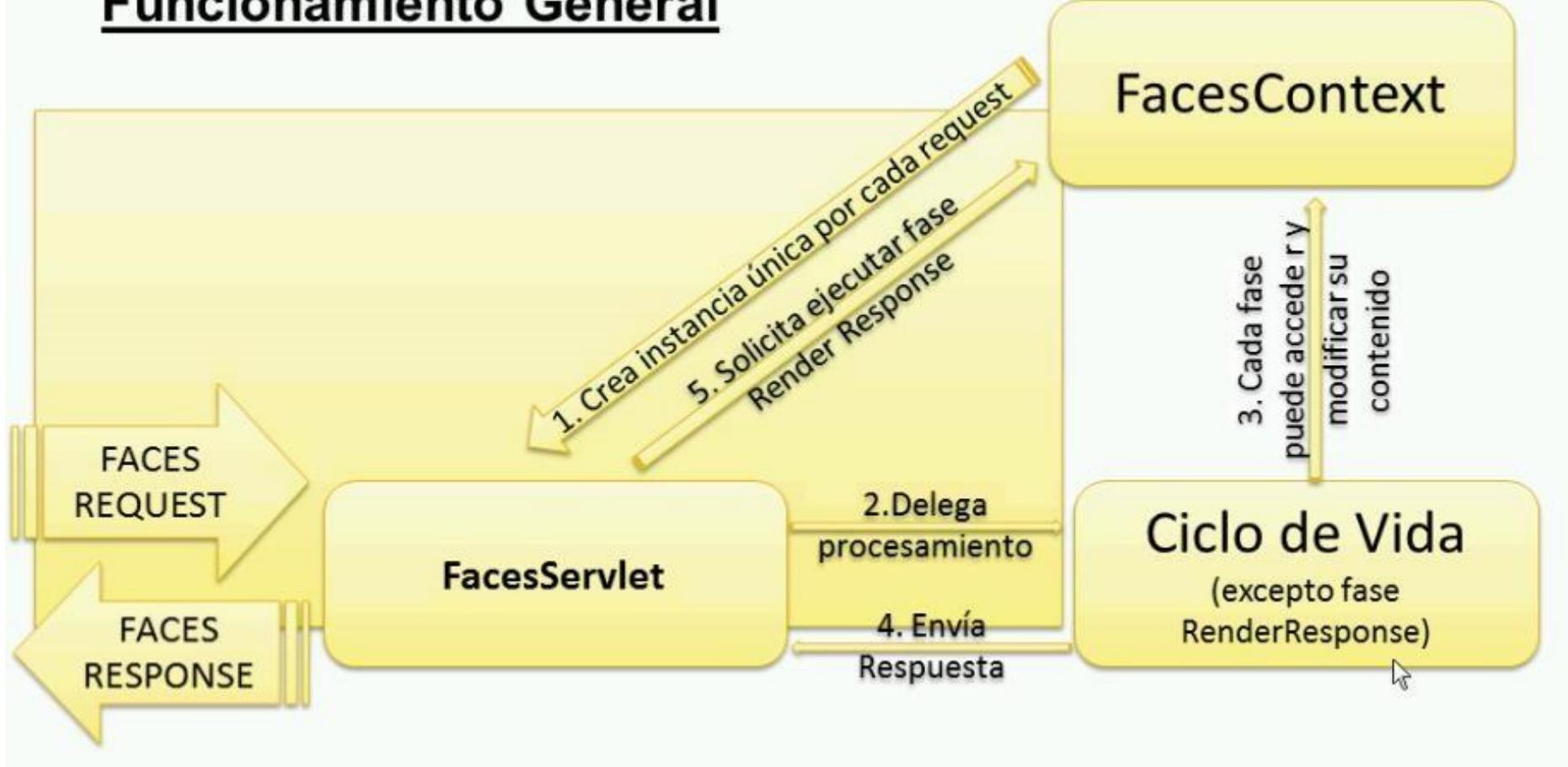
# 5. JavaServer Faces

## Flujo de Navegación en JSF



# 5. JavaServer Faces

## Funcionamiento General



## 5. JavaServer Faces

1. El framework se inicia con la petición GET a una página, por ejemplo, index.xhtml. (En esta etapa el controlador construye en memoria la **estructura de componentes de la página**)
2. Una vez que ya estamos en el contexto JSF, el usuario recibe el contenido de respuesta y envía nuevamente una petición POST al servidor Web
3. El servidor Web recibe la petición y revisa los Managed Beans involucrados en la petición, si es necesario crea una instancia de ellos, dependiendo de su alcance y llama a los métodos setters de las propiedades del bean a llenar. (En esta etapa se recuperan los valores de la request y **se asignan a los beans de la página**).

## 5. JavaServer Faces

4. Se ejecuta el método que procesa la petición, para este momento ya están instanciados los otros Managed Beans involucrados, si es que los hubiera.
5. Se ejecuta la lógica de negocio, con el objetivo de obtener el Modelo.
6. Se selecciona el caso de navegación y se redirecciona a la vista correspondiente.
7. La vista utiliza la información de Modelo para finalmente regresar la respuesta solicitada al cliente.

# 5. JavaServer Faces

## Concepto de Managed Beans

Un Managed Bean es una clase Java que sigue la nomenclatura de los JavaBeans

- Los Managed Beans no están obligados a extender de ninguna otra clase

Aunque JSF no define una clasificación para los Backing Beans, podemos definir las siguientes:

- Beans de Modelo: Representan el Modelo en el patrón MVC
- Beans de Control: Representan el Controlador en el patrón MVC
- Beans de Soporte o Helpers: Contienen código por ejemplo de Convertidores
- Beans de Utilerías: Tareas genéricas, como obtener el objeto *request*

## 5. JavaServer Faces

- A las clases java que se asocian a los formularios JSF se les denomina **backend beans** ya que son los beans (clases java) que están detrás del formulario.
- Estos beans se referencian en el fichero de configuración de JSF o bien mediante anotaciones
- Una JavaBean es una clase Java que sigue las siguientes convenciones:
  - Constructor vacío
  - Atributos de clase privados
  - Por cada atributo, se crean los métodos getters y setters
- El Objetivo de los Managed Beans es controlar el estado de las páginas web.

## 5. JavaServer Faces

- JSF administra automáticamente los **Managed Beans**:
  - Crea las instancias
    - Por ello la necesidad de un constructor vacío.
  - Controla su ciclo de vida
    - JSF determina el ámbito o alcance (request, session, application, etc) de cada Managed Bean
  - Llama los métodos getters o setters
    - Por ejemplo: <h:inputText value="#{empleado.apellidoPaterno}" al hacer submit llamará el método setApellidoPaterno( )
    - Otro ejemplo: #{empleado.nombre} indirectamente llamará al método getNombre( )
- Los Managed Beans los podemos declarar de varias formas, ya sea utilizando anotaciones o en el archivo **faces-config.xml**.

# 5. JavaServer Faces

## Declaración

- faces-config.xml

```
<managed-bean>
    <managed-bean-name>clienteBean</managed-bean-name>
    <managed-bean-class>pe.edu.cibertec.managed.ClienteBean</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

- Anotación

```
@ManagedBean(name="clienteBean")
@RequestScoped
public class ClienteBean implements Serializable
```

Ambas configuraciones son equivalentes.



# 5. JavaServer Faces

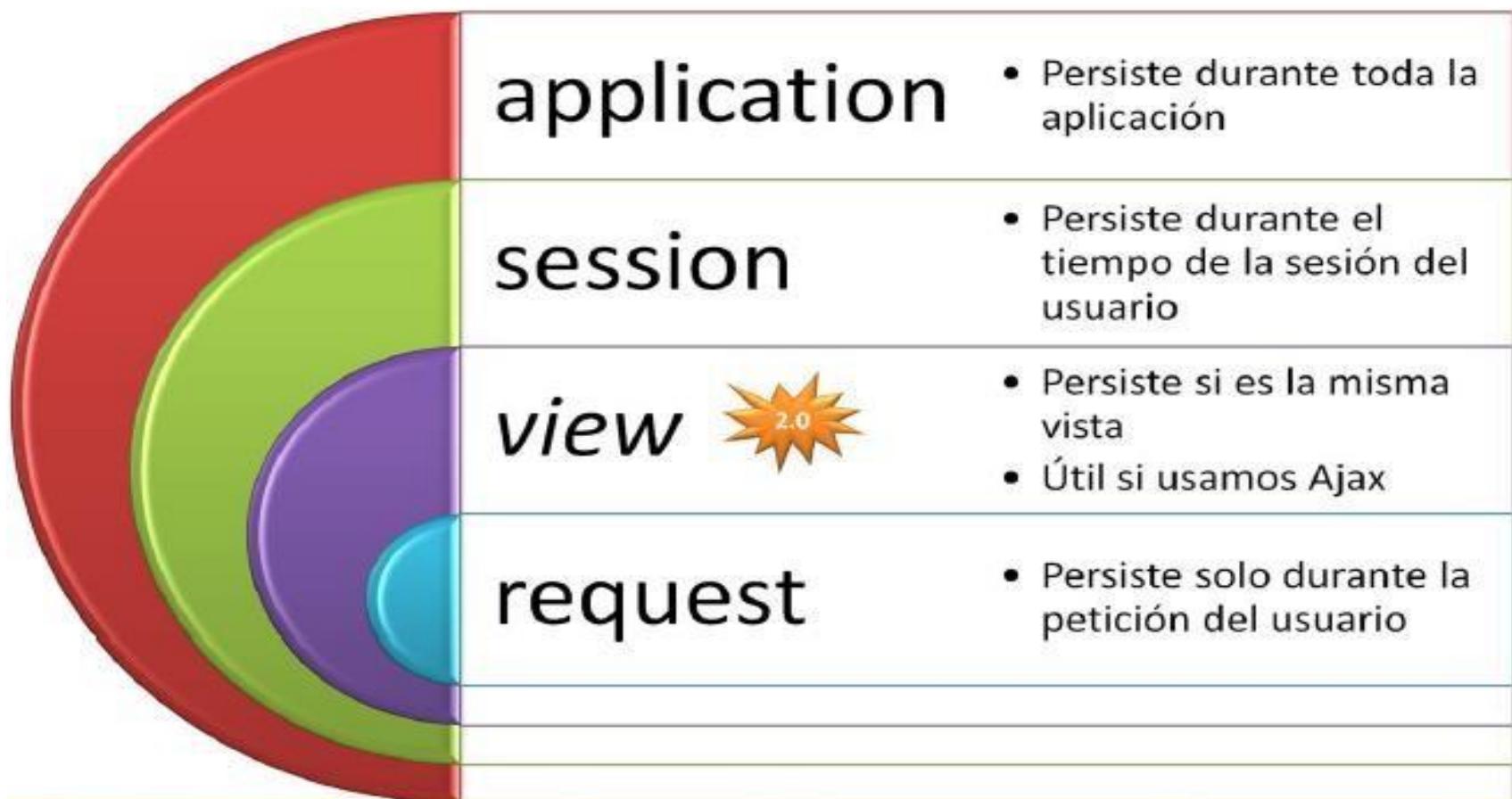
## Uso de los Managed Beans

Un Managed Bean se puede declarar de distintas maneras:

- Con anotaciones, antes del nombre de la clase:
  - `@ManagedBean`
- Como bean CDI (Contexts and Dependency Injection), antes del nombre de la clase:
  - `@Named`
  - Agregar un archivo llamado `beans.xml` en la carpeta:  
`web/WEB-INF`
- En el archivo `faces-config.xml`:
  - `<managed-bean> ... </managed-bean>`

# 5. JavaServer Faces

## Alcance de los Managed Beans



## 5. JavaServer Faces

- Las anotaciones para el manejo de alcances JSF son:
  - @ RequestScoped (ámbito por defecto)
  - @ ViewScoped
  - @ SessionScoped
  - @ ApplicationScoped
- Las anotaciones de alcance se utilizan después de la anotación del Bean, por ejemplo:
  - @ManagedBean @SessionScoped

# 5. JavaServer Faces

## Ejemplo de uso de Managed Beans

### 1. Creación del Bean

```
@ManagedBean  
@SessionScoped  
public class EmpleadoForm{...}
```

Uso en la página JSF con EL (Expression Language):

```
#{{ empleadoForm.atributo }}
```

### 2. Creación del Bean con nombre personalizado y notación CDI

```
@Named  
public class EmpleadoForm{...}
```

Uso en la página JSF con EL:

```
#{{ empleadoForm.atributo }}
```

# 5. JavaServer Faces

## Inyección de Beans

JSF soporta inyección de dependencias simple

La inyección se puede hacer de 2 maneras:

- Con anotaciones dentro del Managed Bean:

```
@ManagedProperty(value= "#{nombreBean}")
```

- En el archivo faces-config.xml utilizando la siguiente etiqueta dentro de un managed bean:

```
<managed-property>"#{nombreBean}"</managed-property>
```

## 5. JavaServer Faces

- En muchas ocasiones una página JSF puede **utilizar varios Managed Beans**, de ahí la necesidad de relacionar distintos Managed Beans.
- La inyección de dependencias (Dependency Injection – DI) es un derivado de IoC. Uno de los iniciadores de este principio fue Spring framework y al día de hoy es una práctica muy utilizada en arquitecturas JEE.
- Podemos lograr la inyección de dependencias de 2 maneras:
  - 1) **Con anotaciones**: Manejando la anotación @ManagedProperty en la propiedad del Managed Bean a injectar.
  - 2) **Con el archivo faces-config.xml**: Agregando la etiqueta <managedproperty> dentro del Managed Bean declarado.

# 5. JavaServer Faces

## Archivo faces-config.xml

Uso del archivo faces-config.xml:

- Configuración Centralizada
- Reglas de navegación
- Declaración de Managed Beans
- Inyección de Dependencias
- Definir configuraciones regionales
- Registro de validadores
- Registro de listeners
- Entre otros puntos...
  
- Ubicación del archivo de configuración:
  - WEB-INF/faces-config.xml

## 5. JavaServer Faces

- El archivo **faces-config.xml** nos permite agregar la configuración de JSF de manera centralizada.
- El formato general del archivo JSF depende de la versión a utilizar, en este caso es un ejemplo de la versión JSF 2.0.

```
<?xml version="1.0"?>
<faces-config xmlns="..." xmlns:xsi="..." xsi:schemaLocation="..."
version="2.0">

...
</faces-config>
```

- Con el uso de anotaciones, cada vez es menos indispensable utilizar este archivo, aunque todavía suele utilizarse para tareas como el manejo regional (varios lenguajes en la aplicación web) entre otras tareas.

# 5. JavaServer Faces

## Ejemplo de declaración de Beans

```
<?xml version="1.0"?>
<faces-config ...>
    <managed-bean>
        <managed-bean-name>empleadoBean</managed-bean-name>
        <managed-bean-class>beans.EmpleadoBean</managed-bean-class>
    </managed-bean>

    <managed-bean>
        <managed-bean-name>empleadoForm</managed-bean-name>
        <managed-bean-class>forms.EmpleadoForm</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
        <managed-property>"#{empleadoBean}"</managed-property>
    </managed-bean>
</faces-config>
```

## 5. JavaServer Faces

- Podemos observar un ejemplo de configuración de Managed Beans utilizando el archivo faces-config.xml.
- El primer bean declarado, empleadoBean, se encuentra en el paquete beans. Al no definir un scope, se crea **por defecto en el alcance de request**.
- El segundo bean declarado (empleadoForm) se encuentra en el paquete forms. Dicho bean tiene una dependencia con empleadoBean, por lo que se manda a llamar el método setEmpleadoBean de la clase EmpleadoForm para injectar la dependencia.
- Esta forma de configurar beans es **más extensa** que al utilizar anotaciones, pero **permite tener centralizada la configuración en un solo archivo**.

# 5. JavaServer Faces

## Expression Language (EL)

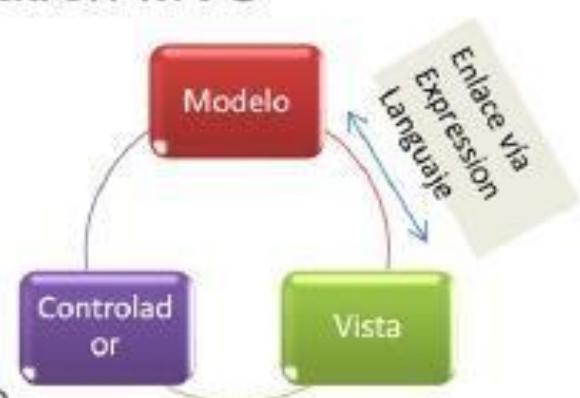
El Lenguaje de Expresión (EL) de JSF nos permite simplificar el manejo de expresiones en las páginas JSF

El lenguaje EL enlaza la Vista con el Modelo del patrón MVC

```
#{{nombreBean.propiedad}}
```

¿Qué sucede al evaluar una expresión EL?

- Se busca el bean a utilizar en cierto alcance
- Se manda a llamar el método get o set de la propiedad indicada



## 5. JavaServer Faces

- El lenguaje de Expresión (EL) nos permite evaluar y crear más fácilmente expresiones en nuestras páginas JSF.
- En JSP's también existe un lenguaje de expresión, pero tiene una diferencia en su sintaxis, por ejemplo:
  - En JSP: \${nombreBean.propiedad}
  - En JSF: #{nombreBean.propiedad}
- EL permite enlazar la Vista con el Modelo, por lo que desde una página JSF podemos acceder fácilmente a los Managed Beans de JSF.
- Cuando una EL encuentra un bean a utilizar, lo busca en cierto alcance. El orden de búsqueda es: request, view, session, application

## 5. JavaServer Faces

- Una vez que se ha ubicado el Bean a utilizar, se manda a llamar el método get o set de la propiedad indicada, por ejemplo:
  - nombreBean.getPropiedad();

# 5. JavaServer Faces

## Objetos implícitos en EL

El Lenguaje EL también nos permite acceder fácilmente a objetos implícitos (ya instanciados) en las páginas JSF, algunos ejemplos son:

```
cookie: Map  
facesContext: FacesContext  
header: Map  
headerValues: Map  
param: Map  
paramValues: Map  
request (JSF 1.2): ServletRequest or PortletRequest  
session (JSF 1.2): HttpSession or PortletSession  
requestScope: Map  
sessionScope: Map  
applicationScope: Map  
initParam: Map  
view: UIViewRoot
```

## 5. JavaServer Faces

- Al igual que en los JSPs, la tecnología JSF permite acceder a varios de los objetos implícitos disponibles en sus propias páginas.
- Por ejemplo, si queremos acceder a un bean llamado empleadoBean, en el alcance de sesión, podemos utilizar cualquiera de las siguientes notaciones:
  - 1) #{empleadoBean.propiedad}
  - 2) #{sessionScope.empleadoBean.propiedad}
  - 3) #{sessionScope["empleadoBean"].propiedad}
- Debido a que el bean de empleado se encuentra en el alcance de sesión, podemos recuperarlo ya sea directamente o a través de la variable implícita sessionScope.

# 5. JavaServer Faces

## Operadores EL

El Lenguaje EL cuenta con operadores para evaluar expresiones:

**Aritméticos:** +, -, \*, / (div), % (mod)

**Relacionales:** == (eq), != (ne), < (lt), > (o gt), <= (le), >= (ge)

**Lógicos:** && (and), || (or), ! (not)

**Condicionales:** x ? y : z

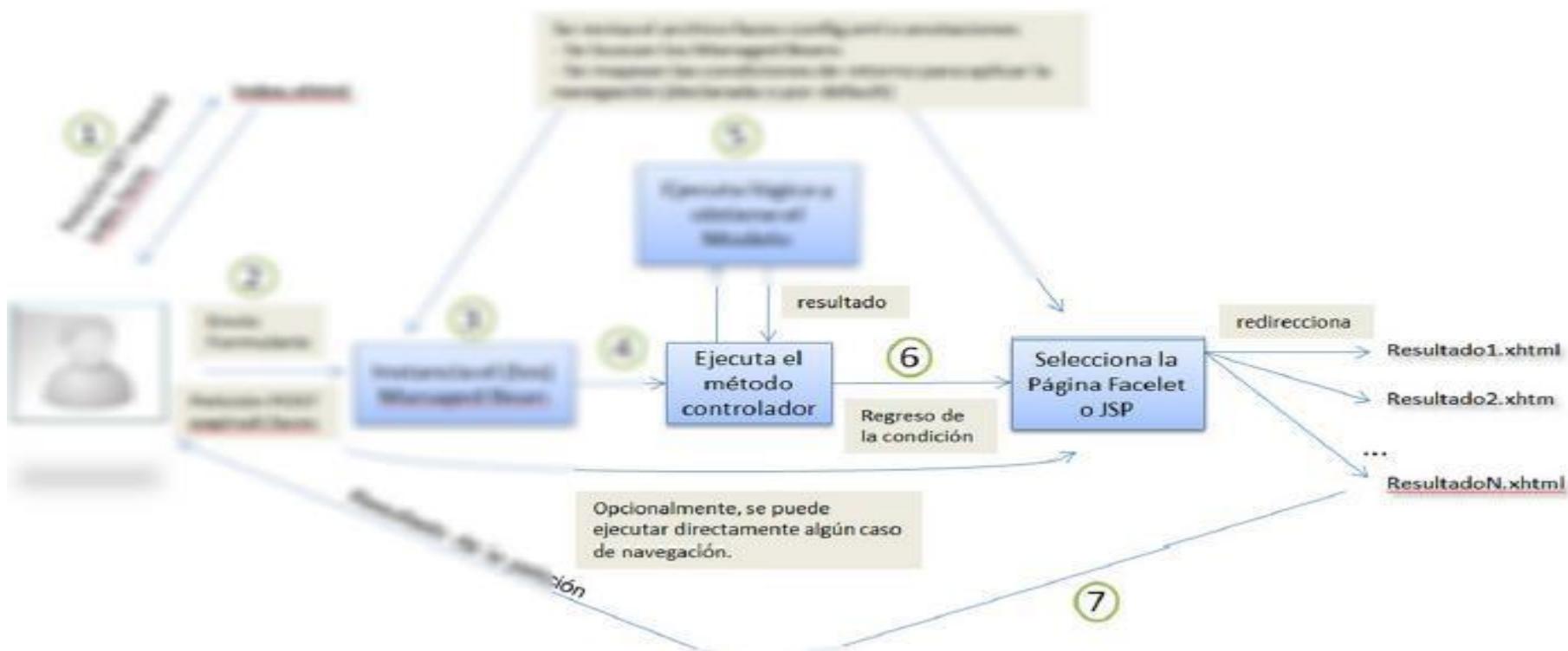
**Empty:** vacíos (regresa verdadero si el valor de la variable es null o sin elementos al manejar String, Arreglos ,Mapas o Colecciones)

### Extracto de Código

```
<h:panelGroup rendered="#{bean.propiedad ne empty and !bean.ocultar} " >  
    ...  
</h:panelGroup>
```

# 5. JavaServer Faces

## Navegación en JSF



- Resaltamos el paso 6. Aquí es donde normalmente definiremos la navegación de nuestra aplicación Web.

# 5. JavaServer Faces

## Navegación en JSF

La navegación en JSF permite movernos entre páginas de la misma tecnología

Existen varios tipos de Navegación

- Navegación Estática
- Navegación Dinámica

Existen varias formas de configurar la navegación

- Navegación Implícita (JSF 2.0)
- Navegación explícita (en el archivo faces-config.xml)

## 5. JavaServer Faces

- La navegación en JSF nos permite movernos entre una y otra página JSF.
- Existen varios tipos de Navegación:
  - **Navegación Estática:** El valor de la siguiente vista está definido por un nombre (cadena) fija.
  - **Navegación Dinámica:** El valor de la siguiente vista depende de la acción que ejecute el usuario y la cadena de regreso del método action ejecutado.

# 5. JavaServer Faces

- Existen varias formas de configurar la navegación:
  - **Navegación explícita** (en el archivo faces-config.xml): Se agrega un elemento xml que permite indicar de donde proviene la petición, cuál es su *salida* y hacia qué página debe redirigirse.
  - **Navegación Implícita** (JSF 2.0): La *salida* especificada, busca directamente una página con un nombre idéntico a la cadena de *salida* en el directorio actual, con la extensión de la página actual (ej. xhtml o jspx)
- En JSF 2.0 la navegación explícita toma precedencia sobre la navegación implícita.

# 5. JavaServer Faces

## Navegación Estática

La navegación estática aplica en los siguientes casos:

- No se requiere ejecutar código Java del lado del Servidor, sino únicamente dirigirse a otra página
- No hay lógica para determinar la página siguiente

Ejemplo de navegación estática:

- Página JSF que inicia la petición:

```
<h:commandButton label="Entrar" action="login"/>
```

- La página siguiente se puede determinar de 2 formas:
  - a) Outcome por default, buscar directamente la página login.xhtml
  - b) Buscar el outcome en faces-config.xml, encontrando el caso que determina la siguiente página a mostrar

## 5. JavaServer Faces

- El elemento para definir cual es la siguiente página a mostrar es una cadena (String) y se le conoce como *salida* (outcome). Se puede indicar el outcome en 2 lugares:
  - Directamente en el atributo action del componente JSF de tipo submit (navegación estática)
  - Como el tipo de retorno de un método action dentro de un Managed Bean(navegación dinámica)
- Ejemplo de navegación estática, el outcome se indica similar al componente:

```
<h:commandButton label="Entrar" action="login"/>
```

## 5. JavaServer Faces

- Una vez que se define la salida (outcome), JSF busca la página siguiente a mostrar, aplicando cualquiera de las 2 formas siguientes:
  - a) En el archivo faces-config.xml (caso de navegación explícito)
  - b) Directamente una página llamada login.xhtml (JSF 2.0) (caso de navegación implícito)

# 5. JavaServer Faces

## Navegación Dinámica

La navegación dinámica utiliza un método action, el cual tiene la lógica para determinar el outcome

Código página JSF (Vista)

```
<h:commandButton label="Aceptar"  
action="#{loginBean.verificarUsuario}"/>
```

-Código LoginBean (Controlador)

```
public String verificarUsuario () {  
    if (...)  
        return "exito";  
    else  
        return "fallo";  
}
```

outcome

## 5. JavaServer Faces

- La navegación dinámica utiliza un método action, el cual tiene la lógica para determinar el outcome.
- Una vez determinado el outcome, se puede aplicar una regla de navegación. El método action regresa un String, el cual es utilizado posteriormente por las reglas de navegación ya sea implícitas (busca una página directamente) o explícitas (busca en el archivo faces-config.xml).
  - En el código mostrado, el método de tipo action llamado verificarUsuario() regresa una cadena dependiendo de cierta lógica aplicada. La cadena que regresa será nuestra salida (outcome) y con ella se buscará una página a mostrar, en nuestro caso exito.xhtml o fracaso.xhtml. Si es navegaciónimplícita buscará un caso llamado éxito o fracaso, con ello determinará qué página debe mostrarse a continuación.

# 5. JavaServer Faces

## Creación de Reglas de Navegación

```
<faces-config ...>

    <navigation-rule>
        <from-view-id>/inicio.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>entrar</from-outcome>
            <to-view-id>/login.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>

    ...
</faces-config>
```

## 5. JavaServer Faces

- Podemos crear reglas de navegación en el archivo faces-config.xml.
- Para ello utilizamos el tag <navigation-rule>, con el indicamos lo siguiente:
  - a) La página de la cual proviene la petición (from-view-id)
  - b) Varios casos de navegación y por cada caso de navegación definimos:
    - i. La salida (outcome), ya sea estática o dinámica (cadena de regreso del método action del Managed Bean).
    - ii. La página destino.

## 5. JavaServer Faces

- Una página de origen puede tener como destino varias páginas y esto depende de la salida (outcome).
- Podemos crear **varias reglas de navegación** (navigation-rule), e incluso reglas de navegación genéricas. Un caso común de reglas genéricas es redirigir cualquier página a la página de inicio.

# 5. JavaServer Faces



## 5. JavaServer Faces

- La especificación de JSF, define 6 fases para el procesamiento de una petición HTTP.
  - **1) Restauración o creación de la Vista:** Obtiene el component tree, donde las etiquetas JSF se convierten en componentes Java.
  - **2) Aplicación de los valores de la petición:** Los parámetros enviados son mapeados con los componentes Java correspondientes.
  - **3) Procesamiento de validaciones y conversiones:** Si alguna validación falla, el estado del componente Java se marca como inválido, y se pasa al estado 6.
  - **4) Actualización de los valores del Modelo:** Los valores de los componentes Java (validados y convertidos) son puestos en los objetos de Modelo utilizando los métodos setters.

## 5. JavaServer Faces

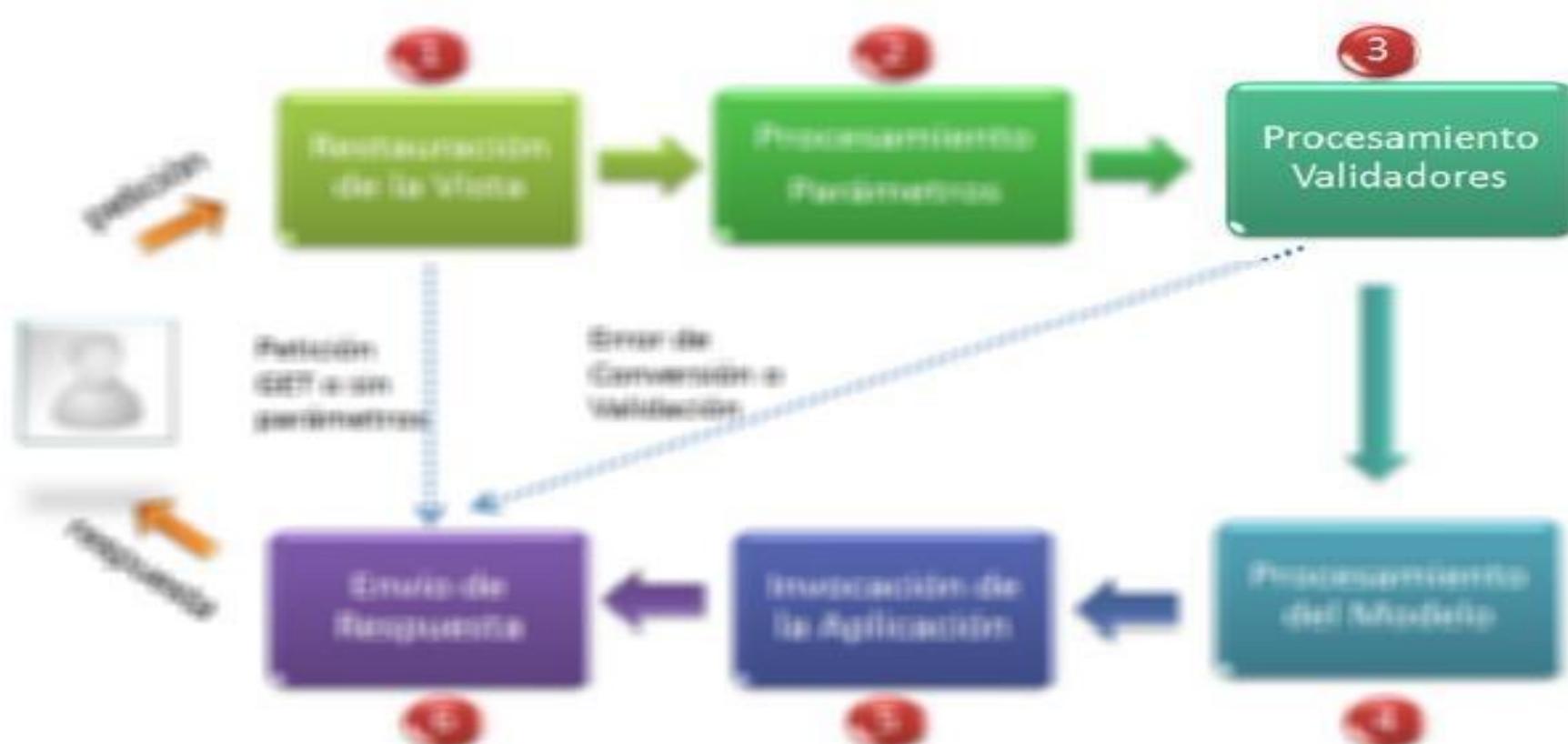
- **5) Invocación de la aplicación:** Se ejecutan los métodos action en los Managed Beans. Los métodos ActionListener son llamados antes que los métodos Action.
- **6) Envío de la respuesta:** Se genera la respuesta hacia el cliente que realizó la petición.
- Es posible crear clases para la depuración de las fases paso por paso. Para ello se debe implementar la interface **javax.faces.application.PhaseListener** y se debe registrar en el archivo faces-config.xml.

# 5. JavaServer Faces

Validadores, Convertidores y  
Manejo de Mensajes en JSF

# 5. JavaServer Faces

## Validadores en JSF



## 5. JavaServer Faces

- Las Validaciones en JSF aseguran que los **datos de nuestra aplicación sean consistentes según los valores introducidos por el usuario**. Algunas cuestiones que podemos validar son: rangos de números o límites, largo de cadenas, formatos de fechas entre otros.
- JSF provee distintos tipos de validaciones:
  - Validaciones estándar
  - Validaciones en los métodos de los managed beans
  - Validaciones personalizadas

# 5. JavaServer Faces

## Validaciones en JSF

Veremos dos maneras de ejecutar la validación en nuestros componentes:

- Validación individual por campo
- Validación por campos interdependientes

Podemos lograr la **Validación Individual** de varias maneras, por ejemplo:

- Agregando un atributo *required* al tag JSF
- Agregando un atributo validador al tag JSF
- Agregando un validador como un tag interno
- El tag h:message se utiliza para mostrar errores de un componente

La **Validación Interdependiente** se ejecuta dentro de los métodos actions

- El tag h:messages se utiliza para mostrar varios errores a la vez

# 5. JavaServer Faces

## Ejemplo de uso de Validadores

Ejemplo de un validador estándar:

```
<h:inputText id="edadId" required="true" value="#{empleadoBean.edad}">
    <f:validateLongRange minimum="18" maximum="50"/>
</h:inputText>
```

Ejemplo de validador personalizado:

```
<h:inputText id="ipID" required="true" value="#{ipBean.ipValor}">
    <f:validator validatorId="validadorIp"/>
</h:inputText>
```

## 5. JavaServer Faces

- JSF también permite crear validadores en caso de que los existentes no cubran nuestras necesidades implementando la interfaz Validator. Además debemos utilizar la anotación @FacesValidator e implementar el método validate (...) en el bean validador.

# 5. JavaServer Faces

## Convertidores en JSF

Los convertidores cubren la necesidad de asignar valores entre la vista y el modelo de manera automática



## 5. JavaServer Faces

- Los convertidores cubren la necesidad de asignar valores entre la vista y el modelo de manera automática, manejando tipos diferentes al de por defecto (String).
  - Por ejemplo, convirtiendo una cadena fecha y asignándola directamente a la propiedad de nuestro modelo manejando el mismo tipo.
- De igual manera, cubre la necesidad de desplegar la información del modelo con un formato específico, por ejemplo una formato de una fecha, un valor decimal con ciertas posiciones, etc.
- A continuación veremos los **tipos de convertidores manejados en JSE**.

# 5. JavaServer Faces

## Convertidores Implícitos y Explícitos

### Conversiones Implícitas:

- Son todas las conversiones que JSF realiza de manera automática, por ejemplo al usar tipos primitivos, BigInteger o String.

```
<h:inputText id="edadId" value="#{empleadoBean.edad}">
```

### Conversiones Explícitas: Tenemos dos maneras de hacerlo

- Utilizando el atributo **converter**, por ejemplo:

```
<h:inputText value="#{empleadoBean.edad}"  
converter="javax.faces.Integer" />
```

- Utilizando un componente de tipo **converter**:

```
<h:inputText value="#{empleadoBean.edad}">  
    <f:converter converterId="javax.faces.Integer" />  
</h:inputText>
```

## 5. JavaServer Faces

- Además, el framework de JSF tiene convertidores estándar, los cuales podemos utilizar de inmediato para ciertos tipos de datos. Algunos ejemplos son:
  - **Convertidor de Números:** Permite convertir el valor de un componente en alguna subclase de `java.lang.Number`.  
Ej. `<f:convertNumber type="currency" />` :
  - **Convertidor de Fechas:** Permite convertir un valor de un componente a la clase `java.util.Date`.

Ej. `<f:converterDateTime pattern="dd/MM/yyyy" />`

# 5. JavaServer Faces

## Convertidores Personalizados

JSF permite crear convertidores personalizados

Elementos de un convertidor:

- a) Elemento JSF, por ejemplo:

```
<h:inputText id="fechaId" value="#{empleadoBean.fechaNacimiento}"  
    convert="util.ConvertidorFecha" />
```

- b) Clase Java URLConverter: Contiene el código del convertidor, debe implementar la interfaz [javax.faces.convert.Converter](#)
- c) El convertidor URLConverter se registra en faces-config.xml o agregar la anotación @FacesConverter a la clase URLConverter

# 5. JavaServer Faces

## Convertidores Personalizados

JSF permite crear convertidores personalizados

Elementos de un convertidor:

- a) Elemento JSF, por ejemplo:

```
<h:inputText id="fechaId" value="#{empleadoBean.fechaNacimiento}"  
    convert="util.ConvertidorFecha" />
```

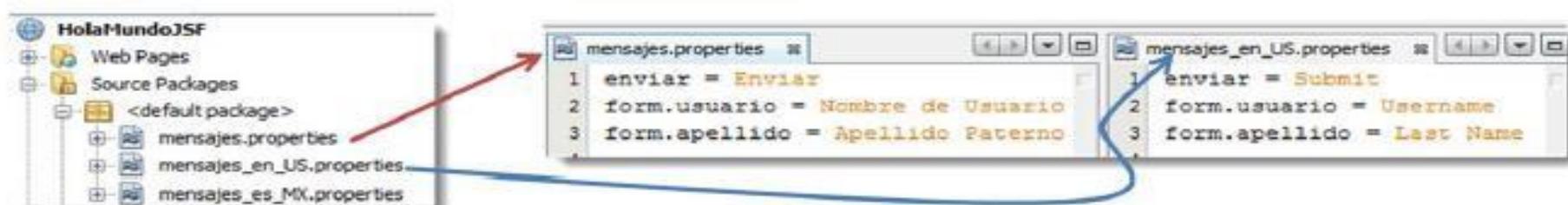
- b) Clase Java URLConverter: Contiene el código del convertidor, debe implementar la interfaz [javax.faces.convert.Converter](#)
- c) El convertidor URLConverter se registra en faces-config.xml o agregar la anotación @FacesConverter a la clase URLConverter

# 5. JavaServer Faces

## Internacionalización en JSF

- JSF tiene soporte total para la especificación I18n de Java, para la especificación del Idioma en la aplicación Web

Código del Lenguaje / Subregión	Descripción
es	Español
es_MX	Español / México
en	Inglés
en_GB	Inglés / Británico
en_US	Inglés / Estados Unidos



- Lista de lenguajes
  - [http://www.loc.gov/standards/iso639-2/php/code\\_list.php](http://www.loc.gov/standards/iso639-2/php/code_list.php)

## 5. JavaServer Faces

- Una aplicación Web puede tener la necesidad de mostrar sus páginas en varios idiomas. JSF resuelve este requerimiento por medio de archivos de propiedades que soportan el estándar I18n de Java.
- Estos archivos nos permiten especificar nuestras etiquetas de las páginas Web en varios idiomas.
- Podemos observar en la figura que tenemos un archivo de mensajes con el idioma por defecto, en este caso español. Y además tenemos 2 idiomas, español de México e inglés de Estados Unidos.

## 5. JavaServer Faces

- El idioma por default de la aplicación se puede especificar ya sea en el archivo faces-config.xml o vía programática en algún método action de un Managed Bean. El código necesario es:
  - `FacesContext.getCurrentInstance().getViewRoot().setLocale("es_ES");`
  - `FacesContext.getCurrentInstance().getViewRoot().setLocale("es_MX")`

# 5. JavaServer Faces

## Resource Bundle en JSF

JSF puede cargar las etiquetas de un archivo de propiedades (Resource Bundle)

En el archivo faces-config.xml se configura el archivo a utilizar.

### faces-config.xml

```
<application>
    <resource-bundle>
        <base-name>mensajes</base-name>
        <var>msg</var>
    </resource-bundle>
</application>
```

Se omite la extensión .properties

Podemos utilizar el archivo de propiedades en la página JSF como sigue:

```
<h:outputText value="#{msg['form.usuario']}"/>

<h:commandButton value="#{msg.enviar}" type="submit"
action="login"/>
```

# 5. JavaServer Faces

- Los archivos de resource bundle nos servirán para:
  - Centralizar las etiquetas de nuestros formularios.
  - Configurar el idioma de la aplicación (internacionalización).
- El archivo a utilizar se puede especificar en el archivo faces-config.xml u opcionalmente, se puede especificar en la página JSF a utilizar:

```
<f:loadBundle basename="mensajes" var="msg"/>
```

- Una vez que hemos configurado el archivo de propiedades, podemos reemplazar las etiquetas de nuestras páginas JSF con sintaxis como:
  - <h:outputText value="#{msg[„form.usuario”]} />
  - <h:commandButton value="#{msg.enviar}" type="submit" action="login" />

# 5. JavaServer Faces

## Sobreescritura de Mensajes JSF

Se debe crear un archivo (no importa el nombre) de propiedades  
jsf.properties

Se sobreescreiben los mensajes del sistema deseados  
javax.faces.component.UILnput.REQUIRED=Valor Requerido  
• Se configura en el archivo faces-config.xml

### faces-config.xml

```
<application>
    <message-bundle>jsf</message-bundle>
</application>
```

Se omite la extensión  
.properties

## 5. JavaServer Faces

- JSF maneja la localización de errores y la información que ocurre en eventos como conversiones, validaciones y otras acciones resultado del ciclo de vida de JSF.
- Podemos sobreescibir estos mensajes, creando un nuevo archivo resource bundle y configurando las entradas que se desean modificar.
- En el código de ejemplo podemos observar que estamos sobre escribiendo el valor por default para la validación de algún campo requerido.

# 5. JavaServer Faces

Manejo de Eventos y Librería  
Estándar de JSF

# 5. JavaServer Faces

## Value Change Listeners en JSF

JSF provee dos maneras de detectar cambios (value change) en los componentes, ya sea agregando un atributo valueChangeListener al componente o como un elemento interno. Por ejemplo:

```
<h:inputText id="codigoPostal" onchange="this.form.submit()"  
value="#{empleadoBean.codigoPostal}"  
valueChangeListener="#{vacanteForm.codigoPostalListener}" />
```

Una vez definido el atributo o elemento interno, se debe agregar el método action en el Managed Bean que escucha el cambio en la Vista.

```
public void codigoPostalListener(ValueChangeEvent valueChangeEvent) { ... }
```

# 5. JavaServer Faces

## Action Listeners en JSF

JSF provee dos maneras de detectar acciones cuando un usuario ejecuta una acción de tipo submit, ya sea agregando el atributo actionListener al componente o como un elemento interno

Algunos componentes de este tipo son h:commandbutton y h:commandLink

```
<h:commandLink actionListener="#{vacanteForm.enviar}" />
```

Una vez definido el atributo o elemento interno, se debe agregar el método action en el Managed Bean que procesa la petición de la Vista

```
public void enviar(ActionEvent actionEvent) { ... }
```

# 5. JavaServer Faces

## Componentes HTML en JSF

El API de los JSF provee varios componentes básicos para el despliegue de información HTML

<h:form />	<h:inputText />	<h:inputTextarea />
<h:inputSecret />	<h:inputHidden />	<h:outputLabel />
<h:outputLink />	<h:outputFormat />	<h:outputText />
<h:commandButton />	<h:commandLink />	<h:message />
<h:messages />	<h:panelGrid />	<h:panelGroup />
<h:dataTable />	<h:column />	<h:selectOneListbox />
<h:selectOneRadio />	<h:selectBooleanCheckbox />	<h:selectManyCheckbox />
<h:selectManyListbox />	<h:selectManyMenu />	

# 5. JavaServer Faces

## Componentes core de JSF

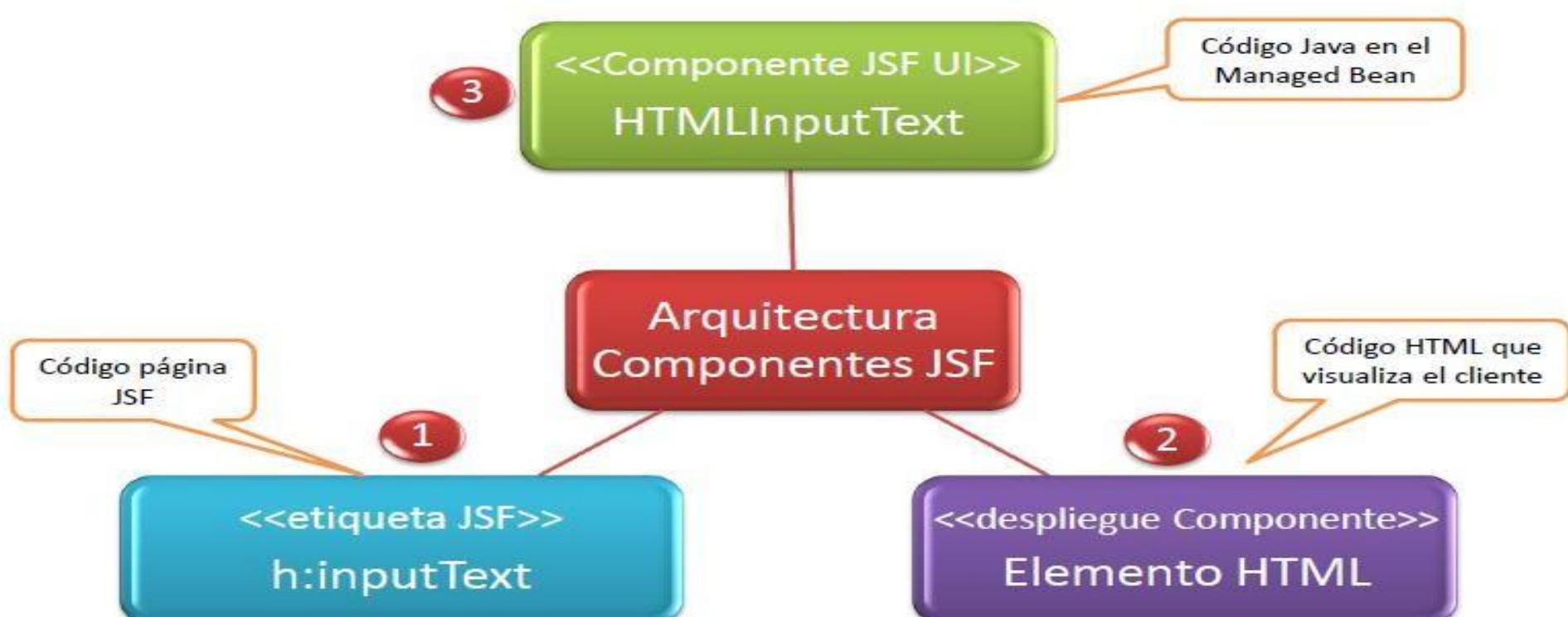
El API de los JSF provee varios componentes core para el manejo de eventos, administración de componentes, entre otros.

<f:view />	<f:subview />	<f:facet />
<f:attribute />	<f:param />	<f:actionListener />
<f:valueChangeListener />	<f:converter />	<f:convertDateTime />
<f:convertNumber />	<f:validator />	<f:validateDoubleRange />
<f:validateLength />	<f:validateLongRange />	<f:loadBundle />
<f:selectItems />	<f:selectItem />	<f:verbatim />

# 5. JavaServer Faces

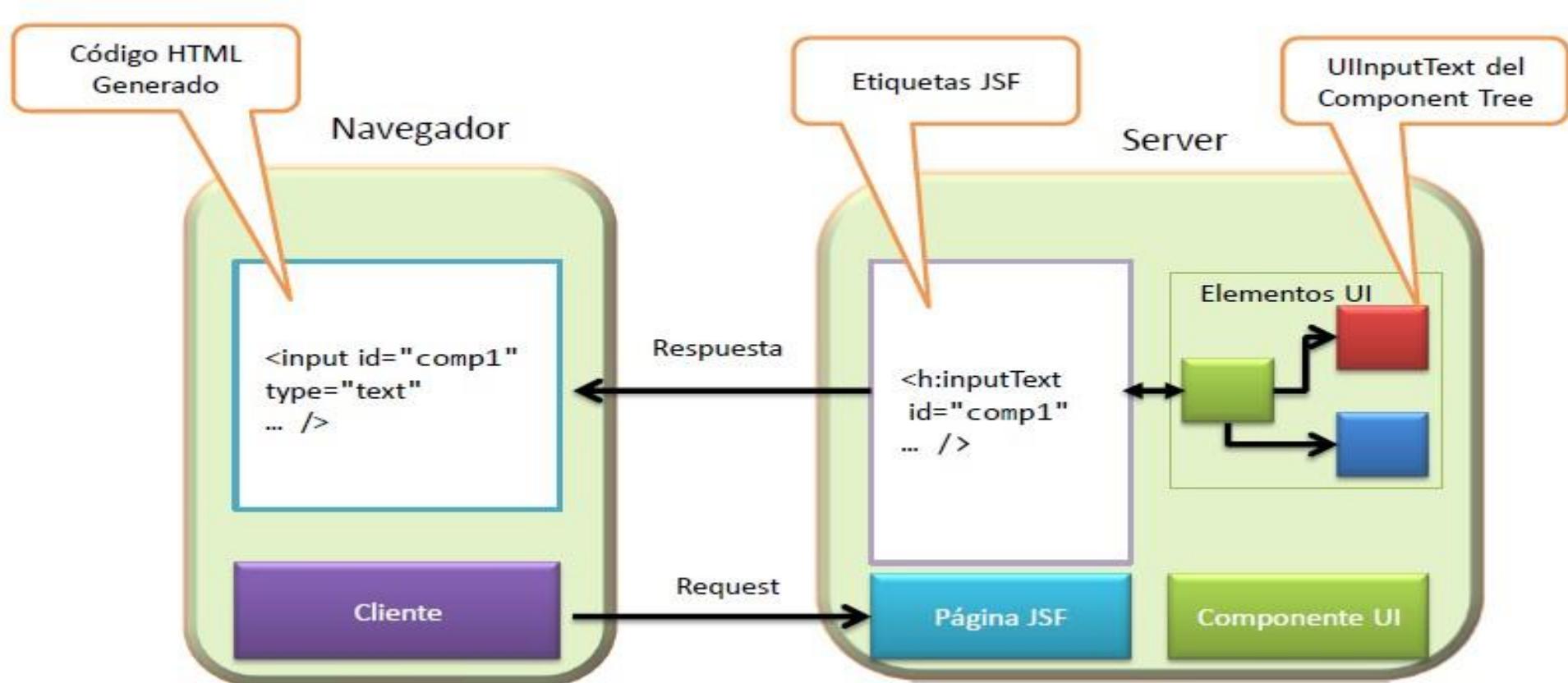
## Arquitectura de Componentes JSF

Un componente JSF se compone de 3 partes:



# 5. JavaServer Faces

## Component Tree en JSF



# 5. JavaServer Faces

## Componentes de Selección en JSF

El API de JSF provee de distintos tipos de componentes de selección

```
<h:selectOneListbox />
<h:selectOneMenu />
<h:selectOneRadio />
<h:selectBooleanCheckbox />
<h:selectManyCheckbox />
<h:selectManyListbox />
<h:selectManyMenu />
```

En la página de corejsf podemos encontrar un ejemplo de los componentes JSF.

[www.corejsf.org/jsf-tags.html](http://www.corejsf.org/jsf-tags.html)

# 5. JavaServer Faces

## Select Items en JSF

Los componentes JSF tales como h:selectOneMenu muestran una lista de datos, los cuales se conocen como **Select Items** (elementos de selección)

Lista de elementos Select Item ligada a un propiedad del Managed Bean:

```
<h:selectOneMenu>
    <f:selectItems value="#{coloniaHelper.coloniaSelectItems}" />
</h:selectOneMenu>
```

Lista de elementos en código duro en la página JSF:

```
<h:selectOneMenu>
    <f:selectItems>
        <f:selectItem itemLabel="Rojo" itemValue="rojo" />
        <f:selectItem itemLabel="Verde" itemValue="verde" />
        <f:selectItem itemLabel="Azul" itemValue="azul" />
    </f:selectItems>
</h:selectOneMenu>
```

# 5. JavaServer Faces

Facelets de JSF

# 5. JavaServer Faces

## Características de Facelets

Los Facelets es la tecnología estándar de despliegue en JSF 2.0

Los Facelets eliminan completamente la necesidad de los JSP's

Utilizan un parser XML en lugar del compilador de JSP

Comparado con JSP, los Facelets crean un component tree más ligero

Los Facelets resultan hasta 30% más rápidos en compilación

Soporte para Templates

Creación de componentes compuestos

# 5. JavaServer Faces

## Los Facelets no son JSPs

Dentro de una página Facelet **NO** es posible utilizar los siguientes taglib de JSP:

```
<jsp:root/>
<jsp:directive.include.../>
<jsp:output.../>
<jsp:directive.content.../>
```

- Sin embargo, **SÍ** es posible utilizar los tags de JSTL en los Facelets:

```
<c:forEach... />
<c:if... />
<c:catch... />
```

# 5. JavaServer Faces

## Plantillas con Facelets

Las plantillas definen las regiones lógicas de una página JSF.

Los elementos generales son:



- **Template** (Plantilla): Página utilizada para controlar el layout (disposición de los elementos)
- **Template-client** (Cliente de la Plantilla) : Página que personaliza su propia distribución (layout) a partir de la plantilla que implementa

Los clientes acceden al Template-client, NO a la plantilla.

# 5. JavaServer Faces

## Componentes Compuestos

La creación de nuevos componentes JSF requerían de crear clases Java

Los facelets permiten crear componentes reutilizables combinando HTML y tags de JSF

- Permite la reutilización de otros componentes
- Se definen en un archivo XHTML, no en clases Java

Los componentes compuestos son básicamente templates asociados a un tag library personalizado

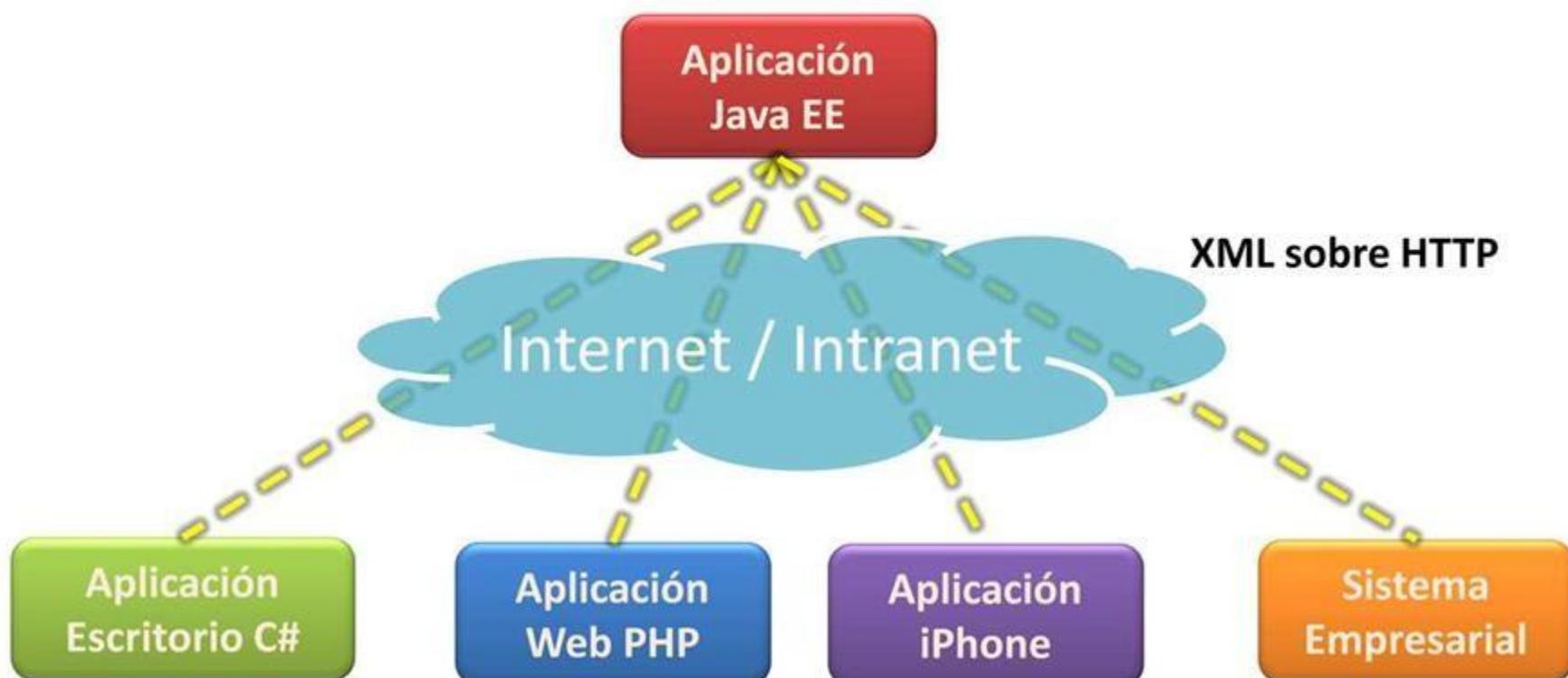
Para referenciar un componente compuesto basta con importar el namespace en el XHTML

# Índice

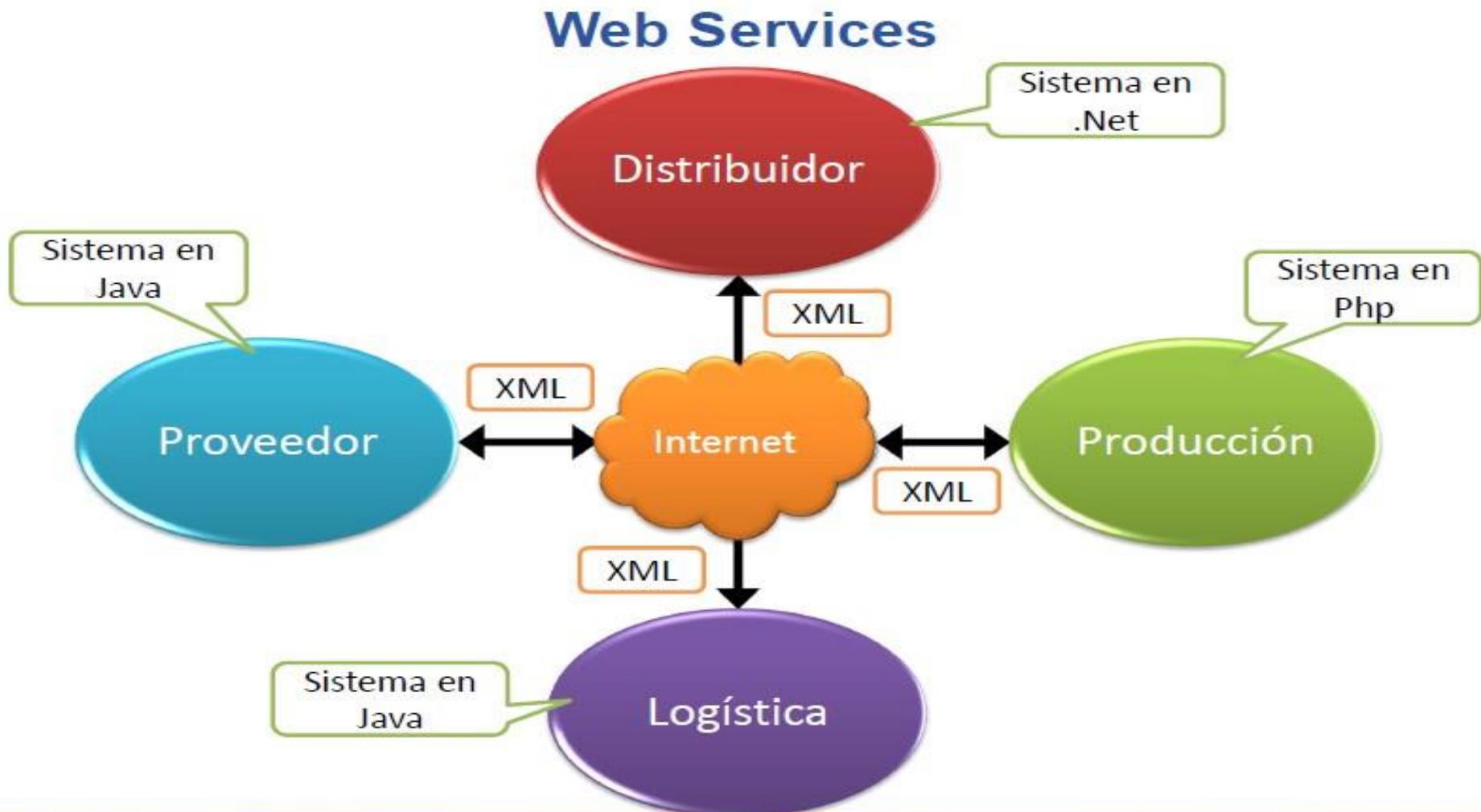
- 
1. Introducción Java Empresarial
  2. Enterprise Java Beans (EJBs) y Context Directory Inyection (CDI)
  3. Java Persistence Api (JPA)
  4. Servlets y jsps en Java EE
  5. JavaServer Faces
  6. **Web Services (SOAP y REST)**
  7. Seguridad en JEE

## 6. Web Services

### ¿Qué son los Web Services?



# 6. Web Services



## 6. Web Services

- El tema de Web Services lleva utilizándose ya durante varios años, a lo largo de estos años se han aprendido técnicas y nuevas formas de establecer una comunicación más eficiente entre sistemas creados en distintas plataformas o tecnologías.
- Los Web Services son una tecnología orientada a la intercomunicación de sistemas.

## 6. Web Services

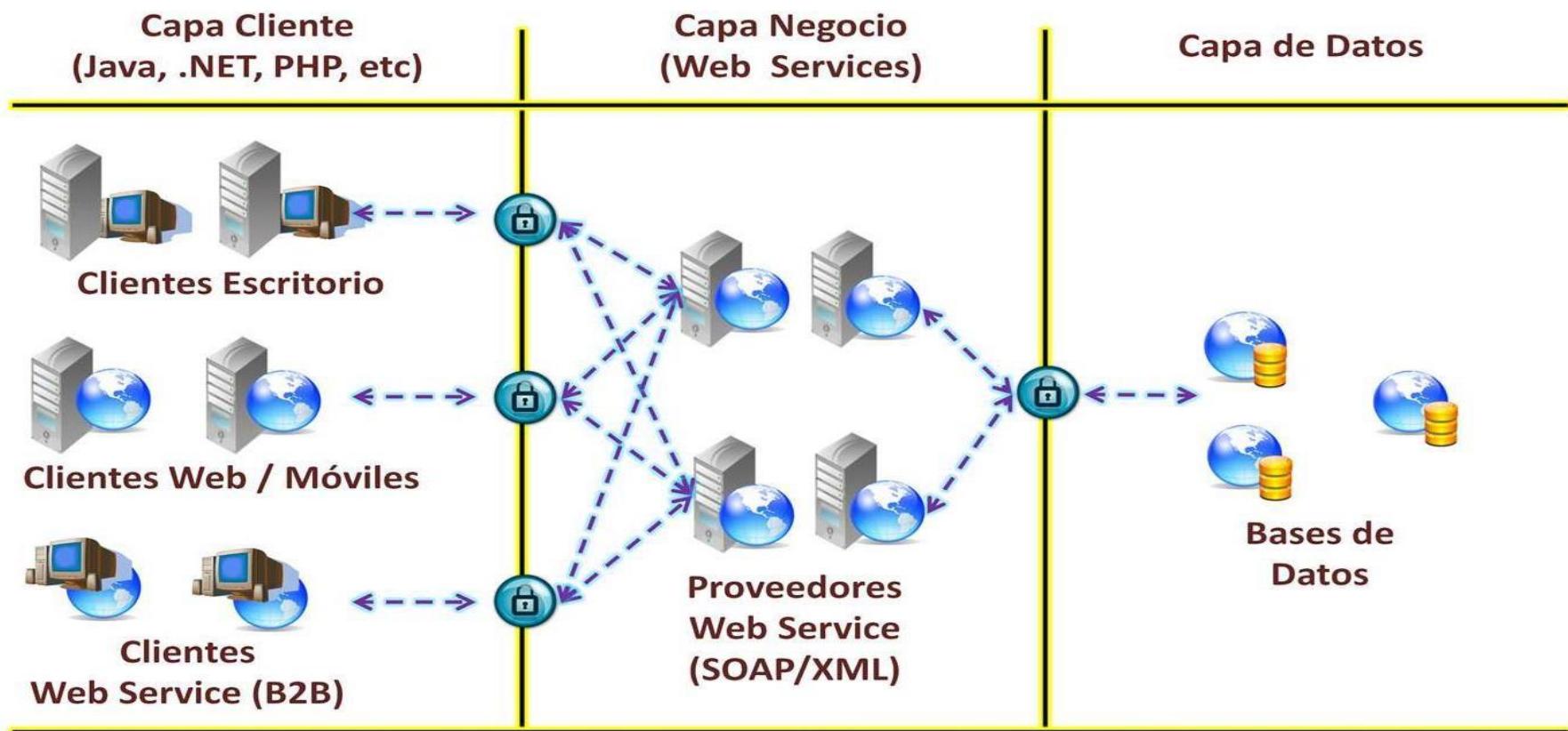
- Algunas de sus características son:
  - **Interoperabilidad y Portabilidad:** Los sistemas que utilizan Servicios Web permiten intercambiar información entre distintas plataformas y lenguajes de programación utilizando la Web (Intranet/Internet) como base para la comunicación. Una de las formas es utilizar el protocolo en SOAP permitiendo la comunicación vía XML, logrando la independencia de plataforma. Todo esto apoyándose del protocolo HTTP.
  - **Reusabilidad:** Permite reutilizar mucha de la lógica de negocio proveniente de sistemas legados o de nuestros sistemas empresariales actuales.
  - **Disponibilidad:** El objetivo de los servicios web es que se encuentran disponibles en cualquier momento y en cualquier lugar, para cualquier sistema y/o persona que necesite utilizarlos. Además, no requieren de intervención humana incluso en transacciones muy complejas.

## 6. Web Services

- En muchas ocasiones necesitamos intercomunicar sistemas de información, los cuales pueden haber sido desarrollados en la misma tecnología o no.
- JEE cuenta con dos APIs principales para el desarrollo de Web Services: Java API for XML Web Services (JAX-WS) y Java API for RESTful Web Services (JAX-RS)

# 6. Web Services

## Arquitectura Web Services



## 6. Web Services

- En una aplicación Java Empresarial, los Web Services se pueden ver como una alternativa para exponer los EJBs sin necesidad de utilizar RMI o el uso de Java.
- Los EJBs se pueden exponer como Servicios Web, con ello reutilizamos su lógica de negocio. Lo anterior permite a **El Cliente** ser escrito en cualquier lenguaje como Objective-C, .Net, PHP, etc, sin ninguna dependencia con el lenguaje Java.
- Los Web Services son una parte primordial para el desarrollo de arquitecturas SOA (Service-Oriented Architecture) con el objetivo de integrar aplicaciones creadas en la misma o distintas plataformas.

# 6. Web Services

## Tipos de Web Services

En Java EE 6, JAX-WS y JAX-RS son los dos estándares para crear Web Services:

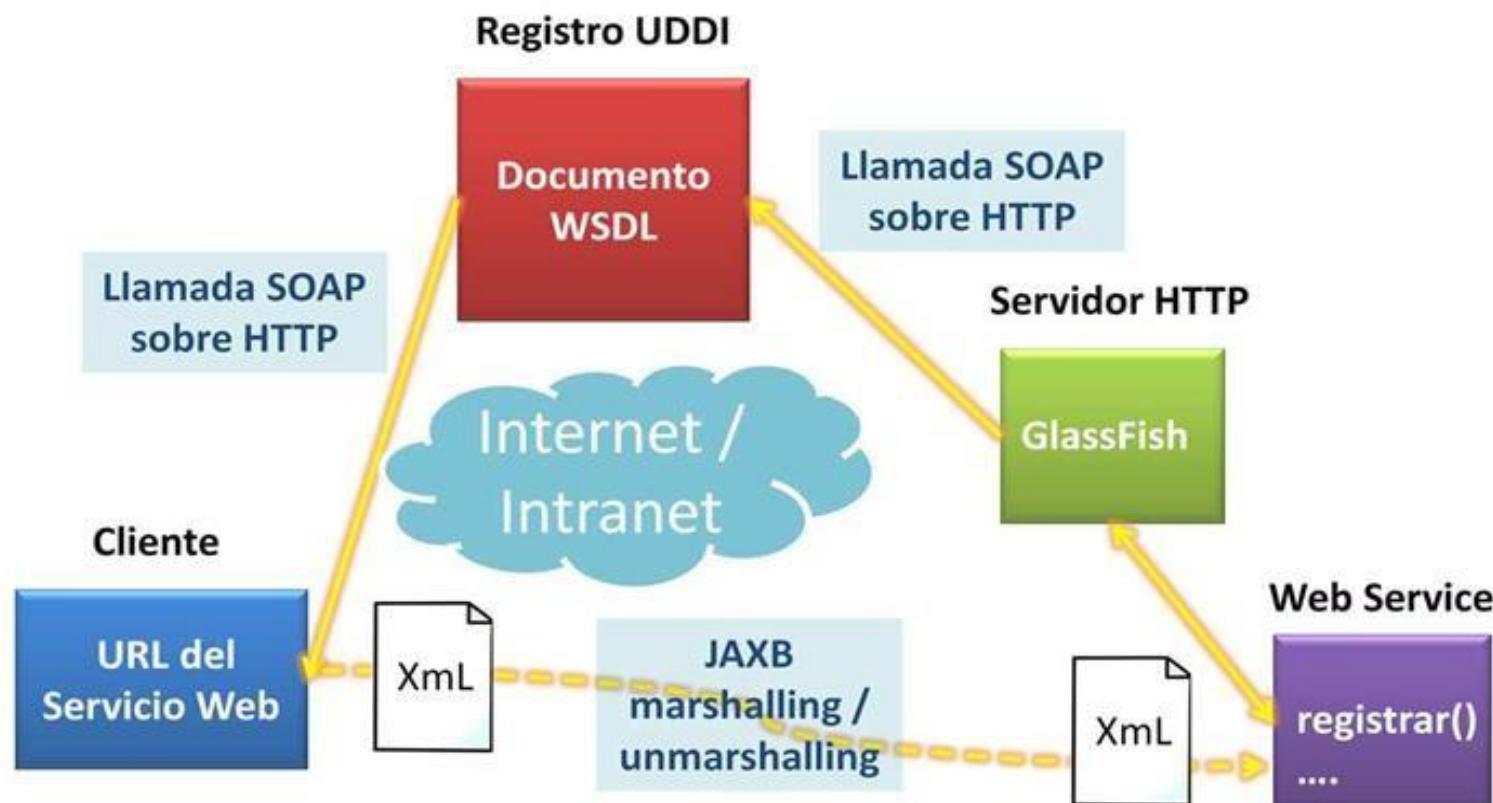
- ✓ **JAX-WS (Java API for XML Web Services)** es un API que permite abordar requerimientos empresariales más complejos al momento de crear Web Services. También se conocen como **SOAP Web Services**.
- ✓ **JAX-RS (Java API for RESTful Web Services)** es un API más simple de utilizar y de implementar al momento de crear de Web Services. También se conocen como Restful Web Services.
- ✓ Se recomienda utilizar JAX-WS para integrar sistemas empresariales.
- ✓ Se recomienda utilizar JAX-RS para exponer funcionalidad a aplicaciones Web 2.0, por ejemplo un cliente iPhone o Android.

## 6. Web Services

- Cuando utilizamos Java EE, debemos decidir si utilizamos el API de JAX-WS o JAX-RS. Para tomar este decisión podemos tomar en cuenta los siguientes puntos:
  - **JAX-WS:** Esta API permite abordar **requerimientos más complejos** y lleva más años en el mercado. Brinda soporte para los protocolos que son estándar en la industria de software al crear Web Services. Estos estándares proveen una manera muy **robusta para agregar seguridad, transaccionalidad, interoperabilidad**, entre varias características entre el cliente y el servidor al utilizar Web Services.
  - **JAX-RS:** Esta API permite crear de manera **más simple** Web Services, solo que está **restringido por el estilo REST**, el cual utiliza el protocolo HTTP y sus métodos soportados y códigos de estado para establecer las operaciones básicas de un Servicio Web (GET, POST, PUT, DELETE, etc).

# 6. Web Services

## SOAP Web Services



## 6. Web Services

- En términos simples un Web Service es una plataforma estándar que provee interoperabilidad entre distintas aplicaciones.
- Para la mayoría de los programadores esto significa envío y recepción de mensajes XML los cuales son transmitidos vía HTTP/HTTPS.
- Tanto el lenguaje XML como el protocolo HTTP son un estándar y son ampliamente aceptados para el envío y recepción de información, así, esta información puede ser procesada por múltiples clientes con distintas tecnologías.

## 6. Web Services

- Como se observa en la figura, un Web Service se publica en un tipo directorio conocido como UDDI, el cual significa **Universal Description Discovery and Integration**.
- UDDI es un estándar y tiene como objetivo publicar y permitir el acceso a los Web Services a través de mensajes SOAP.
- SOAP (Simple Object Access Protocol) es un protocolo estándar el cual define la forma en que se intercambia datos de tipo XML.
- WSDL (Web Services Description Language) nos permite describir los Web Services. WSDL describe la interface pública de nuestros Web Services y se utiliza XML para su descripción.

## 6. Web Services

- Una vez que ya conocemos la ubicación del Web Services a través del WSDL (URI) del mismo, podemos enviar la petición SOAP y así ejecutar el método expuesto del Web Service. En este procedimiento el XML de petición y respuesta suele validarse utilizando XSD (schema).
- Una vez llegado el mensaje XML al servidor Java, este debe convertirse en objetos Java. Para esto es común utilizar la tecnología JAXB, el cual nos permitirá convertir objetos Java en documentos XML (marshaling) y viceversa (unmarshaling).

## 6. Web Services

- Son varias las tecnologías relacionadas para entender a detalle la creación de SOAP Web Services, sin embargo la buena noticia es que el estándar JAX-WS nos permitirá ocultar y automatizar muchos de los pasos necesarios para crear un Servicio Web, así como la creación del Cliente del Web Services.

# 6. Web Services

## WSDL (Web Services Description Language)

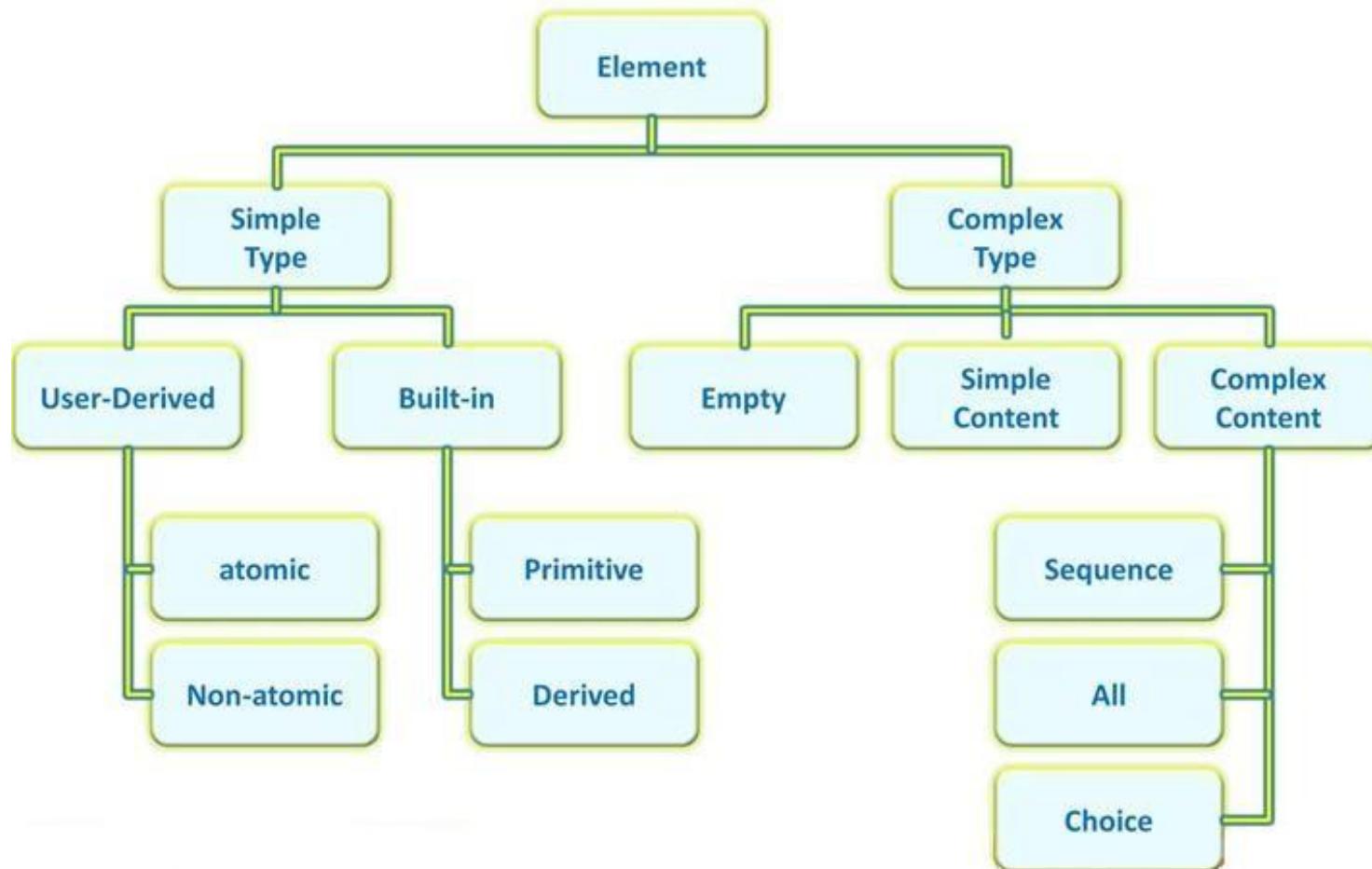


## 6. Web Services

- WSDL (Web Services Description Language) es un lenguaje basado en XML, el cual provee la descripción de un servicio Web.
- De hecho provee una descripción bastante detallada del Servicio Web y con la cual es posible generar tanto el código del Cliente como del Servicio Web asociado.
- El código generado manejará en automático la conversión de código Java a XML (marshaling) y viceversa (unmarshaling).

# 6. Web Services

## XML y Schema (XSD)



## 6. Web Services

- Existen varias formas de validar un documento XML. Esto es básico al momento de transmitir un mensaje a través de un Web Services. La validación nos permitirá automatizar los mensajes entre el Cliente y el Servidor sin necesidad de intervención humana.
- Para validar un documento XML es posible realizarlo por medio de 2 técnicas. La primera conocida como DTD (Document Type Definition) y la segunda como XSD (Schema XML). La validación de archivos por DTD es cada vez menos utilizada, debido a que el lenguaje que se utiliza no es XML, y por lo tanto tiene varias limitantes al momento de validar documentos XML con restricciones más avanzadas.

## 6. Web Services

- Por otro lado, la validación por XML Schema es cada vez más utilizado para validar la estructura y restricciones de un documento XML. Ya que permite agregar mucho más precisión al momento de validar elementos y atributos complejos de un documento XML, y debido a que está creado con el mismo lenguaje XML, favorece bastante para crear validaciones robustas y flexibles.
- En la figura podemos observar las consideraciones a tomar en cuenta para la validación de un documento XML, y se puede observar que son tan detalladas como se necesite, por ejemplo:

# 6. Web Services

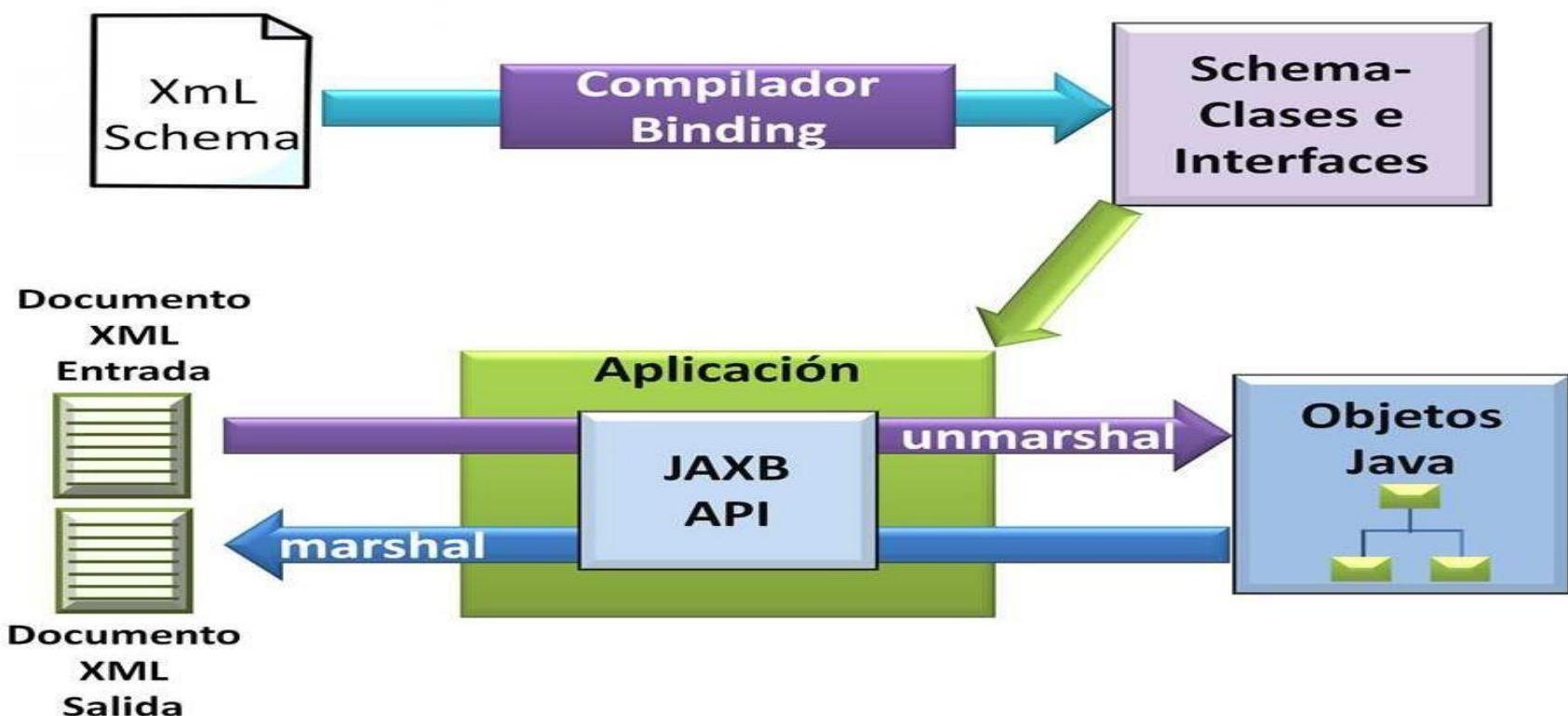
```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="nota">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="para" type="xs:string"/>
        <xs:element name="de" type="xs:string"/>
        <xs:element name="titulo" type="xs:string"/>
        <xs:element name="contenido" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

- Un ejemplo de un XML validado por el esquema anterior puede ser el siguiente ejemplo:

```
<nota>
  <para>Juan</para>
  <de>Karla</de>
  <titulo>Recordatorio</titulo>
  <contenido>Nos vemos el viernes!</contenido>
</nota>
```

# 6. Web Services

## JAXB (Java Architecture for XML Binding)

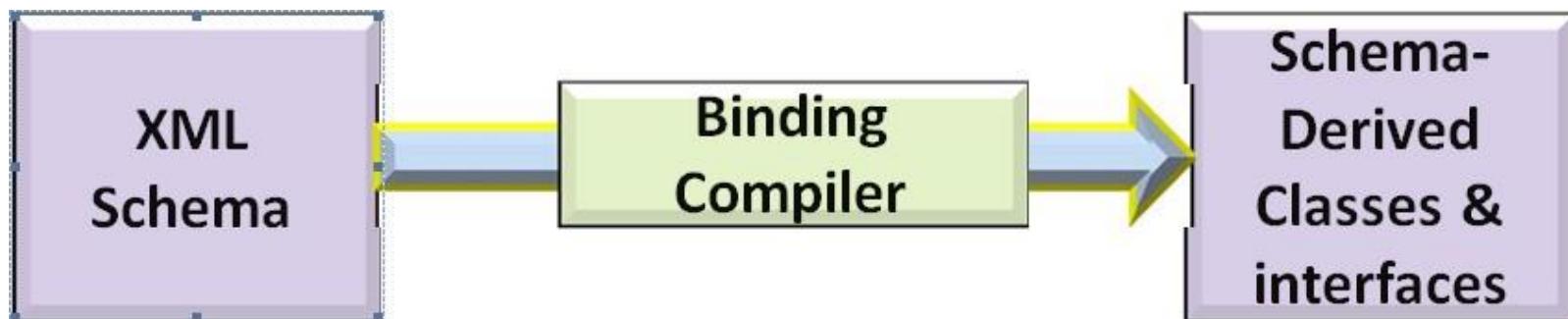


## 6. Web Services

- Al trabajar con Web Services, es necesario en algún momento convertir los mensajes XML en objetos Java y viceversa. Existen varias APIs para realizar esta labor, sin embargo el estándar en la versión Java EE es el API de JAXB (Java Architecture for XML Binding).
- Como podemos observar en la figura, la tecnología JAXB provee dos principales características. La habilidad de convertir un objeto Java en un documento XML (marshal) y viceversa (unmarshaling).
- JAXB permite de manera muy simple acceder y procesar documentos XML sin necesidad de conocer a detalle XML u otras API's de procesamiento.

## 6. Web Services

- JAXB asocia el esquema (XSD) del documento XML a procesar (binding), y posteriormente genera las clases Java que representan el documento XML (unmarshalling).



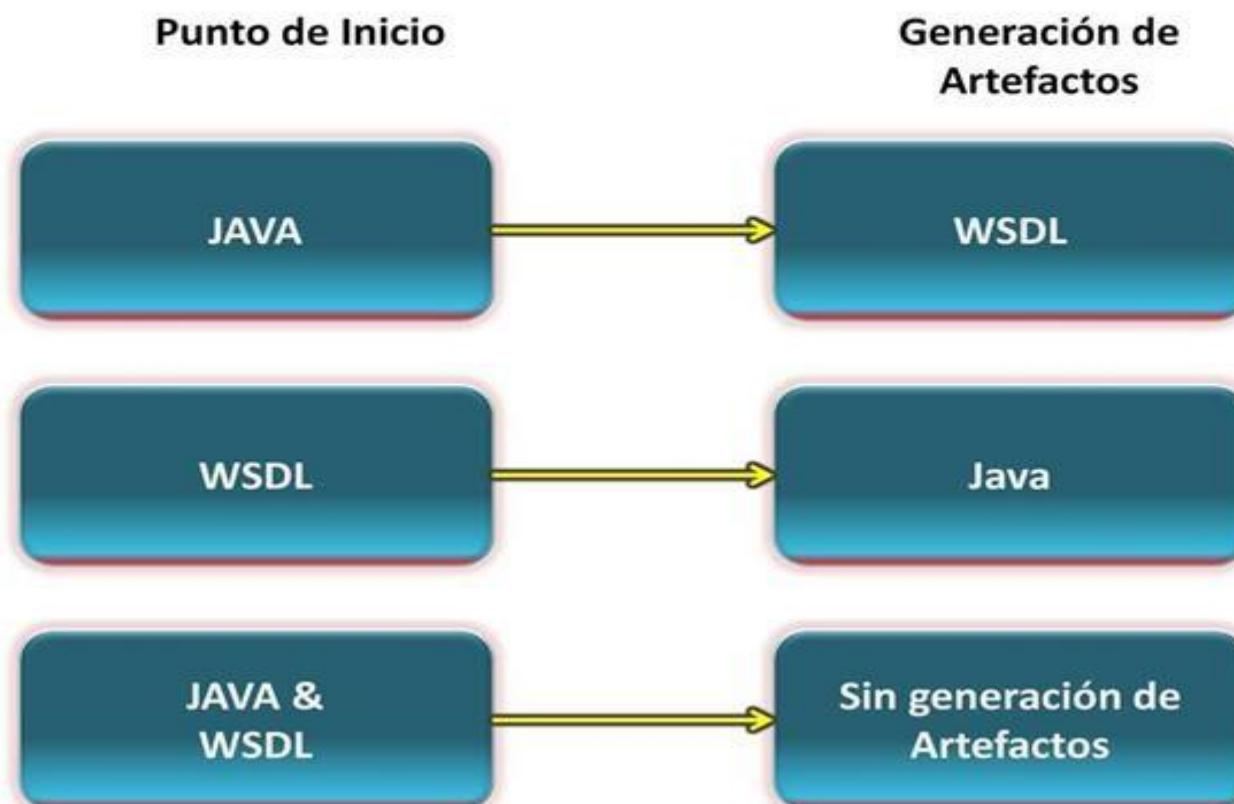
- Después de generado el código, se puede acceder y desplegar los datos del documento XML asociado, simplemente utilizando las clases y objetos Java generados. No hay necesidad de utilizar un parser XML o incluso, no hay necesidad de conocer el documento XML original.

## 6. Web Services

- También es posible el proceso inverso (marshalling), en el cual a partir de código Java se generan el documento XSD que representa la estructura de los objetos Java y su relación.

# 6. Web Services

## Estrategia Generación Web Services



## 6. Web Services

- Cuando creamos un nuevo Servicio Web, hay dos artefactos que deben generarse: El documento WSDL y las clases Java que implementan el Servicio Web. Además, se debe generar el documento XSD (esquema XML) asociado al mensaje XML a intercambiar por el Servicio Web.
- Con el documento WSDL y el esquema XSD, un cliente de un Web Services puede ser autogenerado con ciertas herramientas, por el comando `wstool` de Java. Este comando permite generar el código Java necesario para invocar un Web Service a partir de la URL del WSDL. Existen herramientas similares en otros lenguajes para crear los clientes de los SOAP Web Services de manera muy similar.

## 6. Web Services

- Cuando ya tenemos listo el código Java, delegamos la generación del Servicio Web al API de JAX- WS, lo que generará de manera automática tanto el documento WSDL y el archivo XSD asociado al mensaje XML a intercambiar, todo esto basado en el código Java y la relación existente en el método Web Service a exponer.
- JAX-WS puede exponer métodos de clases Java POJO"s o de EJB"s. Seleccionar uno u otro dependerá de si queremos aprovechar los beneficios de tener un EJB, o utilizar un POJO para exponer métodos Java como un Servicio Web.

## 6. Web Services

- Para exponer un Servicio Web desde una clase Java solamente debemos anotar la clase con `@WebService` y agregar la anotación `@WebMethod` al método a exponer como un Servicio Web. Esto en automático generará el WSDL y el esquema XSD asociado.

# 6. Web Services

## Rest Web Services

**REST:** Representational State Transfer

**Principios de una Arquitectura REST:**

- ✓ **Recursos Direccionables:** Los recursos pueden ser solicitados por medio un URI.
- ✓ **Orientados a Representaciones:** Clientes y Servidores intercambian representaciones, la cual puede ser en XML, JSON, entre otros.
- ✓ **Interfaces Restringida:** Podemos utilizar las operaciones básicas HTTP: GET, POST, PUT y DELETE, esto es similar a SQL: SELECT, INSERT, UPDATE y DELETE respectivamente.

## 6. Web Services

- Los SOAP Web Services utilizan HTTP como el mecanismo de transporte. Una petición GET o POST es ejecutada y un bloque de código XML es enviado al servidor. La URL que identifica al Servicio Web NO necesariamente indica qué tipo de operación debe realizarse del lado del servidor.
- Los RESTful Web Services, por otro lado, se basan completamente en las operaciones soportadas por el protocolo HTTP para ejecutar la funcionalidad del lado del servidor. Cada llamada al Servicio Web, debe utilizar alguno de los siguientes métodos HTTP: GET, POST, PUT, DELETE, URL, HEAD u OPTIONS.

## 6. Web Services

- REST significa Representational State Transfer y nació por la necesidad de simplificar la creación de Web Services utilizando el protocolo HTTP como base. REST es una forma "ligera" y rápida de desarrollar y consumir Web Services. Debido a que el protocolo HTTP es utilizado prácticamente en todos lados donde utilicemos la Web, es posible reutilizar este mecanismo de comunicación como la base para la transmisión de Servicios Web.
- La forma de crear un Web Services utilizando REST es a través de recursos (resources). Cada recurso tiene asociado una URI, y cada URI representa a su vez una operación del Web Service.

## 6. Web Services

- Ej: **/personas** - Es una URI que representa todas las entidades de tipo Persona de nuestro sistema.
- Ej: **/personas/{id}** - Esta URI representa una orden en particular. A partir de esta URI, podremos leer (read), actualizar (update) y eliminar (delete) objetos de tipo Persona.

# 6. Web Services

## Métodos HTTP

Método HTTP	Significado en Restful Web Services
GET	Se utiliza para operaciones de sólo lectura. No generan ningún cambio en el servidor.
DELETE	Elimina un recurso en específico. Ejecutar esta operación múltiples veces no tiene ningún efecto.
POST	Cambia la información de un recurso en el servidor. Puede o no regresar información.
PUT	Almacena información de un recurso en particular. Ejecutar esta operación múltiples veces no tiene efecto, ya que se está almacenando la misma información sobre el recurso.
HEAD	Regresa solo el código de respuesta y cualquier cabecera HTTP asociado con la respuesta.
OPTIONS	Representa las opciones disponible para establecer la comunicación en el proceso de petición/respuesta de una URI.

## 6. Web Services

- El protocolo HTTP está definido por el consorcio [www.w3.org](http://www.w3.org). Un conocimiento detallado de este protocolo para el estudio de Web Services no es necesario, sin embargo conforme se involucren cada vez más en la creación de Rest Web Services, más se deberá conocer acerca de este protocolo.
- HTTP maneja 8 métodos, los cuales son: GET, DELETE, POST, PUT, HEAD, OPTIONS, TRACE y CONNECT. Cualquier petición enviada hacia un Servicio Web debe especificar cualquiera de los 6 métodos HTTP listados en la tabla. Se excluyen los métodos TRACE y CONNECT ya que no son relevantes para los Web Services de tipo RESTful.

## 6. Web Services

- **GET**: Se utiliza para operaciones de sólo lectura. No generan ningún cambio en el servidor.
- **DELETE**: Elimina un recurso en específico. Ejecutar esta operación múltiples veces no tiene ningún efecto.
- **POST**: Cambia la información de un recurso en el servidor. Puede o no regresar información.
- **PUT**: Almacena información de un recurso en particular. Ejecutar esta operación múltiples veces no tiene efecto, ya que se está almacenando la misma información sobre el recurso.
- **HEAD**: Regresa solo el código de respuesta y cualquier cabecera HTTP asociado con la respuesta.
- **OPTIONS**: Representa las opciones disponible para establecer la comunicación en el proceso de petición/respuesta de una URI.

## 6. Web Services

- Con estos métodos debemos tomar en cuenta dos características muy importantes. Los métodos *seguros* e *idempotentes*.
  - Los métodos *seguros* (*no modifican el estado del sistema*) son aquellos que no hace otra tarea que recuperar información, por ejemplo los métodos GET y HEAD.
  - Los métodos *idempotentes* son aquellos que siempre se obtiene el mismo resultado, sin importar cuantas veces se realice cierta operación. Métodos que son *idempotentes* son: GET, HEAD, PUT y DELETE. El resto de los métodos no son ni *seguros* ni *idempotentes*.

## 6. Web Services

- Una gran ventaja de que las operaciones REST sean sobre el protocolo HTTP es que los administradores de servidores, redes y encargados de seguridad de las mismas, saben como configurar este tipo de tráfico, por lo que no es algo nuevo para ellos.
- Aunque a nivel de programación es posible realizar más de una operación al momento de enviar una petición, por ejemplo, actualizar y eliminar, esto NO debería programarse de esta manera, ya que rompe con la idea básica de los REST Web Services.

# 6. Web Services

## Request Headers (Cabeceros de Petición)

Los Cabeceros de Petición permiten obtener metadatos de la petición HTTP, como pueden ser:

- El Método HTTP utilizado en la petición ( GET, POST, etc. )
- La IP del equipo que realizó la petición ( 192.168.1.1 )
- El dominio del equipo que realizó la petición (www.midominio.com)
- El recurso solicitado ( http://midominio.com/recurso )
- El navegador que se utilizó en la petición (Mozilla, MSIE, etc.)
- Entre varios cabeceros más...

## 6. Web Services

- Al enviar y recibir mensajes con REST Web Services, es necesario tener conocimiento de códigos de estado. Algunos de los códigos de estado más utilizados son:
  - 200 (Ok): Significa que la respuesta fue correcta, este el código de estado por default.
  - 204 (Sin Contenido): El navegador continua desplegando el documento previo.
  - 301 (Movido Permanentemente): El documento solicitado ha cambiado de ubicación, y posiblemente se indica la nueva ruta, en ese caso el navegador se redirecciona a la nueva página de manera automática.
  - 302 (Encontrado): El documento se ha movido temporalmente, y el navegador se mueve al nuevo url de manera automática.

## 6. Web Services

- 401 (Sin autorización): No se tiene permiso para visualizar el contenido solicitado, debido a que se trató de acceder a un recurso protegido con contraseña sin la autorización respectiva.
- 404 (No encontrado): El recurso solicitado no se encuentra alojado en el servidor Web.
- 500 (Error Interno del Servidor Web): El servidor web lanzó una excepción irrecuperable, y por lo tanto no se puede continuar procesando la petición.
- Para una lista completa de los códigos de estado del protocolo HTTP se puede consultar el siguiente link:
  - [http://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_status\\_codes](http://en.wikipedia.org/wiki/List_of_HTTP_status_codes)

# 6. Web Services

## Cabeceros de Respuesta y Tipos MIME

Los cabeceros de respuesta se utilizan para indicar al navegador Web cómo debe comportarse ante una respuesta de parte del servidor Web

Un ejemplo común es generar hojas de Excel, PDF's, Audio, Video, etc, en lugar de solamente responder con texto.

Para indicar el tipo de respuesta se utilizan los tipos MIME (Multipurpose Internet Mail Extensions)

Los tipos MIME son un conjunto de especificaciones con el objetivo de intercambiar archivos a través de Internet como puede ser texto, audio, video, entre otros tipos.

## 6. Web Services

- Los tipos MIME(**Multipurpose Internet Mail Extensions**) son el estándar de internet para definir el tipo de información que recibirá el cliente (navegador Web) al realizar una petición al servidor Web.
- Los REST Web Services pueden devolver distintos tipos de contenido para un mismo recurso, por ejemplo:
  - ✓ **application/xml**
  - ✓ **application/json**
  - ✓ **text/html**
  - ✓ **text/plain**
  - ✓ **application/octet-stream**
  - **Mensaje XML**
  - **Mensaje JSON**
  - **Salida HTML**
  - **Salida en texto plano**
  - **Datos binarios**

## 6. Web Services

- Lo anterior son los tipos de recursos que más se utilizan para el envío de información en REST. Lo más común es utilizar xml, sin embargo no está limitado a este tipo de información, ya que si estamos utilizando un cliente con Jquery y AJAX, es común que utilicemos la anotación de JSON en lugar de XML.
- Un listado más completo de tipos MIME es el siguiente:

## 6. Web Services

- Un listado más completo de tipos MIME es el siguiente:
  - **application/msword:** Microsoft Word document
  - **application/pdf:** Acrobat (.pdf) file
  - **application/vnd.ms-excel:** Excel spreadsheet
  - **application/vnd.ms-powerpoint:** Powerpoint presentation
  - **application/zip:** Zip archive
  - **audio/x-wav:** Microsoft Windows sound file
  - **text/css:** HTML cascading style sheet
  - **text/html:** HTML document
  - **text/xml:** XML document
  - **image/gif:** GIF image
  - **image/jpeg:** JPEG image
  - **image/png:** PNG image
  - **video/mpeg:** MPEG video clip
  - **video/quicktime:** QuickTime video clip
- Para una lista más completa de tipos MIME, pueden consultar el siguiente link:
  - <http://www.freeformatter.com/mime-types-list.html>

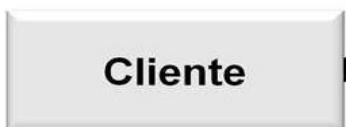
# 6. Web Services

## Representaciones con REST

### Representaciones REST

GET/personas

`http://localhost:8080/sistema-sga/ws/personas/35`



POST /personas/

```
<persona>
  <id>101</id>
  <nombre>Juan</nombre>
  <apePaterno>Perez</apePaterno>
  <email>jperez@mail.com</email>
</persona>
```

DELETE/personas/13

### Recursos



## 6. Web Services

- REST utiliza el concepto de Representaciones, y cada representación apunta a un Recurso(s) del lado del Servidor Java.
- Por ejemplo, para recuperar el objeto con id = 35, podemos utilizar el siguiente URI:
  - <http://localhost:8080/sistema-sga/ws/personas/35>
- Esto regresará la representación del objeto en el tipo MIME solicitado. Por ejemplo, si se solicitó una representación en XML, la respuesta debería ser:

```
<persona>
  <id>35</id>
  <nombre>Ernesto</nombre>
  <apePaterno>Campos</apePaterno>
  <email>ecampos@mail.com</email>
</persona>
```

## 6. Web Services

- De manera similar, podemos tener las operaciones básicas para agregar, modificar y eliminar recursos del lado del servidor. Por ejemplo, las siguientes URL son ejemplos para cada una de las acciones mencionadas:
  - **Agregar:** POST /personas/ - Sigue la URL y solicita agregar un nuevo recurso. Los datos se especifican en el documento XML a enviar. Ej.

```
<persona>
  <id>101</id>
  <nombre>Juan</nombre>
  <apePaterno>Perez</apePaterno>
  <email>jperez@mail.com</email>
</persona>
```

## 6. Web Services

- De manera similar, tenemos los ejemplos para modificar y eliminar:
  - **Modificar:** PUT /personas/123 – Sigue la solicitud de modificar el recurso 123 con un XML respectivo.
  - **Eliminar:** DELETE /personas/13 – Sigue la solicitud de eliminar el recurso 13

# 6. Web Services

## Anotaciones en JAX-RS

```
@Path("/personas")
public class PersonaServiceRS {

    @GET
    @Produces("application/xml , application/json")
    public List<Persona> listarPersonas(){..}

    @GET
    @Produces("application/xml")
    @Path("{id}") //hace referencia a /personas/{id}
    public Persona encontrarPersonaPorId(@PathParam("id") int id){..}

    @POST
    @Produces("application/xml")
    @Consumes("application/xml")
    public Response agregarPersona(Persona persona){..}

    @PUT
    @Produces("application/xml")
    @Consumes("application/xml")
    @Path("{id}")
    public Response modificarPersona(@PathParam("id") int id, Persona personaModificada){..}

    @DELETE
    @Path("{id}")
    public Response eliminarPersonaPorId(@PathParam("id") int id){..}
}
```

## 6. Web Services

- Como podemos observar en la figura, crear una clase que utilice JAX-RS para exponer un método como un servicio RESTfull Web Service en Java EE es muy simple. A continuación mencionaremos algunas de las anotaciones más utilizadas en JAX-RS, sin embargo, existen más anotaciones dependiendo del requerimiento a cubrir.
  - **@Path:** Esta anotación debe aparecer al inicio de la clase o en un método, e indica que esta clase/método se expondrá como un Servicio Web. Además, define la URI inicial del Servicio Web, la cual es relativa a la aplicación Web.

# 6. Web Services

- **@GET, @POST, @PUT y @DELETE**: Estas anotaciones se agregan a los métodos. Cada anotación representa el tipo de método HTTP que se va a utilizar. GET se utiliza para leer información, POST para agregar/modificar información. PUT se utilizar para agregar/modificar información y DELETE se utiliza para eliminar un elemento. En nuestro caso utilizaremos POST para agregar un nuevo recurso y PUT para modificar un recurso.
- **@PathParam**: Para especificar parámetros se utilizan los signos { }. Los parámetros se adjuntan al método utilizando la anotación **@PathParam**. Puede haber múltiples parámetros.
  - Ej.     **@Path**("/personas/{tipo}/{id}")

# 6. Web Services

- **@QueryParam**: Permite procesar los parámetros del URL. Para especificar parámetros HTTP se agregan después del signo ?, y para agregar varios se utiliza el signo &.
  - Ej.  
<http://localhost:8080/webservice/personas?fechalinicio=01012012&fechafin=31122012>
- **@Produces**: Indica el tipo MIME que enviará al cliente y se debe especificar por cada método.
  - Por ejemplo: [@Produces\({"application/json", "application/xml"}\)](#)

## 6. Web Services

- @Consumes: Indica el tipo MIME que puede aceptar. Por ejemplo, en el caso de insertar una nueva Persona, podemos aceptar un mensaje XML indicando lo siguiente en el método a procesar la petición: @Consumes("application/xml"). JAX-RS utilizará JAXB para convertir el documento XML en una clase Java, para ello la clase Java de tipo Persona deberá tener la anotacion @XMLElement al inicio de la clase.

# 6. Web Services

## Integración REST Web Services y una aplicación WEB

Configuración del archivo web.xml para integrar JAX-RS con una aplicación Web.

```
<!-- Configuración de JAX-RS -->
<servlet>

    <servlet-name>Jersey Web Application</servlet-name>
    <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>

    <init-param>
        <param-name>com.sun.jersey.config.property.packages</param-name>
        <param-value>mx.com.gm.sga.servicio.rest</param-value>
    </init-param>

    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>Jersey Web Application</servlet-name>
    <url-pattern>/webservice/*</url-pattern>
</servlet-mapping>
```

## 6. Web Services

- Si estamos utilizando las librerías del proyecto Jersey para desplegar REST Web Services, es necesario integrarlo con nuestra aplicación WEB, debido a que JAX-RS todavía no viene integrado de manera nativa con las aplicaciones Web.
- En la figura podemos observar la configuración necesaria del API del proyecto de Jersey para poder producir y consumir RESTful Web Services.
- Para que la aplicación web reconozca las URI de los Servicios Web respectivos, es necesario configurar el Servlet del API de Jersey. Además, se debe especificar el elemento <servlet-mapping> con el url-pattern a utilizar.

## 6. Web Services

- Por ejemplo, para solicitar el recurso persona id = 101, se debe utilizar el siguiente URL:
  - <http://localhost:8080/sistema-sga/webservices/personas/101>
- Podemos observar que el URI mostrado incluye el url-pattern configurado anteriormente. Sin esta configuración no es posible ejecutar los Servicios Web.

# 6. Web Services

## Integración EJB y JAX-RS

### Código Servicio REST y EJB

```
@Path("/personas")
@Stateless
public class PersonaServiceRS {

    @EJB
    private PersonaService personaService;

    @GET
    @Produces("application/xml , application/json")
    public List<Persona> listarPersonas(){
        return personaService.listarPersonas();
    }
}
```

### Código Clase Dominio Persona (API JAXB)

```
@XmlRootElement
public class Persona {

    private int idPersona;
    private String nombre;
    private String apePaterno;
    private String apeMaterno;
    private String email;
    private String telefono;

    public int getIdPersona() {
        return idPersona;
    }

    //Se omite el código restante
}
```

## 6. Web Services

- Como hemos visto los EJB proveen una serie de servicios como seguridad, transaccionalidad, entre otras características. Este simple hecho tiene varias ventajas sobre clases puras de Java.
- El API JAX-RS en combinación con los EJB hace muy simple la integración entre estas tecnologías y así exponer la funcionalidad de los EJB por medio de RESTful Web Services.

## 6. Web Services

- Lo que sí se debe planear con cuidado son los URI a utilizar, ya que de esto dependerá la interface que utilizará el cliente para poder consumir los RESTful Web Services.
- Como podemos observar en la figura, para exponer la funcionalidad de un método EJB podemos crear una clase enfocada a exponer únicamente los métodos de los EJB que necesitemos. Sin embargo, esta clase a su vez debe ser un EJB de tipo Stateless para poder injectar la funcionalidad de los EJB que se deseé.

## 6. Web Services

- Lo que sí se debe planear con cuidado son los URI a utilizar, ya que de esto dependerá la interface que utilizará el cliente para poder consumir los RESTful Web Services.
- Como podemos observar en la figura, para exponer la funcionalidad de un método EJB podemos crear una clase enfocada a exponer únicamente los métodos de los EJB que necesitemos. Sin embargo, esta clase a su vez debe ser un EJB de tipo Stateless para poder injectar la funcionalidad de los EJB que se deseé.

## 6. Web Services

- Una vez realizada la inyección de dependencia del EJB, ya podemos utilizar los métodos del EJB, por ejemplo para desplegar el listado de personas.
- Sin embargo, debido a que estamos regresando objetos de tipo Persona, esta clase se convierte de código Java a XML utilizando el API de JAXB. Para ello es necesario agregar la anotación `@XmlRootElement` a la clase Persona.
- Con la configuración anterior ya tenemos listas nuestras clase Java para exponer funcionalidad de nuestros EJB como RESTful Web Services.

# 6. Web Services

## Descripción de REST Web Service

Documento WADL:

Ej. <http://localhost:8080/sistema-sga/webservice/application.wadl>



```
<application xmlns="http://wadl.dev.java.net/2009/02">
  <doc xmlns:jersey="http://jersey.java.net/" jersey:generatedBy="Jersey: 1.11 12/09/2011 10:27 AM"/>
  <grammars>
    <include href="application.wadl/xsd0.xsd">
      <doc title="Generated" xml:lang="en"/>
    </include>
  </grammars>
  <resources base="http://localhost:8080/sistema-sga/webservice/">
    <resource path="/personas">
      <method id="listarPersonas" name="GET">
        <response>
          <nsl:representation xmlns:nsl="http://wadl.dev.java.net/2009/02" xmlns="" element="persona" mediaType="application/xml"/>
        </response>
      </method>
      <method id="agregarPersona" name="POST">
        <request>
          <nsl:representation xmlns:nsl="http://wadl.dev.java.net/2009/02" xmlns="" element="persona" mediaType="application/xml"/>
        </request>
        <response>
          <representation mediaType="application/xml"/>
        </response>
      </method>
      <resource path="{id}">
        <param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="id" style="template" type="xs:int"/>
        <method id="encontrarPersonaPorId" name="GET">
          <response>
            <nsl:representation xmlns:nsl="http://wadl.dev.java.net/2009/02" xmlns="" element="persona" mediaType="application/xml"/>
          </response>
        </method>
      </resource>
    </resource>
  </resources>
</application>
```

## 6. Web Services

- Cuando creamos SOAP Web Services, el documento WSDL era el elemento central para describir el Web Service a utilizar.
- De manera similar, con RESTful Web Services tenemos un documento conocido como WADL ( Web Application Description Language). Este documento XML permite describir los RESTful Web Services de manera muy similar a un documento WSDL.

## 6. Web Services

- Este tipo de documento XML está en sus primeras etapas de desarrollo, y no ha sido adoptado como un estándar, a diferencia de WSDL, sin embargo permite autodocumentar el Servicio WEB así como el XSD asociado a dicho Web Services.
- Esto tiene varias ventajas, por ejemplo, existe una herramienta para generar el código del lado del cliente.

```
– java -jar wadl2java.jar -o gen-src -p  
com.personas.rest  
http://localhost:8080/sistema-  
sga/webservice/application.wadl
```

# 6. Web Services

## Descripción de REST Web Service

Documento XSD para validar el mensaje XML del Servicio Web:

Ej. <http://localhost:8080/sistema-sga/webservice/application.wadl/xsd0.xsd>



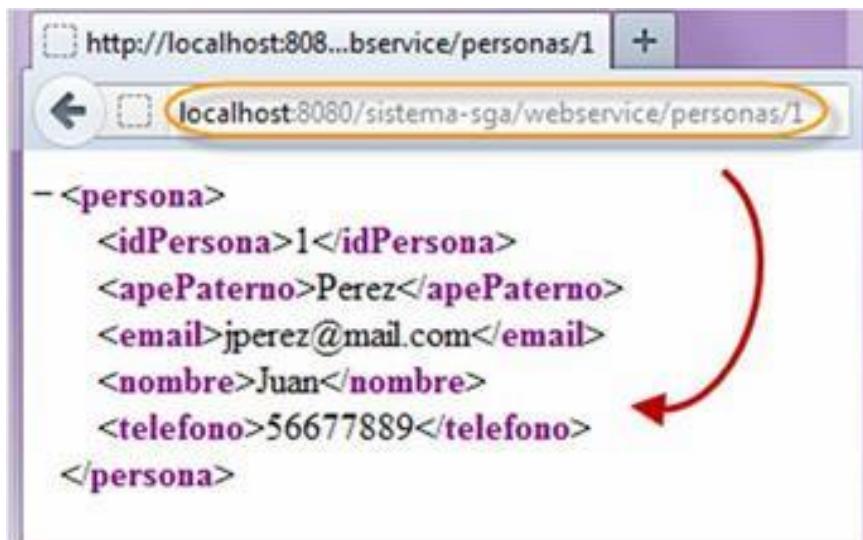
```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0">
  <xs:element name="persona" type="persona"/>
  <xs:complexType name="persona">
    <xs:sequence>
      <xs:element name="idPersona" type="xs:int"/>
      <xs:element name="apeMaterno" type="xs:string" minOccurs="0"/>
      <xs:element name="apePaterno" type="xs:string" minOccurs="0"/>
      <xs:element name="email" type="xs:string" minOccurs="0"/>
      <xs:element name="nombre" type="xs:string" minOccurs="0"/>
      <xs:element name="telefono" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

## 6. Web Services

- Como parte de la autodocumentación del Servicio Web, en el documento WADL se especifica el documento XSD que valida el documento XML a transmitir.
- Con el siguiente link es posible acceder a este documento.  
<http://localhost:8080/sistema-sga/webservice/application.wadl/xsd0.xsd>
- Como podemos observar en la figura, debido a que se está transmitiendo entidades de tipo Persona, es necesario validar que la información XML a transmitir sea válida conforme a este documento XSD.

## 6. Web Services

- Un ejemplo de un XML a transmitir es el siguiente:



A screenshot of a web browser window. The address bar shows the URL `localhost:8080/sistema-sga/webservice/personas/1`. The page content displays an XML document with the following structure and data:

```
- <persona>
  <idPersona>1</idPersona>
  <apePaterno>Perez</apePaterno>
  <email>jperez@mail.com</email>
  <nombre>Juan</nombre>
  <telefono>56677889</telefono>
</persona>
```

A red arrow points from the text "validar los mensajes XML a transmitir." in the list below to the closing tag of the XML document, "</persona>".

- Con el documento XSD mostrado en la lámina, es posible validar los mensajes XML a transmitir.

# 6. Web Services

## Cliente REST Web Service

```
public class TestClienteRS {  
    public static void main(String[] args) {  
        Client client = Client.create();  
        WebResource web = client.resource("http://localhost:8080/sistema-sga/webservice/personas/1");  
        Persona persona = web.get(Persona.class);  
        System.out.println("La persona es: " + persona.getNombre() + " " + persona.getApePaterno());  
    }  
}
```

Código del  
Cliente Java

```
@XmlRootElement  
public class Persona {  
  
    private int idPersona;  
    private String nombre;  
    private String apePaterno;  
    private String apeMaterno;  
    private String email;  
    private String telefono;  
  
    public int getIdPersona() {  
        return idPersona;  
    }  
}
```

Clase de  
Dominio  
Persona del  
Cliente

## 6. Web Services

- La ventaja de un servicio REST es que las peticiones GET el navegador WEB las soporta por default, así que al colocar la URL del recurso REST a buscar, obtendremos en automático la respuesta respectiva. Por ejemplo al colocar la siguiente URI y estar levantado el servidor de GlassFish o el Servicio Web ya publicado, obtendremos la respuesta siguiente:
  - <http://localhost:8080/sistema-sga/webservice/personas/1>

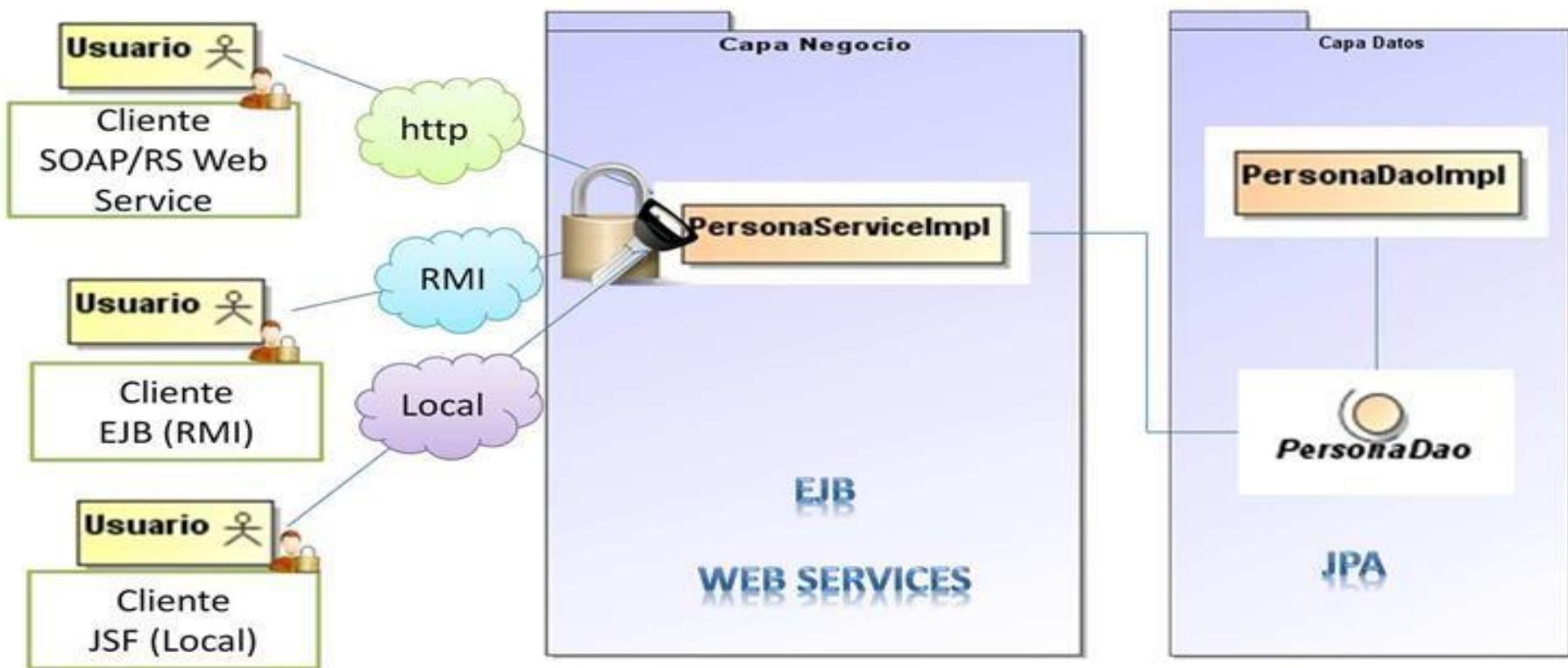
```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<persona>
    <idPersona>1</idPersona>
    <apePaterno>Perez</apePaterno>
    <email>jperez@mail.com</email>
    <nombre>Juan</nombre>
    <telefono>56677889</telefono>
</persona>
```

# Índice

- 
1. Introducción Java Empresarial
  2. Enterprise Java Beans (EJBs) y Context Directory Inyection (CDI)
  3. Java Persistence Api (JPA)
  4. Servlets y jsps en Java EE
  5. JavaServer Faces
  6. Web Services (SOAP y REST)
  7. **Seguridad en JEE**

# 7. Seguridad en JEE

## Seguridad en Java EE



## 7. Seguridad en JEE

- Uno de los temas cruciales al momento de desarrollar aplicaciones empresariales Java es el tema de seguridad.
- Ya sea que nuestra aplicación sea consultada a través de una interface Web utilizando Servlets o JSF, que expongamos la funcionalidad utilizando Web Services SOAP o REST, o que directamente un Cliente Swing tenga acceso a nuestros EJB, es necesario establecer los mecanismos que garanticen la seguridad de la información del sistema.

## 7. Seguridad en JEE

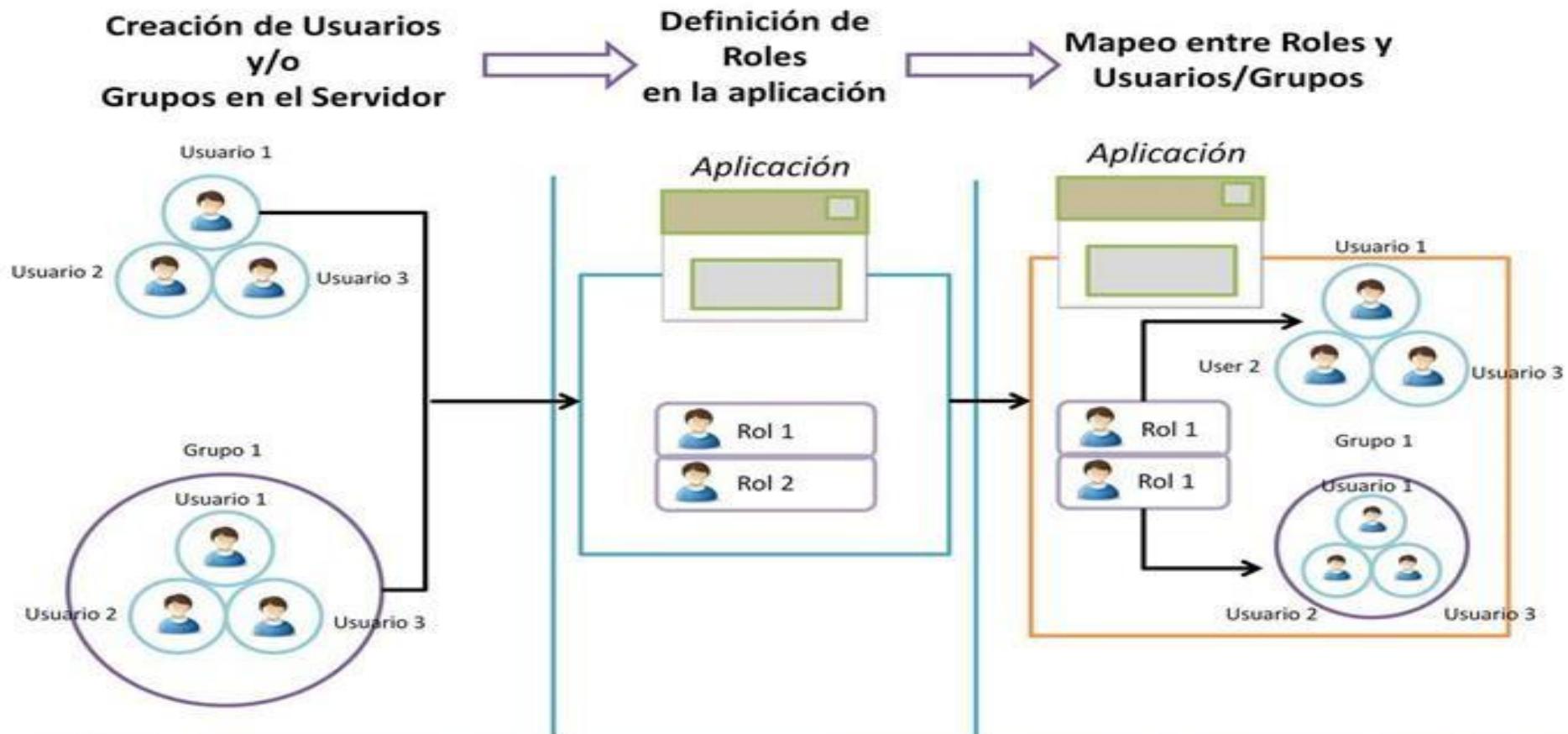
- La seguridad se debe asegurar desde el punto de vista de la vista y la capa de negocio, debido a que un usuario que no pueda visualizar una página, no significa que no pueda ejecutar un método de la capa de negocio.
- Uno de los beneficios de utilizar EJB, es la simplicidad y flexibilidad para agregar seguridad en la ejecución de sus métodos. Similar al manejo de transacciones, el manejo de Seguridad contamos con la Seguridad Declarativa y la Seguridad Programática.

## 7. Seguridad en JEE

- Dos conceptos son básicos respecto a la seguridad de un sistema. Los conceptos de Autenticación y Autorización:
  - **Autenticación:** La autenticación es el **proceso de verificar la identidad del usuario**. Lo más común es identificarnos ante el sistema por medio de un usuario y contraseña. A estos datos se les conoce como credenciales. Formas más avanzadas de autenticación es por medio de autenticación biométrica, tales como: huellas dactilares, patrones de Iris y retina, detección de rasgos físicos como el rostro, etc. **La autenticación se debe ejecutar antes de la autorización.**
  - **Autorización:** Una vez que el usuario se ha autenticado en el sistema, el siguiente paso es verificar los permisos del usuario a ejecutar cierta funcionalidad del sistema. De tal manera, que sin importar que un usuario se haya autenticado correctamente, no necesariamente tendrá la autorización para ejecutar la funcionalidad deseada en el sistema.

# 7. Seguridad en JEE

## Usuarios, Grupos y Roles



## 7. Seguridad en JEE

- Los Usuarios, Grupos y Roles son 3 conceptos inter relacionados que conforman la base del esquema de seguridad en un sistema empresarial.
- Para simplificar la administración los usuarios son divididos por grupos. Los grupos son una partición lógica para la identificación de usuarios, los cuales pueden tener acceso a distintos recursos. Por ejemplo, una aplicación puede tener un grupo de "Administradores" compuesto de usuarios con permisos de eliminación de registros.

## 7. Seguridad en JEE

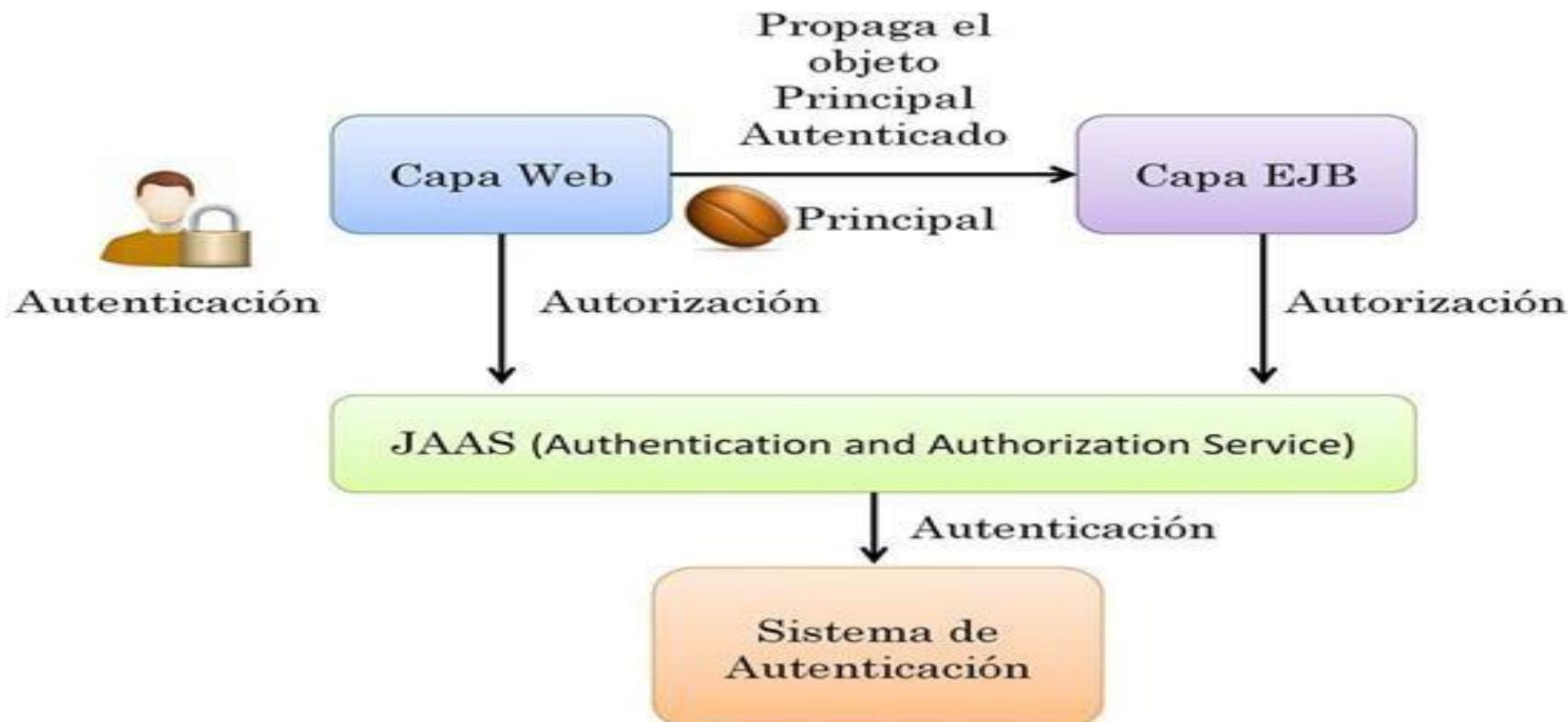
- Antes de ejecutarse cualquier operación en el sistema, la aplicación revisa si el usuario es un miembro del grupo y permite o deniega la ejecución del método/recurso solicitado.
- El manejo de **grupos** permite asignar las funciones comunes de los usuarios a una sola entidad, y de esta manera facilitar la administración de cada individuo.
- Un **rol**, es un concepto más ligado a la aplicación que se está desarrollando. En cambio, un grupo se define a nivel organizacional, ejemplo, en un Directorio Activo de Microsoft o un LDAP (Lightweight Directory Access Protocol). Un rol permite relacionar los grupos de una organización en entidades que entienda y procese correctamente la aplicación.

## 7. Seguridad en JEE

- De esta manera, los roles son una abstracción de los grupos enfocados en el servidor de aplicaciones Java. Un mapeo entre los roles y los grupos es comúnmente realizado. Si los nombres de los grupos y los roles son equivalentes, la mayoría de los servidores de aplicaciones pueden realizar el mapeo de manera automática, de lo contrario se debe configurar el servidor para realizar esta tarea.
- En caso que los nombres de los grupos y roles no coincidan, se debe especificar el mapeo de roles-grupos al servidor Java. Esto se puede hacer por medio de configuración del mismo servidor o por medio de archivos de configuración. Por ejemplo, en el caso de GlassFish se debe agregar el archivo **glassfish-web.xml**

# 7. Seguridad en JEE

## Seguridad en Java EE



## 7. Seguridad en JEE

- La seguridad en Java EE está basada ampliamente en el API de **JAAS (Authentication and Authorization Service)**. Como su nombre lo indica, esta API es responsable del **proceso de autenticación de los usuarios y autorización de recursos de los sistemas Java**, sobre todo enfocado en agregar seguridad en la capa Web y la capa de negocio donde se encuentran los EJB.
- Una vez que el usuario del sistema es autenticado, el contexto de la autenticación es propagado a través de las distintas capas de una aplicación empresarial, siempre que sea posible. Esto se realiza para que evitar el proceso de autenticación por cada una de las capas.

## 7. Seguridad en JEE

- Una vez que el usuario se ha autenticado, el API de seguridad JAAS crea un objeto conocido como ***Principal***. Este objeto es propagado entre las capas con el objetivo de ya no solicitar nuevamente la autenticación del usuario.
- Un usuario se puede autenticar por medio de una aplicación Web, a su vez la capa Web recupera la información y autentica al usuario, creando el objeto ***Principal*** en caso de una autenticación exitosa.
-

## 7. Seguridad en JEE

- El objeto ***Principal*** está asociado con uno o más roles. El sistema de seguridad revisa por cada acción, ya sea en la capa Web o en la capa de EJBs, que se tengan los permisos para ejecutar dicho recurso.
- El objeto ***Principal*** es enviado de manera transparente entre las capas Web y EJB según sea necesario.

# 7. Seguridad en JEE

## Asegurando la Capa Web

```
<!-- Configuración Seguridad del Sistema SGA -->
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Login in</realm-name>
</login-config>

<security-constraint>
    <web-resource-collection>
        <web-resource-name>Aplicación WEB JSF</web-resource-name>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>

    <auth-constraint>
        <role-name>ROLE_ADMIN</role-name>
        <role-name>ROLE_USER</role-name>
    </auth-constraint>
</security-constraint>

</web-app>
```

## 7. Seguridad en JEE

- La seguridad en la capa Web se realizar a través del archivo descriptor ***web.xml***. Existen distintos tipos de autenticación los cuales son HTTP Basic, HTTP DIGEST, HTTPS Client-Cert y FORM based.
- Según observamos en el código de la figura, el primer elemento que agregamos es <login-config>, el cual nos permite especificar la forma en que el contenedor Web recuperará la información para poder autenticar a los usuarios.
-

## 7. Seguridad en JEE

- Posteriormente tenemos el elemento <security-constraint>, el cual nos permitirá especificar los recursos (URL) a las cuales agregaremos seguridad, así como los Roles que participarán en la revisión de la seguridad. Es posible especificar incluso el tipo de Método HTTP que se permitirá (ej. GET, POST, PUT, DELETE, etc).
- El tipo de autenticación mostrado es de tipo **BASIC**, el cual el navegador Web mostrará un pop-up genérico solicitando el usuario y password para poder autenticarse al sistema. Otro método muy comúnmente utilizado es el método basado en una forma HTML (**FORM based**). Esta configuración tiene la ventaja de que es posible personalizar la página de autenticación que se muestra al usuario.

## 7. Seguridad en JEE

- Posteriormente el real-name. Un realm es una abstracción del servidor de aplicaciones, donde se especifican las políticas de seguridad del sistema.
  - Un realm contiene una colección de usuarios, los cuales pueden o no ser asociados a un grupo.
  - Un realm se configura utilizando las herramientas de administración del servidor de aplicaciones en cuestión.
  - Existen realm basados en un archivo (file), en JDBC, LDAP, y podemos agregar realms personalizados, ya que en ocasiones utilizaremos sistemas externos para realizar el proceso de autenticación.

## 7. Seguridad en JEE

- El elemento <security-constraint> nos permite especificar una o más colecciones de URL que deseamos agregar seguridad. Por medio de un url-pattern, especificamos las páginas a asegurar. Finalmente con el elemento <auth-constraint> especificamos los roles permitidos para ejecutar el recurso especificado.
- De esta manera hemos agregado tanto el concepto de Autenticación como el concepto de Autorización para una aplicación Web.

# 7. Seguridad en JEE

## Seguridad en JSF y PrimeFaces

Opciones para restringir componentes en una página JSF con la extensión de PrimeFaces son:

```
#{p:ifGranted('ROLE_ADMIN')}  
#{p:ifAllGranted('ROLE_ADMIN, ROLE_EDITOR')}  
#{p:ifAnyGranted('ROLE_USER, ROLE_ADMIN')}  
#{p:ifNotGranted('ROLE_GUEST')}  
#{p:remoteUser()}  
#{p:userPrincipal()}
```

Algunos ejemplos de su uso en las páginas JSF son:

```
<h:commandButton value="Eliminar Persona" rendered="#{p:ifGranted('ROLE_ADMIN')}" />  
  
<p:commandButton value="Reporte General" disabled="#{p:ifNotGranted('ROLE_USER, ROLE_ADMIN')}" />
```

## 7. Seguridad en JEE

- Al autenticarnos correctamente por medio de la aplicación Web, JSF y en particular la extensión de PrimeFaces, ha agregado varios elementos que nos permiten agregar muy fácilmente seguridad a nuestras páginas JSF.
- De tal manera que podremos habilitar o deshabilitar funcionalidad, incluso a nivel de componentes, botones, links, tablas, etc, dependiendo del rol del usuario que se ha autenticado.

# 7. Seguridad en JEE

## Tipos de Seguridad en Java EE

En Java EE 6, existen 2 tipos de seguridad:

- ✓ **Seguridad Declarativa:** Indicamos al contenedor el tipo de validación deseada, a través de anotaciones o archivos de configuración xml.

El contenedor se hace cargo de la mayoría de las tareas de validación, autenticación y autorización.

- ✓ **Seguridad Programática:** Existen situaciones en las que necesitamos un mayor control sobre la forma en que se realiza la autenticación y/o autorización, por ejemplo, a nivel usuario o grupo.

La seguridad programática se puede combinar con la programación declarativa para incrementar el control sobre los requerimientos de seguridad en el sistema.

## 7. Seguridad en JEE

- En **la seguridad declarativa**, es posible aplicarla a nivel de la clase o a nivel de cada método. El contenedor verifica el rol asignado del usuario autenticado previamente, y si cuenta con el rol solicitado, entonces permite ejecutar el método del EJB respectivo.
- Para manejar la Seguridad Declarativa es muy común que utilicemos anotaciones.
- La seguridad programática nos sirve cuando tenemos requerimientos más detallados, y que serían complicados o imposibles de lograr con la Seguridad Declarativa.

## 7. Seguridad en JEE

- Podemos combinar la Seguridad Declarativa en conjunto con la Seguridad Programática, de tal manera que en los casos que necesitemos realizar validaciones más detalladas, incluso por cada usuario, podremos combinar el poder de ambas opciones de seguridad.

# 7. Seguridad en JEE

## Anotaciones en la capa EJB

Para la Seguridad Declarativa tenemos las siguientes Anotaciones disponibles en los componentes EJB:

- ✓ **@DeclareRoles:** Esta anotación lista los roles que se utilizarán en el EJB. Solamente se puede utilizar a nivel de la clase.
- ✓ **@RolesAllowed:** Permite ejecutar los métodos del EJB siempre y cuando los roles se encuentren listados en esta anotación. Se puede definir al nivel de la clase o a nivel del método.
- ✓ **@PermitAll:** Como su nombre lo indica, permite a cualquier usuario ejecutar el método EJB anotado.
- ✓ **@DenyAll:** Como su nombre lo indica, prohíbe a cualquier usuario ejecutar este método.
- ✓ **@RunAs:** Permite ejecutar el método como si el usuario tuviera otro rol, únicamente durante la ejecución de dicho método.

## 7. Seguridad en JEE

- Para la Seguridad Declarativa tenemos las siguientes Anotaciones disponibles en los componentes EJB:
  - **@DeclareRoles:** Esta anotación lista los roles que se utilizarán en el EJB. Solamente se puede utilizar a nivel de la clase. Si esta anotación no se proporciona, el contenedor buscará en los roles definidos por la anotación **@RolesAllowed** y construye una lista de roles. Se recomienda agregarla debido a que en algunos contenedores, al no agregarla, debemos agregar configuración extra para que detecte los roles que estamos manejando. Si el EJB extiende de otra clase, la lista de roles se concatena, es decir, también se incluyen.

## 7. Seguridad en JEE

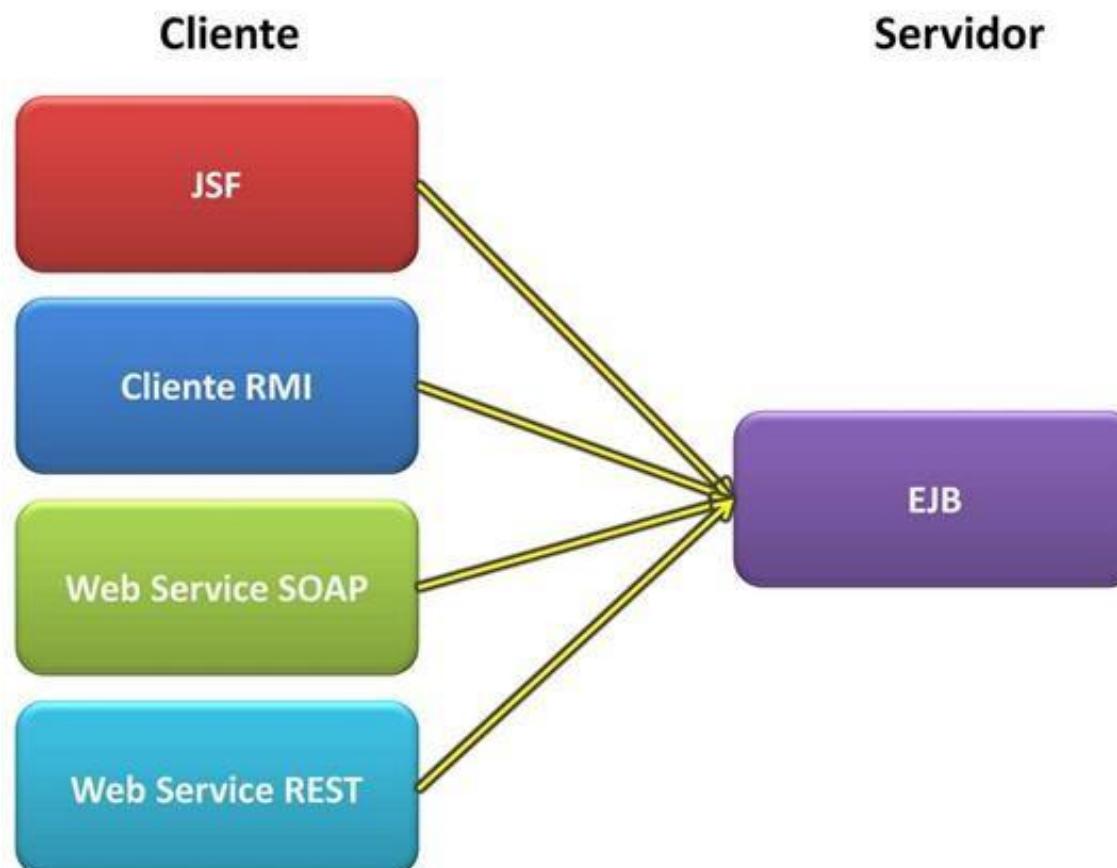
- **@RolesAllowed:** Permite ejecutar los métodos del EJB siempre y cuando los roles se encuentren listados en esta anotación. Se puede definir al nivel de la clase o a nivel del método. Cuando se define al nivel de la clase, estamos indicando al contenedor que los roles listados podrán ejecutar cualquier método del EJB. Si agregamos esta anotación a nivel del método, sobreescrivimos cualquier comportamiento definido a nivel de la clase. Sin embargo debemos tener precaución al combinar los roles a nivel clase y nivel método para evitar que la mezcla o sobreescritura de roles se convierta en un problema.
- **@PermitAll:** Como su nombre lo indica, permite a cualquier usuario ejecutar el método EJB anotado. Se debe utilizar esta anotación con precaución, ya que puede ser un hueco de seguridad si esta mal utilizado.

## 7. Seguridad en JEE

- **@DenyAll:** Como su nombre lo indica, prohíbe a cualquier usuario ejecutar este método. Esta anotación provoca que el método quede inusable, sin embargo para deshabilitar cierta funcionalidad del sistema puede ser una posible solución.
- **@RunAs:** Permite ejecutar el método como si el usuario tuviera otro rol, únicamente durante la ejecución de dicho método. Esto permite a un usuario con rol con menos privilegios, asignarle más privilegios, debido al cambio de rol, únicamente por la ejecutación del método en cuestión.

# 7. Seguridad en JEE

## Algunos tipos de Clientes en Java EE



## 7. Seguridad en JEE

- Los mecanismos de autenticación entre los distintos clientes al día de hoy no son estándar, y muchas de las veces dependemos todavía de la implementación del servidor Java que estemos utilizando para poder comunicar cada cliente y poder autenticarnos de manera exitosa con el servidor Java.
- El caso donde hay mayores pasos de configuración es el Cliente Web, ya que a su vez, según hemos visto, funciona como un punto de autenticación del sistema, por lo tanto la propagación del objeto Principal entre la capa Web y la capa EJB funciona de manera automática al utilizar la seguridad en Java EE.