# **TEMA: Framework para microservicios Thorntail**

# **INTEGRANTESTEMA:** Detección de incendios e inundaciones por imágenes satelitales.

# **INTEGRANTES**

* + - * Erazo Stephano
      * Moreta William
      * Zumba Gema

# **INTEGRANTES**

* + - * David Lara
      * Jaime Alcocer
      * Gema Zumba



**TEMA:** Detección de incendios e inundaciones por imágenes satelitales.

**UNIVERSIDAD CENTRAL DEL ECUADOR**

**INGENIERIA INFORMATICA**

**PROGRAMACIÓN DISTRIBUIDA**

**Fecha:***17/9/2020*

Contenido

[1. Marco teórico 3](#_Toc51207319)

[2. Desarrollo de las aplicaciones 9](#_Toc51207320)

[3. Configuración de Servidor de registro 24](#_Toc51207321)

[4. Configuración Spring Cloud Zookeeper 25](#_Toc51207322)

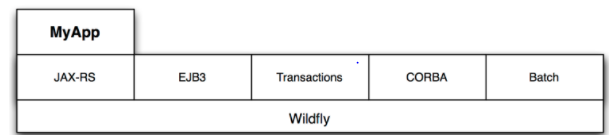
[5. Servidor de configuración Vault 27](#_Toc51207323)

# Marco teórico

**Thorntail**

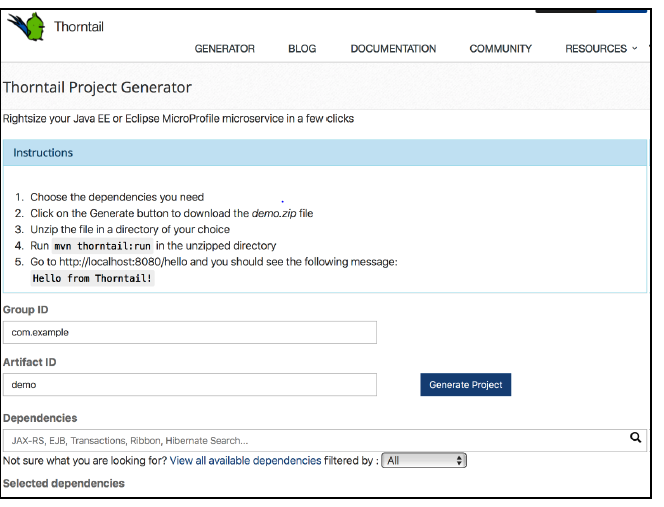
Red Hat desarrolló una versión propia de MicroProfile que cuenta con una herramienta de configuración, este framework originalmente se llamó WildFly Swarmy, finalmente fue designada con el nombre de Thorntail. En el sitio web de la herramienta se puede crear un archivo personalizado de compilación de Maven para un nuevo proyecto con arquitectura en microservicios, las dependencias de Maven se encargan de ensamblar todo. Thorntail elimina las partes de JEE que no son de utilidad y crea un archivo JAR listo para ser implementado. WildFly Thorntail está basado en el servidor de aplicaciones de Red Hat WildFly, trabaja de igual manera que con los servidores de aplicaciones monolíticos tradicionales, una aplicación se implementa sobre WildFly, ver figura.

Estructura de aplicación sobre Wildfly

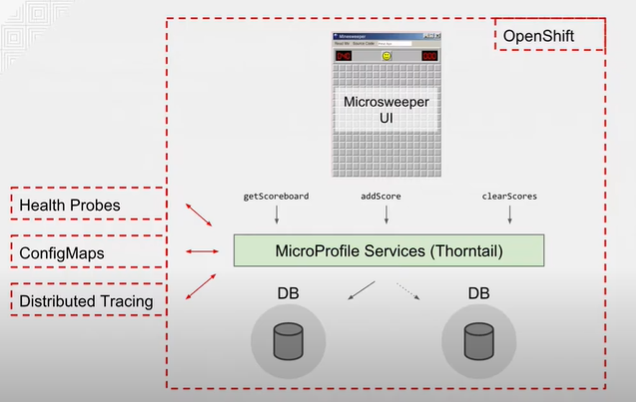


**Generator**

Es una herramienta que le permite crear archivos JAR ejecutables.

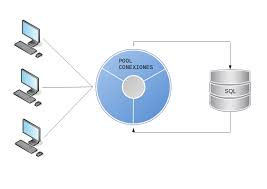


**Arquitectura básica**



**Pool de conexiones**

Un pool de conexiones nos permite agilizar el proceso de obtener una conexión por cada clase Java que necesita de una conexión, ya que el objetivo del pool de conexiones precisamente es tener listas varias conexiones a la base de datos, de manera que sea mucho más eficiente el proceso de obtener y liberar una conexión a base de datos.

El proceso de administración de una conexión se delega al pool de conexiones, por lo que podemos dejar de preocuparnos por abrir y cerrar una conexión, ya que, de este tipo de tareas, entre otras, se encargará el pool de conexiones.

**JPA**

Customer

Orders

Customer

Orders

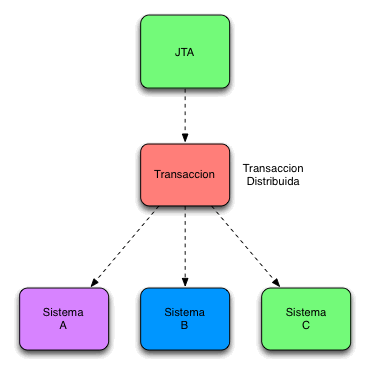


Esta tecnología de Java es el estándar de persistencia en Java, y se cuenta con varias implementaciones, tales como Hibernate, EclipseLink, OpenJPA, entre otras.



**Java Transaction API JTA**

JTA o Java Transaction API es la encargada de administrar las transacciones del lado del servidor, abstrayendo casi en su totalidad al programador de abrir o cerrar las transacciones y él se preocupa por la lógica de negocio aumentando drásticamente la productividad y los errores de Runtime



**JavaServerFaces “JSF”**

JavaServerFaces (JSF) es una tecnología que fue diseñada para simplificar la construcción de interfaz de usuario para aplicaciones web java. Es el marco de aplicaciones web estándar para de Java EE

**Características**

* **MVC:** Implementa el patrón de diseño Modelo-Vista-Controlador, proporcionando un enfoque orientado a eventos.
* **RAD:** Desarrollo rápido de aplicaciones para web.
* **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 otro tipo de clientes.
* **Extensibilidad:** JSF es altamente extensible debido a su arquitectura, permite crear nuevos componentes y existen nuevos framework adicionales como: no primefaces, richfaces, butterfaces, iceface.
* Manejo de condiciones por default más inteligentes, es decir ya no se requiere de un archivo de configuraciones como era en las primeras versiones, como **ocurre con faces-config.xml**
* Manejo de anotaciones para varias configuraciones

**Arquitectura**

**BD**

HTTP

JDBC

Capa cliente

Capa Web

(JSF)

Capa de

Negocio

Capa de datos

Objeto de Negocios

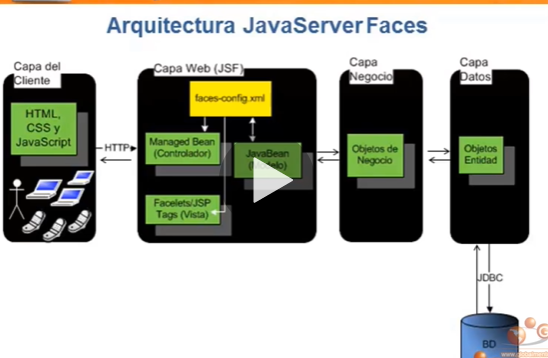
Objeto Entidad

HTML

CSS

Bootstrap

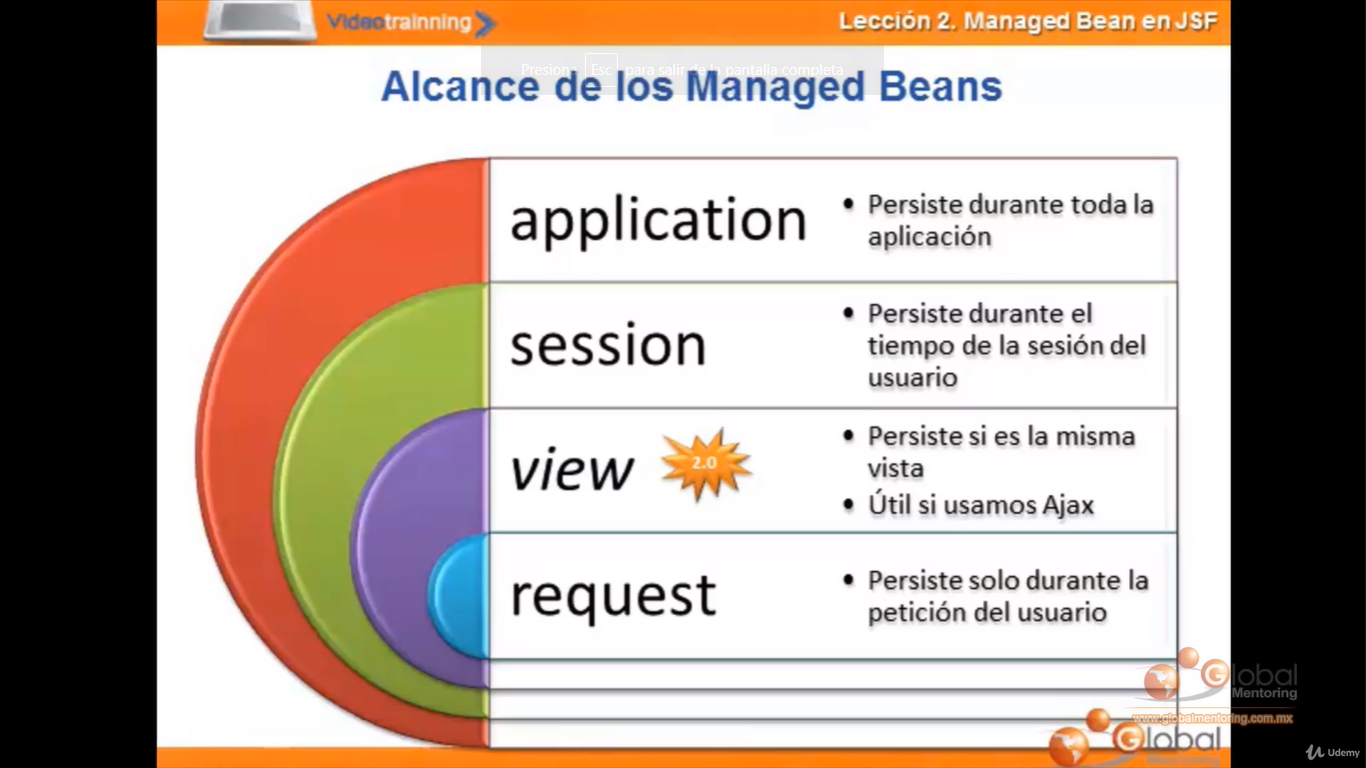
JavaScript



Managed Bean (Controlador)

JavaBean (Modelo)

Vista (Facelets o JSPs)



**Facelets**

Facelets es un marco de plantillas. La ventaja de Facelets es que JSF UIComponents están preintegrados con Facelets, y no requiere un archivo de configuración de Facelets.

El formato por omisión de los facelets es XHTML. Así, las extensiones de los archivos son de extensión .xhtml.

**Características**

* Los Facelets es la tecnología estándar de despliegue en JSF
* Los Facelets eliminan por completo la necesidad de los JSP’s
* Crea un componente tree más ligero con respeto a los JSP’s
* Soporte para templetes
* Creación de componentes compuestos

**Zookeeper**

Apache ZooKeeper es una herramienta para registrar, administrar y descubrir servicios que funcionan en diferentes máquinas. Es un miembro indispensable en la pila de tecnología cuando tenemos que lidiar con un sistema distribuido con muchos nodos que necesitan saber dónde se lanzan sus dependencias

ZooKeeper es de un nivel bastante bajo e incluso los casos de uso estándar requieren muchas líneas de código. Es por eso que nació Apache Curator , una biblioteca contenedora mucho más amigable y fácil de usar que ZooKeeper. Con Curator podemos entregar más con menos código y de una manera mucho más limpia

# Desarrollo de las aplicaciones

* 1. **Creación de la base de datos**

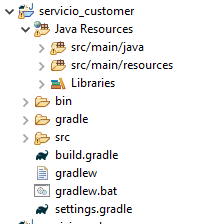
|  |  |  |
| --- | --- | --- |
| **Campo** | **Tipo** | **Descripción** |
| Id | Long | Id del cliente |
| Name | String | Nombre del cliente |
| suname | String | Apellido del cliente |

Para los servicios **REST** se creará una base de datos en PostgreSQL, la llamaremos **dbcustomerOrders** con los siguientes campos y tipo de dato.

|  |  |  |
| --- | --- | --- |
| **Campo** | **Tipo** | **Descripción** |
| id | long | Id de un producto |
| item | String | Producto |
| price | float | Precio |
| Customer\_id | long | Id del cleinte |

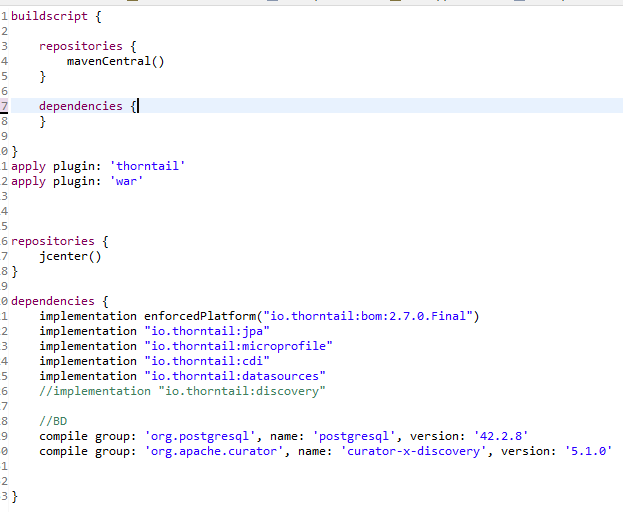
* 1. **Creación del servidor1 (servicios REST) para CRUD de la tabla Customer**

Crear un proyecto Gradle desde el editor de código con el nombre deseado, en este caso se le dará el nombre de **servicio\_customer**



* 1. **Instalación de las dependencias**

Se instalará las dependencias básicas para crear un servicio REST conforme a las especificaciones.

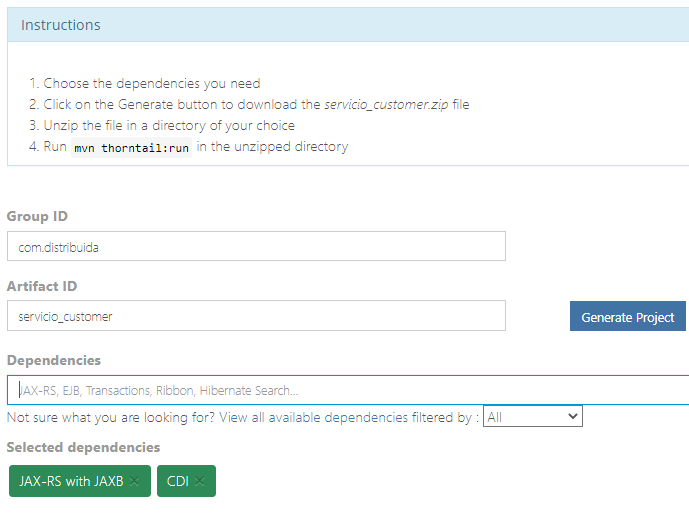


Mencionaremos la funcionalidad de algunas dependencias y de las demás se recomienda revisar la documentación[**https://thorntail.io/documentation/**](https://thorntail.io/documentation/)

implementation "io.thorntail:datasources" Permite la conexión con la base de datos y manejar el pool de conexiones, que lo realiza automáticamente.

implementation "io.thorntail:jpa" es el encargado de convertir los objetos Java en instrucciones para el Manejador de Base de Datos

Crear un proyecto de la página oficial de <https://thorntail.io/generator/> y seguir las instrucciones, ver imagen.



* 1. **Conexión con la base de datos**

La conexión de la base de datos se realiza mediante un archivo yml que trae por defecto Thorntail, **project-defaults.yml** se agrega lo siguiente

|  |
| --- |
| Configuración general de archivo. yml |
|  |
| Base de datos configurada |
|  |
| El archivo project-defaults.yml se debe ubicar en el directorio que se indica en la imagen, se debe crear la carpeta META-INF con el mismo nombre |
|  |

* 1. **Acceso a base de datos: JPA**

Para acceso a la base de datos debemos realizar una serie de configuraciones en el archivo persistence.xml. Ver las imágenes para realizar la siguiente configuración.

|  |  |
| --- | --- |
| Unidad de persistencia | CustomerPu |
| Tipo de transacción | Java Transaction API (JTA) |
| Es el nombre del data-sources que se encuentra en el archivo. yml y se debe seguir la estructura indicada | java:jboss/datasources/**customerDataSource** |

|  |
| --- |
| Configurar el archivo persistence.xml para administrar las transacciones a través de JPA. |
|  |
| **persistence.xml** ubicar en el siguiente directorio |
|  |

Hibernate.dialect, hibernate.show\_sql y hibernate.format\_sql que se encuentran en nuestro archivo de persistence.xml, permite mostrar en consola la sentencia **sql** que se realizó al momento de interactuar con la base de datos.

* 1. **Estructura general de proyecto de servicio REST para la tabla customer**

|  |  |
| --- | --- |
| **Proyecto customer** | **Descripción de cada paquete** |
|  | **paquete dto:** se encuentra la entidad customer  **paquete rest:** contiene la clase rest, donde se encuentran los métodos http básico del crudy **RestApplication** es la clase principal  **paquete services:** se encuentra la lógica de negocios  **En la carpeta resources:** están los archivos de configuración antes mencionados. |

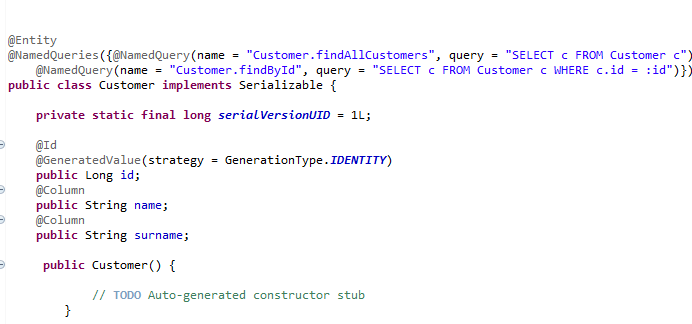
* 1. **Aplicación servidor2 (servicios REST) para** **CRUD de la tabla Orders**

Estructura general del servicio REST CRUD de la tabla Orders. Esta aplicación tiene la funcionalidad de realizar conexiones a la base de datos, mediante datasources y el mapeo de los datos se lo realiza mediante JPA, que se configura mediante un archivo de **persistence.xml** La aplicación del servidor2 tiene casi la misma estructura, tiene agregado una característica particular que se indicará más adelante. En esta aplicación se describen los métodos http que se utilizaron para realizar el crud.

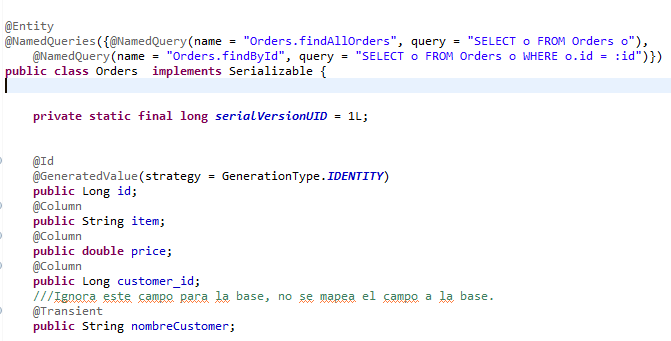
|  |  |
| --- | --- |
| **Estructura general del servicio ordenes** |  |
|  | **Paquete dto:** contiene las entidades Customer y Orders.  **Paquete proxi:** contiene los proxys para poder utilizar al servidor como cliente REST.  **Paquete service:** contiene la lógica de negocios  **Paquete rest:** contiene la clase principal y la clase Rest.  Carpeta META-INF contiene los archivos de configuración, beans.xml, persistence.xml y el archivo. properties  **Carpeta resources:** contiene el archivo. .yml que sirve para la conexión a la base de datos .  **El archivo build.gradle** permite instalar las dependencias y realizar ciertas configuraciones, eso depende del Framework para microservicios |
|  |  |

**Entidad Customer**

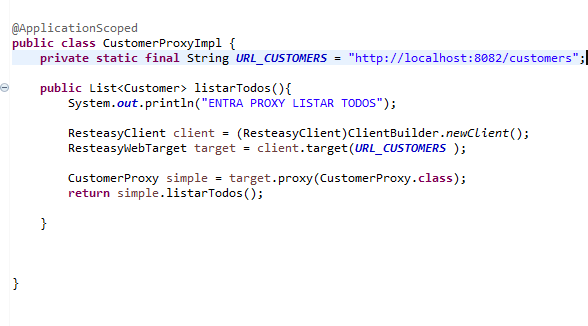
La clase **Customer** se realizó el mapeo mediante JPA, para realizar un correcto acceso a datos se debe usar la anotación @**Entity** y es opcional usar los @NamedQuery, la clave primaria de la tabla se debe usar la anotación **@Id** y si el id es número que se genera automáticamente se debe usar **@GeneratedValue(strategy= GenerationType.IDENTITY)** y las demás variables se anotan con @Column, ver la **imagen** para realizar correctamente todo lo mencionado. Además, se debe agregar los métodos setter y getter y constructores de ser necesario.



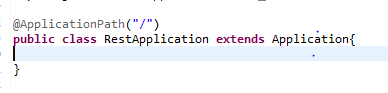
La entidad Orders tiene la misma característica de la entidad Customer para realizar el acceso a datos, con la diferencia de que tiene una variable auxiliar con **nombreCustomer**. Está anotada con @Transient, esta anotación permite ignorar el mapeo a dicho campo.



**La clase CustomerProxyImpl** permite listar todos los customers usando ResteasyClient. Tiene anotación CDI



**RestApplication** es la clase donde definimos el punto de entrada de nuestras peticiones REST y la anotación **@ApplicationPath("/");** con esto ya especificaremos que todas las peticiones vayan a la trayectoria indicada, ver imagen.

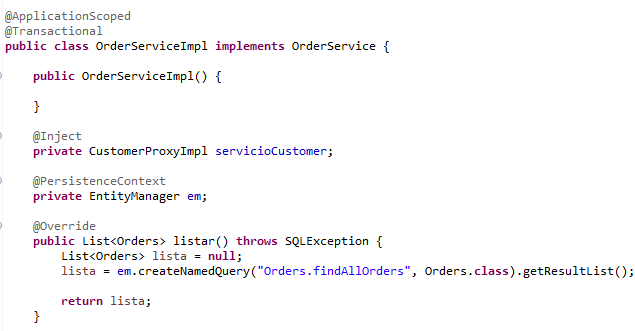


**La interfaz OrderService** contiene los métodos del crud; la clase OrderServiceImpl contiene la implementación de los métodos de la interfaz**,** como se ve en la imagen.

**@ApplicationScoped** esta anotación permite hacer uso de CDI

**@Transactional** permite realizar transacciones en operaciones que necesitan interactuar con la base de datos y realizar algún cambio.

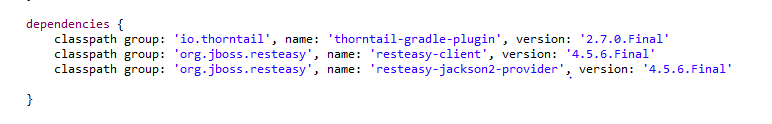
@Transactional se hace uso en la capa de servicio, ya que si se lo hace a nivel de DAO por cada DAO terminará la transacción y si un método de negocio usa varios daos para realizar su operación entonces terminará antes la transacción y eso ocasionará problemas de inconsistencia



Los archivos de la carpeta **META-INF** son los mismo que se explicó en el servidor1 Customer, por lo cual se obviará dicha explicación.

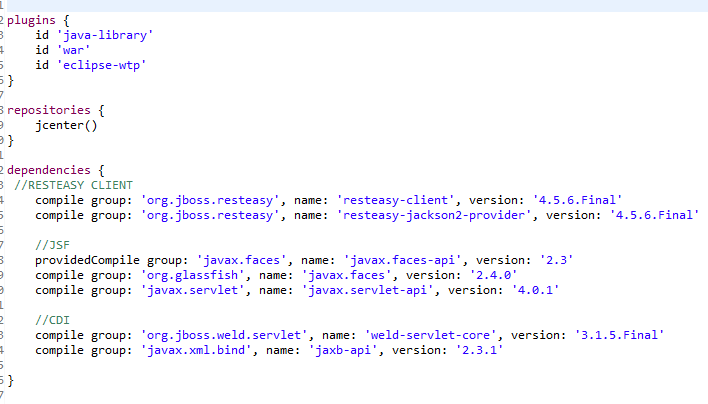
**Instalación de las dependencias**

Son la misma del servidor1, agregando las dependencias de un cliente REST, ver la imagen, se agregaron resteasy-jackson2-provider y resteasy-client



* 1. **Aplicación web cliente con JSF y Bootstrap**

**Instalación de las dependencias**

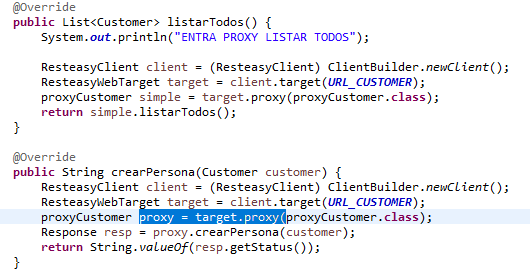


**Resteasy Client** permite realizar solicitudes http a sus servicios web RESTful remotos.

Resteasy Proxy Framework opuesto a la especificación del lado del servidor JAX-RS. En lugar de utilizar anotaciones JAX-RS para mapear una solicitud entrante a su método de servicio web RESTFul, el marco del cliente crea una solicitud HTTP que utiliza para invocar en un servicio web RESTful remoto.

En la imagen se indica dos métodos usando ResteasyClient, lo cual nos permite crear una solicitud al servidor Customer anteriormente mencionado, para obtener una mejor información sobre el uso de esta librería visitar la documentación oficial [https://docs.jboss.org/resteasy/docs/3.0-beta-3/userg uide/html/RESTEasy\_Client\_Framework.html](https://docs.jboss.org/resteasy/docs/3.0-beta-3/userg%09uide/html/RESTEasy_Client_Framework.html)

En el primer método tenemos el listar customer donde nos devuelve todos los customer que están en el servidor.



**Estructura general del cliente web**

|  |  |
| --- | --- |
|  | **Paquete controlador**: contiene el manejador de las vistas  **Paquete datos:** contiene las clases: Customer y Orders  **Paquete proxy:** contiene todos los proxys que hace referencia a clase REST del servidor.  **Paquete service:** contiene la lógica de negocios  **Carpeta webapp** contiene los templates para mostrar la parte visual  **Carpeta META-INF:** contiene el archivo web.xml |

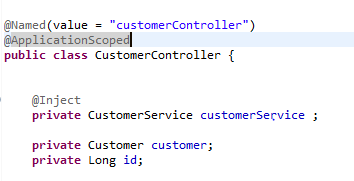
**Clase ControladorCustomer**

Las variables y anotaciones más importantes que se encuentran en la clase controladora

**@Named** se la utiliza en vez de @ManagedBean, debido a la versión de JSF, @ManagedBean se está eliminando gradualmente y el enfoque recomendado es usar @NamedEl alcance de la

**@ApplicationScoped** es anotaciones para objetos inyectados y se crea una vez durante aplicación, ver imagen de las anotaciones que se usa en el proyecto web.

**@Inject** realiza una inyección de dependencias a la interface CustomerService



En la clase también tenemos los métodos para hacer referencia en las plantillas

**La clase OrdersController** se maneja la lógica similar a la clase CustomerController

**Vista de la pagina web**

**Vista principal**

****

**Vista de Customer**

****

**Vista de Orders**

****

# Configuración de Servidor de registro

Para realizar la configuración del registro utilizamos herramienta ZooKeeper**, Curator**

**Pasos**

**Paso 1:** Descargar la herramienta del siguiente link:

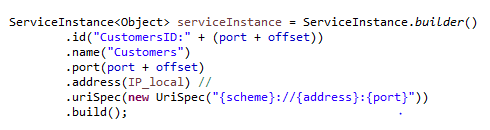
[**https://zookeeper.apache.org/releases.html#download**](https://zookeeper.apache.org/releases.html#download)

**Paso 2:** Instalar la dependencia en los servidores que desea que se registre, descargar del siguiente del link: <https://mvnrepository.com/artifact/org.apache.curator/curator-x-discovery>

**Paso 3:** Crear dos métodos en la clase principal de cada servidor, la clase es class **RestApplication extends Application**

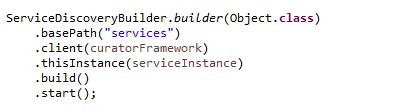


Creamos e iniciamos el cliente CuratorFramework envolviendo todas las operaciones que queremos realizar en la instancia de ZooKeeper, usamos localhost con el puerto predeterminado y se refresca cada cinco segundo.



En este fragmento de código hace referencia a los campos que por defecto necesitamos para realizar un registro de un servidor, cabe mencionar que tomamos obtenemos la IP de la máquina local

Se Crean las instancias que representa a nuestro trabajador. Pasamos todos los "datos de contacto" necesarios para llamar a nuestro servidor de registro



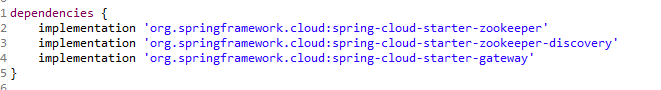
Registran nuestra instancia en un ZooKeeper representado por el cliente CuratorFramework

# Configuración Spring Cloud Zookeeper

El Gateway se realizó con Spring Cloud, para lo cual empezamos con las configuraciones necesarias

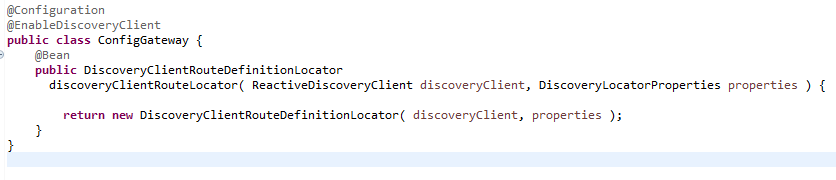
**Pasos**

**Paso 1:** Agreguemos las dependencias requeridas spring-cloud-starter-zookeeper-discovery , spring-web , spring-cloud-dependencies y spring-boot-starter a nuestro archivo build.gradle, ver la imagen y proceder a descargarlo.



Paso 2: Agregamos las anotaciones a la clase ConfigGateway que anteriormente se lo creo

@EnableDiscoveryClient. Esto hará que la aplicación esta hará que la aplicación sea visible , ver la imagen



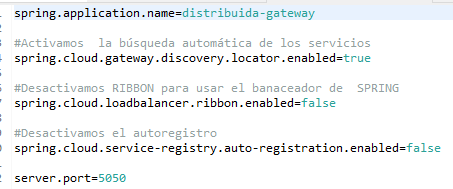
Debemos crear un controlador interactuar con el archivo **application.properties**

**Paso 3**: Configuraciones de archivo. properties

Creemos un archivo **properties** que se utilizará para configurar el nivel de registro de la aplicación e informar a Zookeeper que la aplicación está habilitada y puede ser escaneada.

El nombre de la aplicación con la que se registra en Zookeeper es importante. Más adelante en el consumidor de servicios, un cliente usará este nombre cuando se levante servicio.

En nuestro caso el nombre es **distribuida-gateway,** ver la imagen para obtener más información sobre que hace cada línea de código



**Paso 4:** Consumidor de servicios

Mediante la **url** que definimos, consumimos los servicios desde el cliente web

# Servidor de configuración Vault

La configuración del servidor de configuraciones esta incompleta

**Inicializamos el vault**

****

En esta parte ingresamos el archivo de configuraciones

****

Guardamos una key con su valor para iniciar la configuración del servicio

****