



Universidad Central del Ecuador

Programación Distribuida

Título: Informe de descripción de
herramientas y Tecnologías e
Implementación en el proyecto (
Apache TomEE)



Integrantes Grupo 8: Alexander Columba

Alexis Masache

Santiago Vega



Contenido

1. Objetivo	3
2. Objetivos Específicos	3
3. Marco Teórico Herramientas Tecnológicas	3
3.1. Apache TomEE	3
3.2. JPA e Hibernate	4
3.3. Servidor de Registro y Configuración Zookeeper:.....	5
3.4. Spring Cloud Gateway y Spring Cloud Load Balancer	6
4. Entorno de Desarrollo	8
5. Implementación de las Aplicaciones:	8
5.1. Servidor 1: Customer	8
5.2. Servidor 2: Orders	14
5.3. Creación del Cliente Web	18
6. Servidor de Registro y Configuración Apache Zookeeper	21
Instalación del Servidor Zookeeper	21
Servidor Spring Cloud Gateway	22
7. Conclusiones:	23
8. Referencias:	24

1. Objetivo

- Realizar una aplicación RESTful la cual exponga servicios que permitan realizar un CRUD de las tablas de la base de datos con las tablas Customers y Orders utilizando las tecnologías apache TomEE MicroProfile como framework para microservicios, jpa para acceso a datos, ZooKeeper como servidor de configuración y registro y Spring Cloud para Gateway y balanceador de carga .

2. Objetivos Específicos

- Crear una aplicación servidor 1 con servicios Rest que permita crear , leer, actualizar y borrar datos de la tabla Customer.
- Crear una aplicación servidor 2 con servicios Rest que permita crear, leer, actualizar y borrar datos de la tabla Orders.
- Establecer comunicación remota entre el servidor2 y servidor1 con microprofile rest client
- Conocer las bases teóricas y su implementación de las herramientas tecnológicas Tomee microprofile, Hibernate JPA , Zookeeper, Spring Cloud Gateway.

3. Marco Teórico Herramientas Tecnológicas

3.1. Apache TomEE

Apache TomEE se ensambla a partir de una distribución oficial de vainilla Apache Tomcat. No hay que seleccionar y elegir partes individuales de Tomcat y construir un servidor "nuevo" aprovechando Tomcat. Comenzamos con Tomcat, añadimos nuestros frascos y configuración y comprimimos el resto. El resultado es Tomcat con características EE añadidas.

TomEE viene en cuatro distribuciones: TomEE WebProfile, TomEE MicroProfile, TomEE+ y TomEE PluME. TomEE WebProfile se dirige al perfil web completo de Jakarta EE. Los otros hacen lo mismo, pero añaden soporte para MicroProfile, JMS, JAX-WS, Jakarta Connectors, y para ayudar a esas aplicaciones de migración de Glassfish a TomEE.

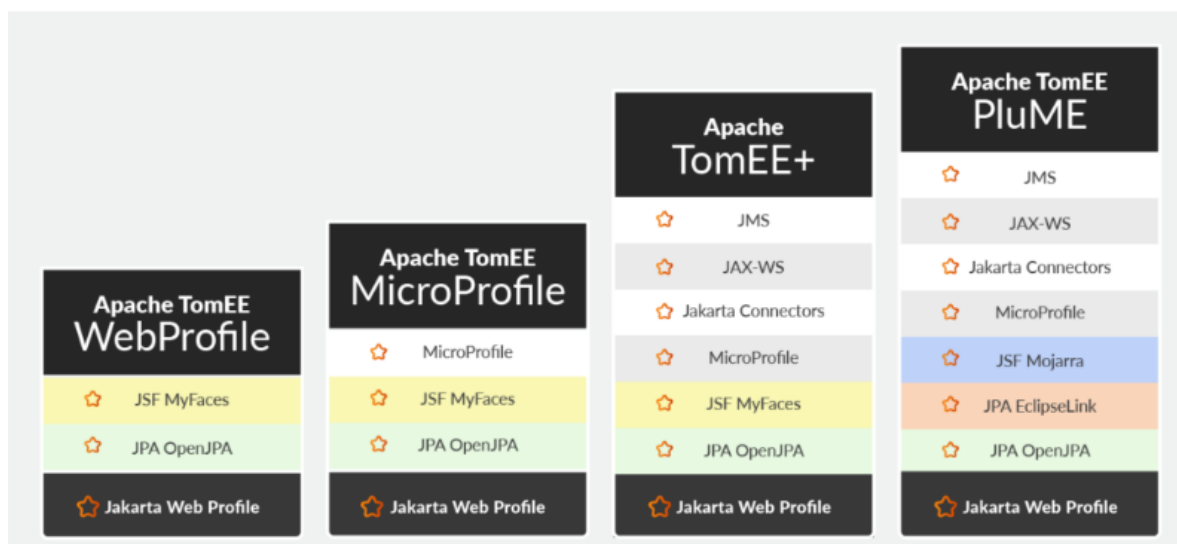
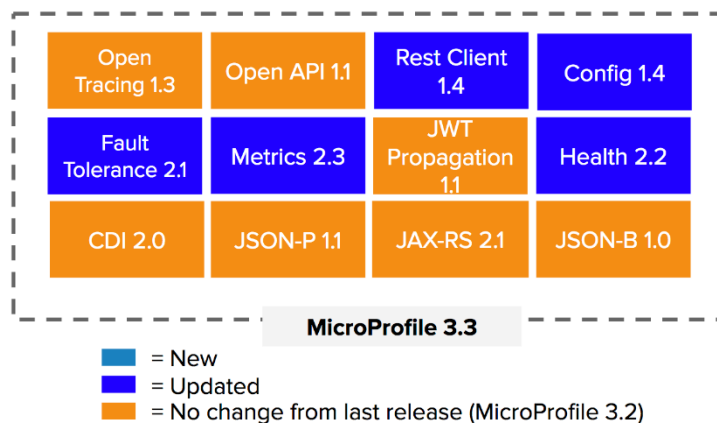


Figura 1. Distribuciones de TomEE

MicroProfile

Eclipse MicroProfile es una colección de librerías Java EE y tecnologías que juntas forman la línea base para microservicios que apunta a entregar aplicaciones portables a través de múltiples sistemas operativos.

MicroProfile fue creado en 2016 y rápidamente se unió a la fundación Eclipse. Desde entonces, ha habido cinco versiones de la plataforma MicroProfile con la adición de muchas especificaciones para abordar las necesidades y comentarios de los usuarios. MicroProfile según la versión de febrero del 2020, se compone de la siguiente especificación



El propósito principal de MicroProfile es crear un marco Java Enterprise para implementar microservicios portátiles entre soluciones de proveedor. MicroProfile se ocupa de un modelo de programación independiente del proveedor, así como de la configuración y los servicios como el seguimiento, la tolerancia a errores, el estado y las métricas, por nombrar algunos.

TomEE MicroProfile 8.0.4

TomEE MicroProfile añade soporte completo para MicroProfile 3.3 para proporcionar una plataforma completa y robusta para implementar microservicios Java.

Estas son las nuevas especificaciones de MicroProfile 3.3 admitidas por TomEE MicroProfile:

- Config 1.4
- Fault Tolerance 2.1
- Health 2.2
- Metrics 2.3
- Rest Client 1.4

3.2. JPA e Hibernate

JPA

JPA (Java Persistence API) es una especificación que indica como hacer la persistencia en aplicaciones Java, al decir especificación quiere decir que únicamente indica que es lo que debe hacer más sin embargo no cómo hacerlo o lo que se conoce como



implementación, para la implementación se usa otros frameworks como por ejemplo Hibernate, EclipseLink entre los más conocidos.

Hibernate

Hibernate es una herramienta de Mapeo objeto-relacional (ORM), para la plataforma Java también disponible para .Net, bajo el nombre de NHibernate – que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación; mediante archivos declarativos (XML) o anotaciones en los beans.

Agroal

es una implementación de pool de conexiones moderna y liviana diseñada para un rendimiento y escalabilidad muy altos, y presenta una integración de primera clase con los otros componentes de Quarkus, como seguridad, componentes de administración de transacciones y métricas de salud.

3.3. Servidor de Registro y Configuración Zookeeper:

ZooKeeper es un proyecto de Apache que nos provee de un servicio centralizado para diversas tareas como por ejemplo mantenimiento de configuración, registro y descubrimiento, monitoreo integrado con Prometheus y Grafana , sincronización distribuida o servicios de agrupación, servicios que normalmente son consumidos por otras aplicaciones distribuidas.

Características:

Sencillo: Permite la coordinación entre procesos distribuidos mediante un namespace jerárquico que se organiza de manera similar a un file system. El namespace consiste en registros (znodes) similares a ficheros o directorios. En contraposición con el típico file system, ZooKeeper mantiene la información en memoria permitiendo obtener latencias bajas y rendimientos altos.

Replicable: Permite las réplicas instaladas en múltiples hosts, llamados conjuntos o agrupaciones. Los servidores que conforman el servicio ZooKeeper deben conocerse todos entre ellos. Estos mantienen una imagen en memoria del estado, completado con un log de transacciones y snapshots almacenados en un store persistente. Mientras la mayoría de servicios este disponible, ZooKeeper se mantendrá disponible.

Ordenado: ZooKeeper se encarga de marcar cada petición con un número que refleja su orden entre todas las transacciones de manera que permite mantener por completo la trazabilidad de las operaciones realizadas.

Rápido: Sobre todo en entornos en los que predominen las operaciones de lectura

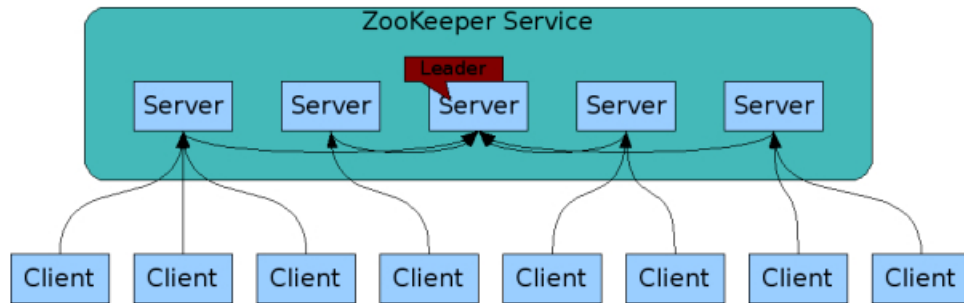


Figura 2. ZooKeeper: Un Servicio de Coordinación Distribuido para Aplicaciones Distribuidas

ZooKeeper además está previsto que se utilice como un almacenamiento de datos de configuración centralizado. Sin embargo, a pesar del hecho de que ZooKeeper tiene un modelo de datos que se parece a un sistema de archivos UNIX OS (ZNode se puede interpretar como un "directorio" que puede tener datos asociados a él), hay varias restricciones, que limitan el uso de ZooKeeper como un sistema de archivos ordinario

3.4. Spring Cloud Gateway y Spring Cloud Load Balancer

¿Qué es API Gateway?

Una puerta de enlace de API proporciona un único punto de entrada para todos los microservicios, incluida la orquestación y las capacidades de seguridad y supervisión añadidas. Netflix Zuul, Amazon API Gateway, Apigee y, por supuesto, Spring Cloud Gateway son algunas de las implementaciones conocidas de api gateway.

Spring Cloud Gateway es la implementación de API Gateway por parte del equipo de Spring Cloud, además del ecosistema reactivo de Spring. Proporciona una forma sencilla y eficaz de enrutar las solicitudes entrantes al destino adecuado mediante la asignación de controlador de puerta de enlace.

Además, Spring Cloud Gateway utiliza el servidor Netty para proporcionar un procesamiento de solicitudes asíncrono sin bloqueo.

A continuación, se muestra un flujo de alto nivel de cómo funciona el enrutamiento de solicitudes en Spring Cloud Gateway:

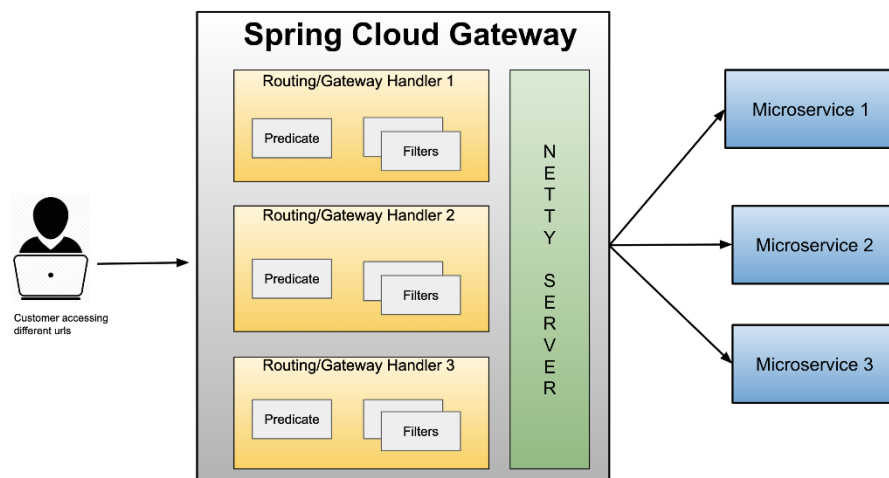


Figura 3. Arquitectura Spring Cloud Gateway

Spring Cloud Gateway consta de 3 bloques de construcción principales:

Ruta: Piense en esto como el destino al que queremos que una solicitud en particular se dirija. Se compone de URI de destino, una condición que tiene que satisfacer o en términos técnicos, predicados y uno o más filtros.

Predicado: Esto es literalmente una condición para coincidir. es decir, una especie de condición de "si". si las solicitudes tienen algo, por ejemplo, el encabezado path-blah o request contiene foo-bar, etc. En términos técnicos, es Java 8 Function Predicate

Filtro: Estas son instancias de Spring Framework WebFilter. Aquí es donde puede aplicar su magia de modificar la solicitud o respuesta. Hay una gran cantidad de WebFilter listo para usar que proporciona el marco de trabajo.

Balanceador de Carga

Load balancer sirve como punto único de contacto para los clientes. El balanceador de carga distribuye el tráfico entrante de aplicaciones entre varios destinos, Esto aumenta la disponibilidad de la aplicación.

Para que podamos usar Spring Cloud Load Balancer, necesitamos tener un registro de servicio en funcionamiento. Un registro de servicio hace que sea trivial consultar mediante programación la ubicación de un servicio determinado en un sistema. Hay varias implementaciones populares, incluyendo Apache Zookeeper, Eureka de Netflix, Hashicorp Consul, y otros. Incluso puede utilizar Kubernetes y Cloud Foundry como registros de servicio. Spring Cloud proporciona una abstracción, , que puede usar para hablar con estos registros de servicio de forma genérica. Hay varios patrones que un registro de servicios habilita que simplemente no son posibles con dns bueno. Una cosa que me encanta hacer es el equilibrio de carga del lado del cliente. El equilibrio de carga del lado cliente requiere que el código de cliente decida qué nodo recibe la solicitud.

Hay cualquier número de instancias del servicio por ahí, y su idoneidad para manejar una solicitud determinada es algo que cada cliente puede decidir. Es aún mejor si puede tomar la decisión antes de lanzar una solicitud que de otro modo podría estar condenada al fracaso.

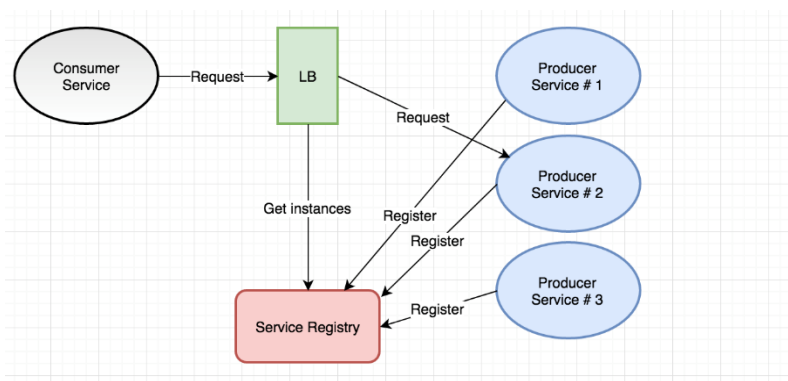


Figura 4. Balanceador de Carga

4. Entorno de Desarrollo

IDE: Spring tool suite 4

versión de Java: 11

Servidor 1 y Servidor 2

Framework: Apache TomEE MicroProfile 8.0.4, versión de MicroProfile: 3.3

Gestor de dependencias: Maven

Cliente Web

servidor: Apache Tomcat 9.0.36

Gestor de dependencias: Gradle

Servidor Gateway:

Framework: Spring boot

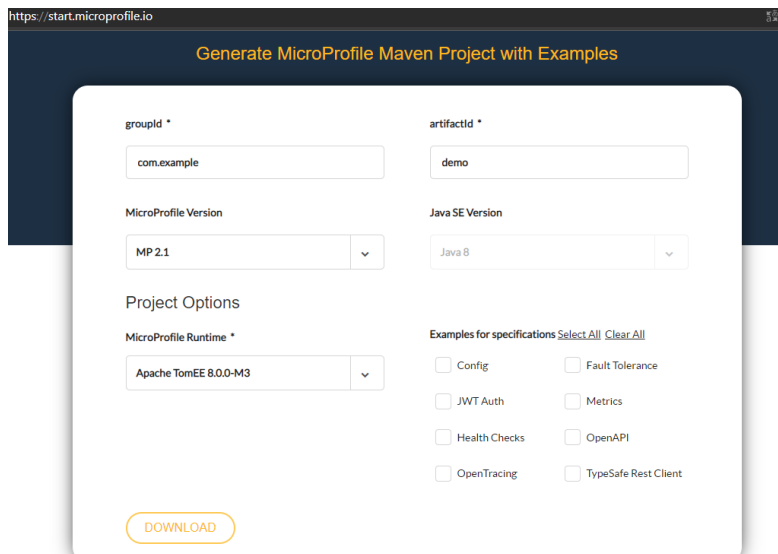
Gestor de dependencias: Gradle

Servidor de Registro y Configuración: Zookeeper 3.6.2

5. Implementación de las Aplicaciones:

5.1. Servidor 1: Customer

- 1) Como primer paso nos dirigimos a <https://start.microprofile.io/> en la pagina llenamos todos los campos que nos piden . Una vez hecho esto podemos descargarnos el código generado con las dependencias de maven. Luego dentro del proyecto podremos cambiar las versiones de MicroProfile y de TomEE.



The screenshot shows the 'Generate MicroProfile Maven Project with Examples' form on the <https://start.microprofile.io/> website. The form includes the following fields and options:

- groupId ***: Input field with 'com.example'.
- artifactId ***: Input field with 'demo'.
- MicroProfile Version**: Dropdown menu with 'MP 2.1' selected.
- Java SE Version**: Dropdown menu with 'Java 8' selected.
- Project Options**:
 - MicroProfile Runtime ***: Dropdown menu with 'Apache TomEE 8.0.0-M3' selected.
 - Examples for specifications**: A grid of checkboxes for various features:
 - ☐ Config
 - ☐ Fault Tolerance
 - ☐ JWT Auth
 - ☐ Metrics
 - ☐ Health Checks
 - ☐ OpenAPI
 - ☐ OpenTracing
 - ☐ TypeSafe Rest Client

A 'DOWNLOAD' button is located at the bottom of the form.



Una vez hecho esto se debe abrir en el IDE en he importamos el proyecto maven, una vez realizado esto nos queda el archivo pom.xml como el siguiente:

Dependencias y versión de TomEE:

```
6 <modelVersion>4.0.0</modelVersion>
7 <groupId>com.distribuida</groupId>
8 <artifactId>servidor-orders-tomee</artifactId>
9 <version>1.0-SNAPSHOT</version>
10 <packaging>war</packaging>
11 <properties>
12   <maven.compiler.target>1.8</maven.compiler.target>
13   <failOnMissingWebXml>false</failOnMissingWebXml>
14   <maven.compiler.source>1.8</maven.compiler.source>
15   <tomee.version>8.0.4</tomee.version>
16   <final.name>servidor-orders-tomee</final.name>
17 </properties>
18 <dependencies>
19
20   <dependency>
21     <groupId>org.eclipse.microprofile</groupId>
22     <artifactId>microprofile</artifactId>
23     <version>3.3</version>
24     <type>pom</type>
25     <scope>provided</scope>
26   </dependency>
27
28   <dependency>
29     <groupId>org.hibernate</groupId>
30     <artifactId>hibernate-agroal</artifactId>
31     <version>5.4.21.Final</version>
32   </dependency>
33
```

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.2.8</version>
</dependency>
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-x-discovery</artifactId>
  <version>5.1.0</version>
</dependency>
</dependencies>
<build>
  <finalName>servidor-orders-tomee</finalName>
</build>
<profiles>
  <profile>
    <id>tomee</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
  </profile>
</profiles>
```

Plugin para crear Uber-JAR

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.tomee.maven</groupId>
      <artifactId>tomee-maven-plugin</artifactId>
      <version>${tomee.version}</version>
      <executions>
        <execution>
          <id>executable-jar</id>
          <phase>package</phase>
          <goals>
            <goal>exec</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <context>ROOT</context>
        <tomeeClassifier>microprofile</tomeeClassifier>
      </configuration>
    </plugin>
  </plugins>
</build>
```

- 2) Como segundo paso procedo a configurar la conexión a la base de datos en el archivo resources.xml ubicado dentro de src/main/webapp/WEB-INF/resources.xml o también se puede configurar en src/main/tomee/conf/system.properties



```

servidor-orders-tomee/pom.xml  resources.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <tomee>
3   <Resource id="jdbc/prueba" type="javax.sql.DataSource">
4     JdbcDriver = org.postgresql.Driver
5     JdbcUrl = jdbc:postgresql://localhost/Prueba
6     UserName = postgres
7     Password = admin
8     jtaManaged = true
9   </Resource>
10 </tomee>

```

3) Crear la entidad y hacemos usamos Hibernate

Creamos la entidad Customer con sus atributos además creamos getters y setters. Para definir una entidad se anota con @Entity, @Table especificamos el nombre de la tabla en la base de datos , además de poder personalizar nuestro Id con @Id y @GeneratedValue para que el id sea autoincrementar y @Column podemos colocar los nombres de nuestras columnas de la base de datos además añadimos unos named queries para hacer consultas personalizadas.

```

@Entity
@Table(name = "customer")
@NamedQueries({
    @NamedQuery(name = "Customer.findAll", query = "SELECT p FROM Customer p"),
    @NamedQuery(name = "Customer.findBySurname", query = "SELECT p FROM Customer p WHERE p.surname = :surname")
})
public class Customer implements Serializable{

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    public Long id;
    @Column(name = "name")
    public String name ;
    @Column(name = "surname")
    public String surname ;
}

```

Luego creamos una clase interfaz he implementación para los métodos de de la siguiente forma

```
import java.util.List;

public interface ServicioCustomer {

    public Customer insertarCustomer( Customer P);

    public List<Customer> obtenerCustomer() ;

    void eliminarCustomer(Integer id) ;

    void actualizarCustomer(Customer p ) ;

    Customer obtenerCustomerPorId(Integer id);

    public List <Customer> obtenerCustomerporApellido(String surname);

}
```

Y la implementación de esta colocamos la lógica del CRUD, la anotación @Transactional es útil para los métodos de eliminar , crear y actualizar que alteran la base de datos. Los proveedores de persistencia como Hibernate utilizan el contexto de persistencia @PersistenceContext para administrar el ciclo de vida de la entidad en una aplicación colocamos el nombre que pusimos en persistence.xml

```
@ApplicationScoped
public class ServicioCustomerImpl implements ServicioCustomer {

    @PersistenceContext(unitName = "myCustomer_PU")
    EntityManager em;

    @Override
    public Customer insertarCustomer(Customer P) {
        Customer cust = null;
        em.persist(P);
        cust = em.find(Customer.class, P.getId());
        return cust;
    }

    @Override
    public List<Customer> obtenerCustomer() {
        return em.createNamedQuery("Customer.findAll", Customer.class).getResultList();
    }

    @Override
    public void eliminarCustomer(Integer id) {
        Customer usuario = em.find(Customer.class, id);
        if (!usuario.equals(null)) {
            em.remove(usuario);
        } else {
            System.out.println("No se pudo eliminar, customer no existe en BD");
        }
    }

    @Override
    public void actualizarCustomer(Customer p) {
        Customer cust = null;
        cust = em.find(Customer.class, p.getId());
        cust.setName(p.getName());
        cust.setSurname(p.getSurname());
        em.merge(cust);
    }
}
```

4) Crear las clases RestAplication y CustomerRest

En nuestra clase Rest Application registramos al servidor en zookeeper de la siguiente manera , primero establecemos la conexión con el servidor zookeeper localhost: 2181 y luego construimos una instancia del servicio ponemos un id que puede ser cualquier identificador ,un nombre , el puerto , la url, y una Uri ip +puerto y lo construimos por defecto se creara en el nodo principal “services”

```
// Path es el mapeo de nuestros servicios
@Path("/")
@ApplicationPath("/")
public class RestApplication extends Application{
    @Inject
    @ConfigProperty(name="http.port", defaultValue = "8082")
    private Integer port;

    public void init(@Observes @Initialized(ApplicationScoped.class) Object event) throws Exception{

        CuratorFramework client = CuratorFrameworkFactory
            .newClient("localhost:2181", new RetryForever(5));

        client.start();

        ServiceInstance<Object> serviceInstance = ServiceInstance.builder()
            .id("customer-server: " + port)
            .name("customer-server")
            .port(port)
            .address("127.0.0.1")
            .uriSpec(new UriSpec("{scheme}://{address}:{port}"))
            .build();

        ServiceDiscoveryBuilder.builder(Object.class)
            .basePath("services")
            .client(client)
            .thisInstance(serviceInstance)
            .build()
            .start();
    }
}
```

En nuestro Customer rest:

```
@Path("/customers")
@ApplicationScoped
public class CustomerRest {

    @Inject
    ServicioCustomer serviCustomer ;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Customer> listaCustomers() {
        List<Customer> lista = serviCustomer.obtenerCustomer();
        return lista;
    }

    @GET
    @Path("/{customer/{id}}")
    @Produces(MediaType.APPLICATION_JSON)
    public Customer customerPorId(@PathParam("id") Integer id) {
        Customer p = serviCustomer.obtenerCustomerPorId(id);
        if (p == null) {
            throw new WebApplicationException("Customer con id : " + id + " no existe.", 404);
        }
        return p;
    }
}
```

Anotamos En la clase customerRest inyectamos a la interfaz ServicioCustomer para usar los métodos de su implementación. Anotamos los métodos http con @Get, @Post, @PUT, @Delete de JAX-RS para indicar que son de tipo consulta de información, creación ,actualización o eliminación de registros.

Ademas anotamos con @Produces indicaremos que la respuesta se dará en formato JSON y @Consumes para indicar que la solicitud recibida se aceptara en formato JSON. anotamos con @Transactional en los métodos que alteren a la base de datos

Por ejemplo:

```
@GET
@Path("/customer/{id}")
@Produces(MediaType.APPLICATION_JSON)
public Customer customerPorId(@PathParam (value="id") Integer id) {
    Customer sing = serviCustomer.obtenerCustomerPorId(id);
    return sing;
}

@POST
@Path("/crear")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Customer crearCustomer(Customer p) {
    Customer cust = null;
    try {
        cust = serviCustomer.insertarCustomer(p);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return cust;
}

@PUT
@Path("/actualizar")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response actualizarPersona(Customer p){
    if (p.id == null) {
        throw new WebApplicationException("el id se establecio de forma no valida.", 422);
    }
    serviCustomer.actualizarCustomer(p);
    return Response.ok(p).build();
}
```

5.2. Servidor 2: Orders

- 1) Creamos el proyecto gradle igual que el anterior servidor con las mismas dependencias y plugin
- 2) Al igual que el anterior servidor se debe colocar la configuración de la conexión a base de datos en el archivo resources.xml ubicado dentro de src/main/webapp/WEB-INF/resources.xml o también se puede configurar en src/main/tomee/conf/system.properties

```
servidor-orders-tomee/pom.xml  resources.xml  ⌕
1 <?xml version="1.0" encoding="UTF-8"?>
2 <tomee>
3   <Resource id="jdbc/prueba" type="javax.sql.DataSource">
4     JdbcDriver = org.postgresql.Driver
5     JdbcUrl = jdbc:postgresql://localhost/Prueba
6     UserName = postgres
7     Password = admin
8     jtaManaged = true
9   </Resource>
10 </tomee>
```

- 3) Creamos la Clase Ordenes que es nuestra entidad, anotamos con @Entity y con @Table(name="orders") ya que el nombre de la tabla es orders para que no haya confusión.. Se debe anotar con @Transient para indicar que este atributo no es persistente y no es tomado en cuenta a la hora de persistir el Objeto, en este caso anotamos al atributo datosCliente ya que en este se guardara el nombre y apellido del cliente que se traerá del servidor 1. Demas añadimos un par de Named Querys para hacer consultas personalizadas que nos serán útiles

```
@Entity
@Table(name="orders")
@NamedQueries({
    @NamedQuery(name = "Ordenes.findAll", query = "SELECT p FROM Ordenes p"),
    @NamedQuery(name = "Ordenes.findByIdCustomer", query = "SELECT p FROM Ordenes p WHERE p.customer_id = :customer_id")
})
public class Ordenes implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    public Long id;
    @Column(name = "item")
    public String item;
    @Column(name = "price")
    public Double price;
    @Column(name = "customer_id")
    public Long customer_id;
    @Transient
    public String datosCliente;
```

- 4) Creamos la clase Customer ya que ambos microservicios se van a conectar necesitamos esta entidad con los mismos atributos pero sin ninguna anotación que en el servidor Customer

```
public class Customer {
    private Long id;
    private String name ;
    private String surname ;

    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getSurname() {
        return surname;
    }
    public void setSurname(String surname) {
        this.surname = surname;
    }
}
```

Luego creamos la interfaz de implementación del servicio Ordenes

```
import java.util.List;

public interface ServicioOrdenes {

    void insertarOrdenes( Ordenes p);

    public List<Ordenes> obtenerOrdenes();

    void eliminarOrdenes(Integer id) ;

    void actualizarOrdenes(Ordenes p );

    Ordenes obtenerOrdenesPorId(Integer id);
}
```

Luego creamos la clase CustomerProxy que se va a conectar con el microservicio Cliente usando MicroProfile client a través de la notación @RegisterRestClient colocamos la url de nuestro microservicio (del Gateway que veremos más adelante) y hacemos un método GET para traer los datos de los clientes

```
@RegisterRestClient(baseUri ="http://localhost:9091/customer-server/customers")
public interface CustomerProxy {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Customer> listarTodos();
}
```


Implementamos esta interfaz en CustomerProxyImpl para poder inyectarlo desde nuestra implementación del servicio ordenes

```
@ApplicationScoped

public class CustomerProxyImpl {
    @Inject
    @RestClient
    private CustomerProxy proxy;

    public List<Customer> obtenerClientes(){
        return proxy.listarTodos();
    }
}
```

Fijémonos en la Implementación del CRUD en @PersistenceContext colocamos el nombre que pusimos en persistence.xml

```
@ApplicationScoped
public class ServicioOrdenesImpl implements ServicioOrdenes{
    @PersistenceContext(unitName = "myCustomer_PU")
    EntityManager em;

    @Inject private CustomerProxyImpl servicioCliente;

    // metodo para insertar Ordenes
    @Override
    public Ordenes insertarOrdenes(Ordenes P) {
        Ordenes ord = null;
        em.persist(P);
        ord = em.find(Ordenes.class, P.getId());
        return ord;
    }

    //metodo para obtener todas las Ordenes
    @Override
    public List<Ordenes> obtenerOrdenes() {
        //List<Ordenes> lista = new ArrayList<>();
        return em.createNamedQuery("Ordenes.findAll", Ordenes.class).getResultList();
        //return em.createQuery("FROM Ordenes ", Ordenes.class).getResultList();
    }

    //metodo para eliminar una orden por Id
    @Override
    public void eliminarOrdenes(Integer id) {
        Ordenes ordenes = em.find(Ordenes.class, id);
        if (!ordenes.equals(null)) {
            em.remove(ordenes);
        } else {
            System.out.println("No se pudo eliminar, orden no existe en BD");
        }
    }
}
```

Luego podemos obtener una lista de todos los Customers para luego usar esta lista con el fin de que en las Ordenes podamos obtener datos adicionales como nombre y apellido del cliente. Esto lo realizo usando streams recorriendo la lista de Ordenes y comparando que el id_customer de las Ordenes sean iguales a Id de los Customers luego con otro forEach asignando al atributo datosCliente de la Entidad Ordenes el nombre y apellido del cliente correspondiente.

```
//metodo para obtener una lista de todos los Customers
@Override
public List<Customer> obtenerClientes() {

    return servicioCliente.obtenerClientes();

}

//metodo para Obtener una lista de todas las Ordenes con datos del Customer
@Override
public List<Ordenes> obtenerOrdenesCustomer() {
    List<Ordenes> listaOrdenes = obtenerOrdenes();
    List<Customer> listaCustomer=obtenerClientes();

    listaOrdenes.forEach(obj2 -> listaCustomer.stream()
        .filter(obj1 -> obj2.customer_id==obj1.getId())
        .forEach(obj1 -> obj2.datosCliente=obj1.getName()+" "+obj1.getSurname());

    return listaOrdenes;

}

//metodo para Obtener una lista de todas las Ordenes con datos del Customer filtrado por ID del Customer
@Override
public List<Ordenes> obtenerOrdenesByIdCustomer(Integer id) {
    List<Ordenes> listaOrdenes = obtenerOrdenesporIdCustomer(id);
    List<Customer> listaCustomer=obtenerClientes();

    listaOrdenes.forEach(obj2 -> listaCustomer.stream()
        .filter(obj1 -> obj2.customer_id==obj1.getId())
        .forEach(obj1 -> obj2.datosCliente=obj1.getName()+" "+obj1.getSurname());

    return listaOrdenes;

}
```

- 5) Crear las clases Rest muy similar al servidor anterior, inyectamos al Servicio Ordenes y creamos los métodos Rest GET, POST, PUT y DELETE de igual forma anotamos con `@Transactional` en los métodos que alteren a la base de datos .
Ademas anotamos con `@Produces` indicaremos que la respuesta se dará en formato JSON y `@Consumes` para indicar que la solicitud recibida se aceptara en formato JSON.

```
@Path("/orders")
@ApplicationScoped
public class OrdenesRest {

    @Inject
    private ServicioOrdenes servicio;
    @Inject
    @ConfigProperty(name="http.port", defaultValue = "8083")
    private Integer port;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Ordenes> listaOrdenes() {
        System.out.println("puerto corriendo: "+port);
        List<Ordenes> lista = servicio.obtenerOrdenesCustomer();
        return lista;
    }

    @GET
    @Path("/order/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Ordenes ordenPorId(@PathParam (value="id") Integer id) {
        Ordenes p = servicio.obtenerOrdenesPorId(id);
        return p;
    }

    @POST
    @Transactional
    @Path("/crear")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Ordenes crearOrden(Ordenes p) {
        Ordenes orden=null;
        try {
            orden = servicio.insertarOrdenes(p);
        }
    }
}
```

- 6) Creamos la clase RestApplication donde se registrara en el servidor de registros zookeeper primero se conecta a zookeeper mediante localhost: 2181, luego creamos una instancia del servidor con un id , nombre, puerto. Ip , y una uri y lo registramos.

```
// Path es el mapeo de nuestros servicios
@ApplicationPath("/")
public class RestApplication extends Application{
    @Inject
    @ConfigProperty(name="http.port", defaultValue = "8082")
    private Integer port;

    public void init(@Observes @Initialized(ApplicationScoped.class) Object event) throws Exception{

        CuratorFramework client = CuratorFrameworkFactory
            .newClient("localhost:2181", new RetryForever(5));

        client.start();

        ServiceInstance<Object> serviceInstance = ServiceInstance.builder()
            .id("customer-server: " + port)
            .name("customer-server")
            .port(port)
            .address("127.0.0.1")
            .uriSpec(new UriSpec("{scheme}://{address}:{port}"))
            .build();

        ServiceDiscoveryBuilder.builder(Object.class)
            .basePath("services")
            .client(client)
            .thisInstance(serviceInstance)
            .build()
            .start();

    }

    public void destroy(@Observes @Destroyed(ApplicationScoped.class) Object event) {
        System.out.println("destroy");
    }
}
```

5.3. Creación del Cliente Web

- 1) Se crea un nuevo proyecto gradle con las siguientes dependencias:

```
plugins {
    id 'java-library'
    id 'war'
    id 'eclipse-wtp'
}

repositories {
    jcenter()
}

dependencies {
    //RESteasy
    compile group: 'org.jboss.resteasy', name: 'resteasy-client', version: '3.0.17.Final'
    compile group: 'org.jboss.resteasy', name: 'resteasy-jackson2-provider', version: '3.9.0.Final'
    compile group: 'org.jboss.resteasy', name: 'resteasy-jaxb-provider', version: '3.9.0.Final'
    //CDI
    providedCompile group: 'javax.servlet', name: 'javax.servlet-api', version: '4.0.1'
    compile group: 'org.jboss.weld.servlet', name: 'weld-servlet-shaded', version: '3.1.0.Final'
    //JSF
    compile group: 'org.glassfish', name: 'javax.faces', version: '2.4.0'
    compile group: 'javax.servlet', name: 'javax.servlet-api', version: '4.0.1'
}

task stage(dependsOn: ['build', 'clean'])
build.mustRunAfter clean
```

- 2) Creamos las dos Entidades Customer y Ordenes con sus getters y setters
- 3) Igualmente, como en el servidor2 hago uso de RestEasy Proxy Framework
Así mismo creando tanto como el proxy del Customer y proxy de Ordenes y sus respectivas implementaciones

```
public interface OrderProxy {

    @GET
    @Consumes(MediaType.APPLICATION_JSON)
    public List<Ordenes> listarTodos();

    @GET
    @Consumes(MediaType.APPLICATION_JSON)
    @Path("/orden/{id}")
    public Ordenes ordenPorId(@PathParam ("id") Integer id);

    @POST
    @Path("/crear")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public void crear(Ordenes o);

    @PUT
    @Path("/actualizar")
    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_JSON)
    public Ordenes actualizar(Ordenes p);

    @DELETE
    @Path("/eliminar/{id}")
    @Consumes(MediaType.APPLICATION_JSON)
    public String eliminar(@PathParam ("id") Integer id);

    @GET
    @Path("/orderbyCustomerId/{id}")
    @Consumes(MediaType.APPLICATION_JSON)
    public List<Ordenes> listarOrdenesconCustomerbyId(@PathParam ("id") Integer id);
}

@ApplicationScoped
public class OrderProxyImpl {

    @Inject private OrderProxy proxy;

    public List<Ordenes> obtenerOrdenes(){
        return proxy.listarTodos();
    }

    public Ordenes buscarPorId(Integer id) {
        return proxy.ordenPorId(id);
    }

    public void eliminarOrden(Integer id) {
        proxy.eliminar(id);
    }

    public void crear(Ordenes p) {
        proxy.crear(p);
    }

    public Ordenes editar(Ordenes p) {
        return proxy.actualizar(p);
    }

    public List<Ordenes> buscarOrdenporCustomerId(Integer id) {
        return proxy.listarOrdenesconCustomerbyId(id);
    }
}
```

```
public interface CustomerProxy {

    @GET
    @Consumes(MediaType.APPLICATION_JSON)
    public List<Customer> listarTodos();

    @POST
    @Path("/crear")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public void crear(Customer c);

    @PUT
    @Path("/actualizar")
    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_JSON)
    public void actualizar(Customer c);

    @DELETE
    @Path("/eliminar/{id}")
    @Consumes(MediaType.APPLICATION_JSON)
    public String eliminar(@PathParam ("id") Integer id);
}

@ApplicationScoped
public class CustomerProxyImpl {

    @Inject private CustomerProxy proxy;

    public List<Customer> obtenerClientes(){
        return proxy.listarTodos();
    }

    public void eliminarClientes(Integer id) {
        proxy.eliminar(id);
    }

    public void crear(Customer c) {
        proxy.crear(c);
    }

    public void editar(Customer c) {
        proxy.actualizar(c);
    }
}
```

- 4) Creamos las clases producer donde ira la configuración para la conexión con los servidores usando RestClientBuilder especificamos la URI del Gateway de ambos microservicios

```
@ApplicationScoped
public class CostumerProducer {
    private static final String URL_SERVIDOR1 = "http://localhost:9091/customer-server/customers";

    @Produces
    @ApplicationScoped
    public CustomerProxy getProxy() {
        ResteasyClient client = new ResteasyClientBuilder().build();
        ResteasyWebTarget target = client.target(UriBuilder.fromPath(URL_SERVIDOR1));
        CustomerProxy proxy = target.proxy(CustomerProxy.class);
        return proxy;
    }
}
```

```
@ApplicationScoped
public class OrderProducer {
    private static final String URL_SERVIDOR2 = "http://localhost:9091/order-server/orders";

    @Produces
    @ApplicationScoped
    public OrderProxy getProxy() {
        ResteasyClient client = new ResteasyClientBuilder().build();
        ResteasyWebTarget target = client.target(UriBuilder.fromPath(URL_SERVIDOR2));
        OrderProxy proxy = target.proxy(OrderProxy.class);
        return proxy;
    }
}
```

- 5) Ahora se debe crear los controladores de Ordenes y Customers, se debe inyectar a sus proxysImpl , se anotan con @Named para usarlos con ese nombre desde los archivos.xhtml

```
@Named(value = "controladorOrdenes")
@SessionScoped
public class OrdenController {

    @Inject private OrderProxyImpl servicioOrden;
```

```
@Named(value = "controladorCliente")
@SessionScoped
public class CustomerController {

    @Inject private CustomerProxyImpl servicioCliente;
```

He implementamos los métodos para enviar los datos desde los servicios rest a las vistas.jsf

```
public String redireccionCrear() {
    cliente.setName(null);
    cliente.setSurname(null);
    return "crearCliente?faces-redirect=true";
}

public String redireccionCliente() {
    listaClientes = servicioCliente.obtenerClientes();
    return "vistaCliente?faces-redirect=true";
}

public String eliminarCliente(Integer id) {
    if(id==null) {
        return "vistaCliente?faces-redirect=true";
    }else {
        servicioCliente.eliminarClientes(id);
        listaClientes = servicioCliente.obtenerClientes();
        return "vistaCliente?faces-redirect=true";
    }
}
```

```
public String crearCliente() {
    cliente.setId(null);
    servicioCliente.crear(cliente);
    listaClientes = servicioCliente.obtenerClientes();
    return "vistaCliente?faces-redirect=true";
}

public String redireccionEditar(Customer cliedit) {
    cliente = cliedit;
    return "editarCliente?faces-redirect=true";
}

public String editar() {
    servicioCliente.editar(cliente);
    return "vistaCliente?faces-redirect=true";
}
```




- 6) Creamos los archivos .xhtml dentro de la carpeta src/main/webapp que son las vistas.

6. Servidor de Registro y Configuración Apache Zookeeper

Instalación del Servidor Zookeeper

- 1) Nos dirigimos a <https://zookeeper.apache.org/releases.html> descargamos la ultima versión que es la 3.6.2
- 2) Descomprimos en cualquier directorio y nos quedara de la siguiente manera además creamos la carpeta data

bin	14/9/2020 20:57
conf	14/9/2020 20:57
data	14/9/2020 21:39
docs	14/9/2020 20:56
lib	14/9/2020 20:57
logs	14/9/2020 21:18
LICENSE	4/9/2020 07:43
NOTICE	4/9/2020 07:43
README.md	4/9/2020 07:43
README_packaging.md	4/9/2020 07:43

- 3) Abrimos conf/zoo_sample.cfg y descomentamos dataDir y colocamos la ruta de nuestra carpeta data ,ademas descomentamos , tickTime, initLimit=10, syncLimit=5, clientPort=2181

```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sakes.
dataDir=C:\\apache-zookeeper-3.6.2\\apache-zookeeper-3.6.2-bin\\data
# the port at which the clients will connect
clientPort=2181
# the maximum number of client connections.
# increase this if you need to handle more clients
#maxClientCnxns=60
#
# Be sure to read the maintenance section of the
# administrator guide before turning on autopurge.
#
# http://zookeeper.apache.org/doc/current/zookeeperAdmin.html#sc_maintenance
#
# The number of snapshots to retain in dataDir
#autopurge.snapRetainCount=3
# Purge task interval in hours
# Set to "0" to disable auto purge feature
#autopurge.purgeInterval=1

## Metrics Providers
#
# https://prometheus.io Metrics Exporter
#metricsProvider.className=org.apache.zookeeper.metrics.prometheus.PrometheusMetricsProvider
#metricsProvider.httpPort=7000
#metricsProvider.exportJvmInfo=true
```



- 4) Cambiamos la variable en bin/zkEnv.cmd tal como esta en la imagen

```
set JAVA="%JAVA_HOME%\bin\java"
```

Y quedaría listo para ejecutarse.

Servidor Spring Cloud Gateway

- 1) Desde Spring tool Suite creamos un nuevo proyecto Spring con gradle y las siguientes dependencias de Spring Cloud

The image shows two screenshots from the Spring Tool Suite. The left screenshot is the 'New Spring Starter Project' dialog. It has fields for 'Service URL' (https://start.spring.io), 'Name' (Gateway), 'Location' (D:\universidad\ultimo semestre\Distribuida 2020\ejemplos\STS\Gate), 'Type' (Gradle (Buildship 3.x)), 'Packaging' (Jar), 'Java Version' (11), 'Language' (Java), 'Group' (com.example), 'Artifact' (Gateway), 'Version' (0.0.1-SNAPSHOT), and 'Description' (Demo project for Spring Boot). The 'Package' field is set to com.example.demo. The right screenshot is the 'New Spring Starter Project Dependencies' dialog. It shows 'Spring Boot Version' as 2.3.3. Under 'Frequently Used', 'Apache Zookeeper Discover' and 'Gateway' are checked. Under 'Available', 'Spring Cloud Circuit Breaker', 'Spring Cloud Config', 'Spring Cloud Discovery', 'Spring Cloud Messaging', 'Spring Cloud Routing' (with 'Gateway' checked), 'Zuul (Maintenance)', 'Ribbon (Maintenance)', 'OpenFeign', and 'Cloud LoadBalancer' are listed. The 'Selected' list on the right shows 'Apache Zookeeper Discovery' and 'Gateway'.

- 2) la clase Main quedaría de la siguiente manera por defecto

```
@SpringBootApplication
public class GatewayApplication {

    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }

}
```

- 3) Creamos la clase GatewayDiscoveryConfiguration anotamos con @Configuration para declarar que esta clase proporciona uno o varios métodos de @Bean y puede ser procesada por el contenedor Spring para generar definiciones de bean y solicitudes de servicio para esos beans en tiempo de ejecución y @EnableDiscoveryClient sirve para descubrir múltiples

implementaciones de Discovery como eureka, consul, y en nuestro caso descubrir los servicios de ZooKeeper.

```
@Configuration
@EnableDiscoveryClient
public class GatewayDiscoveryConfiguration {

    @Bean
    public DiscoveryClientRouteDefinitionLocator
        discoveryClientRouteLocator( ReactiveDiscoveryClient discoveryClient, DiscoveryLocatorProperties properties ) {

        return new DiscoveryClientRouteDefinitionLocator( discoveryClient, properties );
    }
}
```

Modificamos el application.properties de la siguiente manera en server.port correrá nuestro Gateway , además hacemos que nuestro Gateway no se registre en el servidor de registro zookeeper y desactivamos el balanceador de ribbon para que use el balanceador de Spring cloud Load Balancer.

```
spring.application.name=spring-gateway-distribuida

#Activar la búsqueda automática de servicios
spring.cloud.gateway.discovery.locator.enabled=true

#desactivar RIBBON, dejar el balanceador de SPRING
spring.cloud.loadbalancer.ribbon.enabled=false

#No registrar en Zookeeper
spring.cloud.service-registry.auto-registration.enabled=false

server.error.whitelabel.enabled=false

server.port = 9091
```

7. Conclusiones:

- El desarrollo de Aplicaciones REST con el framework TomEE resultó darnos un reto ya que en los días en que se desarrollaron los servidores toda la página, documentación y descargas de TomEE se cayeron, pero logramos encontrar la información suficiente en otras páginas. Al final resultó ser muy parecido a un apache Tomcat en cuanto a configuración de puertos y base de datos.
- El servidor de registro y configuración Apache Zookeeper resultó ser muy completo ya que incluye soporte directo para spring cloud y se integró de manera correcta con nuestro Gateway. además que nuestro Gateway incluye la última versión de Spring Cloud Load Balancer, como futuro trabajo podríamos implementar el Monitoreo de Zookeeper con Prometheus o Grafana que se integran de forma predeterminada con zookeeper.

8. Referencias:

Documentación oficial de TomEE:

- <http://tomee.apache.org/apache-tomee.html>
- <https://tomee.apache.org/tomee-8.0/docs/>

Documentación oficial de ZooKeeper:

- <https://zookeeper.apache.org/doc/current/zookeeperOver.html>

Documentación oficial de Spring Cloud Gateway

- <https://docs.spring.io/spring-cloud-gateway/docs/2.2.5.RELEASE/reference/html/>
- Yermolaiev, L. T. D. S. O. (2020, 15 agosto). Managing configuration of a distributed system with Apache ZooKeeper. Recuperado 15 de septiembre de 2020, de <https://sysgears.com/articles/managing-configuration-of-distributed-system-with-apache-zookeeper/>
- Trivedi, N. (2019, 5 junio). Spring Cloud Gateway Tutorial - Niral Trivedi. Recuperado 14 de septiembre de 2020, de <https://medium.com/@niral22/spring-cloud-gateway-tutorial-5311ddd59816>
- Villapecellín, J. L. R. (2020, 27 julio). Introducción a ZooKeeper. Recuperado 15 de septiembre de 2020, de <https://www.adictosaltrabajo.com/2016/07/03/introduccion-a-zookeeper/>
- Long, J. (2020, 25 marzo). Spring Tips: Spring Cloud Loadbalancer. Recuperado 14 de septiembre de 2020, de <https://spring.io/blog/2020/03/25/spring-tips-spring-cloud-loadbalancer>