



# Universidad Central del Ecuador

## Programación Distribuida

Título: Informe de descripción de  
herramientas y Tecnologías e  
Implementación en el proyecto



Nombre: Alexander Columba



## Contenido

1. Objetivo .....	3
2. Objetivos Específicos .....	3
3. Marco Teórico Herramientas Tecnológicas .....	3
3.1. Quarkus .....	3
3.2. Quarkus Datasources .....	4
3.3. Agroal .....	4
3.4. Hibernate ORM y Panache: .....	4
3.5. Patrón de Registro Activo .....	5
3.6. Patrón de Repositorio .....	5
4. Desarrollo de las Aplicaciones: .....	6
4.1. Servidor 1: Customer .....	6
4.2. Servidor 2: Orders .....	10
4.3. Creación del Cliente Web .....	16
4.4. Desplegar Servidores en Heroku .....	19
5. Ejecución de la Aplicación en heroku .....	20
6. Conclusiones: .....	21
7. Referencias: .....	21



## 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 Quarkus, panaché y agroal.

## 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
- Conocer las bases teóricas y su implementación de las herramientas tecnológicas quarkus, panaché y agroal
- Deployar todos las aplicaciones en heroku.

## 3. Marco Teórico Herramientas Tecnológicas

### 3.1. Quarkus

Quarkus es un framework de Java nativo de Kubernetes diseñado para GraalVM (una Máquina Virtual que es capaz de ejecutar código de diversos lenguajes de programación), elaborado a partir de las mejores bibliotecas y estándares Java. El objetivo de Quarkus es hacer de Java una plataforma líder en Kubernetes y entornos sin servidor, al tiempo que ofrece a los desarrolladores un modelo de programación reactivo e imperativo unificado para abordar de manera óptima una gama más amplia de arquitecturas de aplicaciones distribuidas.

La solución de inyección de dependencias de Quarkus se basa en CDI (inyección de dependencias y contextos) e incluye un marco de extensión para ampliar las funciones y configurar, iniciar e integrar un marco en las aplicaciones.

#### **Algunas Características:**

##### **Unifica imperativo y reactivo**

La mayoría de los desarrolladores de Java están familiarizados con el modelo de programación imperativo y les gustaría utilizar esa experiencia al adoptar una nueva plataforma. Al mismo tiempo, los desarrolladores están adoptando rápidamente un modelo nativo, basado en eventos, asíncrono y reactivo en la nube para abordar los requisitos empresariales y crear aplicaciones altamente simultáneas y con capacidad de respuesta. Quarkus está diseñado para reunir sin problemas los dos modelos en la misma plataforma, lo que resulta en un fuerte apalancamiento dentro de una organización.



## Lo mejor de las bibliotecas y estándares

Quarkus se diseñó para funcionar con las bibliotecas, los marcos y los estándares de Java conocidos, como Eclipse MicroProfile, Apache Kafka, RESTEasy (JAX-RS), Hibernate ORM (JPA), Spring, Infinispan, Camel y muchos más.

## Configuración unificada y sencilla

Toda la configuración en un único archivo de propiedades además permite que los desarrolladores puedan verificar de inmediato el efecto de los cambios en el código y solucionarlos rápidamente. Posee la capacidad de creación sencilla de archivos ejecutables nativos

### 3.2. Quarkus Datasources

La forma habitual de obtener conexiones a una base de datos es utilizar una fuente de datos y configurar un controlador JDBC. Pero también puede preferir utilizar un controlador reactivo para conectarse a su base de datos de forma reactiva.

Para JDBC, la implementación de pool de conexiones y fuente de datos preferida es **Agroal**. Para reactivos, usa los controladores reactivos Vert.x

Ambos se configuran mediante una configuración unificada y flexible.

### 3.3. Agroal

es una implementación de grupo 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.

Se puede agregar las extensiones Agroal como por ejemplo: jdbc-mariadb, jdbc-mssql, jdbc-mysql, or jdbc-postgresql

Teniendo así por ejemplo el archivo de configuración :

```
quarkus.datasource.db-kind=postgresql
quarkus.datasource.username=<your username>
quarkus.datasource.password=<your password>

quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/hibernate_orm_test
quarkus.datasource.jdbc.min-size=4
quarkus.datasource.jdbc.max-size=16
```

### 3.4. Hibernate ORM y Panache:

Hibernate es una implementación de la especificación JPA, esta herramienta nos permite mapeo objeto-relacional (ORM) que facilita el mapeo de atributos en una base de datos tradicional, y el modelo de objetos de una aplicación mediante archivos XML o anotaciones en los beans de las entidades que permiten establecer estas relaciones. Agiliza la relación

entre la aplicación y nuestra base de datos SQL, de un modo que optimiza nuestro flujo de trabajo evitando caer en código repetitivo.

Panache ofrece una nueva capa encima de este marco, este nos permite ampliar la funcionalidad de nuestras entidades y repositorios conocidos como DAOs con algunas funcionalidades que nos proporciona automáticamente.

Por ejemplo el método listAll() que nos sirve para consultar todos los datos de nuestra tabla este toma fragmentos de consultas HQL y adapta el resto, esto lo conciso y legible

Panache nos permite el uso de patrón de registro activo y de patrón de repositorio más clásico

### 3.5. Patrón de Registro Activo

El patrón de diseño Active Record fue nombrado por Martin Fowler en 2003 en su libro *Patterns of Enterprise Application Architecture*.

El registro activo es un enfoque para acceder a los datos en una base de datos relacional. Una tabla de la base de datos se ajusta en una clase; por lo tanto, una instancia de objeto está vinculada a una sola fila en la tabla. Después de la creación de un objeto, una nueva fila se agrega a la tabla al guardar. Cualquier objeto cargado obtiene su información de la base de datos; cuando se actualiza un objeto, la fila correspondiente en la tabla también se actualiza. La clase contenedora implementa métodos o propiedades de acceso para cada columna en la tabla.

<u>OO Language Feature</u>		<u>Relations DB Item</u>
Classes	→	Tables
Objects	→	Records (Rows in a Table)
Attributes	→	Record Values (Columns in a Table)

El patrón le permite realizar operaciones CRUD sin preocuparse por la tecnología de base de datos subyacente específica (por ejemplo, SQLite, MySQL, PostgreSQL, SQL Server, Oracle, etc.).

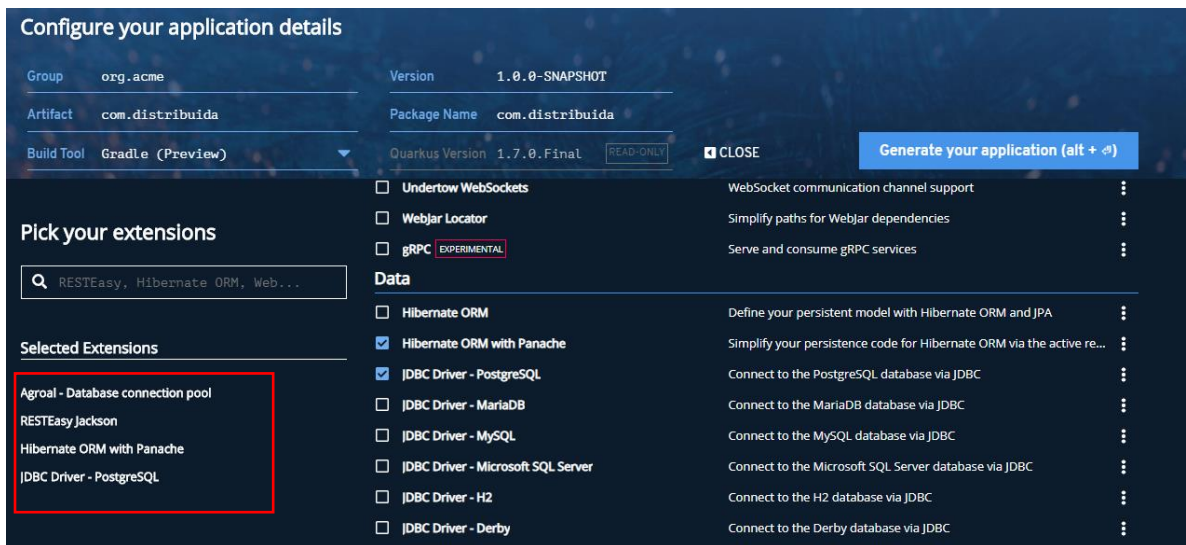
### 3.6. Patrón de Repositorio

Si usamos este patrón debemos definir la entidad JPA regular. Todas las operaciones que están definidas en PanacheEntityBase están disponibles en el repositorio, por lo que usarlo es exactamente lo mismo que usar el patrón de registro activo, excepto que se debe inyectar para esto se debe hacer que la clase implemente a PanacheRepository.

## 4. Desarrollo de las Aplicaciones:

### 4.1. Servidor 1: Customer

- 1) Como primer paso nos dirigimos a <https://code.quarkus.io/> en la pagina elegimos todas las dependencias que necesitamos para nuestro proyecto. Una vez hecho esto podemos descargarnos el código generado con las dependencias de gradle. Por defecto está marcada la dependencia RESTEasy JAX-RS.



The screenshot shows the 'Configure your application details' page of the Quarkus Code Generator. The 'Group' is 'org.acme', 'Artifact' is 'com.distribuida', 'Build Tool' is 'Gradle (Preview)', and 'Version' is '1.0.0-SNAPSHOT'. The 'Package Name' is 'com.distribuida'. The 'Quarkus Version' is '1.7.0.Final'. The 'Selected Extensions' list includes 'Agroal - Database connection pool', 'RESTEasy Jackson', 'Hibernate ORM with Panache', and 'JDBC Driver - PostgreSQL'. The 'Data' section shows 'Hibernate ORM with Panache' and 'JDBC Driver - PostgreSQL' selected.

Si solo se desea la dependencia gradle damos clic en la librería y seleccionamos copiar para gradle.

Una vez hecho esto se debe abrir en el IDE en este caso STS he importamos el proyecto gradle, una vez realizado esto nos queda el archivo build.gradle como el siguiente:

```
plugins {
    id 'java'
    id 'io.quarkus'
}

repositories {
    mavenLocal()
    mavenCentral()
}

dependencies {
    implementation 'io.quarkus:quarkus-jdbc-postgresql'
    implementation 'io.quarkus:quarkus-agroal'
    implementation 'io.quarkus:quarkus-resteasy-jackson'
    implementation 'io.quarkus:quarkus-hibernate-orm-panache'
    implementation enforcedPlatform("${quarkusPlatformGroupId}:${quarkusPlatformArtifactId}:${quarkusPlatformVersion}")
    implementation 'io.quarkus:quarkus-resteasy'
}

group 'com.distribuida'
version '1.0.0-SNAPSHOT'
```

- 2) Como segundo paso procedo a configurar la conexión a la base de datos en el archivo application.properties ubicado dentro de src/main/resources.



```
build.gradle application.properties
1 quarkus.datasource.db-kind = postgresql
2 quarkus.datasource.username = postgres
3 quarkus.datasource.password = admin
4 quarkus.datasource.jdbc.url = jdbc:postgresql://localhost:5432/Prueba
5
```

### 3) Crear la entidad y hacemos uso de Hibernate con Panache

Creamos la entidad Customer con sus atributos no es necesario crear getters y setters ya que cuando se llama a customer.name realmente se llamara a su método getName() y de manera similar con su setter.

```
import java.io.Serializable;

//Usando patron de registro activo

//para definir una entidad Panache se debe anotar con @Entity y extender de PanacheEntity o Panache EntityBase
@Entity
public class Customer extends PanacheEntityBase implements Serializable{

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

Para definir una entidad se anota con @Entity y extendemos de PanacheEntityBase para seguir el patrón de registro activo, esto nos proporciona métodos muy útiles para consultas y transacciones en la base de datos, además de poder personalizar nuestro Id con @Id y @GeneratedValue para que el id sea autoincremental

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

```
import com.distribuida.entidades.Customer;

public interface ServicioCustomer {

    void 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, accedemos a los métodos desde la Entidad

```
@ApplicationScoped
public class ServicioCustomerImpl implements ServicioCustomer{

    //metodo para insertar customer
    @Override
    public void insertarCustomer(Customer p) {
        Customer.persist(p);
    }

    //metodo para obtener todos los customers
    @Override
    public List<Customer> obtenerCustomer() {
        // el metodo list() toma fragmentos de consultas HQL y adapta el resto esto lo conciso y legible
        List<Customer> listaCustomers = Customer.listAll();
        return listaCustomers;
    }

    //metodo para eliminar un customers po su id
    @Override
    public void eliminarCustomer(Integer id) {
        Long id2=Long.valueOf(id);
        Customer.deleteById(id2);
    }

    //metodo para actualizar un customer
    @Override
    public void actualizarCustomer(Customer p) {
        //update("name = "+p.getName()+",surname"+p.getSurname()+ "where id = ?1", p.getId());
        Customer.update("name = ?1,surname=?2 where id = ?3",p.name,p.surname, p.id);
    }
}
```

Como observamos la misma Clase Customer ahora nos provee de varios métodos muy prácticos y sencillos de implementar como el `Customer.listAll()`; el cual obtiene todos los customers de la base de datos , `Customer.update` para actualizar, `Customer.persist` para insertar , `Customer.find` para buscar por un atributo y `Customer.deleteById` para eliminar por id entre otros .

Ademas Panache nos permite también trabajar con streams de java 8 para hacer consultas y otras operaciones, por ejemplo: obtener una lista de customers por su apellido

```
//metodo para obtener una lista de customers por su Apellido
@Override
public List<Customer> obtenerCustomerporApellido(String surname) {
    try (Stream<Customer> cust = Customer.streamAll()) {

        List<Customer> listaApellidos = cust
            .filter( n -> n.surname.equals(surname) )
            .collect(Collectors.toList());

        return listaApellidos;
    }
}
```

#### 4) Crear las clases RestApplication y CustomerRest



```
*import javax.ws.rs.ApplicationPath;

// Path es el mapeo de nuestros servicios
@Path("/")
public class RestApplication extends Application{

}
```

```
@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.

Se debe usar la anotación @Transactional en los métodos que alteren de alguna manera la base de datos como lo es en el caso de insertar , actualizar y eliminar. Por ejemplo :

```
@Transactional
@POST
@Path("/crear")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response crearCustomer(Customer p) {
    //el id debe ser null ya que se autogenera automaticamente
    if (p.id != null) {
        throw new WebApplicationException("el id se establecio de forma no valida.", 422);
    }
    serviCustomer.insertarCustomer(p);
    return Response.ok(p).build();
}

@Transactional
@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();
}
```

```
@Transactional
@DELETE
@Produces(MediaType.APPLICATION_JSON)
@Path("/eliminar/{id}")
public Response eliminarAlbum(@PathParam("id") Integer id) {
    Long id2=Long.valueOf(id);
    Customer cust = Customer.findById(id2);

    if (cust == null) {
        throw new WebApplicationException("Customer con id : " + id + " no existe.", 404);
    }
    serviCustomer.eliminarCustomer(id);
    return Response.ok().build();
}

@GET
@Path("/customersbySurname/{surname}")
@Produces(MediaType.APPLICATION_JSON)
public List<Customer> customerPorSurname(@PathParam("surname") String surname) {
    List<Customer> p = serviCustomer.obtenerCustomerporApellido(surname);
    if (p == null) {
        throw new WebApplicationException("Customer con id : " + surname + " no existe.", 404);
    }
    return p;
}
```

## 4.2. Servidor 2: Orders

- 1) Creamos el proyecto gradle igual que el anterior se agrega adicionalmente la dependencia de RESTeasy Client para la comunicación remota del servidor 1 con el servidor 2.

```
dependencies {
    implementation 'io.quarkus:quarkus-jdbc-postgresql'
    implementation 'io.quarkus:quarkus-agroal'
    implementation 'io.quarkus:quarkus-resteasy-jackson'
    implementation 'io.quarkus:quarkus-hibernate-orm-panache'
    implementation enforcedPlatform("${quarkusPlatformGroupId}:${quarkusPlatformArtifactId}:${quarkusPlatformVersion}")
    implementation 'io.quarkus:quarkus-resteasy'
    compile group: 'org.jboss.resteasy', name: 'resteasy-client', version: '4.5.6.Final'
}
```

- 2) Al igual que el anterior servidor se debe colocar la configuración de la conexión a base de datos en el applications.properties

```
quarkus.datasource.db-kind = postgresql
quarkus.datasource.username = postgres
quarkus.datasource.password = admin
quarkus.datasource.jdbc.url = jdbc:postgresql://localhost:5432/Prueba
```

- 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. Además para seguir con el patrón active record extendemos de `PanacheEntityBase` para que le dote a la entidad de métodos útiles para manipulación y consulta a base de datos, así como crear id personalizados. 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.

```
@Entity
@Table(name="orders")
public class Ordenes extends PanacheEntityBase implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Long id;
    public String item ;
    public Double price ;
    public Long customer_id;
    @Transient //indica que este atributo no es persistente no es tomado en cuenta a la hora de persistir el Objeto
    public String datosCliente ;
```

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);
```

Fijémonos en la Implementación aquí accedemos a los métodos de `PanacheEntityBase` desde la Entidad `Ordenes`

```
@ApplicationScoped
public class ServicioOrdenesImpl implements ServicioOrdenes{

    // metodo para insertar Ordenes
    @Override
    public void insertarOrdenes(Ordenes P) {
        Ordenes.persist(P);
    }

    //metodo para obtener todas las Ordenes
    @Override
    public List<Ordenes> obtenerOrdenes() {
        List<Ordenes> listaOrdenes = Ordenes.listAll();
        //List<Ordenes> listaOrdenes = list("select o.id, o.item, o.price, o.customer_id from Ordenes o");

        return listaOrdenes;
    }

    //metodo para eliminar una orden por Id
    @Override
    public void eliminarOrdenes(Integer id) {
        Long id2=Long.valueOf(id);
        Ordenes.deleteById(id2);
    }

    //metodo para actualizar una orden
    @Override
    public void actualizarOrdenes(Ordenes p) {
        Ordenes.update("item = ?1, price=?2, customer_id=?3 where id = ?4",p.item,p.price,p.customer_id, p.id);
    }

    //metodo para Obtener Ordenes por id
    @Override
    public Ordenes obtenerOrdenesPorId(Integer id) {
        Long id2=Long.valueOf(id);
        return Ordenes.find("id", id2).firstResult();
    }
}
```

- 4) 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.

```
package com.distribuida.rest;

import javax.ws.rs.ApplicationPath;

// Path es el mapeo de nuestros servicios
@ApplicationPath("/")
public class RestApplication extends Application{

}
```

```
@Path("/orders")
@ApplicationScoped

public class OrdenesRest {

    @Inject
    private ServicioOrdenes servicio;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Ordenes> listaOrdenes() throws SQLException{

        List<Ordenes> lista = servicio.obtenerOrdenesCustomer();

        return lista;
    }

    @GET
    @Path("/order/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Ordenes ordenPorId(@PathParam("id") Integer id) {
        Ordenes p = servicio.obtenerOrdenesPorId(id);
        if (p == null) {
            throw new WebApplicationException("Orden con id : " + id + " no existe.", 404);
        }
        return p;
    }
}

@Transactional
@POST
@Path("/crear")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response crearOrden(Ordenes p) {
    servicio.insertarOrdenes(p);
    //el id debe ser null ya que se autogenera automaticamente
    if (p.id != null) {
        throw new WebApplicationException("el id se establecio de forma no valida.", 422);
    }
    return Response.ok(p).build();
}

@Transactional
@PUT
@Path("/actualizar")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response actualizarOrden(Ordenes p) {
    if (p.id == null) {
        throw new WebApplicationException("el id se establecio de forma no valida.", 422);
    }
    servicio.actualizarOrdenes(p);
    return Response.ok(p).build();
}

@Transactional
@DELETE
@Produces(MediaType.APPLICATION_JSON)
@Path("/eliminar/{id}")
public Response eliminarOrden(@PathParam("id") Integer id) {
    Long id2=Long.valueOf(id);
    Ordenes cust = Ordenes.findById(id2);
    if (cust == null) {
        throw new WebApplicationException("Customer con id : " + id + " no existe.", 404);
    }
    servicio.eliminarOrdenes(id);
    return Response.ok().build();
}
```

##### 5) Comunicación entre servidor1 y servidor2 mediante RestEasy Client

Para que la tabla Ordenes tenga los datos del cliente como su nombre y apellido no solo su id. Se debe traer los datos del cliente al servidor 2 Ordeners , para esto creamos en el paso 3 el atributo datosCliente anotado con @Transient.

Usando RESTeasy proxy framework : este framework cliente crea una solicitud HTTP que utiliza para invocar en un servicio web REST remoto.

Este servicio remoto no tiene que ser un servicio JAX-RS y puede ser cualquier recurso web que acepte solicitudes HTTP. Tomado de

[https://docs.jboss.org/resteasy/docs/4.3.1.Final/userguide/html/RESTEasy\\_Client\\_Framework.html](https://docs.jboss.org/resteasy/docs/4.3.1.Final/userguide/html/RESTEasy_Client_Framework.html)

Creamos la Clase Customer sin anotaciones solo con sus setters y setters

```
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;  
    }  
}
```

RESTEasy tiene un framework proxy de cliente que le permite utilizar anotaciones JAX-RS para invocar en un recurso HTTP remoto. La forma en que funciona es que se escribe una interfaz Customer proxy y utiliza anotaciones JAX-RS en los métodos y la interfaz: indicamos con @GET para indicar que es un método de consulta y @Consumes para indicar que la solicitud recibida se aceptara en formato JSON

```
import com.distribuida.entidades.Customer;  
  
public interface CustomerProxy {  
    @GET  
    @Consumes(MediaType.APPLICATION_JSON)  
    public List<Customer> listarTodos();  
}
```

Su implementación: inyectamos a CustomerProxy



```
@ApplicationScoped
public class CustomerProxyImpl {

    @Inject private CustomerProxy proxy;

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

}
```

RETEasy tiene una API simple basada en Apache HttpClient. Genera un proxy y, a continuación, puede invocar métodos en el proxy. El método invocado se traduce a una solicitud HTTP en función de cómo anote el método y se haya publicado en el servidor. Así es como configuraría esto:

```
@ApplicationScoped
public class CustomerProducer {

    private static final String URL_SERVIDOR1 = "https://columba-distribuida41.herokuapp.com/customers";

    @Produces
    @ApplicationScoped
    public CustomerProxy getProxy() {
        ResteasyClient client = (ResteasyClient)ClientBuilder.newClient();
        ResteasyWebTarget target = client.target(UriBuilder.fromPath(URL_SERVIDOR1));
        CustomerProxy proxy = target.proxy(CustomerProxy.class);
        return proxy;
    }

}

@Inject private CustomerProxyImpl servicioCliente;
```

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;
}
```

Adicionalmente hice otra lista que me retorna las Ordenes con los datos del customer pero filtrada por Id del customer esto me servirá para la aplicación web cliente pueda filtrar datos de la tabla

```
//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;
}
```

Y por último creamos los métodos Rest correspondientes en OrdenesRest

```
@GET
@Path("/orderbyCustomerId/{id}")
@Produces(MediaType.APPLICATION_JSON)
public List<Ordenes> ordenconCustomerbyId(@PathParam("id") Integer id) {
    List<Ordenes> p = servicio.obtenerOrdenesbyIdCustomer(id);
    if (p == null) {
        throw new WebApplicationException("Customer con id : " + id + " no existe.", 404);
    }
    return p;
}
```

### 4.3. 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("/{order}/{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

```
@ApplicationScoped
public class CustomerProducer {
    private static final String URL_SERVIDOR1 = "https://columba-distribuida41.herokuapp.com/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 = "https://columba-distribuida42.herokuapp.com/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

<pre>@Named(value = "controladorOrdenes") @SessionScoped public class OrdenController {      @Inject private OrderProxyImpl servicioOrden;</pre>	<pre>@Named(value = "controladorCliente") @SessionScoped public class CustomerController {      @Inject private CustomerProxyImpl servicioCliente;</pre>
--	--

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

#### 4.4. Desplegar Servidores en Heroku

- 1) Tanto para el Servidor 1 como el Servidor 2 hechos en Quarkus se debe implementar en el build.gradle lo siguiente:

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

- 2) Añadir archivo Procfile en la Raíz del proyecto de la siguiente manera de ejemplo

```
web: java -Dquarkus.http.port=$PORT -jar build/servidor2-quarkus-panache-agroal-1.0.0-SNAPSHOT-runner.jar
```

- 3) Añadir el archivo system.properties en la raíz del proyecto

```
#heroku config JDK
java.runtime.version=11
```

- 4) Creamos un nuevo proyecto heroku en la pestaña de resources agregamos heroku postgres una vez hecho esto vamos a Database Credentials ahí nos proporciona todas las credenciales de nuestra base de datos y luego procedemos a crear nuestras tablas con ayuda de PgAdmin. Una vez hecho esto creamos las variables de ambiente , abrimos la pestaña de settings y luego Reval Config Vars, son los mismos nombres de la configuración de application.properties solo que con mayúsculas y reemplazando los puntos por guion bajo y quedaría listo para funcionar

Config Vars		Hide Config Vars
DATABASE_URL	postgres://cjrbezdwtnlwlb:bc105b369ca020c	
QUARKUS_DATASOURCE_DB-KIND	postgresql	
QUARKUS_DATASOURCE_JDBC_URL	jdbc:postgresql://ec2-54-147-209-121.comp	
QUARKUS_DATASOURCE_PASSWORD	bc105b369ca020cbf5e701cca03ef619194a29f55	
QUARKUS_DATASOURCE_USERNAME	cjrbezdwtnlwlb	

- 5) Nos ubicamos en la raíz del proyecto he iniciamos CMD y realizamos las siguientes instrucciones :

```
$ heroku login

Clone the repository
Use Git to clone columba-distribuida42's source code

$ heroku git:clone -a columba-distribuida42
$ cd columba-distribuida42

Deploy your changes
Make some changes to the code you just cloned

$ git add .
$ git commit -am "make it better"
$ git push heroku master
```



- 6) Para subir el cliente web en el build.gradle colocamos

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

- 7) Creamos el archivo Procfile con lo siguiente

```
1web: java $JAVA_OPTS -jar webapp-runner.jar --port $PORT build/libs/*.war
```

- 8) Copiamos el archivo webapp-runner.jar en la raíz del proyecto

- 9) Creamos un nuevo proyecto de heroku abrimos nuestro Cmd en la raíz del proyecto y ejecutamos lo siguiente

```
$ heroku login

Clone the repository
Use Git to clone columba-distribuida43's source code to yo

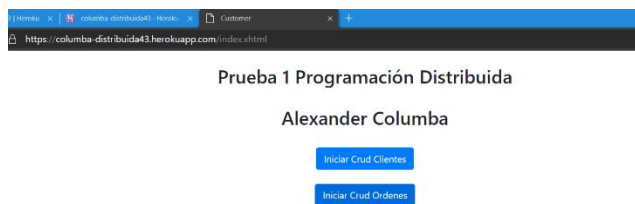
$ heroku git:clone -a columba-distribuida43
$ cd columba-distribuida43

Deploy your changes
Make some changes to the code you just cloned and deplo

$ git add .
$ git commit -am "make it better"
$ git push heroku master
```

## 5. Ejecución de la Aplicación en heroku

- 1) Primera pantalla donde podemos escoger entre el CRUD de Clientes o de Ordenes



- 2) Pantalla Que muestra el listado de los clientes ai como un pequeño buscador por apellidos , podemos actualizar, eliminar o crear nuevos clientes

### GESTION DE CLIENTES

Listado de los Cientes

[Crear Cliente](#)

Seleccione un Apellido: todos [Buscar](#)

Id	Nombre	Apellido	Opciones
1	alexander	columba	<a href="#">Editar</a> <a href="#">Eliminar</a>
2	ramiro	aguiar	<a href="#">Editar</a> <a href="#">Eliminar</a>
3	loia	suares	<a href="#">Editar</a> <a href="#">Eliminar</a>
5	gabriela	columba	<a href="#">Editar</a> <a href="#">Eliminar</a>

[Regresar menu principal](#)

### CREAR NUEVA CLIENTE

Nombre

Apellido

[Guardar](#) [Cancelar](#)

### EDITAR CLIENTE

Nombre

Apellido

[Guardar](#) [Cancelar](#)



- 3) Vista que muestra el listado de Ordenes con un buscador por Cliente. Además de todas las operaciones crud.

GESTION DE ORDENES

Listado de las Ordenes

Crear Ordenes

Seleccione un Cliente: todos

Buscar

Id	Producto	Precio	Cliente	Datos Cliente	Opciones
2	lapiz	0.35	1	alexander columba	<div>Editar</div> <div>Eliminar</div>
3	borrador	0.6	1	alexander columba	<div>Editar</div> <div>Eliminar</div>
4	cuaderno	1.36	2	ramiro aguiar	<div>Editar</div> <div>Eliminar</div>
6	carpeta	0.6	3	lola suarez	<div>Editar</div> <div>Eliminar</div>
5	escritorio	350.0	3	lola suarez	<div>Editar</div> <div>Eliminar</div>

Regresar menu principal

CREAR NUEVA ORDEN

Producto

Precio

Cliente: alexander columba

Guardar

Cancelar

EDITAR ORDEN

Producto: escritorio

Precio: 350.0

Cliente: lola suarez

Guardar

Cancelar

## 6. Conclusiones:

- El de desarrollo de Aplicaciones REST con el framework Quarkus resulto ser muy intuitivo y practico ya que desde la misma pagina nos da una extensa guía de sus librerías que incorpora y como usarlas además de ejemplos prácticos detallados.
- El acceso a base de datos con Hibernate y Panache usando el patrón Active record facilita muchas tareas, reduce el código y lo hace mas legible para los programadores, además que es muy sencillo de implementar. Incorpora además el uso de streams para hacer operaciones complejas y eficientes.
- Quarkus nos da una solución eficaz para ejecutar Java en el mundo sin servidor, microservicios, contenedores, Kubernetes, y la nube porque se ha diseñado teniendo en cuenta estos elementos. Su enfoque de contenedor primero para aplicaciones Java nativas de la nube unifica paradigmas de programación imperativos y reactivos para el desarrollo de microservicios y ofrece un conjunto extensible de bibliotecas y marcos Java empresariales.

## 7. Referencias:

- ¿Qué es Quarkus? (s. f.). Recuperado 24 de agosto de 2020, de <https://www.redhat.com/es/topics/cloud-native-apps/what-is-quarkus>



- Greene, J. (2019, 3 septiembre). Introducing Quarkus: a next-generation Kubernetes native Java framework. Recuperado 24 de agosto de 2020, de <https://developers.redhat.com/blog/2019/03/07/quarkus-next-generation-kubernetes-native-java-framework/>
- Quarkus - Simplified Hibernate ORM with Panache. (s. f.). Recuperado 24 de agosto de 2020, de <https://quarkus.io/guides/hibernate-orm-panache>
- Fowler, M. (s. f.). P of EAA: Active Record. Recuperado 24 de agosto de 2020, de <https://www.martinfowler.com/eaCatalog/activeRecord.html>
- Chapter 51. RESTEasy Client API. (s. f.). Recuperado 24 de agosto de 2020, de [https://docs.jboss.org/resteasy/docs/4.3.1.Final/userguide/html/RESTEasy\\_Client\\_Framework.html](https://docs.jboss.org/resteasy/docs/4.3.1.Final/userguide/html/RESTEasy_Client_Framework.html)
- Quarkus - Datasources. (s. f.). Recuperado 24 de agosto de 2020, de <https://quarkus.io/guides/datasource>