



UNIVERSIDAD DEL CAUCA
FACULTAD DE INGENIERÍA ELECTRÓNICA Y TELECOMUNICACIONES
PROGRAMA DE INGENIERÍA DE SISTEMAS
LABORATORIO DE INGENIERIA DE SOFTWARE II
JULIO A. HURTADO, W. LIBARDO PANTOJA Y.

PRACTICA DE LABORATORIO

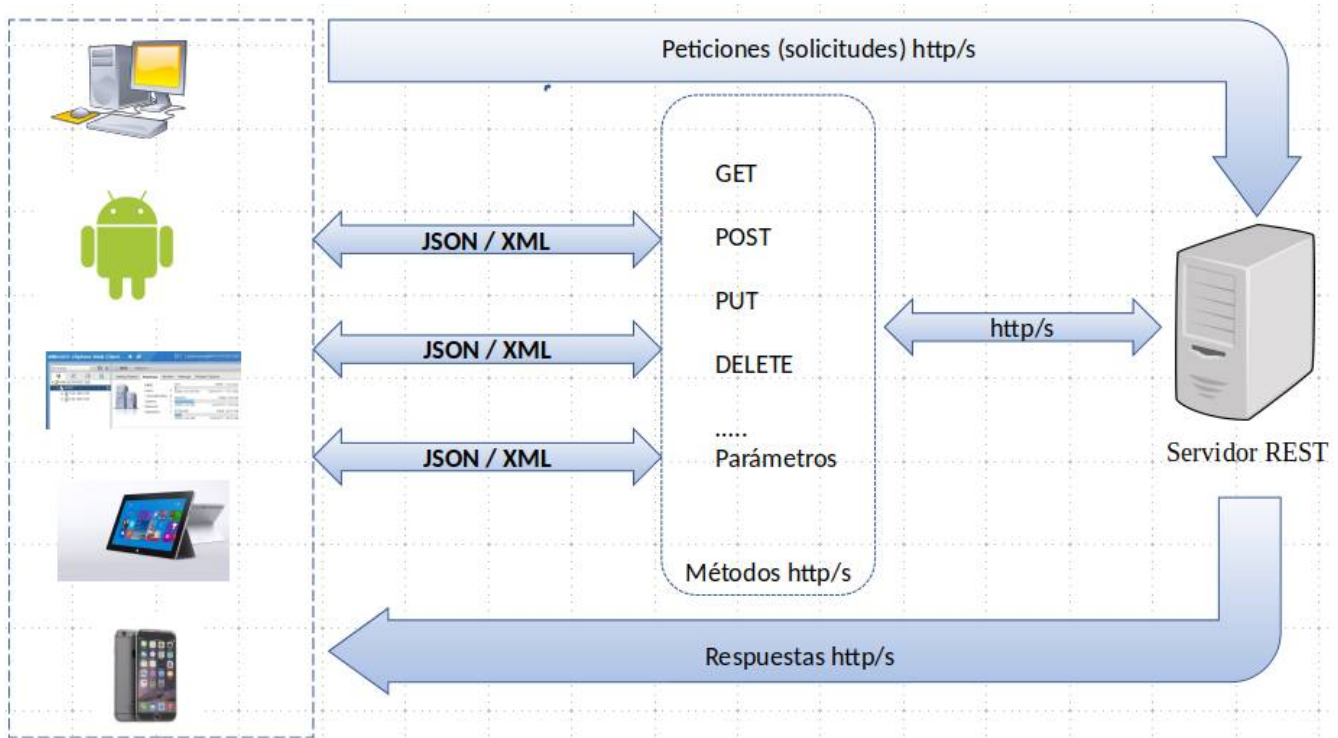
Creando una API REST

Última modificación: Septiembre 29 de 2020

OBJETIVO

- Construir una API REST sencilla en java con el fin de conocer su funcionamiento básico.
- Crear un cliente desktop que consuma esta API REST.

¿Qué es una API de REST?



Una API REST es una aplicación web en el lado del back-end. Tiene una serie de rutas y métodos que interactúan con la base de datos y hacen lógica. Por ejemplo, un método que autentique un usuario, otro

método que registre un usuario, otro método que liste productos, etc. Estos métodos pueden ser **consumidos** por cualquier tipo de cliente, ya sea web, móvil, de escritorio o cualquier otro tipo que se comunique con el protocolo **HTTP**. Cada método es invocado por uno de los verbos de HTTP:

- GET (consulta y lectura)
- POST (crear)
- DELETE (eliminar)
- PUT (editar).

El back-end y los clientes se comunican enviando mensajes ya sea XML o JSON. JSON es más recomendable actualmente por ser un formato simple.

Este tipo de aplicaciones es muy utilizado en la actualidad. Hoy en día las aplicaciones móviles, web están en auge. Y en su arquitectura es muy común que consuman servicios web REST.

REST son las siglas de **R**epresentational **S**tate **T**ransfer (Transferencia de Estado Representacional), un concepto que establece una serie de restricciones importantes para definir a los sistemas que responden a sus principios. Las restricciones son las siguientes:

- Conexión cliente-servidor libre. El cliente no necesita saber los detalles de la implementación del servidor, y este tampoco debe preocuparse por cómo se usan los datos que envía.
- Cada petición enviada al servidor es independiente.
- Compatibilidad con un sistema de almacenamiento en caché.
- Cada recurso del servicio REST debe tener una única dirección, manteniendo una interfaz genérica.
- Disposición de diferentes capas para la implementación del servidor.

Creando el proyecto

Se va a crear una aplicación back-end que suministre el acceso a datos de productos. A esta aplicación la llamaremos **Simple-API-REST**. Posteriormente, se creará una aplicación cliente de escritorio (**products-client**) que consuma los servicios de dicha API. Sin más explicaciones, comencemos...

En Netbeans 12 o superior, ir al menú File / New Project / Web Application (ver Figura 1). Luego clic en Next.

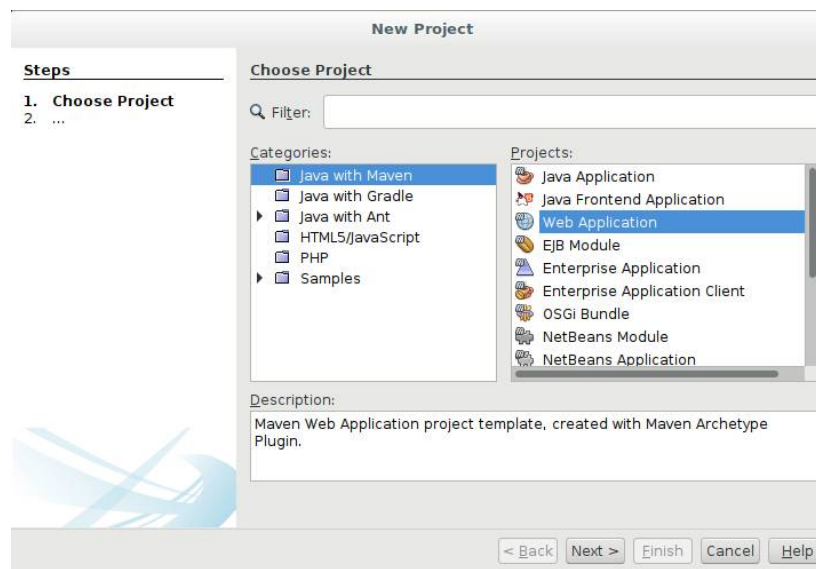


Figura 1. Creando el proyecto

Luego colocamos el nombre al proyecto, Grupo Id y Paquete acorde a la Figura 2. Luego clic en Next.

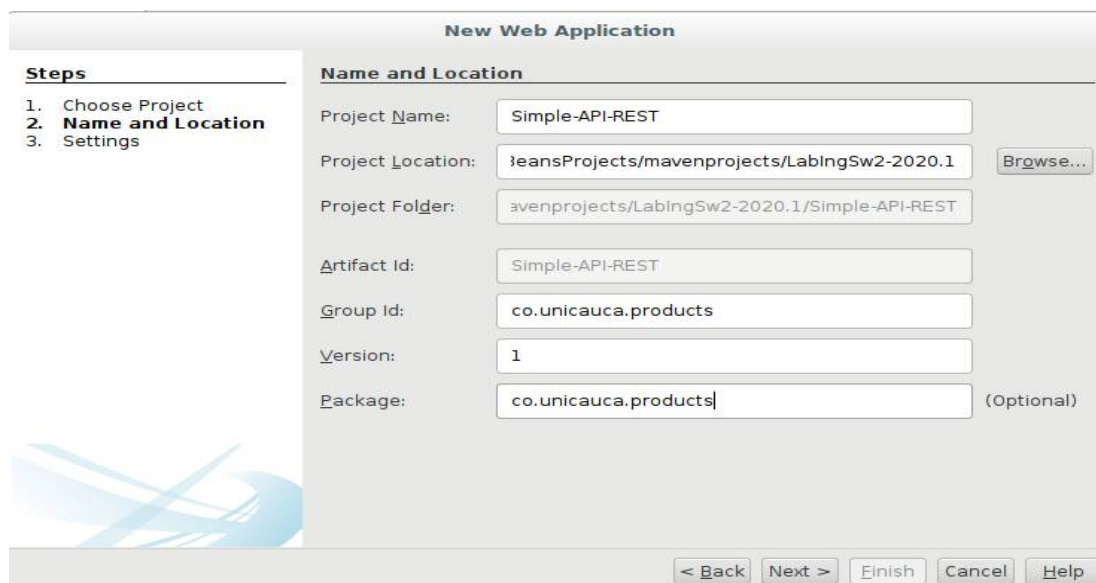


Figura 2. Datos del proyecto

En la siguiente ventana elegir el servidor de aplicaciones **Payara** (también se puede elegir GlassFish, Tomcat) y la versión 8 de Java EE (sirven las versiones 6, 7) acorde a la Figura 3 (Si Payara no aparece en el listado, se lo puede descargar haciendo clic en el botón Add).

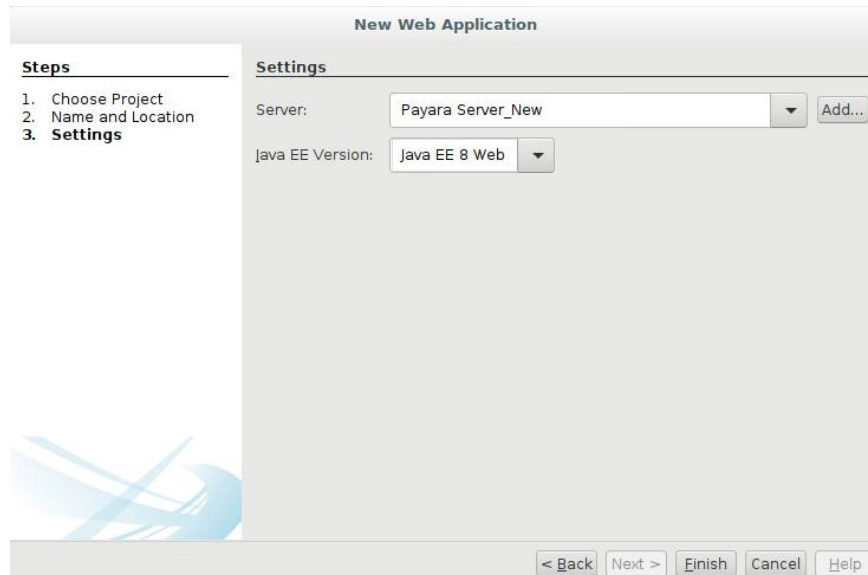


Figura 3. Opciones del servidor

Finalmente clic en Finish.

Se debe crear un proyecto con la estructura de la Figura 4.

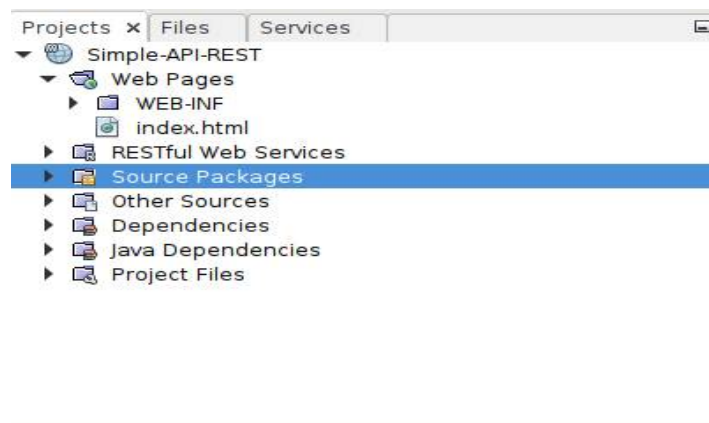


Figura 4. Estructura del proyecto generado por Netbeans

Lo primero que debemos de hacer para iniciar nuestra API REST es indicarle el Path base desde el cual estará respondiendo nuestra API. Este path corresponde a la URL a partir de la cual se expondrá nuestros servicios. Para lograr esto, será necesario crear una clase que extienda de “Application”, esta clase puede llamarse como sea y puede colocarse en cualquier paquete, lo único importante es que extienda de Application y defina la anotación @ApplicationPath. En nuestro caso crearemos la clase **AppConfiguration** en el package raíz del proyecto:

```
package co.unicauca.products;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

/**
 * Clase que indice el Path base desde el cual estará respondiendo la API Rest.
 * Este path corresponde a la URL a partir de la cual se expondrá nuestros
 * servicios. Para lograr esto, será necesario crear una clase que extienda de
 * “Application”, esta clase puede llamarse como sea y puede colocarse en
 * cualquier paquete.
 *
 * @author Libardo, Julio
 */
@ApplicationPath("product-service")
public class AppConfiguration extends Application {

}
```

La Figura 5 muestra las clases y paquetes que se crearán en este proyecto:

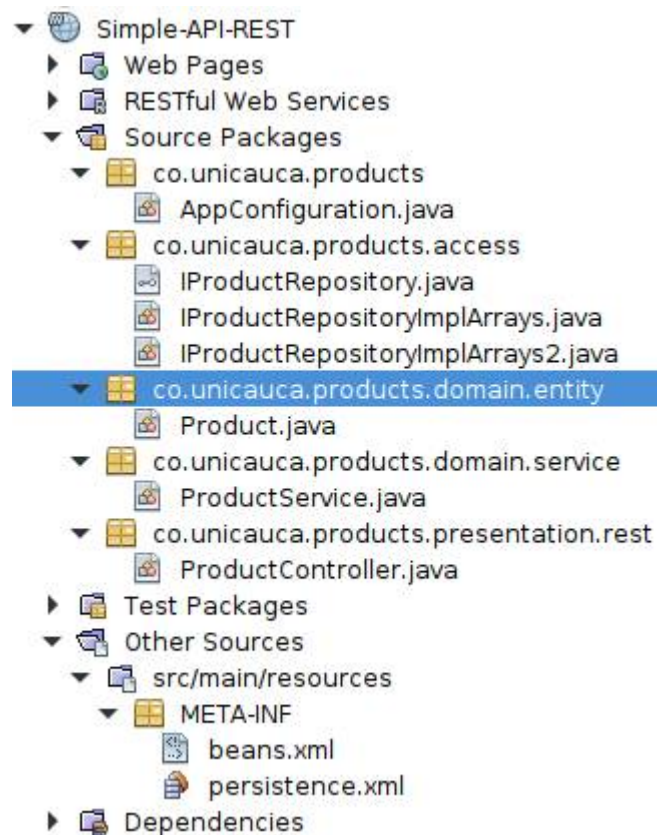


Figura 5. Paquetes y clases a crear en el proyecto

Ahora creamos la clase Product de nuestro modelo:

```
package co.unicauca.products.domain.entity;

/**
 * Representa un producto de la tienda
 *
 * @author Libardo, Julio
 */
public class Product {

    private Integer id;

    private String name;

    private Double price;

    public Product(Integer id, String name, Double price) {
        this.id = id;
        this.name = name;
    }
}
```

```
        this.price = price;
    }

    public Product() {
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Double getPrice() {
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }

    @Override
    public String toString() {
        return "Product{" + "id=" + id + ", name=" + name + ", price=" + price +
    }';
    }
}
```

Ahora creamos las clases de la capa de acceso a datos. Se creará la Interface del Repositorio y dos implementación concretas mediante arreglos.

```
package co.unicauca.products.access;

import co.unicauca.products.domain.entity.Product;
import java.util.List;

/**
 * Interface de los servicios del repositorio
 *
 * @author Libardo, Julio
 */
public interface IProductRepository {
```

```
List<Product> findAll();

Product findById(Integer id);

boolean create(Product newProduct);

boolean update(Product newProduct);

boolean delete(Integer id);
}
```

Ahora la primera implementación del repositorio:

```
package co.unicauca.products.access;

import co.unicauca.products.domain.entity.Product;
import java.util.ArrayList;
import java.util.List;
import javax.enterprise.inject.Default;

/**
 * Implementación por defecto. El framework contenedor de CDI (Contexts and
 * Dependency Injection) carga la implementación por defecto.
 *
 * @author Libardo, Julio
 */
@Default
public class IProductRepositoryImplArrays implements IProductRepository {

    /**
     * Por simplicidad los datos se cargan en un array.
     */
    public static List<Product> products;

    public IProductRepositoryImplArrays() {
        if (products == null){
            products = new ArrayList<>();
            initialize();
        }
    }

    private void initialize() {
        products.add(new Product(1, "Tv samsung", 2000000d));
        products.add(new Product(2, "Tv lg", 3000000d));
        products.add(new Product(1, "Tablet asus RESD-TD-34", 4000000d));
    }

    @Override
    public List<Product> findAll() {
        return products;
    }

    @Override
    public Product findById(Integer id) {
```



```
        for (Product prod : products) {
            if (prod.getId() == id) {
                return prod;
            }
        }
        return null;
    }

    @Override
    public boolean create(Product newProduct) {
        Product prod = this.findById(newProduct.getId());
        if (prod != null) {
            //Ya existe
            return false;
        }
        products.add(newProduct);
        return true;
    }

    @Override
    public boolean update(Product product) {
        Product prod = this.findById(product.getId());
        if (prod != null) {
            prod.setName(product.getName());
            prod.setPrice(product.getPrice());
            return true;
        }
        return false;
    }

    @Override
    public boolean delete(Integer id) {
        int i = 0;
        for (Product prod : products) {
            if (prod.getId() == id) {
                products.remove(i++);
                return true;
            }
        }
        return false;
    }
}
```

A continuación la segunda implementación del repositorio:

```
package co.unicauca.products.access;

import co.unicauca.products.domain.entity.Product;
import java.util.ArrayList;
import java.util.List;
import javax.enterprise.context.RequestScoped;
import javax.enterprise.inject.Alternative;
```

```
/**
 * Implementación del repositorio alternativa. Se puede cambiar las anotaciones
 * @Default y @Alternative al gusto.
 *
 * @author Libardo, Julio
 */
@RequestScoped
@Alternative
public class IProductRepositoryImplArrays2 implements IProductRepository {

    public static List<Product> products;

    public IProductRepositoryImplArrays2() {
        if (products == null){
            products = new ArrayList<>();
            initialize();
        }
    }

    private void initialize() {
        products.add(new Product(1, "Cama duplex", 300000d));
        products.add(new Product(2, "Sofa cama", 300000d));
        products.add(new Product(1, "Nochero", 400000d));
    }

    @Override
    public List<Product> findAll() {
        return products;
    }

    @Override
    public Product findById(Integer id) {
        for (Product prod : products) {
            if (prod.getId() == id) {
                return prod;
            }
        }
        return null;
    }

    @Override
    public boolean create(Product newProduct) {
        Product prod = this.findById(newProduct.getId());
        if (prod != null) {
            //Ya existe
            return false;
        }
        products.add(newProduct);
        return true;
    }
}
```

```
@Override
public boolean update(Product product) {
    Product prod = this.findById(product.getId());
    if (prod != null) {
        prod.setName(product.getName());
        prod.setPrice(product.getPrice());
        return true;
    }
    return false;
}

@Override
public boolean delete(Integer id) {
    int i = 0;
    for (Product prod : products) {
        if (prod.getId() == id) {
            products.remove(i++);
            return true;
        }
    }
    return false;
}
}
```

Ahora la clase ProductService que es una Fachada de acceso al negocio por parte de la presentación. Se utiliza inyección de dependencias mediante el framework de JavaEE

```
package co.unicauca.products.domain.service;

import co.unicauca.products.access.IProductRepository;
import co.unicauca.products.domain.entity.Product;
import java.util.List;
import javax.enterprise.context.RequestScoped;
import javax.enterprise.inject.Default;
import javax.inject.Inject;

/**
 * Fachada de acceso al negocio por parte de la presentación. Usa el repositorio
 * por defecto. Si no se pone @Default también funciona, pues inyecta la
 * implementación por defecto
 *
 * @author Libardo, Julio
 */
@RequestScoped
public class ProductService {
    /**
     * Inyecta una implementación del repositorio
     */
    @Inject
    @Default
    IProductRepository repo;
```

```
public ProductService() {  
    }  
  
    public List<Product> findAll() {  
        return repo.findAll();  
    }  
  
    public Product findById(int id) {  
        return repo.findById(id);  
    }  
  
    public boolean create(Product prod) {  
        return repo.create(prod);  
    }  
  
    public boolean update(Product prod) {  
        return repo.update(prod);  
    }  
  
    public boolean delete(int id) {  
        return repo.delete(id);  
    }  
}
```

Finalmente la clase que contiene nuestros servicios web REST Full. La anotación `@Path` indica la URL en la cual responderá los servicios. Esta anotación se puede poner a nivel de clase y método. En este ejemplo todos los servicios comparten el mismo Path, la acción se hace mediante la anotación GET (consultar), POST (agregar), PUT (editar), DELETE (eliminar):

```
package co.unicauca.products.presentation.rest;  
  
import co.unicauca.products.domain.service.ProductService;  
import co.unicauca.products.domain.entity.Product;  
import java.util.List;  
import javax.ejb.Stateless;  
import javax.inject.Inject;  
import javax.ws.rs.Consumes;  
import javax.ws.rs.DELETE;  
import javax.ws.rs.GET;  
import javax.ws.rs.POST;  
import javax.ws.rs.PUT;  
import javax.ws.rs.Path;  
import javax.ws.rs.PathParam;  
import javax.ws.rs.Produces;  
import javax.ws.rs.core.MediaType;  
  
/**  
 * API REST de los servicios web. Es muy simple por ahora, en otra versión se  
 * hará una API más robusta. Son nuestros servicios web. La anotación @Path  
 * indica la URL en la cual responderá los servicios. Esta anotación se puede
```

```
* poner a nivel de clase y método. En este ejemplo todos los servicios
* comparten el mismo Path, la acción se hace mediante la anotación GET
* (consultar), POST (agregar), PUT (editar), DELETE (eliminar).
*
* @author Libardo, Julio
*/
@Stateless
@Path("products")
public class ProductController {

    /**
     * Se inyecta la única implementación que hay de ProductService
     */
    @Inject
    private ProductService service;

    public ProductController() {

    }

    /**
     * Su uso desde consola mediante client url:
     * curl -X GET http://localhost:8080/Simple-API-REST/product-service/products/
     */
    @GET
    @Produces({MediaType.APPLICATION_JSON})
    public List<Product> findAll() {
        return service.findAll();
    }

    /**
     * Su uso desde consola mediante client url:
     * curl -X GET http://localhost:8080/Simple-API-REST/product-
service/products/1

    */
    @GET
    @Path("{id}")
    @Produces({MediaType.APPLICATION_JSON})
    public Product findById(@PathParam("id") int id) {
        return service.findById(id);
    }

    /**
     * Su uso desde consola mediante client url:
     * curl -X POST \
     * http://localhost:8080/Simple-API-REST/product-service/products/ \
     * -H 'Content-Type: application/json' \
     * -d '{
     *     "id":1,
     *     "name":"Nevera Lg",
     *     "price":6700000
     * }'
     */
}
```

```
@POST
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public String create(Product prod) {
    if (service.create(prod)) {
        return "{\"ok\":\"true\", \"mensaje\":\"Producto
creado\", \"errores\":\"\"}";
    } else {
        return "{\"ok\":\"false\", \"mensaje\":\"No se pudo crear el
producto\", \"errores\":\"Id ya existe\"}";
    }
}

/*
    Su uso desde consola mediante client url:
    curl -X PUT \
    http://localhost:8080/Simple-API-REST/product-service/products/\
    -H 'Content-Type: application/json' \
    -d '{
        "name":"Nevera Lg REF. JDK3-34-343",
        "price":2450000
    }'
*/
@PUT
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public String update(Product prod) {
    if (service.update(prod)) {
        return "{\"ok\":\"true\", \"mensaje\":\"Producto
modificado\", \"errores\":\"\"}";
    } else {
        return "{\"ok\":\"false\", \"mensaje\":\"No se pudo modificar el
producto\", \"errores\":\"Id no existe\"}";
    }
}

/*
    Su uso desde consola mediante client url:
    curl -X DELETE http://localhost:8080/Simple-API-REST/product-
service/products/

*/
@DELETE
@Path("/{id}")
public String remove(@PathParam("id") Integer id) {
    if (service.delete(id)) {
        return "{\"ok\":\"true\", \"mensaje\":\"Producto
borrado\", \"errores\":\"\"}";
    } else {
        return "{\"ok\":\"false\", \"mensaje\":\"No se pudo borrar el
producto\", \"errores\":\"Id no existe\"}";
    }
}
}
```

El archivo “beans.xml”

Para que funcione la inyección de dependencias con CDI (Contexts and Dependency Injection), se debe crear el archivo “beans.xml” en la carpeta “src/main/resources/META-INF/”(ver Figura 6). Incluso si este archivo no contiene ninguna directiva DI específica en absoluto, es necesario para que CDI esté en funcionamiento:

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
</beans>
```

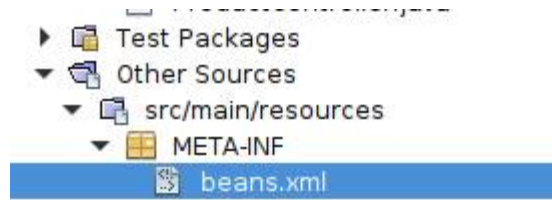


Figura 6. Ubicación del archivo beans.xml en el proyecto

Ahora se debe desplegar la aplicación, para ello, clic derecho en el proyecto / Run. Se debe abrir el navegador tal como lo muestra la Figura 7.

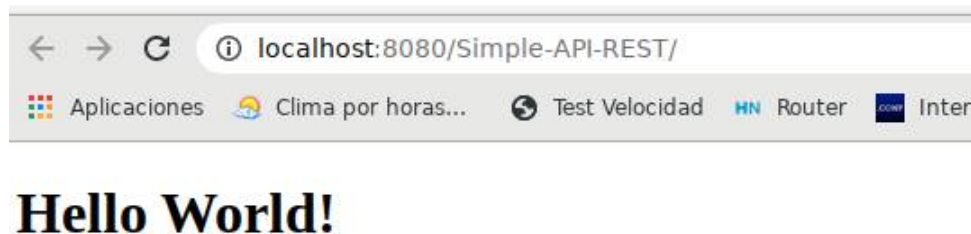


Figura 7. Ejecutando la aplicación del lado del servidor.

Probar los servicios web

Para probar los servicios se utiliza la herramienta **Postman** (también se puede utilizar el comando **curl** por consola). Descargarla de Internet e instalarla. Se recomienda crear una colección (o carpeta) llamada *pruebas* y colocar cada uno de los servicios (ver Figura 8).

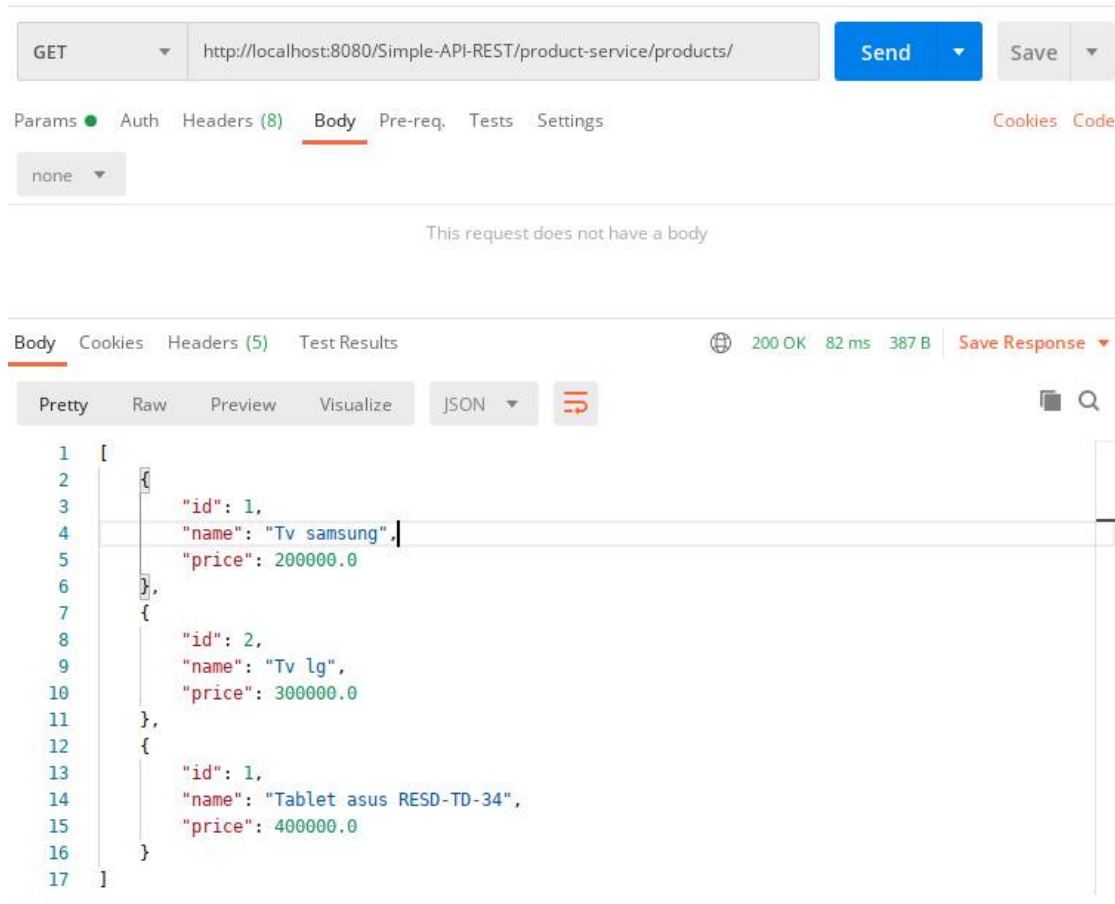


Figura 8. Probando los servicios web en postman

A continuación se colocan las URI de los servicios a invocar desde postman:

Método	URI	Descripción
GET	http://localhost:8080/Simple-API-REST/product-service/products/	Listar todos los productos
GET	http://localhost:8080/Simple-API-REST/product-service/products/1	Listar un producto en particular, en este ejemplo con id=1
POST	http://localhost:8080/Simple-API-REST/product-service/products/	Crear un producto. Se debe ir a la opcion body y escribir en formato json el producto a crear
DELETE	http://localhost:8080/Simple-API-REST/product-service/products/2	Eliminar un producto con id=2
PUT	http://localhost:8080/Simple-API-REST/product-service/products/	Editar un producto

Creando la aplicación cliente: simca-cliente

Ahora se va a crear una sencilla aplicación desktop en java que consuman los servicios web creados en la sección anterior.

El primer paso es crear el proyecto, clic en el menú File / New Project / Java with Maven/Java Application y seguir los pasos del asistente. El nombre del proyecto será **Cliente-Rest** (ver Figura 9):

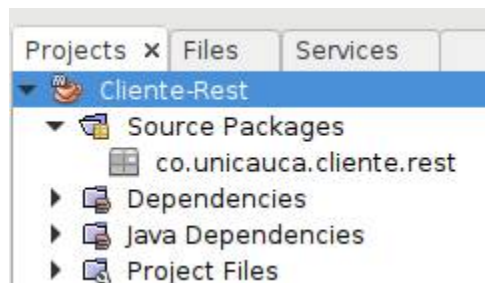


Figura 9. Aplicación cliente que consume los servicios web

El código del cliente se basa en la API del cliente de **Jersey**. Para ello, clic derecho en el paquete cliente / New / Web Services / RESTful java client (ver Figura 10). Luego clic en Next.

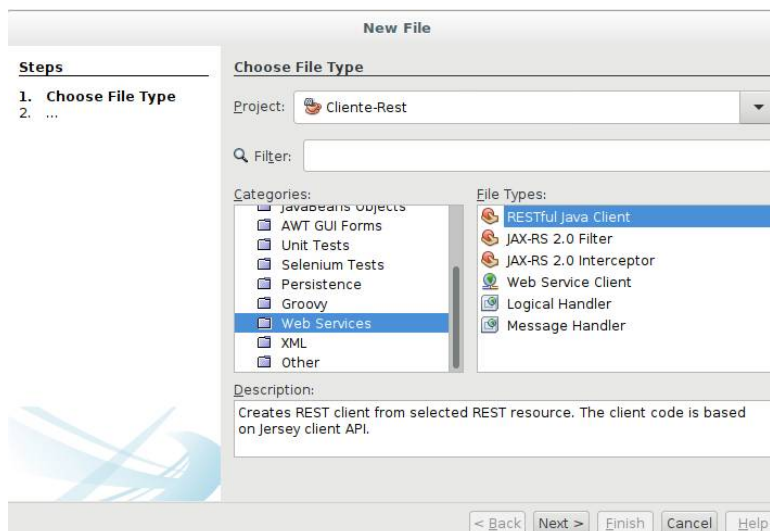


Figura 10. Creando el cliente RESTful.

En la siguiente ventana del asistente, se configura la clase `NewJerseyClient`. Se hace clic en el botón browser y se elige el servicio web al que se va a conectar. Se elige el proyecto Simple-API-REST y el servicio ProductController [products] (ver Figura 11).

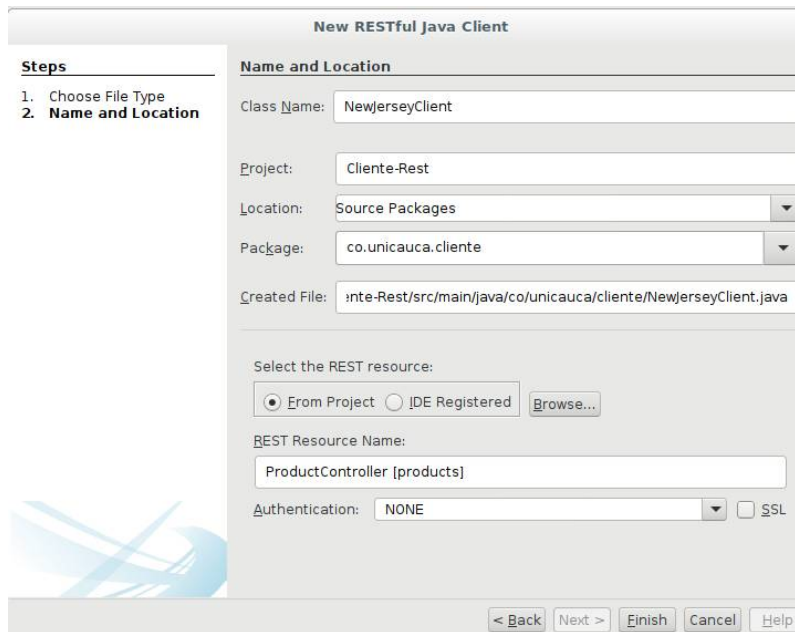


Figura 11. Configurando la clase NewJerseyClient.

Entonces, el asistente genera el código fuente de la clase `NewJerseyClient` para conectarse al servicio web de productos:

```
package co.unicauca.cliente;

import javax.ws.rs.ClientErrorException;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.WebTarget;

/**
 * Jersey REST client generated for REST resource:ProductController
 * [products]<br>
 * USAGE:
 * <pre>
 *     NewJerseyClient client = new NewJerseyClient();
 *     Object response = client.XXX(...);
 *     // do whatever with response
 *     client.close();
 * </pre>
 *
 * @author libardo
```

```
*/
public class NewJerseyClient {

    private WebTarget webTarget;
    private Client client;
    private static final String BASE_URI = "http://localhost:8080/Simple-API-REST/product-
service";

    public NewJerseyClient() {
        client = javax.ws.rs.client.ClientBuilder.newClient();
        webTarget = client.target(BASE_URI).path("products");
    }

    public <T> T findById(Class<T> responseType, String id) throws ClientErrorException {
        WebTarget resource = webTarget;
        resource = resource.path(java.text.MessageFormat.format("{0}", new Object[]{id}));
        return
resource.request(javax.ws.rs.core.MediaType.APPLICATION_JSON).get(responseType);
    }

    public String create_XML(Object requestEntity) throws ClientErrorException {
        return
webTarget.request(javax.ws.rs.core.MediaType.APPLICATION_XML).post(javax.ws.rs.client.Entity
.entity(requestEntity, javax.ws.rs.core.MediaType.APPLICATION_XML), String.class);
    }

    public String create_JSON(Object requestEntity) throws ClientErrorException {
        return
webTarget.request(javax.ws.rs.core.MediaType.APPLICATION_JSON).post(javax.ws.rs.client.Entit
y.entity(requestEntity, javax.ws.rs.core.MediaType.APPLICATION_JSON), String.class);
    }

    public String update_XML(Object requestEntity) throws ClientErrorException {
        return
webTarget.request(javax.ws.rs.core.MediaType.APPLICATION_XML).put(javax.ws.rs.client.Entity
.entity(requestEntity, javax.ws.rs.core.MediaType.APPLICATION_XML), String.class);
    }

    public String update_JSON(Object requestEntity) throws ClientErrorException {
        return
webTarget.request(javax.ws.rs.core.MediaType.APPLICATION_JSON).put(javax.ws.rs.client.Entity
.entity(requestEntity, javax.ws.rs.core.MediaType.APPLICATION_JSON), String.class);
    }

    public <T> T findAll(Class<T> responseType) throws ClientErrorException {
        WebTarget resource = webTarget;
        return
resource.request(javax.ws.rs.core.MediaType.APPLICATION_JSON).get(responseType);
    }

    public String remove(String id) throws ClientErrorException {
        return webTarget.path(java.text.MessageFormat.format("{0}", new Object[]
{id})).request().delete(String.class);
    }

    public void close() {
        client.close();
    }
}
```

```
}
```

Ahora agregar las siguientes dependencias al archivo `pom.xml`:

```
<dependency>
  <groupId>javax.activation</groupId>
  <artifactId>activation</artifactId>
  <version>1.1.1</version>
</dependency>
<dependency>
  <groupId>javax.xml.bind</groupId>
  <artifactId>jaxb-api</artifactId>
  <version>2.3.0</version>
</dependency>
```

Finalmente, se crea una clase `ClienteMain` que utilice los métodos de la clase `NewJerseyClient`.

```
package co.unicauca.cliente;

/**
 *
 * @author Libardo, Julio
 */
public class ClienteMain {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // CREANDO UN ESTUDIANTE
        NewJerseyClient jersey = new NewJerseyClient();
        String rta = jersey.create_JSON(new Product(5, "Nevera Lg", 4000000d));
        System.out.println("Rta " + rta);

        // BUSCANDO UN PRODUCTO
        Product product = jersey.findById(Product.class, "1");

        System.out.println(product.getId());
        System.out.println(product.getName());
        System.out.println(product.getPrice());

        // PROBAR LOS DEMAS SERVICIOS
    }
}
```

Esta aplicación cliente también necesita la clase `Product`. Se puede copiar/pegar la misma del proyecto `Simple-API-REST`.

Al correr el cliente, la salida debe ser:

```
--- exec-maven-plugin:1.5.0:exec (default-cli) @ Cliente-Rest ---  
Rta {"ok":"true", "mensaje":"Producto creado","errores":""}  
1  
Tv samsung  
200000.0  
-----  
BUILD SUCCESS  
-----
```

FIN!