



Nombre:

Rayner Steven Palta Tenecela

Materia:

Gerencia Informática

Ciclo:

10° Ciclo

Carrera:

Ingenierías de Sistemas

Docente:

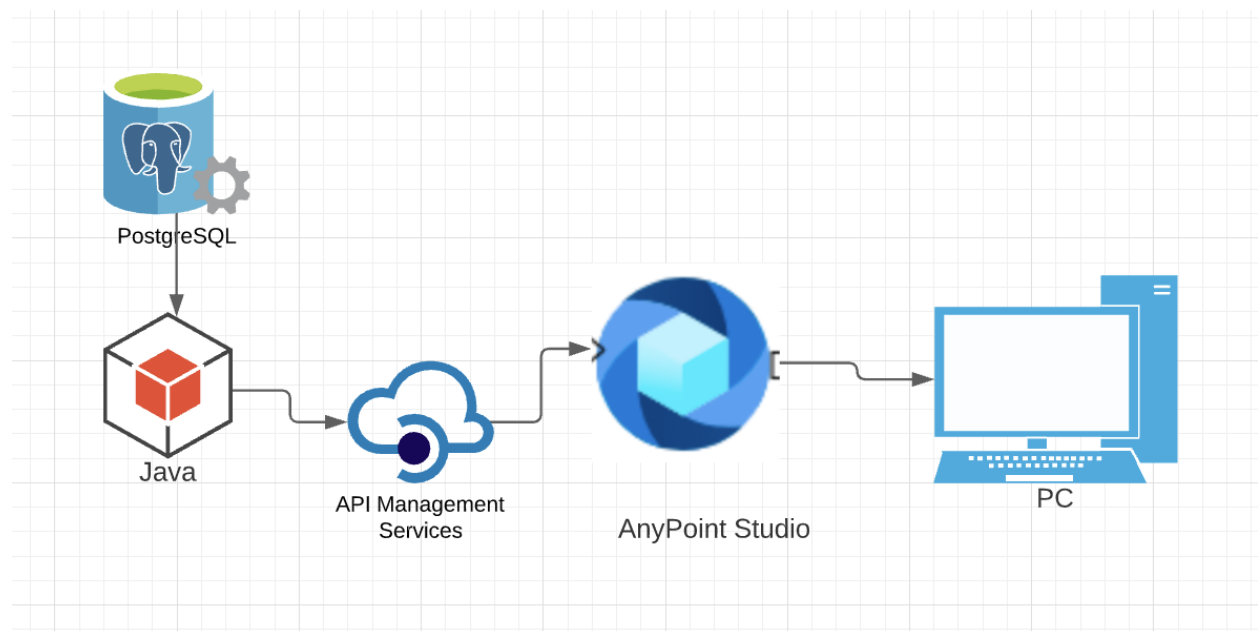
Ing. Christian Timbi

Informe

El proyecto está basado en un sistema de recargas bancario desarrollado en Eclipse. El programa permite crear clientes, cuenta; registrar transacciones y transferencias; asimismo permite listar cada uno de estos. Para el proceso de la transferencia se ha simplificado el proceso. Los datos que se ingresan son: cuenta de origen, monto a transferir, concepto de la transferencia, tarifa por transferencia y la cuenta destino.

Arquitectura:

La arquitectura propuesta consta de una base de datos PostgreSQL, la cual se conecta con el sistema bancario realizada en Eclipse, dentro del cual están los servicios web SOAP y Rest, los cuales son consumidos mediante el ESB en nuestro caso usamos la aplicación llamada AnyPoint Studio. Además, para el consumo de los servicios creados en el ESB se hace uso de la herramienta Postman.



Eclipse:

Dentro del IDE Eclipse empezamos creando un nuevo proyecto de tipo Maven. Una vez creado el proyecto se debe declarar los paquetes correspondientes y sus clases.



Se sigue la arquitectura MVC para este proyecto integrando otras características como es el paquete servicios.

Asimismo se procede a definir los atributos respectivos para cada clase y generar los get y set de cada uno.

```
@Entity
public class Cliente implements Serializable {
    private static final long serialVersionUID = 1L;
    // Atributos de la entidad
    @Id
    private String cedula;
    private String nombre;
    private String apellido;
    private String correo;
    private String celular;
    @ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    @JoinColumn(name = "idTelefono")
    private Cuenta cuentaCliente;

    /**
     * Constructor de la clase
     */
    public Cliente() {
    }

    /**
     * Metodo que permite obtener el atributo cedula
     * @return El atributo cedula de esta clase
     */
    public String getCedula() {
        return cedula;
    }

    /**
     * Metodo que permite asignarle un valor al atributo cedula
     * @param cedula parametro para poder obtener
     */
    public void setCedula(String cedula) {
        this.cedula = cedula;
    }
}
```

Además, creamos los objetos de acceso a datos o por sus siglas DAO. En estas clases se declaran los métodos de insertar, buscar, eliminar, entre otros. Estos métodos serán accedidos por los objetos de negocio.

```
@Stateless
public class ClienteDao {

    private Connection con;
    // Atributo de la clase
    @PersistenceContext(name = "SistemasGerencialesESBPersistenceUnit")
    private EntityManager em;

    /**
     * Constructor que permite inicializar la clase.
     */
    public ClienteDao() {
        // TODO Auto-generated constructor stub
    }

    /**
     * Metodo que permite insertar un cliente dentro de la base de datos por medio
     * de JPA
     *
     * @param cliente Cliente que se va a usar los servicios
     * @return true devuelve verdadero en caso de que el cliente exista.
     */
    public boolean insert(Cliente cliente) throws SQLException {
        em.persist(cliente);
        return true;
    }

    /**
     * Metodo que permite buscar un cliente a traves de la base por medio de JPA
     *
     * @param cedula variable que se ingresa para la consulta
     * @return empleado Cliente cuya cedula es igual a la que se ingresa
     */
    public Cliente read(String cedula) {
        Cliente empleado = new Cliente();
        empleado = em.find(Cliente.class, cedula);
        System.out.println(empleado);
        return empleado;
    }
}
```

Se crean los objetos de negocio para cada clase creada. En esta clase se llama a los DAO y se crean métodos para cada opción que se necesite.

```
@Stateless
public class ClienteON {
    private String cedulaCliente;
    @Inject
    private ClienteDao daoCliente;

    public ClienteON() throws Exception {
        super();
    }

    /**
     * Metodo que permite registrar el ingreso de un cliente al sistema
     *
     * @param empleado que se obtiene por medio de los Beans
     *
     * @throws Exception captura algún error que pueda ocurrir al momento de
     * utilizar el metodo
     */
    public void registrarCliente(Cliente empleado) throws Exception {
        try {
            // System.out.println("BEAN"+empleado);
            System.out.println("entra al on ");
            daoCliente.insert(empleado);
        } catch (SQLException e) {
            e.printStackTrace();
            System.out.println("Erro al insertar");
        }
    }

    /**
     * Metodo que permite buscar a un cliente por medio de la cedula
     *
     * @param cedula que se obtiene por medio de las clase de texto en la interfaz
     *
     * @throws Exception captura algún error que pueda ocurrir al momento de
     * utilizar el metodo
     */
    public Cliente buscarCliente(String cedula) throws Exception {
        Cliente cliente = daoCliente.read(cedula);
        cedulaCliente = cliente.getCedula();
        System.out.println("BUSQUEDAD CLIENTE CORRECTA " + cedulaCliente);
        return cliente;
    }
}
```

Con el modelo, DAO y Objeto de negocio creado procedemos a crear los servicios web. Para esto primero se debe crear una nueva clase en la cual se va a definir el path o ruta para los servicios.

```
@javax.ws.rs.ApplicationPath("/ws")
public class ServiciosPath extends Application {

}
```

Con el path definido se procede a crear el servicio Rest y Soap. Cada servicio es diferente por lo que el modo de declaración cambia. Aquí se hace uso de los objetos de negocio y los modelos.

```
@WebService
public class ServicioSOAP {
    @Inject
    private ClienteON onCliente;
    @Inject
    private CuentaON onCuenta;
    @Inject
    private TransaccionON onTransaccion;
    @Inject
    private TransferenciaON onTransferencia;

    private Transaccion tran;
    private Transferencia transferencia;
    private Cuenta c;
    private Cuenta c2;
    private double saldoNuevo;
    private double saldoCuenta;
    private double saldoNuevo2;
    private Date fechaA;

    @WebMethod
    public String crearCuenta(double saldo, String tipoCuenta, String entidadBancaria, String cedula) throws Exception {
        Cuenta cc = new Cuenta();
        Date fechaApertura = new Date();
        cc.setNumeroCuenta(onCuenta.generarNumeroDeCuenta());
        cc.setFechaApertura(fechaApertura);
        cc.setSaldo(saldo);
        cc.setTipoCuenta(tipoCuenta);
        cc.setEntidadBancaria(entidadBancaria);
        cc.setCuenta_fk(cedula);
        onCuenta.registrarCuenta(cc);
        return "Cuenta creada con éxito";
    }

    @WebMethod
    public String crearCliente(String cedula, String nombre, String apellido, String correo, String celular) throws Exception {
        Cliente c = new Cliente();
        c.setNombre(nombre);
        c.setApellido(apellido);
        c.setCedula(cedula);
        c.setCorreo(correo);
        c.setCelular(celular);
        onCliente.registrarCliente(c);
        return "Cliente registrado con éxito";
    }
}

@Inject
private ClienteON onCliente;
@Inject
private CuentaON onCuenta;
@Inject
private TransaccionON onTransaccion;

private Transaccion tran;
private Cliente cli;
private Cuenta c;

private double saldoNuevo;
private Date fechaA;

public ServicioREST() {
    // TODO Auto-generated constructor stub
}

@GET
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)

@Path("/crearCliente")
public String crearCliente(@QueryParam("cedula") String cedula, @QueryParam("nombre") String nombre,
    @QueryParam("apellido") String apellido, @QueryParam("correo") String correo,
    @QueryParam("celular") String celular) throws Exception {
    Cliente c = new Cliente();
    c.setCedula(cedula);
    c.setNombre(nombre);
    c.setApellido(apellido);
    c.setCorreo(correo);
    c.setCelular(celular);
    onCliente.registrarCliente(c);
    return "Cliente registrado con éxito";
}
}
```

Como se puede observar en las imágenes superiores la forma para crear un cliente cambia entre cada servicio, en uno se hace uso de mas anotaciones y en el otro no. Teniendo esto en cuenta se crea el método para posibilitar las transferencias bancarias.

```
@POST
@Path("/transferencia/{cuentaOrigen}/{monto}/{conceptoTransferencia}/{tarifa}/{cuentaDestino}")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public String registrarTransferencia(@PathParam("cuentaOrigen") String cuentaOrigen, @PathParam("monto") double monto, @PathParam("conceptoTransferencia") String conceptoTransferencia, @PathParam("tarifa") double tarifa, @PathParam("cuentaDestino") String cuentaDestino) {
    c = new Cuenta();
    transferencia = new Transferencia();

    try {
        Cuenta c3 = new Cuenta();
        c3 = onCuenta.obtenerCuentaPorNumero(cuentaOrigen);

        System.out.println("cuenta origente"+ c3.getNumeroCuenta()+" "+c3.getSaldo()+"monto que llega "+" "+monto);
        transferencia.setMonto(monto);
        if (transferencia.getMonto() > c3.getSaldo()) {
            System.out.println("El valor a transferir es mayor al que se tiene en la cuenta "+" "+c3.getSaldo());
        } else {
            c = onCuenta.obtenerCuentaPorNumero(cuentaDestino);
            transferencia.setClienteLocal(c);
            transferencia.setConceptoTransferencia(conceptoTransferencia);
            transferencia.setTarifaTransaccion(tarifa);
            transferencia.setCuenta(c);
            double saldoCuenta = c3.getSaldo() - monto;
            saldoNuevo2 = c.getSaldo() + monto;
            System.out.println("saldo CUENTA RESTANTO"+ saldoCuenta + " "+" saldo a la cuenta destino"+ " "+saldoNuevo2);
            onCuenta.actaulizarCuentaCliente(cuentaOrigen, saldoCuenta);
            onCuenta.actaulizarCuentaCliente(cuentaDestino, saldoNuevo2);

            onTransferencia.agregarCuentaTransferecia(transferencia);
        }
    } catch (Exception e) {
        System.err.println(e.getMessage());
        e.printStackTrace();
    }
    return "Transferencia exitosa";
}

@WebMethod
public String registrarTransferencia(String cuentaOrigen, double monto, String conceptoTransferencia, double tarifa, String cuentaDestino) {
    c2 = new Cuenta();
    transferencia = new Transferencia();

    try {
        Cuenta c3 = new Cuenta();
        c3 = onCuenta.obtenerCuentaPorNumero(cuentaOrigen);






        System.out.println("cuenta origente"+ c3.getNumeroCuenta()+" "+c3.getSaldo()+"monto que llega "+" "+monto);
        transferencia.setMonto(monto);
        if (transferencia.getMonto() > c3.getSaldo()) {
            System.out.println("El valor a transferir es mayor al que se tiene en la cuenta "+" "+c3.getSaldo());
        } else {
            c2 = onCuenta.obtenerCuentaPorNumero(cuentaDestino);
            transferencia.setClienteLocal(c3);
            transferencia.setConceptoTransferencia(conceptoTransferencia);
            transferencia.setTarifaTransaccion(tarifa);
            transferencia.setCuenta(c2);
            saldoCuenta = c3.getSaldo() - monto;
            saldoNuevo2 = c2.getSaldo() + monto;
            System.out.println("saldo CUENTA RESTANTO"+ saldoCuenta + " "+" saldo a la cuenta destino"+ " "+saldoNuevo2);
            onCuenta.actaulizarCuentaCliente(cuentaOrigen, saldoCuenta);
            onCuenta.actaulizarCuentaCliente(cuentaDestino, saldoNuevo2);

            onTransferencia.agregarCuentaTransferecia(transferencia);
        }
    } catch (Exception e) {
        System.err.println(e.getMessage());
        e.printStackTrace();
    }
    return "Transferencia exitosa";
}
```

Una vez creado los servicios mandamos a ejecutar en el servidor Wildfly para poder continuar con el proceso de desarrollo.

PostgreSQL

Dentro de PostgreSQL debemos crear una nueva base que coincida con los datos ingresados en el archivos xml dentro de Eclipse. Esta base servirá para almacenar todos los datos que se ingresen a través de los servicios consumidos por el ESB.

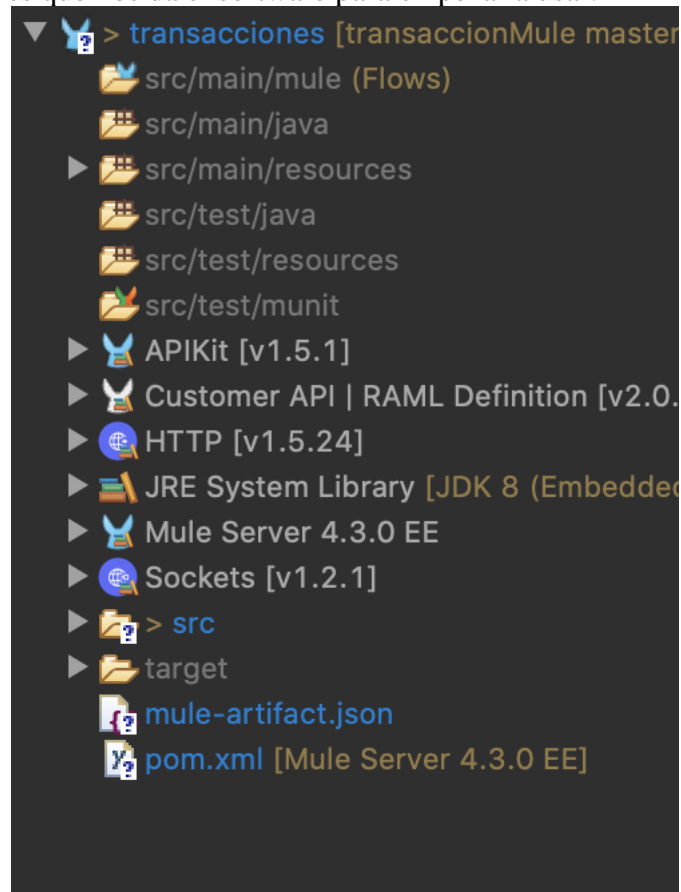
- ▼  Tablas (4)
 - >  cliente
 - >  cuenta
 - >  transaccion
 - >  transferencia

MuleSoft: AnyPoint Studio

El ESB elegido para este proyecto es MuleSoft, hacemos uso de su aplicación de escritorio llamada AnyPointStudio. Aquí creamos un nuevo proyecto de tipo Mule en el cual haremos el consumo de los servicios creados en Eclipse.



Una vez creado el proyecto se crearán varios archivos dentro del proyecto. En la parte derecha tenemos varias opciones que nos da el software para empezar a usar.

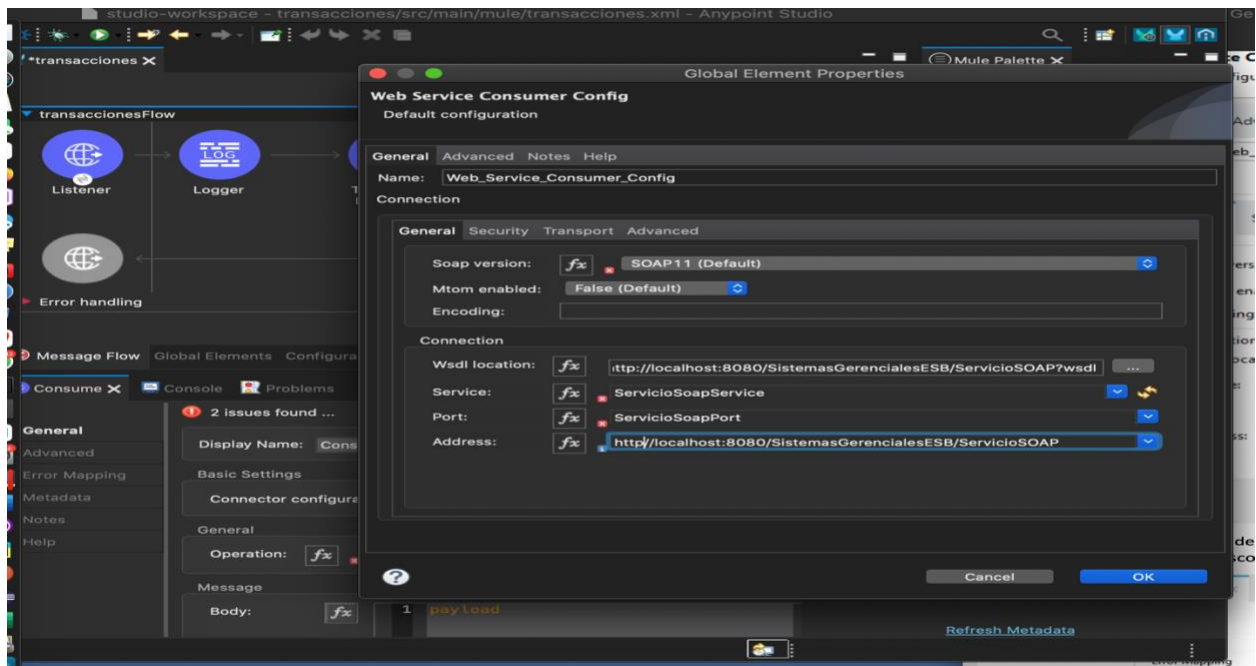


El primer objeto que usaremos es el Listener. Este objeto nos permite declara el puerto y la ip además del path a ser usado para el consumo.

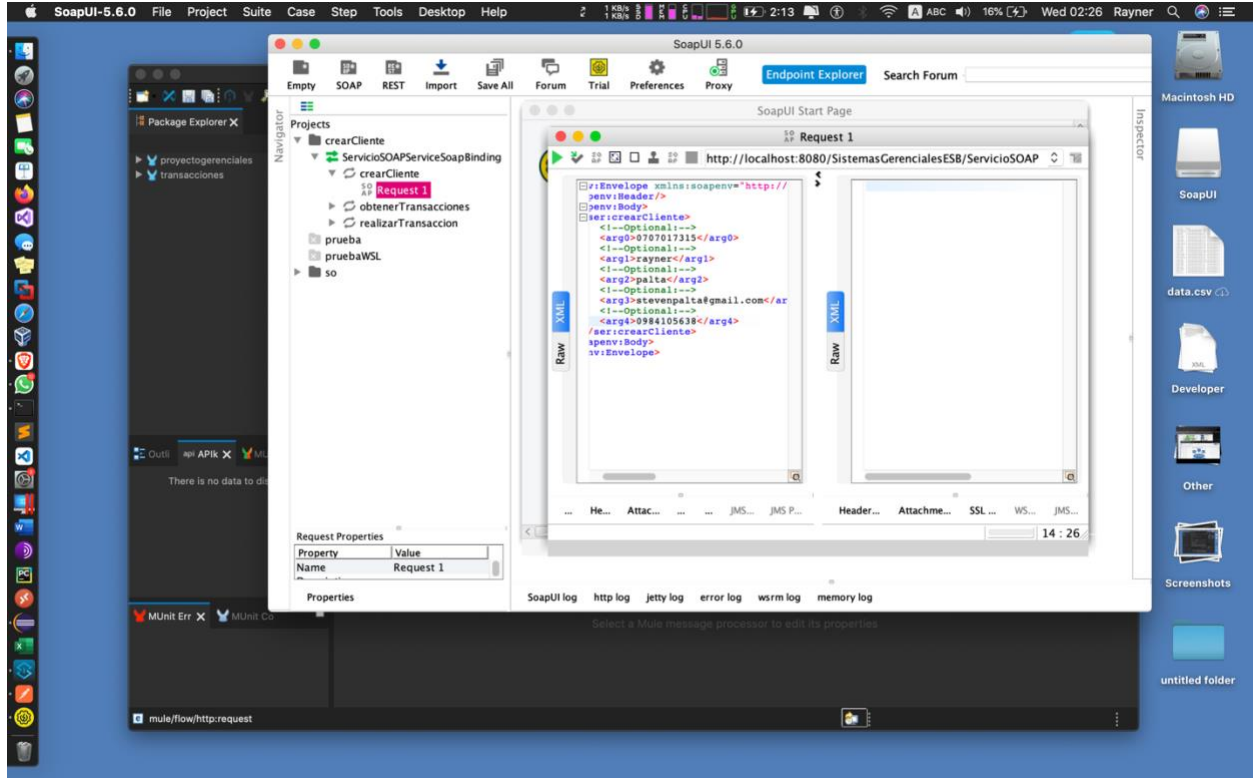
Agregamos el componente Transform Message, permite realizar transformaciones sobre los datos de entrada que recibe. Puede escribir explícitamente una permite realizar transformación en lenguaje DataWeave, o puede usar la interfaz de usuario para construirla arrastrando y soltando elementos.



Finalmente arrastramos el componente llamado web service consume el cual consume un servicio web SOAP de una aplicación Mule para adquirir datos de una fuente externa.



Consumimos el servicio



URL: <https://anypoint.mulesoft.com/exchange/fe9a2009-8fb9-4a78-ba9f-82cc8753099a/transacciones/minor/1.0/draft/edit/home/>