UNIVERSIDAD TÉCNICA ESTATAL DE QUEVEDO

Facultad de ciencias de la ingeniería

ESTUDIANTE:

CHICA VALFRE VALESKA SOFIA

LETURNE PLUAS JHON BYRON

CARRERA:

Ingeniería de software

CURSO:

7to Software “A”

ASIGNATURA:

Aplicaciones distribuidas

DOCENTE:

Ing. Guerrero Ulloa Gleiston Cicerón, MSc

TEMA:

Generalidades de los sistemas distribuidos

Contenido

Ilustraciones

Tablas

1. Introducción

En el contexto de la era digital, el diseño de la arquitectura de sistemas y aplicaciones informáticas se ha vuelto una pieza fundamental e indispensable para poder desarrollar e implementar soluciones tecnológicas verdaderamente eficientes, escalables y sostenibles en el tiempo [1]. Dentro de este campo, existen tres pilares o conceptos clave que sostienen y dan forma al complejo edificio de la tecnología de la información en la actualidad: las arquitecturas distribuidas, las arquitecturas de aplicaciones y la arquitectura web [2].

Las arquitecturas distribuidas implican un paradigma donde múltiples computadoras o nodos de procesamiento, interconectados mediante una red, trabajan de manera coordinada y colaborativa para lograr un objetivo en común. Este enfoque permite compartir recursos informáticos, almacenamiento y responsabilidades entre los distintos nodos. Las principales ventajas de este modelo son: mejor aprovechamiento de la capacidad de procesamiento total del sistema, mayor eficiencia y disponibilidad de las aplicaciones, mejor escalabilidad ante incrementos de la demanda y una forma más efectiva de gestionar grandes volúmenes de datos distribuidos [3].

Por otro lado, las arquitecturas de aplicaciones definen la estructura de componentes de un sistema de software, la interacción entre ellos y el flujo de información. Tener una arquitectura de aplicación bien diseñada y optimizada es indispensable para construir soluciones de software de alta calidad, fáciles de mantener, extender y que aprovechen de forma eficiente los recursos informáticos subyacentes [4].

Finalmente, en las últimas décadas la arquitectura web se ha posicionado como la columna vertebral técnica del ciberespacio. Implica importantes consideraciones de diseño para construir aplicaciones y servicios web globalmente accesibles, capaces de gestionar una gran concurrencia de usuarios, altos niveles de disponibilidad y tiempos de respuesta aceptables. Aspectos de seguridad, privacidad de datos y experiencia de usuario son también críticos [5].

En este trabajo nos enfocaremos en analizar tres arquitecturas, los principios y mejores prácticas que guían su diseño e implementación, así como los desafíos y oportunidades que presentan en un panorama tecnológico en constante cambio.

2. Arquitecturas distribuidas

2.1. Definición y características

Las arquitecturas distribuidas representan un paradigma en el diseño de sistemas y aplicaciones informáticas en el que los componentes se encuentran dispersos en diferentes ubicaciones geográficas pero trabajan de manera conjunta y coordinada para cumplir un objetivo común. Gracias a la evolución de las tecnologías de comunicación y la necesidad de sistemas escalables, esta arquitectura ha ganado popularidad, permitiendo la distribución de cargas de trabajo y recursos a través de múltiples nodos o servidores [6].

Además, las arquitecturas distribuidas son escalables, ya que permiten añadir o eliminar recursos conforme a la demanda, sin necesidad de rediseñar el sistema completo. Esto posibilita a las empresas adaptarse rápidamente a cambios en el mercado o en las necesidades de los usuarios [7].

Otra característica relevante es la concurrencia, que permite que múltiples procesos se ejecuten simultáneamente en diferentes nodos. Esto mejora el rendimiento y la eficiencia del sistema, al dividir las tareas y procesarlas en paralelo. Sin embargo, la concurrencia también introduce complejidad en la gestión de estados y la sincronización entre procesos [8].

La heterogeneidad es también una característica común en las arquitecturas distribuidas, donde diferentes nodos pueden tener distintos sistemas operativos, hardware y software, lo que requiere un diseño cuidadoso para asegurar la interoperabilidad y la correcta comunicación entre los componentes [9].

En cuanto a la seguridad, las arquitecturas distribuidas enfrentan desafíos únicos debido a su naturaleza dispersa. Es esencial implementar mecanismos robustos de autenticación, autorización y cifrado para proteger la información y los recursos de accesos no autorizados o ataques maliciosos[7], [9].

2.2. Ventajas y desventajas

Las arquitecturas distribuidas, mientras ofrecen numerosos beneficios, también vienen con desafíos inherentes. A continuación, se detallan las ventajas y desventajas de estas arquitecturas.

Ventajas

- **Escalabilidad:** Permiten escalar recursos horizontalmente, añadiendo más nodos según sea necesario, lo que facilita el manejo de cargas de trabajo crecientes [10].
- **Flexibilidad:** Son capaces de integrar diferentes tipos de hardware y software, proporcionando una mayor flexibilidad operativa [11].
- **Tolerancia a Fallos:** La distribución de recursos y tareas entre múltiples nodos mejora la resistencia del sistema frente a fallos individuales [12].
- **Eficiencia de Recursos:** Optimizan el uso de recursos al distribuir cargas de trabajo a través de múltiples nodos [13].

Desventajas

1. **Complejidad de Diseño y Gestión:** El diseño, implementación y mantenimiento de sistemas distribuidos son más complejos que los de sistemas centralizados [14].
2. **Problemas de Seguridad:** La distribución de datos y recursos a través de una red puede exponer el sistema a riesgos de seguridad adicionales [15].
3. **Consistencia de Datos:** Mantener la consistencia de datos en múltiples nodos es un desafío, especialmente en sistemas que requieren actualizaciones frecuentes [16].
4. **Costos de Infraestructura y Operación:** La necesidad de múltiples servidores y conexiones de red puede incrementar los costos [17].

3. Arquitecturas de aplicaciones

3.1. Monolíticas vs. Microservicios

Las arquitecturas monolíticas son aquellas donde todas las funciones de una aplicación están unidas en un solo código base. En estas, los componentes de la aplicación, como la interfaz de usuario, el servidor backend, y la base de datos, están entrelazados y desplegados como una unidad única [18].

Por otro lado, las arquitecturas basadas en microservicios descomponen una aplicación en una colección de servicios más pequeños y autónomos. Cada microservicio es independiente y realiza una función específica, comunicándose con otros microservicios a través de APIs bien definidas [19].

Un aspecto crucial de las arquitecturas monolíticas es que cualquier cambio o actualización requiere redeployar toda la aplicación es decir desplegar la aplicación para que pueda ser accedida desde internet, lo cual puede ser menos eficiente y más propenso a errores. También,

en aplicaciones de gran escala, la complejidad del código puede incrementar significativamente, haciendo más difícil la mantenibilidad y la escalabilidad [20].

En contraste, en los microservicios, los cambios se pueden realizar en un servicio específico sin afectar a otros, lo que facilita la implementación de nuevas características y la resolución de problemas. Además, esta arquitectura es altamente escalable y eficiente para aplicaciones grandes y complejas, ya que permite distribuir y balancear la carga entre diferentes servicios [21].

Sin embargo, los microservicios traen desafíos adicionales, como la gestión de la comunicación entre servicios, la complejidad en el despliegue y la necesidad de un sistema de orquestación robusto. Además, pueden requerir una inversión inicial mayor en términos de diseño y configuración de la infraestructura [22].

3.2. Componentes y módulos

En la arquitectura de aplicaciones, los componentes son fundamentales para la estructura y organización del software. Estos elementos incluyen bases de datos, servidores de aplicaciones y clientes web. Cada componente cumple una función específica, operando de manera independiente dentro del sistema global [23].

La interacción entre los componentes se realiza a través de interfaces bien definidas, como APIs o servicios web. Esta comunicación clara y estructurada permite que distintas partes del sistema colaboren sin conflictos. La modularidad de los componentes facilita realizar cambios o actualizaciones en una parte del sistema sin alterar otras áreas [24].

Los módulos, ubicados dentro de los componentes, agrupan funcionalidades relacionadas. Estos pueden consistir en clases, funciones y servicios que colaboran para ejecutar tareas específicas. La organización en módulos ayuda a mejorar la estructura del código, haciéndolo más legible y manejable [22].

Esta modularidad mejora la reutilización del código en diferentes partes de la aplicación. Los módulos pueden ser desarrollados, probados y desplegados independientemente, lo que agiliza el desarrollo y aumenta la eficiencia operativa [25].

4. Arquitectura web

4.1. Ejemplo

Dentro de los diferentes tipos de arquitecturas, hemos creado un ejemplo de registro de personas, proporcionando tres enfoques: Cliente-Servidor, Arquitectura en tres capas y arquitectura RESTful y SOAP.

En las dos primeras arquitecturas, presentamos cómo funcionaría la aplicación si se desarrollara en Cliente-Servidor y Arquitectura en Tres Capas. Luego, llevamos a cabo la implementación en RESTful y SOAP para que se pueda observar la conexión y manejo de datos utilizando estas arquitecturas.

Para la realización de este ejemplo, se utilizaron las siguientes tecnologías:

Tabla 1 Herramientas utilizadas

realizado por LETURNE PLUAS JHON BYRON & CHICA VALFRE VALESKA SOFIA (Grupo E)

Herramientas utilizadas					
Frontend	**Backend**	**Servidores**	**Base de datos**	**Pruebas**	**Versiones de Java**
HTML	Java	Apache Tomcat	Postgresql	SOAP UI	JDK 8
CSS	SpringBoot	Payara5		Postman	JDK 11
Bootstrap	JavaWeb				JDK 19
JavaScript	Lombok				
Jquery					
AngularJS					
x2js (Parcing XML a JSON)					

4.2. Cliente-servidor

La arquitectura cliente-servidor en la web es un modelo donde la interacción se divide entre dos entidades principales. Por un lado, está el cliente, generalmente un navegador web, que realiza solicitudes al servidor. Por otro lado, está el servidor, que alberga recursos y servicios web, procesando las solicitudes del cliente [26].

El cliente inicia las solicitudes, como cargar páginas web o enviar datos de formularios. Estas solicitudes son recibidas y procesadas por el servidor, que puede involucrar tareas como recuperación de datos o ejecución de lógica de negocio [27].

Una vez procesada la solicitud, el servidor envía una respuesta al cliente. Esta respuesta puede variar desde una página web completa hasta datos en formatos específicos como JSON, dependiendo de la naturaleza de la solicitud [28].

Esta arquitectura se caracteriza por su separación clara de responsabilidades. El cliente se enfoca en la presentación y la interacción con el usuario, mientras que el servidor maneja el procesamiento de datos y la lógica del negocio. Esta separación facilita el desarrollo y mantenimiento de aplicaciones web [29], [30].

No obstante, existen desafíos inherentes a este modelo, particularmente en términos de escalabilidad y seguridad. Gestionar un alto volumen de solicitudes simultáneas y asegurar las transacciones entre cliente y servidor son aspectos cruciales en el diseño de sistemas basados en la arquitectura cliente-servidor [30].

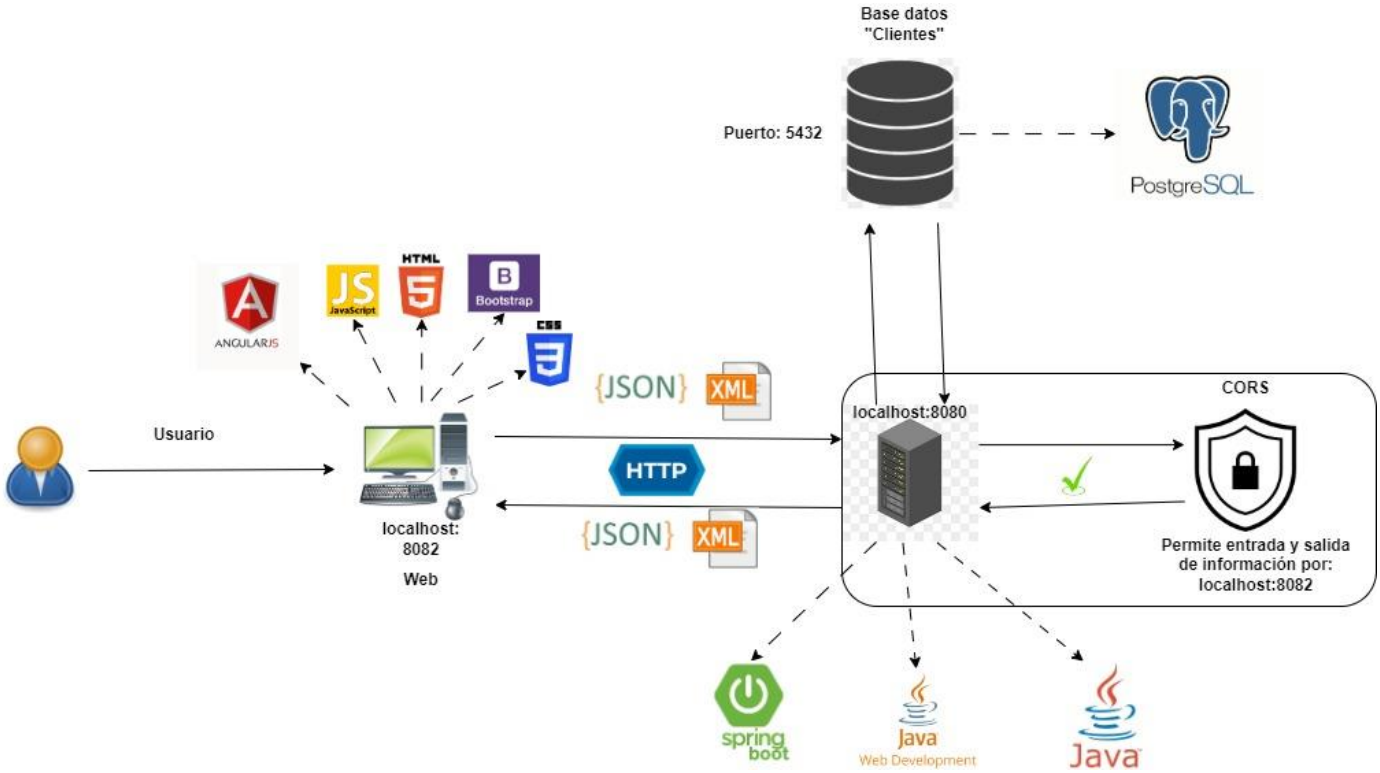

Ilustración 1 Esquema de arquitectura cliente servidor

realizado por LETURNE PLUAS JHON BYRON & CHICA VALFRE VALESKA SOFIA (Grupo E)

4.3. Arquitectura de tres capas

La arquitectura de tres capas es un modelo de diseño de aplicaciones informáticas que se estructura en tres niveles lógicos y funcionales distintos. Esta división tiene como objetivo separar claramente las responsabilidades dentro de la aplicación, mejorando su mantenibilidad, escalabilidad y flexibilidad [31].

4.3.1. Capa de presentación

El primer nivel es la Capa de Presentación, también conocida como capa de interfaz de usuario. Esta es la capa con la que los usuarios interactúan directamente. Se encarga de mostrar la información al usuario y de recoger sus entradas, asegurando que la experiencia del usuario sea intuitiva y eficiente[32].

4.3.2. Capa de lógica de negocio

Esta capa actúa como el núcleo funcional de la aplicación, procesando las solicitudes de los usuarios, ejecutando las reglas de negocio, y tomando decisiones. Esta capa es crucial ya que determina cómo se operan y transforman los datos dentro de la aplicación [33].

4.3.3. Capa de datos

Esta capa se ocupa de todo lo relacionado con el almacenamiento y la gestión de datos, como las bases de datos y los servidores de archivos. Su función principal es la recuperación, almacenamiento y actualización de datos en respuesta a las solicitudes de la capa de lógica de negocio [34].

La independencia entre estas capas es una característica clave de la arquitectura de tres capas. Permite que cada capa se desarrolle y mantenga de manera aislada, lo que facilita la actualización y la mejora de componentes individuales sin afectar el resto del sistema. Este enfoque modular también contribuye a una mejor organización del código y a una gestión más sencilla de la complejidad del software [35].

Ejemplo

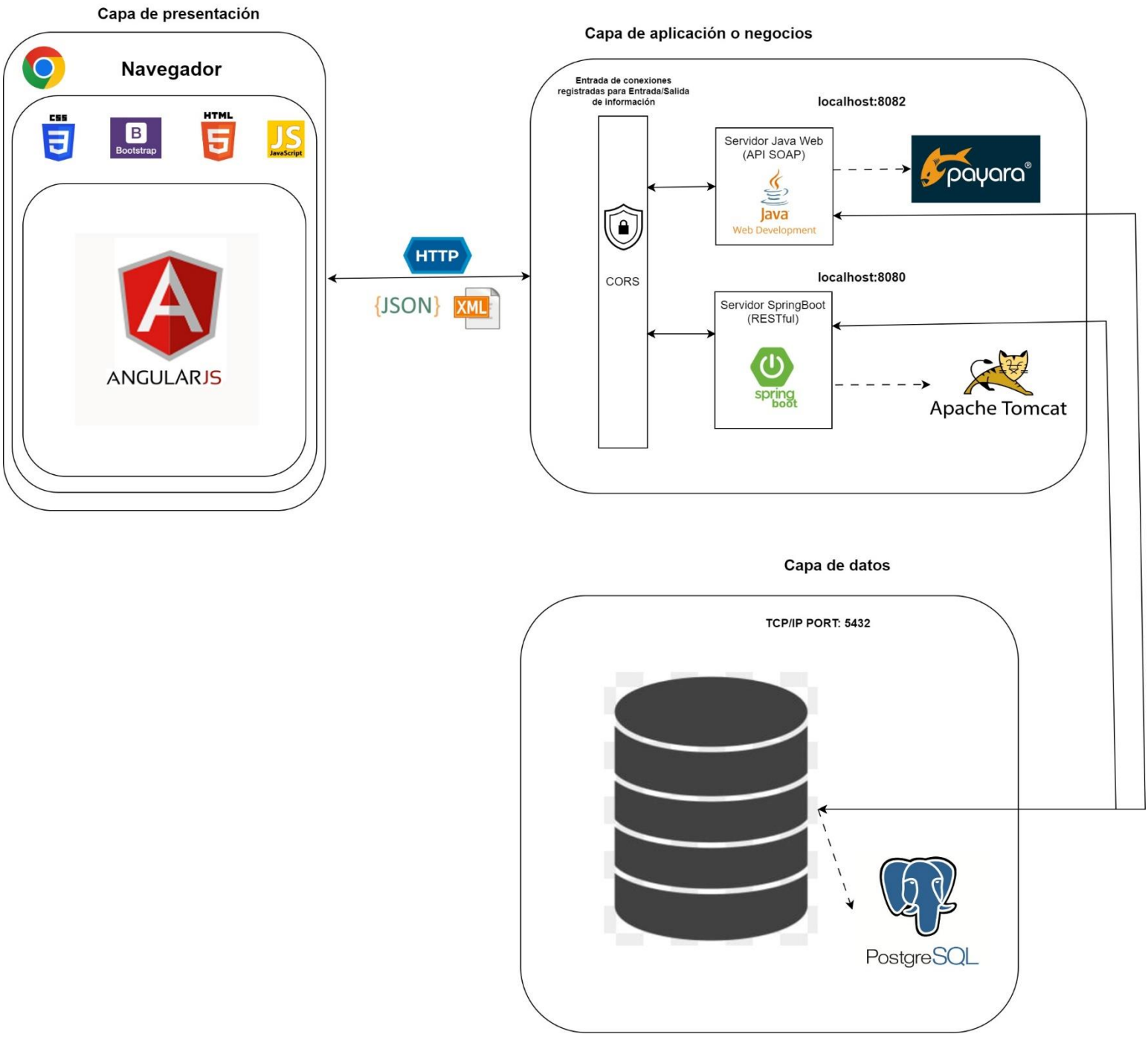

Ilustración 2 Esquema de arquitectura en tres capas

realizado por LETURNE PLUAS JHON BYRON & CHICA VALFRE VALESKA SOFIA (Grupo E)

4.4. Arquitectura RESTful y SOAP

4.4.1. RESTful

RESTful utiliza URIs para identificar recursos, facilitando su acceso y manipulación. Esta arquitectura se apoya en métodos HTTP estándar, como GET y POST, lo que simplifica su implementación en aplicaciones web [36]. Una característica clave de RESTful es su enfoque sin estado; cada solicitud es independiente. Esto mejora la escalabilidad y el rendimiento, siendo ideal para aplicaciones que requieren actualizaciones en tiempo real [37].

Además, RESTful admite varios formatos de datos, incluyendo JSON y XML. Esta flexibilidad lo hace adaptable a diferentes necesidades de aplicaciones, especialmente en entornos web y móviles [38].

4.4.2. SOAP

SOAP se basa en un protocolo estricto con un formato de mensaje XML específico. Este enfoque garantiza una comunicación estructurada y confiable entre los sistemas, crucial para aplicaciones empresariales complejas [39]. Es versátil en cuanto a transporte, operando sobre protocolos como HTTP y SMTP. Además, SOAP puede mantener el estado en transacciones, facilitando operaciones complejas [40].

La seguridad es un punto fuerte de SOAP. Con WS-Security, ofrece protección a nivel de mensaje, esencial para transacciones seguras en ambientes empresariales donde la integridad de los datos es primordial [41]. SOAP es preferido en entornos que requieren rigurosos estándares de seguridad y pueden manejar una sobrecarga comunicacional mayor, siendo ideal para aplicaciones empresariales complejas [41], [42].

5. Ejemplo

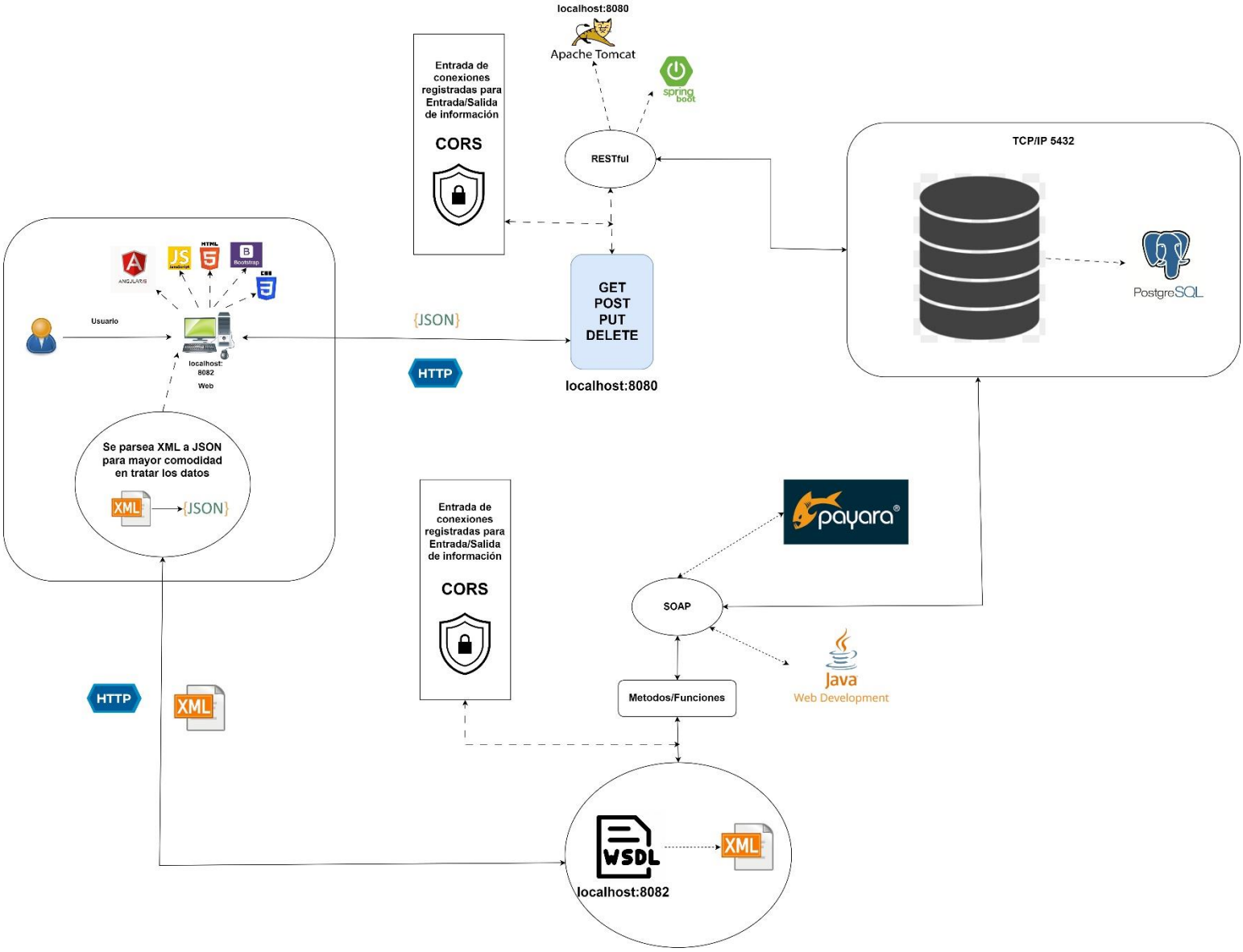

Ilustración 3 Esquema RESTful y SOAP

realizado por LETURNE PLUAS JHON BYRON & CHICA VALFRE VALESKA SOFIA (Grupo E)

5.1. Creación RESTful

En esta no se define el servidor ya que SpringBoot lo ejecuta automáticamente con ApacheTomcat. La versión utilizada para este proyecto es JDK 19.

5.1.1. Configuración de acceso a datos

Se definen las configuraciones principales para el acceso a datos desde nuestra aplicación, incluyendo la especificación del puerto en el que se desplegará la aplicación. Posteriormente, se detallan las configuraciones JDBC, proporcionando el nombre de la base de datos correspondiente. Estas configuraciones son esenciales para establecer una conexión efectiva

entre la aplicación y la base de datos, asegurando un despliegue exitoso y un acceso adecuado a los datos.

```
server.port=8081
spring.jpa.database=postgresql
spring.jpa.hibernate.ddl-auto=update
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.url=jdbc:postgresql://localhost:5432/RSPractica
spring.datasource.username=postgres
spring.datasource.password=123
```

5.1.2. Modelo

Para la implementación del registro de persona, se utilizó SpringBoot, partiendo desde la creación de la entidad o modelo que define la estructura de la tabla en la base de datos. La estructura final de la entidad quedó de la siguiente manera:

```
@Entity
@Table(name="persona")
@Data
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class Persona
{
    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    int idpersona;

    String nombre;
    String apellido;
    String cedula;
    String correo;

    @Column(name = "estado", nullable = false)
    Boolean estado=true;
}
```

Es importante destacar que, con el objetivo de evitar la redundancia en el código, se optó por utilizar Lombok. Esta biblioteca proporciona una ayuda significativa al generar automáticamente los métodos Getters y Setters en las clases. Para lograr esto, simplemente especificamos las anotaciones pertinentes, como @Data, @Getter, @Setter, @AllArgsConstructor y @NoArgsConstructor, en la clase. Estas anotaciones aseguran la

creación automática de los métodos necesarios, incluyendo los constructores, simplificando así el desarrollo.

Es fundamental señalar que, para que la clase sea reconocida como una tabla por el Framework, es necesario utilizar la anotación @Table. Esta anotación indica que la clase es una tabla y permite el mapeo adecuado para la creación en la base de datos especificada en las propiedades del proyecto.

5.1.3. Repositorio

Para habilitar el acceso a la gestión de datos en nuestro aplicativo, creamos una interfaz donde se especifica mediante JpaRepository<clase, Serializable> que las clases que la implementen tendrán la capacidad de conectarse con la base de datos. En estas interfaces, también se definen métodos con una estructura específica que facilita la ejecución de consultas. Además, se emplea el lenguaje JPQL, similar al SQL, para llevar a cabo tareas específicas relacionadas con el acceso a la información.

Similar a la clase modelo, es fundamental incluir la anotación @Repository en la interfaz. Esta anotación asegura que Spring Boot obtenga la referencia adecuada y realice el mapeo necesario para la correcta gestión de datos.

```
@Repository
public interface Ipersona extends JpaRepository<Persona,Serializable>
{
    public Persona findByIdpersona(Long id);

    @Query("Select p from Persona p where p.cedula LIKE %?1%")
    public List<Persona> findByCedula(String cedula);
    public void deleteByIdpersona(Long id);
}
```

5.1.4. Servicio

En esta clase se detallan los servicios que serán utilizados por el controlador, incluyendo los métodos de búsqueda, inserción y eliminación de datos. Es importante señalar que, para acceder a estos servicios, se requiere utilizar la interfaz creada anteriormente, utilizando la anotación @Autowired. Esta anotación permite obtener la referencia del objeto necesario para la interacción con los servicios mencionados. Al igual que las clases e interfaces anteriores se debe usar la notación @Service para especificar que la clase se manejará como un servicio para los datos.

```
@Service
public class Spersona
{
    @Autowired
    private Ipersona person_inter;

    public Persona buscarByIdPersona(Long id)
    {
        return person_inter.findByIdpersona(id);
    }

    public List<Persona> buscarByCedula(String cedula)
    {
        return person_inter.findByCedula(cedula);
    }

    public List<Persona> listar()
    {
        return person_inter.findAll();
    }


    public Persona guardar(Persona per)
    {
        return person_inter.save(per);
    }

    public void eliminar(Persona person)
    {
        person_inter.delete(person);
    }
}
```

5.1.5. Controlador

En esta clase se definen todos los métodos correspondientes a las operaciones CRUD (Create, Read, Update y Delete). En cada uno de ellos, es necesario especificar las operaciones deseadas, ya sea para listar datos, eliminar o insertar. Es relevante destacar que esta misma clase utiliza las anotaciones @RestController, indicando así que será utilizada como un servicio RESTful.

Posteriormente, es necesario especificar la ruta a través de la cual este servicio será accedido mediante la anotación @RequestMapping(). Además, se utiliza la anotación @CrossOrigin() para definir qué direcciones tendrán acceso a la información proporcionada por este servicio.

Es importante tener en cuenta que este servicio hace uso de las clases e interfaces creadas anteriormente. De esta manera, se observa la definición de clases y servicios que contribuyen a la creación del servicio RESTful.

```
@RestController
@RequestMapping("/api")
@CrossOrigin(origins = "http://localhost:8080")
public class ControllerApi
{
    @Autowired
    Spersona spersona;

    @PostMapping("/insertar")
    public ResponseEntity<?> insertar(Persona person)
    {
        Map<String,Object> response=new HashMap();
        try
        {

            Persona p=spersona.guardar(person);

            return new ResponseEntity<Persona>(p,HttpStatus.OK);
        }
        catch(Exception ex)
        {
            response.put("error", ex.getMessage());
            return new
ResponseEntity<Map<String,Object>>(response,HttpStatus.NOT_FOUND);
        }
    }

}
```

5.2. Creación SOAP

Para la creación de este ejemplo hay que tener instalado payara5 y Java8 para evitar conflictos en la ejecución del ejemplo.

5.2.1. Base de datos

Para la creación del ejemplo se procedió a crear la siguiente tabla

```
create table persona
(
      idpersona bigserial primary key,
      nombre varchar(50) not null,
      apellido varchar(50) not null,
      cedula varchar(50) not null,
      correo varchar(50) not null,
      estado boolean default true
```

```
);
```

5.2.2. Dependencia

Esta dependencia es muy importante en el ejemplo SOAP ya que esta permite trabajar con las notaciones son necesarias para su implementación.

```
<dependency>
  <groupId>javax.xml</groupId>
  <artifactId>webservices-api</artifactId>
  <version>2.0.1</version>
  <type>jar</type>
</dependency>
```

5.2.3. Conexión base de datos

A diferencia del servicio RESTful esta mismo no tiene un archivo propiedad en la que se pueda definir las configuraciones directas de la base de datos, de tal manera hay que crearlas manualmente mediante clases.

```
@Getter
public class ConexionPostgres {

    static String login = "postgres";
    static String password = "123";
    static String url = "jdbc:postgresql://localhost/RSPractica";

    private Connection conexion;

    public ConexionPostgres() throws Exception {
        conexion = null;
        Class.forName("org.postgresql.Driver");
        conexion = DriverManager.getConnection(url, login, password);
        if (conexion != null) {
            System.out.println("Conexion a la base de datos " + url + "
......OK");
        }
    }
}
```

5.2.4. Modelo

Para la creación de un servicio SOAP, es necesario definir modelos que faciliten la gestión de datos dentro del servicio. Para lograr esto, creamos una clase utilizando las bibliotecas de Lombok, las cuales contribuyen a la reducción de código. A diferencia del modelo creado con Spring Boot, esta clase no requiere notaciones especiales de un Framework. Esto se debe a que,

al tratarse de un proyecto JavaWeb, la conexión se establece directamente mediante consultas SQL para la comunicación con la base de datos.

```
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class Persona
{
    int idpersona;
    String nombre;
    String apellido;
    String cedula;
    String correo;
    Boolean estado;
}
```

5.2.5. Manejo de datos

Para gestionar los datos, es necesario realizar consultas directas a PostgreSQL. Por lo tanto, se deben crear métodos que lleven a cabo estas consultas. Es importante destacar que en este proceso, se utiliza la clase de conexión para obtener acceso a la base de datos y así poder interactuar con ella.

```
private static String insertar = "insert into persona
(nombre,apellido,cedula,correo) values (?,?,?,?) returning
idpersona,nombre,apellido,cedula,correo,estado";

public Persona insertar(Persona p) {
    try {
        ConexionPostgres cxp = new ConexionPostgres();

        if (p != null) {
            CallableStatement cs = cxp.getConexion().prepareCall(insertar);
            cs.setString(1, p.getNombre());
            cs.setString(2, p.getApellido());
            cs.setString(3, p.getCedula());
            cs.setString(4, p.getCorreo());
            cs.execute();
            ResultSet rs=cs.getResultSet();
            rs.next();
            Persona pp=new Persona();
            pp.setIdpersona(rs.getInt("idpersona"));
            pp.setCedula(rs.getString("cedula"));
            pp.setNombre(rs.getString("nombre"));
            pp.setApellido(rs.getString("apellido"));
            pp.setCorreo(rs.getString("correo"));
```

```
            pp.setEstado(rs.getBoolean("estado"));
            System.out.println(pp.getCedula());
            return pp;
        }

    } catch (Exception ex) {
        System.out.println(ex.getMessage());
    }
    return null;
}
```

5.2.6. Exposición del SOAP

Para exponer el servicio, primero debemos aplicar la anotación @WebService, que indica el nombre que tendrá el servicio SOAP cuando se despliegue. Posteriormente, mediante la anotación @WebMethod, especificamos el nombre del método que será referenciado cuando el servicio esté activo. Cabe destacar que, si no se le asigna un nombre, se nombrará automáticamente al método con el nombre por defecto. Finalmente, la anotación @WebParam se utiliza para indicar el nombre de las variables de entrada que utilizará el método.

```
@WebService(serviceName = "swspersona")
public class swspersona
{
    @WebMethod(operationName = "insertarPersona")
    public Persona insertarPersona(@WebParam(name = "persona") Persona person)
    {
        ManejoDatos md=new ManejoDatos();
        Persona p=md.insertar(person);
        return p;
    }
}
```

5.2.7. CORS

La implementación de esta clase es crucial, ya que define qué dominios tendrán acceso a nuestra información. Es necesario especificar la URL para la cual se otorgarán permisos. A diferencia de Spring Boot, donde utilizábamos una única anotación, en este caso, se requiere una implementación más detallada para definir las restricciones de acceso.

```
@WebFilter(asyncSupported = true, urlPatterns = { "/*" })
public class ManejoCors  implements Filter {
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException
{
```

```
        HttpServletRequest request = (HttpServletRequest) servletRequest;

        ((HttpServletResponse) servletResponse).addHeader("Access-Control-
Allow-Origin", "http://localhost:8080");
        ((HttpServletResponse) servletResponse).addHeader("Access-Control-
Allow-Headers", "*");
        ((HttpServletResponse) servletResponse).addHeader("Access-Control-
Allow-Methods",
                    "GET, OPTIONS, HEAD, PUT, POST, DELETE");

        HttpServletResponse resp = (HttpServletResponse) servletResponse;

        if (request.getMethod().equals("OPTIONS")) {
            resp.setStatus(HttpServletResponse.SC_ACCEPTED);
            return;
        }
        filterChain.doFilter(request, servletResponse);
    }
}
```

5.3. Consumo

Para parte de la vista hay que crear un proyecto JavaWeb está ejecutándose con ApacheTomcat y JDK 11.

5.3.1. SOAP

Para consumir el servicio SOAP, es crucial definir la función JavaScript encargada de realizar la conexión. En este proceso, se utiliza el objeto XMLHttpRequest para estructurar la petición, especificando el WSDL que contiene los métodos a los que se desea conectarse. Posteriormente, se emplea la herramienta SoapUI para obtener el XML de la petición y así poder extraer datos del servicio.

Para facilitar el procesamiento de los datos obtenidos en formato XML, se realiza el parsing a JSON utilizando la librería JavaScript (x2js). Este paso mejora la comodidad al trabajar con los datos obtenidos del servicio SOAP.

```
$scope.url='http://localhost:8082/soapproject/swspersona?wsdl';

consumer_soap("I",undefined,$scope.url,{
    nombre:'Jhon',
    apellido:'Leturne',
    cedula:'1250808373',
    correo:'jlturnep@uteq.edu.ec',
    estado:true,
});
```

```
function consumer_soap(body_type,type,url,object)
{
    var soap=""
    if(body_type=="I")
    {
        soap=`<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soap="http://soapproject.ejemploagentemovil.mycompany.com/">
        <soapenv:Header/>
        <soapenv:Body>
           <soap:insertarPersona>
              <!--Optional:-->
              <persona>
                 <!--Optional:-->
                 <apellido>${object.apellido}</apellido>
                 <!--Optional:-->
                 <cedula>${object.cedula}</cedula>
                 <!--Optional:-->
                 <correo>${object.correo}</correo>
                 <!--Optional:-->
                 <estado>${object.estado}</estado>
                 <idpersona>?</idpersona>
                 <!--Optional:-->
                 <nombre>${object.nombre}</nombre>
              </persona>
           </soap:insertarPersona>
        </soapenv:Body>
     </soapenv:Envelope>`
    }
    else if(body_type=="B")
    {
        soap=`<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soap="http://soapproject.ejemploagentemovil.mycompany.com/">
        <soapenv:Header/>
        <soapenv:Body>
           <soap:buscarPersona>
              <!--Optional:-->
              <cedula>${object.cedula}</cedula>
           </soap:buscarPersona>
        </soapenv:Body>
     </soapenv:Envelope>`
    }

    else soap=undefined;
```

```
    if (soap==undefined) {console.log("Error no coincide con el tipo"); return
undefined;}

    var objXMLHttpRequest = new XMLHttpRequest();
    objXMLHttpRequest.open(type==undefined?'POST':type, url, true);
    objXMLHttpRequest.onreadystatechange = function () {
        if(objXMLHttpRequest.readyState == 4 && objXMLHttpRequest.status ==
200 && body_type=="L" || body_type=="B"){
            result = objXMLHttpRequest;
            console.log(objXMLHttpRequest.responseText);
            var x2js = new X2JS();
            var jsonObj = x2js.xml_str2json( objXMLHttpRequest.responseText );
            console.log(jsonObj);
            resultado="";
            if(body_type=="L")
                resultado=jsonObj.Envelope.Body.listarPersonasResponse.return
            else
                resultado=jsonObj.Envelope.Body.buscarPersonaResponse.return
            $scope.$apply(function()
            {
                console.log(resultado+" ----")
                $scope.personas=resultado;
            });
        }else if (body_type!="L" && body_type!="B")
consumer_soap("L",undefined,$scope.url,undefined);
    }
    objXMLHttpRequest.setRequestHeader("Content-Type", "text/xml; charset=utf-
8");
    objXMLHttpRequest.send(soap);
}
```

5.3.2. RESTful

Para el consumo de servicios RESTful, el proceso es algo similar al descrito anteriormente, con la diferencia de que se utiliza la librería $.ajax en este caso. Esta librería se configura mediante un objeto JSON, donde se especifican los datos que se desean obtener o enviar al servidor. En este ejemplo, los datos se envían al servidor mediante un objeto FormData. Es importante destacar que las claves de FormData deben tener el mismo nombre que los atributos de la clase para que se establezca la correcta asociación.

```
$scope.url='http://localhost:8081';

$scope.actualizar_guardar=(form)=> {
    console.log(form);
```

```
    let formData=new FormData();
    if(form.id.$viewValue!="")
        formData.append("idpersona",form.id.$viewValue);
    formData.append("nombre",form.nombre.$viewValue);
    formData.append("apellido",form.apellido.$viewValue);
    formData.append("cedula",form.cedula.$viewValue);
    formData.append("correo",form.email.$viewValue);
    formData.append("estado",form.activo.$viewValue);

    $.ajax({
        method:form.id.$viewValue==""?"POST":"PUT",
        url:$scope.url+"/api/"+(form.id.$viewValue==""?"insertar":"actualizar"
),
        processData:false,
        contentType: false,
        data:formData,
        beforeSend: function (xhr) {
            console.log("cargando...");
        },
        success: function (data) {
            $scope.$apply(function()
            {
                consumer_rest();
            });
            console.log(data);
        },
        error: function (objXMLHttpRequest)
        {
            console.log("error: ", objXMLHttpRequest);
        }
    });
}
```

5.4. Diseño

Para el diseño de la interfaz de usuario, se ha optado por utilizar AngularJS. A través de JavaScript y jQuery, se lleva a cabo el consumo de las APIs creadas.

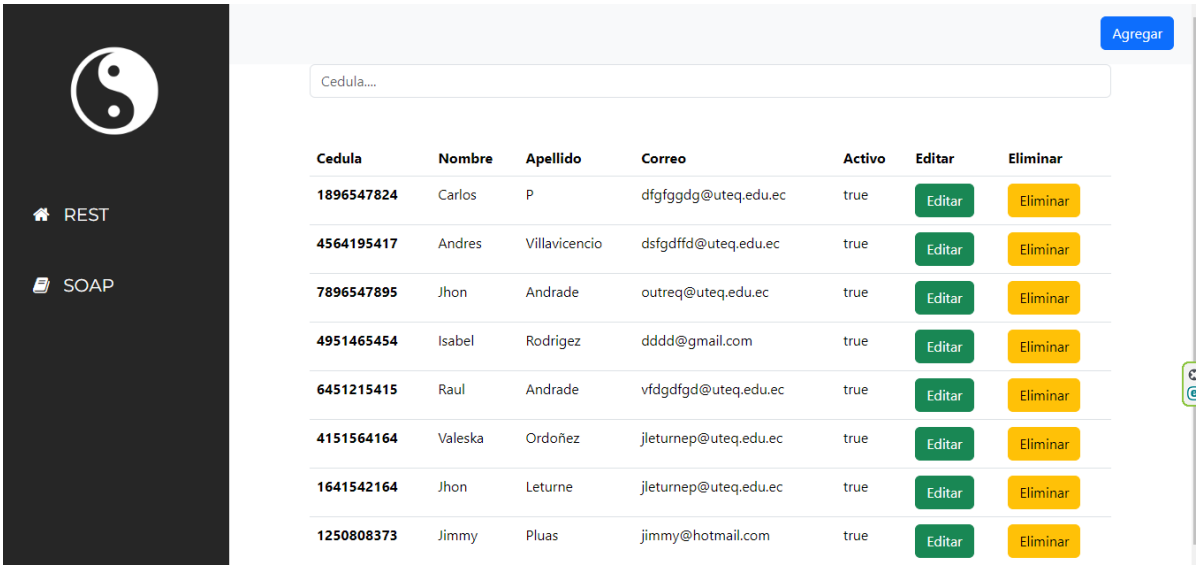

Ilustración 4 Diseño del ejemplo de registro de persona

realizado por LETURNE PLUAS JHON BYRON & CHICA VALFRE VALESKA SOFIA (Grupo E)

6. Recursos utilizados

Todos los recursos utilizados para la creación de los ejemplos de Restful, SOAP y la parte de consumo de la vista están disponibles en el siguiente repositorio de GitHub. Además, se proporcionan las herramientas necesarias para probar los servicios creados.

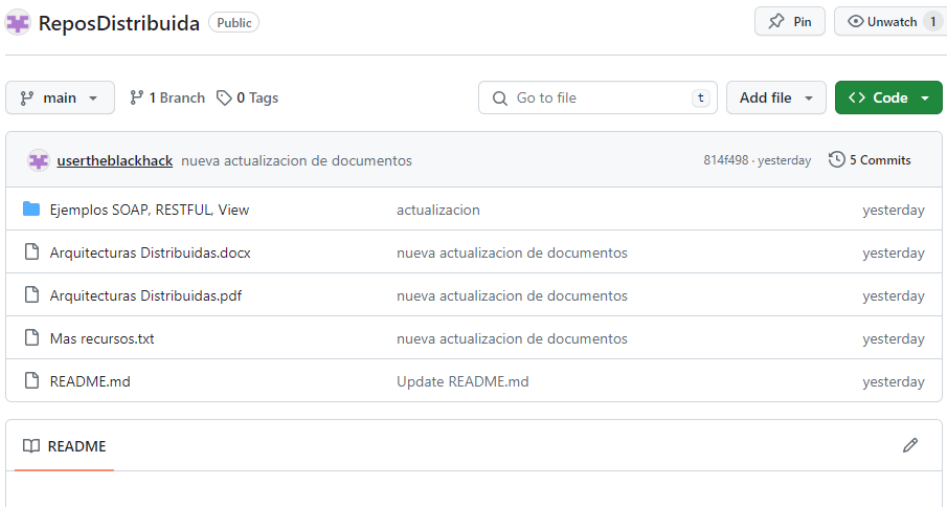

Ilustración 5 Repositorio del ejemplo

realizado por LETURNE PLUAS JHON BYRON & CHICA VALFRE VALESKA SOFIA (Grupo E)

Link GitHub: https://github.com/usertheblackhack/ReposDistribuida

7. Conclusión

Las arquitecturas distribuidas representan una gran ventaja en términos de optimización y ahorro de recursos. Ofrecen flexibilidad en los diseños de redes informáticas gracias a su implementación eficiente. Al descentralizar funciones y capacidades, estas arquitecturas agilizan el procesamiento de datos en comparación con enfoques centralizados.

A largo plazo, estas arquitecturas son beneficiosas para el mantenimiento y la escalabilidad del sistema. Sin embargo, se enfrentan a ciertos desafíos, como la identificación de nodos fallidos, que puede ser un proceso largo al requerir revisar uno por uno.

A través de ejemplos como el modelo cliente-servidor, la arquitectura en capas, Restful y SOAP, se puede apreciar cómo cada uno posee cualidades específicas que afectan la ejecución de servicios de diversas maneras, además de organizar de manera más efectiva los elementos según el objetivo que se esté cumpliendo.

Cuando se utilizan estos tipos de servicios, es necesario realizar tareas como gestionar quién se conecta y qué datos se desean extraer. Todas estas operaciones deben llevarse a cabo de la manera más precisa posible para evitar posibles fugas de información. Esto queda evidenciado al comparar Restful y SOAP, ya que cada uno se orienta de manera diferente, lo que puede complicar la implementación en algunos casos.

Restful, sin embargo, facilita tanto la implementación como la ejecución de tareas. A pesar de que ambas tecnologías manejan estándares de obtención de información diferentes, Restful simplifica la tarea, lo que contribuye a una experiencia más eficiente en comparación con SOAP.

8. Referencias

[1] F. Saleem, F. Farooq, I. S. Chaudhry, y N. Safdar, “How do Globalization, Technological Change and Employment Impact Economic Growth in Developing Countries? Evidence from Panel Data Analysis”, *Review of Applied Management and Social Sciences*, vol. 1, núm. 1, pp. 39–49, dic. 2018, doi: 10.47067/ramss.v1i1.9.

[2] M. Cristiá, “Introduccion a la Arquitectura de Software Testing software from set-based specifications View project”, 2008. doi: 10.13140/RG.2.2.22760.08261.

[3] I. Gorton, “Distributed Systems — What Every Software Architect Should Know”, in *IEEE on Software Architecture Companion (ICSA-C)*, mar. 2023, pp. 339–340. doi: 10.1109/ICSA-C57050.2023.00078.

[4] N. Kratzke, “A Brief History of Cloud Application Architectures”, *Applied Sciences*, vol. 8, núm. 8, p. 1368, ago. 2018, doi: 10.3390/app8081368.

[5] D. Kurata, “Web Application Architectures”, en *Doing Web Development*, Berkeley, CA: Apress, 2002, pp. 415–446. doi: 10.1007/978-1-4302-0852-5_15.

[6] S. K. Lo, Y. Liu, G. Yu, Q. Lu, X. Xu, y L. Zhu, “Distributed Trust Through the Lens of Software Architecture”, may 2023, doi: arxiv-2306.08056.

[7] U. Buy y S. Shatz, “Distributed Software Engineering”, en *Encyclopedia of Software Engineering*, Wiley, 2002. doi: 10.1002/0471028959.sof094.

[8] M. B. McIlrath, D. S. Boning, y D. E. Troxel, “<title>Architecture for distributed design and fabrication</title&gt”;, B. L. M. Goldstein, Ed., ene. 1997, pp. 134–147. doi: 10.1117/12.263462.

[9] M. Mosleh, K. Dalili, y B. Heydari, “Distributed or Monolithic? A Computational Architecture Decision Framework”, *IEEE Syst J*, vol. 12, núm. 1, pp. 125–136, mar. 2018, doi: 10.1109/JSYST.2016.2594290.

[10] H.-M. Heyn, E. Knauss, y P. Pelliccione, “A compositional approach to creating architecture frameworks with an application to distributed AI systems”, *Journal of Systems and Software*, vol. 198, p. 111604, abr. 2023, doi: 10.1016/j.jss.2022.111604.

[11] A. Khole, A. Thakar, A. Kulkarni, H. Jadhav, S. Shende, y V. Karajkhede, “A Compendium on Distributed Systems”, feb. 2023, Consultado: el 18 de enero de 2024. [En línea]. Disponible en: http://arxiv.org/abs/2302.03990

[12] Y. Okuya, N. Ladeveze, O. Gladin, C. Fleury, y P. Bourdot, “Distributed Architecture for Remote Collaborative Modification of Parametric CAD Data”, in *IEEE Fourth VR International Workshop on Collaborative Virtual Environments (3DCVE)*, IEEE, mar. 2018, pp. 1–4. doi: 10.1109/3DCVE.2018.8637112.

[13] M. van Rooij, S. van Rooij, H. Bouma, y A. Pimentel, “Secure Sparse Gradient Aggregation in Distributed Architectures”, in IEEE *on Internet of Things: Systems, Management and Security (IOTSMS)*, nov. 2022, pp. 1–8. doi: 10.1109/IOTSMS58070.2022.10062180.

[14] J. Sventek, “The Distributed Application Architecture”, en *Enterprise Integration Modeling*, The MIT Press, 1992, pp. 481–492. doi: 10.7551/mitpress/2768.003.0058.

[15] J. J. Paul, “Distributed Serverless Architectures”, en *Distributed Serverless Architectures on AWS*, Berkeley, CA: Apress, 2023, pp. 13–22. doi: 10.1007/978-1-4842-9159-7_2.

[16] S. J. Mullender, “Distributed operating systems”, *ACM Comput Surv*, vol. 28, núm. 1, pp. 225–227, mar. 1996, doi: 10.1145/234313.234407.

[17] P. Homburg, M. van Steen, y A. S. Tanenbaum, “An architecture for a wide area distributed system”, en *Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications*, New York, NY, USA: ACM, sep. 1996, pp. 75–82. doi: 10.1145/504450.504465.

[18] H. Hammad, T. Sahmoud, y A. A. R. A. Ghazala, “Convert Monolithic Application to Microservice Application”, jun. 2023, [En línea]. Disponible en: http://arxiv.org/abs/2306.08851

[19] J. Kazanavičius y D. Mažeika, “Evaluation of Microservice Communication While Decomposing Monoliths”, *Computing and Informatics*, vol. 42, núm. 1, pp. 1–36, 2023, doi: 10.31577/cai_2023_1_1.

[20] Y. Abgaz *et al.*, “Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review”, *IEEE Transactions on Software Engineering*, vol. 49, núm. 8, pp. 4213–4242, ago. 2023, doi: 10.1109/TSE.2023.3287297.

[21] J. Kazanavičius y D. Mažeika, “An Approach to Migrate from Legacy Monolithic Application into Microservice Architecture”, in *IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)*, abr. 2023, pp. 1–6. doi: 10.1109/eStream59056.2023.10135021.

[22] Y.-Y. Chen, K.-H. Hsu, y A. W. Hou, “MAT: Automating Go monolithic applications transform into microservices through dependency analysis and AST”, in IEEE *on Applied System Innovation (ICASI)*, abr. 2023, pp. 133–135. doi: 10.1109/ICASI57738.2023.10179517.

[23] N. Kratzke, “A Brief History of Cloud Application Architectures”, *Applied Sciences*, vol. 8, núm. 8, p. 1368, ago. 2018, doi: 10.3390/app8081368.

[24] H. Li, L. Gao, y X. Yang, “Research and Application of Application System Architecture Based on Big Data”, *J Phys Conf Ser*, vol. 1852, núm. 4, p. 042086, abr. 2021, doi: 10.1088/1742-6596/1852/4/042086.

[25] N. Vermeir, “Application Architecture”, en *Introducing .NET 6*, N. Vermeir, Ed., Berkeley, CA: Apress, 2022, pp. 259–273. doi: 10.1007/978-1-4842-7319-7_9.

[26] P. Achimugu, O. Oluwagbemi, y I. Gambo, “Solving Web-based Applications Architectural Problems in the Cloud: The Way Forward”, *International Journal of Information Technology and Computer Science*, vol. 4, núm. 5, pp. 8–15, may 2012, doi: 10.5815/ijitcs.2012.05.02.

[27] R. Peinl, “Architecture of web applications”, *it - Information Technology*, vol. 56, núm. 3, pp. 87–89, jun. 2014, doi: 10.1515/itit-2014-1044.

[28] P. Singh y N. Singh, “Analysis of Free and Open Source Software (FOSS) Product in Web Based Client-Server Architecture”, *International Journal of Open Source Software and Processes*, vol. 9, núm. 3, pp. 36–47, jul. 2018, doi: 10.4018/IJOSSP.2018070103.

[29] M. Goodyear *et al.*, *Enterprise System Architectures*. CRC Press, 2017. doi: 10.1201/9780203757239.

[30] A. Shah, M. G. Servar, y Ms. U. Tomer, “Realtime Chat Application using Client-Server Architecture”, *Int J Res Appl Sci Eng Technol*, vol. 10, núm. 5, pp. 2575–2578, may 2022, doi: 10.22214/ijraset.2022.42848.

[31] I. M. Ibrahim, S. R. M. Zeebaree, H. M. Yasin, M. A. M. Sadeeq, H. M. Shukur, y A. Alkhayyat, “Hybrid Client/Server Peer to Peer Multitier Video Streaming”, in IEEE *on Advanced Computer Applications (ACA)*, jul. 2021, pp. 84–89. doi: 10.1109/ACA52198.2021.9626808.

[32] A. Hussain y P. K. Sharma, “Deployment of Web Application in LAN based 3 Tier Architecture”, *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, pp. 341–345, dic. 2019, doi: 10.32628/CSEIT195661.

[33] W. H. Abdulsalam, “Security For Three-Tiered Web Application”. doi: 10.5281/zenodo.7775908.

[34] D. Sanchez, O. Mendez, y H. Florez, “An Approach of a Framework to Create Web Applications”, 2018, pp. 341–352. doi: 10.1007/978-3-319-95171-3_27.

[35] R. Peinl, “Architecture of web applications”, *it - Information Technology*, vol. 56, núm. 3, pp. 87–89, jun. 2014, doi: 10.1515/itit-2014-1044.

[36] S. H. Toman, “Review of Web Service Technologies: REST over SOAP”, *Journal of Al-Qadisiyah for Computer Science and Mathematics*, vol. 12, núm. 4, nov. 2020, doi: 10.29304/jqcm.2020.12.4.715.

[37] D. Sanchez, O. Mendez, y H. Florez, “An Approach of a Framework to Create Web Applications”, en *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10963 LNCS, Springer Verlag, 2018, pp. 341–352. doi: 10.1007/978-3-319-95171-3_27.

[38] M. I. Beer y M. F. Hassan, “Adaptive security architecture for protecting RESTful web services in enterprise computing environment”, *Service Oriented Computing and Applications*, vol. 12, núm. 2, pp. 111–121, jun. 2018, doi: 10.1007/s11761-017-0221-1.

[39] S. Ahmad, S. Ali, N. Waqar, N. S. Naz, y M. Hassaan Mehmood, “Comparative Evaluation of the Maintainability of RESTful and SOAP-WSDL Web Services”, in IEEE *on Business Analytics for Technology and Security (ICBATS)*, mar. 2023, pp. 1–9. doi: 10.1109/ICBATS57792.2023.10111436.

[40] A. Banubakode y P. Chore, “Growth of Individualizing Web Services using APIs: REST and SOAP”, *SAMRIDDHI : A Journal of Physical Sciences, Engineering and Technology*, vol. 14, núm. Spl-2 issu, pp. 284–290, jun. 2022, doi: 10.18090/samriddhi.v14spli02.15.

[41] S. Malik y D.-H. Kim, “A comparison of RESTful vs. SOAP web services in actuator networks”, in IEEE *on Ubiquitous and Future Networks (ICUFN)*, jul. 2017, pp. 753–755. doi: 10.1109/ICUFN.2017.7993893.

[42] A. Banubakode y P. Chore, “Growth of Individualizing Web Services using APIs: REST and SOAP”, *SAMRIDDHI : A Journal of Physical Sciences, Engineering and Technology*, vol. 14, núm. Spl-2 issu, pp. 284–290, jun. 2022, doi: 10.18090/samriddhi.v14spli02.15.