SOAP

Simbaña Moreira Adrian Isaee
Socasi Guallichico Moisés Benjamín
Torres Chávez Marlon Pavel

Universidad de las Fuerzas Armadas ESPE
Departamento de Ciencias de la Computación
Aplicaciones Distribuidas

Tabla de Contenido

Introducción

SOAP es un protocolo estándar de comunicación que facilita el intercambio de datos estructurados entre aplicaciones distribuidas a través de redes. Su diseño se centra en garantizar la interoperabilidad entre sistemas heterogéneos, independientemente del lenguaje de programación o las plataformas utilizadas. SOAP utiliza XML como formato para estructurar los mensajes, lo que asegura una representación universalmente comprensible y estandarizada de los datos.

SOAP generalmente utiliza el protocolo HTTP como transporte, aunque también puede operar sobre otros como SMTP. Esto permite que los mensajes SOAP viajen a través de firewalls y redes utilizando tecnologías ampliamente aceptadas. El protocolo se compone de tres partes principales: un sobre (envelope) que define el marco del mensaje, un conjunto de reglas de codificación para representar los datos, y un mecanismo para definir las llamadas y respuestas en la interacción cliente-servidor.

Aunque este protocolo a veces se percibe como más complejo que alternativas modernas, su robustez y características avanzadas lo convierten en una elección ideal para escenarios que requieren transacciones distribuidas, manejo de errores detallado y capacidades de seguridad avanzadas. En este informe, exploraremos la implementación de un cliente SOAP en Node.js, destacando cómo se utiliza este protocolo para consumir servicios web de manera eficiente y estructurada.

Objetivos

Objetivo Principal

Implementar un cliente SOAP en Node.js para consumir los servicios de una calculadora, analizando su funcionamiento y destacando las ventajas del protocolo en este tipo de aplicaciones.

Objetivos específicos

- Evaluar la facilidad de integración y la interoperabilidad del servicio SOAP en un entorno basado en Node.js.
- Probar y validar la precisión y el rendimiento del servicio web consumido, asegurando que las operaciones matemáticas respondan correctamente.

Justificación

Este informe se enfoca en el consumo de un servicio SOAP que implementa una calculadora, lo que permite explorar un caso práctico y sencillo para demostrar las capacidades de este protocolo. La elección de este caso de uso se justifica porque las operaciones matemáticas básicas (suma, resta, multiplicación y división) son ideales para ilustrar la estructura de las solicitudes y respuestas SOAP, así como los principios de diseño subyacentes.

Además, el uso de Node.js como entorno de desarrollo agrega valor al análisis, ya que es una plataforma ampliamente utilizada para construir aplicaciones escalables y eficientes. Al implementar un cliente SOAP en Node.js, se busca destacar su compatibilidad con tecnologías tradicionales como SOAP, lo que reafirma su versatilidad en proyectos de integración.

Desarrollo

Estructura del proyecto

Implementamos un servidor en Node.js usando Express para consumir un servicio SOAP de una calculadora en línea. Define una función genérica para realizar operaciones SOAP (suma, resta, multiplicación y división) basándose en el WSDL proporcionado. Valida las entradas de

las solicitudes para asegurarse de que sean números y maneja errores específicos, como la división por cero. Cada operación está expuesta como una ruta POST que procesa los valores enviados en el cuerpo de la solicitud, finalmente, el servidor se ejecuta en el puerto 3000 para responder a las solicitudes de los clientes.

app.js

app.js

```
const express = require("express");
const soap = require("soap");
const app = express();

const URL_SOAP = "http://www.dneonline.com/calculator.asmx?WSDL";

// Middleware para manejar JSON
app.use(express.json());

// Función genérica para realizar una operación SOAP
const realizarOperacion = async (operacion, argumentos) => {
  const cliente = await soap.createClientAsync(URL_SOAP);
  return cliente[operacion + "Async"](argumentos);
};



// Función para validar los valores de entrada
const validarEntradas = (req, res, next) => {
  const { intA, intB } = req.body;

  if (intA === undefined || intA === null || intB === undefined || intB === null) {
    return res.status(400).json({ error: "Por favor, ingrese ambos valores (intA e intB)." });
  }

  if (typeof intA !== "number" || typeof intB !== "number") {
    return res.status(400).json({ error: "Los valores intA e intB deben ser números." });
  }

  next();
};

// Rutas
app.post("/sumar", validarEntradas, async (req, res) => {
  const { intA, intB } = req.body;
  try {
    const resultado = await realizarOperacion("Add", { intA, intB });
```

```
    res.json({ resultado: resultado[0].AddResult });
  } catch (error) {
    res.status(500).json({ error: "Error al realizar la suma: " + error.message });
  }
});

app.post("/restar", validarEntradas, async (req, res) => {
  const { intA, intB } = req.body;
  try {
    const resultado = await realizarOperacion("Subtract", { intA, intB });
    res.json({ resultado: resultado[0].SubtractResult });
  } catch (error) {
    res.status(500).json({ error: "Error al realizar la resta: " + error.message });
  }
});

app.post("/multiplicar", validarEntradas, async (req, res) => {
  const { intA, intB } = req.body;
  try {
    const resultado = await realizarOperacion("Multiply", { intA, intB });
    res.json({ resultado: resultado[0].MultiplyResult });
  } catch (error) {
    res.status(500).json({ error: "Error al realizar la multiplicación: " + error.message });
  }
});

app.post("/dividir", validarEntradas, async (req, res) => {
  const { intA, intB } = req.body;

  // Validar división por cero
  if (intB === 0) {
    return res.status(400).json({ error: "No se puede dividir entre 0." });
  }
  try {
    const resultado = await realizarOperacion("Divide", { intA, intB });
    res.json({ resultado: resultado[0].DivideResult });
```

```
  } catch (error) {
    res.status(500).json({ error: "Error al realizar la división: " + error.message });
  }
});
// Iniciar el servidor
const PUERTO = 3000;
app.listen(PUERTO, () => {
  console.log(`Servidor corriendo en http://localhost:${PUERTO}`);
});
```

DockerFile

DockerFile

```
# Usa una imagen base de Node.js
FROM node:16

# Establece el directorio de trabajo dentro del contenedor
WORKDIR /usr/src/app

# Copia los archivos del proyecto
COPY package*.json ./
COPY app.js ./

# Instala las dependencias
RUN npm install

# Expone el puerto de la aplicación
EXPOSE 3000

# Comando para ejecutar la aplicación
CMD ["node", "app.js"]
```

package.json

Definimos los metadatos y las configuraciones de la aplicación Node.js especificando la versión del proyecto, el archivo de inicio (app.js), junto con dos scripts: start para ejecutar la aplicación y un script de prueba predeterminado. Además, listamos las dependencias necesarias, incluyendo express para gestionar el servidor web y soap para consumir servicios SOAP.

package.json

```
{
  "name": "api-soap",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node app.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "express": "^4.21.1",
    "soap": "^1.1.6"
  }
}
```

Ejecución del proyecto

Iniciar el proyecto

Dentro de nuestro proyecto ejecutamos nuestra aplicación con el comando

```
npm start
```

Una vez ya se encuentre ejecutando nuestra aplicación se nos mostrará un mensaje

```
PS D:\ESPE OCT24 - MAR25\Distribuidas\Primer Parcial\api-soap\api-soap> npm start

> api-soap@1.0.0 start
> node app.js

Servidor corriendo en http://localhost:3000
```

Figura 1. Iniciación del proyecto

Realizar solicitudes

Realizamos las solicitudes en este caso haciendo uso de Postman a nuestro localhost en el puerto 3000 y revisamos el resultado.

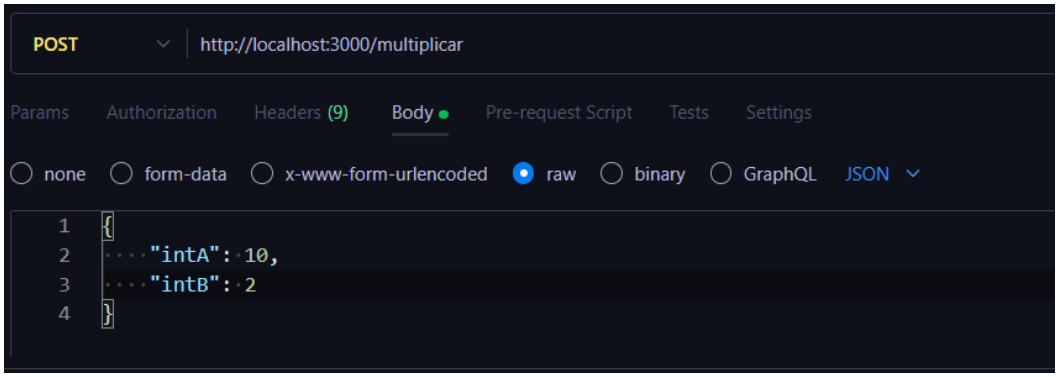

Imagen 2. Solicitud mediante postman

Imagen 3. Resultado de respuesta

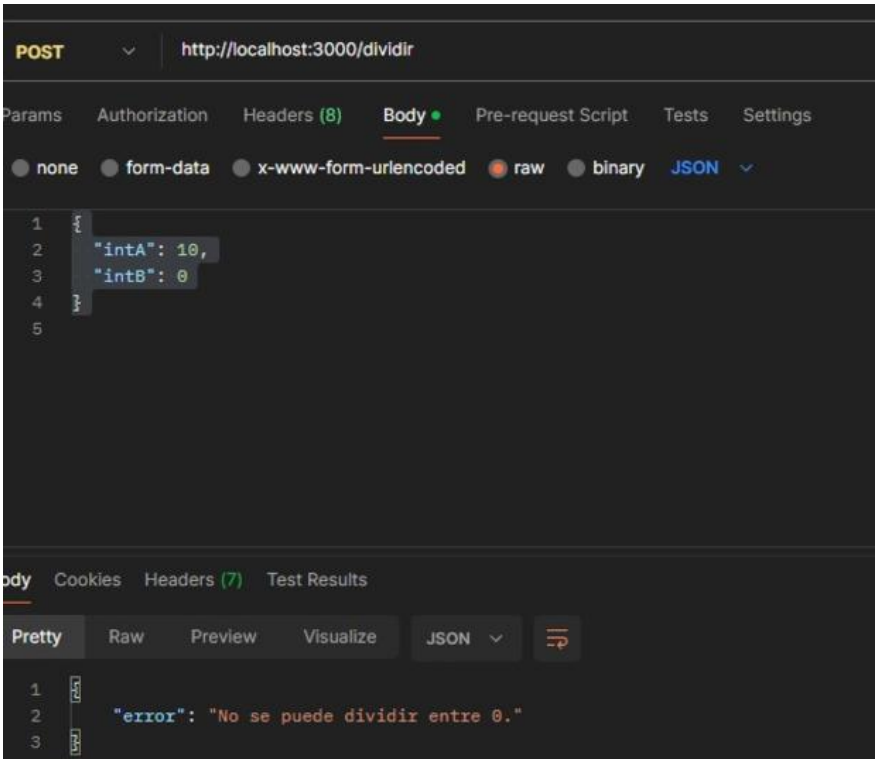

Imagen 4. Validación sobre la división entre 0

Conclusiones

La implementación de un cliente SOAP en Node.js demuestra la versatilidad de esta plataforma para integrar servicios web basados en tecnologías tradicionales como SOAP. A través de un caso práctico como una calculadora, se evidencia la robustez del protocolo, especialmente en escenarios que requieren una estructura de mensajes estandarizada y soporte multiplataforma.

Aunque SOAP puede percibirse como más complejo frente a alternativas modernas como REST, su capacidad de manejar operaciones seguras y consistentes lo hace ideal para ciertos entornos empresariales. Node.js, con sus herramientas como el paquete soap, simplifica el proceso de consumo de servicios, integrando tecnologías antiguas en aplicaciones modernas de manera eficiente.

Recomendaciones

Para proyectos futuros que utilicen SOAP, es importante documentar correctamente las operaciones disponibles en el servicio WSDL, lo que agilizará la implementación y permitirá identificar posibles problemas de integración de forma temprana.

Aunque SOAP es útil en muchos casos, se recomienda evaluar alternativas como REST si el proyecto no requiere las características avanzadas de SOAP, como su soporte robusto para transacciones y seguridad. Esto puede simplificar el desarrollo y mejorar el rendimiento en aplicaciones menos críticas.

Bibliografía

Rational Software Architect Standard Edition 7.5.5. (2020). https://www.ibm.com/docs/es/rsas/7.5.0?topic=standards-soap

Arsys. (2019). *¿Qué es SOAP (Simple Object Access Protocol)?* https://www.arsys.es/blog/que-es-soap-simple-object-access-protocol#:~:text=SOAP%20es%20un%20protocolo%20estandarizado,tipo%20de%20sistemas%20y%20servidores.