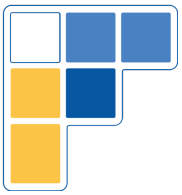
UNIVERSIDAD DE TARAPACÁ

FACULTAD DE INGENIERÍA



DEPARTAMENTO DE INGENIERÍA EN COMPUTACIÓN E INFORMÁTICA

Taller 2:

Microservicios

Alumnos: Patricio Chang
Karen Correa
Juan Carlos Yampara

Asignatura: Sistema Distribuidos

Profesor: Leonel Alarcón Bravo

Arica, 20 de noviembre del 2024

1. Tabla de Contenidos

1. Tabla de Contenidos **2**
2. Tabla de figuras **2**
3. Introducción **3**
4. Objetivo General **4**
5. Objetivos Específicos **4**
6. Desarrollo **5**
6.1. Análisis de funcionalidades 5
Gestión de Productos 5
6.2. Elección y justificación de herramientas tecnológicas 5
Nestjs 5
6.3. Desarrollo y aplicación del middleware 6
Funcionalidades 6
1. Gestión de solicitudes 6
2. Tolerancia a fallos 6
3. Comunicación entre frontend y backend 6
4. Sincronización de operaciones 6
6.4. Desarrollo y explicación de microservicios 6
Microservicio de Login 8
Microservicio de Registro 12
Microservicio de Gestión de productos 13
Microservicio Espejo 14
6.5. Simulación de ejecución de microservicios (fallos) 14
7. Conclusiones **15**
8. Referencias bibliográficas **15**

Arica, 20 de noviembre del 2024

2. Tabla de figuras

Figura 1: Controlador “usuario.controller.ts” del microservicio “Auth” 8
Figura 2: Login en “usuario.service.ts” del microservicio “Auth”. 9
Figura 3: Función “existUsuario” del microservicio “Auth”. 10
Figura 4: Clase “LoginUsuarioDto” del microservicio “Auth”. 10
Figura 5: “main.ts” del microservicio “Auth”. 11
Figura 6: “main.ts” del microservicio “Registro”. 12
Figura 7: Clase “CreateUsuarioDto” del microservicio “Registro”. 12
Figura 8: Función del servicio usuario del microservicio “Registro”. 13
Figura 9: Controlador del microservicio “Registro”. 13
Figura 10: Servicio del microservicio “Producto”. 14
Figura 11: Controlador del microservicio “Producto”. 15
Figura 12: Error del microservicio principal “Productos” 15
Figura 13: Manejo de errores para servicio “Productos”, función “Post” 16
Figura 14: Frontend de Productos, “Ver-productos”. 16
Figura 15: Manejo de errores para servicio “Productos”, función “Get”. 17

Arica, 20 de noviembre del 2024

3. Introducción

La arquitectura de microservicios se basa en que dentro de un sistema, una aplicación se divide en distintos servicios, todos independientes entre ellos.
Cada uno de estos microservicios cumple con una función específica, y se pueden comunicar entre ellos según sea requerido.
En este informe se documenta el desarrollo de un sistema de gestión de productos utilizando la arquitectura de microservicios.
El sistema consiste en gestionar productos de un inventario, además se debe garantizar la disponibilidad del sistema en caso de que algún microservicio falle, en este caso, para asegurar esto, se utiliza un microservicio espejo que debe estar disponible en caso de que el microservicio principal de gestión de productos falle.
Además, el sistema cuenta con microservicios de registro y login de usuarios para proteger la seguridad de los datos.
En este trabajo se evidencia la implementación y pruebas del sistema frente a fallos, se detallan las tecnologías utilizadas, se explican las funcionalidades realizadas y la aplicación del middleware.

Arica, 20 de noviembre del 2024

4. Objetivo General

Desarrollar un sistema de gestión de productos utilizando una arquitectura de microservicios que se comuniquen entre sí y permita garantizar la disponibilidad ante fallos.

5. Objetivos Específicos

- Establecer una conexión entre los microservicios utilizando protocolo TCP.
- Desarrollar las funcionalidades para gestionar los productos.
- Mantener la sincronización entre el microservicio para la gestión de productos y su espejo.

Arica, 20 de noviembre del 2024

6. Desarrollo

6.1. Análisis de funcionalidades

Gestión de Productos

La gestión de productos es aquella que implementada en dos microservicios (espejo y principal) son la parte central del proyecto. En un caso hipotético, que la consulta de productos sea mayor o se cae un microservicio relacionado a este, el servicio como tal no se verá interrumpido debido a que está el microservicio espejo que podrá responder a las consultas que el principal contestaba.

La funcionalidad de gestión de productos se basa en la obtención, creación y eliminación de productos.

6.2. Elección y justificación de herramientas tecnológicas

Nestjs

Es un framework para el desarrollo del backend, posee librerías para poder gestionar los microservicios. No es tan solo por el hecho de ser una excelente herramienta para el backend sino también por la experiencia que tiene gran parte del equipo por lo cual se escogió este framework.

Prisma

Es un conjunto de herramientas que sirven para poder realizar consultas a una base de datos, fue seleccionado por su simplicidad y la experiencia que posee el equipo en esta herramienta.

Mysql

Es un sistema de gestión de base de datos ampliamente utilizado por el equipo, por lo cual fue seleccionado por el conocimiento previo que tiene el equipo sobre esta herramienta.

Angular

Es un framework ampliamente conocido y utilizado para desarrollar el frontend de un proyecto web. Se seleccionó esta herramienta debido a que posee una facilidad de uso y es una excelente opción para desarrollar en conjunto con nestjs.

6.3. Desarrollo y aplicación del middleware

En nuestra arquitectura de microservicios, el middleware actúa como un puente que interconecta los microservicios de login, registro, gestión de productos y su espejo. Su función principal es garantizar la comunicación y validación de datos entre ellos, permitiendo un flujo seguro y eficiente.

Funcionalidades

1. Gestión de solicitudes
    - Permite crear y consultar productos mediante solicitudes al microservicio principal o al espejo en caso de fallos.
    - Conecta las acciones de registro y login con sus respectivos microservicios.
2. Tolerancia a fallos
    - Detecta errores de conexión con el microservicio principal de productos y redirige automáticamente la solicitud al microservicio espejo.
3. Comunicación entre frontend y backend
    - Recibe solicitudes HTTP del frontend, las transforma en mensajes basados en comandos y las envía al microservicio correspondiente utilizando protocolos de comunicación asincrónica.
4. Sincronización de operaciones
    - Mantiene la integridad de los datos sincronizando los cambios realizados entre los microservicios de productos y su espejo

6.4. Desarrollo y explicación de microservicios

Los microservicios de login, registro y de productos se desarrollaron como proyectos independientes, en los cuales se posee la siguiente estructura:

- **Servicio**: Aquel que se encarga de obtener datos de la base de datos y contener la lógica para tratarlos.
- **Controlador**: Se encarga de definir los endpoint o rutas para las peticiones HTTP y devolver los datos solicitados al cliente (como lo puede ser el frontend).
- **Módulo**: Se encarga de agrupar los servicios, controladores e importar, básicamente agrupa diferentes elementos con la finalidad de organizarlos.

Además, para operar en la base de datos se implementa el cliente de prisma, así con las funcionalidades '.create()', '.findMany()', entre otras, se puede obtener o alterar la base de datos.

Microservicio de Login

El controlador de servicio define el endpoint ‘usuario’, hace uso del ‘usuarioService’ para poder contestar solicitudes HTTP. Dentro de este se puede observar la función ‘login’ el cual recibe un dto (data transfer object) que contiene la información con la cual el usuario pueda iniciar sesión.

```
@Controller('usuario')

export class UsuarioController {


    constructor(
        private readonly _usuarioService: UsuarioService,
    ){}
    //@Post('login')
    @MessagePattern({cmd:'login-user'})
    public login(@Payload() login: LoginUsuarioDto){
        return this._usuarioService.loginUsuario(login);
    }
}
```

Figura 1: Controlador “usuario.controller.ts” del microservicio “Auth”

Arica, 20 de noviembre del 2024

Este es la parte del servicio de usuario, el cual recibe como parámetro al dto del login de usuario, dada la conexión con la base de datos mediante el servicio 'DatabasaService' inyectado como '_databaseService' se puede realizar consultas a la base de datos con la finalidad de validar los datos del usuario y así poder autenticarlo. También se posee el manejo de excepciones, por ejemplo 'RpcException' es un tipo de error el cual se recomienda usar al momento de trabajar con microservicios, mediante un bloque try-catch se lanza un error el cual mostrará si el usuario no ha sido validado.

```
public async loginUsuario(loginUsuario: LoginUsuarioDto){
    try {
        if(!await this.existUsuario(loginUsuario.correo)){
            throw new RpcException({
                status: HttpStatus.BAD_REQUEST,
                message: 'contraseña o correo inválidos'
            });
        };
        const auth = await this._databaseService.usuario.findFirst({
            where:{
                correo: loginUsuario.correo,
                password: loginUsuario.password,
            }
        });
        if(!auth){
            throw new RpcException({
                status: HttpStatus.UNAUTHORIZED,
                message: 'Contraseña o correo inválidos'
            });
        }
        return true;
    }catch(error){
        if(error instanceof RpcException){
            throw error;
        }else {
            throw new RpcException({
                status: HttpStatus.INTERNAL_SERVER_ERROR,
                message: 'Error interno del servidor al logear usuario'
            });
        }
    }
}
```

Figura 2: Login en "usuario.service.ts" del microservicio "Auth".

Sumado a lo anterior, mediante el correo se puede validar la existencia del usuario mediante el correo.

```
public async existUsuario(correo: string){
    const existeUsuario = await this._databaseService.usuario.findFirst({
        where: {
            correo: correo,
        }
    });


    if(!existeUsuario){
        return false;
    }
    return true;
}
```

Figura 3: Función “existUsuario” del microservicio “Auth”.

El dto que debe recibir el controller mediante la ruta ‘usuario/login-user’ posee la siguiente estructura:

```
export class LoginUsuarioDto {

    @IsNotEmpty()
    @IsString()
    correo: string;

    @IsNotEmpty()
    @IsString()
    password: string;
}
```

Figura 4: Clase “LoginUsuarioDto” del microservicio “Auth”.

Arica, 20 de noviembre del 2024

El archivo main.ts del microservicio de autenticación posee la siguiente estructura, la cual genera un microservicio en lugar de una aplicación tradicional HTTP, para ello se puede observar la sección de código ‘const app…’

```
async function bootstrap() {

  const logger = new Logger('Auth');

  const app = await NestFactory.createMicroservice<MicroserviceOptions>(AppModule,{
    transport: Transport.TCP,
    options: {
      port: envs.port
    }
  }
  );

  await app.listen();
  logger.log(`Auth microservice running on port ${envs.port}`)
}
bootstrap();
```

Figura 5: “main.ts” del microservicio “Auth”.

Microservicio de Registro

El microservicio de registro consta de una estructura similar al de autenticación.

```
async function bootstrap() {

  const logger = new Logger('Registro');
  const app = await NestFactory.createMicroservice<MicroserviceOptions>(AppModule,
    {
      transport: Transport.TCP,
      options:{
        port: envs.port
      }
    }
  );

  await app.listen();
  logger.log(`Register microservice running on port ${envs.port}`)
}
bootstrap();
```

Figura 6: “main.ts” del microservicio “Registro”.

Para crear un usuario se definen la siguiente clase:

```
export class CreateUsuarioDto {

    nombre: string;
    correo: string;
    password: string;

}
```

Figura 7: Clase “CreateUsuarioDto” del microservicio “Registro”.

La lógica contenido en el servicio de usuario es la siguiente:

```
public async crearUsuario(usuarioCrear: CreateUsuarioDto) {
    try {
      // Crear el nuevo usuario sin verificación de correo duplicado
      const nuevoUsuario = await this._dataBaseService.usuario.create({
        data: usuarioCrear,
      });

      return nuevoUsuario;
    } catch (error) {
      // Eliminar la parte de verificación del correo
      throw new RpcException({
        status: HttpStatus.INTERNAL_SERVER_ERROR,
        message: 'Error interno al crear el usuario',
      });
    }
  }
```

Figura 8: Función del servicio usuario del microservicio “Registro”.

Mediante el controller se genera la ruta para que se puedan crear usuarios mediante el microservicio ‘register’.

```
@Controller()
export class UsuarioController {

    constructor(
        private readonly _usuarioService: UsuarioService,
    ){}

    //@Post('crear-cuenta')
    @MessagePattern({ cmd: 'crear' })
    public crearCuenta(@Body() usuario: CreateUsuarioDto){
        return this._usuarioService.crearUsuario(usuario);
    }
}
```

Figura 9: Controlador del microservicio “Registro”.

Microservicio de Gestión de productos

El servicio de gestión de producto deconstruye el objeto ‘crearProductoDto’ para poder generar un producto en la base de datos.

```
@Injectable()
export class ProductosService extends PrismaClient implements OnModuleInit {
  private readonly logger = new Logger('ProductosService');
  onModuleInit() {
    this.$connect();
    this.logger.log('Database connected');
  }

  async crear(crearProductoDto: CrearProductoDto) {
    const productoCreado = await this.productos.create({
      data: {
        nombre: crearProductoDto.nombre,
        descripcion: crearProductoDto.descripcion,
        precio: crearProductoDto.precio,
      },
    });
    return productoCreado;
  }

  async findAll() {
    const listaProductos = await this.productos.findMany();
    return listaProductos;
  }
}
```

Figura 10: Servicio del microservicio “Producto”.

El controlador define las rutas para poder crear y obtener los productos del microservicio productos.

```
@Controller()
export class ProductosController {
  constructor(private readonly productosService: ProductosService) {}

  @MessagePattern({ cmd: 'crear' })
  crear(@Payload() crearProductoDto: CrearProductoDto) {
    return this.productosService.crear(crearProductoDto);
  }

  @MessagePattern({ cmd: 'obtener' })
  findAll() {
    console.log('Solicitando todos los productos');
    return this.productosService.findAll();
  }
}      Kassi-2, hace 3 días • microservicio de agregar producto: fronte
```

Figura 11: Controlador del microservicio “Producto”.

Microservicio Espejo

6.5. Simulación de ejecución de microservicios (fallos)

Al cerrarse inesperadamente la conexión, se lanza el error “ECONNRESET”.

```
[Nest] 9012  - 19-11-2024, 22:16:44     LOG [RouterExplorer] Mapped {/auth/login, POST} route +0ms
[Nest] 9012  - 19-11-2024, 22:16:44     LOG [NestApplication] Nest application successfully started +2ms
[Nest] 9012  - 19-11-2024, 22:16:44     LOG [Gateway] Gateway running on port 3000
[Nest] 9012  - 19-11-2024, 22:39:51   ERROR [ClientTCP] Error: read ECONNRESET
```

Figura 12: Error del microservicio principal “Productos”

Debido a que este sistema mantiene disponible la gestión de productos frente a fallos, al cerrarse la conexión del microservicios principal de productos, se habilita el microservicio espejo.

```typescript
@Controller('productos')
export class ProductosController {
  constructor(
    @Inject('ProductosService') private readonly productosClient: ClientProxy,
    @Inject('ProductosEspejoService') private readonly productosEspejoClient: ClientProxy,

  ) {}
  @Post()
  public async create(@Body() newProducto: CrearProductoDto) {
    try {
      const producto = await firstValueFrom(
        this.productosClient.send({ cmd: 'crear' }, newProducto),
      );
      return producto;
    } catch (error) {
      if (error.code === 'ECONNREFUSED') {
        return this.productosEspejoClient
          .send({ cmd: 'crear' }, newProducto)
          .pipe(
            catchError((err) => {
              throw new RpcException(err);
            }),
          );
      }
      throw new RpcException(error);
    }
  }
```

Figura 13: Manejo de errores para servicio “Productos”, función “Post”

Por ejemplo en esta función de crear productos, si falla la conexión principal, se habilita la función de agregar productos, para este trabajo se diferencian en que el microservicio espejo agrega el nombre y la descripción de los productos en mayúscula.

Figura 14: Frontend de Productos, “Ver-productos”.

Igualmente la función de ver los productos está en ambos servicios, por lo que si falla la del microservicio principal, funciona la del microservicio espejo.

Arica, 20 de noviembre del 2024

```
@Get()
public async findAll() {
  try {
    const criminals = await firstValueFrom(
      this.productosClient.send({ cmd: 'obtener' }, {}),
    );
    return criminals;
  } catch (error) {
    if (error.code === 'ECONNREFUSED') {
      return this.productosEspejoClient
        .send({ cmd: 'obtener' }, {})
        .pipe(
          catchError((err) => {
            throw new RpcException(err);
          }),
        );
    }

    throw new RpcException(error);
  }
}
```

Figura 15: Manejo de errores para servicio “Productos”, función “Get”.

7. Conclusiones

Los microservicios tienen el objetivo de ser utilizados para la creación de aplicaciones escalables, modulares y sencillas de mantener, sin embargo, para ello debe existir una correcta coordinación entre estos al momento de interactuar. Durante la implementación de microservicios de este taller, se definió el gateway que se encarga de organizar las peticiones para redirigirlas al microservicio encargado de responder dicha petición, también se observó la utilización de microservicios ante fallos, como lo fue al implementar un microservicio espejo.

Arica, 20 de noviembre del 2024

8. Referencias bibliográficas

https://docs.nestjs.com/microservices/basics

Arica, 20 de noviembre del 2024