

Desacoplamiento de un monolito

Introducción

En el desarrollo de software, los monolitos han sido una arquitectura común debido a su simplicidad inicial y facilidad de despliegue. Sin embargo, a medida que los sistemas crecen en complejidad y demanda, surgen limitaciones en términos de escalabilidad, mantenibilidad y disponibilidad. Para abordar estos desafíos, es fundamental explorar técnicas que permitan desacoplar los componentes de un monolito, reduciendo su interdependencia y mejorando su capacidad de evolución.

Este trabajo se centra en la implementación de un enfoque de desacoplamiento de un monolito utilizando Amazon SQS (Simple Queue Service), un servicio de mensajería distribuida que permite la comunicación asíncrona entre componentes. Mediante el uso de SQS, los diferentes módulos del monolito pueden operar de manera más independiente, permitiendo una arquitectura más flexible y escalable. Se analizará cómo esta técnica no solo mejora la resiliencia del sistema, sino que también facilita una transición gradual hacia arquitecturas más modernas, como microservicios, sin comprometer la funcionalidad existente.

Nuestro monolito

El monolito que vamos a desacoplar gestiona la operación de Autoescuelas Eco, cubriendo funcionalidades clave como la administración de autoescuelas, profesores y vehículos. Actualmente, los módulos del sistema están conectados mediante un EventBus en memoria, lo que permite la comunicación interna sincrónica. Sin embargo, para mejorar la escalabilidad, resiliencia y mantenibilidad del sistema, implementaremos Amazon SQS para manejar la comunicación asíncrona entre módulos, lo que permitirá desacoplarlos progresivamente.

Este enfoque sigue lo que se conoce como un Loosely Coupled Monolith, una arquitectura monolítica que, aunque conserva la estructura de un monolito, facilita el desacoplamiento interno mediante prácticas de Domain-Driven Design (DDD) y Arquitectura Hexagonal.

- *Domain-Driven Design (DDD): Permite organizar el sistema en Bounded Contexts claros, donde cada módulo, como Autoescuelas, Profesores y Vehículos, encapsula su propia lógica de negocio y sus entidades. Los eventos del dominio se utilizan para comunicar cambios de estado importantes entre contextos, manteniendo la coherencia sin acoplar fuertemente los módulos.*
- *Arquitectura Hexagonal: Separa la lógica de negocio central de las preocupaciones externas, como la persistencia o la comunicación con otros servicios. Esto se logra mediante puertos y adaptadores, permitiendo que la lógica central interactúe con el exterior a través de interfaces bien definidas. En este caso, el EventBus (que será reemplazado por SQS) actúa como un puerto que conecta los módulos internos con los sistemas externos.*

Los eventos clave que maneja este monolito incluyen:

1. *TeacherHasBeenFiredEvent: Evento que se dispara al despedir a un profesor, con detalles como el DNI, nombre, apellido, y la autoescuela.*
2. *TeacherHasBeenHiredEvent: Evento que indica la contratación de un nuevo profesor, con información sobre el DNI, nombre, apellido, y la autoescuela.*
3. *VehicleHasBeenAssociatedToTeacherEvent: Se emite cuando un vehículo es asociado a un profesor, incluyendo datos del vehículo y la autoescuela.*
4. *VehicleHasBeenBoughtEvent: Evento que registra la compra de un nuevo vehículo, especificando marca, modelo, matrícula y autoescuela.*
5. *DrivingSchoolSectionHasBeenCreated: Evento que se dispara al crear una nueva sección en una autoescuela, con el ID de la sección.*

Al implementar Amazon SQS para la comunicación entre estos eventos, transformaremos este monolito en un sistema más flexible y preparado para el escalado, manteniendo la cohesión de la lógica de negocio central pero reduciendo el acoplamiento entre los módulos.

Desacoplamiento

Al haber sido creado nuestro monolito partiendo de base de una arquitectura Event-Driven solo tendremos que implementar la interfaz del EventBus con SQS.

Interfaz:

```
package com.eduortza.pepeducacion.core.shared.application;

import ...

public interface IEventBus { 2 implementations  EduardoOrtegaZerpa +1
    void publish(final List<DomainEvent> events); 6 usages 2 implementations  EduardoOrtegaZerpa
    void subscribe(IEventHandler handler); 4 usages 2 implementations  Pepe
}
```

Implementación:

```
public class SqsEventBus implements IEventBus { 2 usages  EduardoOrtegaZerpa *
    private final AmazonSQS sqsClient = AmazonSQSClientBuilder.standard() 3 usages
        .withRegion(Regions.US_EAST_1)
        .build();
    private final String queueUrl; 4 usages
    private final Map<String, List<IEventHandler>> handlers = new HashMap<>(); 3 usages

    public SqsEventBus(String queueUrl) { 1 usage  EduardoOrtegaZerpa *
        this.queueUrl = queueUrl;
        startPolling();
    }

    @Override 6 usages  EduardoOrtegaZerpa
    public void publish(final List<DomainEvent> events) {
        for (DomainEvent event : events) {
            String messageBody = event.toJson();
            SendMessageRequest sendMsgRequest = new SendMessageRequest()
                .withQueueUrl(queueUrl)
                .withMessageBody(messageBody);
            sqsClient.sendMessage(sendMsgRequest);
        }
    }

    @Override 4 usages  EduardoOrtegaZerpa
    public void subscribe(IEventHandler handler) {
        this.handlers.putIfAbsent(handler.getEventId(), new ArrayList<>());
        this.handlers.get(handler.getEventId()).add(handler);
    }

    private void startPolling() { 1 usage new *
        Thread pollingThread = new Thread(() -> {
            while (true) {
                pollMessages();
            }
        });
        pollingThread.start();
    }
}
```

```

public class SqsEventBus implements IEventBus { 2 usages EduardoOrtegaZerpa *
    private void startPolling() { 1 usage new *
        Thread pollingThread = new Thread() -> {
            while (true) {
                pollMessages();
                try {
                    Thread.sleep(millis: 1000); // Espera antes de la siguiente lectura
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    break;
                }
            }
        });
        pollingThread.setDaemon(true);
        pollingThread.start();
    }

    private void pollMessages() { 1 usage EduardoOrtegaZerpa *
        ReceiveMessageRequest receiveMessageRequest = new ReceiveMessageRequest()
            .withQueueUrl(queueUrl)
            .withMaxNumberOfMessages(10)
            .withWaitTimeSeconds(10);

        List<Message> messages = sqsClient.receiveMessage(receiveMessageRequest).getMessages();
        for (Message message : messages) {
            try {
                System.out.println("Procesando mensaje: " + message.getBody());
                DomainEvent event = DomainEvent.fromJson(message.getBody());
                List<IEventHandler> eventHandlers = handlers.get(event.getEventId());
                if (eventHandlers != null) {
                    for (IEventHandler handler : eventHandlers) {
                        handler.handle(event);
                    }
                }
                sqsClient.deleteMessage(queueUrl, message.getReceiptHandle());
            } catch (Exception e) {
                System.err.println("Error procesando el mensaje: " + e.getMessage());
            }
        }
    }
}

```

A continuación creamos una nueva cola utilizando los servicios de AWS.

Crear una cola

Detalles

Tipo
Elige el tipo de cola para su aplicación o infraestructura en la nube.

☒ **Estándar Información**
No se conserva el orden de los mensajes donde se entrega al menos una vez

- Entrega al menos una vez
- Orden de mejor esfuerzo

☐ **FIFO Información**
Se conserva el orden de mensajes en donde el primero que en entrar, es el primero en salir

- Entrega primero en entrar/primer en salir
- Procesamiento único

☐ No puede cambiar el tipo de cola después de crear una cola.

Nombre
decoupling

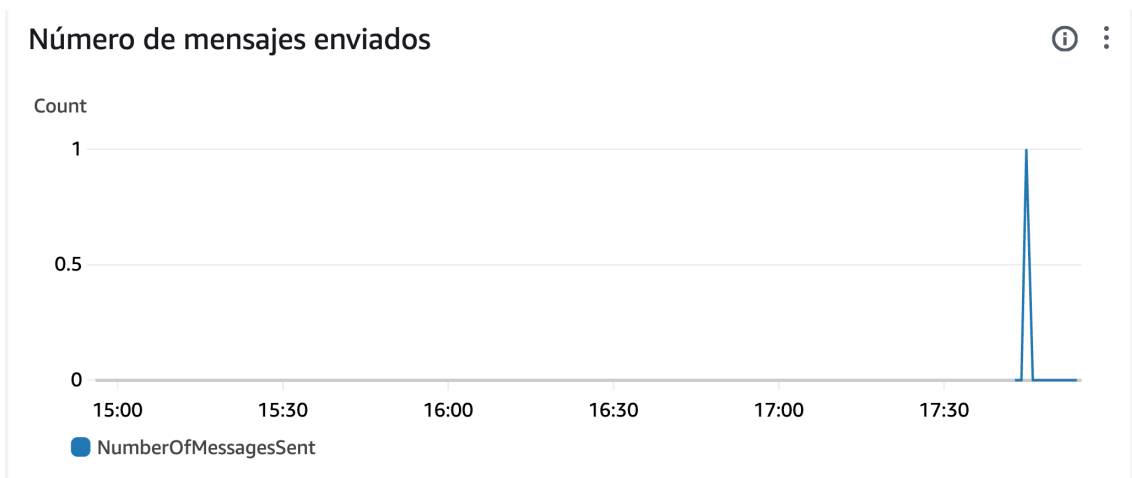
El nombre de una cola distingue entre mayúsculas y minúsculas y puede tener hasta 80 caracteres. Puede utilizar caracteres alfanuméricos, guiones (-) y guiones bajos (_).

Finalmente, con el Dependency Injector de SpringBoot inyectamos la dependencia del SQS.

```
@Configuration  @ Pepe *
public class AppConfig {

    @Bean  @ Pepe *
    public IEventBus eventBus() {
        return new SqsEventBus( queueUrl: "https://sqs.us-east-1.amazonaws.com/927928737244/decoupling");
    }
}
```

Si ejecutamos nuestro servidor podemos ver que los eventos se emiten y se reciben correctamente.



Cuando creamos un coche, manda la notificación al servicio de autoescuelas y este actualiza la lista de coches de una autoescuela.

POST localhost:8080/v1/vehicles/ Send 200 OK 486 ms 372 B 8 Minutes Ago

Params Body Auth Headers (4) Scripts Docs Preview Headers (3) Cookies Tests 0 / 0 → Mock Console

JSON

```
1 {
2   "plate": "AZF-1234",
3   "brand": "Mercedes",
4   "model": "Clase A",
5   "lastInspectionDate": "2024-05-15",
6   "nextInspectionDate": "2025-05-15",
7   "fuelType": "gasoline",
8   "drivingSchoolId": "8d9ce0a1-430d-455c-9448-09ea4719f63f"
9 }
```


Preview

```
1 {
2   "value": {
3     "id": "7d1e034a-3d6a-404e-93b3-13d2a8079af1",
4     "drivingSchool": "8d9ce0a1-430d-455c-9448-09ea4719f63f",
5     "teacher": null,
6     "plate": {
7       "plateNumber": "AZF-1234"
8     },
9     "brand": "Mercedes",
10    "model": "Clase A",
11    "itv": {
12      "lastInspectionDate": "2024-05-15",
13      "nextInspectionDate": "2025-05-15",
14      "itvexpired": false
15    },
16    "fuelType": {
17      "fuel": "GASOLINE"
18    },
19    "active": true
20  },
21  "success": true,
22  "errorMessage": null
23 }
```

GET http://localhost:8080/v1/driving_schools/ Send 200 OK 36 ms 282 B Just Now

Params Body Auth Headers (3) Scripts Docs Preview Headers (3) Cookies Tests 0 / 0 → Mock Console

No Body



Enter a URL and send to get a response

Select a body type from above to send data in the body of a request

[Introduction to Insomnia](#)

Preview

```
1 {
2   "value": [
3     {
4       "id": "8d9ce0a1-430d-455c-9448-09ea4719f63f",
5       "name": "hola",
6       "sections": [],
7       "teachers": [],
8       "vehicles": [
9         {
10          "id": "fda42846-68c4-4e3c-9274-9999ff95c87b",
11          "plate": {
12            "plateNumber": "AZF-1234"
13          },
14          "active": true
15        }
16      ],
17      "cif": {
18        "cif": "B63272603"
19      },
20      "active": true
21    }
22  ],
23  "success": true,
24  "errorMessage": null
25 }
```

Nuestro monolito ahora es capaz de ser desplegado de forma independiente en 3 grupos de servidores diferentes conectados de forma Event Driven. Con 3 modulos: autoescuelas, vehículos y profesores.