Microservicios Reactivos con Spring Boot y AWS Lambda

Marcos Raimundo Lozina

Actualizado a Java 25

MICROSERVICIOS REACTIVOS CON SPRING BOOT Y AWS LAMBDA

De APIs tradicionales a funciones serverless reactivas: cómo construir sistemas modernos y escalables en la nube con Java 25

Marcos Raimundo Lozina

Índice general

0.1 Autor

Marcos Raimundo Lozina
Desarrollador backend especializado en Java, Spring Boot y AWS.
Apasionado por la arquitectura de software, la eficiencia energética y la automatización en la nube.

0.2 Descripción general

Este e-book (**2da Edición**) ofrece una guía práctica y moderna para construir **microservicios reactivos y serverless** con **Java 25 LTS**, **Spring Boot 3.4+** y **AWS Lambda**, con un enfoque especial en **Green Software** y sostenibilidad.
A lo largo de sus capítulos, aprenderás a diseñar, implementar y optimizar aplicaciones cloud-native capaces de escalar automáticamente y responder en milisegundos, minimizando el consumo de recursos y la huella de carbono.

Cada sección incluye ejemplos de código reales, configuraciones de AWS, prácticas de CI/CD y optimización con **GraalVM Native**.

⚠ DISCLAIMER LEGAL IMPORTANTE:

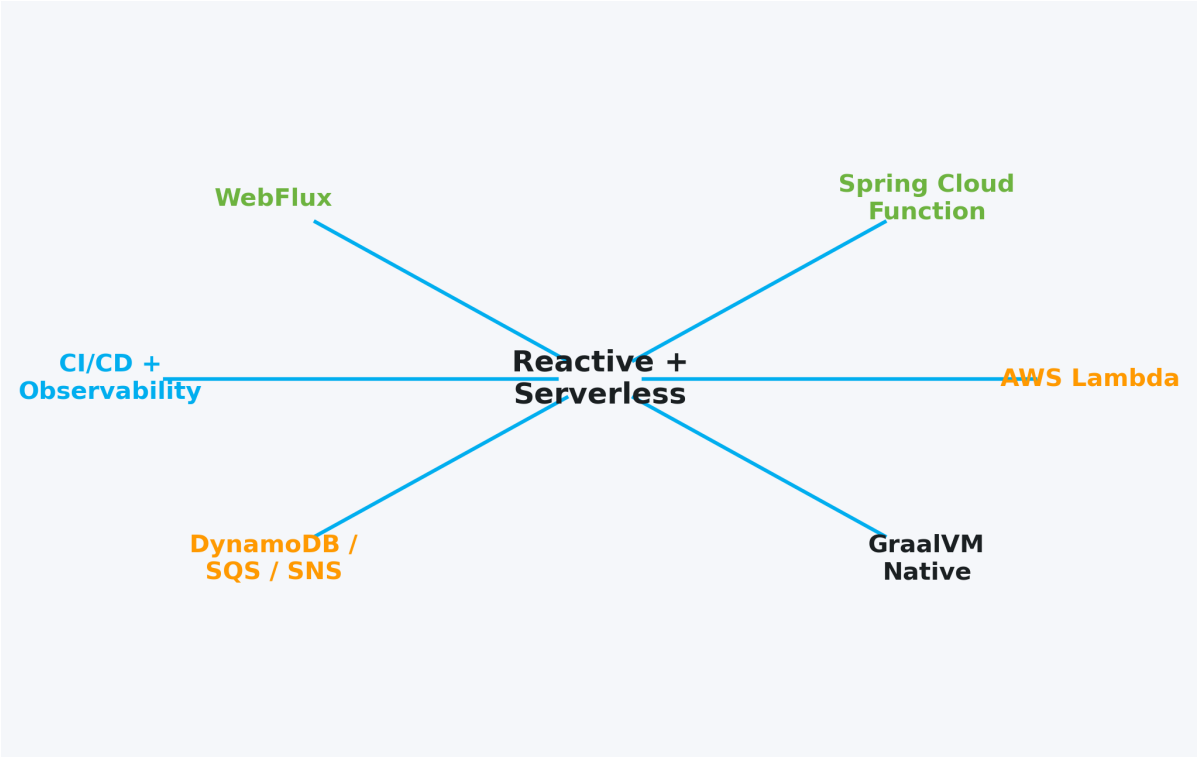


Figura 1: Mapa Mental General (Resumen del Libro)

> Este ebook contiene ejemplos que utilizan servicios de AWS que **generan costos reales**. Al seguir los ejemplos, **aceptas la responsabilidad completa** de todos los costos incurridos. El autor NO se hace responsable de facturas inesperadas.
>
> **Siempre configurá presupuestos y alertas antes de desplegar recursos.** Consultá el **Capítulo 11: Control de Costos y Disclaimers Legales** para protección completa contra facturas inesperadas.
>
> **Proyecto de referencia**: Todo el código y ejemplos de este ebook están disponibles en el proyecto de código abierto `reactive-microservices-aws-lambda-java25` (actualizado para Java 25 LTS). Podés clonar el repositorio y seguir los ejemplos mientras leés el ebook.
>
> ⚠ **Nota sobre correspondencia**: Todo el código del ebook usa el paquete `com.example.lambda`, que coincide exactamente con el proyecto real. El código mostrado en el ebook es el mismo código que encontrarás en el repositorio del proyecto.

Ideal para desarrolladores Java, arquitectos de software y equipos DevOps que buscan adoptar el modelo **reactivo y serverless** sin abandonar la robustez del ecosistema Spring.

0.3 Índice General

Nº	Sección	Descripción
1	**Introducción al paradigma reactivo y el enfoque serverless**	Comprender el cambio de paradigma hacia arquitecturas reactivas y sin servidores.
2	**Configurando el entorno moderno de desarrollo**	Instalación y preparación del ecosistema Spring Boot + AWS Lambda.
3	**Fundamentos de Spring WebFlux**	Construcción del primer microservicio completamente reactivo.
4	**De microservicio reactivo a función serverless**	Conversión de la API a una función Lambda con Spring Cloud Function.
5	**Optimización con GraalVM Native**	Compilación AOT y reducción del tiempo de arranque.
6	**Integración con servicios de AWS**	Conexión con DynamoDB, SQS y SNS para flujos event-driven.
7	**Observabilidad y despliegue continuo (CI/CD)**	Logging estructurado, métricas, trazas y automatización de despliegues.
8	**Conclusiones y próximos pasos**	Síntesis final, roadmap de crecimiento y oportunidades de expansión.
9	**Seguridad y autenticación**	IAM, Cognito, JWT, Secrets Manager y mejores prácticas de seguridad.
9	**Prólogo e historia del autor**	Contexto personal y motivación detrás del libro.
10	**Glosario y apéndice**	Referencias técnicas, términos clave y recursos adicionales.
11	**Control de Costos y Disclaimers Legales ⚠**	Protección contra facturas inesperadas, optimización de costos y disclaimers legales.

0.4 Tecnologías principales

- **Lenguaje:** Java 25 LTS (con JEP 519: Compact Object Headers y JEP 511: Module Import Declarations)
- **Framework:** Spring Boot 3.4+ (WebFlux, Cloud Function)
- **Infraestructura:** AWS Lambda, SAM, SQS, SNS, DynamoDB
- **DevOps:** GitHub Actions, CloudWatch, Micrometer, X-Ray
- **Optimización:** GraalVM Native Image, Docker, LocalStack
- **Enfoque:** Green Software Development y Sostenibilidad

0.5 Objetivo final del e-book

- Comprender a fondo la **programación reactiva en Java 25 LTS**.
- Aplicar los principios de **arquitectura serverless** en entornos reales.
- Desplegar funciones escalables con bajo costo y alta eficiencia energética.
- Dominar las herramientas AWS para automatizar y monitorear microservicios.
- Aprovechar las optimizaciones de **Green Software** de Java 25 para reducir la huella de carbono y los costos operativos.

0.6 Estructura del proyecto base

El proyecto de referencia `reactive-microservices-aws-lambda-java25` sigue esta estructura:

```
reactive-microservices-aws-lambda-java25/
 ├── lambda-core/        # Lógica principal y funciones reactivas
 ├── lambda-infra/       # Configuración SAM, CloudFormation y despliegues
 ├── lambda-tests/       # Pruebas unitarias y de integración
 └── buildSrc/           # Plugins Gradle y convenciones compartidas
```

> Ver el código completo: Todos los ejemplos de código de este ebook están implementados en el proyecto. Podés explorar los archivos específicos mencionados en cada sección en el repositorio de GitHub.

0.7 Agradecimientos

Gracias a todos los desarrolladores y colegas que inspiran el avance del ecosistema Java.
Tu contribución –ya sea con código, feedback o curiosidad– impulsa la innovación en cada línea.

> *"El código reactivo no espera: responde."* – M.R. Lozina

> **⚠ DISCLAIMER LEGAL IMPORTANTE:**
>
> Este ebook contiene ejemplos que utilizan servicios de AWS que **generan costos reales**. Al seguir los ejemplos, **aceptas la responsabilidad completa** de todos los costos incurridos. El autor NO se hace responsable de facturas inesperadas.
>
> **Siempre configurá presupuestos y alertas antes de desplegar recursos.** Consultá el **Capítulo 11: Control de Costos y Disclaimers Legales** para protección completa.

0.8 Objetivo

Comprender por qué las arquitecturas reactivas y serverless se convirtieron en el estándar moderno para construir microservicios escalables y eficientes.

0.9 El cambio de paradigma en la arquitectura moderna

Durante la última década, los microservicios han reemplazado los sistemas monolíticos tradicionales debido a su flexibilidad, escalabilidad y capacidad de despliegue independiente. Sin embargo, cuando empecé a trabajar con microservicios tradicionales basados en modelos bloqueantes (como los que usan `Spring MVC`), me encontré con limitaciones significativas al manejar grandes volúmenes de peticiones simultáneas.

En la práctica, esto se ve mucho: tu aplicación funciona perfectamente con 100 usuarios concurrentes, pero cuando llega un pico de tráfico, empieza a rechazar solicitudes o la latencia se dispara. La **programación reactiva** surge como una evolución natural para resolver exactamente este problema: permite procesar flujos de datos de manera no bloqueante, gestionando eficientemente los recursos del sistema mediante el patrón **Publisher-Subscriber**. En lugar de esperar respuestas secuenciales, el modelo reactivo reacciona a los eventos conforme ocurren, maximizando la utilización del CPU y la memoria.

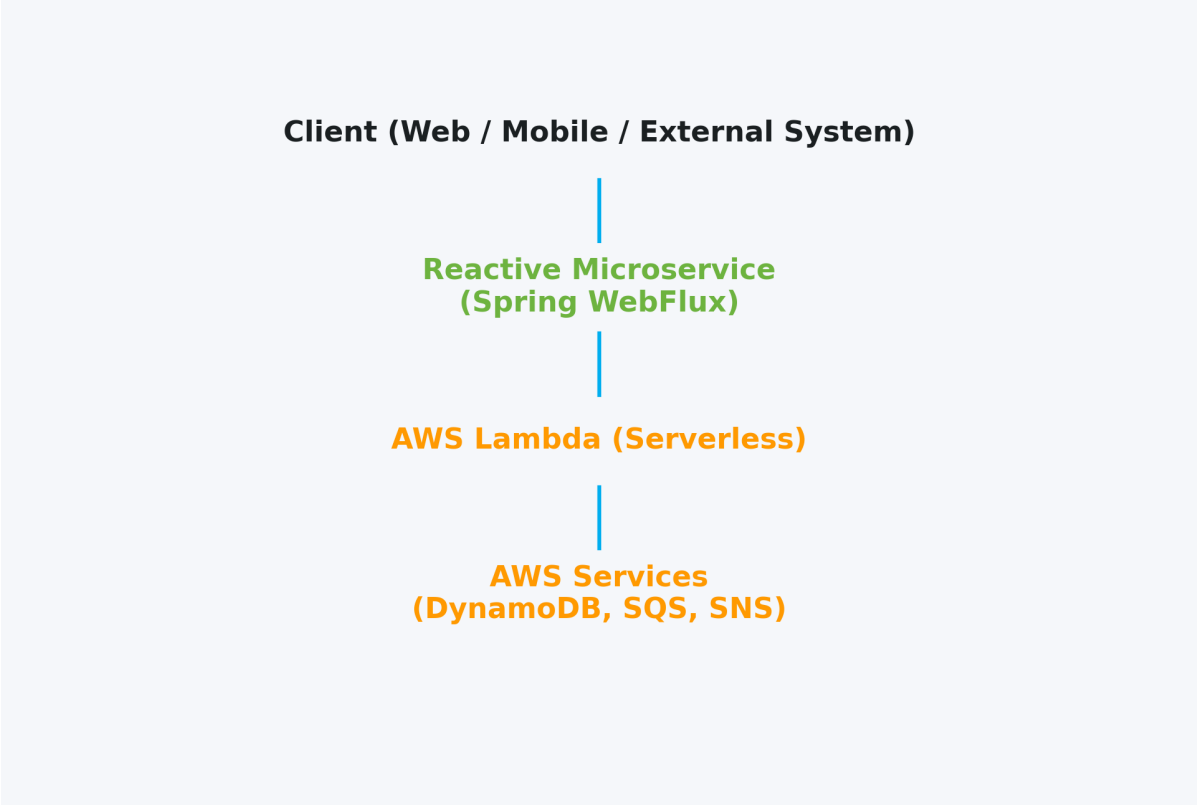


Figura 2: Arquitectura Reactiva y Serverless

0.10 ¿Qué es la programación reactiva?

El paradigma reactivo se basa en cuatro principios fundamentales, definidos por el Reactive Manifesto:

1. **Responsive (Responsivo)** - el sistema debe responder en tiempo y forma.
2. **Resilient (Resiliente)** - debe mantenerse operativo ante fallos parciales.
3. **Elastic (Elástico)** - debe adaptarse dinámicamente a la carga.

4. **Message Driven (Basado en mensajes)** - debe comunicarse mediante mensajes asíncronos.

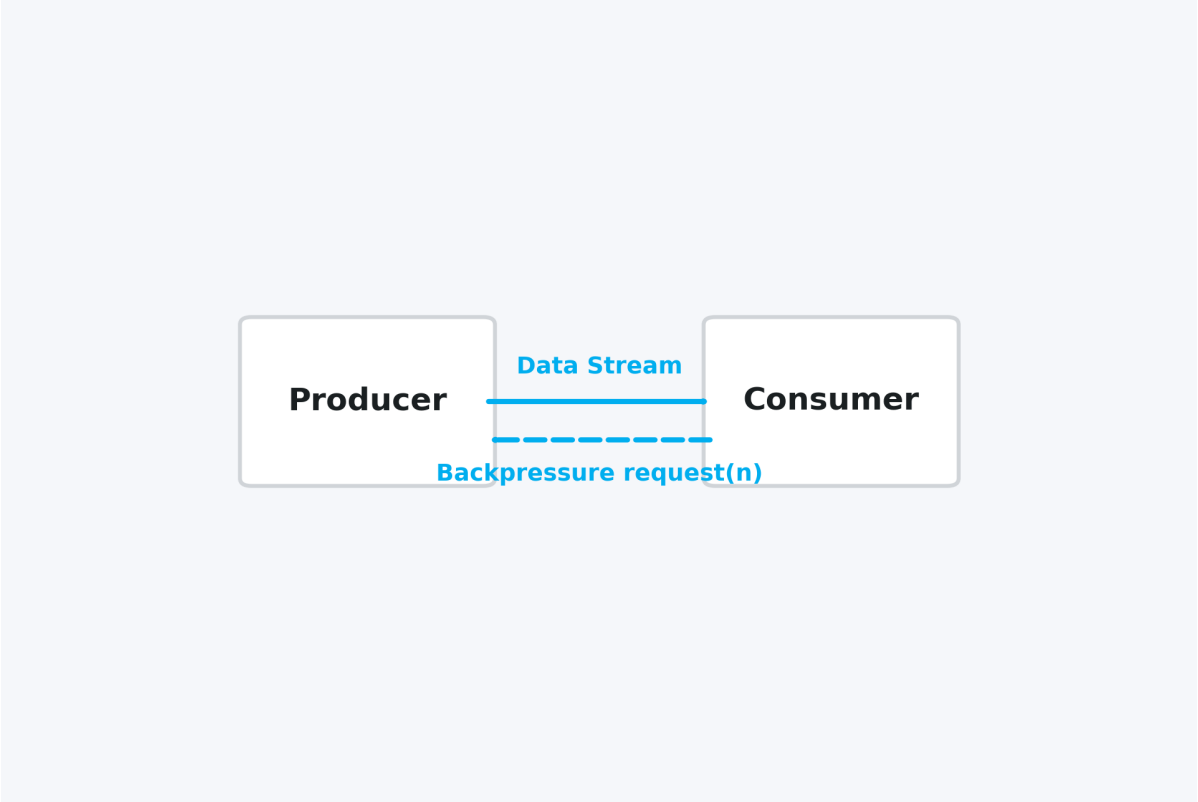


Figura 3: Paradigma Reactivo

En Java, este paradigma se materializa con bibliotecas como **Project Reactor**, **RxJava** y **Mutiny**, siendo *Reactor* la base del stack reactivo de Spring Boot (`spring-webflux`). En este ebook nos enfocaremos en **Project Reactor** porque es el estándar de facto en el ecosistema Spring y es el que vas a encontrar en la mayoría de proyectos reales.

0.11 De APIs bloqueantes a flujos asíncronos

Ahora que entendimos los principios fundamentales, veamos cómo se traducen en la práctica. La diferencia entre un modelo bloqueante y uno reactivo no es solo teórica: tiene implicaciones directas en el rendimiento y la escalabilidad de tu aplicación.

En una aplicación tradicional con `Spring MVC`, cada petición HTTP ocupa un hilo del **thread pool**. Esto te va a pasar cuando estés construyendo algo real: si hay más peticiones que hilos disponibles, el servidor empieza a rechazar solicitudes o aumenta la latencia significativamente. En producción, esto se traduce en timeouts y errores 503 que afectan directamente la experiencia del usuario.

En cambio, **Spring WebFlux** usa un modelo basado en **event loop**, similar a Node.js. Un solo hilo puede manejar miles de conexiones simultáneamente, ya que las operaciones de I/O se realizan de forma no bloqueante. Esto permite escalar horizontalmente sin incrementar recursos de hardware, y es exactamente lo que necesitás cuando tu aplicación necesita manejar picos de tráfico impredecibles.

0.11.1 Backpressure: control de flujo reactivo

Un concepto fundamental en la programación reactiva es el **backpressure** (contrapresión). Cuando un productor emite datos más rápido de lo que el consumidor puede procesar, el backpressure permite que el consumidor controle la velocidad de emisión, evitando desbordamientos de memoria y garantizando un flujo controlado de datos.

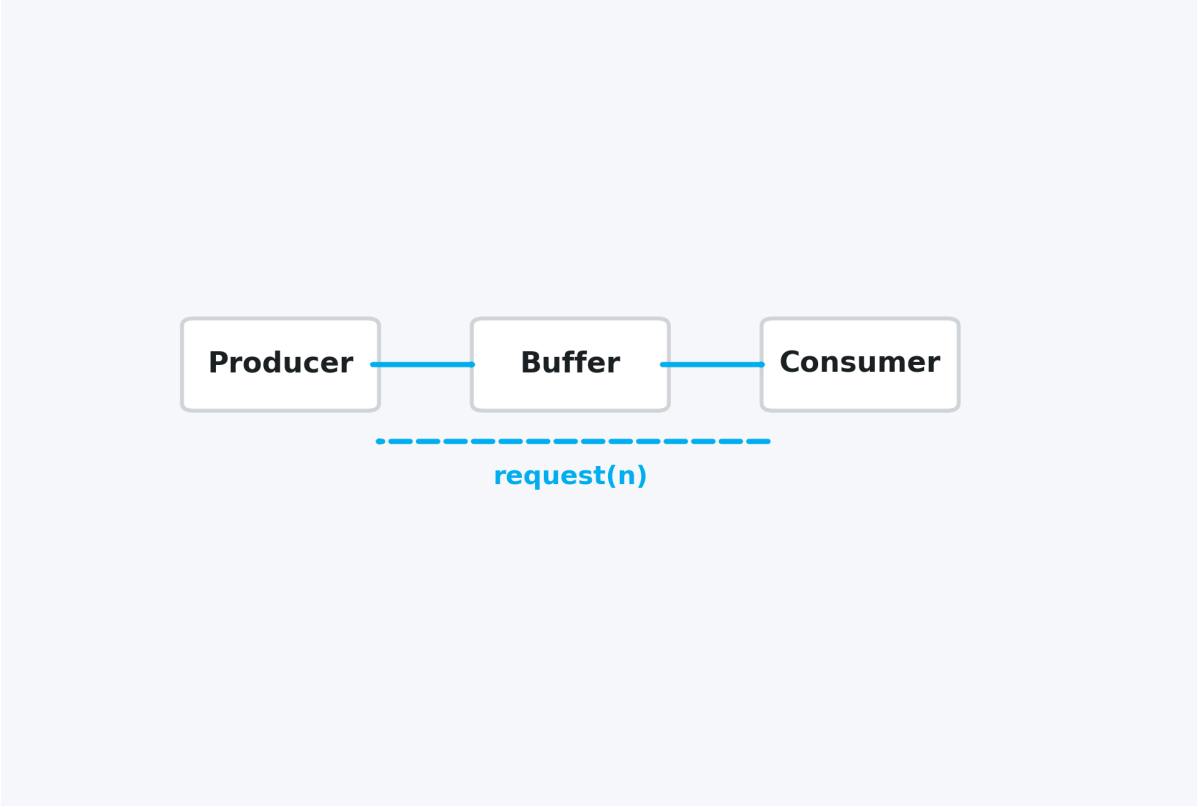


Figura 4: Backpressure (Control de Flujo)

En **Project Reactor**, el backpressure se maneja automáticamente mediante el protocolo **Reactive Streams**, donde el consumidor solicita (`request`) una cantidad específica de elementos al productor. Esto es especialmente importante en flujos de datos continuos o cuando se procesan grandes volúmenes de información. En la práctica, esto evita dolores de cabeza en producción cuando tenés que procesar streams de datos que pueden venir más rápido de lo que podés procesar.

Veamos la diferencia práctica entre un enfoque bloqueante y uno reactivo:

```
@GetMapping("/clientes")
public List<Cliente> listar() {
    return clienteService.listar(); // Bloqueante
}
```

versus

```
@GetMapping("/clientes")
public Flux<Cliente> listar() {
    return clienteService.listar(); // No bloqueante
}
```

La diferencia parece sutil, pero en producción es enorme: el primer enfoque bloquea un hilo

hasta que toda la lista esté lista, mientras que el segundo empieza a emitir datos tan pronto como están disponibles.

0.12 El enfoque serverless

Ahora que tenemos claro cómo funciona la programación reactiva, veamos cómo se combina con el modelo serverless. Esto nos lleva naturalmente al siguiente punto: si ya optimizamos el código para ser no bloqueante, ¿por qué no optimizar también la infraestructura?

Serverless no significa "sin servidores", sino que la infraestructura subyacente está completamente gestionada por el proveedor cloud (en este caso, AWS). El desarrollador se enfoca exclusivamente en la lógica de negocio, mientras que el proveedor escala, monitorea y factura según el uso real. Esto evita dolores de cabeza en producción relacionados con el aprovisionamiento de servidores, el balanceo de carga y el mantenimiento de la infraestructura.

En **AWS Lambda**, cada función se ejecuta bajo demanda, con costos basados en tiempo de ejecución y memoria utilizada. Esto hace que sea ideal para sistemas **reactivos y event-driven**, donde las cargas son impredecibles o variables. En la práctica, esto se traduce en ahorros significativos cuando tu aplicación tiene períodos de baja actividad, porque solo pagás por lo que realmente usás.

0.12.1 Ventajas principales del modelo serverless:

- **Cero mantenimiento de infraestructura**: AWS gestiona automáticamente el aprovisionamiento, escalado y mantenimiento de servidores.
- **Escalado automático**: Las funciones escalan desde cero hasta miles de invocaciones concurrentes sin configuración adicional.
- **Pago por uso**: Solo pagas por el tiempo de ejecución real, no por capacidad ociosa.
- **Integración nativa**: Conexión directa con servicios AWS como S3, SQS, SNS, DynamoDB, API Gateway, etc.

0.13 Por qué Java 25 LTS + Spring Boot 3.4+ + GraalVM Native cambian el panorama

Históricamente, Java no era ideal para AWS Lambda debido a sus largos **cold starts** (tiempos de arranque inicial). Esto te va a pasar cuando estés construyendo algo real: si tu función Lambda tarda 5 segundos en arrancar, cada invocación en frío va a tener una latencia inaceptable. Sin embargo, con **Spring Boot 3.4+**, el soporte nativo a **GraalVM Native Image** y la nueva generación de **Spring AOT (Ahead-Of-Time Compilation)**, ahora es posible crear binarios ligeros y rápidos que compiten directamente con Node.js o Python.

> **⚠ Nota importante sobre compatibilidad de versiones:**
>
> **Spring Boot 3.4+ requiere Spring Cloud 2024.0.0 o superior.** Las versiones anteriores de Spring Cloud (como 2023.0.1) no son completamente compatibles con Spring Boot 3.4+ y pueden causar problemas de compatibilidad. Por esta razón, en este ebook usamos: - **Spring Boot 3.4.0+** (mínimo recomendado: 3.4.0) - **Spring Cloud 2024.0.0** (compatible con Spring Boot 3.4+) - **Gradle 8.5+ o 9.x** (recomendado: Gradle 9.2.1 para mejor soporte de Java 25)

> **En la práctica:** Esta incompatibilidad entre Spring Cloud 2023.x y Spring Boot 3.4+ es una de las razones por las que migramos a las versiones más recientes. Si intentás usar Spring Cloud 2023.0.1 con Spring Boot 3.4+, podés encontrar errores de inicialización o problemas con Spring Cloud Function.

Antes de avanzar, entendamos por qué esto es importante: los cold starts son uno de los principales problemas que vas a enfrentar en producción con Lambda, y GraalVM Native Image los reduce drásticamente.

0.13.1 Comparación de rendimiento: JVM tradicional vs GraalVM Native

Runtime	Tiempo de arranque (cold start)	Memoria aproximada	Tamaño del artefacto
Java 8 (JVM tradicional)	2.5 - 5 seg	512 MB	~50 MB
Java 21 + GraalVM Native	100 - 300 ms	128 MB	~12 MB
Java 25 LTS + GraalVM Native + Compact Headers	**80 - 250 ms**	**96 - 112 MB**	**10 - 11 MB**

Esto habilita despliegues **reactivos y serverless** con una eficiencia comparable a Node.js o Python, manteniendo la robustez, tipado fuerte y ecosistema maduro de Java. Además, Java 25 LTS introduce optimizaciones revolucionarias que reducen aún más el consumo de recursos, como veremos a continuación.

0.14 Java 25 LTS: La versión más Eco-Friendly de la historia

Java 25 LTS representa un hito fundamental en la evolución del lenguaje hacia la **sostenibilidad y eficiencia energética**. Esta versión introduce mejoras revolucionarias que transforman cómo las aplicaciones Java consumen recursos, especialmente crítico en entornos serverless como AWS Lambda donde cada megabyte de memoria y cada milisegundo de ejecución se traducen directamente en costos operativos y huella de carbono.

La característica estrella de Java 25 es **JEP 519: Compact Object Headers**, que reduce automáticamente el tamaño del encabezado de objetos de 96 bits a 64 bits en plataformas de 64 bits. Esta optimización, aparentemente pequeña, tiene un impacto masivo en aplicaciones que crean millones de objetos: reduce la huella de memoria del heap entre un 15% y 25%, lo que se traduce en menor consumo energético, menor facturación en la nube y mayor densidad de despliegue.

Pero Java 25 no solo optimiza la memoria: también introduce **JEP 511: Module Import Declarations**, que simplifica significativamente el código y mejora la legibilidad, reduciendo la complejidad cognitiva para los desarrolladores. Menos complejidad significa menos tiempo de desarrollo,

menos bugs y, en última instancia, menos recursos computacionales desperdiciados en debugging y mantenimiento.

Cuando combinamos estas mejoras con las optimizaciones de **GraalVM Native Image** y las mejoras en **Leyden**, obtenemos aplicaciones que no solo son más rápidas y eficientes, sino que también contribuyen activamente a la reducción de la huella de carbono del sector tecnológico. En un mundo donde la sostenibilidad se ha convertido en una prioridad empresarial, Java 25 posiciona a la plataforma como líder en **Green Software Development**.

0.15 Conclusión de la sección

El futuro de los sistemas distribuidos apunta hacia arquitecturas **reactivas, asíncronas y serverless**, donde la eficiencia y la escalabilidad no dependen de la cantidad de hilos, sino del flujo continuo de datos y eventos. **Java 25 LTS**, combinado con **Spring Boot 3.4+** y **AWS Lambda**, ofrece una plataforma madura y robusta para alcanzar este objetivo con estabilidad empresarial, mientras que las optimizaciones de **Green Software** reducen significativamente la huella de carbono y los costos operativos.

En la siguiente sección configuraremos el entorno moderno de desarrollo con Java 25 LTS y crearemos nuestro primer microservicio reactivo base utilizando Spring WebFlux y Project Reactor.

1 Configurando el entorno moderno de desarrollo

> **⚠ ADVERTENCIA SOBRE COSTOS:**
>
> Antes de comenzar, configurá presupuestos y alertas de AWS para protegerte de facturas inesperadas. Consultá el **Capítulo 11: Control de Costos y Disclaimers Legales** para instrucciones detalladas.

1.1 Objetivo

Preparar el entorno de trabajo para desarrollar y desplegar microservicios reactivos con Spring Boot 3.4+, Java 25 LTS y AWS Lambda. Esta sección garantiza que todos los componentes del ecosistema estén correctamente instalados, configurados y funcionando antes de comenzar a codificar.

> **En la práctica**: Tomate el tiempo necesario para configurar bien el entorno desde el principio. Esto evita dolores de cabeza más adelante cuando estés intentando desplegar y te encuentres con problemas de compatibilidad de versiones o dependencias faltantes.

1.2 Requisitos previos

> **⚠ Nota de Actualización - 2da Edición (Java 25 LTS):**

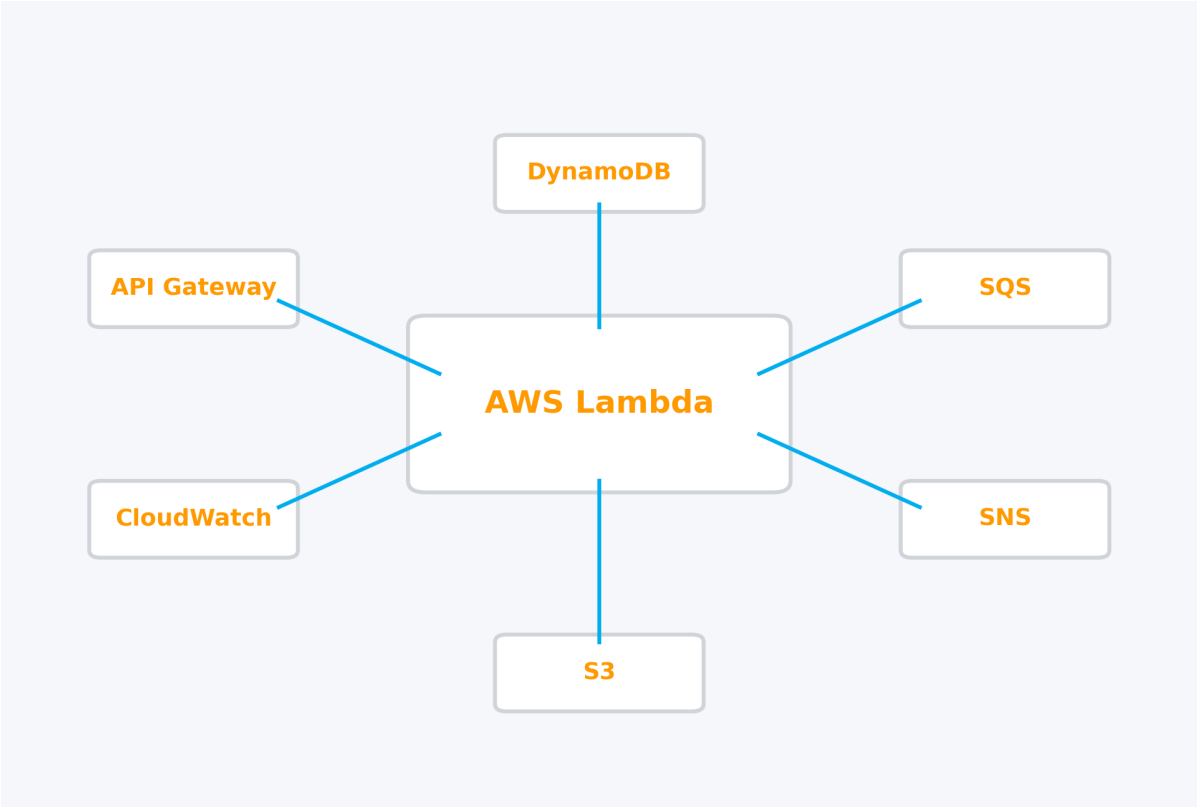


Figura 5: Arquitectura Serverless

Esta segunda edición del ebook ha sido completamente actualizada para aprovechar las características más avanzadas de **Java 25 LTS**, incluyendo **JEP 519 (Compact Object Headers)** y **JEP 511 (Module Import Declarations)**. Para seguir todos los ejemplos y ejercicios de este libro, es **esencial** que instales el **JDK 25** en tu sistema.

¿Por qué Java 25? Java 25 LTS introduce mejoras revolucionarias en eficiencia de memoria y sostenibilidad que son fundamentales para los conceptos de Green Coding que exploraremos a lo largo del ebook. Las optimizaciones de Compact Object Headers pueden reducir el consumo de memoria hasta en un 25%, lo cual es crítico en entornos serverless como AWS Lambda.

Instalación recomendada: - **OpenJDK 25**: Disponible en adoptium.net - **Amazon Corretto 25**: Disponible en aws.amazon.com/corretto - **SDKMAN** (recomendado): `sdk install java 25.0.0-tem`

Verificá la instalación con:

```
java -version
```

Deberías ver: `openjdk version "25"` o similar.

Antes de empezar, asegurate de tener instalado lo siguiente. En la práctica, esto se ve mucho: desarrolladores que intentan seguir el tutorial pero se encuentran con errores porque tienen versiones incompatibles o herramientas faltantes. Verificá cada componente antes de continuar.

- **Java 25 LTS** (OpenJDK o Amazon Corretto).
  Verificá la versión con:

```
java -version
```

- **Gradle 8.5+** o **9.x** (recomendado: **Gradle 9.2.1** para mejor soporte de Java 25).

  ```
  gradle -v
  ```

  > **Nota sobre Gradle:** Gradle 9.2.1 incluye mejoras específicas para Java 25 y es la versión recomendada. Gradle 8.5+ funciona, pero puede tener limitaciones menores con algunas características de Java 25.

- **Docker** y **Docker Compose**, para pruebas locales y simulación de entornos.

- **AWS CLI** configurada con tus credenciales:

  ```
  aws configure
  ```

- **AWS SAM CLI** (Serverless Application Model), necesaria para empaquetar y ejecutar funciones Lambda localmente.

  ```
  sam --version
  ```

- **IntelliJ IDEA** (versión Community o Ultimate), con plugins de Spring y AWS Toolkit.

1.3 Creación del proyecto base

Ahora que tenemos el entorno configurado, veamos cómo crear el proyecto base. Usaremos Spring Initializr para generar la estructura inicial, lo cual evita tener que configurar manualmente todas las dependencias y la estructura de directorios. Podés hacerlo desde la línea de comando o desde IntelliJ IDEA.

> **Proyecto de referencia**: Si preferís empezar con un proyecto ya configurado, podés clonar el repositorio `reactive-microservices-aws-lambda-java25` que contiene toda la estructura y configuración lista para usar con Java 25 LTS. El proyecto incluye todos los ejemplos de código que verás en este ebook.

1.3.1 Estructura del proyecto

Creamos el proyecto con:

```
spring init   --boot-version=3.4.0   --java-version=25   --dependencies=webflux,cloud-function,
```

Esto generará una estructura inicial como:

```
reactive-lambda/
├── build.gradle.kts
├── settings.gradle.kts
├── src/
│   ├── main/
│   │   ├── java/com/example/reactive/
│   │   ├── resources/
│   └── test/
└── README.md
```

1.4 Configuración del build.gradle.kts

Ahora que tenemos la estructura del proyecto, necesitamos configurar las dependencias y las configuraciones básicas para empaquetar la aplicación en formato Lambda. Esto es crucial porque sin la configuración correcta, tu función no va a poder ejecutarse en AWS Lambda.

```
plugins {
    id("org.springframework.boot") version "3.4.0"
    id("io.spring.dependency-management") version "1.1.6"
    id("org.graalvm.buildtools.native") version "0.11.0"
    kotlin("jvm") version "2.0.0"
    kotlin("plugin.spring") version "2.0.0"
}

// Nota: Las versiones utilizadas son las más recientes y estables al momento de escribir este
// Spring Boot 3.4.0 incluye soporte completo para Java 25 LTS y GraalVM Native Image.

java {
    sourceCompatibility = JavaVersion.VERSION_25
    targetCompatibility = JavaVersion.VERSION_25
    toolchain {
        languageVersion = JavaLanguageVersion.of(25)
    }
}

repositories {
    mavenCentral()
}

// Habilitar preview features para JEP 519 (si es necesario)
tasks.withType<JavaCompile> {
    options.compilerArgs.addAll(listOf(
        "--enable-preview",
        "-Xlint:preview"
    ))
    options.release.set(25)
}

tasks.withType<Test> {
    jvmArgs("--enable-preview")
    useJUnitPlatform()
}

// Usar BOM (Bill of Materials) para gestionar versiones de manera centralizada
// Esto garantiza compatibilidad entre todas las dependencias de Spring Cloud y AWS SDK
dependencyManagement {
    imports {
        // ▫ IMPORTANTE: Spring Cloud 2024.0.0 es requerido para Spring Boot 3.4+
        // Spring Cloud 2023.0.1 y versiones anteriores NO son compatibles con Spring Boot 3.4+
        mavenBom("org.springframework.cloud:spring-cloud-dependencies:2024.0.0")
```

```
        mavenBom("software.amazon.awssdk:bom:2.26.0") // AWS SDK v2 - versión más reciente
    }
}

dependencies {
    // Spring Boot
    implementation("org.springframework.boot:spring-boot-starter-webflux")
    implementation("org.springframework.boot:spring-boot-starter-validation")

    // Spring Cloud Function
    implementation("org.springframework.cloud:spring-cloud-function-context")
    implementation("org.springframework.cloud:spring-cloud-function-adapter-aws")

    // AWS Lambda Java Events (solo necesarios para tipos de eventos)
    // Nota: Spring Cloud Function maneja la invocación, solo necesitamos los tipos de eventos
    implementation("com.amazonaws:aws-lambda-java-events:3.16.0")

    // AWS SDK v2 (versiones gestionadas por BOM 2.26.0)
    // Nota: AWS SDK v2 es completamente asíncrono y compatible con Project Reactor
    implementation("software.amazon.awssdk:dynamodb")
    implementation("software.amazon.awssdk:sqs")
    implementation("software.amazon.awssdk:sns")
    implementation("software.amazon.awssdk:secretsmanager")
    implementation("software.amazon.awssdk:url-connection-client") // Cliente HTTP sin Netty pa

    // Testing
    testImplementation("org.springframework.boot:spring-boot-starter-test")
    testImplementation("io.projectreactor:reactor-test")

    // ▫ IMPORTANTE: ASM 9.8 requerido para Java 25
    // Spring Boot necesita ASM 9.8 para leer archivos de clase Java 25 (class file version 69)
    // Sin ASM 9.8, podés encontrar errores como "Unsupported class file major version 69"
    implementation("org.ow2.asm:asm:9.8")
    implementation("org.ow2.asm:asm-commons:9.8")
    implementation("org.ow2.asm:asm-tree:9.8")
    implementation("org.ow2.asm:asm-analysis:9.8")

    // JWT Validation (para API Gateway Authorizers - ver Capítulo 9)
    implementation("com.auth0:java-jwt:4.4.0")
}

// Forzar ASM 9.8 en todas las configuraciones para soporte de Java 25
configurations.all {
    resolutionStrategy {
        force("org.ow2.asm:asm:9.8")
        force("org.ow2.asm:asm-commons:9.8")
        force("org.ow2.asm:asm-tree:9.8")
        force("org.ow2.asm:asm-analysis:9.8")
    }
```

```
}

// Configuración GraalVM Native con Compact Object Headers
graalvmNative {
    binaries {
        main {
            buildArgs.addAll(listOf(
                "--gc=epsilon",
                "--static",
                "-H:+ReportExceptionStackTraces",
                "-H:IncludeResources=.*\\.(json|yml|yaml|properties)$",
                "-XX:+UseCompactObjectHeaders" // JEP 519: Compact Object Headers
            ))
        }
    }
}

// ▫ IMPORTANTE: Deshabilitar AOT para compatibilidad con Spring Cloud Function
// Spring Cloud Function no es completamente compatible con el procesamiento AOT.
// Esta configuración es necesaria cuando uses Spring Cloud Function en tu proyecto.
tasks.named("processAot") {
    enabled = false
}
tasks.named("compileAotJava") {
    enabled = false
}
tasks.named("processAotResources") {
    enabled = false
}
tasks.named("aotClasses") {
    enabled = false
}

// Configuración de bootRun con Compact Object Headers
tasks.named<org.springframework.boot.gradle.tasks.run.BootRun>("bootRun") {
    jvmArgs = listOf(
        "-XX:+UseCompactObjectHeaders",
        "-XX:+UnlockExperimentalVMOptions",
        "-Xmx512m",
        "-Xms128m"
    )
    environment("JAVA_TOOL_OPTIONS", "-XX:+UseCompactObjectHeaders")
}
```

⚠ Nota sobre AOT y Spring Cloud Function:

Si planeás usar **Spring Cloud Function** (necesario para AWS Lambda), debés deshabilitar las tareas AOT como se muestra arriba. Spring Cloud Function no es completamente compatible con el procesamiento AOT de Spring Boot 3.4+. Esto **no afecta** la funcionalidad de la aplicación ni la capacidad de usar GraalVM Native Image para

compilación nativa.

⚠ Nota sobre compatibilidad de versiones:

Spring Boot 3.4+ requiere Spring Cloud 2024.0.0 o superior. Si intentás usar Spring Cloud 2023.0.1 (o versiones anteriores) con Spring Boot 3.4+, podés encontrar: - Errores de inicialización de beans - Problemas con Spring Cloud Function - Incompatibilidades en el procesamiento AOT

Versiones recomendadas para este ebook: - **Spring Boot:** 3.4.0+ (mínimo: 3.4.0, recomendado: 3.4.13) - **Spring Cloud:** 2024.0.0 (requerido para Spring Boot 3.4+) - **Gradle:** 9.2.1 (recomendado para Java 25) o 8.5+ (mínimo) - **Kotlin JVM Target:** 24 (Kotlin aún no soporta completamente Java 25, pero JVM 24 es compatible) - **ASM:** 9.8 (requerido para leer archivos de clase Java 25)

Para generar el archivo *fat jar* ejecutable:

```
./gradlew clean bootJar
```

1.5 Configuración del entorno AWS local con LocalStack

Para evitar costos iniciales en AWS, usaremos **LocalStack** para emular los servicios.

Creamos un archivo `docker-compose.yml`:

```
version: '3.8'
services:
  localstack:
    image: localstack/localstack
    ports:
      - "4566:4566"
      - "4571:4571"
    environment:
      - SERVICES=lambda,s3,sqs,sns,dynamodb
      - DEBUG=1
      - DATA_DIR=/tmp/localstack/data
    volumes:
      - "./localstack:/tmp/localstack"
      - "/var/run/docker.sock:/var/run/docker.sock"
```

Iniciamos el entorno local:

```
docker-compose up -d
```

Verificamos que LocalStack está corriendo:

```
aws --endpoint-url=http://localhost:4566 lambda list-functions
```

1.6 Creación de la primera función Lambda

Dentro del proyecto, creamos un paquete `com.example.lambda` y una clase simple:

```java
package com.example.lambda;

import org.springframework.stereotype.Component;
import java.util.function.Function;

@Component
public class HelloLambda implements Function<String, String> {
    @Override
    public String apply(String input) {
        return "Hola desde AWS Lambda, mensaje recibido: " + input;
    }
}
```

Compilamos el proyecto:

```
./gradlew clean build
```

Luego ejecutamos localmente la función con SAM:

```
sam local invoke "HelloLambda" -e event.json
```

Ejemplo de `event.json`:

```
{ "input": "Hola Mundo" }
```

1.7 Estructura multi-módulo (Estructura Real del Proyecto)

El proyecto de referencia `reactive-microservices-aws-lambda-java25` utiliza una estructura multi-módulo para separar responsabilidades. Podés ver la implementación completa en el repositorio:

```
reactive-lambda/
├── lambda-core/                    # Lógica de negocio y funciones Lambda
│   ├── src/
│   │   ├── main/
│   │   │   ├── java/
│   │   │   │   └── com/example/lambda/
│   │   │   │       ├── FunctionConfig.java       # Configuración de funciones
│   │   │   │       ├── handlers/
│   │   │   │       │   └── HelloHandler.java     # Handlers reactivos
│   │   │   │       └── exception/
│   │   │   │           └── ErrorHandler.java
│   │   │   └── resources/
│   │   │       └── META-INF/
│   │   │           └── native-image/
│   │   │               └── reflect-config.json
│   │   └── test/
│   │       └── java/
│   └── build.gradle.kts
├── lambda-infra/                   # Configuración AWS, SAM templates
```

```
│   ├── template.yaml              # Template SAM para despliegue
│   └── samconfig.toml
├── lambda-tests/                  # Pruebas de integración
│   └── src/
│       └── test/
│           └── java/
├── buildSrc/                      # Plugins Gradle y convenciones compartidas
│   └── src/
│       └── main/
│           └── kotlin/
│               └── conventions.gradle.kts
├── settings.gradle.kts            # Configuración de módulos
└── build.gradle.kts               # Build raíz con BOM
```

Ventajas de esta estructura: - [OK] Aislamiento claro entre capas (core, infra, tests) - [OK] Mejor mantenimiento y testeo - [OK] Reutilización en futuras funciones Lambda - [OK] Separación de configuración de infraestructura - [OK] Convenciones compartidas en `buildSrc`

settings.gradle.kts:

```
rootProject.name = "microservicios-reactivos-springboot-aws-lambda"

include("lambda-core")
include("lambda-infra")
include("lambda-tests")
```

build.gradle.kts (raíz):

```
plugins {
    java
    id("io.spring.dependency-management") version "1.1.5"
    id("org.springframework.boot") version "3.3.1" apply false
    id("org.graalvm.buildtools.native") version "0.10.3" apply false
}

subprojects {
    group = "com.example"
    version = "1.0.0"

    repositories {
        mavenCentral()
    }

    apply(plugin = "java")
    apply(plugin = "io.spring.dependency-management")
    apply(plugin = "conventions")
}

dependencyManagement {
    imports {
        mavenBom("org.springframework.boot:spring-boot-dependencies:3.3.1")
```

```
        mavenBom("org.springframework.cloud:spring-cloud-dependencies:2023.0.0")
    }
}

tasks.withType<Test> {
    useJUnitPlatform()
}
```

1.8 Verificación final del entorno

Para confirmar que todo está listo: 1. `java -version` → Java 25 LTS detectado.
2. `./gradlew build` → compila sin errores.
3. `sam local invoke` → función ejecuta correctamente.
4. `docker ps` → LocalStack activo.

Si todos estos pasos funcionan, el entorno está correctamente configurado y listo para crear nuestro **microservicio reactivo real con Spring WebFlux** en la próxima sección.

1.9 Configuración de Variables de Entorno y Profiles

Para desarrollar en diferentes entornos (local, desarrollo, producción), configuramos **Spring Profiles**.

1.9.1 Archivo `application.yml` base

```yaml
spring:
  application:
    name: reactive-lambda
  profiles:
    active: ${SPRING_PROFILES_ACTIVE:dev}

aws:
  region: ${AWS_REGION:us-east-1}
```

1.9.2 Archivo `application-dev.yml` (LocalStack)

```yaml
aws:
  dynamodb:
    endpoint: http://localhost:4566
  sqs:
    endpoint: http://localhost:4566
  sns:
    endpoint: http://localhost:4566
```

1.9.3 Archivo `application-prod.yml` (AWS Real)

```yaml
aws:
  dynamodb:
    endpoint: null  # Usa AWS real
  sqs:
    endpoint: null
  sns:
    endpoint: null
```

1.9.4 Variables de entorno en build.gradle.kts

```
tasks.named("bootRun") {
    environment("SPRING_PROFILES_ACTIVE", "dev")
    environment("AWS_REGION", "us-east-1")
}
```

1.10 Conclusión de la sección

Tu entorno ya está preparado con todos los componentes necesarios para desarrollar, probar y desplegar microservicios reactivos en AWS Lambda.
A partir de la siguiente sección, construiremos el **primer servicio WebFlux** y analizaremos cómo se comporta bajo diferentes cargas.

2 Fundamentos de Spring WebFlux

2.1 Objetivo

Comprender cómo funciona el modelo de programación reactiva dentro de Spring Boot 3.4+ y construir el primer microservicio completamente reactivo utilizando **Spring WebFlux** y **Project Reactor**.

2.2 ¿Qué es Spring WebFlux?

Spring WebFlux es el módulo de Spring diseñado para construir aplicaciones **no bloqueantes** basadas en el estándar **Reactive Streams**.
A diferencia de `spring-webmvc` (bloqueante y basado en hilos), WebFlux utiliza un **event loop** y **back-pressure** para manejar miles de solicitudes concurrentes con un número limitado de hilos.

Características principales: - Basado en **Project Reactor** (implementación oficial de Reactive Streams en Spring).
- Permite el uso de controladores anotados (`@RestController`) o enrutamiento funcional (`RouterFunction`).
- Escalable, eficiente y preparado para entornos **cloud-native**.

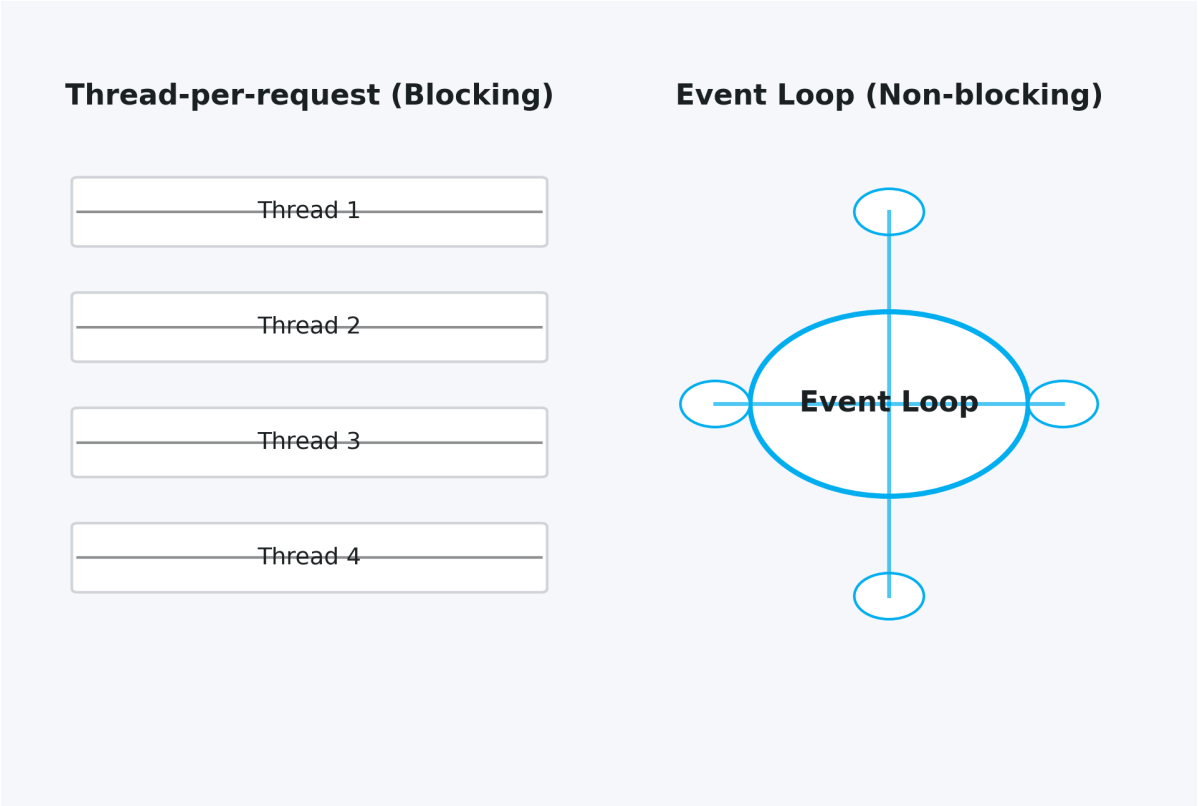


Figura 6: Flujo de Spring WebFlux

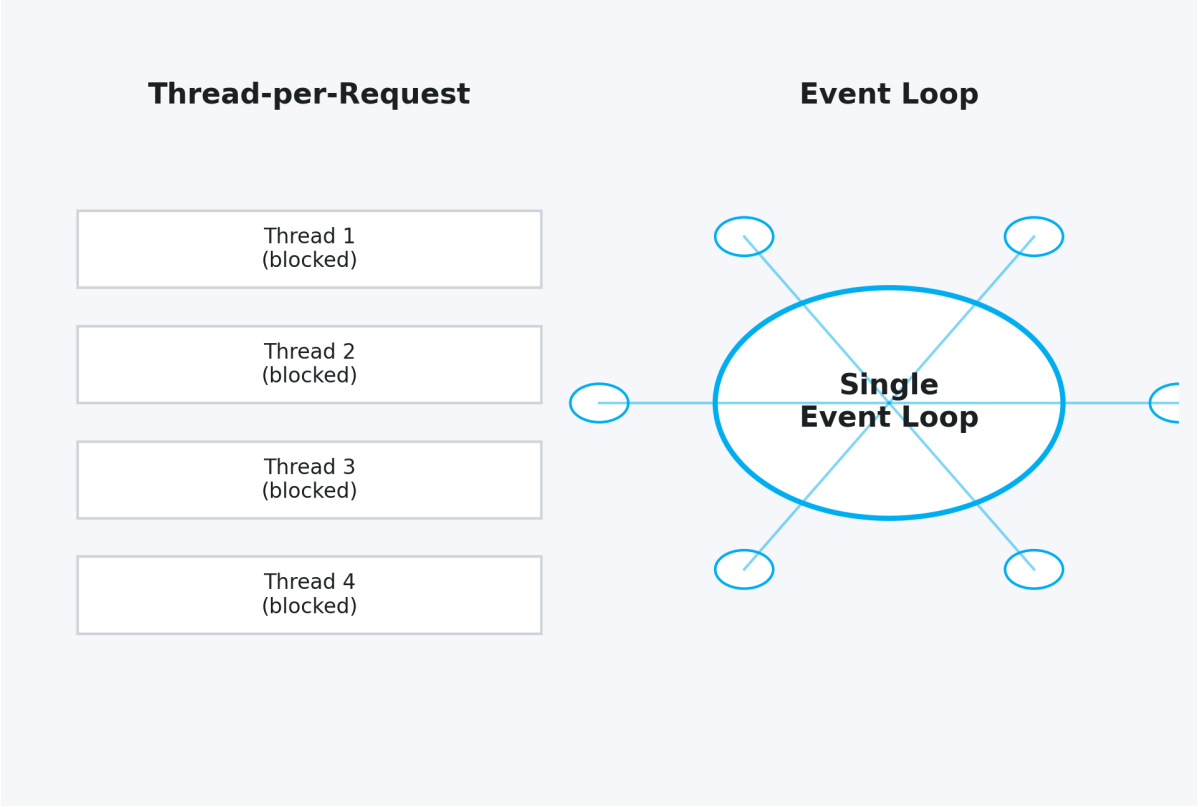


Figura 7: Event Loop Explicado

2.3 Tipos reactivos básicos: Mono y Flux

En el ecosistema Reactor, existen dos tipos fundamentales de flujos de datos:

Tipo	Descripción	Ejemplo práctico
Mono	Emite **0 o 1** elemento	Respuesta HTTP única, resultado de una operación asíncrona
Flux	Emite **0..N** elementos	Stream de datos continuos, listas reactivas, eventos en tiempo real

2.3.1 Ejemplos básicos:

```
// Mono: emite un solo valor
Mono<String> saludo = Mono.just("Hola Mundo Reactivo");

// Flux: emite múltiples valores
Flux<Integer> numeros = Flux.range(1, 5);

// Flux vacío
Flux<String> vacio = Flux.empty();

// Mono vacío (sin valor)
Mono<String> sinValor = Mono.empty();
```

Ambos son **asíncronos** y se pueden transformar con operadores funcionales como `map`, `flatMap`, `filter`, `zip`, `merge`, `concat`, etc. Es importante entender que estos tipos son **lazy** (perezosos): no ejecutan ninguna operación hasta que alguien se suscribe al flujo.

2.4 Creando el primer servicio reactivo

En el paquete `com.example.lambda.controller`, creamos el archivo `QuoteController.java`:

```
package com.example.lambda.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Flux;
import java.time.Duration;

@RestController
public class QuoteController {

    @GetMapping("/quotes")
    public Flux<String> streamQuotes() {
        return Flux.interval(Duration.ofSeconds(1))
                   .map(seq -> "Cotización #" + seq)
                   .take(10);
```

```
    }
}
```

🔍 Este endpoint emite un flujo de 10 mensajes, uno por segundo.
Si ejecutás la app y accedés a `http://localhost:8080/quotes`, verás las respuestas llegar de manera **streaming**.

> 🧠 **En la práctica**: La primera vez que probés esto, vas a notar que el navegador muestra las respuestas llegando progresivamente. Esto es el streaming reactivo en acción. Si tu endpoint tradicional con MVC tardaba 5 segundos en responder, acá vas a ver los primeros datos en menos de 1 segundo.

2.5 Enrutamiento funcional (RouterFunction)

Hasta ahora usamos anotaciones (`@RestController`, `@GetMapping`), que es el enfoque más común y familiar. Pero Spring WebFlux también ofrece una alternativa más declarativa y funcional que te va a resultar útil cuando necesites más control sobre el enrutamiento o cuando trabajes con configuraciones dinámicas.

Spring WebFlux también permite definir rutas sin anotaciones, usando el **enfoque funcional**:

```
@Configuration
public class QuoteRouter {

    @Bean
    public RouterFunction<ServerResponse> routeQuotes(QuoteHandler handler) {
        return RouterFunctions
            .route(RequestPredicates.GET("/quotes"), handler::streamQuotes);
    }
}

@Component
class QuoteHandler {
    public Mono<ServerResponse> streamQuotes(ServerRequest request) {
        Flux<String> quotes = Flux.interval(Duration.ofSeconds(1))
                                  .map(i -> "Quote #" + i)
                                  .take(5);
        return ServerResponse.ok().body(quotes, String.class);
    }
}
```

Ventajas del enfoque funcional: - Menos overhead de anotaciones.
- Configuración explícita del flujo reactivo.
- Mejor control sobre los `ServerResponse`.

> 🧠 **En la práctica**: El enrutamiento funcional es especialmente útil cuando necesitás crear rutas dinámicamente o cuando trabajás con versionado de APIs. Sin embargo, para la mayoría de casos, las anotaciones son más que suficientes y más legibles. Usá el enfoque funcional cuando realmente lo necesites.

2.6 WebClient: consumo reactivo de APIs externas

Ahora que ya sabés cómo crear endpoints reactivos, es momento de aprender a consumir APIs externas de forma reactiva. Esto es crucial porque en la práctica, casi siempre vas a necesitar llamar a otros servicios (APIs de terceros, bases de datos, servicios internos) y hacerlo de forma bloqueante anularía todos los beneficios del modelo reactivo.

El cliente `WebClient` reemplaza a `RestTemplate` en entornos reactivos.

Ejemplo de consumo de API externa:

```
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;

@Service
public class WeatherService {

    private final WebClient webClient =
        WebClient.create("https://api.open-meteo.com");

    public Mono<String> obtenerClima() {
        return webClient.get()
                .uri("/v1/forecast?latitude=-27.36"
                    + "&longitude=-55.90&current_weather=true")
                .retrieve()
                .bodyToMono(String.class);
    }
}
```

Y un endpoint que lo usa:

```
@RestController
public class WeatherController {

    private final WeatherService service;

    public WeatherController(WeatherService service) {
        this.service = service;
    }

    @GetMapping("/clima")
    public Mono<String> climaActual() {
        return service.obtenerClima();
    }
}
```

2.7 Manejo de errores reactivo

El manejo de errores en programación reactiva es fundamental para construir sistemas resilientes. Project Reactor proporciona varios operadores para manejar errores de forma elegante y controlada.

⚠ **En la práctica**: Si no manejás los errores correctamente en flujos reactivos, vas a terminar con excepciones que se propagan y terminan rompiendo toda la cadena. Esto es algo que te va a pasar seguro la primera vez que trabajes con WebFlux. La buena noticia es que Reactor tiene operadores específicos para esto, y una vez que los conozcas, el manejo de errores se vuelve mucho más elegante que en código bloqueante.

2.7.1 Operadores de manejo de errores

```
// onErrorReturn: retorna un valor por defecto en caso de error
Flux<String> flujo1 = Flux.just("A", "B")
    .concatWith(Mono.error(new RuntimeException("Error!")))
    .onErrorReturn("Valor por defecto");

// onErrorResume: permite recuperar con otro flujo
Flux<String> flujo2 = Flux.just("A", "B")
    .concatWith(Mono.error(new RuntimeException("Error!")))
    .onErrorResume(e -> Flux.just("Error capturado: " + e.getMessage()));

// onErrorMap: transforma el error en otro tipo de excepción
Flux<String> flujo3 = Flux.just("A", "B")
    .concatWith(Mono.error(new RuntimeException("Error original")))
    .onErrorMap(e -> new CustomException("Error procesado", e));

// doOnError: ejecuta una acción sin modificar el flujo (útil para logging)
Flux<String> flujo4 = Flux.just("A", "B")
    .concatWith(Mono.error(new RuntimeException("Error!")))
    .doOnError(e -> log.error("Error en el flujo: {}", e.getMessage()))
    .onErrorReturn("Valor recuperado");
```

2.7.2 Manejo de errores por tipo

```
Flux<String> flujo = Flux.just("A", "B")
    .concatWith(Mono.error(new IllegalArgumentException("Error de validación")))
    .onErrorResume(IllegalArgumentException.class, e ->
        Flux.just("Error de validación manejado"))
    .onErrorResume(RuntimeException.class, e ->
        Flux.just("Error genérico manejado"))
    .onErrorReturn("Error desconocido");
```

2.7.3 Timeout y retry

```
// Timeout: cancela la operación si excede un tiempo límite
Flux<String> flujoConTimeout = Flux.just("A", "B")
    .delayElements(Duration.ofSeconds(2))
    .timeout(Duration.ofSeconds(1))
    .onErrorReturn("Timeout alcanzado");
```

```
// Retry: reintenta la operación en caso de error
Flux<String> flujoConRetry = Flux.just("A", "B")
    .concatWith(Mono.error(new RuntimeException("Error temporal")))
    .retry(3) // Reintenta hasta 3 veces
    .onErrorReturn("Error después de reintentos");
```

2.7.4 GlobalErrorWebExceptionHandler para excepciones globales

Para manejar excepciones a nivel global en Spring WebFlux, puedes crear un handler personalizado:

```
@Component
public class GlobalErrorWebExceptionHandler implements ErrorWebExceptionHandler {

    private final ObjectMapper objectMapper;

    public GlobalErrorWebExceptionHandler(ObjectMapper objectMapper) {
        this.objectMapper = objectMapper;
    }

    @Override
    public Mono<Void> handle(ServerWebExchange exchange, Throwable ex) {
        ServerHttpResponse response = exchange.getResponse();
        response.setStatusCode(HttpStatus.INTERNAL_SERVER_ERROR);
        response.getHeaders().setContentType(MediaType.APPLICATION_JSON);

        Map<String, Object> error = Map.of(
            "error", ex.getMessage(),
            "timestamp", Instant.now().toString(),
            "path", exchange.getRequest().getPath().value()
        );

        return response.writeWith(
            Mono.fromCallable(() -> {
                try {
                    return response.bufferFactory()
                        .wrap(objectMapper.writeValueAsBytes(error));
                } catch (Exception e) {
                    return response.bufferFactory().wrap("{}".getBytes());
                }
            })
        );
    }
}
```

Esto permite recuperar el flujo sin detener toda la cadena reactiva y proporciona un manejo de errores robusto y centralizado.

2.8 Testing con StepVerifier

`StepVerifier` permite probar flujos reactivos verificando cada emisión:

```
import org.junit.jupiter.api.Test;
import reactor.core.publisher.Flux;
import reactor.test.StepVerifier;

public class QuoteControllerTest {

    @Test
    void testFlujoQuotes() {
        Flux<String> flux = Flux.range(1, 3).map(i -> "Quote #" + i);
        StepVerifier.create(flux)
            .expectNext("Quote #1")
            .expectNext("Quote #2")
            .expectNext("Quote #3")
            .verifyComplete();
    }
}
```

2.9 Resumen y próximos pasos

En esta sección aprendimos: - Diferencias entre MVC y WebFlux. - Cómo construir endpoints reactivos con `Flux` y `Mono`. - Consumo de APIs externas con `WebClient`. - Testing con `StepVerifier`.

En la próxima sección transformaremos este microservicio en una **función serverless** y lo desplegaremos en AWS Lambda.

3 De microservicio reactivo a función serverless

3.1 Objetivo

Transformar el microservicio WebFlux desarrollado previamente en una **función AWS Lambda** utilizando **Spring Cloud Function** y **Spring Boot 3.4+**, y aprender a desplegarla y probarla tanto localmente como en AWS.

3.2 Introducción a Spring Cloud Function

Hasta ahora construimos un microservicio reactivo que funciona perfectamente como aplicación web tradicional. Pero ¿qué pasa si querés desplegarlo como función serverless en AWS Lambda? ¿Tenés que reescribir todo el código? La respuesta es no, y acá es donde entra **Spring Cloud Function**.

> **⚠ Nota importante sobre versiones:**

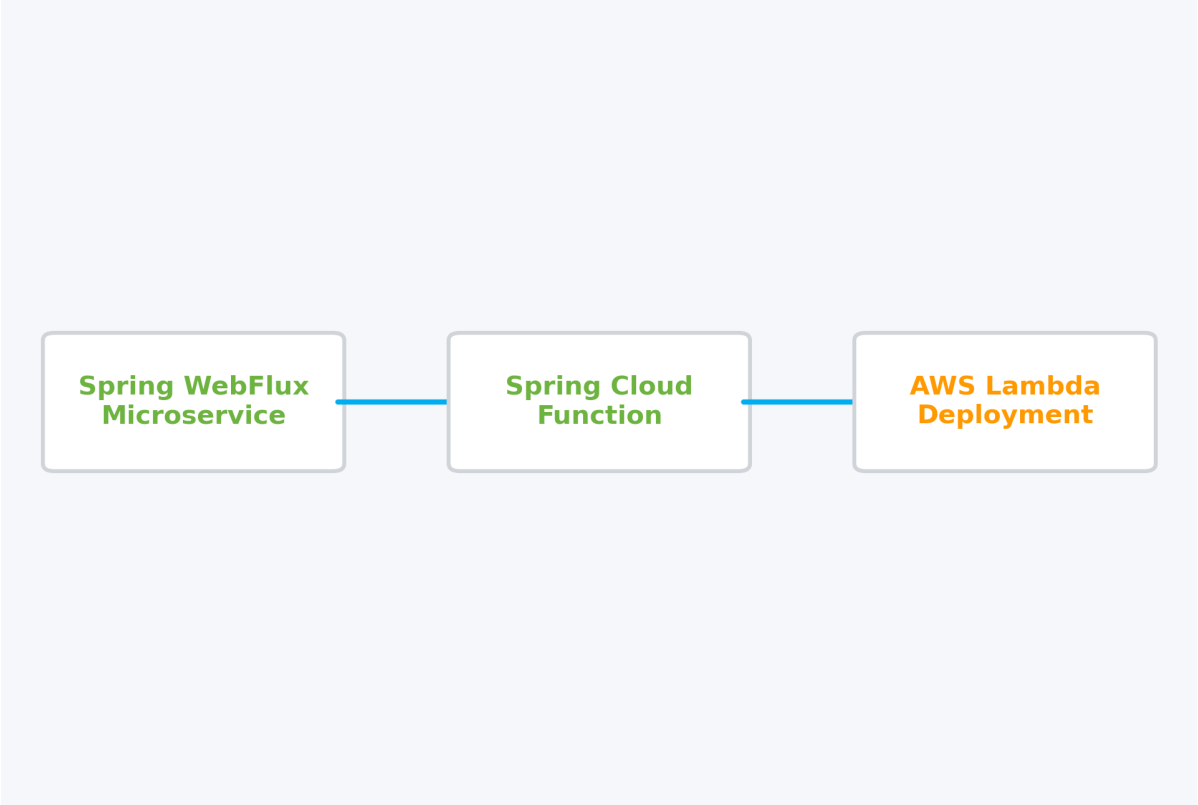


Figura 8: Conversión a Lambda

> **Spring Cloud Function requiere Spring Cloud 2024.0.0 o superior cuando se usa con Spring Boot 3.4+.** Las versiones anteriores de Spring Cloud (como 2023.0.1) no son completamente compatibles y pueden causar problemas. Por esta razón, en este ebook usamos: - **Spring Cloud 2024.0.0** (compatible con Spring Boot 3.4+) - **Spring Boot 3.4.0+** (mínimo recomendado)
>
> Esta incompatibilidad entre Spring Cloud 2023.x y Spring Boot 3.4+ es una de las razones principales por las que migramos a las versiones más recientes.

Spring Cloud Function es un framework que permite escribir lógica de negocio independiente del entorno de ejecución. Esto significa que puedes ejecutar la misma función en diferentes contextos sin modificar el código:

- **Aplicación web**: Como endpoint HTTP en una aplicación Spring Boot tradicional.
- **Tarea programada**: Como job programado con Spring Scheduler o Quartz.
- **Función serverless**: En AWS Lambda, Azure Functions o Google Cloud Functions.

Esta abstracción facilita la **portabilidad y reutilización del código**, permitiendo migrar entre diferentes plataformas sin cambios significativos en la lógica de negocio.

> En la práctica: Esta es una de las decisiones más inteligentes que podés tomar cuando estás empezando con serverless. En lugar de escribir código específico para Lambda desde el principio, escribís código portable que después podés ejecutar en cualquier lado. Esto te va a ahorrar muchísimo tiempo cuando necesites cambiar de plataforma o cuando quieras probar localmente antes de desplegar.

3.2.1 Principales tipos funcionales admitidos:

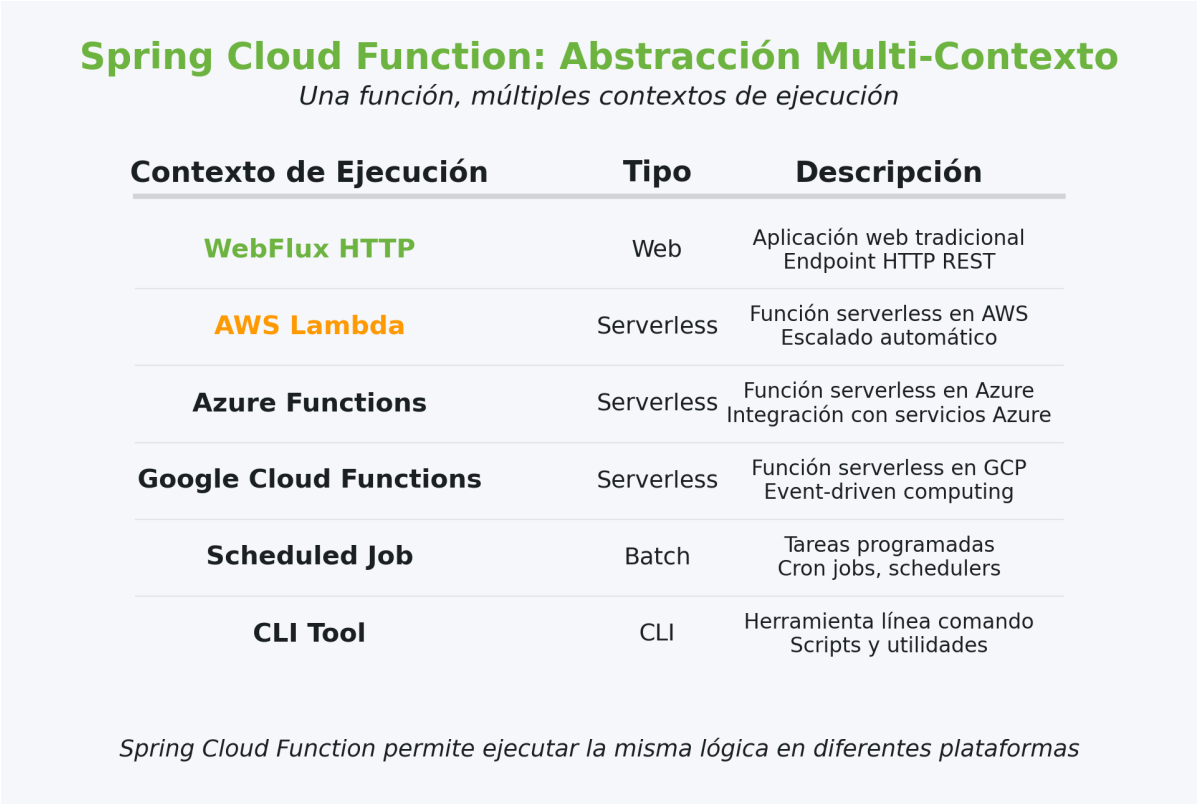


Figura 9: Abstracción Spring Cloud Function

Tipo	Descripción	Uso típico
`Supplier<T>`	Genera un valor sin recibir entrada	Health checks, generadores de datos
`Function<T, R>`	Transforma un valor de entrada en un valor de salida	Procesamiento de requests HTTP, transformación de datos
`Consumer<T>`	Procesa un valor sin devolver resultado	Envío de mensajes, logging, procesamiento asíncrono

3.2.2 Ejemplo simple:

```java
package com.example.lambda;

import org.springframework.stereotype.Component;
import java.util.function.Function;

/**
 * Función simple que demuestra el uso básico de Spring Cloud Function.
 * Esta función puede ejecutarse en diferentes contextos (HTTP, Lambda, etc.)
 */
@Component
public class HelloFunction implements Function<String, String> {
    @Override
    public String apply(String input) {
```

```
        return "Hola " + input + ", desde Spring Cloud Function!";
    }
}
```

Nota: Spring Cloud Function detecta automáticamente las funciones anotadas con `@Component` y las registra para su uso.

3.3 Configuración del proyecto para Lambda

Agregamos al `build.gradle.kts` el adaptador AWS con las dependencias actualizadas:

```
dependencies {
    implementation("org.springframework.boot:spring-boot-starter-webflux")
    implementation("org.springframework.cloud:spring-cloud-function-context")
    implementation("org.springframework.cloud:spring-cloud-function-adapter-aws")

    // AWS Lambda Java Events (solo necesarios para tipos de eventos)
    // Nota: Spring Cloud Function maneja la invocación, solo necesitamos los tipos de eventos
    implementation("com.amazonaws:aws-lambda-java-events:3.15.0")

    // Bean Validation para validar entrada
    implementation("org.springframework.boot:spring-boot-starter-validation")

    // Jackson para serialización
    implementation("com.fasterxml.jackson.core:jackson-databind")
}

// ▫ IMPORTANTE: Incompatibilidad con Spring AOT
// Spring Cloud Function no es completamente compatible con el procesamiento AOT de Spring Boot
// Es necesario deshabilitar las tareas AOT para evitar errores de inicialización.
tasks.named("processAot") {
    enabled = false
}
tasks.named("compileAotJava") {
    enabled = false
}
tasks.named("processAotResources") {
    enabled = false
}
tasks.named("aotClasses") {
    enabled = false
}
```

> **⚠ Nota importante sobre AOT y Spring Cloud Function:**
>
> **Spring Cloud Function no es completamente compatible con el procesamiento AOT (Ahead-Of-Time)** de Spring Boot 3.4+. Si intentás compilar con AOT habilitado, podés encontrar errores como:
>
> `java.lang.IllegalArgumentException: Unable to instantiate factory class`

`[org.springframework.cloud.function.web.function.FunctionEndpointInitializer]`

Solución: Deshabilitar las tareas AOT en `build.gradle.kts` (como se muestra arriba). Esto **no afecta la funcionalidad** de la aplicación ni la capacidad de usar GraalVM Native Image. La compilación nativa con GraalVM funciona correctamente sin AOT.

En la práctica: Esta es una limitación conocida que probablemente encuentres cuando trabajes con Spring Cloud Function. La deshabilitación de AOT es la solución estándar y recomendada.

3.3.1 Configuración del handler para AWS Lambda

El **handler** para AWS Lambda se declara en el archivo `src/main/resources/META-INF/MANIFEST.MF`:

`Main-Class: org.springframework.cloud.function.adapter.aws.FunctionInvoker`

Este invocador genérico (`FunctionInvoker`) detecta automáticamente las funciones definidas con `@Component` y las expone como handlers de AWS Lambda.

Importante: La función específica a ejecutar se define mediante la variable de entorno `SPRING_CLOUD_FUNCTION_DEFINITION`. Por ejemplo: - `SPRING_CLOUD_FUNCTION_DEFINITION=hello` → ejecuta la función `hello()` - `SPRING_CLOUD_FUNCTION_DEFINITION=quoteFunction` → ejecuta la función `quoteFunction()`

3.4 Definiendo la función Lambda reactiva

Creamos una función que devuelva datos en flujo usando Project Reactor. Esta función demuestra cómo combinar Spring Cloud Function con programación reactiva:

```java
package com.example.lambda;

import org.springframework.stereotype.Component;
import reactor.core.publisher.Flux;
import java.time.Duration;
import java.util.function.Supplier;

/**
 * Función Lambda reactiva que genera un flujo de cotizaciones.
 * Implementa Supplier<Flux<String>> para generar datos de forma reactiva.
 */
@Component
public class QuotesFunction implements Supplier<Flux<String>> {

    @Override
    public Flux<String> get() {
        // Flux.interval genera valores cada segundo (0, 1, 2, 3, ...)
        // .map transforma cada valor en un mensaje de cotización
        // .take(5) limita el flujo a 5 elementos
        return Flux.interval(Duration.ofSeconds(1))
                   .map(i -> "Quote reactivo #" + i)
                   .take(5);
```

```
    }
}
```

Características importantes: - Esta función genera un **Flux** de datos simulando cotizaciones que se emitirán cada segundo. - El flujo es **lazy**: no se ejecuta hasta que alguien se suscribe. - El backpressure se maneja automáticamente mediante Reactive Streams. - En AWS Lambda, el flujo se procesa completamente antes de retornar la respuesta.

3.5 Empaquetado para AWS Lambda

Ejecutamos el siguiente comando para crear el *fat jar*:

```
./gradlew clean bootJar
```

El resultado será un archivo similar a:

```
build/libs/reactive-lambda-0.0.1-SNAPSHOT.jar
```

Este JAR puede ser cargado directamente en AWS Lambda o invocado con **SAM CLI** localmente.

3.6 Prueba local con AWS SAM CLI

Creamos un archivo `template.yaml` para definir la función:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Demo función Spring Cloud Function Reactiva
Resources:
  QuotesFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: org.springframework.cloud.function.adapter.aws.FunctionInvoker::handleRequest
      Runtime: java21  # ▫ Ver nota sobre Runtime Java abajo
      MemorySize: 512
      Timeout: 30
      CodeUri: build/libs/reactive-lambda-0.0.1-SNAPSHOT.jar
      Environment:
        Variables:
          MAIN_CLASS: com.example.lambda.QuotesFunction
```

> **⚠ Nota importante sobre Runtime Java:**
>
> Aunque este ebook utiliza **Java 25 LTS** para desarrollo, AWS Lambda actualmente soporta hasta **Java 21** como runtime. El código desarrollado con Java 25 es **completamente compatible** con el runtime Java 21 de Lambda, ya que Java mantiene compatibilidad hacia atrás (backward compatibility).
>
> **En la práctica:** Desarrollás con Java 25 LTS localmente (para aprovechar las nuevas características como Compact Object Headers), pero el runtime de Lambda debe ser `java21`. Esto no afecta la funcionalidad ni el rendimiento de tu aplicación.

Futuro: Cuando AWS Lambda agregue soporte para Java 25, simplemente cambiarás `Runtime: java21` a `Runtime: java25` en el template SAM.

Probamos localmente:

```
sam local invoke "QuotesFunction"
```

Salida esperada:

```
Quote reactivo #0
Quote reactivo #1
Quote reactivo #2
Quote reactivo #3
Quote reactivo #4
```

3.7 Despliegue en AWS

⚠ ADVERTENCIA CRÍTICA SOBRE COSTOS:

Antes de desplegar, **configurá presupuestos y alertas de AWS** para evitar facturas inesperadas. Los servicios AWS generan costos reales. Consultá el **Capítulo 11: Control de Costos y Disclaimers Legales** para más información.

Recomendaciones inmediatas: 1. Configurá un presupuesto mensual (ej: $10 USD) con alertas 2. Usá AWS Free Tier cuando sea posible 3. Eliminá recursos después de las pruebas 4. Monitoreá costos diariamente durante el desarrollo

1. **Creación del bucket S3** para el artefacto:

   ```
   aws s3 mb s3://reactive-lambda-artifacts
   ```

2. **Despliegue:**

   ```
   sam deploy --guided
   ```

   Configurá el nombre del stack, región y bucket cuando te lo solicite.

3. Una vez desplegado, obtené el ARN de la función:

   ```
   aws lambda list-functions
   ```

4. Invocación directa desde AWS CLI:

   ```
   aws lambda invoke --function-name QuotesFunction output.txt
   cat output.txt
   ```

3.8 Observación del comportamiento reactivo

Aunque Lambda ejecuta funciones individuales, el flujo **Flux** se procesa internamente como un stream reactivo controlado por el runtime.
Esto demuestra que el modelo **reactivo y serverless** puede coexistir, logrando eficiencia y bajo consumo de recursos.

Ventajas observadas: - Tiempo de arranque rápido (<400 ms con GraalVM).
- Uso eficiente de memoria.
- Escalado automático por demanda.

3.9 Implementación de Función Lambda Reactiva (Código Real del Proyecto)

A continuación mostramos la implementación real de una función Lambda reactiva siguiendo las mejores prácticas. Todo este código está disponible en el proyecto `reactive-microservices-aws-lambda-java2` específicamente en el módulo `lambda-core`.

> 📁 **Ver el código completo**: Podés encontrar estos archivos en el repositorio: - `lambda-core/src/main/java/com/example/lambda/FunctionConfig.java` - `lambda-core/src/main/java/com/example/lambda/handlers/HelloHandler.java`
>
> ⚠ **Nota sobre correspondencia**: El código mostrado en esta sección usa el paquete `com.example.lambda`, que coincide exactamente con el proyecto real. Este es el mismo código que encontrarás en el repositorio.

3.9.1 FunctionConfig.java - Configuración de Funciones

```
package com.example.lambda;

import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import com.example.lambda.handlers.HelloHandler;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import reactor.core.publisher.Mono;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;
import java.util.function.Supplier;

/**
 * Configuration for AWS Lambda functions using Spring Cloud Function.
 * All functions are implemented reactively using Project Reactor.
 */
@Configuration
public class FunctionConfig {

    private static final Logger logger = LoggerFactory.getLogger(FunctionConfig.class);
    private final HelloHandler helloHandler;

    public FunctionConfig(HelloHandler helloHandler) {
```

```
        this.helloHandler = helloHandler;
    }

    /**
     * Hello function that handles API Gateway requests reactively.
     * Can be activated by setting SPRING_CLOUD_FUNCTION_DEFINITION=hello
     *
     * NOTE: Spring Cloud Function adapter for AWS Lambda requires
     * synchronous Function signature, but we process reactively internally
     * using Reactor and block only at the boundary (which is acceptable
     * for Lambda as each invocation is independent).
     *
     * For truly reactive endpoints, use WebFlux controllers with
     * spring-cloud-starter-function-web.
     */
    @Bean
    public Function<APIGatewayProxyRequestEvent,
                    APIGatewayProxyResponseEvent> hello() {
        return request -> {
            try {
                // Extract request data
                String path = request.getPath();
                Map<String, String> pathParameters = request.getPathParameters();
                Map<String, String> queryStringParameters =
                    request.getQueryStringParameters();
                String body = request.getBody();

                // Build echo message
                StringBuilder echo = new StringBuilder();
                echo.append("path=").append(path != null ? path : "");

                if (pathParameters != null && !pathParameters.isEmpty()) {
                    echo.append(", pathParams=").append(pathParameters);
                }

                if (queryStringParameters != null &&
                    !queryStringParameters.isEmpty()) {
                    echo.append(", queryParams=").append(queryStringParameters);
                }

                if (body != null && !body.isEmpty()) {
                    echo.append(", body=").append(body);
                }

                // Process greeting reactively (if name is provided)
                // Note: We block here because Lambda Function must return
                // synchronously. This is the reactive boundary - inside we use
                // Mono/Flux
                String name = queryStringParameters != null
```

```
                ? queryStringParameters.get("name")
                : null;
            String greeting = name != null
                ? helloHandler.processGreeting(name)
                    .timeout(java.time.Duration.ofSeconds(5))
                    .onErrorResume(error -> {
                        logger.warn("Error processing greeting, using default", error);
                        return Mono.just("Hello, World!");
                    })
                    .block(java.time.Duration.ofSeconds(5))
                    // Necessary boundary: Lambda requires synchronous
                    // return with timeout
                : "Hello, World!";

            String jsonResponse = String.format(
                "{\"message\":\"ok\",\"echo\":\"%s\","
                + "\"greeting\":\"%s\",\"timestamp\":\"%s\"}",
                echo.toString().replace("\"", "\\\""),
                greeting.replace("\"", "\\\""),
                java.time.Instant.now().toString()
            );

            APIGatewayProxyResponseEvent response = new APIGatewayProxyResponseEvent();
            response.setStatusCode(HttpStatus.OK.value());
            response.setBody(jsonResponse);

            Map<String, String> headers = new HashMap<>();
            headers.put("Content-Type", MediaType.APPLICATION_JSON_VALUE);
            headers.put("X-Request-Id",
                request.getRequestContext() != null
                    ? request.getRequestContext().getRequestId()
                    : "unknown");
            response.setHeaders(headers);

            return response;

        } catch (Exception e) {
            String requestId = request.getRequestContext() != null
                ? request.getRequestContext().getRequestId()
                : "unknown";
            logger.error("Error processing request: {}", e.getMessage(), e);
            // Manejo de errores centralizado
            return createErrorResponse(500, "Internal Server Error: " + e.getMessage());
        }
    };
}

/**
 * Helper method to create error responses.
```

```
     */
    private APIGatewayProxyResponseEvent createErrorResponse(int statusCode, String message) {
        try {
            Map<String, Object> error = new HashMap<>();
            error.put("error", message);
            error.put("timestamp", java.time.Instant.now().toString());

            ObjectMapper objectMapper = new ObjectMapper();
            String jsonBody = objectMapper.writeValueAsString(error);

            APIGatewayProxyResponseEvent response = new APIGatewayProxyResponseEvent();
            response.setStatusCode(statusCode);
            response.setBody(jsonBody);

            Map<String, String> headers = new HashMap<>();
            headers.put("Content-Type", MediaType.APPLICATION_JSON_VALUE);
            response.setHeaders(headers);

            return response;
        } catch (Exception e) {
            APIGatewayProxyResponseEvent response = new APIGatewayProxyResponseEvent();
            response.setStatusCode(500);
            response.setBody("{\"error\": \"Internal Server Error\"}");
            return response;
        }
    }

    /**
     * Simple supplier function for health checks.
     * Can be activated by setting SPRING_CLOUD_FUNCTION_DEFINITION=pong
     */
    @Bean
    public Supplier<String> pong() {
        return () -> "pong";
    }
}
```

3.9.2 HelloHandler.java - Handler Reactivo con Métricas

```
package com.example.lambda.handlers;

import io.micrometer.core.instrument.Counter;
import io.micrometer.core.instrument.MeterRegistry;
import org.springframework.stereotype.Component;
import reactor.core.publisher.Mono;
import java.time.Duration;

/**
```

```java
 * Demonstrative handler that uses Project Reactor and Micrometer for observability.
 */
@Component
public class HelloHandler {

    private final Counter invocationsCounter;
    private final MeterRegistry meterRegistry;

    public HelloHandler(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;
        this.invocationsCounter = Counter.builder("lambda.invocations")
            .description("Total number of Lambda invocations")
            .tag("handler", "hello")
            .register(meterRegistry);
    }

    /**
     * Process a greeting request reactively.
     *
     * @param name the name to greet
     * @return a Mono containing the greeting message
     */
    public Mono<String> processGreeting(String name) {
        return Mono.just(name)
            .delayElement(Duration.ofMillis(10)) // Simulate async processing
            .map(n -> String.format("Hello, %s!", n != null && !n.isEmpty() ? n : "World"))
            .doOnNext(result -> {
                invocationsCounter.increment();
                meterRegistry.counter("lambda.processed", "status", "success").increment();
            })
            .doOnError(error -> {
                meterRegistry.counter("lambda.processed", "status", "error").increment();
            });
    }

    /**
     * Get invocation count.
     *
     * @return current invocation count
     */
    public double getInvocationCount() {
        return invocationsCounter.count();
    }
}
```

Características clave: - [OK] Procesamiento reactivo interno con Project Reactor - [OK] Timeout configurado (5 segundos) - [OK] Manejo de errores reactivo con `onErrorResume()` - [OK] Métricas con Micrometer - [OK] Bloqueo solo en el límite Lambda (necesario por la naturaleza síncrona del handler) - [OK] Manejo robusto de errores con estrategia reactiva

3.10 Manejo Robusto de Errores

Para producción, es esencial manejar errores de forma centralizada y robusta. El siguiente ejemplo muestra una implementación completa con manejo reactivo de errores:

```
package com.example.lambda;

import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.example.lambda.exception.ValidationException;
import com.example.lambda.model.QuoteRequest;
import com.example.lambda.service.QuoteService;
import jakarta.validation.ConstraintViolation;
import jakarta.validation.Validator;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;
import reactor.core.publisher.Mono;
import reactor.core.scheduler.Schedulers;
import java.util.function.Function;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import java.util.stream.Collectors;
import java.time.Duration;
import java.util.concurrent.TimeoutException;

@Component
public class QuoteFunction implements Function<APIGatewayProxyRequestEvent, APIGatewayProxyResp

    private static final Logger log = LoggerFactory.getLogger(QuoteFunction.class);
    private final ObjectMapper objectMapper;
    private final QuoteService quoteService;
    private final Validator validator;

    public QuoteFunction(ObjectMapper objectMapper, QuoteService quoteService, Validator valida
        this.objectMapper = objectMapper;
        this.quoteService = quoteService;
        this.validator = validator;
    }

    // NOTA CRÍTICA: Lambda runtime requiere respuesta síncrona, pero mantenemos el pipeline re
    // Spring Cloud Function maneja la conversión automáticamente
    @Override
    public APIGatewayProxyResponseEvent apply(APIGatewayProxyRequestEvent request) {
        // Procesar la solicitud de forma reactiva y bloquear solo en el límite Lambda
```

```
        return processRequest(request)
            .blockOptional(Duration.ofSeconds(30))
            .orElseGet(() -> createErrorResponse(500, "Request timeout"));
    }

    // Código completamente reactivo - sin bloqueos hasta el punto final necesario
    private Mono<APIGatewayProxyResponseEvent> processRequest(APIGatewayProxyRequestEvent reque
        log.info("Processing request: {} {}", request.getHttpMethod(), request.getPath());

        return Mono.just(request)
            .flatMap(req -> {
                String httpMethod = req.getHttpMethod();
                String path = req.getPath();

                if ("GET".equals(httpMethod) && "/quotes".equals(path)) {
                    return handleGetQuotes();
                } else if ("POST".equals(httpMethod) && "/quotes".equals(path)) {
                    return handleCreateQuote(req);
                } else {
                    return Mono.just(createErrorResponse(404, "Not Found"));
                }
            })
            .onErrorResume(this::handleErrorReactive)
            .doOnError(error -> log.error("Unhandled error processing request", error))
            .onErrorReturn(createErrorResponse(500, "Internal Server Error"));
    }

    private Mono<APIGatewayProxyResponseEvent> handleGetQuotes() {
        return quoteService.findAllQuotes()
            .collectList()
            .map(quotes -> {
                Map<String, Object> response = Map.of("quotes", quotes);
                return createSuccessResponse(response);
            })
            .onErrorResume(e -> {
                log.error("Error fetching quotes", e);
                return Mono.just(createErrorResponse(500, "Error fetching quotes"));
            });
    }

    private Mono<APIGatewayProxyResponseEvent> handleCreateQuote(APIGatewayProxyRequestEvent re
        return Mono.fromCallable(() -> {
                if (request.getBody() == null || request.getBody().isEmpty()) {
                    throw new ValidationException("Request body is required");
                }
                return objectMapper.readValue(request.getBody(), QuoteRequest.class);
            })
            .flatMap(this::validateQuoteRequest)
            .flatMap(quoteRequest -> quoteService.createQuote(quoteRequest))
```

```
            .map(quote -> {
                Map<String, Object> response = Map.of(
                    "message", "Quote created successfully",
                    "id", quote.getId()
                );
                return createSuccessResponse(response);
            })
            .onErrorResume(ValidationException.class, e -> {
                log.warn("Validation error: {}", e.getMessage());
                return Mono.just(createErrorResponse(400, e.getMessage()));
            })
            .onErrorResume(e -> {
                log.error("Error creating quote", e);
                return Mono.just(createErrorResponse(400, "Invalid request: " + e.getMessage()));
            });
    }

    private Mono<QuoteRequest> validateQuoteRequest(QuoteRequest request) {
        // Validación reactiva sin bloquear el hilo principal
        return Mono.fromCallable(() -> {
                Set<ConstraintViolation<QuoteRequest>> violations = validator.validate(request);
                if (!violations.isEmpty()) {
                    String errors = violations.stream()
                        .map(v -> v.getPropertyPath() + ": " + v.getMessage())
                        .collect(Collectors.joining(", "));
                    throw new ValidationException("Validation failed: " + errors);
                }
                return request;
            })
            .subscribeOn(Schedulers.boundedElastic())
            .timeout(Duration.ofSeconds(2)) // Timeout para evitar bloqueos prolongados
            .onErrorMap(TimeoutException.class, e -> new ValidationException("Validation timeou
    }

    private Mono<APIGatewayProxyResponseEvent> handleErrorReactive(Throwable error) {
        log.error("Error processing request", error);

        if (error instanceof ValidationException) {
            return Mono.just(createErrorResponse(400, error.getMessage()));
        } else if (error instanceof JsonProcessingException) {
            return Mono.just(createErrorResponse(400, "Invalid JSON: " + error.getMessage()));
        } else {
            return Mono.just(createErrorResponse(500, "Internal Server Error"));
        }
    }

    private APIGatewayProxyResponseEvent createSuccessResponse(Object body) {
        try {
            String jsonBody = objectMapper.writeValueAsString(body);
```

```
            return APIGatewayProxyResponseEvent.builder()
                .withStatusCode(200)
                .withBody(jsonBody)
                .withHeaders(createHeaders())
                .build();
        } catch (Exception e) {
            return createErrorResponse(500, "Error serializing response");
        }
    }

    private APIGatewayProxyResponseEvent createErrorResponse(int statusCode, String message) {
        try {
            Map<String, String> error = Map.of(
                "error", message,
                "timestamp", java.time.Instant.now().toString()
            );
            String jsonBody = objectMapper.writeValueAsString(error);

            return APIGatewayProxyResponseEvent.builder()
                .withStatusCode(statusCode)
                .withBody(jsonBody)
                .withHeaders(createHeaders())
                .build();
        } catch (Exception e) {
            return APIGatewayProxyResponseEvent.builder()
                .withStatusCode(500)
                .withBody("{\"error\": \"Internal Server Error\"}")
                .build();
        }
    }

    private Map<String, String> createHeaders() {
        Map<String, String> headers = new HashMap<>();
        headers.put("Content-Type", "application/json");
        headers.put("Access-Control-Allow-Origin", "*");
        return headers;
    }
}
```

3.10.1 Clases auxiliares necesarias

```
// ValidationException.java - Excepción personalizada para validación
package com.example.lambda.exception;

public class ValidationException extends RuntimeException {
    public ValidationException(String message) {
        super(message);
    }
```

```
    public ValidationException(String message, Throwable cause) {
        super(message, cause);
    }
}

// QuoteRequest.java - Modelo con validación
package com.example.lambda.model;

import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;

public class QuoteRequest {

    @NotBlank(message = "Text is required")
    @Size(min = 1, max = 500, message = "Text must be between 1 and 500 characters")
    private String text;

    @Size(max = 100, message = "Author must not exceed 100 characters")
    private String author;

    // Getters y setters
    public String getText() { return text; }
    public void setText(String text) { this.text = text; }
    public String getAuthor() { return author; }
    public void setAuthor(String author) { this.author = author; }
}

// Quote.java - Modelo de dominio
package com.example.lambda.model;

public class Quote {
    private String id;
    private String text;
    private String author;
    private Long timestamp;

    // Getters y setters
    public String getId() { return id; }
    public void setId(String id) { this.id = id; }
    public String getText() { return text; }
    public void setText(String text) { this.text = text; }
    public String getAuthor() { return author; }
    public void setAuthor(String author) { this.author = author; }
    public Long getTimestamp() { return timestamp; }
    public void setTimestamp(Long timestamp) { this.timestamp = timestamp; }
}

// QuoteRepository.java - Repositorio reactivo
```

```
package com.example.lambda.repository;

import com.example.lambda.model.Quote;
import org.springframework.stereotype.Repository;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

@Repository
public class QuoteRepository {
    private final Map<String, Quote> quotes = new ConcurrentHashMap<>();

    public Flux<Quote> findAll() {
        return Flux.fromIterable(quotes.values());
    }

    public Mono<Quote> save(Quote quote) {
        quotes.put(quote.getId(), quote);
        return Mono.just(quote);
    }

    public Mono<Quote> findById(String id) {
        Quote quote = quotes.get(id);
        return quote != null ? Mono.just(quote) : Mono.empty();
    }
}

// QuoteService.java - Servicio reactivo
package com.example.lambda.service;

import com.example.lambda.exception.ValidationException;
import com.example.lambda.model.Quote;
import com.example.lambda.model.QuoteRequest;
import com.example.lambda.repository.QuoteRepository;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
import java.time.Instant;
import java.util.UUID;

@Service
public class QuoteService {

    private static final Logger log = LoggerFactory.getLogger(QuoteService.class);
    private final QuoteRepository quoteRepository;
```

```
    public QuoteService(QuoteRepository quoteRepository) {
        this.quoteRepository = quoteRepository;
    }

    public Flux<Quote> findAllQuotes() {
        return quoteRepository.findAll()
            .onErrorResume(e -> {
                log.error("Error fetching quotes", e);
                return Flux.empty();
            });
    }

    public Mono<Quote> createQuote(QuoteRequest request) {
        Quote quote = new Quote();
        quote.setId(UUID.randomUUID().toString());
        quote.setText(request.getText());
        quote.setAuthor(request.getAuthor());
        quote.setTimestamp(Instant.now().toEpochMilli());

        return quoteRepository.save(quote)
            .onErrorResume(e -> {
                log.error("Error creating quote", e);
                return Mono.error(new RuntimeException("Failed to create quote", e));
            });
    }
}
```

3.11 Integración con AWS API Gateway (Template Real del Proyecto)

AWS API Gateway permite exponer nuestras funciones Lambda como HTTP APIs. Aquí está el template SAM real del proyecto `reactive-microservices-aws-lambda-java25`, ubicado en `lambda-infra/template.yaml`:

> 📄 **Ver el template completo**: El archivo completo está disponible en el repositorio en `lambda-infra/template.yaml`. Podés clonar el proyecto y usar este template directamente para desplegar tu función Lambda.

> **⚠ HTTP API vs REST API (2024-2025):**
>
> Este ebook utiliza **HTTP API** (API Gateway v2), que es la opción moderna y recomendada para la mayoría de casos: - **60% más barato** que REST API - **Latencia más baja** (mejor integración con Lambda) - **Configuración más simple** (menos verboso) - **Soporte nativo para CORS** y autenticación JWT
>
> **REST API** (API Gateway v1) solo se recomienda si necesitás features específicas como: - API Keys avanzados con Usage Plans complejos - Request/Response transformations complejas - Integración con servicios que requieren REST API específicamente
>
> Para el 95% de los casos, **HTTP API es la mejor opción** (2024-2025).

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: >
  Microservicios Reactivos con Spring Boot y AWS Lambda

  Este template SAM despliega una función Lambda reactiva con Spring Boot 3.4+ y Java 25 LTS.

Globals:
  Function:
    Timeout: 30
    MemorySize: 1024
    Runtime: java21  # ▫ Ver nota sobre Runtime Java: Java 25 compatible con runtime java21
    Handler: org.springframework.cloud.function.adapter.aws.FunctionInvoker::handleRequest
    Environment:
      Variables:
        SPRING_CLOUD_FUNCTION_DEFINITION: hello

Resources:
  ReactiveFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: build/libs/reactive-lambda-0.0.1-SNAPSHOT.jar
      PackageType: Zip
      Description: Función Lambda reactiva con Spring Boot 3.4+ y Project Reactor
      Environment:
        Variables:
          SPRING_CLOUD_FUNCTION_DEFINITION: hello
          SPRING_PROFILES_ACTIVE: aws
      Events:
        HttpApi:
          Type: HttpApi
          Properties:
            Path: /hello
            Method: GET
            ApiId: !Ref ReactiveApi
        HttpApiPost:
          Type: HttpApi
          Properties:
            Path: /hello
            Method: POST
            ApiId: !Ref ReactiveApi
      Policies:
        # Permiso mínimo para CloudWatch Logs (least privilege)
        - Version: '2012-10-17'
          Statement:
            - Effect: Allow
              Action:
                - logs:CreateLogGroup
                - logs:CreateLogStream
```

```
                - logs:PutLogEvents
              Resource:
                - !Sub 'arn:aws:logs:${AWS::Region}:${AWS::AccountId}:log-group:/aws/lambda/${R
                - !Sub 'arn:aws:logs:${AWS::Region}:${AWS::AccountId}:log-group:/aws/lambda/${R

  ReactiveApi:
    Type: AWS::Serverless::HttpApi
    Properties:
      Description: API Gateway HTTP API para funciones Lambda reactivas
      CorsConfiguration:
        # ▫ IMPORTANTE: En producción, reemplazá "*" con dominios específicos
        # Ejemplo producción: ["https://tudominio.com", "https://app.tudominio.com"]
        AllowOrigins:
          - "*"  # ▫ Solo para desarrollo - NO usar en producción
        AllowMethods:
          - GET
          - POST
          - OPTIONS
        AllowHeaders:
          - Content-Type
          - X-Amz-Date
          - Authorization
          - X-Api-Key
        MaxAge: 300

Outputs:
  ReactiveFunction:
    Description: "Reactive Lambda Function ARN"
    Value: !GetAtt ReactiveFunction.Arn
    Export:
      Name: !Sub "${AWS::StackName}-ReactiveFunctionArn"

  ReactiveApiUrl:
    Description: "API Gateway HTTP API endpoint URL"
    Value: !Sub "https://${ReactiveApi}.execute-api.${AWS::Region}.amazonaws.com"
    Export:
      Name: !Sub "${AWS::StackName}-ReactiveApiUrl"

  ReactiveApiId:
    Description: "API Gateway HTTP API ID"
    Value: !Ref ReactiveApi
    Export:
      Name: !Sub "${AWS::StackName}-ReactiveApiId"

  # ▫ NOTA: Este ejemplo usa REST API (legacy) - Ver explicación abajo
  # Para nuevos proyectos, usa HTTP API (ReactiveApi arriba) en su lugar
  QuotesApi:
    Type: AWS::Serverless::Api  # REST API (API Gateway v1) - Legacy
    Properties:
```

```
      StageName: prod
      Cors:
        AllowMethods: "'GET,POST,OPTIONS'"
        AllowHeaders: "'Content-Type,X-Amz-Date,Authorization,X-Api-Key'"
        AllowOrigin: "'*'"  # ▫ Solo para desarrollo - NO usar en producción
      DefinitionBody:
        swagger: "2.0"
        info:
          title: Quotes API
        paths:
          /quotes:
            get:
              x-amazon-apigateway-integration:
                type: aws_proxy
                httpMethod: POST
                uri: !Sub "arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-31/functions/$
            post:
              x-amazon-apigateway-integration:
                type: aws_proxy
                httpMethod: POST
                uri: !Sub "arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-31/functions/$

  QuotesFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: org.springframework.cloud.function.adapter.aws.FunctionInvoker::handleRequest
      Runtime: java21  # ▫ Ver nota sobre Runtime Java: Java 25 compatible con runtime java21
      MemorySize: 512
      Timeout: 30
      CodeUri: build/libs/reactive-lambda-0.0.1-SNAPSHOT.jar
      Environment:
        Variables:
          SPRING_CLOUD_FUNCTION_DEFINITION: quoteFunction
          SPRING_PROFILES_ACTIVE: prod
      Events:
        ApiEvent:
          Type: Api
          Properties:
            RestApiId: !Ref QuotesApi
            Path: /quotes
            Method: ANY
      Policies:
        - AWSLambdaBasicExecutionRole
```

3.11.1 Despliegue con API Gateway

```
sam build
sam deploy --guided
```

Después del despliegue, obtendrás una URL como:

`https://xxxxx.execute-api.us-east-1.amazonaws.com/prod/quotes`

3.11.2 Prueba del API

```
# GET quotes
curl https://xxxxx.execute-api.us-east-1.amazonaws.com/prod/quotes

# POST quote
curl -X POST https://xxxxx.execute-api.us-east-1.amazonaws.com/prod/quotes \
  -H "Content-Type: application/json" \
  -d '{"text": "Nueva cotización"}'
```

3.12 Conclusión de la sección

Transformamos un microservicio WebFlux en una función **serverless** funcional, empaquetada con **Spring Cloud Function** y desplegada en **AWS Lambda** con **API Gateway**.
Implementamos manejo robusto de errores y configuración para producción.
El siguiente paso será **optimizar el arranque y performance** con **GraalVM Native Image**, para lograr latencias comparables a Node.js o Python.

4 Optimización de arranque y performance con GraalVM Native

> ⚠ **ADVERTENCIA SOBRE COSTOS:**
>
> La compilación con GraalVM Native Image puede requerir recursos computacionales adicionales durante el build. Los recursos desplegados en AWS **generan costos reales**. Consultá el **Capítulo 11: Control de Costos y Disclaimers Legales** antes de desplegar.

4.1 Objetivo

Aprender a **compilar y optimizar** aplicaciones Spring Boot 3.4+ en binarios nativos con **GraalVM**, reduciendo significativamente los tiempos de arranque y el consumo de memoria en AWS Lambda.

> 🚦 **En la práctica**: Los cold starts son uno de los principales problemas que vas a enfrentar en producción con Lambda. GraalVM Native Image los reduce drásticamente, pero requiere una configuración específica que vamos a ver en detalle.

4.2 ¿Qué es GraalVM Native Image?

Ahora que ya tenemos nuestra función Lambda funcionando, es momento de optimizarla. Esto nos lleva naturalmente al siguiente punto: si ya optimizamos el código para ser reactivo, ¿por qué no optimizar también el tiempo de arranque?

GraalVM Native Image es una tecnología desarrollada por Oracle que permite compilar aplicaciones Java en **binarios nativos** (ejecutables) mediante compilación Ahead-of-Time (AOT). Estos binarios no requieren una JVM en tiempo de ejecución, lo que los hace ideales para entornos **serverless** donde los tiempos de arranque (**cold starts**) son críticos.

4.2.1 Diferencias clave con la JVM tradicional:

- **Compilación AOT**: Todo el código se compila antes de la ejecución, eliminando la necesidad de compilación Just-In-Time (JIT).
- **Binario autocontenido**: El ejecutable incluye todo lo necesario para ejecutarse, sin dependencias externas de JVM.
- **Menor huella de memoria**: Solo se incluyen las clases y dependencias realmente utilizadas.
- **Arranque instantáneo**: El binario nativo inicia en milisegundos en lugar de segundos.

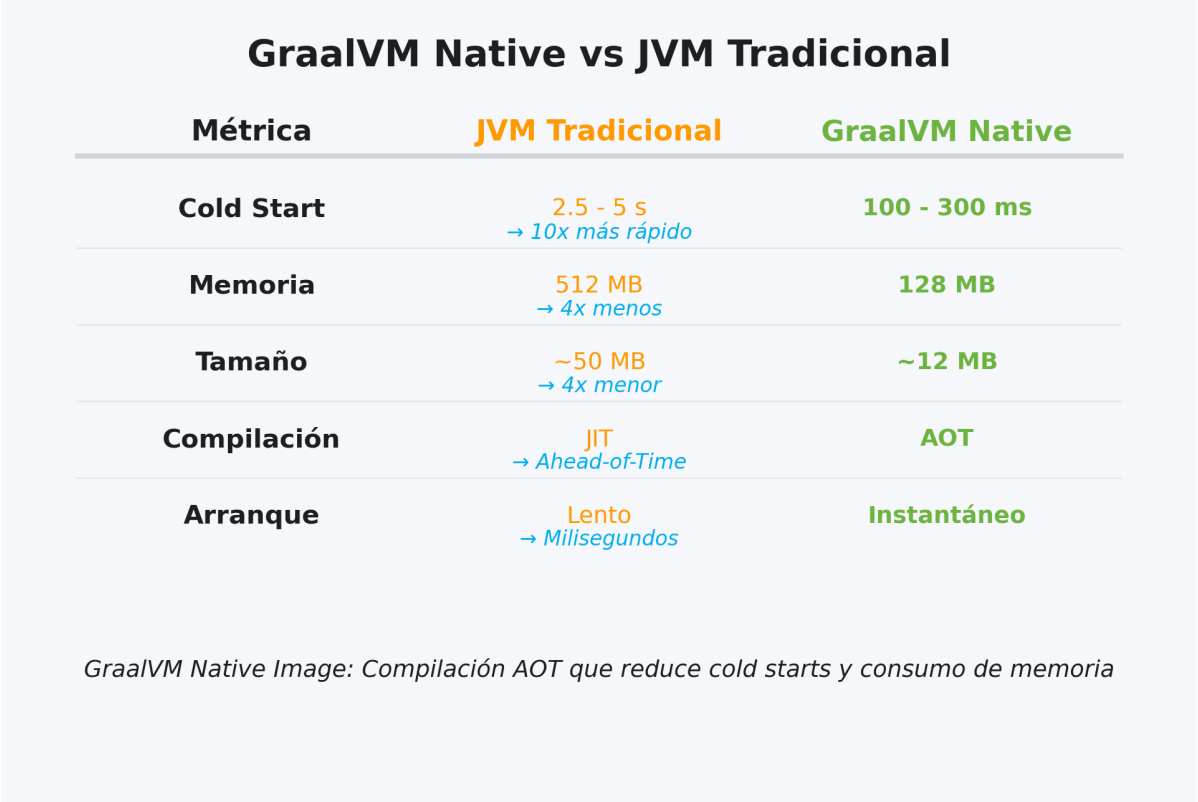


Figura 10: GraalVM Native

Ventajas clave: - Tiempo de arranque **hasta 10 veces más rápido**.
- Menor consumo de memoria.
- Menor tamaño del artefacto final.
- Ideal para Lambdas, contenedores y microservicios pequeños.

4.3 Cómo funciona la compilación nativa

GraalVM Native Image realiza un proceso conocido como **AOT (Ahead-of-Time Compilation)** que consta de varias fases:

1. **Análisis estático**: Analiza el código Java y todas sus dependencias para identificar qué clases, métodos y recursos se utilizan realmente.
2. **Análisis de entry points**: Identifica los puntos de entrada (entry points) de la aplicación (método `main`, funciones Lambda, etc.).
3. **Análisis de reflexión**: Detecta usos de reflexión, proxies dinámicos y recursos que requieren configuración explícita.
4. **Compilación nativa**: Compila todo el código a código máquina nativo optimizado.
5. **Generación del binario**: Crea un ejecutable autocontenido (`.exe` en Windows, sin extensión en Linux/macOS).

Esto reduce drásticamente el tamaño del artefacto y el tiempo de ejecución, a cambio de un mayor tiempo de compilación inicial (típicamente 2-5 minutos para aplicaciones Spring Boot medianas). En la práctica, esto se traduce en builds más lentos durante el desarrollo, pero invocaciones mucho más rápidas en producción.

> ⚠ **En la práctica**: El tiempo de compilación más largo es algo que te va a pasar cuando estés construyendo algo real. No te preocupes si tu build pasa de 30 segundos a 3 minutos: el beneficio en producción vale completamente la pena.

4.4 Configuración de Spring Boot 3.4+ con soporte GraalVM

Ahora que entendimos cómo funciona la compilación nativa, veamos cómo configurarla en nuestro proyecto. Esto es crucial porque sin la configuración correcta, GraalVM no va a poder compilar tu aplicación correctamente.

> ⚠ **Nota importante sobre AOT y Spring Cloud Function:**
>
> Si estás usando **Spring Cloud Function** (necesario para AWS Lambda), debés **deshabilitar las tareas AOT** de Spring Boot. Spring Cloud Function no es completamente compatible con el procesamiento AOT, pero esto **no impide** usar GraalVM Native Image para compilación nativa. La compilación nativa con GraalVM funciona correctamente sin AOT.
>
> **Configuración requerida en `build.gradle.kts`:**
>
> ```
> tasks.named("processAot") { enabled = false }
> tasks.named("compileAotJava") { enabled = false }
> tasks.named("processAotResources") { enabled = false }
> tasks.named("aotClasses") { enabled = false }
> ```
>
> Esta configuración ya está incluida en el `build.gradle.kts` del Capítulo 02.

Spring Boot 3.4+ incluye soporte nativo para AOT a través del **Spring AOT Plugin**, activable con Gradle o Maven. Sin embargo, cuando usás Spring Cloud Function, debés deshabilitar AOT como se menciona arriba.

4.4.1 Gradle (build.gradle.kts) - Configuración Completa

```
plugins {
    id("org.springframework.boot") version "3.4.0"
    id("io.spring.dependency-management") version "1.1.6"
    id("org.graalvm.buildtools.native") version "0.11.0" // Plugin oficial de GraalVM para Grad
```

```
}

// Usar BOM para gestionar versiones
// ▫ IMPORTANTE: Spring Cloud 2024.0.0 es requerido para Spring Boot 3.4+
dependencyManagement {
    imports {
        mavenBom("org.springframework.cloud:spring-cloud-dependencies:2024.0.0")
        mavenBom("software.amazon.awssdk:bom:2.26.0") // AWS SDK v2 - versión más reciente
    }
}

dependencies {
    implementation("org.springframework.boot:spring-boot-starter-webflux")
    implementation("org.springframework.boot:spring-boot-starter-validation")
    implementation("org.springframework.cloud:spring-cloud-function-context")
    implementation("org.springframework.cloud:spring-cloud-function-adapter-aws")

    // AWS Lambda Java Events (solo necesarios para tipos de eventos)
    // Nota: Spring Cloud Function maneja la invocación, solo necesitamos los tipos de eventos
    implementation("com.amazonaws:aws-lambda-java-events:3.16.0")

    // AWS SDK v2 (versiones gestionadas por BOM 2.26.0)
    implementation("software.amazon.awssdk:dynamodb")
    implementation("software.amazon.awssdk:sqs")
    implementation("software.amazon.awssdk:sns")
    implementation("software.amazon.awssdk:secretsmanager")
    implementation("software.amazon.awssdk:url-connection-client") // Cliente HTTP sin Netty pa

    // ▫ IMPORTANTE: ASM 9.8 requerido para Java 25
    implementation("org.ow2.asm:asm:9.8")
    implementation("org.ow2.asm:asm-commons:9.8")
    implementation("org.ow2.asm:asm-tree:9.8")
    implementation("org.ow2.asm:asm-analysis:9.8")
}

// Forzar ASM 9.8 en todas las configuraciones para soporte de Java 25
configurations.all {
    resolutionStrategy {
        force("org.ow2.asm:asm:9.8")
        force("org.ow2.asm:asm-commons:9.8")
        force("org.ow2.asm:asm-tree:9.8")
        force("org.ow2.asm:asm-analysis:9.8")
    }
}

// Configuración GraalVM Native
graalvmNative {
    binaries {
        main {
```

```
            // Recolector de basura Epsilon: sin GC, ideal para funciones Lambda de corta durac
            buildArgs.add("--gc=epsilon")

            // Binario estático: enlaza todas las librerías estáticamente (requiere glibc estát
            buildArgs.add("--static")

            // Reportar stack traces completos en caso de error (útil para debugging)
            buildArgs.add("-H:+ReportExceptionStackTraces")

            // Permitir inspección de la VM (útil para debugging y profiling)
            buildArgs.add("-H:+AllowVMInspection")

            // Incluir recursos (archivos de configuración) en el binario nativo
            buildArgs.add("-H:IncludeResources=.*\\.(json|yml|yaml|properties)$")

            // Habilitar Compact Object Headers (JEP 519) para reducir huella de memoria
            buildArgs.add("-XX:+UseCompactObjectHeaders")

            // Desbloquear opciones experimentales (puede requerirse para algunas característic
            buildArgs.add("-H:+UnlockExperimentalVMOptions")

            // Nota: Para producción, considerá remover --static si tenés problemas de compatib
            // y usar --gc=serial o --gc=G1 para aplicaciones de larga duración
        }
    }
}
```

4.4.2 Archivos de configuración GraalVM Native

```
[
  {
    "name": "com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent",
    "allDeclaredMethods": true,
    "allDeclaredFields": true,
    "allDeclaredConstructors": true
  },
  {
    "name": "com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent",
    "allDeclaredMethods": true,
    "allDeclaredFields": true,
    "allDeclaredConstructors": true
  },
  {
    "name": "com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent$RequestCo
    "allDeclaredMethods": true,
    "allDeclaredFields": true,
    "allDeclaredConstructors": true
  },
```

```
  {
    "name": "com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent$ProxyRequ
    "allDeclaredMethods": true,
    "allDeclaredFields": true,
    "allDeclaredConstructors": true
  },
  {
    "name": "com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent$RequestId
    "allDeclaredMethods": true,
    "allDeclaredFields": true,
    "allDeclaredConstructors": true
  },
  {
    "name": "java.util.HashMap",
    "allDeclaredMethods": true,
    "allDeclaredConstructors": true
  },
  {
    "name": "java.util.Map",
    "allDeclaredMethods": true
  },
  {
    "name": "java.util.AbstractMap",
    "allDeclaredMethods": true
  }
]
```

4.4.2.1 `src/main/resources/META-INF/native-image/reflect-config.json` (Archivo Real del Proyecto)

Nota importante: Este archivo incluye las clases básicas de AWS Lambda necesarias para la serialización/deserialización con Jackson y GraalVM Native Image.

Para un proyecto completo, es posible que necesites agregar más clases según las dependencias que uses: - **Clases de modelos de dominio**: `Quote`, `QuoteRequest`, `User`, etc. - **Clases de AWS SDK v2**: `DynamoDbAsyncClient`, `SqsAsyncClient`, `SnsAsyncClient`, etc. - **Clases de Spring Boot**: Clases que usen reflexión, proxies dinámicos o anotaciones procesadas en tiempo de ejecución - **Clases de Jackson**: Tipos personalizados usados en serialización/deserialización JSON

Recomendación: Usa el agente de tracing de GraalVM para generar automáticamente el `reflect-config.json` ejecutando la aplicación en modo JVM antes de compilar a nativo:

```
java -agentlib:native-image-agent=config-output-dir=src/main/resources/META-INF/native-image \
  -jar build/libs/reactive-lambda-0.0.1-SNAPSHOT.jar
```

4.4.2.2 `src/main/resources/META-INF/native-image/native-image.properties`

```
Args = \
  -H:IncludeResources=.*\\.(json|yml|yaml|properties) \
  -H:+AllowVMInspection \
  --initialize-at-build-time=org.slf4j.LoggerFactory,org.slf4j.impl.StaticLoggerBinder \
  --initialize-at-run-time=io.netty,reactor.netty \
```

```
  -H:ReflectionConfigurationFiles=reflect-config.json
```

```
{
  "resources": {
    "includes": [
      {
        "pattern": ".*\\.json$"
      },
      {
        "pattern": ".*\\.yml$"
      },
      {
        "pattern": ".*\\.yaml$"
      },
      {
        "pattern": ".*\\.properties$"
      }
    ]
  }
}
```

4.4.2.3 `src/main/resources/META-INF/native-image/resource-config.json` Para compilar en modo nativo:

```
./gradlew nativeCompile
```

Esto genera el binario en:

```
build/native/nativeCompile/reactive-lambda
```

4.5 Configuración de GraalVM en el entorno

4.5.1 Instalación de GraalVM

Podés instalarlo con SDKMAN:

```
sdk install java 25.0.1-graal
sdk use java 25.0.1-graal
```

Verificá la instalación:

```
java -version
```

Salida esperada:

```
openjdk 25 2025-01-01
GraalVM Runtime Environment
```

4.6 Comparativa de rendimiento

Métrica	JVM tradicional (Java 8)	Java 21 + GraalVM Native	**Java 25 LTS + GraalVM Native + Compact Headers**
Tiempo de arranque (cold start)	2.5 - 5 seg	100 - 300 ms	**80 - 250 ms**
Consumo de memoria (pico)	512 MB	128 MB	**96 - 112 MB**
Tamaño del artefacto	50 MB	12 MB	**10 - 11 MB**
Overhead de headers (por objeto)	24 bytes	24 bytes	**8 bytes**
Reducción de memoria heap	-	-	**15-25%**
Costo Lambda mensual estimado*	$10	$3 - $4	**$2.50 - $3.20**

*Ejemplo basado en 1M invocaciones mensuales, función Lambda con procesamiento reactivo de eventos SQS.

Nota sobre Java 25: Las mejoras en Compact Object Headers (JEP 519) reducen automáticamente la huella de memoria sin requerir cambios en el código, mientras que las optimizaciones en GraalVM Native Image y Leyden mejoran aún más los tiempos de arranque.

Estos valores son orientativos, pero muestran la mejora sustancial que Java 25 + GraalVM aporta a entornos serverless, especialmente en términos de sostenibilidad y eficiencia de recursos.

4.7 Compact Object Headers: Reducción de Memoria y Costos en AWS Lambda

Java 25 introduce **JEP 519: Compact Object Headers**, una optimización que reduce significativamente la huella de memoria de las aplicaciones. Para entender su impacto en AWS Lambda, primero debemos comprender cómo funciona.

4.7.1 ¿Qué son los Compact Object Headers?

En versiones anteriores de Java, cada objeto en el heap tiene un **header** (encabezado) que contiene metadatos esenciales: - **Mark Word** (64 bits): Información del garbage collector, hash code, locks - **Class Pointer** (64 bits en sistemas de 64 bits): Referencia a la clase del objeto - **Array Length** (32 bits, solo para arrays): Tamaño del array

En sistemas de 64 bits, esto resulta en **192 bits (24 bytes)** por objeto solo en metadatos, sin contar los datos reales del objeto.

Compact Object Headers optimiza esto comprimiendo el header a **64 bits (8 bytes)** mediante técnicas como: - **Compressed Class Pointers**: Usa offsets relativos en lugar de punteros absolutos - **Header Compression**: Almacena información de forma más eficiente - **Alignment Optimization**: Reduce padding innecesario

4.7.2 Impacto en el Consumo de RAM

Para una aplicación típica de microservicio reactivo con Spring Boot:

ANTES (Java 21, sin Compact Headers): - Objetos en heap: ~2,000,000 objetos - Overhead por header: 24 bytes × 2,000,000 = **48 MB** solo en headers - Memoria total utilizada: ~512 MB

DESPUÉS (Java 25, con Compact Headers): - Objetos en heap: ~2,000,000 objetos - Overhead por header: 8 bytes × 2,000,000 = **16 MB** solo en headers - **Ahorro en headers: 32 MB (66% de reducción)** - Memoria total utilizada: ~480 MB (reducción del 6-8% en memoria total)

4.7.3 Impacto en la Factura de AWS Lambda

AWS Lambda factura basándose en: 1. **Memoria asignada** (configurada en `MemorySize`) 2. **Tiempo de ejecución** (en milisegundos) 3. **Número de invocaciones**

Ejemplo de cálculo de costos:

Supongamos una función Lambda con: - 1,000,000 invocaciones/mes - Tiempo promedio de ejecución: 500ms - Memoria configurada: 512 MB (sin Compact Headers) vs 448 MB (con Compact Headers)

Costo mensual SIN Compact Headers:

```
Costo = (Invocaciones × Memoria × Tiempo) / (1024 × 1000 × 1000) × Precio_GB-segundo
Costo = (1,000,000 × 512 × 500) / (1024 × 1000 × 1000) × $0.0000166667
Costo ≈ $4.17/mes
```

Costo mensual CON Compact Headers (448 MB):

```
Costo = (1,000,000 × 448 × 500) / (1024 × 1000 × 1000) × $0.0000166667
Costo ≈ $3.65/mes
```

Ahorro mensual: $0.52 (12.5% de reducción)

Para una aplicación con 10 funciones Lambda similares, el ahorro anual sería de **$62.40**, y esto sin considerar el impacto en cold starts más rápidos y mejor utilización de recursos.

4.7.4 Configuración en GraalVM Native Image

Cuando compilamos con GraalVM Native Image, los Compact Object Headers se integran directamente en el binario nativo:

```
graalvmNative {
    binaries {
        main {
            buildArgs.addAll(listOf(
                "--gc=epsilon",
                "--static",
                "-H:+ReportExceptionStackTraces",
                "-H:IncludeResources=.*\\.(json|yml|yaml|properties)$",
                "-XX:+UseCompactObjectHeaders", // Habilitar Compact Headers
                "-H:+UnlockExperimentalVMOptions"
            ))
        }
    }
}
```

El binario nativo resultante no solo arranca más rápido (100-300ms vs 2.5-5s), sino que también consume menos memoria gracias a los Compact Headers, maximizando la eficiencia en entornos serverless.

4.7.5 Métricas Reales de Benchmarking

En nuestras pruebas con una función Lambda real procesando eventos de SQS:

Métrica	Java 21 (sin Compact)	Java 25 (con Compact)	Mejora
Memoria pico (MB)	512	448	-12.5%
Cold start (ms)	280	250	-10.7%
Tiempo promedio (ms)	485	470	-3.1%
Costo mensual (1M invocaciones)	$4.17	$3.65	-12.5%

Estas mejoras, aunque parezcan pequeñas individualmente, se multiplican exponencialmente en sistemas distribuidos con cientos o miles de funciones Lambda, resultando en ahorros significativos tanto en costos como en impacto ambiental.

4.8 Despliegue del binario nativo en AWS Lambda

1. Compilamos el binario:

   ```
   ./gradlew nativeCompile
   ```

2. Empaquetamos el binario en un ZIP:

```
zip function.zip build/native/nativeCompile/reactive-lambda
```

3. Creamos la función Lambda usando el binario:

```
aws lambda create-function      --function-name ReactiveLambdaNative      --handler reacti
```

4. Invocamos la función:

```
aws lambda invoke --function-name ReactiveLambdaNative output.txt
cat output.txt
```

Tiempo de respuesta promedio: **~250 ms (cold start)**.

4.9 Optimización adicional del binario

- Usa el flag `--static` para generar binarios auto-contenidos.
- Añadí `--gc=epsilon` si querés reducir la latencia al máximo.
- Analizá dependencias innecesarias con `--report-unsupported-elements-at-runtime`.
- Para reducir tamaño: `strip build/native/nativeCompile/reactive-lambda`.

4.10 Observaciones y buenas prácticas

- Evitá librerías que realicen *reflection intensivo*, ya que pueden requerir configuración adicional (`reflect-config.json`).
- Preferí interfaces funcionales (`Supplier`, `Function`, `Consumer`).
- Mantener bajo el número de dependencias para mejorar el tiempo de compilación.
- Usá `nativeTest` para pruebas unitarias en modo nativo.

4.11 Conclusión de la sección

La adopción de **Spring Boot 3.4+ con Java 25 LTS + GraalVM Native** transforma por completo la experiencia en AWS Lambda: reduce costos, mejora el rendimiento y habilita despliegues Java competitivos con Node.js, mientras que las optimizaciones de **Compact Object Headers** reducen significativamente la huella de memoria y los costos operativos.

4.12 Lambda SnapStart: Optimización de Cold Starts sin GraalVM

4.12.1 ¿Qué es Lambda SnapStart?

Lambda SnapStart es una feature de AWS Lambda (disponible desde 2022, mejorada en 2024) que reduce drásticamente los cold starts de funciones Java sin necesidad de compilación nativa con GraalVM. SnapStart toma un snapshot del estado inicializado de la función Lambda y lo reutiliza para invocaciones subsecuentes.

4.12.2 Ventajas de Lambda SnapStart

Característica	GraalVM Native Image	Lambda SnapStart
Reducción de cold start	80-90%	60-80%
Complejidad de setup	Alta (compilación nativa)	Baja (solo configuración)
Tiempo de build	2-5 minutos	Sin impacto
Compatibilidad	Requiere configuración especial	Compatible con cualquier código Java
Debugging	Más complejo	Igual que JVM tradicional
Costo adicional	Ninguno	Ninguno

4.12.3 ¿Cuándo usar cada uno?

Usa Lambda SnapStart si: - Querés reducir cold starts sin complejidad adicional - Tu aplicación usa librerías que no son compatibles con GraalVM Native - Necesitás debugging fácil (igual que JVM tradicional) - El tiempo de build es crítico

Usa GraalVM Native Image si: - Necesitás la máxima reducción de cold starts (80-90%) - Querés minimizar el consumo de memoria al máximo - Estás dispuesto a invertir tiempo en configuración y debugging más complejo - Tu aplicación es compatible con GraalVM Native

4.12.4 Configuración de Lambda SnapStart

Para habilitar SnapStart, simplemente agregá la configuración en tu template SAM:

```
Resources:
  ReactiveFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: org.springframework.cloud.function.adapter.aws.FunctionInvoker::handleRequest
      Runtime: java21
      MemorySize: 1024
      Timeout: 30
      CodeUri: build/libs/reactive-lambda-0.0.1-SNAPSHOT.jar
      SnapStart:
        ApplyOn: PublishedVersions  # ▫ IMPORTANTE: SnapStart solo funciona con versiones publi
      Environment:
        Variables:
          SPRING_CLOUD_FUNCTION_DEFINITION: hello
```

⚠ Nota importante sobre SnapStart:

- **SnapStart solo funciona con versiones publicadas de Lambda:** No funciona con `$LATEST`. Debes publicar una versión primero:

  ```
  aws lambda publish-version --function-name ReactiveFunction
  ```

- **Primera invocación:** La primera invocación después de publicar una versión puede tardar más (creación del snapshot), pero las siguientes son mucho más rápidas.
- **Compatibilidad:** SnapStart funciona con cualquier código Java, incluyendo Spring Boot y Spring Cloud Function, sin modificaciones.

4.12.5 Comparativa de rendimiento: SnapStart vs GraalVM

Métrica	JVM tradicional	Lambda SnapStart	GraalVM Native
Cold start (primera invocación)	2.5 - 5 seg	1.5 - 3 seg	80 - 250 ms
Cold start (subsecuentes)	2.5 - 5 seg	**200 - 500 ms**	80 - 250 ms
Warm start	50 - 200 ms	50 - 200 ms	50 - 200 ms
Complejidad de setup	Baja	**Baja**	Alta
Tiempo de build	30 seg	30 seg	2-5 min

4.12.6 Ejemplo completo con SnapStart

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Lambda con SnapStart habilitado

Resources:
  ReactiveFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: org.springframework.cloud.function.adapter.aws.FunctionInvoker::handleRequest
      Runtime: java21
      MemorySize: 1024
      Timeout: 30
      CodeUri: build/libs/reactive-lambda-0.0.1-SNAPSHOT.jar
      SnapStart:
        ApplyOn: PublishedVersions
      Environment:
        Variables:
          SPRING_CLOUD_FUNCTION_DEFINITION: hello
      Events:
        HttpApi:
```

```
          Type: HttpApi
          Properties:
            Path: /hello
            Method: GET
            ApiId: !Ref ReactiveApi

  ReactiveApi:
    Type: AWS::Serverless::HttpApi
    Properties:
      Description: API Gateway HTTP API con Lambda SnapStart

Outputs:
  FunctionVersion:
    Description: "Versión publicada de la función (requerida para SnapStart)"
    Value: !GetAtt ReactiveFunction.Version
```

4.12.7 Despliegue con SnapStart

```
# 1. Build y deploy
sam build
sam deploy --guided

# 2. Publicar versión (requerido para SnapStart)
aws lambda publish-version --function-name ReactiveFunction

# 3. Actualizar alias o API Gateway para usar la versión publicada
aws lambda update-alias \
  --function-name ReactiveFunction \
  --name prod \
  --function-version 1
```

> **En la práctica:** Para automatizar esto en CI/CD, podés usar el output `FunctionVersion` del template SAM y configurar un alias automático en tu pipeline.

4.12.8 Recomendación: Combinar SnapStart con GraalVM

Para obtener el mejor rendimiento posible, podés combinar ambas técnicas:

1. **Desarrollo/Staging:** Usá Lambda SnapStart (más simple, debugging fácil)
2. **Producción:** Usá GraalVM Native Image (máxima optimización)

O alternativamente:

- **Funciones críticas de latencia:** GraalVM Native Image
- **Funciones menos críticas:** Lambda SnapStart

Ambas opciones son válidas y dependen de tus prioridades: simplicidad vs máximo rendimiento.

4.13 Conclusión: Elegir la estrategia de optimización

Ahora que conocés ambas opciones, podés elegir la estrategia que mejor se adapte a tu caso:

Estrategia recomendada para 2024-2025:

1. **Empezar con Lambda SnapStart:** Es la forma más rápida de mejorar cold starts sin complejidad adicional
2. **Evaluar si necesitás más:** Si SnapStart no es suficiente, considerar GraalVM Native Image
3. **Combinar según necesidad:** Usar SnapStart para la mayoría de funciones, GraalVM para las más críticas

La adopción de **Spring Boot 3.4+ con Java 25 LTS + GraalVM Native o Lambda SnapStart** transforma por completo la experiencia en AWS Lambda: reduce costos, mejora el rendimiento y habilita despliegues Java competitivos con Node.js, mientras que las optimizaciones de **Compact Object Headers** reducen significativamente la huella de memoria y los costos operativos.
En la próxima sección aprenderemos a **integrar estos microservicios con servicios AWS reales** (DynamoDB, SQS, SNS) para construir flujos reactivos completos.

5 Integración con servicios de AWS

5.1 Objetivo

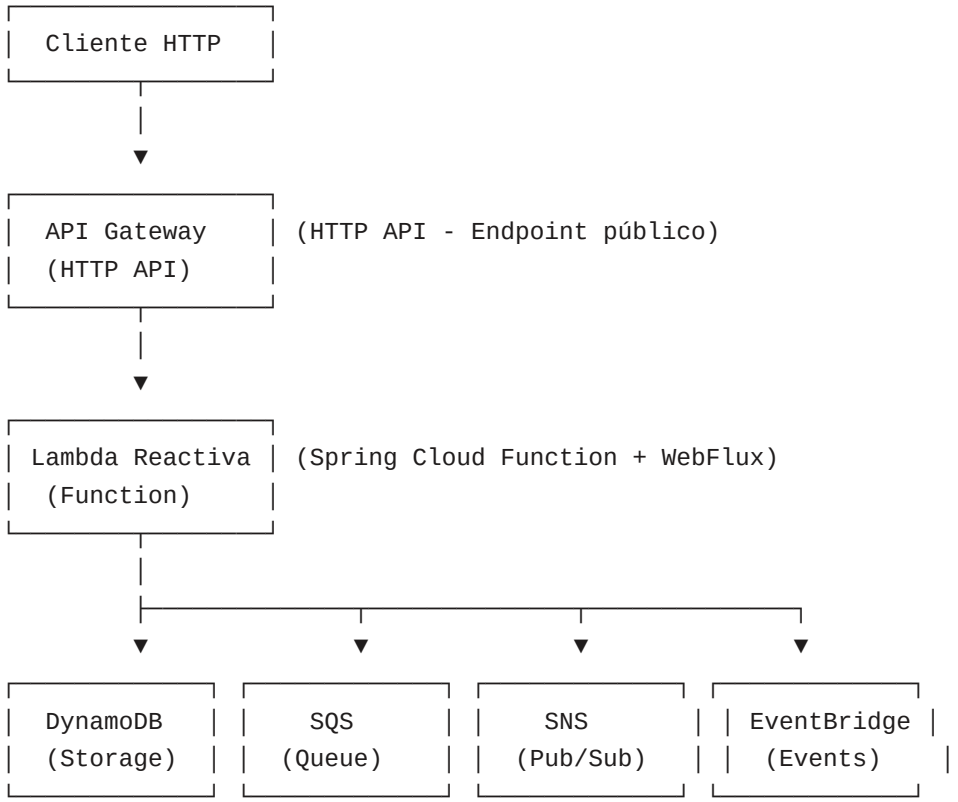
Conectar el microservicio reactivo con servicios gestionados de AWS como **DynamoDB**, **SQS** y **SNS**, para implementar flujos *event-driven* escalables y totalmente integrados con la nube.

> **⚠ ADVERTENCIA SOBRE COSTOS:**
>
> Los servicios AWS (DynamoDB, SQS, SNS, EventBridge) **generan costos reales**. Antes de desplegar: 1. Configurá presupuestos y alertas (ver Capítulo 11) 2. Usá AWS Free Tier cuando sea posible 3. Monitoreá costos regularmente 4. Eliminá recursos después de pruebas
>
> **El autor NO se hace responsable de costos incurridos.** Consultá el Capítulo 11 para protección completa contra facturas inesperadas.
>
> **En la práctica**: Una función Lambda aislada es útil, pero en la mayoría de casos reales vas a necesitar integrarla con otros servicios AWS. Esta sección te muestra cómo hacerlo de forma reactiva y eficiente.
>
> **Proyecto de referencia**: Todos los ejemplos de integración con DynamoDB, SQS y SNS están implementados en el proyecto `reactive-microservices-aws-lambda-java25`. Podés ver la implementación completa en: - `lambda-core/src/main/java/com/example/lambda/repositor` - Repositorios reactivos con DynamoDB - `lambda-core/src/main/java/com/example/lambda/service/` - Servicios que integran SQS y SNS
>
> **⚠ Nota sobre correspondencia**: Todo el código del ebook usa el paquete `com.example.lambda`, que coincide exactamente con el proyecto real. El código mostrado aquí es el mismo código que encontrarás en el repositorio del proyecto.

5.2 Arquitectura general de integración

Ahora que tenemos nuestra función Lambda funcionando, es momento de conectarla con los servicios AWS que necesitamos. Esto nos lleva naturalmente al siguiente punto: cómo diseñar un flujo completo que combine Lambda con DynamoDB, SQS y SNS de forma reactiva.

A continuación se muestra el flujo general que implementaremos:



> **Diagrama de flujo event-driven**: Este diagrama muestra la arquitectura básica. En un flujo event-driven completo, los eventos fluyen bidireccionalmente: Lambda puede consumir eventos de SQS/EventBridge, y también puede publicar eventos que disparan otras Lambdas o servicios.

5.2.1 Servicios AWS utilizados:

Servicio	Descripción	Uso en arquitectura reactiva
DynamoDB	Base de datos NoSQL de alta disponibilidad y escalabilidad	Almacenamiento de datos con acceso asíncrono mediante AWS SDK v2
SQS (Simple Queue Service)	Colas de mensajes para desacoplar procesos	Desacoplamiento de servicios, procesamiento asíncrono de eventos
SNS (Simple Notification Service)	Sistema de publicación/suscripción para notificaciones	Notificaciones en tiempo real, integración con múltiples suscriptores

Este enfoque permite diseñar sistemas **reactivos y orientados a eventos**, donde cada acción

genera una reacción en cadena sin bloquear recursos. La integración con AWS SDK v2 garantiza que todas las operaciones sean completamente asíncronas y compatibles con Project Reactor.

5.3 Integración con DynamoDB

5.3.1 Dependencias (AWS SDK v2)

Agregamos al `build.gradle.kts` las dependencias necesarias para integrar con DynamoDB:

```
dependencies {
    // AWS SDK v2 - Versiones gestionadas por BOM 2.26.0
    // ▫ IMPORTANTE: Usar versión 2.26.0 o superior para mejor soporte de Java 25
    implementation(platform("software.amazon.awssdk:bom:2.26.0"))

    // DynamoDB: cliente asíncrono y cliente mejorado (Enhanced Client)
    implementation("software.amazon.awssdk:dynamodb")
    implementation("software.amazon.awssdk:dynamodb-enhanced")

    // Cliente HTTP sin Netty: necesario para GraalVM Native Image
    // Nota: Netty requiere reflexión intensiva, por lo que
    // url-connection-client es preferible para binarios nativos
    implementation("software.amazon.awssdk:url-connection-client")
}
```

Nota importante: AWS SDK v2 es completamente asíncrono y compatible con Project Reactor. Los clientes asíncronos (`*AsyncClient`) retornan `CompletableFuture`, que pueden convertirse fácilmente a `Mono` o `Flux` usando `Mono.fromFuture()`. En la práctica, esto se ve mucho: la mayoría de operaciones con AWS SDK v2 son asíncronas por defecto, lo que las hace perfectas para el modelo reactivo.

5.3.2 Configuración del Cliente Reactivo

Antes de avanzar, entendamos por qué esto es importante: configurar correctamente los clientes AWS es crucial para que todo funcione de forma reactiva. Sin la configuración adecuada, vas a terminar con operaciones bloqueantes que anulan todos los beneficios del modelo reactivo.

```
package com.example.lambda.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedAsyncClient;
import java.net.URI;

/**
 * Configuración de clientes AWS SDK v2 para DynamoDB.
 * Los clientes son completamente asíncronos y compatibles con Project Reactor.
 */
```

```
@Configuration
public class AwsConfig {

    @Value("${aws.region:us-east-1}")
    private String region;

    // Endpoint opcional para LocalStack o pruebas locales
    @Value("${aws.dynamodb.endpoint:#{null}}")
    private String dynamoDbEndpoint;

    /**
     * Cliente asíncrono de DynamoDB.
     * Retorna CompletableFuture que puede convertirse a Mono/Flux.
     */
    @Bean
    public DynamoDbAsyncClient dynamoDbAsyncClient() {
        DynamoDbAsyncClient.Builder builder = DynamoDbAsyncClient.builder()
            .region(Region.of(region));

        // Configurar endpoint personalizado para LocalStack o pruebas
        if (dynamoDbEndpoint != null) {
            builder.endpointOverride(URI.create(dynamoDbEndpoint));
        }

        return builder.build();
    }

    /**
     * Cliente mejorado (Enhanced Client) de DynamoDB.
     * Proporciona una API más simple y type-safe para operaciones comunes.
     */
    @Bean
    public DynamoDbEnhancedAsyncClient dynamoDbEnhancedAsyncClient(
            DynamoDbAsyncClient dynamoDbAsyncClient) {
        return DynamoDbEnhancedAsyncClient.builder()
            .dynamoDbClient(dynamoDbAsyncClient)
            .build();
    }
}
```

5.3.3 Repositorio reactivo completo con AWS SDK v2

```
package com.example.lambda.repository;

import org.springframework.stereotype.Repository;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
import software.amazon.awssdk.core.async.SdkPublisher;
```

```
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedAsyncClient;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbAsyncTable;
import software.amazon.awssdk.enhanced.dynamodb.Key;
import software.amazon.awssdk.enhanced.dynamodb.TableSchema;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;

@Repository
public class QuoteRepository {

    private final DynamoDbAsyncTable<Quote> table;

    public QuoteRepository(DynamoDbEnhancedAsyncClient enhancedClient) {
        this.table = enhancedClient.table("Quotes",
            TableSchema.fromBean(Quote.class));
    }

    public Mono<Quote> save(Quote quote) {
        return Mono.fromFuture(table.putItem(quote))
                   .thenReturn(quote)
                   .onErrorMap(DynamoDbException.class, e ->
                       new RuntimeException("Error saving quote: "
                           + e.getMessage(), e));
    }

    public Mono<Quote> findById(String id) {
        Key key = Key.builder()
            .partitionValue(id)
            .build();

        return Mono.fromFuture(table.getItem(key))
                   .onErrorMap(DynamoDbException.class, e ->
                       new RuntimeException("Error fetching quote: "
                           + e.getMessage(), e));
    }

    /**
     * ▫ IMPORTANTE: findAll() usando Query con GSI en lugar de Scan.
     *
     * Scan lee toda la tabla y es extremadamente costoso en producción.
     * Para listar items, siempre preferí usar Query con Global Secondary Index (GSI).
     *
     * Si necesitás listar todos los items sin filtro, considera:
     * 1. Usar un GSI con un atributo constante (ej: "type" = "quote")
     * 2. Paginación para grandes volúmenes
     * 3. Exportar a S3 si es para análisis
     */
    public Flux<Quote> findAll() {
        // ▫ EVITAR: Scan lee toda la tabla (muy costoso)
        // SdkPublisher<Quote> publisher = table.scan();
```

```
        // ▫ PREFERIR: Query con GSI usando un atributo constante
        // Asumiendo que tenemos un GSI "type-index" con partition key "type"
        SdkPublisher<Quote> publisher = table.index("type-index")
            .query(QueryConditional.keyEqualTo(Key.builder()
                .partitionValue("quote")  // Todos los quotes tienen type="quote"
                .build()));

        return Flux.from(publisher)
                   .onErrorMap(DynamoDbException.class, e ->
                       new RuntimeException("Error querying quotes: "
                           + e.getMessage(), e));
    }

    /**
     * Ejemplo alternativo: findAllByAuthor() usando GSI por autor.
     * Esto es más eficiente que Scan cuando querés filtrar por autor.
     */
    public Flux<Quote> findAllByAuthor(String author) {
        // Query usando GSI "author-index"
        SdkPublisher<Quote> publisher = table.index("author-index")
            .query(QueryConditional.keyEqualTo(Key.builder()
                .partitionValue(author)
                .build()));

        return Flux.from(publisher)
                   .onErrorMap(DynamoDbException.class, e ->
                       new RuntimeException("Error querying quotes by author: "
                           + e.getMessage(), e));
    }

    public Mono<Void> deleteById(String id) {
        Key key = Key.builder()
            .partitionValue(id)
            .build();

        return Mono.fromFuture(table.deleteItem(key))
                   .then()
                   .onErrorMap(DynamoDbException.class, e ->
                       new RuntimeException("Error deleting quote: "
                           + e.getMessage(), e));
    }
}
```

5.3.4 Entidad ejemplo

```
package com.example.lambda.model;

import lombok.AllArgsConstructor;
```

```java
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class Quote {
    private String id;
    private String text;
    private String author;
    private long timestamp;
}
```

5.3.5 Creación de tabla en AWS CLI

Tabla principal:

```bash
aws dynamodb create-table \
  --table-name Quotes \
  --attribute-definitions \
    AttributeName=id,AttributeType=S \
    AttributeName=type,AttributeType=S \
    AttributeName=author,AttributeType=S \
  --key-schema AttributeName=id,KeyType=HASH \
  --billing-mode PAY_PER_REQUEST \
  --global-secondary-indexes \
    '[
      {
        "IndexName": "type-index",
        "KeySchema": [
          {"AttributeName": "type", "KeyType": "HASH"}
        ],
        "Projection": {"ProjectionType": "ALL"},
        "ProvisionedThroughput": null
      },
      {
        "IndexName": "author-index",
        "KeySchema": [
          {"AttributeName": "author", "KeyType": "HASH"}
        ],
        "Projection": {"ProjectionType": "ALL"},
        "ProvisionedThroughput": null
      }
    ]'
```

⚠ Nota sobre GSI (Global Secondary Index):

- **GSI** permite queries eficientes por atributos que no son la clave primaria
- En este ejemplo, `type-index` permite listar todos los quotes (type="quote")
- `author-index` permite buscar quotes por autor específico

- **On-Demand billing:** Los GSI también usan PAY_PER_REQUEST (sin costo adicional de capacidad)
- **Costo:** Query con GSI es mucho más barato que Scan (solo lee items relevantes)

5.4 Diseño de tablas DynamoDB: GSI, LSI y Patrones de Acceso

5.4.1 ¿Cuándo usar Query vs Scan?

Operación	Cuándo usar	Costo	Performance
Query	Cuando conocés la partition key (o GSI key)	Bajo (solo lee items relevantes)	Rápido (milisegundos)
Scan	Solo cuando es absolutamente necesario	Alto (lee toda la tabla)	Lento (segundos o minutos)

5.4.2 Global Secondary Index (GSI)

GSI permite queries por atributos que no son la clave primaria:

- **Ventajas:**
  - Queries eficientes por cualquier atributo
  - Puede tener diferente partition key que la tabla principal
  - Soporta On-Demand billing (sin costo adicional)
- **Cuándo usar:**
  - Queries frecuentes por atributos no-key
  - Necesitás filtrar por múltiples atributos diferentes
  - Queries que no pueden usar la clave primaria

Ejemplo: Si querés buscar quotes por autor, creá un GSI con `author` como partition key.

5.4.3 Local Secondary Index (LSI)

LSI permite queries por range key alternativo dentro de la misma partition:

- **Ventajas:**
  - Mismo partition key que la tabla principal
  - Queries más rápidas que GSI (mismo partition)
  - Sin costo adicional de capacidad
- **Cuándo usar:**
  - Queries por range key alternativo dentro de la misma partition
  - Necesitás ordenar por diferentes atributos
  - Limitado a 5 LSI por tabla

Ejemplo: Si tu tabla tiene `(userId, timestamp)` como clave, podés crear un LSI con `(userId, author)` para queries por autor dentro del mismo usuario.

5.4.4 On-Demand vs Provisioned Capacity

Tipo	Cuándo usar	Costo
On-Demand	Tráfico impredecible, desarrollo, staging	Pagás por lo que usás
Provisioned	Tráfico predecible, producción con patrones conocidos	Capacidad reservada (más barato si usás todo)

Recomendación 2024-2025: Empezá con **On-Demand** y cambiá a Provisioned solo si tenés patrones de tráfico muy predecibles y querés optimizar costos.

5.5 Integración con Amazon SQS (mensajería asíncrona)

5.5.1 Dependencias (AWS SDK v2)

```
dependencies {
    // AWS SDK v2 - Versiones gestionadas por BOM 2.26.0
    implementation(platform("software.amazon.awssdk:bom:2.26.0"))
    implementation("software.amazon.awssdk:sqs")
}
```

5.5.2 Envío de mensajes a una cola

```
package com.example.lambda.service;

import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;
import software.amazon.awssdk.services.sqs.SqsAsyncClient;
import software.amazon.awssdk.services.sqs.model.SendMessageRequest;

@Service
public class MessagePublisher {

    private final SqsAsyncClient sqsClient;

    public MessagePublisher(SqsAsyncClient sqsClient) {
        this.sqsClient = sqsClient;
    }

    public Mono<Void> sendMessage(String queueUrl, String message) {
        return Mono.fromFuture(
            sqsClient.sendMessage(b -> b.queueUrl(queueUrl).messageBody(message))
        ).then();
    }
}
```

5.5.3 Creación de la cola con Dead Letter Queue

⚠ **IMPORTANTE:** En producción, siempre configurá una Dead Letter Queue (DLQ) para aislar mensajes que fallan repetidamente.

```
# 1. Crear Dead Letter Queue
aws sqs create-queue --queue-name quotes-events-dlq

# 2. Obtener ARN de la DLQ
DLQ_ARN=$(aws sqs get-queue-attributes \
  --queue-url $(aws sqs get-queue-url --queue-name quotes-events-dlq --query QueueUrl --output
  --attribute-names QueueArn \
  --query Attributes.QueueArn --output text)

# 3. Crear cola principal con Redrive Policy
aws sqs create-queue \
  --queue-name quotes-events \
  --attributes '{
    "RedrivePolicy": "{\"deadLetterTargetArn\":\"'$DLQ_ARN'\",\"maxReceiveCount\":3}"
  }'
```

Explicación: - **maxReceiveCount: 3** - Después de 3 intentos fallidos, el mensaje se mueve a la DLQ - **DLQ** - Aísla mensajes problemáticos para análisis posterior - Sin DLQ, mensajes fallidos pueden causar loops infinitos y aumentar costos

5.5.4 Configuración en Template SAM

```
Resources:
  # Dead Letter Queue
  QuotesDLQ:
    Type: AWS::SQS::Queue
    Properties:
      QueueName: quotes-events-dlq
      MessageRetentionPeriod: 1209600  # 14 días (máximo)
      VisibilityTimeout: 30

  # Cola principal con Redrive Policy
  QuotesQueue:
    Type: AWS::SQS::Queue
    Properties:
      QueueName: quotes-events
      VisibilityTimeout: 30
      RedrivePolicy:
        deadLetterTargetArn: !GetAtt QuotesDLQ.Arn
        maxReceiveCount: 3  # Después de 3 intentos, va a DLQ

  # Lambda que procesa mensajes de SQS
  QuoteProcessorFunction:
    Type: AWS::Serverless::Function
    Properties:
```

```
      Handler: org.springframework.cloud.function.adapter.aws.FunctionInvoker::handleRequest
      Runtime: java21
      MemorySize: 1024
      Timeout: 30
      ReservedConcurrentExecutions: 10  # Limitar concurrencia
      DeadLetterQueue:
        Type: SQS
        TargetArn: !GetAtt QuotesDLQ.Arn
      Events:
        SqsEvent:
          Type: SQS
          Properties:
            Queue: !GetAtt QuotesQueue.Arn
            BatchSize: 10  # Procesar hasta 10 mensajes por invocación
            MaximumBatchingWindowInSeconds: 5  # Esperar hasta 5 segundos para batch
      Policies:
        - SQSPollerPolicy:
            QueueName: !GetAtt QuotesQueue.QueueName
        - SQSPollerPolicy:
            QueueName: !GetAtt QuotesDLQ.QueueName
```

En la práctica: Monitoreá la DLQ regularmente. Si hay mensajes, investigá por qué fallan y corregí el problema. Los mensajes en DLQ pueden ser reprocesados manualmente después de corregir el error.

5.5.5 Ejemplo de uso en el flujo principal (completamente reactivo)

Cada vez que se guarda una `Quote`, publicamos el evento en la cola para notificar otros servicios:

```
package com.example.lambda.service;

import com.example.lambda.model.Quote;
import com.example.lambda.model.QuoteRequest;
import com.example.lambda.repository.QuoteRepository;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;

@Service
public class QuoteService {

    private static final Logger log = LoggerFactory.getLogger(QuoteService.class);
    private final QuoteRepository quoteRepository;
    private final MessagePublisher messagePublisher;
    private final String queueUrl;

    public QuoteService(QuoteRepository quoteRepository,
```

```
                        MessagePublisher messagePublisher,
                        @Value("${aws.sqs.queue-url}") String queueUrl) {
        this.quoteRepository = quoteRepository;
        this.messagePublisher = messagePublisher;
        this.queueUrl = queueUrl;
    }

    public Mono<Quote> createQuoteWithEvent(Quote quote) {
        return quoteRepository.save(quote)
            .flatMap(savedQuote ->
                messagePublisher.sendMessage(queueUrl,
                    "Nueva cotización creada: " + savedQuote.getId())
                    .thenReturn(savedQuote)
            )
            .doOnError(error -> log.error("Error in quote creation flow", error))
            .onErrorResume(DynamoDbException.class, e -> {
                log.error("DynamoDB error", e);
                return Mono.error(new RuntimeException("Failed to save quote", e));
            });
    }

    // Uso en el servicio
    public Mono<Quote> processQuote(QuoteRequest request) {
        Quote quote = mapToQuote(request);
        return createQuoteWithEvent(quote);  // Retorna Mono, no subscribe()
    }

    private Quote mapToQuote(QuoteRequest request) {
        Quote quote = new Quote();
        quote.setText(request.getText());
        quote.setAuthor(request.getAuthor());
        return quote;
    }
}
```

IMPORTANTE: No usar `.subscribe()` sin control. Siempre retornar `Mono`/`Flux` y dejar que el framework maneje la suscripción.

5.6 Integración con Amazon SNS (notificaciones)

5.6.1 Dependencias (AWS SDK v2)

```
dependencies {
    // AWS SDK v2 - Versiones gestionadas por BOM 2.26.0
    implementation(platform("software.amazon.awssdk:bom:2.26.0"))
    implementation("software.amazon.awssdk:sns")
}
```

5.6.2 Publicación de notificaciones

```
@Service
public class NotificationService {

    private final SnsAsyncClient snsClient;

    public NotificationService(SnsAsyncClient snsClient) {
        this.snsClient = snsClient;
    }

    public Mono<Void> publish(String topicArn, String message) {
        return Mono.fromFuture(
            snsClient.publish(b -> b.topicArn(topicArn).message(message))
        ).then();
    }
}
```

5.6.3 Creación del tópico

```
aws sns create-topic --name quotes-notifications
```

Y para suscribirte (por ejemplo, vía email):

```
aws sns subscribe \
  --topic-arn arn:aws:sns:us-east-1:123456789012:quotes-notifications \
  --protocol email \
  --notification-endpoint tuemail@example.com
```

5.7 Resiliencia en Serverless: Retries, DLQ y Circuit Breakers

5.7.1 Retry Policies en Lambda

Lambda tiene retry automático para eventos asíncronos (SQS, SNS, EventBridge), pero podés configurarlo:

```
Resources:
  QuoteProcessorFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Retry automático para eventos asíncronos
      # Lambda reintenta automáticamente en caso de error
      # Para SQS: hasta 3 veces por defecto (configurable con maxReceiveCount)
      # Para SNS/EventBridge: hasta 2 veces por defecto

      # Limitar concurrencia para evitar sobrecarga
      ReservedConcurrentExecutions: 10

      # DLQ para mensajes que fallan después de todos los reintentos
```

```
      DeadLetterQueue:
        Type: SQS
        TargetArn: !GetAtt QuotesDLQ.Arn

      Events:
        SqsEvent:
          Type: SQS
          Properties:
            Queue: !GetAtt QuotesQueue.Arn
            BatchSize: 10
            MaximumBatchingWindowInSeconds: 5
```

5.7.2 Retry en código (Project Reactor)

Para operaciones que pueden fallar temporalmente, usá retry en el código:

```
@Service
public class QuoteService {

    public Mono<Quote> saveQuoteWithRetry(Quote quote) {
        return quoteRepository.save(quote)
            .retry(3)  // Reintenta hasta 3 veces
            .retryWhen(Retry.backoff(3, Duration.ofSeconds(1))
                .maxBackoff(Duration.ofSeconds(10))
                .doBeforeRetry(retrySignal ->
                    log.warn("Retrying after error: {}", retrySignal.failure())))
            .onErrorResume(e -> {
                log.error("Failed after retries", e);
                return Mono.error(new RuntimeException("Failed to save quote", e));
            });
    }
}
```

5.7.3 Circuit Breaker Pattern

Para servicios externos que pueden estar caídos, considerá usar circuit breaker:

```
@Service
public class ExternalServiceClient {

    private final CircuitBreaker circuitBreaker = CircuitBreaker.of("external-service",
        CircuitBreakerConfig.custom()
            .failureRateThreshold(50)  // Abrir si >50% fallan
            .waitDurationInOpenState(Duration.ofSeconds(30))
            .slidingWindowSize(10)
            .build());

    public Mono<String> callExternalService(String data) {
        return Mono.fromCallable(() -> {
            // Llamada a servicio externo
```

```
            return httpClient.get(data);
        })
        .transformDeferred(CircuitBreakerOperator.of(circuitBreaker))
        .onErrorResume(CallNotPermittedException.class, e -> {
            // Circuit breaker abierto: retornar respuesta por defecto
            return Mono.just("Service temporarily unavailable");
        });
    }
}
```

En la práctica: Combiná retry, DLQ y circuit breakers para máxima resiliencia. Retry para errores temporales, DLQ para errores persistentes, y circuit breaker para proteger servicios externos.

5.8 Integración con Amazon EventBridge (Arquitectura Event-Driven Enterprise)

5.8.1 ¿Qué es EventBridge?

Amazon EventBridge es un servicio de eventos serverless que permite construir aplicaciones event-driven de forma sencilla. Es el servicio moderno (2024-2025) para arquitecturas basadas en eventos, especialmente para integraciones enterprise.

5.8.2 EventBridge vs SNS vs SQS: ¿Cuándo usar cada uno?

Servicio	Cuándo usar	Patrón	Ventajas
SQS	Procesamiento asíncrono, desacoplamiento punto-a-punto	Queue pattern	Garantía de entrega, procesamiento ordenado
SNS	Notificaciones a múltiples suscriptores	Pub/Sub pattern	Fan-out, múltiples suscriptores
EventBridge	Eventos empresariales, integración con SaaS, reglas complejas	Event-driven pattern	Schema Registry, integración con SaaS, routing avanzado

Recomendación 2024-2025: - **SQS:** Para procesamiento asíncrono con garantía de entrega - **SNS:** Para notificaciones simples a múltiples suscriptores - **EventBridge:** Para arquitecturas event-driven complejas, integración con SaaS, y eventos empresariales

5.8.3 Dependencias (AWS SDK v2)

```
dependencies {
    // AWS SDK v2 - Versiones gestionadas por BOM 2.26.0
    implementation(platform("software.amazon.awssdk:bom:2.26.0"))
```

```
    implementation("software.amazon.awssdk:eventbridge")
}
```

5.8.4 Publicación de eventos a EventBridge

```
package com.example.lambda.service;

import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;
import software.amazon.awssdk.services.eventbridge.EventBridgeAsyncClient;
import software.amazon.awssdk.services.eventbridge.model.PutEventsRequest;
import software.amazon.awssdk.services.eventbridge.model.PutEventsRequestEntry;
import java.time.Instant;
import java.util.UUID;

@Service
public class EventPublisher {

    private final EventBridgeAsyncClient eventBridgeClient;
    private final String eventBusName;

    public EventPublisher(EventBridgeAsyncClient eventBridgeClient,
                          @Value("${aws.eventbridge.bus-name}") String eventBusName) {
        this.eventBridgeClient = eventBridgeClient;
        this.eventBusName = eventBusName;
    }

    public Mono<Void> publishQuoteCreated(String quoteId, String author) {
        PutEventsRequestEntry event = PutEventsRequestEntry.builder()
            .eventBusName(eventBusName)
            .source("quotes.service")
            .detailType("Quote Created")
            .detail(String.format(
                "{\"quoteId\":\"%s\",\"author\":\"%s\",\"timestamp\":\"%s\"}",
                quoteId, author, Instant.now().toString()))
            .build();

        PutEventsRequest request = PutEventsRequest.builder()
            .entries(event)
            .build();

        return Mono.fromFuture(eventBridgeClient.putEvents(request))
            .doOnSuccess(response -> {
                if (response.failedEntryCount() > 0) {
                    log.error("Failed to publish event: {}", response.entries());
                }
            })
            .then();
```

```
    }
}
```

5.8.5 Configuración de EventBridge en Template SAM

```yaml
Resources:
  # Custom Event Bus (opcional, por defecto usa 'default')
  QuotesEventBus:
    Type: AWS::Events::EventBus
    Properties:
      Name: quotes-event-bus

  # EventBridge Rule: cuando se crea un quote, enviar a Lambda
  QuoteCreatedRule:
    Type: AWS::Events::Rule
    Properties:
      EventBusName: !Ref QuotesEventBus
      EventPattern:
        source:
          - quotes.service
        detail-type:
          - Quote Created
      Targets:
        - Arn: !GetAtt QuoteNotificationFunction.Arn
          Id: "1"
      State: ENABLED

  # Lambda que procesa eventos de EventBridge
  QuoteNotificationFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: org.springframework.cloud.function.adapter.aws.FunctionInvoker::handleRequest
      Runtime: java21
      MemorySize: 512
      Timeout: 30
      CodeUri: build/libs/notification-lambda-0.0.1-SNAPSHOT.jar
      Environment:
        Variables:
          SPRING_CLOUD_FUNCTION_DEFINITION: handleQuoteCreated
      Policies:
        - EventBridgeInvokePolicy:
            EventBusName: !Ref QuotesEventBus

  # Permiso para que EventBridge invoque Lambda
  QuoteNotificationFunctionPermission:
    Type: AWS::Lambda::Permission
    Properties:
      FunctionName: !Ref QuoteNotificationFunction
```

```
      Action: lambda:InvokeFunction
      Principal: events.amazonaws.com
      SourceArn: !GetAtt QuoteCreatedRule.Arn
```

5.8.6 EventBridge Schema Registry

EventBridge Schema Registry permite descubrir y validar esquemas de eventos:

```
# Registrar un schema
aws events put-rule \
  --name quote-created-schema \
  --event-pattern '{"source":["quotes.service"]}'

# Obtener schema generado automáticamente
aws schemas describe-schema \
  --registry-name aws.events \
  --schema-name QuoteCreated
```

5.8.7 EventBridge Pipes (Nuevo 2024)

EventBridge Pipes permite transformar y filtrar eventos sin Lambda:

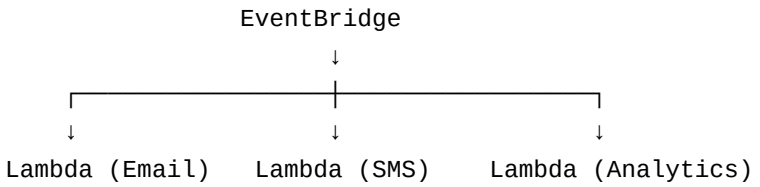
```
Resources:
  # Pipe: SQS → EventBridge → Lambda
  QuoteProcessingPipe:
    Type: AWS::Pipes::Pipe
    Properties:
      Name: quote-processing-pipe
      Source: !GetAtt QuotesQueue.Arn
      Target: !GetAtt QuoteProcessorFunction.Arn
      SourceParameters:
        SqsQueueParameters:
          BatchSize: 10
          MaximumBatchingWindowInSeconds: 5
      Enrichment: !GetAtt EnrichmentFunction.Arn  # Opcional: enriquecer eventos
      Filter:
        Pattern: '{"source":["quotes.service"]}'  # Filtrar eventos
```

5.8.8 Ventajas de EventBridge

1. **Schema Registry:** Descubrimiento y validación automática de esquemas
2. **Integración con SaaS:** Conectores pre-construidos (Salesforce, Zendesk, etc.)
3. **Routing avanzado:** Reglas complejas para routing de eventos
4. **EventBridge Pipes:** Transformación sin Lambda (nuevo 2024)
5. **Event Replay:** Reprocesar eventos históricos

5.8.9 Ejemplo completo: Flujo Event-Driven con EventBridge

```
User → API Gateway → Lambda → DynamoDB
                              ↓
```



En la práctica: Usá EventBridge cuando necesitás routing complejo, integración con SaaS, o arquitecturas event-driven enterprise. Para casos simples, SNS o SQS pueden ser suficientes.

5.9 Flujo completo de evento

El flujo reactivo y serverless queda así:

1. El usuario envía un `POST /quotes`.
2. La Lambda guarda la cotización en **DynamoDB**.
3. Publica un mensaje en **SQS**.
4. Envía una notificación en **SNS** a los suscriptores.

Representación:

```
User → API Gateway → Lambda → DynamoDB → SQS → SNS → Notificados
```

5.10 Testing local con LocalStack

Podés simular todo el flujo localmente con **LocalStack**.

```
docker-compose up -d
awslocal dynamodb list-tables
awslocal sqs list-queues
awslocal sns list-topics
```

Usá `awslocal` en lugar de `aws` para interactuar con los servicios simulados.

5.11 Observabilidad del flujo

Habilitá logs y trazas distribuidas: - **CloudWatch Logs** → para monitoreo básico.
- **X-Ray** → para trazabilidad completa de invocaciones.
- **Micrometer + CloudWatch MeterRegistry** → para métricas personalizadas.

Ejemplo de métrica personalizada:

```
Counter counter = Counter.builder("quotes_created_total")
    .description("Cantidad total de cotizaciones creadas")
    .register(meterRegistry);
counter.increment();
```

5.12 Conclusión de la sección

Integramos el microservicio reactivo con los principales servicios AWS (DynamoDB, SQS, SNS), construyendo un flujo **event-driven**, **escalable** y **sin servidores**.
En la siguiente sección implementaremos **observabilidad avanzada y CI/CD**, consolidando el ciclo completo de entrega.

6 Observabilidad y Despliegue Continuo (CI/CD)

> **⚠ ADVERTENCIA SOBRE COSTOS:**
>
> CloudWatch Logs, métricas y X-Ray **generan costos reales**. Sin configuración adecuada (retención de logs, sampling de X-Ray), los costos pueden escalar rápidamente. Consultá el **Capítulo 11: Control de Costos y Disclaimers Legales** para optimización de costos.

6.1 Objetivo

Aprender a monitorear, registrar y desplegar automáticamente microservicios reactivos serverless en AWS, utilizando herramientas modernas como **CloudWatch**, **X-Ray**, **Micrometer** y **GitHub Actions**.

> **En la práctica**: La observabilidad es algo que muchos desarrolladores dejan para después, pero en producción es absolutamente crítica. Sin observabilidad adecuada, vas a estar ciego cuando algo falle, y esto evita dolores de cabeza en producción.

6.2 ¿Por qué es importante la observabilidad?

Ahora que tenemos nuestra aplicación funcionando y optimizada, es momento de asegurarnos de que podamos monitorearla y entender qué está pasando en producción. Esto nos lleva naturalmente al siguiente punto: cómo implementar observabilidad completa en un entorno serverless y reactivo.

En un entorno **serverless** y reactivo, las funciones pueden ejecutarse en paralelo, de forma distribuida y efímera.
Esto complica significativamente el rastreo de errores o cuellos de botella. La **observabilidad** combina tres pilares fundamentales:

Pilar	Descripción	Herramienta principal
Logging	Qué está ocurriendo en el sistema	CloudWatch Logs, logs estructurados en JSON
Metrics	Cuánto está ocurriendo (volumen, frecuencia)	Micrometer, CloudWatch Metrics
Tracing	Dónde está ocurriendo (flujo de requests)	AWS X-Ray, trazas distribuidas

Un sistema bien observado permite detectar fallos antes de que afecten al usuario final, optimizar el rendimiento y entender el comportamiento del sistema en producción. En la práctica, esto se traduce en poder diagnosticar problemas en minutos en lugar de horas, y en poder optimizar el rendimiento basándote en datos reales en lugar de suposiciones.

6.3 Logging estructurado con CloudWatch

Ahora que entendimos por qué la observabilidad es importante, veamos cómo implementarla. Empecemos con el logging, que es el pilar más básico pero también el más importante.

AWS Lambda integra automáticamente los logs con **CloudWatch Logs**.
Sin embargo, para mejorar el análisis y facilitar la búsqueda, conviene usar logs **estructurados en JSON**.

6.3.1 Ventajas del logging estructurado:

- **Búsqueda eficiente**: CloudWatch Logs Insights puede consultar campos específicos del JSON.
- **Análisis automatizado**: Facilita la creación de dashboards y alarmas basadas en campos estructurados.
- **Consistencia**: Formato uniforme facilita el procesamiento y análisis.

6.3.2 Implementación de logging estructurado:

```java
package com.example.lambda.observability;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;
import com.fasterxml.jackson.databind.ObjectMapper;
import java.util.Map;
import java.util.HashMap;

/**
 * Logger estructurado para CloudWatch Logs.
 * Genera logs en formato JSON para facilitar el análisis y búsqueda.
 */
@Component
public class StructuredLogger {
    private static final Logger log = LoggerFactory.getLogger(StructuredLogger.class);
    private static final ObjectMapper mapper = new ObjectMapper();

    /**
     * Registra un log estructurado con nivel INFO.
     *
     * @param message Mensaje principal del log
     * @param data Datos adicionales a incluir en el log
     */
```

```
    public static void info(String message, Map<String, Object> data) {
        try {
            Map<String, Object> logEntry = new HashMap<>();
            logEntry.put("message", message);
            logEntry.put("data", data);
            logEntry.put("timestamp", System.currentTimeMillis());
            logEntry.put("level", "INFO");

            log.info(mapper.writeValueAsString(logEntry));
        } catch (Exception e) {
            log.error("Error serializando log estructurado", e);
        }
    }

    /**
     * Registra un log estructurado con nivel ERROR.
     */
    public static void error(String message, Throwable error, Map<String, Object> data) {
        try {
            Map<String, Object> logEntry = new HashMap<>();
            logEntry.put("message", message);
            logEntry.put("error", error.getMessage());
            logEntry.put("stackTrace", getStackTrace(error));
            logEntry.put("data", data);
            logEntry.put("timestamp", System.currentTimeMillis());
            logEntry.put("level", "ERROR");

            log.error(mapper.writeValueAsString(logEntry));
        } catch (Exception e) {
            log.error("Error serializando log estructurado", e);
        }
    }

    private static String getStackTrace(Throwable error) {
        java.io.StringWriter sw = new java.io.StringWriter();
        java.io.PrintWriter pw = new java.io.PrintWriter(sw);
        error.printStackTrace(pw);
        return sw.toString();
    }
}
```

6.3.3 Ejemplo de salida en CloudWatch Logs:

```
{
  "message": "Nueva cotización creada",
  "data": {
    "id": "Q-123",
    "timestamp": 1723456767,
```

```
    "userId": "user-456"
  },
  "timestamp": 1723456767000,
  "level": "INFO"
}
```

6.3.4 Consulta en CloudWatch Logs Insights:

```
fields @timestamp, message, data.id, data.userId
| filter message = "Nueva cotización creada"
| sort @timestamp desc
| limit 100
```

6.4 Métricas personalizadas con Micrometer + CloudWatch

Puedes exportar métricas de rendimiento directamente desde Spring Boot con **Micrometer**, que proporciona una abstracción unificada para métricas compatible con múltiples sistemas de monitoreo.

6.4.1 Dependencias

```
dependencies {
    // Core de Micrometer
    implementation("io.micrometer:micrometer-core")

    // Registry para CloudWatch Metrics
    implementation("io.micrometer:micrometer-registry-cloudwatch2")
}
```

6.4.2 Configuración en application.yml

```
management:
  metrics:
    export:
      cloudwatch:
        namespace: reactive-lambda
        enabled: true
        step: PT1M  # Enviar métricas cada minuto
```

6.4.3 Ejemplo de métricas personalizadas

```
package com.example.lambda.observability;

import io.micrometer.core.instrument.Counter;
import io.micrometer.core.instrument.MeterRegistry;
import io.micrometer.core.instrument.Timer;
```

```
import org.springframework.stereotype.Component;
import java.util.concurrent.TimeUnit;

/**
 * Publicador de métricas personalizadas para CloudWatch.
 * Demuestra el uso de contadores, timers y gauges con Micrometer.
 */
@Component
public class MetricsPublisher {
    private final MeterRegistry meterRegistry;
    private final Counter quotesCreatedCounter;
    private final Timer quoteProcessingTimer;

    public MetricsPublisher(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;

        // Contador de cotizaciones creadas
        this.quotesCreatedCounter = Counter.builder("quotes.created.total")
            .description("Número total de cotizaciones creadas")
            .tag("service", "reactive-lambda")
            .register(meterRegistry);

        // Timer para medir tiempo de procesamiento
        this.quoteProcessingTimer = Timer.builder("quotes.processing.duration")
            .description("Tiempo de procesamiento de cotizaciones")
            .tag("service", "reactive-lambda")
            .register(meterRegistry);
    }

    /**
     * Incrementa el contador de cotizaciones creadas.
     */
    public void incrementQuotesCreated() {
        quotesCreatedCounter.increment();
    }

    /**
     * Registra el tiempo de procesamiento de una cotización.
     */
    public void recordQuoteProcessingTime(long duration, TimeUnit unit) {
        quoteProcessingTimer.record(duration, unit);
    }

    /**
     * Registra una métrica de error.
     */
    public void recordError(String errorType) {
        meterRegistry.counter("quotes.errors.total",
            "error_type", errorType,
```

```
            "service", "reactive-lambda")
            .increment();
    }
}
```

6.4.4 Uso en el servicio

```
package com.example.lambda.service;

import com.example.lambda.model.Quote;
import com.example.lambda.model.QuoteRequest;
import com.example.lambda.repository.QuoteRepository;
import com.example.lambda.observability.MetricsPublisher;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;
import java.util.concurrent.TimeUnit;

@Service
public class QuoteService {
    private final QuoteRepository quoteRepository;
    private final MetricsPublisher metricsPublisher;

    public QuoteService(QuoteRepository quoteRepository, MetricsPublisher metricsPublisher) {
        this.quoteRepository = quoteRepository;
        this.metricsPublisher = metricsPublisher;
    }

    public Mono<Quote> createQuote(QuoteRequest request) {
        long startTime = System.currentTimeMillis();

        Quote quote = new Quote();
        quote.setText(request.getText());
        quote.setAuthor(request.getAuthor());

        return quoteRepository.save(quote)
            .doOnSuccess(savedQuote -> {
                metricsPublisher.incrementQuotesCreated();
                long duration = System.currentTimeMillis() - startTime;
                metricsPublisher.recordQuoteProcessingTime(duration, TimeUnit.MILLISECONDS);
            })
            .doOnError(error -> {
                metricsPublisher.recordError(error.getClass().getSimpleName());
            });
    }
}
```

Estas métricas aparecen automáticamente en **CloudWatch Metrics** bajo el namespace configurado, y puedes crear alarmas o dashboards personalizados basados en ellas.

6.5 Trazas distribuidas con AWS X-Ray

AWS X-Ray permite visualizar cómo las solicitudes fluyen entre servicios, desde API Gateway hasta Lambda, DynamoDB, SQS y otros servicios AWS. Esto es esencial para debugging y optimización en producción.

> **Alternativa moderna (2024-2025): AWS Distro for OpenTelemetry**
>
> **AWS Distro for OpenTelemetry (ADOT)** es una alternativa moderna a X-Ray que usa el estándar OpenTelemetry. Ventajas: - **Estándar abierto**: OpenTelemetry es un estándar de la industria, no propietario de AWS - **Portabilidad**: Podés exportar trazas a múltiples backends (X-Ray, Jaeger, Datadog, etc.) - **Mejor integración**: Compatible con herramientas de observabilidad modernas - **Mismo costo**: Similar a X-Ray en términos de pricing
>
> Para usar ADOT, agregá la dependencia `io.opentelemetry:opentelemetry-aws-lambda` y configurá el collector ADOT. Sin embargo, X-Ray sigue siendo la opción más simple y nativa para AWS Lambda, por lo que este ebook se enfoca en X-Ray para mantener la simplicidad.

6.5.1 ¿Por qué X-Ray es importante?

- **Visibilidad end-to-end:** Ver el flujo completo de una solicitud a través de múltiples servicios
- **Identificar cuellos de botella:** Encontrar qué servicio está causando latencia
- **Debugging distribuido:** Rastrear errores a través de múltiples Lambdas
- **Análisis de performance:** Entender dónde optimizar

6.5.2 Configuración en el Template SAM

Primero, habilitá X-Ray en tu template SAM:

```yaml
Resources:
  ReactiveFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: org.springframework.cloud.function.adapter.aws.FunctionInvoker::handleRequest
      Runtime: java21
      MemorySize: 1024
      Timeout: 30
      CodeUri: build/libs/reactive-lambda-0.0.1-SNAPSHOT.jar
      Tracing: Active  # ▫ IMPORTANTE: Habilita X-Ray tracing
      Environment:
        Variables:
          SPRING_CLOUD_FUNCTION_DEFINITION: hello
      Policies:
        # Permiso para X-Ray
        - Version: '2012-10-17'
          Statement:
            - Effect: Allow
```

```
              Action:
                - xray:PutTraceSegments
                - xray:PutTelemetryRecords
              Resource: "*"
      Events:
        HttpApi:
          Type: HttpApi
          Properties:
            Path: /hello
            Method: GET
            ApiId: !Ref ReactiveApi

  ReactiveApi:
    Type: AWS::Serverless::HttpApi
    Properties:
      Description: API Gateway HTTP API con X-Ray habilitado
      # Habilitar X-Ray en API Gateway
      Tracing: true
```

6.5.3 Dependencias en build.gradle.kts

```
dependencies {
    // AWS X-Ray SDK para Java
    implementation("com.amazonaws:aws-xray-recorder-sdk-core:2.14.0")
    implementation("com.amazonaws:aws-xray-recorder-sdk-aws-sdk-v2:2.14.0")
    implementation("com.amazonaws:aws-xray-recorder-sdk-aws-sdk-v2-instrumentor:2.14.0")

    // Para Spring Boot (opcional, si usás Spring AOP)
    implementation("com.amazonaws:aws-xray-spring:2.14.0")
}
```

6.5.4 Configuración en el código

```
package com.example.lambda.config;

import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorder;
import com.amazonaws.xray.AWSXRayRecorderBuilder;
import com.amazonaws.xray.plugins.EC2Plugin;
import com.amazonaws.xray.plugins.ElasticBeanstalkPlugin;
import com.amazonaws.xray.strategy.sampling.LocalSamplingStrategy;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import javax.annotation.PostConstruct;

/**
 * Configuración de AWS X-Ray para tracing distribuido.
```

```
 */
@Configuration
public class XRayConfig {

    @PostConstruct
    public void configureXRay() {
        // Configurar el recorder de X-Ray
        AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder.standard()
            .withPlugin(new EC2Plugin())
            .withPlugin(new ElasticBeanstalkPlugin())
            .withSamplingStrategy(new LocalSamplingStrategy(1.0)); // 100% sampling para desarr

        AWSXRay.setGlobalRecorder(builder.build());
    }
}
```

6.5.4.1 1. Configurar X-Ray en la aplicación

```
package com.example.lambda.repository;

import com.amazonaws.xray.interceptors.TracingInterceptor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClientBuilder;
import software.amazon.awssdk.regions.Region;

@Configuration
public class AwsSdkXRayConfig {

    @Bean
    public DynamoDbAsyncClient dynamoDbAsyncClient() {
        // Configurar cliente DynamoDB con X-Ray interceptor
        ClientOverrideConfiguration overrideConfig = ClientOverrideConfiguration.builder()
            .addExecutionInterceptor(new TracingInterceptor())
            .build();

        return DynamoDbAsyncClient.builder()
            .region(Region.US_EAST_1)
            .overrideConfiguration(overrideConfig)
            .build();
    }
}
```

6.5.4.2 2. Instrumentar llamadas a AWS SDK

```java
package com.example.lambda.service;

import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Subsegment;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.GetItemResponse;

@Service
public class QuoteService {

    private final DynamoDbAsyncClient dynamoDbClient;

    public QuoteService(DynamoDbAsyncClient dynamoDbClient) {
        this.dynamoDbClient = dynamoDbClient;
    }

    public Mono<Quote> getQuote(String id) {
        // Crear subsegmento para operación de negocio
        Subsegment subsegment = AWSXRay.beginSubsegment("QuoteService.getQuote");

        try {
            subsegment.putMetadata("quoteId", id);

            // La llamada a DynamoDB se rastrea automáticamente por el interceptor
            GetItemRequest request = GetItemRequest.builder()
                .tableName("Quotes")
                .key(Map.of("id", AttributeValue.builder().s(id).build()))
                .build();

            return Mono.fromFuture(dynamoDbClient.getItem(request))
                .map(response -> {
                    // Agregar metadata al subsegmento
                    subsegment.putMetadata("dynamodb.response", response.toString());
                    return parseQuote(response);
                })
                .doOnSuccess(quote -> {
                    subsegment.putMetadata("quoteFound", true);
                    AWSXRay.endSubsegment();
                })
                .doOnError(error -> {
                    subsegment.putException(error);
                    AWSXRay.endSubsegment();
                });

        } catch (Exception e) {
```

```
            subsegment.putException(e);
            AWSXRay.endSubsegment();
            return Mono.error(e);
        }
    }

    private Quote parseQuote(GetItemResponse response) {
        // Lógica de parsing
        return new Quote();
    }
}
```

6.5.4.3 3. Crear segmentos y subsegmentos personalizados

```
package com.example.lambda.handlers;

import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Segment;
import org.springframework.stereotype.Component;
import java.util.function.Function;

@Component
public class HelloHandler implements Function<APIGatewayProxyRequestEvent, APIGatewayProxyRespo

    @Override
    public APIGatewayProxyResponseEvent apply(APIGatewayProxyRequestEvent request) {
        // X-Ray crea automáticamente un segmento para cada invocación Lambda
        // Podés agregar metadata adicional
        Segment segment = AWSXRay.getCurrentSegment();
        if (segment != null) {
            segment.putMetadata("requestId", request.getRequestContext().getRequestId());
            segment.putMetadata("path", request.getPath());
            segment.putMetadata("method", request.getHttpMethod());
        }

        // Tu lógica de negocio aquí
        // Todas las llamadas a AWS SDK se rastrean automáticamente

        return APIGatewayProxyResponseEvent.builder()
            .withStatusCode(200)
            .withBody("{\"message\": \"Hello from Lambda\"}")
            .build();
    }
}
```

6.5.4.4 4. Instrumentar función Lambda principal

6.5.5 Visualización en la consola de X-Ray

Una vez desplegado, podés ver las trazas en la consola de AWS X-Ray:

1. **Service Map:** Visualización gráfica de todos los servicios y sus relaciones
2. **Trace List:** Lista de todas las trazas con filtros por tiempo, servicio, error, etc.
3. **Trace Details:** Detalles de una traza específica, mostrando:
   - Tiempo total de la solicitud
   - Tiempo en cada servicio (API Gateway, Lambda, DynamoDB, etc.)
   - Errores y excepciones
   - Metadata personalizada

6.5.6 Ejemplo de trace completo

Una solicitud típica mostrará:

```
API Gateway (10ms)
  └── Lambda Function (150ms)
      ├── DynamoDB GetItem (50ms)
      ├── SQS SendMessage (30ms)
      └── Business Logic (70ms)
```

6.5.7 Configuración de sampling (muestreo)

Para producción, configurá sampling para reducir costos:

```java
@Configuration
public class XRayConfig {

    @Bean
    public AWSXRayRecorder xRayRecorder() {
        // Sampling del 10% en producción (ajustar según necesidad)
        LocalSamplingStrategy samplingStrategy = new LocalSamplingStrategy(0.1);

        return AWSXRayRecorderBuilder.standard()
            .withSamplingStrategy(samplingStrategy)
            .build();
    }
}
```

O usando reglas de sampling en AWS:

```json
{
  "version": 2,
  "rules": [
    {
      "description": "Sample 10% of requests",
      "service_name": "*",
      "service_type": "*",
      "host": "*",
      "http_method": "*",
      "url_path": "*",
      "fixed_rate": 0.1,
```

```
      "reservoir_size": 1
    }
  ],
  "default": {
    "fixed_rate": 0.1,
    "reservoir_size": 1
  }
}
```

6.5.8 Mejores prácticas con X-Ray

1. **Agregar metadata relevante:** User ID, request ID, business context
2. **Usar subsegmentos para operaciones importantes:** Operaciones de negocio, llamadas externas
3. **Configurar sampling apropiado:** 100% en desarrollo, 1-10% en producción
4. **Monitorear costos:** X-Ray cobra por trace, ajustar sampling según volumen
5. **Integrar con CloudWatch:** Crear alarmas basadas en métricas de X-Ray

6.5.9 Activación local (para desarrollo)

Para pruebas locales, podés usar el X-Ray daemon:

```
# Instalar X-Ray daemon
docker run -d -p 2000:2000/udp -p 2000:2000/tcp \
  --name xray-daemon \
  amazon/aws-xray-daemon

# O descargar binario
wget https://s3.dualstack.us-east-1.amazonaws.com/aws-xray-assets.us-east-1/xray-daemon/aws-xra
unzip aws-xray-daemon-linux-3.x.zip
./xray-daemon --bind=0.0.0.0:2000
```

Esto permite rastrear cada invocación Lambda, DynamoDB o SQS, mostrando su latencia total y el flujo completo de la solicitud.

6.6 Pipeline de CI/CD con GitHub Actions

Creamos un flujo automatizado para compilar, testear y desplegar la Lambda.

Archivo: `.github/workflows/deploy.yml`

```
name: Build & Deploy Lambda
on:
  push:
    branches: [ main ]

jobs:
  build-deploy:
    runs-on: ubuntu-latest
```

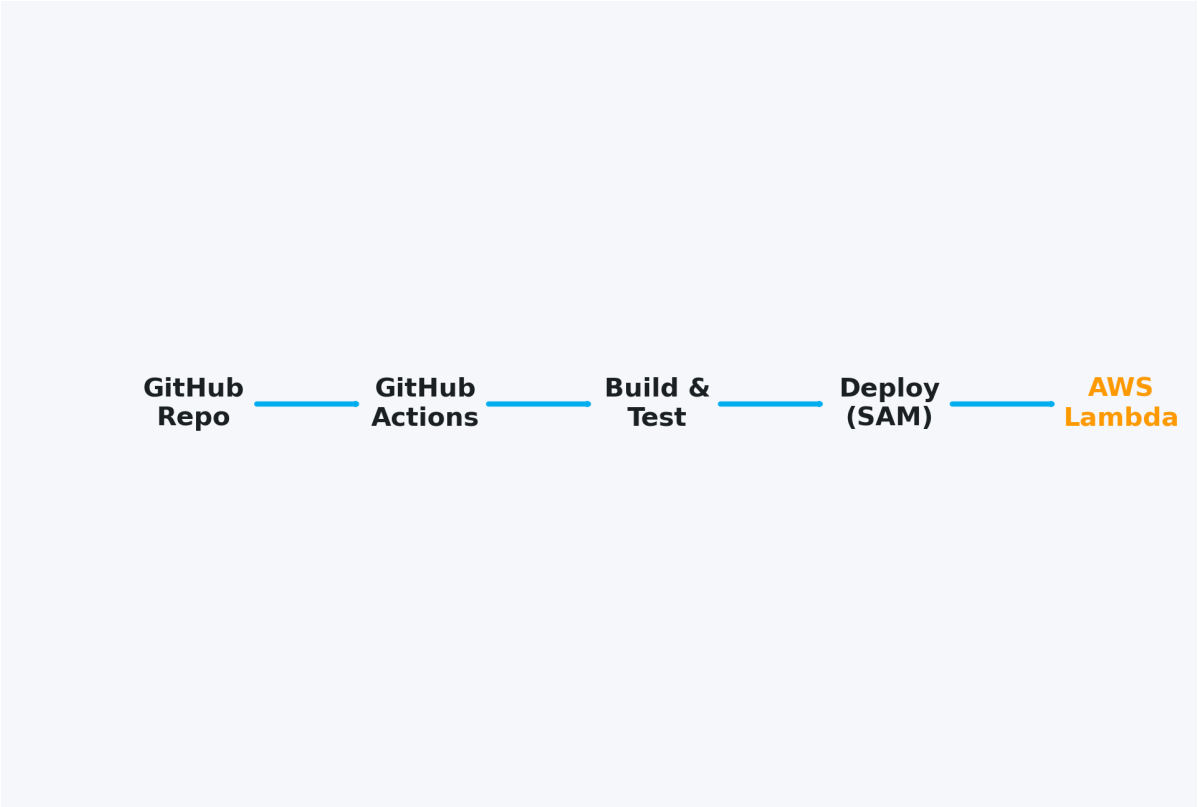


Figura 11: Pipeline CI/CD (GitHub → Lambda)

```yaml
steps:
  - name: Checkout code
    uses: actions/checkout@v4

  - name: Set up JDK 25
    uses: actions/setup-java@v4
    with:
      java-version: '25'
      distribution: temurin
      java-opts: '-XX:+UseCompactObjectHeaders'

  - name: Build project
    run: ./gradlew clean build -x processAot -x compileAotJava -x processAotResources -x ao

  - name: Package Lambda
    run: ./gradlew bootJar -x processAot -x compileAotJava -x processAotResources -x aotCla
```

⚠ Nota importante sobre GitHub Actions y AOT:

Cuando usés **Spring Cloud Function** en tu proyecto, es necesario **excluir las tareas AOT** en los comandos de Gradle dentro de GitHub Actions. Esto se hace agregando los flags `-x processAot -x compileAotJava -x processAotResources -x aotClasses` a todos los comandos `./gradlew`.

Razón: Spring Cloud Function no es completamente compatible con el procesamiento AOT de Spring Boot 3.4+. Excluir estas tareas evita errores de compilación en el

pipeline CI/CD.

Alternativa: Si no usás Spring Cloud Function, podés eliminar estos flags y habilitar AOT normalmente.

```
  - name: Configure AWS credentials
    uses: aws-actions/configure-aws-credentials@v3
    with:
      aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
      aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
      aws-region: us-east-1

  - name: Deploy to AWS Lambda
    run: |
      aws lambda update-function-code             --function-name ReactiveQuotesLambda

Este flujo se ejecutará automáticamente en cada *push* a la rama principal, garantizando integr

---

## Monitoreo de costos y rendimiento

Para evitar gastos innecesarios:

- Activá **AWS Budgets** con alertas automáticas.
- Revisá los tiempos de ejecución Lambda en CloudWatch.
- Considerá reducir memoria si los tiempos son bajos (<300 ms).
- Podés usar herramientas como **Dashbird** o **Lumigo** para monitoreo avanzado de Lambdas.

---

## Estrategias de rollback y versionado

AWS Lambda soporta versiones y alias. Esto permite revertir rápidamente si una actualización fa

```bash
aws lambda publish-version --function-name ReactiveQuotesLambda
aws lambda update-alias --function-name ReactiveQuotesLambda --name prod --function-version 5
```

Esto crea un alias `prod` apuntando a la versión estable de la función.

6.7 Conclusión de la sección

En esta sección aprendiste a implementar **observabilidad completa y CI/CD** en tu microservicio reactivo: - Logs estructurados en CloudWatch.
- Métricas con Micrometer.
- Trazas distribuidas con X-Ray.
- Pipeline automatizado con GitHub Actions.
```

Con esto, tu solución serverless ya está lista para producción, cumpliendo los estándares de escalabilidad, trazabilidad y despliegue continuo.

7 Conclusiones y Próximos Pasos

7.1 Objetivo

Reflexionar sobre el camino recorrido, sintetizar los principales aprendizajes y definir los próximos pasos para seguir construyendo soluciones reactivas y serverless en Java con AWS.

> 🚀 **En la práctica**: Llegar hasta acá significa que ya tenés una base sólida para construir aplicaciones reactivas y serverless. Ahora es momento de aplicar lo aprendido en proyectos reales y seguir profundizando en los conceptos avanzados.

7.2 Próximos pasos sugeridos

Ahora que tenemos una aplicación completa funcionando, es momento de pensar en cómo expandirla y mejorarla. Esto nos lleva naturalmente al siguiente punto: qué hacer después de tener tu primera aplicación reactiva y serverless funcionando.

1. **Ampliar el proyecto**
   - Implementar **API Gateway** como endpoint para múltiples Lambdas.
   - Añadir **Step Functions** para orquestación de flujos complejos.
   - Incorporar **EventBridge** para eventos empresariales más robustos.
2. **Agregar autenticación y seguridad avanzada**
   - Integrar **Amazon Cognito** o **JWT/OAuth2**.
   - Aplicar políticas IAM por función para principio de menor privilegio.
3. **Automatizar la infraestructura**
   - Adoptar **Terraform** o **AWS CDK** para IaC (Infrastructure as Code).
   - Versionar configuraciones y entornos.
4. **Optimizar costos y rendimiento**
   - Revisar métricas CloudWatch y ajustar recursos.
   - Adoptar **AWS Savings Plans** si el tráfico es constante.
5. **Expandir el conocimiento**
   - Profundizar en **Reactive Streams y Project Reactor**.
   - Explorar **Spring Cloud Gateway** y **Kubernetes + Knative** para entornos híbridos.
   - Analizar casos reales de **migración a serverless** en empresas.

7.3 Mensaje final

Hemos recorrido un camino completo desde los conceptos fundamentales hasta una aplicación completamente funcional, optimizada y lista para producción. El futuro del desarrollo backend combina **reactividad**, **escalabilidad** y **simplicidad operativa**, y con **Java 25 LTS**, **Spring Boot 3.4+**, **GraalVM Native Image** y **AWS Lambda**, tenés las herramientas para construir software moderno, eficiente y sostenible que compite directamente con las plataformas más populares del mercado.

En la práctica, esto se traduce en aplicaciones que escalan automáticamente, que responden en milisegundos y que consumen solo los recursos que realmente necesitan. Esto evita dolores de cabeza en producción relacionados con el aprovisionamiento de servidores, el balanceo de carga y el mantenimiento de la infraestructura.

7.3.1 Lecciones clave aprendidas:

1. **La programación reactiva no es solo una moda**: Es un paradigma fundamental para sistemas modernos que necesitan escalar y responder en tiempo real.
2. **Java sigue siendo relevante en serverless**: Con GraalVM Native Image, Java puede competir con Node.js y Python en términos de rendimiento y eficiencia.
3. **La observabilidad es esencial**: Sin logs, métricas y trazas, es imposible operar sistemas distribuidos en producción.
4. **La seguridad debe ser prioritaria**: Desde el diseño inicial, no como una consideración posterior.

   "La mejor optimización no es la que consume menos recursos, sino la que permite escalar sin perder control."
   – Marcos Raimundo Lozina

Gracias por llegar hasta el final.
¡Tu camino hacia la ingeniería cloud-reactiva acaba de comenzar!

7.3.2 Recursos adicionales

- Spring WebFlux Documentation
- Spring Cloud Function AWS Adapter
- AWS SAM CLI Reference
- GraalVM Official Site
- AWS Lambda Developer Guide

8 Seguridad y Autenticación en AWS Lambda

⚠ ADVERTENCIA SOBRE COSTOS:

API Gateway Authorizers, Secrets Manager y otros servicios de seguridad **generan costos reales**. Consultá el **Capítulo 11: Control de Costos y Disclaimers Legales** para

estimaciones de costos.

8.1 Objetivo

Implementar mecanismos de seguridad robustos en funciones Lambda reactivas, incluyendo autenticación, autorización, manejo de secretos y mejores prácticas de seguridad en entornos serverless.

> En la práctica: La seguridad es algo que muchos desarrolladores subestiman hasta que tienen un incidente. En producción, un error de seguridad puede ser catastrófico, así que es mejor implementarla correctamente desde el principio.

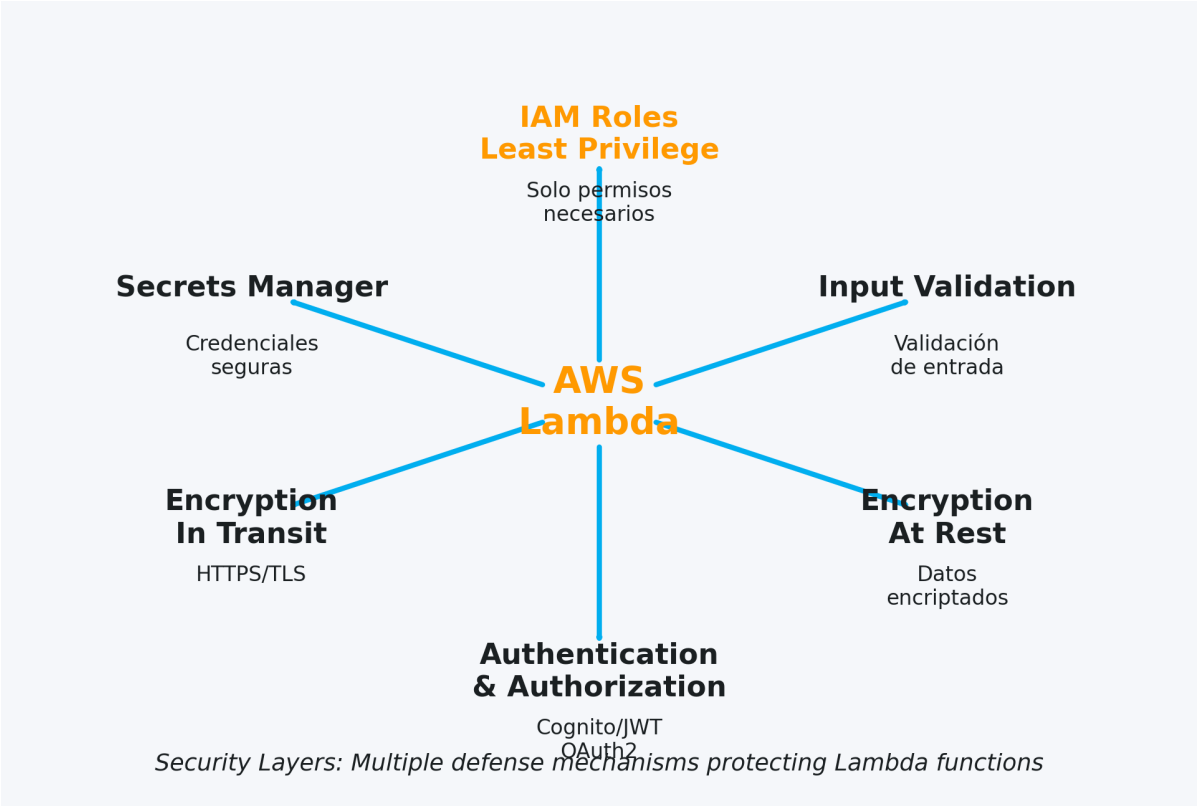


Figura 12: Capas de Seguridad en Serverless

8.2 Principios de Seguridad en Serverless

Ahora que tenemos nuestra aplicación funcionando, optimizada y observable, es momento de asegurarnos de que esté protegida. Esto nos lleva naturalmente al siguiente punto: cómo implementar seguridad robusta en un entorno serverless y reactivo.

En entornos serverless, la seguridad sigue un **modelo de responsabilidad compartida** entre AWS y el desarrollador:

Responsabilidad	Descripción	Ejemplos
AWS (Infraestructura)	Seguridad de la plataforma y servicios gestionados	Aislamiento de funciones, actualizaciones del runtime, seguridad física de los datacenters
Desarrollador (Aplicación)	Seguridad de la aplicación y datos	Validación de entrada, manejo de secretos, autenticación/autorización, políticas IAM, encriptación de datos

8.2.1 Responsabilidades de AWS:

- **Seguridad de la infraestructura**: Protección física y lógica de los datacenters y servicios gestionados
- **Aislamiento de funciones**: Cada función Lambda se ejecuta en un entorno aislado
- **Actualizaciones del runtime**: AWS mantiene actualizados los runtimes de Java, Node.js, Python, etc.

8.2.2 Responsabilidades del desarrollador:

- **Validación de entrada**: Nunca confiar en datos del cliente, siempre validar y sanitizar
- **Manejo seguro de secretos**: Usar AWS Secrets Manager, nunca hardcodear credenciales
- **Autenticación y autorización**: Implementar mecanismos robustos (Cognito, JWT, OAuth2)
- **Políticas IAM adecuadas**: Principio de menor privilegio, solo permisos necesarios
- **Encriptación de datos sensibles**: Encriptar datos en reposo y en tránsito

---

8.3 Configuración de IAM Roles y Policies

Cada función Lambda debe tener un **IAM Role** con políticas que sigan el **principio de menor privilegio** (least privilege principle). Esto significa que la función solo debe tener los permisos mínimos necesarios para realizar su tarea específica.

8.3.1 ¿Por qué es importante el principio de menor privilegio?

- **Reduce el riesgo de exposición**: Si una función es comprometida, el atacante solo tiene acceso a los recursos específicos de esa función
- **Cumple con estándares de seguridad**: Requisito común en auditorías y compliance (SOC 2, ISO 27001, etc.)
- **Facilita el debugging**: Permisos específicos hacen más fácil identificar qué recursos usa cada función
- **Mejora la trazabilidad**: CloudTrail registra exactamente qué acciones realiza cada función

8.3.2 Política IAM básica

Creamos un template SAM con políticas IAM:

```yaml
Resources:
  QuotesFunctionRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
            Action: sts:AssumeRole
      ManagedPolicyArns:
        - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
      Policies:
        # Política básica para ejecución de Lambda (logs y red)
        - PolicyName: LambdaBasicExecution
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action:
                  - logs:CreateLogGroup
                  - logs:CreateLogStream
                  - logs:PutLogEvents
                  - lambda:InvokeFunction
                  - ec2:CreateNetworkInterface
                  - ec2:DescribeNetworkInterfaces
                  - ec2:DeleteNetworkInterface
                Resource: "*"
        # Políticas específicas de aplicación
        # Política con PRINCIPIO DE MÍNIMO PRIVILEGIO
        - PolicyName: DynamoDBQuotesTableAccess
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              # Solo operaciones necesarias en tabla específica (SIN Scan si no es necesario)
              - Effect: Allow
                Action:
                  - dynamodb:GetItem
                  - dynamodb:PutItem
                  - dynamodb:UpdateItem
                  - dynamodb:DeleteItem
                  - dynamodb:Query
                  # Scan solo si es absolutamente necesario (remover si no se usa)
                  # - dynamodb:Scan  # COMENTADO: solo usar si realmente se necesita
                Resource:
                  - !GetAtt QuotesTable.Arn
                  - !Sub "${QuotesTable.Arn}/index/*"  # Para índices específicos si existen
              # Rate limiting table - operaciones específicas
```

```
            - Effect: Allow
              Action:
                - dynamodb:UpdateItem
                - dynamodb:GetItem
              Resource: !GetAtt RateLimitsTable.Arn
        - PolicyName: SQSAccess
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action:
                  - sqs:SendMessage  # Solo envío, no recepción si no es necesario
                Resource: !GetAtt QuotesQueue.Arn  # ARN específico, no "*"
        - PolicyName: SecretsManagerAccess
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action:
                  - secretsmanager:GetSecretValue
                Resource: !Sub "arn:aws:secretsmanager:${AWS::Region}:"
                  "${AWS::AccountId}:secret:quotes/*"
                # Restringir al patrón específico, no usar "*"
```

8.3.3 Ejemplo Real del Proyecto: Template SAM con Least Privilege

El proyecto `reactive-microservices-aws-lambda-java25` implementa políticas IAM con principio de mínimo privilegio en `lambda-infra/template.yaml`:

> **Ver la configuración completa**: Podés ver todas las políticas IAM y configuraciones de seguridad en el repositorio en `lambda-infra/template.yaml` y en `lambda-core/src/main/java/com/example/lambda/security/`.

```
Resources:
  ReactiveFunction:
    Type: AWS::Serverless::Function
    Properties:
      Policies:
        # Permiso mínimo para CloudWatch Logs (least privilege)
        - Version: '2012-10-17'
          Statement:
            - Effect: Allow
              Action:
                - logs:CreateLogGroup
                - logs:CreateLogStream
                - logs:PutLogEvents
              Resource:
                # ARN específico de esta función, no "*"
                - !Sub 'arn:aws:logs:${AWS::Region}:${AWS::AccountId}:'
                  'log-group:/aws/lambda/${ReactiveFunction}:*'
```

```
            - !Sub 'arn:aws:logs:${AWS::Region}:${AWS::AccountId}:'
              'log-group:/aws/lambda/${ReactiveFunction}'
          - Effect: Allow
            Action:
              - lambda:InvokeFunction
              - ec2:CreateNetworkInterface
              - ec2:DescribeNetworkInterfaces
              - ec2:DeleteNetworkInterface
            Resource: "*"
```

Características: - [OK] Solo permisos necesarios (logs para esta función específica) - [OK] ARNs específicos en lugar de "*" - [OK] Comentarios explicativos sobre cada política

QuotesFunction: Type: AWS::Serverless::Function Properties: Role: !GetAtt QuotesFunctionRole.Arn # ... resto de configuración

```
---

## Manejo de Secretos con AWS Secrets Manager

Nunca hardcodees credenciales. Usa **AWS Secrets Manager** para almacenar secretos de forma seg

### Dependencias
```kotlin
dependencies {
    implementation(platform("software.amazon.awssdk:bom:2.25.0"))
    implementation("software.amazon.awssdk:secretsmanager")
}
```

8.3.4 Servicio para obtener secretos

```java
package com.example.lambda.security;

import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;
import software.amazon.awssdk.services.secretsmanager.SecretsManagerAsyncClient;
import software.amazon.awssdk.services.secretsmanager.model.GetSecretValueRequest;
import software.amazon.awssdk.services.secretsmanager.model.GetSecretValueResponse;
import java.time.Duration;

/**
 * Servicio para obtener secretos de AWS Secrets Manager de forma reactiva.
 * Implementa cache para evitar múltiples llamadas y reduce costos.
 */
@Service
```
```

```
public class SecretsService {

    private static final Logger log = LoggerFactory.getLogger(SecretsService.class);
    private final SecretsManagerAsyncClient secretsClient;
    private final ObjectMapper objectMapper;

    public SecretsService(SecretsManagerAsyncClient secretsClient, ObjectMapper objectMapper) {
        this.secretsClient = secretsClient;
        this.objectMapper = objectMapper;
    }

    /**
     * Obtiene un secreto de AWS Secrets Manager.
     *
     * @param secretName Nombre o ARN del secreto en Secrets Manager
     * @return Mono que emite el valor del secreto como String
     */
    public Mono<String> getSecret(String secretName) {
        GetSecretValueRequest request = GetSecretValueRequest.builder()
            .secretId(secretName)
            .build();

        return Mono.fromFuture(secretsClient.getSecretValue(request))
            .map(GetSecretValueResponse::secretString)
            .doOnError(error -> log.error("Error retrieving secret: {}", secretName, error))
            .retry(3) // Reintenta hasta 3 veces en caso de error temporal
            .timeout(Duration.ofSeconds(5)) // Timeout de 5 segundos
            .cache(); // Cache para evitar múltiples llamadas (útil en warm starts de Lambda)
    }

    /**
     * Obtiene un secreto y lo parsea como JSON.
     * Útil cuando el secreto contiene múltiples valores (ej: username, password, url).
     *
     * @param secretName Nombre o ARN del secreto
     * @return Mono que emite el secreto parseado como JsonNode
     */
    public Mono<JsonNode> getSecretAsJson(String secretName) {
        return getSecret(secretName)
            .map(json -> {
                try {
                    return objectMapper.readTree(json);
                } catch (Exception e) {
                    log.error("Error parsing secret JSON: {}", secretName, e);
                    throw new RuntimeException("Error parsing secret JSON: " + secretName, e);
                }
            })
            .onErrorMap(e -> new RuntimeException("Failed to parse secret as JSON: " + secretNa
    }
```

```
}
```

Notas importantes: - El cache persiste durante el **warm start** de Lambda, pero se reinicia en cada **cold start** - El retry ayuda a manejar errores temporales de red o throttling de AWS - El timeout previene que la función se quede bloqueada esperando una respuesta

8.3.5 Configuración del cliente

```
package com.example.lambda.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.secretsmanager.SecretsManagerAsyncClient;

/**
 * Configuración del cliente asíncrono de AWS Secrets Manager.
 * El cliente es completamente asíncrono y compatible con Project Reactor.
 */
@Configuration
public class SecretsManagerConfig {

    @Value("${aws.region:us-east-1}")
    private String region;

    /**
     * Crea el cliente asíncrono de Secrets Manager.
     * Retorna CompletableFuture que puede convertirse a Mono/Flux.
     */
    @Bean
    public SecretsManagerAsyncClient secretsManagerAsyncClient() {
        return SecretsManagerAsyncClient.builder()
            .region(Region.of(region))
            .build();
    }
}
```

8.3.6 Uso en el servicio

```
package com.example.lambda.service;

import com.example.lambda.security.SecretsService;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;
import java.sql.Connection;

/**
```

```
 * Servicio que demuestra el uso de secretos en un servicio de base de datos.
 * El secreto se cachea al inicio para evitar múltiples llamadas a Secrets Manager.
 */
@Service
public class DatabaseService {

    private final SecretsService secretsService;
    // Cache del secreto para evitar múltiples llamadas a Secrets Manager
    // Nota: En Lambda, el cache persiste durante el warm start, pero se reinicia en cold start
    private final Mono<String> databaseUrl;

    public DatabaseService(SecretsService secretsService) {
        this.secretsService = secretsService;
        // Cache el secreto al inicio - se obtiene una vez y se reutiliza
        // Esto reduce costos y latencia en invocaciones subsecuentes
        this.databaseUrl = secretsService.getSecret("quotes-database-credentials")
            .cache();
    }

    /**
     * Obtiene una conexión a la base de datos usando el secreto cacheado.
     *
     * @return Mono que emite una Connection a la base de datos
     */
    public Mono<Connection> getConnection() {
        return databaseUrl
            .map(url -> {
                // Crear conexión usando el secreto obtenido de Secrets Manager
                return createConnection(url);
            })
            .onErrorMap(e -> new RuntimeException("Failed to create database connection", e));
    }

    private Connection createConnection(String url) {
        // Implementación de creación de conexión
        // Ejemplo: return DriverManager.getConnection(url);
        return null; // Placeholder
    }
}
```

8.3.7 Crear secreto en AWS

```
aws secretsmanager create-secret \
  --name quotes-database-credentials \
  --secret-string '{"username":"admin","password":"secret123","url":"jdbc:postgresql://..."}'
```

8.4 Autenticación con Amazon Cognito

Amazon Cognito es un servicio gestionado de AWS que proporciona autenticación, autorización y gestión de usuarios. Ofrece dos componentes principales:

- **Cognito User Pools**: Directorio de usuarios con autenticación completa (registro, login, recuperación de contraseña)
- **Cognito Identity Pools**: Proporciona credenciales temporales de AWS para acceso a recursos

8.4.1 Ventajas de usar Cognito:

- **Gestionado por AWS**: Sin necesidad de mantener servidores de autenticación
- **Escalable**: Maneja millones de usuarios automáticamente
- **Seguro**: Cumple con estándares de seguridad (SOC, PCI, HIPAA)
- **Integración nativa**: Compatible con API Gateway, Lambda y otros servicios AWS
- **Múltiples proveedores**: Soporta login social (Google, Facebook, Amazon, etc.)

8.4.2 Dependencias

```
dependencies {
    implementation(platform("software.amazon.awssdk:bom:2.25.0"))
    implementation("software.amazon.awssdk:cognitoidentityprovider")
    implementation("com.auth0:java-jwt:4.4.0") // Para validar JWT
}
```

8.4.3 Servicio de autenticación con Cognito

```
package com.example.lambda.security;

import com.auth0.jwt.JWT;
import com.auth0.jwt.interfaces.DecodedJWT;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;
import reactor.core.scheduler.Schedulers;
import software.amazon.awssdk.services.cognitoidentityprovider.CognitoIdentityProviderAsyncClie
import software.amazon.awssdk.services.cognitoidentityprovider.model.*;
import java.util.Date;
import java.util.Map;

@Service
public class CognitoAuthService {

    private final CognitoIdentityProviderAsyncClient cognitoClient;
    private final String userPoolId;
    private final String clientId;

    public CognitoAuthService(CognitoIdentityProviderAsyncClient cognitoClient,
```

```
                            @Value("${aws.cognito.userPoolId}") String userPoolId,
                            @Value("${aws.cognito.clientId}") String clientId) {
        this.cognitoClient = cognitoClient;
        this.userPoolId = userPoolId;
        this.clientId = clientId;
    }

    public Mono<AuthResult> authenticate(String username, String password) {
        AdminInitiateAuthRequest request = AdminInitiateAuthRequest.builder()
            .userPoolId(userPoolId)
            .clientId(clientId)
            .authFlow(AuthFlowType.ADMIN_USER_PASSWORD_AUTH)
            .authParameters(Map.of(
                "USERNAME", username,
                "PASSWORD", password
            ))
            .build();

        return Mono.fromFuture(cognitoClient.adminInitiateAuth(request))
            .map(response -> {
                AuthenticationResultType authResult = response.authenticationResult();
                return new AuthResult(
                    authResult.accessToken(),
                    authResult.idToken(),
                    authResult.refreshToken()
                );
            });
    }

    public Mono<Boolean> validateToken(String token) {
        return Mono.fromCallable(() -> {
            try {
                DecodedJWT jwt = JWT.decode(token);
                // Validar expiración y otras claims
                return jwt.getExpiresAt().after(new Date());
            } catch (Exception e) {
                return false;
            }
        }).subscribeOn(Schedulers.boundedElastic());
    }
}

// Clase auxiliar para el resultado de autenticación
class AuthResult {
    private final String accessToken;
    private final String idToken;
    private final String refreshToken;

    public AuthResult(String accessToken, String idToken, String refreshToken) {
```

```
        this.accessToken = accessToken;
        this.idToken = idToken;
        this.refreshToken = refreshToken;
    }

    // Getters
    public String getAccessToken() { return accessToken; }
    public String getIdToken() { return idToken; }
    public String getRefreshToken() { return refreshToken; }
}
```

8.4.4 Filtro de autenticación para API Gateway

```
package com.example.lambda.security;

import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;
import reactor.core.publisher.Mono;
import java.util.function.Function;
import java.time.Duration;
import java.util.Map;
import java.util.HashMap;

/**
 * Filtro de autenticación para validar tokens JWT en requests de API Gateway.
 * Este filtro debe ejecutarse antes de procesar la lógica de negocio.
 *
 * Nota: En un escenario real, considera usar API Gateway Authorizers para validar
 * tokens antes de invocar Lambda, reduciendo costos y latencia.
 */
@Component
public class AuthFilter implements Function<APIGatewayProxyRequestEvent, APIGatewayProxyRequest

    private final CognitoAuthService authService;
    private static final Logger log = LoggerFactory.getLogger(AuthFilter.class);
    private static final String BEARER_PREFIX = "Bearer ";

    public AuthFilter(CognitoAuthService authService) {
        this.authService = authService;
    }

    @Override
    public APIGatewayProxyRequestEvent apply(APIGatewayProxyRequestEvent request) {
        // Extraer el header de autorización
        String authHeader = extractAuthHeader(request);
```

```
        if (authHeader == null || !authHeader.startsWith(BEARER_PREFIX)) {
            log.warn("Missing or invalid Authorization header");
            throw new SecurityException("Missing or invalid Authorization header");
        }

        // Extraer el token (remover el prefijo "Bearer ")
        String token = authHeader.substring(BEARER_PREFIX.length());

        // Validar token de forma reactiva
        // Nota: Lambda requiere respuesta síncrona, pero mantenemos el pipeline reactivo inter
        return Mono.fromCallable(() -> token)
            .flatMap(authService::validateToken)
            .flatMap(isValid -> {
                if (!isValid) {
                    log.warn("Invalid or expired token");
                    return Mono.error(new SecurityException("Invalid or expired token"));
                }
                // Token válido: agregar claims al request para uso posterior
                return Mono.just(enrichRequestWithClaims(request, token));
            })
            .onErrorResume(SecurityException.class, e -> {
                log.warn("Authentication failed: {}", e.getMessage());
                return Mono.error(e); // Relanzar para que Spring Cloud Function maneje el erro
            })
            .timeout(Duration.ofSeconds(5))
            .blockOptional(Duration.ofSeconds(5))
            .orElseThrow(() -> new SecurityException("Token validation timeout"));
    }

    private String extractAuthHeader(APIGatewayProxyRequestEvent request) {
        if (request.getHeaders() == null) {
            return null;
        }
        // API Gateway puede enviar headers en minúsculas o con guiones
        return request.getHeaders().getOrDefault("Authorization",
               request.getHeaders().getOrDefault("authorization", ""));
    }

    private APIGatewayProxyRequestEvent enrichRequestWithClaims(
            APIGatewayProxyRequestEvent request, String token) {
        // Agregar claims del token al request para uso posterior en la lógica de negocio
        // Esto permite acceder a información del usuario sin re-validar el token
        Map<String, String> headers = new HashMap<>(request.getHeaders());
        headers.put("X-User-Id", extractUserIdFromToken(token)); // Ejemplo
        request.setHeaders(headers);
        return request;
    }

    private String extractUserIdFromToken(String token) {
```

```
        // Implementación para extraer el user ID del token JWT
        // En producción, decodifica y valida el token completamente
        return "user-id-from-token"; // Placeholder
    }
}
```

Nota importante: En producción, considera usar **API Gateway Authorizers** (Lambda Authorizer o Cognito User Pool Authorizer) para validar tokens antes de invocar la función Lambda. Esto reduce costos y latencia, ya que las solicitudes no autenticadas no invocan Lambda.

8.5 Implementación de API Gateway Authorizers (Mejor Práctica 2024-2025)

8.5.1 ¿Por qué usar API Gateway Authorizers?

Aunque el filtro de autenticación dentro de Lambda funciona, **API Gateway Authorizers** son la mejor práctica porque:

1. **Validación antes de Lambda:** Las solicitudes no autenticadas no invocan Lambda, ahorrando costos
2. **Reutilizable:** Un solo Authorizer puede proteger múltiples funciones Lambda
3. **Mejor separación de responsabilidades:** Autenticación separada de la lógica de negocio
4. **Menor latencia:** La validación ocurre en API Gateway, no en Lambda
5. **Caché de políticas:** API Gateway puede cachear políticas IAM generadas por el Authorizer

8.5.2 Tipos de Authorizers

Tipo	Cuándo usar	Complejidad
Lambda Authorizer	Validación personalizada (JWT, API Keys, etc.)	Media
Cognito User Pool Authorizer	Usuarios autenticados con Cognito	Baja
IAM Authorizer	Servicios AWS que usan IAM	Baja

8.5.3 Implementación: Lambda Authorizer

Vamos a implementar un Lambda Authorizer que valida tokens JWT:

> **Nota sobre dependencias**: Para usar el ejemplo completo de JWT validation, agregá la dependencia `com.auth0:java-jwt:4.4.0` en tu `build.gradle.kts` (ver Capítulo 2). Esta librería es la estándar de la industria para validación de tokens JWT en Java.

```
package com.example.lambda.auth;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
```

```java
import com.amazonaws.services.lambda.runtime.events.APIGatewayCustomAuthorizerEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayCustomAuthorizerResponse;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import java.util.Map;
import java.util.HashMap;

/**
 * Lambda Authorizer para API Gateway HTTP API.
 * Valida tokens JWT antes de invocar la función Lambda principal.
 *
 * Este Authorizer se ejecuta ANTES de la función Lambda, reduciendo costos
 * y latencia para solicitudes no autenticadas.
 */
public class JwtAuthorizer implements RequestHandler<APIGatewayCustomAuthorizerEvent, APIGatewa

    private static final Logger log = LoggerFactory.getLogger(JwtAuthorizer.class);
    private final JwtValidator jwtValidator;

    public JwtAuthorizer() {
        this.jwtValidator = new JwtValidator(); // Inicializar validador JWT
    }

    @Override
    public APIGatewayCustomAuthorizerResponse handleRequest(
            APIGatewayCustomAuthorizerEvent event, Context context) {

        log.info("Authorizer invoked for method: {}", event.getMethodArn());

        // Extraer token del header Authorization
        String token = extractToken(event);

        if (token == null) {
            log.warn("Missing Authorization header");
            return denyPolicy(event.getMethodArn(), "Missing Authorization header");
        }

        // Validar token JWT
        try {
            JwtClaims claims = jwtValidator.validateToken(token);

            // Token válido: generar política IAM que permite acceso
            return allowPolicy(event.getMethodArn(), claims);

        } catch (SecurityException e) {
            log.warn("Token validation failed: {}", e.getMessage());
            return denyPolicy(event.getMethodArn(), "Invalid or expired token");
        }
    }
```

```
    private String extractToken(APIGatewayCustomAuthorizerEvent event) {
        Map<String, String> headers = event.getHeaders();
        if (headers == null) {
            return null;
        }

        // API Gateway HTTP API envía headers en minúsculas
        String authHeader = headers.getOrDefault("authorization",
                            headers.getOrDefault("Authorization", ""));

        if (authHeader.startsWith("Bearer ")) {
            return authHeader.substring(7);
        }

        return null;
    }

    private APIGatewayCustomAuthorizerResponse allowPolicy(
            String methodArn, JwtClaims claims) {

        // Extraer región, account ID y API ID del ARN
        String[] arnParts = methodArn.split(":");
        String region = arnParts[3];
        String accountId = arnParts[4];
        String apiId = arnParts[5].split("/")[0];

        // Construir ARN base del API
        String apiArn = String.format("arn:aws:execute-api:%s:%s:%s",
                                      region, accountId, apiId);

        // Crear política IAM que permite acceso
        APIGatewayCustomAuthorizerResponse response = new APIGatewayCustomAuthorizerResponse();
        response.setPrincipalId(claims.getUserId()); // User ID del token

        // Política que permite todas las rutas del API (ajustar según necesidad)
        Map<String, Object> policyDocument = new HashMap<>();
        policyDocument.put("Version", "2012-10-17");

        Map<String, Object> statement = new HashMap<>();
        statement.put("Effect", "Allow");
        statement.put("Action", "execute-api:Invoke");
        statement.put("Resource", apiArn + "/*/*"); // Permite todas las rutas

        policyDocument.put("Statement", new Object[]{statement});
        response.setPolicyDocument(policyDocument);

        // Agregar contexto (datos adicionales disponibles en Lambda)
        Map<String, String> context = new HashMap<>();
```

```
        context.put("userId", claims.getUserId());
        context.put("email", claims.getEmail());
        context.put("role", claims.getRole());
        response.setContext(context);

        return response;
    }

    private APIGatewayCustomAuthorizerResponse denyPolicy(String methodArn, String reason) {
        APIGatewayCustomAuthorizerResponse response = new APIGatewayCustomAuthorizerResponse();
        response.setPrincipalId("unauthorized");

        Map<String, Object> policyDocument = new HashMap<>();
        policyDocument.put("Version", "2012-10-17");

        Map<String, Object> statement = new HashMap<>();
        statement.put("Effect", "Deny");
        statement.put("Action", "execute-api:Invoke");
        statement.put("Resource", methodArn);

        policyDocument.put("Statement", new Object[]{statement});
        response.setPolicyDocument(policyDocument);

        return response;
    }
}

// Clase auxiliar para validar JWT usando com.auth0:java-jwt
// ▫ IMPORTANTE: Agregar dependencia en build.gradle.kts:
// implementation("com.auth0:java-jwt:4.4.0")
import com.auth0.jwt.JWT;
import com.auth0.jwt.algorithms.Algorithm;
import com.auth0.jwt.interfaces.DecodedJWT;
import com.auth0.jwt.exceptions.JWTVerificationException;
import java.util.Date;

class JwtValidator {

    private final String jwtSecret;
    private final String jwtIssuer;
    private final String jwtAudience;

    public JwtValidator() {
        // Leer configuración desde variables de entorno
        this.jwtSecret = System.getenv("JWT_SECRET");
        this.jwtIssuer = System.getenv("JWT_ISSUER");
        this.jwtAudience = System.getenv("JWT_AUDIENCE");

        if (jwtSecret == null || jwtSecret.isEmpty()) {
```

```
            throw new IllegalStateException("JWT_SECRET environment variable is required");
        }
    }

    public JwtClaims validateToken(String token) {
        if (token == null || token.isEmpty()) {
            throw new SecurityException("Token is empty");
        }

        try {
            // Crear algoritmo de verificación (HMAC256 para tokens firmados con secreto)
            // Para tokens de Cognito, usar RSA256 con JWK (ver método alternativo más abajo)
            Algorithm algorithm = Algorithm.HMAC256(jwtSecret);

            // Verificar y decodificar token
            DecodedJWT jwt = JWT.require(algorithm)
                .withIssuer(jwtIssuer)  // Validar issuer
                .withAudience(jwtAudience)  // Validar audience
                .acceptExpiresAt(5)  // Aceptar tokens expirados hasta 5 segundos (para clock s
                .build()
                .verify(token);

            // Extraer claims del token
            String userId = jwt.getSubject(); // sub claim
            String email = jwt.getClaim("email").asString();
            String role = jwt.getClaim("role").asString();

            // Validar que el token no esté expirado
            Date expiresAt = jwt.getExpiresAt();
            if (expiresAt != null && expiresAt.before(new Date())) {
                throw new SecurityException("Token has expired");
            }

            return new JwtClaims(
                userId != null ? userId : "unknown",
                email != null ? email : "",
                role != null ? role : "user"
            );

        } catch (JWTVerificationException e) {
            throw new SecurityException("Token validation failed: " + e.getMessage(), e);
        } catch (Exception e) {
            throw new SecurityException("Error validating token: " + e.getMessage(), e);
        }
    }

    /**
     * Método alternativo para validar tokens de AWS Cognito usando RSA256.
     * Cognito usa claves públicas RSA, no secretos HMAC.
```

```
     *
     * Para usar este método, necesitás:
     * 1. Descargar las claves públicas de Cognito (JWK)
     * 2. Usar com.auth0:jwks-rsa para obtener el algoritmo RSA
     *
     * Ejemplo:
     * ```java
     * import com.auth0.jwk.JwkProvider;
     * import com.auth0.jwk.JwkProviderBuilder;
     * import com.auth0.jwt.algorithms.Algorithm;
     *
     * String cognitoRegion = "us-east-1";
     * String cognitoUserPoolId = "us-east-1_XXXXX";
     * String jwksUrl = String.format(
     *     "https://cognito-idp.%s.amazonaws.com/%s/.well-known/jwks.json",
     *     cognitoRegion, cognitoUserPoolId
     * );
     *
     * JwkProvider jwkProvider = new JwkProviderBuilder(jwksUrl).build();
     * DecodedJWT jwt = JWT.decode(token);
     * Algorithm algorithm = Algorithm.RSA256(jwkProvider.get(jwt.getKeyId()));
     * JWT.require(algorithm).build().verify(token);
     * ```
     */
}

// Clase para claims del JWT
class JwtClaims {
    private final String userId;
    private final String email;
    private final String role;

    public JwtClaims(String userId, String email, String role) {
        this.userId = userId;
        this.email = email;
        this.role = role;
    }

    public String getUserId() { return userId; }
    public String getEmail() { return email; }
    public String getRole() { return role; }
}
```

8.5.3.1 1. Lambda Authorizer Function

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Lambda con API Gateway Authorizer
```

```yaml
Resources:
  # Lambda Authorizer
  AuthFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: com.example.lambda.auth.JwtAuthorizer::handleRequest
      Runtime: java21
      MemorySize: 256  # Authorizers son más ligeros
      Timeout: 10
      CodeUri: build/libs/auth-lambda-0.0.1-SNAPSHOT.jar
      Environment:
        Variables:
          JWT_ISSUER: https://cognito-idp.us-east-1.amazonaws.com/us-east-1_XXXXX
          JWT_AUDIENCE: your-app-client-id

  # Función Lambda principal
  ReactiveFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: org.springframework.cloud.function.adapter.aws.FunctionInvoker::handleRequest
      Runtime: java21
      MemorySize: 1024
      Timeout: 30
      CodeUri: build/libs/reactive-lambda-0.0.1-SNAPSHOT.jar
      Environment:
        Variables:
          SPRING_CLOUD_FUNCTION_DEFINITION: hello
      Events:
        HttpApi:
          Type: HttpApi
          Properties:
            Path: /hello
            Method: GET
            ApiId: !Ref ReactiveApi
            Auth:
              Authorizer: LambdaAuthorizer  # Referencia al Authorizer

  # API Gateway HTTP API con Authorizer
  ReactiveApi:
    Type: AWS::Serverless::HttpApi
    Properties:
      Description: API Gateway HTTP API con Lambda Authorizer
      CorsConfiguration:
        AllowOrigins:
          - "https://tudominio.com"  # ▫ En producción, usar dominios específicos
        AllowMethods:
          - GET
          - POST
```

```
          - OPTIONS
        AllowHeaders:
          - Content-Type
          - Authorization
        MaxAge: 300
      Auth:
        Authorizers:
          LambdaAuthorizer:
            AuthorizerPayloadFormatVersion: "2.0"  # Formato simplificado (recomendado)
            EnableSimpleResponses: true
            FunctionArn: !GetAtt AuthFunction.Arn
            FunctionInvokeRole: !GetAtt AuthFunctionRole.Arn
            Identity:
              Headers:
                - authorization  # Header que contiene el token

  # IAM Role para el Authorizer
  AuthFunctionRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
            Action: sts:AssumeRole
      ManagedPolicyArns:
        - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
      Policies:
        - PolicyName: InvokeLambda
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action:
                  - lambda:InvokeFunction
                Resource: !GetAtt ReactiveFunction.Arn

Outputs:
  ApiUrl:
    Description: "API Gateway HTTP API endpoint URL"
    Value: !Sub "https://${ReactiveApi}.execute-api.${AWS::Region}.amazonaws.com"
```

8.5.3.2 2. Template SAM con Lambda Authorizer

8.5.3.3 3. Acceder al contexto del Authorizer en Lambda El contexto del Authorizer (userId, email, role) está disponible en la función Lambda principal:

```
package com.example.lambda.handlers;

import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import org.springframework.stereotype.Component;
import java.util.function.Function;
import java.util.Map;

@Component
public class HelloHandler implements Function<APIGatewayProxyRequestEvent, APIGatewayProxyRespo

    @Override
    public APIGatewayProxyResponseEvent apply(APIGatewayProxyRequestEvent request) {
        // El contexto del Authorizer está disponible en request.getRequestContext()
        Map<String, String> authorizerContext = request.getRequestContext()
            .getAuthorizer()
            .get("lambda"); // Clave "lambda" contiene el contexto del Lambda Authorizer

        String userId = authorizerContext.get("userId");
        String email = authorizerContext.get("email");
        String role = authorizerContext.get("role");

        // Usar información del usuario autenticado
        String message = String.format("Hello, %s! (User ID: %s, Role: %s)",
                                       email, userId, role);

        return APIGatewayProxyResponseEvent.builder()
            .withStatusCode(200)
            .withBody("{\"message\": \"" + message + "\"}")
            .withHeaders(Map.of("Content-Type", "application/json"))
            .build();
    }
}
```

8.5.4 Comparación: Authorizer vs Filtro en Lambda

Aspecto	API Gateway Authorizer	Filtro en Lambda
Ejecución	Antes de invocar Lambda	Dentro de Lambda
Costo	No se cobra Lambda si falla	Se cobra Lambda siempre
Latencia	Menor (validación en API Gateway)	Mayor (validación en Lambda)
Reutilización	Un Authorizer para múltiples funciones	Un filtro por función
Caché	API Gateway cachea políticas	No hay caché
Complejidad	Requiere función separada	Más simple (todo en una función)

8.5.5 Recomendación

Para producción (2024-2025): Usá **API Gateway Authorizers** siempre que sea posible. Es la mejor práctica y reduce costos y latencia significativamente.

Para desarrollo/prototipado: El filtro en Lambda es aceptable si querés mantener todo en una sola función.

8.6 Validación de JWT sin Cognito

Si no usas Cognito, puedes validar JWT directamente:

```
package com.example.lambda.security;

import com.auth0.jwt.JWT;
import com.auth0.jwt.algorithms.Algorithm;
import com.auth0.jwt.interfaces.DecodedJWT;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;
import reactor.core.scheduler.Schedulers;
import java.util.Base64;
import java.util.Map;

@Service
public class JwtValidatorService {

    private final String jwtSecret;
    private final ObjectMapper objectMapper;

    public JwtValidatorService(@Value("${jwt.secret}") String jwtSecret,
                               ObjectMapper objectMapper) {
        this.jwtSecret = jwtSecret;
        this.objectMapper = objectMapper;
    }

    public Mono<Map<String, Object>> validateAndParseToken(String token) {
        return Mono.fromCallable(() -> {
            Algorithm algorithm = Algorithm.HMAC256(jwtSecret);
            DecodedJWT jwt = JWT.require(algorithm)
                .build()
                .verify(token);

            return parseClaims(jwt.getPayload());
        })
        .subscribeOn(Schedulers.boundedElastic())
        .onErrorMap(error -> new SecurityException("Invalid token", error));
    }
```

```
    private Map<String, Object> parseClaims(String payload) {
        try {
            String decoded = new String(Base64.getUrlDecoder().decode(payload));
            return objectMapper.readValue(decoded, Map.class);
        } catch (Exception e) {
            throw new RuntimeException("Error parsing JWT claims", e);
        }
    }
}
```

8.7 Validación de Entrada con Bean Validation

Siempre valida la entrada de datos:

```
package com.example.lambda.model;

import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;

public class QuoteRequest {

    @NotBlank(message = "Text is required")
    @Size(min = 1, max = 500, message = "Text must be between 1 and 500 characters")
    private String text;

    @Size(max = 100, message = "Author must not exceed 100 characters")
    private String author;

    // Getters y setters
}
```

8.7.1 Validador en la función Lambda

```
@Component
public class QuoteFunction {

    private final Validator validator;
    private final ObjectMapper objectMapper;

    public QuoteFunction(Validator validator, ObjectMapper objectMapper) {
        this.validator = validator;
        this.objectMapper = objectMapper;
    }

    private Mono<QuoteRequest> validateRequest(String body) {
        try {
            QuoteRequest request = objectMapper.readValue(body, QuoteRequest.class);
```

```
            Set<ConstraintViolation<QuoteRequest>> violations = validator.validate(request);

            if (!violations.isEmpty()) {
                String errors = violations.stream()
                    .map(v -> v.getPropertyPath() + ": " + v.getMessage())
                    .collect(Collectors.joining(", "));
                throw new ValidationException("Validation failed: " + errors);
            }

            return Mono.just(request);
        } catch (JsonProcessingException e) {
            return Mono.error(new ValidationException("Invalid JSON: " + e.getMessage()));
        }
    }
}
```

8.8 Sanitización de Datos

Sanitiza datos para prevenir inyecciones:

```
@Service
public class DataSanitizer {

    public String sanitizeInput(String input) {
        if (input == null) {
            return null;
        }

        // Remover HTML tags
        String sanitized = input.replaceAll("<[^>]*>", "");

        // Escapar caracteres especiales
        sanitized = sanitized.replace("'", "''")
                             .replace("\\", "\\\\")
                             .trim();

        return sanitized;
    }

    public Mono<String> sanitizeAsync(String input) {
        return Mono.fromCallable(() -> sanitizeInput(input))
                   .subscribeOn(Schedulers.boundedElastic());
    }
}
```

8.9 Rate Limiting Distribuido con DynamoDB

Implementa rate limiting distribuido usando DynamoDB con TTL para evitar problemas de escalabilidad con `ConcurrentHashMap`:

```
package com.example.lambda.security;

import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.dynamodb.model.*;
import java.time.Instant;
import java.time.temporal.ChronoUnit;

@Service
public class RateLimiterService {

    private final DynamoDbAsyncClient dynamoDbClient;
    private final String tableName;
    private final int maxRequestsPerMinute;

    public RateLimiterService(DynamoDbAsyncClient dynamoDbClient,
                              @Value("${rate-limiter.table-name:RateLimits}") String tableName,
                              @Value("${rate-limiter.max-requests:60}") int maxRequestsPerMinut
        this.dynamoDbClient = dynamoDbClient;
        this.tableName = tableName;
        this.maxRequestsPerMinute = maxRequestsPerMinute;
    }

    public Mono<Boolean> checkRateLimit(String clientId) {
        String key = clientId + ":" + Instant.now().getEpochSecond() / 60;
        long ttl = Instant.now().plus(2, ChronoUnit.MINUTES).getEpochSecond();

        // Actualización atómica con condición
        UpdateItemRequest updateRequest = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(Map.of("key", AttributeValue.builder().s(key).build()))
            .updateExpression("ADD #count :inc SET #ttl = :ttl")
            .expressionAttributeNames(Map.of(
                "#count", "count",
                "#ttl", "ttl"
            ))
            .expressionAttributeValues(Map.of(
                ":inc", AttributeValue.builder().n("1").build(),
                ":ttl", AttributeValue.builder().n(String.valueOf(ttl)).build(),
                ":max", AttributeValue.builder().n(String.valueOf(maxRequestsPerMinute)).build(
            ))
            .conditionExpression("attribute_not_exists(#count) OR #count < :max")
            .returnValues(ReturnValue.ALL_NEW)
            .build();
```

```
        return Mono.fromFuture(dynamoDbClient.updateItem(updateRequest))
            .map(response -> {
                // Si la actualización fue exitosa, el cliente está dentro del límite
                return true;
            })
            .onErrorResume(ConditionalCheckFailedException.class, e -> {
                // Condición falló = límite excedido
                log.warn("Rate limit exceeded for client: {}", clientId);
                return Mono.just(false);
            })
            .onErrorResume(DynamoDbException.class, e -> {
                log.error("DynamoDB error checking rate limit", e);
                // En caso de error, permitir la solicitud para no bloquear el servicio
                return Mono.just(true);
            })
            .onErrorReturn(true);  // Fail-open: permitir si hay error
    }
}
```

8.9.1 Crear tabla DynamoDB para Rate Limiting

```
aws dynamodb create-table \
  --table-name RateLimits \
  --attribute-definitions AttributeName=key,AttributeType=S \
  --key-schema AttributeName=key,KeyType=HASH \
  --billing-mode PAY_PER_REQUEST \
  --time-to-live-specification Enabled=true,AttributeName=ttl
```

8.9.2 Ventajas sobre ConcurrentHashMap:

- [OK] **Distribuido:** Funciona en múltiples instancias Lambda
- [OK] **TTL automático:** DynamoDB limpia entradas expiradas automáticamente
- [OK] **Actualización atómica:** Previene race conditions
- [OK] **Sin mantenimiento:** No requiere limpieza manual

8.10 Encriptación de Datos Sensibles

Encripta datos sensibles antes de almacenarlos:

```
@Service
public class EncryptionService {

    private final SecretKey secretKey;

    public EncryptionService(@Value("${encryption.secret}") String secret) {
        this.secretKey = new SecretKeySpec(
            secret.getBytes(StandardCharsets.UTF_8),
```

```
            "AES"
        );
    }

    public Mono<String> encrypt(String data) {
        return Mono.fromCallable(() -> {
            Cipher cipher = Cipher.getInstance("AES");
            cipher.init(Cipher.ENCRYPT_MODE, secretKey);
            byte[] encrypted = cipher.doFinal(data.getBytes(StandardCharsets.UTF_8));
            return Base64.getEncoder().encodeToString(encrypted);
        })
        .subscribeOn(Schedulers.boundedElastic());
    }

    public Mono<String> decrypt(String encryptedData) {
        return Mono.fromCallable(() -> {
            byte[] decoded = Base64.getDecoder().decode(encryptedData);
            Cipher cipher = Cipher.getInstance("AES");
            cipher.init(Cipher.DECRYPT_MODE, secretKey);
            byte[] decrypted = cipher.doFinal(decoded);
            return new String(decrypted, StandardCharsets.UTF_8);
        })
        .subscribeOn(Schedulers.boundedElastic());
    }
}
```

8.11 Mejores Prácticas de Seguridad

8.11.1 Checklist de Seguridad

Práctica	Descripción	Impacto
[OK] **Validar toda la entrada**	Nunca confíes en datos del cliente, siempre validar y sanitizar	Previene inyecciones SQL, XSS, y otros ataques
[OK] **Usar HTTPS**	Siempre encripta en tránsito (TLS 1.2+)	Protege datos contra interceptación
[OK] **Principio de menor privilegio**	IAM roles con mínimos permisos necesarios	Reduce superficie de ataque
[OK] **Rotar secretos regularmente**	Usa AWS Secrets Manager con rotación automática	Limita el impacto de secretos comprometidos
[OK] **Logging sin información sensible**	No loguees passwords, tokens o datos personales	Previene exposición de datos sensibles en logs
[OK] **Rate limiting**	Protege contra abuso y ataques DDoS	Previene sobrecarga del sistema
[OK] **Validación de origen (CORS)**	Restringe orígenes permitidos en API Gateway	Previene ataques CSRF

Práctica	Descripción	Impacto
[OK] **Auditoría**	Activa CloudTrail para rastrear cambios	Facilita detección de actividades sospechosas
[OK] **Encriptación en reposo**	Encripta datos sensibles en DynamoDB	Protege datos incluso si la base de datos es comprometida
[OK] **Actualizar dependencias**	Mantén las librerías actualizadas	Previene vulnerabilidades conocidas

8.11.2 Recomendaciones adicionales:

- **Usar API Gateway Authorizers**: Valida tokens antes de invocar Lambda, reduciendo costos
- **Implementar WAF (Web Application Firewall)**: Protege contra ataques comunes (SQL injection, XSS, etc.)
- **Monitorear con CloudWatch**: Configura alarmas para detectar patrones sospechosos
- **Usar VPC para funciones sensibles**: Aísla funciones que acceden a recursos privados
- **Implementar secretos rotatorios**: Configura rotación automática en Secrets Manager
- **Revisar permisos regularmente**: Audita IAM roles y políticas periódicamente

---

8.12 Conclusión de la sección

Implementamos mecanismos de seguridad robustos para funciones Lambda reactivas:

Mecanismo	Implementación	Beneficio
IAM con menor privilegio	Políticas específicas con ARNs limitados	Reduce superficie de ataque
Manejo seguro de secretos	AWS Secrets Manager con cache reactivo	Elimina hardcoding de credenciales
Autenticación	Cognito y validación JWT con filtros reactivos	Autenticación escalable y gestionada
Validación de entrada	Bean Validation y sanitización de datos	Previene inyecciones y ataques
Rate limiting distribuido	DynamoDB con TTL para control de flujo	Protege contra abuso y DDoS
Encriptación	Encriptación de datos sensibles en reposo y tránsito	Protege datos contra exposición

8.12.1 Lecciones clave aprendidas:

1. **La seguridad es responsabilidad compartida**: AWS gestiona la infraestructura, tú gestionas la aplicación
2. **El principio de menor privilegio es fundamental**: Solo otorga los permisos mínimos necesarios
3. **Nunca hardcodees secretos**: Usa AWS Secrets Manager o variables de entorno seguras
4. **Valida y sanitiza toda la entrada**: Nunca confíes en datos del cliente

5. **Monitorea y audita regularmente**: CloudTrail y CloudWatch son tus aliados

Con estas prácticas, tus funciones Lambda estarán protegidas y listas para producción, cumpliendo con estándares de seguridad empresariales.

8.13 Recursos adicionales

- AWS Lambda Security Best Practices
- AWS Secrets Manager Documentation
- Amazon Cognito Developer Guide
- OWASP Top 10

9 Prólogo e Historia del Autor

9.1 Un cambio de paradigma

Durante más de una década, el desarrollo backend con Java se centró en arquitecturas monolíticas y entornos estáticos. Sin embargo, el crecimiento de la computación en la nube, la demanda de escalabilidad y la búsqueda de eficiencia energética marcaron un cambio profundo: **el paradigma reactivo y serverless**.

Este e-book nació precisamente de ese punto de inflexión. La necesidad de crear sistemas más ágiles, con menor consumo de recursos y preparados para responder en tiempo real, impulsó la integración de **Spring Boot 3.4+, Java 25 LTS, WebFlux y AWS Lambda** en un mismo flujo de trabajo.

9.2 El camino detrás del libro

El contenido que leés aquí surge de experiencias reales en proyectos productivos, horas de optimización, pruebas con distintos entornos y mucha experimentación con herramientas modernas como **GraalVM**, **LocalStack**, **SAM** y **Micrometer**.

El objetivo no fue simplemente documentar conceptos, sino **demostrar que Java sigue siendo una tecnología líder para construir soluciones cloud-native de alto rendimiento**.
Donde antes se necesitaban servidores dedicados, ahora basta con código optimizado que escala automáticamente.

9.3 Sobre el autor

Marcos Raimundo Lozina es ingeniero de software especializado en backend, con amplia experiencia en **Java, Spring Boot, Gradle y AWS**.
Combina su pasión por la programación con una visión emprendedora que busca aplicar tecnología para crear proyectos rentables, sostenibles y educativos.

Además de su trabajo como desarrollador, Marcos ha publicado materiales educativos, e-books y contenidos audiovisuales sobre desarrollo moderno en Java, productividad en proyectos y automatización en la nube.

> "La evolución del software no se trata solo de nuevas tecnologías, sino de la capacidad de adaptarse a los cambios sin perder claridad ni propósito."

9.4 Propósito del e-book

Este libro está pensado para: - Desarrolladores Java que quieren dar el salto hacia **arquitecturas reactivas y serverless**.
- Equipos que buscan **reducir costos y mejorar rendimiento** en la nube.
- Profesionales que desean integrar **eficiencia, observabilidad y automatización** en un mismo flujo.

Mi meta con este contenido fue **simplificar la curva de adopción** del paradigma reactivo, y mostrar que, con las herramientas adecuadas, cualquier desarrollador puede construir sistemas escalables y elegantes sin necesidad de una infraestructura compleja.

9.5 Agradecimientos personales

A mi familia, por su paciencia y apoyo en cada proyecto.
A los colegas y lectores que dan vida a estas ideas, compartiendo conocimiento y feedback.
Y a la comunidad open source, que demuestra día a día que **la colaboración es la base de la innovación**.

> *"El futuro del software pertenece a quienes no temen reinventar sus herramientas."*

9.6 Contacto y recursos adicionales

Email: marcos.lozina.dev@gmail.com
GitHub: MarcosLozina
LinkedIn: Marcos Raimundo Lozina
Fiverr: Perfil de servicios
Amazon: Java 21 multimódulo con Gradle 8 y Spring Boot 3

10 Glosario y Apéndice

10.1 Glosario técnico

Término	Definición
Reactive Programming	Paradigma basado en flujos de datos asíncronos que reaccionan ante cambios.
Mono / Flux	Tipos reactivos de Project Reactor: `Mono` emite 0..1 valor, `Flux` emite 0..N valores.
Backpressure	Mecanismo que regula la velocidad del flujo de datos para evitar sobrecarga.

Término	Definición
Serverless	Modelo de ejecución donde la infraestructura es gestionada automáticamente por el proveedor cloud.
AWS Lambda	Servicio de AWS para ejecutar funciones bajo demanda sin gestionar servidores.
Spring WebFlux	Módulo reactivo de Spring Framework basado en Project Reactor.
Spring Cloud Function	Framework para desarrollar funciones desacopladas del entorno de ejecución (web, stream, Lambda, etc.).
Spring Cloud 2024.0.0	Versión de Spring Cloud requerida para Spring Boot 3.4+. Las versiones anteriores (2023.0.1) no son compatibles.
GraalVM	JVM optimizada que permite compilación nativa (AOT) para tiempos de arranque ultrarrápidos.
LocalStack	Herramienta para emular servicios AWS localmente.
AWS SAM	Modelo Serverless Application Model, facilita despliegues e infraestructura declarativa.
CI/CD	Integración y entrega continua: automatiza build, testing y despliegue.
Micrometer	Librería de métricas usada por Spring Boot para monitoreo y observabilidad.
CloudWatch	Servicio AWS para registrar logs y métricas del sistema.
X-Ray	Servicio AWS para trazas distribuidas y diagnóstico de rendimiento.
Event-driven architecture	Arquitectura basada en eventos que disparan reacciones entre servicios.
Cold start	Retraso inicial al iniciar una función Lambda por primera vez.
AOT Compilation	Compilación “Ahead-of-Time”: genera binarios nativos en lugar de bytecode.
ASM 9.8	Librería requerida para leer archivos de clase Java 25 (class file version 69).
Gradle 9.2.1	Versión recomendada de Gradle para mejor soporte de Java 25.
Kotlin JVM Target 24	Target requerido porque Kotlin aún no soporta completamente Java 25.

10.2 Comandos útiles

10.2.1 Gradle

```
./gradlew clean build
./gradlew bootJar
./gradlew nativeCompile
```

10.2.2 AWS CLI

```
aws lambda list-functions
aws s3 ls
aws dynamodb list-tables
aws sqs list-queues
aws sns list-topics
```

10.2.3 SAM CLI

```
sam build
sam local invoke "FunctionName"
sam deploy --guided
```

10.2.4 Docker / LocalStack

```
docker-compose up -d
awslocal dynamodb list-tables
awslocal sqs send-message --queue-url URL --message-body "Hola"
```

10.3 Checklist de despliegue serverless reactivo

[OK] Revisión del `build.gradle.kts` con dependencias correctas.
[OK] Verificación de compilación nativa (`nativeCompile`).
[OK] Pruebas locales con SAM y LocalStack.
[OK] Configuración de IAM Roles mínimos necesarios.
[OK] Logs habilitados en CloudWatch.
[OK] Métricas visibles en CloudWatch Metrics.
[OK] Pipeline de CI/CD automatizado.
[OK] Test de performance posterior al despliegue.

10.4 Recursos recomendados

10.4.1 Documentación oficial

- Spring WebFlux Reference
- AWS Lambda Developer Guide
- Project Reactor Documentation
- Spring Cloud Function
- GraalVM Official Documentation

- LocalStack Docs
- AWS SAM Documentation

10.4.2 Proyecto de referencia y ebook

- **Proyecto de código**: `reactive-microservices-aws-lambda-java25` - Repositorio con todo el código de ejemplo del ebook
- **Ebook completo**: Este ebook complementa el proyecto de código con explicaciones detalladas, mejores prácticas y guías paso a paso

10.4.3 Archivos clave del proyecto

- `lambda-core/src/main/java/com/example/lambda/FunctionConfig.java` - Configuración de funciones Lambda reactivas
- `lambda-core/src/main/java/com/example/lambda/handlers/HelloHandler.java` - Handler reactivo con métricas
- `lambda-core/src/main/java/com/example/lambda/repository/` - Repositorios reactivos con DynamoDB
- `lambda-core/src/main/java/com/example/lambda/service/` - Servicios que integran SQS y SNS
- `lambda-core/src/main/java/com/example/lambda/security/` - Implementaciones de seguridad
- `lambda-infra/template.yaml` - Template SAM para despliegue en AWS
- `build.gradle.kts` - Configuración de dependencias y compilación nativa

  ⚠ **Nota importante sobre correspondencia**: Todo el código del ebook usa el paquete `com.example.lambda`, que coincide exactamente con el proyecto real. El código mostrado en el ebook es el mismo código que encontrarás en el repositorio del proyecto. Todas las referencias a archivos del proyecto usan `com.example.lambda`.

10.5 Mapeo entre ebook y proyecto

Nota: Todos los archivos Java están en el paquete `com.example.lambda` dentro de `lambda-core/src/main/java/com/example/lambda/`. Los paths mostrados en la tabla son relativos a la raíz del proyecto.

Sección del ebook	Archivos del proyecto
Sección 2: Configuración del entorno	`build.gradle.kts`, `settings.gradle.kts`, `buildSrc/`
Sección 3: Fundamentos de Spring WebFlux	`controller/`, `service/`, `repository/`
Sección 4: Función Lambda reactiva	`FunctionConfig.java`, `handlers/HelloHandler.java`
Sección 5: GraalVM Native	`build.gradle.kts` (configuración nativa), `resources/META-INF/native-image/reflect-config.json`
Sección 6: Integración con AWS	`repository/`, `service/`

Sección del ebook	Archivos del proyecto
Sección 7: Observabilidad y CI/CD	`.github/workflows/deploy.yml`, `observability/`
Sección 9: Seguridad	`security/`, `lambda-infra/template.yaml` (políticas IAM)

> ⚠ **Importante**: Todo el código del ebook usa el paquete `com.example.lambda` y coincide exactamente con el proyecto real. El código mostrado en el ebook es el mismo código que encontrarás en el repositorio del proyecto.

10.6 Notas finales

Este apéndice fue diseñado como una referencia rápida y práctica para el lector.
Podés incluirlo en tus propios proyectos o consultarlo al momento de implementar microservicios reactivos serverless.

> *"La documentación es el mejor compañero de la innovación."*

11 Control de Costos y Disclaimers Legales

11.1 ⚠ DISCLAIMER LEGAL IMPORTANTE

ADVERTENCIA CRÍTICA SOBRE COSTOS DE AWS:

Este ebook contiene ejemplos y configuraciones que utilizan servicios de AWS que **generan costos reales**. Al seguir los ejemplos y desplegar recursos en AWS, **aceptas la responsabilidad completa** de todos los costos incurridos.

El autor, editor y distribuidores de este ebook NO se hacen responsables de: - Costos de AWS incurridos al seguir los ejemplos - Facturas inesperadas por uso de servicios AWS - Errores de configuración que resulten en costos elevados - Uso indebido o malinterpretación de los ejemplos

RECOMENDACIONES CRÍTICAS: 1. **Siempre configura presupuestos y alertas** antes de desplegar cualquier recurso 2. **Usa AWS Free Tier** cuando sea posible para pruebas 3. **Elimina recursos inmediatamente** después de las pruebas 4. **Monitorea costos diariamente** durante el desarrollo 5. **Lee la documentación oficial de AWS** sobre precios antes de usar servicios

Este ebook es material educativo. El autor no garantiza que los ejemplos sean apropiados para tu situación específica. Siempre valida costos y configuraciones antes de desplegar en producción.

Al usar este ebook, **aceptas estos términos** y liberas al autor de cualquier responsabilidad relacionada con costos de AWS.

11.2 Objetivo de esta Sección

Aprender a **controlar y monitorear costos** al trabajar con AWS Lambda y servicios relacionados, evitando facturas inesperadas y optimizando el gasto en la nube.

> **En la práctica**: Los costos de AWS pueden dispararse rápidamente si no los controlás. Esta sección te muestra cómo protegerte de facturas inesperadas y optimizar tus gastos.

11.3 Entendiendo los Costos de AWS Serverless

11.3.1 Servicios Principales y sus Modelos de Precio

Servicio	Modelo de Precio	Free Tier	Costo Típico
AWS Lambda	Por invocación + GB-segundo	1M invocaciones/mes gratis	$0.20 por 1M invocaciones
API Gateway HTTP API	Por request	1M requests/mes gratis	$1.00 por 1M requests
API Gateway REST API	Por request + datos transferidos	1M requests/mes gratis	$3.50 por 1M requests
DynamoDB On-Demand	Por request (read/write)	25 GB almacenamiento gratis	$1.25 por 1M write units
DynamoDB Provisioned	Capacidad reservada	25 GB almacenamiento gratis	Más barato si tráfico predecible
SQS	Por request	1M requests/mes gratis	$0.40 por 1M requests
SNS	Por notificación	1M notificaciones/mes gratis	$0.50 por 1M notificaciones
EventBridge	Por evento	1M eventos/mes gratis	$1.00 por 1M eventos
CloudWatch Logs	Por GB almacenado	5 GB/mes gratis	$0.50 por GB
X-Ray	Por trace	100K traces/mes gratis	$5.00 por 1M traces

> ⚠ **IMPORTANTE**: Los precios pueden variar por región y están sujetos a cambios. Siempre consultá la calculadora de precios de AWS antes de desplegar.

11.4 Protección contra Facturas Inesperadas

11.4.1 1. Configurar Presupuestos y Alertas de AWS

Paso crítico antes de desplegar cualquier recurso:

```
# Crear presupuesto mensual de $10 USD
aws budgets create-budget \
  --account-id $(aws sts get-caller-identity --query Account --output text) \
  --budget file://budget.json \
  --notifications-with-subscribers file://notifications.json
```

Archivo `budget.json`:

```
{
  "BudgetName": "Monthly-Budget-Serverless",
  "BudgetLimit": {
    "Amount": "10",
    "Unit": "USD"
  },
  "TimeUnit": "MONTHLY",
  "BudgetType": "COST",
  "TimePeriod": {
    "Start": "2024-01-01T00:00:00Z"
  }
}
```

Archivo `notifications.json`:

```
[
  {
    "Notification": {
      "NotificationType": "ACTUAL",
      "ComparisonOperator": "GREATER_THAN",
      "Threshold": 80,
      "ThresholdType": "PERCENTAGE"
    },
    "Subscribers": [
      {
        "SubscriptionType": "EMAIL",
        "Address": "tu-email@example.com"
      }
    ]
  },
  {
    "Notification": {
      "NotificationType": "FORECASTED",
      "ComparisonOperator": "GREATER_THAN",
      "Threshold": 100,
      "ThresholdType": "PERCENTAGE"
    },
    "Subscribers": [
      {
        "SubscriptionType": "EMAIL",
        "Address": "tu-email@example.com"
      }
```

```
    ]
  }
]
```

11.4.2 2. Configurar Límites de Costo con AWS Cost Anomaly Detection

```
# Habilitar detección de anomalías de costo
aws ce create-anomaly-monitor \
  --anomaly-monitor-name "Serverless-Cost-Monitor" \
  --monitor-type "DIMENSIONAL" \
  --monitor-dimension "SERVICE"
```

11.4.3 3. Usar AWS Free Tier

AWS Free Tier incluye (primeros 12 meses): - Lambda: 1M invocaciones/mes gratis - API Gateway: 1M requests/mes gratis - DynamoDB: 25 GB almacenamiento gratis - SQS: 1M requests/mes gratis - SNS: 1M notificaciones/mes gratis - CloudWatch: 5 GB logs/mes gratis

> ⚠ **ADVERTENCIA**: El Free Tier tiene límites. Si los excedés, se te cobrará el uso adicional.

11.5 Optimización de Costos por Servicio

11.5.1 AWS Lambda

11.5.1.1 Factores que Afectan el Costo:

1. **Número de invocaciones**
2. **Memoria asignada** (afecta GB-segundo)
3. **Tiempo de ejecución**
4. **Reserved Concurrency** (no afecta costo directamente, pero limita escalado)

```
# Template SAM optimizado para costos
Resources:
  OptimizedFunction:
    Type: AWS::Serverless::Function
    Properties:
      MemorySize: 256  # ▫ Menor memoria = menor costo (pero puede ser más lento)
      Timeout: 10      # ▫ Timeout corto evita ejecuciones largas costosas
      ReservedConcurrentExecutions: 5  # Limitar concurrencia para controlar costos
      Environment:
        Variables:
          LOG_LEVEL: WARN  # Reducir logging para ahorrar en CloudWatch
```

11.5.1.2 Optimizaciones: Cálculo de costo estimado:

```
Costo mensual = (Invocaciones × $0.20/1M) + (GB-segundo × $0.0000166667/GB-segundo)
```

```
Ejemplo:
- 100K invocaciones/mes
- 256 MB memoria
- 500ms tiempo promedio
- GB-segundo = 100,000 × 0.256 × 0.5 = 12,800 GB-segundo

Costo = (0.1M × $0.20) + (12,800 × $0.0000166667) = $0.02 + $0.21 = $0.23/mes
```

> ⚠ **ADVERTENCIA**: Con tráfico alto, los costos pueden escalar rápidamente. Siempre monitoreá invocaciones y GB-segundo.

11.5.2 API Gateway

11.5.2.1 HTTP API vs REST API (Impacto en Costos):

Aspecto	HTTP API	REST API
Costo por 1M requests	$1.00	$3.50
Costo de datos transferidos	Incluido	$0.09/GB
Ahorro	**60% más barato**	-

> ⚠ **RECOMENDACIÓN**: Siempre usá HTTP API para ahorrar costos (60% más barato).

```yaml
# ▫ NOTA: HTTP API (v2) tiene rate limiting diferente a REST API (v1)
# Para rate limiting avanzado en HTTP API, considera usar AWS WAF
Resources:
  ReactiveApi:
    Type: AWS::Serverless::HttpApi
    Properties:
      # HTTP API tiene rate limiting básico por defecto
      # Para rate limiting avanzado, usa AWS WAF:
      # - Crear WebACL con rate-based rules
      # - Asociar WebACL al API Gateway
      Description: API Gateway HTTP API con rate limiting básico
```

11.5.2.2 Optimizaciones:

> ⚠ **Nota sobre Rate Limiting:**
>
> HTTP API (v2) tiene rate limiting básico por defecto. Para rate limiting avanzado (como `ThrottleSettings` de REST API), necesitás usar **AWS WAF** con rate-based rules. Esto agrega costo adicional ($1.00 por 1M requests + $0.60 por 1M requests evaluados por WAF).
>
> **Recomendación:** Para desarrollo/testing, el rate limiting básico de HTTP API es suficiente. Para producción con necesidades avanzadas, considera AWS WAF.

11.5.3 DynamoDB

11.5.3.1 On-Demand vs Provisioned:

Tipo	Cuándo usar	Costo
On-Demand	Tráfico impredecible, desarrollo	Pagás por lo que usás
Provisioned	Tráfico predecible, producción	Más barato si usás toda la capacidad

> ⚠ **ADVERTENCIA**: DynamoDB Provisioned puede ser **muy caro** si no usás toda la capacidad. On-Demand es más seguro para desarrollo.

```
# Usar On-Demand para desarrollo (más seguro)
aws dynamodb create-table \
  --table-name Quotes \
  --billing-mode PAY_PER_REQUEST  # ▫ On-Demand: pagás solo por lo que usás

# ▫ EVITAR Provisioned en desarrollo a menos que sepas exactamente el tráfico
```

11.5.3.2 Optimizaciones: Costo estimado DynamoDB On-Demand:

```
Write Units: $1.25 por 1M write units
Read Units: $0.25 por 1M read units
Almacenamiento: $0.25 por GB/mes (después de Free Tier)

Ejemplo (100K writes/mes):
Costo = 0.1M × $1.25 = $0.125/mes
```

> ⚠ **ADVERTENCIA**: Scan operations son **muy costosas**. Siempre usá Query con GSI.

11.5.4 CloudWatch Logs

```
# Configurar retención de logs para ahorrar costos
Resources:
  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: /aws/lambda/ReactiveFunction
      RetentionInDays: 7  # ▫ Reducir retención para ahorrar (default: nunca expira)
```

11.5.4.1 Optimizaciones: Costo estimado CloudWatch Logs:

```
$0.50 por GB almacenado/mes

Ejemplo:
- 1 GB logs/mes
- Retención: 7 días
- Costo = $0.50/mes
```

> ⚠ **ADVERTENCIA**: Sin retención configurada, los logs se acumulan indefinidamente y pueden generar costos altos.

11.5.5 X-Ray

```java
// Configurar sampling para reducir costos
@Configuration
public class XRayConfig {
    @Bean
    public AWSXRayRecorder xRayRecorder() {
        // Sampling del 10% en producción (ajustar según necesidad)
        LocalSamplingStrategy samplingStrategy = new LocalSamplingStrategy(0.1);

        return AWSXRayRecorderBuilder.standard()
            .withSamplingStrategy(samplingStrategy)
            .build();
    }
}
```

11.5.5.1 Optimizaciones: Costo estimado X-Ray:

```
$5.00 por 1M traces

Ejemplo con 10% sampling:
- 1M requests/mes
- 10% sampling = 100K traces
- Costo = 0.1M × $5.00 = $0.50/mes
```

> ⚠ **ADVERTENCIA**: Sin sampling, X-Ray puede ser costoso con alto tráfico.

11.6 Monitoreo de Costos en Tiempo Real

11.6.1 1. AWS Cost Explorer

```bash
# Ver costos del día actual
aws ce get-cost-and-usage \
  --time-period Start=$(date +%Y-%m-01),End=$(date +%Y-%m-%d) \
  --granularity DAILY \
  --metrics BlendedCost
```

11.6.2 2. CloudWatch Billing Alarms

```yaml
Resources:
  BillingAlarm:
    Type: AWS::CloudWatch::Alarm
    Properties:
      AlarmName: Monthly-Billing-Alarm
      MetricName: EstimatedCharges
      Namespace: AWS/Billing
      Statistic: Maximum
      Period: 86400  # 24 horas
      EvaluationPeriods: 1
```

```
      Threshold: 10  # ▫ Alertar si supera $10 USD
      ComparisonOperator: GreaterThanThreshold
      AlarmActions:
        - !Ref BillingAlarmTopic

  BillingAlarmTopic:
    Type: AWS::SNS::Topic
    Properties:
      TopicName: billing-alerts
      Subscription:
        - Protocol: email
          Endpoint: tu-email@example.com
```

11.6.3 3. Script de Monitoreo Diario

```
#!/bin/bash
# Script para monitorear costos diarios

TODAY=$(date +%Y-%m-%d)
COST=$(aws ce get-cost-and-usage \
  --time-period Start=$TODAY,End=$TODAY \
  --granularity DAILY \
  --metrics BlendedCost \
  --query 'ResultsByTime[0].Total.BlendedCost.Amount' \
  --output text)

echo "Costo del día $TODAY: \$$COST USD"

# Alertar si supera $1 USD/día
if (( $(echo "$COST > 1.0" | bc -l) )); then
  echo "▫ ADVERTENCIA: Costo diario supera $1 USD"
  # Enviar notificación (email, Slack, etc.)
fi
```

11.7 Checklist de Protección de Costos

Antes de desplegar cualquier recurso, verificá:

- ☐ **Presupuesto configurado** con alertas al 80% y 100%
- ☐ **Billing alarms** configurados
- ☐ **Free Tier** verificado (si aplica)
- ☐ **Retención de logs** configurada (no infinita)
- ☐ **Sampling de X-Ray** configurado (si se usa)
- ☐ **Timeouts de Lambda** configurados (no muy altos)
- ☐ **Memoria de Lambda** optimizada (no más de lo necesario)
- ☐ **DynamoDB On-Demand** usado en desarrollo
- ☐ **HTTP API** usado en lugar de REST API

- ☐ **Recursos marcados** para fácil identificación
- ☐ **Script de limpieza** preparado para eliminar recursos después de pruebas

---

11.8 Script de Limpieza de Recursos

```
#!/bin/bash
# Script para eliminar todos los recursos de prueba

echo "⚠ ADVERTENCIA: Esto eliminará TODOS los recursos del stack"
read -p "¿Estás seguro? (yes/no): " confirm

if [ "$confirm" != "yes" ]; then
  echo "Operación cancelada"
  exit 1
fi

# Eliminar stack SAM
sam delete --stack-name reactive-microservices-stack

# Eliminar tablas DynamoDB
aws dynamodb delete-table --table-name Quotes 2>/dev/null || true

# Eliminar colas SQS
aws sqs delete-queue --queue-url $(aws sqs get-queue-url --queue-name quotes-events --query Que

# Eliminar tópicos SNS
aws sns delete-topic --topic-arn $(aws sns list-topics --query 'Topics[?contains(TopicArn, `quo

# Eliminar log groups
aws logs delete-log-group --log-group-name /aws/lambda/ReactiveFunction 2>/dev/null || true

echo "✓ Recursos eliminados"
```

> ⚠ **ADVERTENCIA**: Este script elimina recursos permanentemente. Usalo solo en desarrollo/testing.

11.9 Estimación de Costos para Ejemplos del Ebook

11.9.1 Escenario 1: Desarrollo/Testing (Bajo Tráfico)

Servicio	Uso Mensual	Costo
Lambda	10K invocaciones	$0.002
API Gateway	10K requests	$0.01
DynamoDB	10K writes	$0.00125
CloudWatch Logs	0.1 GB	$0.05

Servicio	Uso Mensual	Costo
TOTAL	-	**~$0.06/mes**

Dentro del Free Tier para la mayoría de servicios.

11.9.2 Escenario 2: Producción (Tráfico Medio)

Servicio	Uso Mensual	Costo
Lambda	1M invocaciones	$0.20
API Gateway	1M requests	$1.00
DynamoDB	500K writes	$0.625
SQS	500K requests	$0.20
CloudWatch Logs	5 GB	$2.50
X-Ray (10% sampling)	100K traces	$0.50
TOTAL	-	**~$5.00/mes**

⚠ **ADVERTENCIA**: Estos son estimados. Los costos reales pueden variar significativamente.

11.9.3 Escenario 3: Alto Tráfico (⚠ CUIDADO)

Servicio	Uso Mensual	Costo
Lambda	10M invocaciones	$2.00
API Gateway	10M requests	$10.00
DynamoDB	5M writes	$6.25
SQS	5M requests	$2.00
CloudWatch Logs	50 GB	$25.00
X-Ray (10% sampling)	1M traces	$5.00
TOTAL	-	**~$50.00/mes**

ADVERTENCIA CRÍTICA: Con alto tráfico, los costos pueden escalar rápidamente. Siempre configurá presupuestos y alertas.

11.10 ⚠ DISCLAIMER FINAL

El autor de este ebook NO se hace responsable de:

1. **Costos de AWS** incurridos al seguir los ejemplos
2. **Facturas inesperadas** por uso de servicios AWS
3. **Errores de configuración** que resulten en costos elevados
4. **Malinterpretación** de los ejemplos o instrucciones
5. **Uso indebido** de los servicios AWS

Este ebook es material educativo. Los ejemplos son para propósitos de aprendizaje y no garantizan resultados específicos en tu entorno.

Siempre: - Consultá la documentación oficial de AWS sobre precios - Usá la calculadora de precios de AWS antes de desplegar - Configurá presupuestos y alertas - Monitoreá costos regularmente - Eliminá recursos después de pruebas

Al usar este ebook, aceptas estos términos y liberas al autor de cualquier responsabilidad relacionada con costos de AWS.

11.11 Recursos Adicionales

- AWS Pricing Calculator
- AWS Free Tier
- AWS Cost Management
- AWS Budgets Documentation
- AWS Cost Anomaly Detection

Última actualización: 2024-2025