



Universidad
Rey Juan Carlos



Máster en Cloud Apps
Desarrollo y despliegue de aplicaciones en la nube

Curso académico 2019/2020

Trabajo de Fin de Máster

**Sistema de venta de seguros basado en
una arquitectura de microservicios**

Autor: François Poirier
Tutor: Francisco Gortázar

Índice de contenidos

1. Introducción	4
1.1 Documentos de referencia	4
1.2 Definiciones y glosario	4
1.3 Control de modificaciones sobre el documento	4
2. Proyecto	5
2.1 Introducción	5
2.2 Casos de usos implementados	7
3. Diseño, implementación y testing de un microservicio	8
3.1 Principios de arquitectura Hexagonal	8
3.2 Principio de arquitectura CRQS / Event Sourcing	10
3.3 Desarrollo del microservicio	12
3.3.1 Open Api	12
Dependencia	12
Configuración	12
3.3.2 Async Api	15
3.3.3 Arquitectura Hexagonal + CQRS	17
Business core	17
Domain model	17
Ports	17
3.3.4 Gestión de cambios en base de datos con liquibase	21
3.3.5 Integración Postgresql	25
Dependencia	25
Configuración del Datasource	25
Configuración de Docker para Spring Boot con PostgreSQL	25
El archivo Dockerfile	25
El archivo docker-compose.yml	26
Prueba de integración	27
3.3.6 Integración MongoDB	30
Dependencia	30
Configuración mongodb	30
Configuración de Docker para Spring Boot con MongoDB	31
El archivo Dockerfile	31
3.3.7 Integración Spring Cloud Stream Kafka	32
Dependencia	32



Configuración	32
Publicación de un evento (command).	34
Consumo de un evento (query).	37
3.3.8 Api Gateway	39
Características y funcionalidades proporcionadas por Spring Cloud Gateway	39
Dependencia	40
Configuración	40
3.4 Componentes elevados a nivel de arquitectura	42
3.4.1 Clases de soporte para CQRS	42
3.4.2 Health Check	42
Dependencia	42
Habilitar Endpoints	42
Deshabilitar Endpoints	43
/health	44
3.4.3 Intercepción de mensajes/errores: Feign Decoder	46
3.4.4 Manejador de excepciones	46
3.4.5 Clases de soporte para testing	46
4. Integración continua	47
4.1 Definición de un workflow	48
4.2 Presentación de la solución de integración continua	49
4.2.1 Configuración de secretos	49
4.2.2 Workflows	51
4.2.2.1 Flujo nightly	51
Definición del flujo nightly	52
4.2.2.2 Flujo release	56
Definición del flujo release	56
Configuración del proyecto y de maven	57
4.2.3 Presentación de los informes	60
5. Despliegue en Kubernetes (minikube)	62
5.1 Instalar y configurar Minikube en Ubuntu	62
5.2 Despliegue del proyecto en minikube	63
6. Conclusiones y próximos pasos	67
6.1 Beneficios de una arquitectura de microservicios	67
6.2 Inconvenientes de una arquitectura de microservicios	68
6.3 Próximos pasos	69
7. Anexos	70

1. Introducción

1.1 Documentos de referencia

Título	Autor	Descripción
Get Your Hands Dirty on Clean Architecture	Tom Hombergs	libro que podemos encontrar en la siguiente url https://leanpub.com/get-your-hands-dirty-on-clean-architecture
Organizing Layers Using Hexagonal Architecture, DDD, and Spring	baeldung	Artículo https://www.baeldung.com/hexagonal-architecture-ddd-spring
Domain Driven Design Distilled	Vaughn Vernon	libro que podemos encontrar en la siguiente url https://www.oreilly.com/library/view/domain-driven-design-distilled/9780134434964/

1.2 Definiciones y glosario

Término	Definición

1.3 Control de modificaciones sobre el documento

Fecha	Versión	Capítulo afectado	Observaciones	Autor
28/11/2020	1.0	Todos	Versión inicial	F.Poirier Troyano

2. Proyecto

2.1 Introducción

El trabajo fin de máster consiste en desarrollar una solución de microservicios para un sistema de ventas de seguros simplificado, basándose en los paradigmas de arquitectura CQRS Event Sourcing y arquitectura Hexagonal.

Esta prueba de concepto sencilla va permitir exponer las distintas tecnologías y buenas prácticas visto durante el máster, entre otras:

- Creación y desarrollo de microservicios
- Acceso a base de datos relacional y NoSQL
- Comunicación síncrona y asíncrona entre servicios
- Operaciones bloqueantes y no bloqueantes.
- Patrones de microservicios (Service Config, Service Discovery, Service Registry, Api Gateway, etc ..)

La solución backend está compuesta de los siguientes servicios :

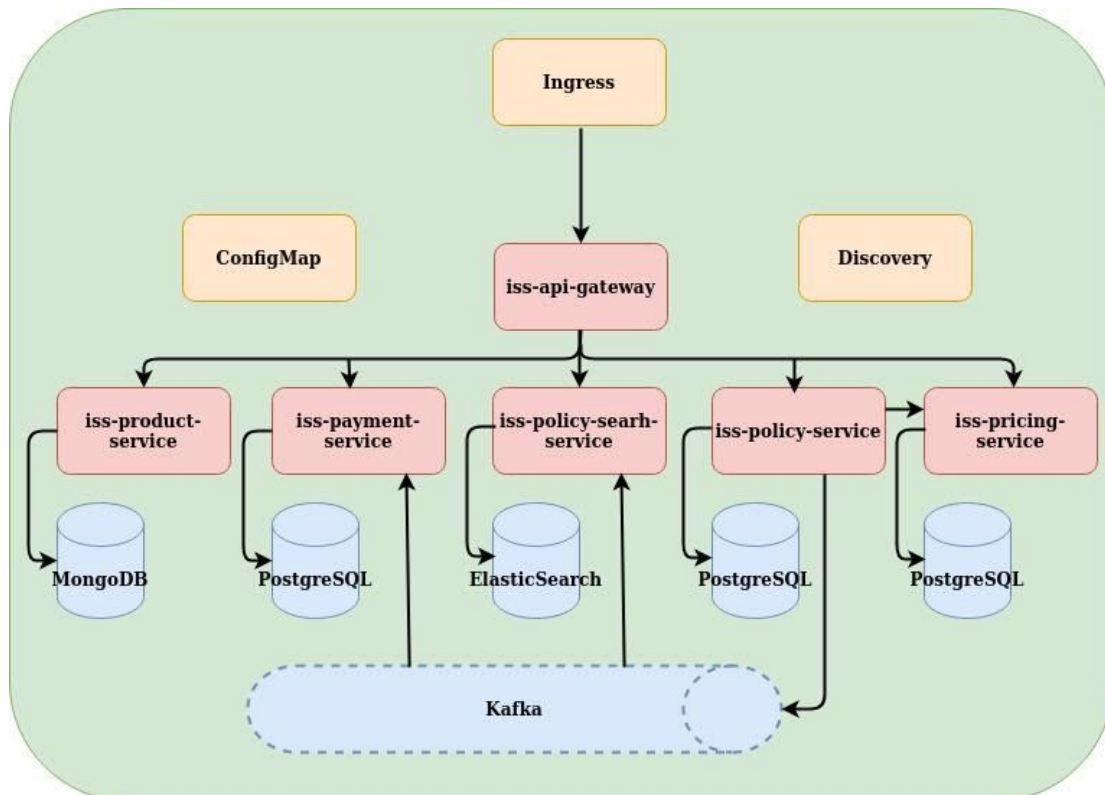
- **policy-service:** Este servicio se encarga de la creación de ofertas, las convierten en pólizas de seguro y permite su correcta gestión hasta que finalicen. En este servicio se aplicará el uso del **patrón CQRS** para un mejor aislamiento de las operaciones de lecturas / escrituras. Se implementará dos formas de comunicación entre servicios: llamadas síncronas basadas en REST (Feign Client) al servicio de pricing-service para obtener el precio, y llamadas asíncronas basadas en eventos utilizando el broker Apache Kafka para publicar en un Topic información sobre pólizas recién creadas y finalizadas.
- **policy-search-service:** Este servicio se encarga de exponer un endpoint REST de búsqueda de pólizas, fruto del consumo de eventos del Topic Kafka interior y convertidos en un modelo de vista de lectura persistidos en un elasticsearch (este elemento está simulado por un mock de tipo map debido a una limitación de recursos a la hora de desplegar la solución en un minikube).
- **payment-service:** Este servicio se encarga de gestionar las cuentas de pólizas. Cuando llega el evento de creación de póliza, este servicio se encarga de crear la cuenta para



Universidad Rey Juan Carlos

poder atender a los futuros pagarés esperados. El servicio de pago se implementa con un proceso batch que lee desde un fichero CSV los pagos asignados a cuentas de pólizas.

- **pricing-service:** Este servicio calcula el precio para el producto de seguro seleccionado. Para cada producto, se define una tarifa. La tarifa se compone de un conjunto de reglas sobre cuya base se calcula el precio. Se utiliza el lenguaje MVEL para definir estas reglas. Durante el proceso de compra de la **póliza**, el **servicio de póliza policy-service** solicita a este servicio el cálculo del precio. El precio se calcula en función de las respuestas del usuario a las preguntas definidas en la oferta.
- **product-service:** Este servicio se encarga de gestionar un simple catálogo de productos de seguros. El catálogo de productos se almacena en una MongoDB. Cada producto se caracteriza por un código, un nombre, una imagen, una descripción, una lista de coberturas de pólizas y una lista de preguntas. Esto últimos elementos afectaba el precio definido por la tarifa.
- **api-gateway-service:** Este servicio se basa en el patrón api-gateway.



2.2 Casos de usos implementados

En esta prueba de concepto encontraremos los siguientes casos de usos:

Microservicios	Caso de Uso
iss-policy-service	Creación de oferta.
iss-policy-service	Creación de póliza.
iss-policy-service	Finalización de póliza.
iss-policy-service	Consulta de póliza, búsqueda por el número de póliza.
iss-pricing-service	Cálculo del precio de la oferta a partir de las respuestas relativas a las preguntas de las distintas coberturas del producto personalizado (la oferta).
iss-product-service	Consulta de los productos.
iss-product-service	Consulta de un producto a partir del código.
iss-policy-search-service	Búsqueda indexada de pólizas
iss-payment-service	.Consulta de las cuentas de pólizas
iss-payment-service	Consulta de una cuenta de póliza, búsqueda por número de cuenta.
iss-payment-service	Actualización de los pagarés de cuentas de pólizas por batch.

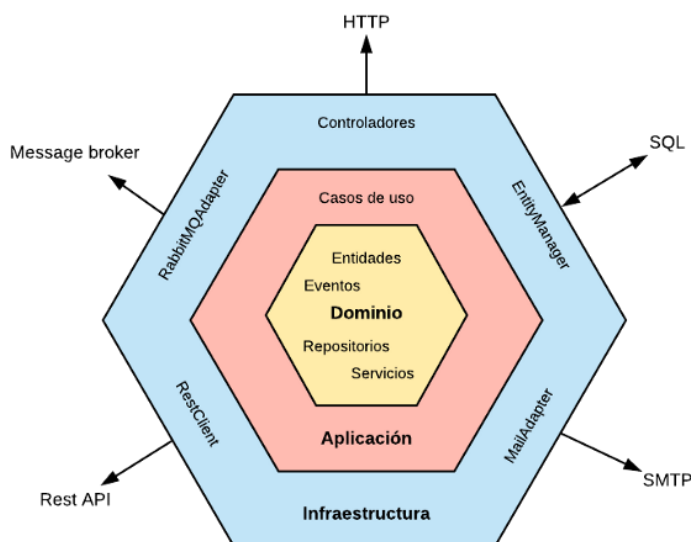
Falta por implementar los pagarés, procesando un fichero CSV cada día a las 8h00 de la mañana. Esta funcionalidad se implementará utilizando Spring Batch.

3. Diseño, implementación y testing de un microservicio

3.1 Principios de arquitectura Hexagonal

La Arquitectura Hexagonal, dada a conocer por [Alistair Cockburn](#) y también conocida como arquitectura de puertos y adaptadores, tiene como principal motivación separar nuestra aplicación en distintas capas o regiones con su propia responsabilidad. De esta manera consigue desacoplar capas de nuestra aplicación permitiendo que evolucionen de manera aislada. Además, tener el sistema separado por responsabilidades nos facilitará la reutilización.

Gracias a este desacoplamiento obtenemos también la ventaja de poder testear estas capas sin que intervengan otras externas, falseando el comportamiento con dobles de pruebas, por ejemplo. Esta arquitectura se suele representar con forma de hexágono, pero el número de lados no es lo que importa, sino lo que estos representan. Cada lado representa un puerto hacia dentro o fuera de la aplicación. Por ejemplo, un puerto puede ser el HTTP, y hacer peticiones a nuestra aplicación, otro puerto puede ser el SOAP y también hace peticiones a la aplicación. Otro puede ser un servidor de base de datos en donde persisten los datos de nuestro dominio.





Universidad
Rey Juan Carlos

La Arquitectura Hexagonal propone que nuestro dominio sea el núcleo de las capas y que este no se acople a nada externo. En lugar de hacer uso explícito y mediante el principio de inversión de dependencias nos acoplamos a contratos (interfaces o puertos) y no a implementaciones concretas.

A grandes rasgos, y sin mucho detalle, lo que propone es que nuestro núcleo sea visto como una API con unos contratos bien especificados. Definiendo puertos o puntos de entrada e interfaces (adaptadores) para que otros módulos (UI, BBDD, Test) puedan implementarlas y comunicarse con la capa de negocio sin que ésta deba saber el origen de la conexión. Esto es lo llamado puertos y adaptadores, que podrían ser definidos de la siguiente manera:

- Puerto: definición de una interfaz pública.
- Adapter: especialización de un puerto para un contexto concreto.

Pues bien, en el apartado [3.3](#) ya veremos una implementación de este paradigma de arquitectura.

3.2 Principio de arquitectura CRQS / Event Sourcing

Command query responsibility segregation (CQRS) es un patrón de arquitectura que se fundamenta en la separación entre la persistencia de datos y la consulta de los mismos. Esta característica supone la aplicación del principio de única responsabilidad (SRP), que tantos sistemas monolíticos incumplen (en algunos casos por falta de necesidad o por utilizar frameworks orientados a CRUD), y que garantiza que estas dos responsabilidades con requisitos tan distintos puedan cumplir con los mismos de forma autónoma. De esta manera, los sistemas que aplican este diseño, están constituidos por dos subsistemas completamente independientes: el de consultas y el de comandos. Ambos módulos del sistema presentan diseño, modelo de datos y paradigma de persistencia distintos, optimizados para cada necesidad, aunque obviamente deben estar conectados entre sí y esto se realiza mediante sincronización.

El módulo de consultas es bastante más sencillo y suele estar compuesto por una primera fase de filtrado y transformación, de la información que le llega del módulo de comandos vía sincronización, para a continuación consolidarla en su modelo de datos orientado a que las consultas sean óptimas y que suele ser SQL o NOSQL, dependiendo de la necesidad.

El módulo de comandos es el receptor de las peticiones y del procesamiento de las mismas. Cada petición que es procesada supone un cambio de estado del sistema y tiene asociada un proceso de sincronización que hará posible que el módulo de consulta sea poseedor de la información (por PULL, PUSH o como sea).

Al profundizar en las diferentes maneras de diseñar el módulo de comandos es cuando entra en juego Event Sourcing.

El patrón de arquitectura Event Sourcing se fundamenta en una premisa: Toda la información del sistema debe estar vinculada a sus estados y debe guardarse como una secuencia ordenada de eventos, siendo un evento el mecanismo que posibilita un cambio de estado del sistema.

Este paradigma rompe con los modelos de datos relacionales y con el patrón ORM ya que no es necesario persistir las entidades ni relacionarlas entre sí. Solo basta con tener una secuencia ordenada de eventos serializados ya que esto contiene toda la información del sistema. Los tipos de repositorio para almacenar la información que mejor encajan son: sistemas Nosql orientados a documentos como MongoDB, sistemas clave valor como Redis o incluso escritura en fichero (hay que tener en cuenta que solo existe inserción no aplica ni borrado ni actualizaciones) y también sería posible una BBDD SQL tradicional aunque no es lo más óptimo.

Volviendo a la sincronización entre los módulos de consulta y comandos, vemos ahora que la situación más óptima es la solicitud desde el subsistema de consultas de los eventos persistidos desde la última sincronización o en caso de producirse algún problema podría restablecerse desde un estado anterior o incluso desde el origen.



Universidad
Rey Juan Carlos

Pues con todo esto estamos en disposición de apreciar las importantes ventajas que supone combinar los patrones de arquitectura CQRS y Event Sourcing.

- Escalabilidad
- Resiliencia
- Responsividad
- Comunicación asíncrona
- Rendimiento
- Recuperación y restablecimiento del sistema en caso de fallo o pérdida de información en el módulo de consultas.
- Sistema de auditoría avanzado de serie(intrínseco en el diseño)

En el caso de la prueba de concepto, se proporciona un ejemplo entre los microservicios **iss-policy-service** y **iss-policy-search-service** con la ayuda de Spring Cloud Stream y Kafka.

3.3 Desarrollo del microservicio

3.3.1 Open Api

Si necesitamos pasar a producción rápidamente, o crear un prototipo, "Code First" puede ser un buen enfoque. Luego podemos generar nuestra documentación a partir de la API que ya hemos programado.

Otro beneficio del "Code First" es el hecho de que la documentación se generará a partir del código, lo que significa que no tenemos que mantener manualmente la documentación sincronizada con nuestro código. Es más probable que la documentación coincida con el comportamiento del código y siempre esté actualizada.

Pues bien, para eso utilizaremos Spring Boot junto con **springdoc-openapi** en cada microservicio.

Dependencia

Para agregar la dependencia a nuestro proyecto maven, insertamos lo siguiente en el bloque de dependencias:

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-ui</artifactId>
  <version>1.4.6</version>
</dependency>
```

Configuración

Primero, definamos la ruta de nuestra documentación en el archivo **application.yaml** del microservicio.

```
springdoc:
  api-docs:
    path: /policy-openapi
```



Universidad
Rey Juan Carlos

A continuación, tenemos que configurar en una clase anotada con un `@Configuration` un Bean de tipo **OpenApi** como en el siguiente ejemplo:

```
@Configuration
public class OpenApiConfig {

    @Bean
    public OpenAPI customOpenAPI() {
        return new OpenAPI()
            .components(new Components())
            .info(new Info().title("Policy service API").description(
                "This is a policy service API using springdoc-openapi and OpenAPI 3.").version("1.0.0"));
    }
}
```

Definimos el api REST, con las anotaciones proporcionadas por la librería.

Con la anotación `@Tag`, agregamos información adicional a la API. Ahora, tenemos que implementar esta interfaz y anotar nuestro controlador con `@RestController`. Esto le permitirá a Springdoc saber que se trata de un controlador y que debería producir una documentación para él. A continuación un ejemplo de implementación de un Controller documentado.

```
@RestController
@Tag(name = "offers", description = "the Offers API")
@Validated
public class OffersCommandController {

    private final Bus bus;

    @Autowired
    public OffersCommandController(Bus bus) {
        this.bus = bus;
    }

    @Operation(summary = "Create offer", description = "", tags = { "offers" })
    @ApiResponse(value = {
        @ApiResponse(responseCode = "201", description = "Create offer",
            content = @Content(schema = @Schema(implementation = CreateOfferResult.class))),
        @ApiResponse(responseCode = "400", description = "Invalid input")})
    @PostMapping("/api/v1/offers")
    public ResponseEntity<CreateOfferResult> create(@Parameter(description="Create offer command. Cannot null or empty.",
        required=true, schema=@Schema(implementation = CreateOfferCommand.class)) @Valid @RequestBody CreateOfferCommand cmd) {
        final CreateOfferResult response = bus.executeCommand(cmd);
        return ResponseEntity.status(HttpStatus.CREATED).body(response);
    }
}
```

Ahora, si arrancamos la aplicación Spring Boot y navegamos hasta la url <http://localhost:8082/swagger-ui/index.html?configUrl=/policy-openapi/swagger-config>, deberíamos ver la información que definimos anteriormente:

Policy service API 1.0.0 OAS3

/policy-openapi

This is a policy service API using springdoc-openapi and OpenAPI 3.

Servers

http://localhost:8082 - Generated server url

offers the Offers API

POST /api/v1/offers Create offer

policies the policies API

POST /api/v1/policies/terminate Terminate policy

POST /api/v1/policies Create policy

GET /api/v1/policies/{policyNumber} Get policy by policy number

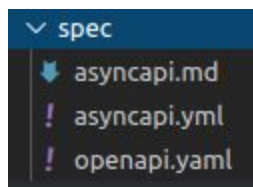
Schemas

ErrorResponse >

TerminatePolicyCommand >

TerminatePolicyResult >

Podemos encontrar en la carpeta spec de cada microservicio el archivo openapi.yaml generado.



3.3.2 Async Api

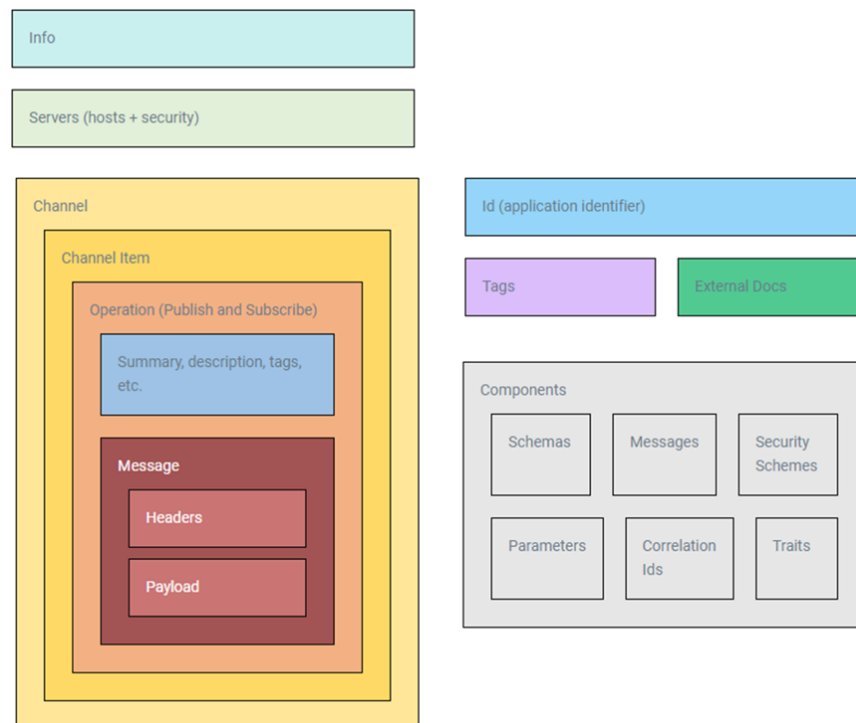
AsyncAPI es una especificación liderada por Fran Méndez que permite definir APIs y que se enfoca en arquitecturas orientadas en eventos (EDA).

El objetivo de la iniciativa AsyncAPI es conseguir que trabajar con EDA sea tan fácil como es hoy en día trabajar con APIs RESTful. Uno de los motivos del éxito de las APIs REST ha sido entre otros la existencia de estándares como OpenAPI que permiten editar especificaciones, generar documentación automáticamente, portales de desarrollo, generadores de código, generadores de mocks, etc.

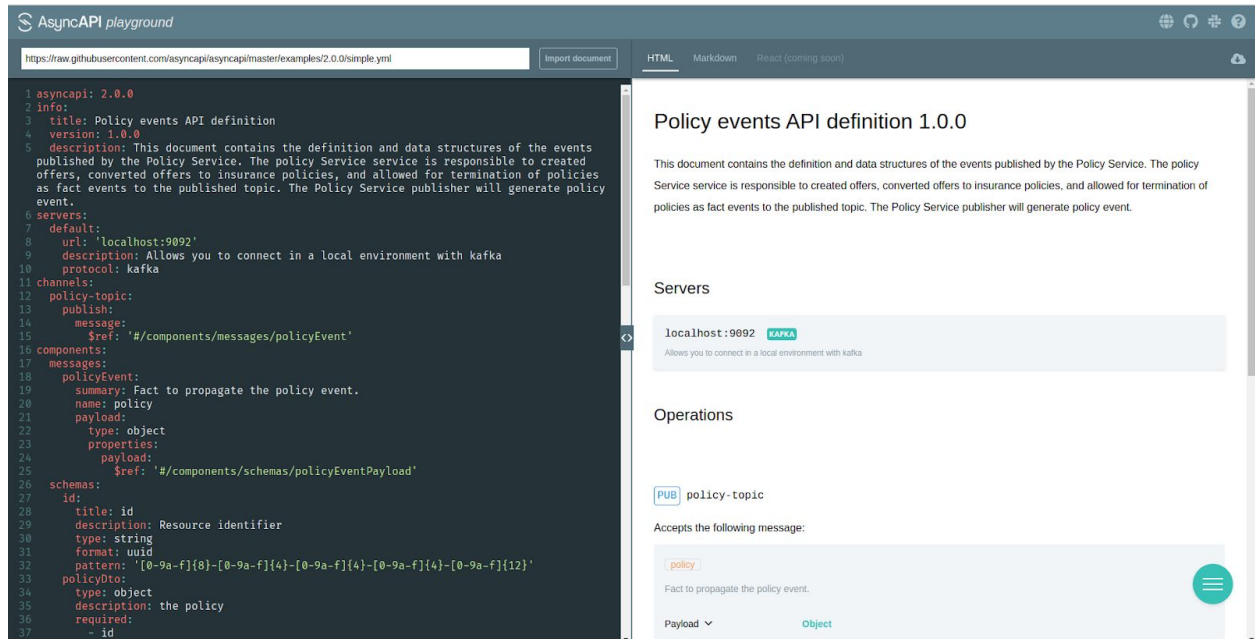
<https://playground.asyncapi.io/?load=https://raw.githubusercontent.com/asyncapi/asyncapi/master/examples/2.0.0/simple.yml>

Se trata de una especificación que es amigable tanto para máquinas como para humanos. Es agnóstica del protocolo, de tal manera que se puede utilizar para describir APIs que funcionan sobre diferentes protocolos como MQTT o WebSockets.

Estos son los principales elementos de la especificación:



En el contexto del proyecto, necesitamos definir una API que describa los escenarios asíncronos mediante el cual cada vez que un cliente nuevo contra una nueva póliza o finaliza dicha póliza, un consumidor es avisado mediante un evento, de tal manera que este puede realizar algún tipo de acción.



The screenshot shows the AsyncAPI playground interface. On the left, the AsyncAPI definition is displayed in a code editor. The definition includes information about the API (title, version, description), servers (localhost:9092), channels (policy-topic), and components (messages, schemas). The right panel shows the rendered HTML output, which includes a title 'Policy events API definition 1.0.0', a description, a list of servers, and a section for operations (policy-topic).

```

1 asyncapi: 2.0.0
2 info:
3   title: Policy events API definition
4   version: 1.0.0
5   description: This document contains the definition and data structures of the events
6     published by the Policy Service. The policy Service service is responsible to created
7     offers, converted offers to insurance policies, and allowed for termination of policies
8     as fact events to the published topic. The Policy Service publisher will generate policy
9     event.
10  servers:
11    default:
12      url: 'localhost:9092'
13      description: Allows you to connect in a local environment with kafka
14      protocol: kafka
15  channels:
16    policy-topic:
17      publish:
18        message:
19          $ref: '#/components/messages/policyEvent'
20  components:
21    messages:
22      policyEvent:
23        summary: Fact to propagate the policy event.
24        name: policy
25        payload:
26          type: object
27          properties:
28            payload:
29              $ref: '#/components/schemas/policyEventPayload'
30    schemas:
31      id:
32        title: id
33        description: Resource identifier
34        type: string
35        format: uuid
36        pattern: '[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}'
37      policyDto:
38        type: object
39        description: the policy
40        required:
41          - id
  
```

Policy events API definition 1.0.0

This document contains the definition and data structures of the events published by the Policy Service. The policy Service service is responsible to created offers, converted offers to insurance policies, and allowed for termination of policies as fact events to the published topic. The Policy Service publisher will generate policy event.

Servers

localhost:9092 **KAFKA**

Allows you to connect in a local environment with kafka

Operations

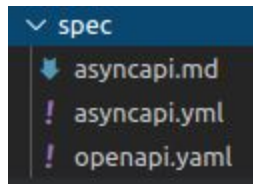
PUB policy-topic

Accepts the following message:

Fact to propagate the policy event.

Payload **Object**

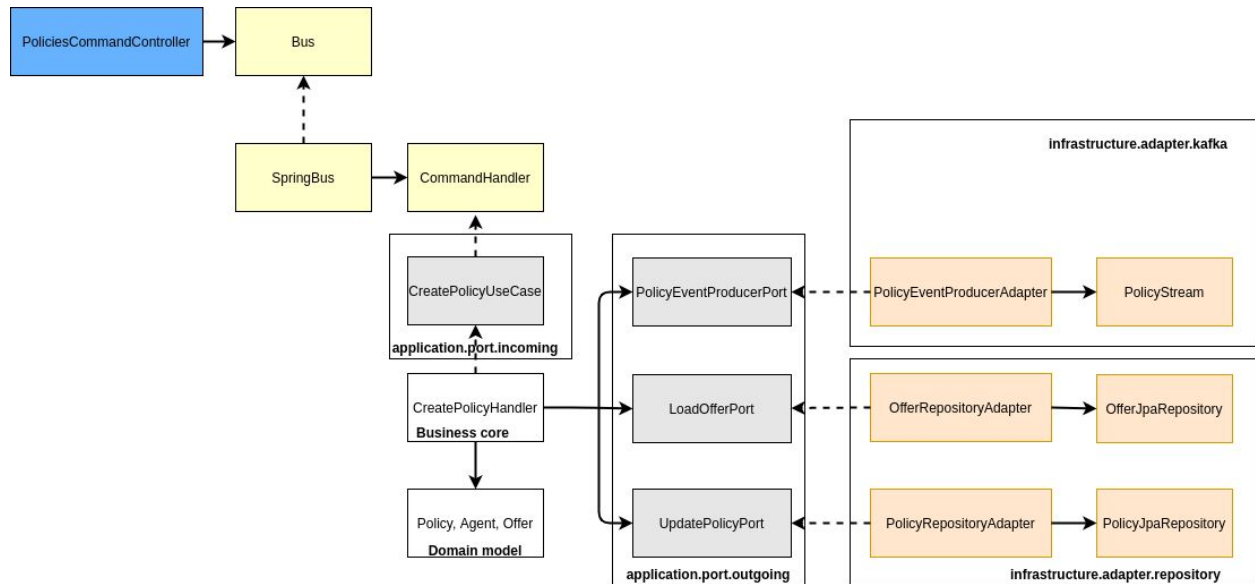
Podemos encontrar en la carpeta spec del microservicio **iss-policy-service** el archivo **asyncapi.yaml** y el **asyncapi.md** generados por el editor.





3.3.3 Arquitectura Hexagonal + CQRS

A continuación presentamos un ejemplo de caso de uso que está estructurado siguiendo los principios de arquitectura Hexagonal y CQRS.



La arquitectura hexagonal es un estilo arquitectónico que se centra en mantener la lógica empresarial dissociada de las preocupaciones externas. En cuenta la arquitectura CQRS es un estilo arquitectónico que se fundamenta en la separación de las preocupaciones, como explicado en el apartado [3.2 Principio de arquitectura CRQS y Event Sourcing](#).

Business core

La capa de negocio (business core) interactúa con otros componentes a través de puertos y adaptadores. De esta forma, podemos cambiar las tecnologías subyacentes sin tener que modificar el núcleo de la aplicación.

Domain model

La principal responsabilidad de la capa de dominio es modelar las reglas comerciales. También verifica que los objetos estén siempre en un estado válido. El modelo de dominio no debe depender de ninguna tecnología específica.

Ports

Ahora es el momento de que nuestra lógica de negocio interactúe con el mundo exterior. Para lograr esto, presentaremos algunos puertos.



Primero, definamos 1 puerto entrante. Estos son utilizados por componentes externos para llamar a nuestra aplicación. En este caso, tendremos uno por caso de uso. Uno para la creación de póliza.

```
public interface CreatePolicyUseCase extends CommandHandler<CreatePolicyResult,CreatePolicyCommand> {  
    public CreatePolicyResult handle(CreatePolicyCommand command);  
}
```

Del mismo modo, también tendremos 3 puertos de salida. Estos son para que nuestra aplicación interactúe con la base de datos y el broker de mensajería Kafka.

- uno para recuperar la oferta
- otro para crear la póliza
- y por último, un último para publicar un evento de póliza creada.

```
public interface LoadOfferPort {  
    public Offer getOffer(String offerNumber);  
}  
  
public interface UpdatePolicyPort {  
    public Policy updateTerminateState(String policyNumber);  
    public Policy createPolicy(Offer offer, Person policyHolder, AgentRef agent);  
}  
  
public interface PolicyEventProducerPort {  
    public void terminated(Policy policy);  
    public void registered(Policy policy);  
}
```

A continuación, crearemos un servicio o un componente para unir todas las piezas e impulsar la ejecución:

```
@Component
public class CreatePolicyHandler implements CreatePolicyUseCase {

    private final LoadOfferPort loadOfferPort;
    private final UpdatePolicyPort updatePolicyPort;
    private final PolicyEventProducerPort policyEventProducerPort;

    @Autowired
    public CreatePolicyHandler (LoadOfferPort loadOfferPort, UpdatePolicyPort updatePolicyPort, PolicyEventProducerPort policyEventProducerPort) {
        this.loadOfferPort = loadOfferPort;
        this.updatePolicyPort = updatePolicyPort;
        this.policyEventProducerPort = policyEventProducerPort;
    }

    @Transactional
    @Override
    public CreatePolicyResult handle(CreatePolicyCommand command) {

        final Person policyHolder = new Person.Builder()
            .withFirstName(command.getPolicyHolder().getFirstName())
            .withLastName(command.getPolicyHolder().getLastName())
            .withPesel(command.getPolicyHolder().getTaxId())
            .build();
        final AgentRef agent = new AgentRef.Builder().withLogin(command.getAgentLogin()).build();
        final Offer offer = loadOfferPort.getOffer(command.getOfferNumber());
        final Policy policy = updatePolicyPort.createPolicy(offer, policyHolder, agent);
        policyEventProducerPort.registered(policy);
        return new CreatePolicyResult.Builder().withPolicyNumber(policy.getNumber()).build();
    }
}
```

Para completar nuestra aplicación, necesitamos proporcionar implementaciones para los puertos definidos. A estos los llamamos adaptadores.

Adjunto enlace a los distintos adaptadores:

- [OfferRepositoryAdapter](#)
- [PolicyRepositoryAdapter](#)
- [PolicyEventProducerAdapter](#)

Para las interacciones entrantes, crearemos un controlador REST:

```
@RestController
@Tag(name = "policies", description = "the policies API")
@Validated
public class PoliciesCommandController {

    private final Bus bus;

    @Autowired
    public PoliciesCommandController(Bus bus) {
        this.bus = bus;
    }

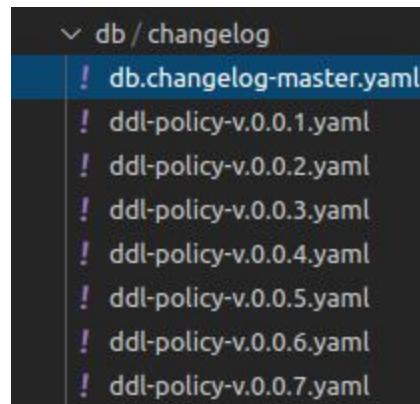
    @Operation(summary = "Create policy", description = "", tags = { "policies" })
    @ApiResponses(value = {
        @ApiResponse(responseCode = "201", description = "Create policy",
            content = @Content(schema = @Schema(implementation = CreatePolicyResult.class))),
        @ApiResponse(responseCode = "400", description = "Invalid input")})
    @PostMapping("/api/v1/policies")
    public ResponseEntity<CreatePolicyResult> create(@Parameter(description="Create policy command. Cannot null or empty.",
        required=true, schema=@Schema(implementation = CreatePolicyCommand.class)) @Valid @RequestBody CreatePolicyCommand cmd) {
        final CreatePolicyResult response = bus.executeCommand(cmd);
        return ResponseEntity.status(HttpStatus.CREATED).body(response);
    }

    @Operation(summary = "Terminate policy", description = "", tags = { "policies" })
    @ApiResponses(value = {
        @ApiResponse(responseCode = "201", description = "Terminate policy",
            content = @Content(schema = @Schema(implementation = TerminatePolicyResult.class))),
        @ApiResponse(responseCode = "400", description = "Invalid input")})
    @PostMapping("/api/v1/policies/terminate")
    public ResponseEntity<TerminatePolicyResult> terminate(@Parameter(description="Terminate policy command. Cannot null or empty.",
        required=true, schema=@Schema(implementation = TerminatePolicyCommand.class)) @Valid @RequestBody TerminatePolicyCommand cmd) {
        final TerminatePolicyResult response = bus.executeCommand(cmd);
        return ResponseEntity.status(HttpStatus.CREATED).body(response);
    }
}
```

3.3.4 Gestión de cambios en base de datos con liquibase

La herramienta soporta 2 sintaxis de definición de un cambio xml o yaml y cada changeLog puede tener varios changeSet, estos últimos deben tener un identificador único por autor.

De forma predeterminada, Spring Boot configura automáticamente Liquibase cuando agregamos la dependencia Liquibase. Al declarar esta dependencia, Spring Boot espera encontrar en el classpath un archivo db.changelog-master en la ruta db/changelog.



Liquibase se basa en el concepto de changeLog, que define y documenta todos los cambios en el modelo de datos.

La recomendación es mantener un fichero de changeLog por versión liberada, de modo que tras liberar una nueva versión añadiremos al fichero maestro de changeLog la referencia al nuevo número de versión.

```
databaseChangeLog:
- include:
  file: db/changelog/ddl-policy-v.0.0.1.yaml
- include:
  file: db/changelog/ddl-policy-v.0.0.2.yaml
- include:
  file: db/changelog/ddl-policy-v.0.0.3.yaml
- include:
  file: db/changelog/ddl-policy-v.0.0.4.yaml
- include:
  file: db/changelog/ddl-policy-v.0.0.5.yaml
- include:
  file: db/changelog/ddl-policy-v.0.0.6.yaml
- include:
  file: db/changelog/ddl-policy-v.0.0.7.yaml
```

La herramienta nos proporciona una serie de etiquetas con las que construir nuestro conjunto de cambios. Lo cierto es que es bastante intuitivo, y como podemos apreciar, tanto el nombre de las etiquetas, como el nombre de los atributos de las mismas nos van a dar pistas sobre cuál es la operación que van a realizar.

```
databaseChangeLog:
- changeSet:
  id: 01_policy_create_schema
  author: fpoirier
  changes:
  - sql:
    sql: CREATE SCHEMA IF NOT EXISTS policy

- changeSet:
  id: 02_create_policy_table
  author: fpoirier
  preConditions:
  - onFail: MARK_RAN
    not:
      tableExists:
        schemaName: policy
        tableName: policy
  changes:
  - createTable:
    schemaName: policy
    tableName: policy
    columns:
    - column:
      name: id
      type: UUID
      constraints:
        primaryKeyName: policy_pkey
        primaryKey: true
        nullable: false
    - column:
      name: agent_login
      type: VARCHAR(50)
      constraints:
        nullable: false
    - column:
      name: number
      type: VARCHAR(10)
      constraints:
        nullable: false
```

El objetivo es incorporar un control de cambios de la base de datos a la arquitectura del proyecto, lo que nos proporciona los siguientes beneficios:

- mantener un histórico de cada una de las modificaciones sobre el modelo de datos y, con ello, sensación de control,
- ejecución automática de los cambios gracias a la integración de liquiBase con Spring Boot, no será necesario ejecutar manualmente los scripts en los distintos entornos en



Universidad
Rey Juan Carlos

los que se despliegue la aplicación, se ejecutarán al arranque de la aplicación si no se han ejecutado ya; de este modo el despliegue desde el entorno de integración continua se hará de forma transparente sin necesidad de ejecutar manualmente script alguno en base de datos.

Pues bien en el caso de los siguientes microservicios, el control de cambios de la base de datos está gestionado por liquibase:

- **iss-pricing-service**
- **iss-policy-service**
- **iss-payment-service**

3.3.5 Integración Postgresql

Vamos a ver cómo integrar Spring Boot con una base de datos PostgreSQL para que se ejecute dentro de un contenedor Docker.

Dependencia

Necesitamos definir las siguientes dependencias para utilizar Spring Boot con Postgresql.

```
<dependency>
|   <groupId>org.springframework.boot</groupId>
|   <artifactId>spring-boot-starter-data-jpa</artifactId>
| </dependency>
<dependency>
|   <groupId>org.postgresql</groupId>
|   <artifactId>postgresql</artifactId>
| </dependency>
```

Configuración del Datasource

Definimos en el archivo application.yaml que tendrá la base de datos con la url de la conexión, el username y el password. Veremos más adelante que estos valores por defecto salen de la imagen PostgreSQL que usaremos en el archivo de configuración del docker compose.

```
spring:
| datasource:
|   url: jdbc:postgresql://${POSTGRES_HOST:localhost}:${POSTGRES_PORT:5432}/${POSTGRES_DB:postgres}
|   username: ${POSTGRES_USER:postgres}
|   password: ${POSTGRES_PASSWORD:postgres123}
|   hikari:
|     connection-timeout: 2000
|     initialization-fail-timeout: 0
|   jpa.database-platform: org.hibernate.dialect.PostgreSQL9Dialect
```

Configuración de Docker para Spring Boot con PostgreSQL

El archivo Dockerfile

En el archivo de Dockerfile le daremos las siguientes instrucciones a Docker.



Universidad
Rey Juan Carlos

```
FROM maven:3.6.3-jdk-11-slim AS BUILD_STAGE
RUN mkdir -p /workspace
WORKDIR /workspace
COPY pom.xml /workspace
COPY src /workspace/src
RUN mvn -B -f pom.xml clean package -DskipTests

FROM openjdk:11-jdk-slim
COPY --from=BUILD_STAGE /workspace/target/iss-pricing-service*.jar iss-pricing-service.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "iss-pricing-service.jar"]
```

El archivo *docker-compose.yml*

Debemos crear un archivo *docker-compose.yml* en el cual vas a establecer dos servicios. El primer servicio que llamaremos como el nombre del microservicio hace referencia al servicio de Spring Boot que se configuró en el Dockerfile. Dentro de ese servicio la instrucción *build* indica que ese servicio viene del Dockerfile que previamente definimos.

```
version: '3'

services:
  iss-pricing-service:
    container_name: iss-pricing-service
    image: iss-pricing-service
    build:
      context: ../
      dockerfile: ../docker/Dockerfile
    environment:
      - SPRING_PROFILES_ACTIVE=test
      - POSTGRES_HOST=postgres_db
      - POSTGRES_DB=postgres
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres123
      - POSTGRES_PORT=5432
    ports:
      - "8080:8080"
    depends_on:
      - postgres_db

  postgres_db:
    image: "postgres:9.6-alpine"
    container_name: postgres_db
    volumes:
      - postgres-data:/var/lib/postgresql/data
    ports:
      - "5432:5432"
    environment:
      - POSTGRES_HOST=postgres_db
      - POSTGRES_DB=postgres
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres123

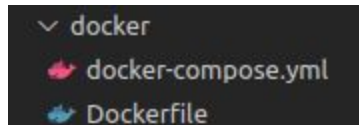
volumes:
  postgres-data:
```



Universidad
Rey Juan Carlos

El segundo servicio que llamarás `postgres_db` utiliza una imagen `postgresql` del hub de docker que docker descargará de allí. Dejamos el `host`, `password`, `user`, y nombre de la db que viene por defecto.

En todos los proyectos se ha proporcionado una carpeta **docker** con el archivo **Dockerfile** del servicio y un archivo **docker-compose.yml** para poder arrancar el servicio de forma autónoma en local.



Prueba de integración

Antes de ver cómo realizar una prueba de integración con `postgresql` vamos a incluir las dependencias necesarias para poder ejecutar tests de integración, como sigue:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.junit.vintage</groupId>
      <artifactId>junit-vintage-engine</artifactId>
    </exclusion>
  </exclusions>
  <scope>test</scope>
</dependency>
<!-- test container support for postgresql module -->
<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>postgresql</artifactId>
  <version>${testcontainer.version}</version>
  <scope>test</scope>
</dependency>
<!-- JUnit 5 integration with testcontainer -->
<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>${testcontainer.version}</version>
  <scope>test</scope>
</dependency>
```

La clase repositorio es una implementación para el acceso a la base de datos haciendo uso de las facilidades que proporciona Spring Data. La interfaz **JpaRepository** ofrece métodos con las operaciones de lecturas básicas que en su invocación generan las consultas SQL select correspondientes.

```
@Repository
public interface TariffJpaRepository extends JpaRepository<Tariff, UUID>{

    @Query("SELECT t FROM Tariff t LEFT JOIN FETCH t.discountMarkupRules where t.code=:code")
    public Optional<Tariff> findByCode(@Param("code") String code);

    @Query("SELECT t FROM Tariff t LEFT JOIN FETCH t.discountMarkupRules where t.code=:code")
    public Tariff getByCode(@Param("code") String code);
}
```

Para probar su comportamiento en una prueba de integración se usa **TestContainers** que arranca el contenedor Docker de **PostgreSQL**. La prueba está implementada con JUnit 5 y la aplicación hace uso de Spring Boot. Para eso, se implementa una clase abstracta que se puede reutilizar.

```
@Testcontainers
public abstract class AbstractContainerBaseTest {

    protected static final PostgreSQLContainer postgresContainer = new PostgreSQLContainer("postgres:9.6.15").withDatabaseName("pricing")
        .withUsername("postgres").withPassword("password");

    @BeforeAll
    static void setUpAll() {
        if (!postgresContainer.isRunning()) {
            postgresContainer.start();
        }
    }

    public static class PropertiesInitializer implements ApplicationContextInitializer<ConfigurableApplicationContext> {
        public void initialize(ConfigurableApplicationContext configurableApplicationContext) {
            TestPropertyValues
                .of("spring.datasource.driver-class-name=" + postgresContainer.getDriverClassName(),
                    "spring.datasource.url=" + postgresContainer.getJdbcUrl(),
                    "spring.datasource.username=" + postgresContainer.getUsername(),
                    "spring.datasource.password=" + postgresContainer.getPassword())
                .applyTo(configurableApplicationContext.getEnvironment());
        }
    }
}
```

La clase **PostgreSQLContainer** permite encapsular el inicio del contenedor para Postgresql y configurar las variables spring.datasource.url, spring.datasource.username y spring.datasource.password con la URL de conexión, usuario y contraseña antes de que el contexto de Spring se inicie. La clase **PostgreSQLContainer** permite reutilizar esta configuración en diferentes tests y hacer uso de ella donde sea necesario con la anotación ContextConfiguration.

Una vez hecho esto vamos a comprobar las consultas implementadas con el siguiente test de integración:

```
@DataJpaTest
@AutoConfigureTestDatabase(replace = Replace.NONE)
@ContextConfiguration(initializers = { AbstractContainerBaseTest.PropertiesInitializer.class })
@ActiveProfiles("test")
class TariffJpaRepositoryIT extends AbstractContainerBaseTest {

    private static final String CODE_NOT_EXIST = "NOT_EXIST";
    private static final String CODE_CAR = "CAR";

    @Autowired
    private TariffJpaRepository jpaTariffRepository;

    private Tariff entity;

    @BeforeEach
    public void setUp() {
        entity = new Tariff(CODE_CAR);
        entity.addBasePriceRule("C1", null, "100B");
        entity.addPercentMarkup(new PercentMarkupRule("NUM_OF_CLAIM > 2", new BigDecimal("50.00")));
        jpaTariffRepository.save(entity);
    }

    @Test
    void testWhenFindByCodeThenReturnTariff() {
        var t = jpaTariffRepository.findByCode(CODE_CAR);
        assertTrue(t.isPresent());
    }

    @Test
    void testWhenFindByCodeThenNotReturnTariff() {
        var t = jpaTariffRepository.findByCode(CODE_NOT_EXIST);
        assertTrue(!t.isPresent());
    }

    @Test
    void testWhenGetByCodeThenReturnTariff() {
        var t = jpaTariffRepository.getByCode(CODE_CAR);
        assertEquals(CODE_CAR, t.getCode());
    }
}
```



3.3.6 Integración MongoDB

Vamos a ver cómo integrar Spring Boot con una base de datos MongoDB para que se ejecute dentro de un contenedor Docker.

Dependencia

Necesitamos definir la siguiente dependencia para utilizar Spring Boot con MongoDB.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

Configuración mongodb

Definimos en el archivo application.yaml la propiedad uri de conexión con el username, el password y el nombre de la base de datos mongodb. Veremos más adelante que estos valores por defecto salen de la imagen Mongodb que usaremos en el archivo de configuración del docker compose.

```
spring:
  data:
    mongodb:
      uri: mongodb://${MONGO_DB_HOST}:${MONGO_DB_PORT}/${MONGO_DB_NAME}
```


Configuración de Docker para Spring Boot con MongoDB

El archivo Dockerfile

En el archivo de Dockerfile le daremos las siguientes instrucciones a Docker.

```
FROM maven:3.6.3-jdk-11-slim AS BUILD_STAGE
RUN mkdir -p /workspace
WORKDIR /workspace
COPY pom.xml /workspace
COPY src /workspace/src
RUN mvn -B -f pom.xml clean package -DskipTests

FROM openjdk:11-jdk-slim
COPY --from=BUILD_STAGE /workspace/target/iss-product-service*.jar iss-product-service.jar
EXPOSE 8081
ENTRYPOINT ["java", "-jar", "iss-product-service.jar"]
```

Debemos crear un archivo docker-compose.yml en el cual vas a establecer dos servicios. El primer servicio que llamaremos como el nombre del microservicio hace referencia al servicio de Spring Boot que se configuró en el Dockerfile. Dentro de ese servicio la instrucción build indica que ese servicio viene del Dockerfile que previamente definimos.

```
version: '3.2'
services:
  iss-product-service:
    container_name: iss-product-service
    image: iss-product-service
    build:
      context: ../
      dockerfile: ../docker/Dockerfile
    environment:
      - SPRING_PROFILES_ACTIVE=dev
      - MONGO_DB_HOST=mongodb
      - MONGO_DB_PORT=27017
      - MONGO_DB_NAME=MDB
    ports:
      - "8081:8081"
    depends_on:
      - mongo_db
  mongo_db:
    image: mongo:4.2.6
    container_name: "mongodb"
    ports:
      - "27017:27017"
    volumes:
      - mongodb_data_container:/data/db
    restart: always

volumes:
  mongodb_data_container:
```



3.3.7 Integración Spring Cloud Stream Kafka

Spring Cloud Stream es un proyecto de Spring Cloud construido sobre la base de Spring Boot y Spring Integration que nos facilita la creación de microservicios bajo los patrones de message-driven y event-driven.

El patrón message-driven, en el entorno de microservicios, introduce el concepto de comunicación asíncrona entre microservicios dirigida por el envío de mensajes; un mensaje es un dato que se envía a un destino específico.

El patrón event-driven, sobre la base del patrón anterior y usando el mismo canal de comunicaciones, se orienta al envío de eventos; un evento es una señal emitida por un componente al alcanzar un determinado estado.

Sobre la base de estos principios se construyen patrones como CQRS y Sagas que, en un entorno de microservicios nos ayudan a mantener, eventualmente, consistente la información, mediante la generación de eventos de dominio (conceptos de DDD).

Spring Cloud Stream es el elegido para implementar dichos patrones en un entorno de microservicios, abstrayéndose totalmente de la capa de transporte y sin necesidad de conocer detalles sobre el protocolo de comunicación.

Dependencia

Para agregar las dependencias a nuestro proyecto maven, insertamos lo siguiente en el bloque de dependencias:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
</dependency>
```

Configuración

En cuanto incluimos la dependencia de Spring Cloud Stream Kafka, que arrastra de forma transitiva la de Spring Cloud Stream Binder, cualquier aplicación Spring Boot con la anotación `@EnableBinding` intentará engancharse al broker de mensajería externo que encuentre en el classpath.

A continuación definiremos los canales de comunicaciones en el fichero de configuración de nuestras aplicaciones (application.yml) y aquí vamos a hacer una distinción entre el productor de los eventos (command) y el consumidor de los mismos (query).

Esta sería la configuración del productor (command) presente en el microservicio **iss-policy-service**.

```
spring:
  stream:
    kafka:
      binder:
        brokers: ${KAFKA_HOST:localhost}:${KAFKA_PORT:9092}
        auto-add-partitions: true # (*) all services create topic and add partitions
      bindings:
        output:
          destination: policy-topic
          content-type: application/json # it's redundant here, 'application/json' is the default
          producer:
            partition-key-expression: headers['partitionKey']
            partition-count: 3 # (*)
```

A continuación la configuración del consumer (query) presente en los microservicios **iss-policy-search-service** e **iss-payment-service**.

```
spring:
  stream:
    default-binder: kafka
    kafka:
      binder:
        brokers: ${KAFKA_HOST:localhost}:${KAFKA_PORT:9092}
        auto-add-partitions: true # (*) all services create topic and add partitions
        min-partition-count: 3 # (*)
      bindings:
        input:
          consumer:
            auto-commit-offset: false
            ack-each-record: true
      bindings:
        input:
          destination: policy-topic
          content-type: application/json # it's redundant here, 'application/json' is the default
          group: eventServiceGroup
```




La diferencia está en la semántica del canal, en este segundo usamos «input» en vez de «output». Por último, antes de comenzar con nuestra emisión de eventos, vamos a incluir las dependencias necesarias para poder ejecutar tests de integración, como sigue:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-test-support</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka-test</artifactId>
  <scope>test</scope>
</dependency>
```

Publicación de un evento (command).

Para publicar un evento lo primero es definir la estructura de datos del evento a emitir, en nuestro caso vamos a publicar un evento de generación de póliza, con la información de la póliza creado:

```
public class PolicyEvent {

    @NotNull
    private String eventId;
    @NotNull
    private Long eventTimestamp;
    @NotNull
    private EventType eventType;
    @NotNull
    private UUID policyId;
    @NotNull
    private PolicyDto policy;
```

En este punto debemos tomar una decisión, ¿compartimos estructuras de los datos de los eventos entre microservicios?. Si añadimos a la estructura de los eventos tipos complejos como en este ejemplo, se convertirá en un problema de dependencias y no queremos hacer que nuestros microservicios dependan de una única estructura de dominio con lo que, pensándolo primero, la estructura de datos de los eventos debería tener su propia estructura de datos



Universidad
Rey Juan Carlos

independiente del dominio de negocio de cada microservicio. No deberíamos incluir en el evento objetos específicos de dominio. por este motivo la presencia de un dto.

A continuación, vamos a añadir a nuestra configuración la anotación

@EnableBinding(Source.class) para realizar un binding con el stream correspondiente (por defecto con Source.class tendremos el canal «output» que ya hemos definido en el **application.yml**).

Lo siguiente es publicar el evento y lo vamos a enviar en nuestro servicio de negocio.

```
@Component
@EnableBinding(Source.class)
public class PolicyStream {

    private static final Logger LOGGER = LoggerFactory.getLogger(PolicyStream.class);
    private final Source source;

    @Autowired
    public PolicyStream(Source source) {
        this.source = source;
    }

    public void policyTerminated(UUID id, PolicyDto policyDto) {
        PolicyEvent event = new PolicyEvent.Builder().withEventId(UUID.randomUUID().toString())
            .withEventTimestamp(System.currentTimeMillis()).withEventType(EventType.TERMINATED)
            .withPolicy(policyDto).withPolicyId(policyDto.getId()).build();
        sendToBus(id, event);
    }

    public void policyRegistered(UUID id, PolicyDto policyDto) {
        PolicyEvent event = new PolicyEvent.Builder().withEventId(UUID.randomUUID().toString())
            .withEventTimestamp(System.currentTimeMillis()).withEventType(EventType.REGISTERED)
            .withPolicy(policyDto).withPolicyId(policyDto.getId()).build();
        sendToBus(id, event);
    }

    private void sendToBus(UUID partitionKey, PolicyEvent event) {
        Message<PolicyEvent> message = MessageBuilder.withPayload(event)
            .setHeader("partitionKey", partitionKey)
            .setHeader(MessageHeaders.CONTENT_TYPE, MimeTypeUtils.APPLICATION_JSON)
            .build();

        source.output().send(message);
        LOGGER.info("\n---\nHeaders: {}\n\nPayload: {}\n---", message.getHeaders(), message.getPayload());
    }
}
```



Universidad
Rey Juan Carlos

Una vez hecho esto vamos a comprobar que realmente se envía escribiendo el siguiente test de integración:

```
@SpringBootTest(classes = Application.class, webEnvironment = SpringBootTest.WebEnvironment.NONE)
@DirtiesContext
@ActiveProfiles("test")
class PolicyStreamIT extends AbstractContainerIntegrationTest {

    private static final String POLICY HOLDER = "François Poirier";
    private static final String PRODUCT_CODE = "CAR";
    private static final String ADMIN_AGENT = "admin";
    public static final String POLICY_NUMBER = "P1212121";
    private static final UUID POLICY_ID = UUID.randomUUID();

    @Autowired
    private Source channels;

    @Autowired
    private MessageCollector collector;

    private PolicyStream sut;

    @BeforeEach
    public void setUp() {
        this.sut = new PolicyStream(channels);
    }

    @Test
    void shouldPropagatePolicyEvents ()
    {
        // given
        final PolicyDto policyDto = getPolicyDto();
        final BlockingQueue<Message<?>> messages = this.collector.forChannel(channels.output());
        // when
        this.sut.policyRegistered(policyDto.getId(), policyDto);
        // then
        assertThat(messages, hasSize(1));
        List<String> sList = messages.stream().map(Message::toString).collect(Collectors.toList());
        String jsonString = sList.get(0);
        assertTrue(jsonString.contains(POLICY HOLDER));
        assertTrue(jsonString.contains(PRODUCT_CODE));
        assertTrue(jsonString.contains(ADMIN_AGENT));
        assertTrue(jsonString.contains(POLICY_NUMBER));
    }

    private PolicyDto getPolicyDto() {
        return new PolicyDto.Builder().withId(POLICY_ID)
            .withAgentLogin(ADMIN_AGENT)
            .withFrom(LocalDate.of(2018, 1, 1))
            .withNumber(POLICY_NUMBER)
            .withProductCode(PRODUCT_CODE)
            .withPolicyHolder(POLICY HOLDER)
            .withTo(LocalDate.of(2018, 12, 31))
            .withTotalPremium( new BigDecimal(1000))
            .build();
    }
}
```

Declaramos por inyección de dependencias un recolector de mensajes y el canal fuente definido en el binder; con este último configuramos el recolector y comprobamos que una vez invocado el servicio de negocio podemos consumir un evento almacenado en el recolector que tiene como payload la estructura de datos del evento enviado.

Consumo de un evento (query).

Consumir el evento desde un test de integración está más que bien, pero el objetivo es consumirlo desde otro microservicio, para lo cual debemos realizar la misma configuración en cuanto a dependencias de maven en nuestro segundo microservicio y crear la estructura de datos a la que se mapeara la recepción del evento.

Lo siguiente es configurar el binder y un método de listener que escuchará los eventos:

```
@Component
@Transactional
@EnableBinding(Sink.class)
public class PolicyEventConsumerAdapter implements PolicyEventConsumerPort {

    private static final Logger LOGGER = LoggerFactory.getLogger(PolicyEventConsumerAdapter.class);

    private final PolicyAccountLoadPort policyAccountLoadPort;
    private final PolicyAccountUpdatePort policyAccountUpdatePort;

    @Autowired
    public PolicyEventConsumerAdapter(PolicyAccountLoadPort policyAccountLoadPort, PolicyAccountUpdatePort policyAccountUpdatePort) {
        this.policyAccountLoadPort = policyAccountLoadPort;
        this.policyAccountUpdatePort = policyAccountUpdatePort;
    }

    @StreamListener(Sink.INPUT)
    @Override
    public void process(Message<PolicyEvent> event, @Header(KafkaHeaders.ACKNOWLEDGMENT) Acknowledgment acknowledgment) {
        LOGGER.info("event received {}", event);
        PolicyEvent payload = event.getPayload();
        if (EventType.REGISTERED.equals(payload.getEventType())) {
            PolicyAccount account = policyAccountLoadPort.findByPolicyNumber(payload.getPolicy().getNumber());
            if (account == null) {
                createAccount(payload.getPolicy());
            }
        }
        LOGGER.info("Acknowledgment provided");
        acknowledgment.acknowledge();
    }

    private void createAccount(PolicyDto policy) {
        PolicyAccount newAccount = new PolicyAccount.Builder().withPolicyNumber(policy.getNumber()).withPolicyAccountNumber(UUID.randomUUID().toString()).build();
        newAccount.expectedPayment(policy.getTotalPremium(), policy.getFrom());
        policyAccountUpdatePort.save(newAccount);
    }
}
```


Vamos a probar el envío y la recepción con un test de integración dentro del contexto del propio microservicio:

```
@SpringBootTest(classes = Application.class, webEnvironment = SpringBootTest.WebEnvironment.NONE)
@DirtiesContext
@ActiveProfiles("test")
class PolicyEventConsumerAdapterIT extends AbstractContainerIntegrationTest {

    private static final String POLICY HOLDER = "François Poirier";
    private static final String PRODUCT_CODE = "CAR";
    private static final String ADMIN_AGENT = "admin";
    public static final String POLICY_NUMBER = "P1212121";
    private static final UUID POLICY_ID = UUID.randomUUID();

    @Autowired
    private Sink channels;

    @Autowired
    private PolicyAccountJpaRepository policyAccountJpaRepository;

    @Test
    void shouldConsumePolicyEventAndSave() {
        // given
        var optPolicyAccount = policyAccountJpaRepository.findByPolicyNumber(POLICY_NUMBER);
        assertTrue(!optPolicyAccount.isPresent());

        // when
        PolicyDto policyDto = getPolicyDto();
        PolicyEvent event = new PolicyEvent.Builder().withEventId(UUID.randomUUID().toString())
            .withEventTimestamp(System.currentTimeMillis()).withEventType(EventType.REGISTERED)
            .withPolicy(policyDto).withPolicyId(policyDto.getId()).build();
        Message<PolicyEvent> message = MessageBuilder.withPayload(event)
            .setHeader("partitionKey", policyDto.getId())
            .setHeader(MessageHeaders.CONTENT_TYPE, MimeTypeUtils.APPLICATION_JSON)
            .setHeader(KafkaHeaders.ACKNOWLEDGMENT, Mockito.mock(Acknowledgment.class))
            .build();
        this.channels.input().send(message);
        // then
        optPolicyAccount = policyAccountJpaRepository.findByPolicyNumber(POLICY_NUMBER);
        assertTrue(optPolicyAccount.isPresent());
    }

    private PolicyDto getPolicyDto() {
        return new PolicyDto.Builder().withId(POLICY_ID)
            .withAgentLogin(ADMIN_AGENT)
            .withFrom(LocalDate.of(2018, 1, 1))
            .withNumber(POLICY_NUMBER)
            .withProductCode(PRODUCT_CODE)
            .withPolicyHolder(POLICY HOLDER)
            .withTo(LocalDate.of(2018, 12, 31))
            .withTotalPremium( new BigDecimal(1000))
            .build();
    }
}
```

Simulamos el servicio que emite un evento generándolo nosotros manualmente y como el listener persiste en nuestro caso una cuenta asociada a la póliza, comprobamos su existencia en base de datos con la ayuda de un repository.

3.3.8 Api Gateway

Para la implementación del patrón Api Gateway, utilizaremos el proyecto Spring Cloud Gateway. Este proyecto se encarga de proporcionar un punto de entrada a nuestro ecosistema de microservicios, proporcionando capacidades de enrutamiento dinámico, seguridad y monitorización de las llamadas que se realicen.

Spring Cloud Gateway se basa en Spring Framework 5, Project Reactor ([permitiendo la integración con Spring WebFlux](#)) y Spring Boot 2 utilizando API sin bloqueo. Esto hace que sea compatible con conexiones de larga duración (Websockets) y tiene una mejor integración con Spring. Además con el lanzamiento de Spring Boot 2 y Spring Cloud 2, Spring Cloud Gateway tiene un rendimiento superior en comparación a otras soluciones de Spring Cloud (Zuul).

Características y funcionalidades proporcionadas por Spring Cloud Gateway

Todas las características y funciones que vamos a nombrar a continuación se pueden configurar mediante una clase Java o mediante el archivo application.yaml.

Routing Handler	Spring Cloud Gateway envía las solicitudes al Gateway Handler Mapping que determina qué se debe hacer con las solicitudes que coinciden con una ruta específica
Dynamic Routing	Al igual que Zuul , Spring Cloud Gateway proporciona medios para enrutar solicitudes a diferentes servicios
Routing Factories	Spring Cloud Gateway combina las rutas usando la infraestructura Spring WebFlux HandlerMapping. Incluye muchos built-in Route Predicate Factories. Todos estos Predicates coinciden con diferentes atributos de la solicitud HTTP pudiéndose combinarlos.
WebFilter Factories	Los filtros de enrutamiento hacen posible la modificación de la solicitud HTTP entrante o de la respuesta HTTP saliente
Spring Cloud DiscoveryClient Support	Spring Cloud Gateway se puede integrar fácilmente con las bibliotecas Service Discovery y Registry, como Eureka Server y Consul
Monitoring	Spring Cloud Gateway hace uso de la API Actuator, una conocida biblioteca Spring-Boot que proporciona varios servicios listos para monitorear la aplicación



Dependencia

Para agregar la dependencia a nuestro proyecto maven, insertamos lo siguiente en el bloque de dependencias:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

Configuración

Se puede realizar utilizando una clase Java de configuración haciendo uso de RouteLocator o usando la configuración de propiedades en el archivo application.yaml.

```
spring:
  jackson.serialization.indent_output: true
  application:
    name: iss-api-gateway
  cloud:
    loadbalancer:
      ribbon:
        enabled: false
    gateway:
      discovery:
        locator:
          enabled: true
      routes:
        - id: iss-payment-service
          uri: lb://iss-payment-service
          predicates:
            - Path=/iss-payment-service/**
          filters:
            - RewritePath=/iss-payment-service/(?<path>.*), /${path}
        - id: iss-policy-search-service
          uri: lb://iss-policy-search-service
          predicates:
            - Path=/iss-policy-search-service/**
          filters:
            - RewritePath=/iss-policy-search-service/(?<path>.*), /${path}
        - id: iss-policy-service
          uri: lb://iss-policy-service
          predicates:
            - Path=/iss-policy-service/**
          filters:
            - RewritePath=/iss-policy-service/(?<path>.*), /${path}
        - id: iss-product-service
          uri: lb://iss-product-service
          predicates:
            - Path=/iss-product-service/**
          filters:
            - RewritePath=/iss-product-service/(?<path>.*), /${path}
```

En ambos procesos, lo que estamos haciendo es configurar nuestro enrutamiento para que cuando nos llegue una petición por «/foo/**» (el path) lo envíe al microservicio de id foo al que queremos enviar la petición.

3.4 Componentes elevados a nivel de arquitectura

A continuación presentamos algunos componentes que podrían perfectamente elevarse a nivel de arquitectura, externalizando todos estos componentes en unos starters o librerías.

3.4.1 Clases de soporte para CQRS

Para implementar los casos de usos de creación de ofertas, creación y finalización de pólizas, el proyecto proporciona un conjunto de clases abstractas de soporte al paradigma de arquitectura [CQRS y Event Sourcing](#).

3.4.2 Health Check

Cuando ya tenemos una aplicación que está corriendo en producción, es muy importante mantenernos al tanto de la salud de dicha aplicación y asegurarnos de que siempre esté corriendo. En especial cuando son partes críticas de nuestro negocio que de no estar funcionando afectará directamente el éxito del negocio.

Tradicionalmente, antes de Spring Actuator, necesitábamos escribir código a mano para revisar la salud de nuestras aplicaciones basándonos en algunos criterios, pero con la llegada de Spring Actuator ya no es necesario escribir ese código nosotros mismos ya que este provee algunos endpoints listos para usarse y que son muy útiles para revisar la salud de la aplicación.

Dependencia

Para agregar la dependencia a nuestro proyecto maven, insertamos lo siguiente en el bloque de dependencias:

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>
```

Habilitar Endpoints

Para habilitar o deshabilitar un endpoint, simplemente debemos agregarlo al application.yaml del proyecto con la siguiente llave:



Universidad
Rey Juan Carlos

```
management:
  endpoint:
    <id>:
      enabled: <true|false>
```

También podemos especificar una lista separada por comas de los endpoints que necesitamos exponer a través de http:

```
management:
  endpoints:
    web:
      exposure:
        include: <lista de endpoints>
```

O bien podemos exponerlos todos (no recomendable para producción):

```
management:
  endpoints:
    web:
      exposure:
        include: *
```

O exponer algunos y excluir algunos otros:

```
management:
  endpoints:
    web:
      exposure:
        include: *
        exclude: env,beans
```

Deshabilitar Endpoints

Para deshabilitar uno o todos los endpoints podemos usar cualquiera de las siguientes propiedades:

```
management:
  server:
    port: -1
```

O bien incluirlas todas:

```
management:
  endpoint:
    health:
      show-details: always
  endpoints:
    web:
      exposure:
        include: '*'
```

/health

Quizás uno de los endpoints más importantes es el de health que nos indica si nuestra aplicación está corriendo con normalidad, por defecto veremos una respuesta como esta:

```
{
  "status": "UP"
}
```

Si queremos podemos crear un indicador de salud acorde a nuestras necesidades con una clase como proporcionada en cada microservicio ([CustomHealthIndicator](#)). En el caso del proyecto, veremos una respuesta como esta:

```
{
  "status" : "UP",
  "components" : {
    "binders" : {
      "status" : "UP",
      "components" : {
        "kafka" : {
          "status" : "UP"
        }
      }
    }
  },
  "custom" : {
    "status" : "UP",
    "details" : {
      "iss-payment-service" : "iss-payment-service Available"
    }
  }
},
```



Universidad
Rey Juan Carlos

```
"db" : {
  "status" : "UP",
  "details" : {
    "database" : "PostgreSQL",
    "validationQuery" : "isValid()"
  }
},
"discoveryComposite" : {
  "description" : "Discovery Client not initialized",
  "status" : "UNKNOWN",
  "components" : {
    "discoveryClient" : {
      "description" : "Discovery Client not initialized",
      "status" : "UNKNOWN"
    }
  }
},
"diskSpace" : {
  "status" : "UP",
  "details" : {
    "total" : 414989475840,
    "free" : 237915484160,
    "threshold" : 10485760,
    "exists" : true
  }
},
"kubernetes" : {
  "status" : "UP",
  "details" : {
    "inside" : false
  }
},
"ping" : {
  "status" : "UP"
},
"refreshScope" : {
  "status" : "UP"
}
}
```

3.4.3 Intercepción de mensajes/errores: Feign Decoder

Lo normal es que los errores dentro de nuestra nube de servicios tengan una normalización en cuanto a formato y tipología, aunque si consumimos servicios externos quizás nos tengamos que adaptar a otros formatos de mensaje; Sobrescribiendo el comportamiento por defecto del framework que usemos, para por ejemplo, añadir un identificador único del error o permitir devolver una colección de errores que devuelvan información de validación de un recurso anotado con el soporte de la JSR-303.

Si damos por hecho que nuestros servicios pueden devolver errores con el siguiente formato teniendo en cuenta que la tipología del error la delegamos en el estado http:

```
{  
  "status": 404,  
  "message": "errorMessage"  
}
```

Si estamos pensando en disponer de una capa de clientes que consuman servicios que pueden devolver ese tipo de formato de salida, deberíamos pensar también en preparar un componente que parsee esa información de manera transversal. Para cubrir este requisito basta con implementar un `ErrorDecoder` como el que facilitamos en este proyecto ([DefaultErrorDecoder](#)).

Se podría decir que este decoder tiene la lógica inversa al `ExceptionHandler` (punto [3.4.4 Manejador de excepciones](#)) que ha generado la respuesta de error. Para que funcione solo tenemos que configurarlo en una clase anotada con un `@Configuration` ([DefaultDecoderConfiguration](#)).

3.4.4 Manejador de excepciones

A lo largo del proyecto se han creado excepciones nuevas, para casos concretos. Para poder gestionarlas, se crea un manejador de excepciones, que implementa un aspecto (AOP) de tipo `ThrowAdvice` ([RestExceptionHandler](#)) que incorporará la anotación `@RestControllerAdvice` en complemento de la anotación `@ExceptionHandler`.

3.4.5 Clases de soporte para testing

Las pruebas de integración y e2e se han llevado a cabo con el soporte de `TestContainers`, probando los diferentes casos de uso de los distintos microservicios. Para la implementación de dichas pruebas se proporcionan 2 clases abstractas que permiten en un caso el arranque de la imagen docker de la base de datos postgresql ([AbstractContainerBaseTest](#)) y por otro las imágenes dockers de la base de datos postgresql y del broker kafka ([AbstractContainerIntegrationTest](#)).

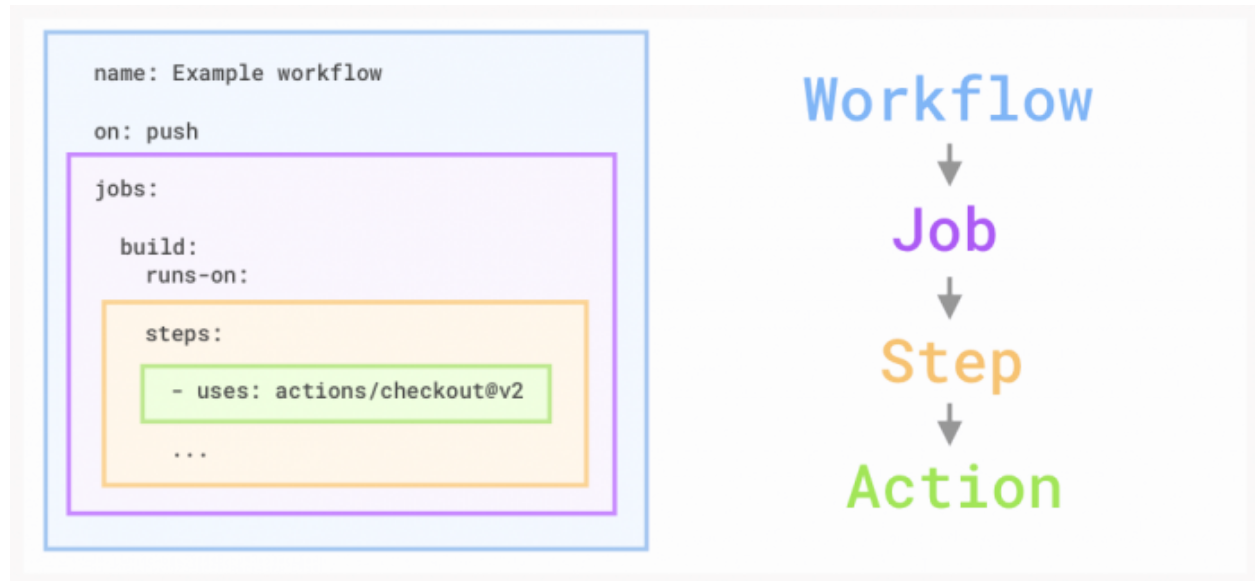
En el caso de la base de datos mongoDB al igual que en lo interior se proporciona una clase abstracta ([AbstractContainerIntegrationTest](#)).

4. Integración continua

Como bien sabemos, la **Integración Continua** es una práctica que nos requiere añadir frecuentemente nuevo código a un repositorio compartido para detectar errores a la mayor brevedad posible. Para ello Github propuso en 2018 no solo alojar nuestro código en sus repositorios como hemos hecho siempre, sino que además la posibilidad de automatizar los distintos pasos de construcción de proyectos.

Para ello Github Actions ha creado el concepto de **workflow** el cual es el encargado principal de todo nuestro proceso o Pipeline. Se puede configurar de manera que Github reaccione a ciertos eventos (por ejemplo cuando se hace un nuevo push a una rama), automáticamente de forma periódica o por eventos externos. Especificando en dicho workflow que se analicen los componentes del proyecto, una vez terminados se mostrarán los resultados de los mismos y se podrá comprobar si el cambio en dicha rama ha producido algún error o ha ido todo bien. Este mismo es ejecutado en un **runner** o instancia en un servidor y Github te da la posibilidad de utilizar un runner hosteado por Github o añadir un host propio.

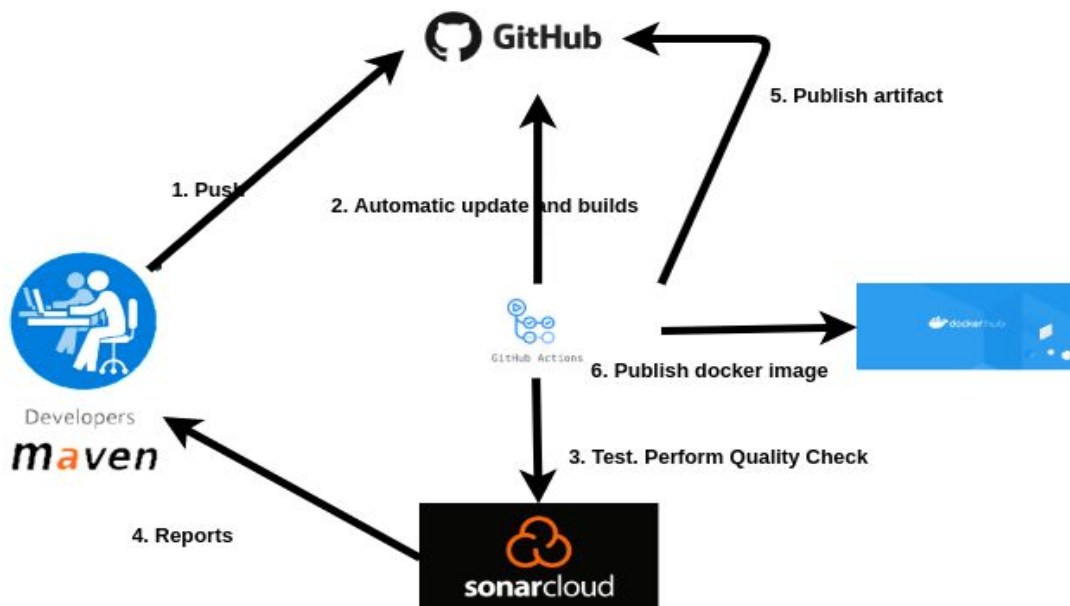
4.1 Definición de un workflow



- **Workflow:** como ya hemos comentado anteriormente, es un procedimiento automatizado el cual se añade a un repositorio. Con él se puede hacer el build, test, package, release o deploy de un proyecto dentro de Github.
- **Job:** es un conjunto de **steps** que se ejecutan en runner de nuestro proceso.
- **Step:** es una tarea individual que puede ejecutar comandos dentro de un **job**. Un job está formado por uno o más steps y éstos están ejecutados sobre el mismo runner a la hora de ejecutarse el workflow.
- **Action:** Son los comandos de ejecución del proceso, ejecutados en un **step** para crear un **job**. Son el bloque de construcción más pequeño que hay. Puedes crear tus propios actions o utilizar algunos de ellos que ya están creados por la comunidad de Github. Obligatoriamente para utilizar un action en un workflow, éste debe ir incluido en un step.

4.2 Presentación de la solución de integración continua

Para nuestra solución de integración continua vamos a utilizar de forma conjunta Maven, GitHub, GitHub Actions, SonarCloud y Docker Hub.



4.2.1 Configuración de secretos

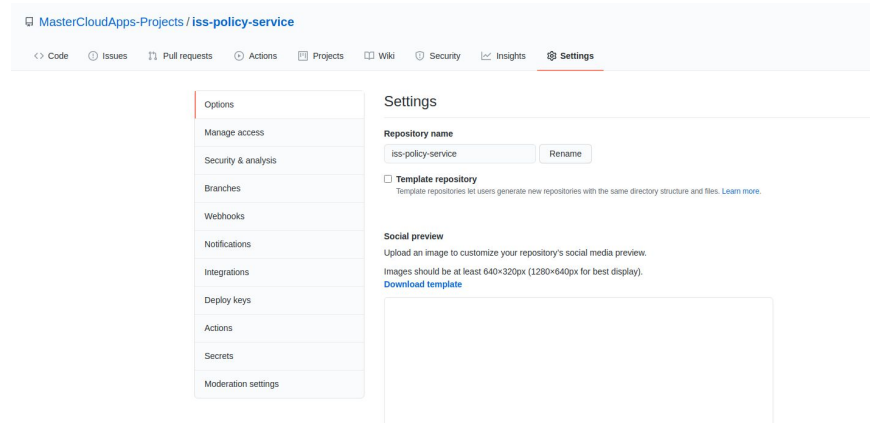
Los secretos de Github Action nos permiten almacenar información sensible en nuestro repositorio para luego poder usarlo en nuestros procesos de workflow. Son variables de entorno encriptadas por Github, haciendo uso de libsodium sealed box para asegurar que la información de los secretos se aseguran antes de siquiera ser parte de los repositorios de Github y por supuesto mientras se utilizan en cualquiera de los workflows.

Para añadir un secreto a nuestro repositorio:

1. Navegamos al menú principal del repositorio.
2. Debajo del nombre del repositorio, seleccionamos Settings.
3. En el menú de la izquierda seleccionamos Secrets y a continuación nos sale el botón New secret para añadir un nuevo secreto.

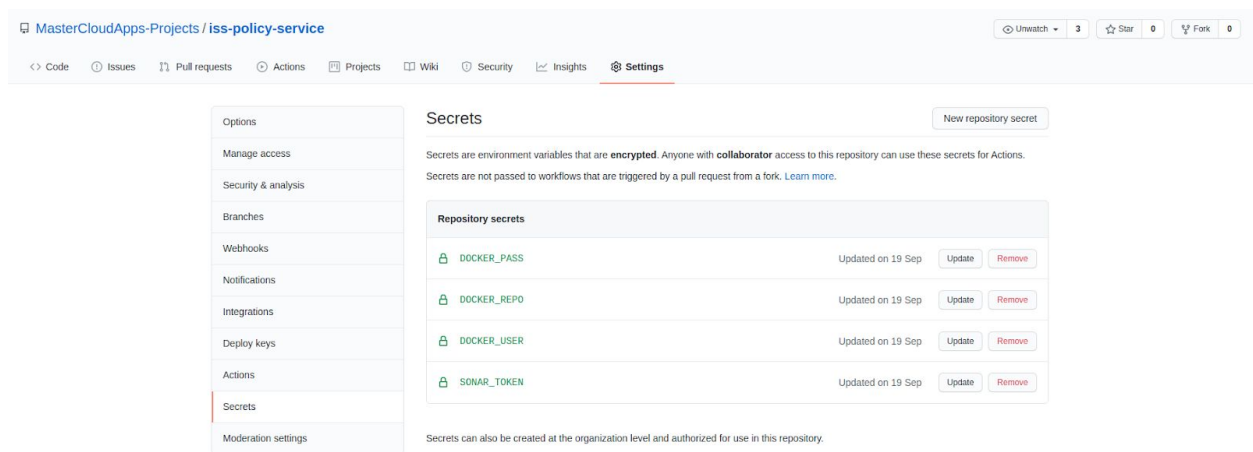


Universidad
Rey Juan Carlos



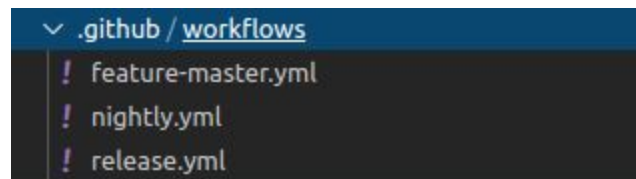
4. Añadimos nombre y valor y pulsamos sobre Add secret. Ahora ya nos aparece creado y listo para poder añadirlo a nuestros pipelines. **En nuestro caso necesitamos crear 4 secret:**

- `DOCKER_PASS` password de la cuenta dockerhub.
- `DOCKER_USER` user de la cuenta dockerhub.
- `DOCKER_REPO` el nombre del proyecto en la cuenta dockerhub que sigue el siguiente pattern (YOUR_USERNAME/YOUR_PROJECT_NAME), ejemplo **fpoirier2020/iss-policy-search-service**
- `SONAR_TOKEN` el token solicitado a SonarCloud para poder controlar la calidad del proyecto.



4.2.2 Workflows

Se presenta a continuación los 3 flujos de trabajos que se han implementado para cada proyecto. Estos flujos se encuentran presente en la carpeta **.github/workflows** de cada microservicio.



- El flujo de trabajo **feature-master** se ejecuta cada vez que se valida una pull request de una rama **feature** hacia **master** o cuando se realiza un push desde **master**.
- El flujo **nightly** se ejecuta por la noche a la 1 de la mañana.
- El flujo **release** se ejecuta cada vez que se realiza un push contra la rama release en curso.

A continuación nos vamos a centrar sobre los flujos **nightly** y **release**, ya que los Jobs definidos en el workflow **feature-master** están presente en el flujo **nightly**.

4.2.2.1 Flujo nightly

Requisito previo

- [Tener creado los secretos](#)
- Configurar la cuenta de dockerhub en el fichero settings.xml de maven.

```
<server>
  <id>registry.hub.docker.com</id>
  <username>my_username</username>
  <password>my_password</password>
</server>
```

Definición del flujo nightly

El flujo nightly está compuesta de 4 jobs

- **Test — Units & Integrations**

Básicamente, este job pide a Maven que ejecute el comando **mvn verify** pero con un perfil adicional de Maven que tiene unas dependencias asociadas. Y por lo tanto, para que funcione correctamente, necesitamos declarar este perfil en el POM del proyecto. Os invito a ver la declaración del perfil **integration-test** en el propio [fichero](#).

1. Checkout del proyecto.
2. Cacheado del repositorio de maven.
3. Empaquetado de artefacto
4. Ejecución de test unitarios, de integración y e2e

```
test:
  name: Test - Units & Integrations
  runs-on: ubuntu-18.04

  steps:
    - uses: actions/checkout@v1
    - name: Set up JDK 11
      uses: actions/setup-java@v1
      with:
        java-version: 11.0.4
    - name: Cache Maven packages
      uses: actions/cache@v2
      with:
        path: ~/.m2
        key: ${ runner.os }-m2-${ hashFiles('**/pom.xml') }
        restore-keys: ${ runner.os }-m2
    - name: Maven Package
      run: mvn -B clean package -DskipTests
    - name: Maven Verify
      run: mvn -B clean verify -Pintegration-test
```

- **Test — SonarCloud Scan**

Al igual que en el job de prueba, primero definimos en qué máquina y con qué versión de Java se ejecutará. La única diferencia está en el comando `mvn verify` donde debemos proporcionar dos parámetros: `-Psonar` y `-Dsonar.login = $ {{secrets.SONAR_TOKEN}}`.

Lo primero indica que queremos ejecutar el comando Maven con el perfil de sonar, cuya definición se encuentra en el POM del proyecto. Con este perfil habilitamos el análisis de SonarCloud y definimos dónde se encuentra el código fuente, qué clases y paquetes deben excluirse del análisis y la URL de SonarCloud.

Para que funcione, se debe tener previamente creado una cuenta en [SonarCloud](https://sonarcloud.io).

Os invito a ver la declaración del perfil **sonar** en el propio [fichero](#).

1. Checkout del proyecto.
2. Cacheado del repositorio de maven.
3. Empaquetado de artefacto
4. Escaneo **Sonar** con ejecución previa de test unitarios, de integración y e2e

```
sonar:
  name: Test - SonarCloud Scan
  runs-on: ubuntu-18.04

  steps:
    - uses: actions/checkout@v1
    - name: Set up JDK 11
      uses: actions/setup-java@v1
      with:
        java-version: 11.0.4
    - name: Cache Maven packages
      uses: actions/cache@v2
      with:
        path: ~/.m2
        key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
        restore-keys: ${{ runner.os }}-m2
    - name: SonarCloud Scan
      run: mvn -B clean verify -Psonar,integration-test -Dsonar.login=${{ secrets.SONAR_TOKEN }}
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

- **Publish — GitHub Packages**

Este job permite publicar el artefacto en el repositorio de artefactos “**Github packages**”.

Previamente, necesitamos declarar la etiqueta **<distributionManagement>** al POM del proyecto para definir la ubicación del repositorio de artefactos donde queremos publicarlo. Os invito a ver la declaración en el propio [fichero](#). Por último, este job se puede iniciar sólo cuando los 2 primeros jobs acaban con éxito.

1. Checkout del proyecto.



Universidad Rey Juan Carlos

2. Cacheado del repositorio de maven.
3. Empaquetado y publicación de artefacto en **Github Packages**

```
artifact:
  name: Publish - GitHub Packages
  runs-on: ubuntu-18.04
  needs: [test, sonar]

steps:
  - uses: actions/checkout@v2
  - name: Set up JDK 11
    uses: actions/setup-java@v1
    with:
      java-version: 11.0.4
  - name: Cache Maven packages
    uses: actions/cache@v2
    with:
      path: ~/.m2
      key: ${ runner.os }-m2-${ hashFiles('**/pom.xml') }
      restore-keys: ${ runner.os }-m2
  - name: Publish artifact on GitHub Packages
    run: mvn -B clean deploy -DskipTests
    env:
      GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }
```

• Publish — Docker Hub

Y finalmente el último job: publicar la imagen de Docker de la aplicación en Docker Hub. Antes de comenzar a configurar este job, tenemos que tener una cuenta en Docker Hub. De manera similar al job anterior, este también podría iniciarse sólo cuando los 2 primeros jobs acaban con éxito.

1. Checkout del proyecto.
2. Login en la cuenta de dockerhub
3. Creación de la imagen docker
4. Publicación de la imagen docker



Universidad
Rey Juan Carlos

```
docker:
  name: Publish - Docker Hub
  runs-on: ubuntu-18.04
  needs: [test, sonar]
  env:
    REPO: ${ secrets.DOCKER_REPO }

  steps:
    - uses: actions/checkout@v2
    - name: Login to Docker Hub
      run: docker login -u ${ secrets.DOCKER_USER } -p ${ secrets.DOCKER_PASS }
    - name: Build Docker image
      run: docker build -f ./docker/Dockerfile -t $REPO:latest -t $REPO:${GITHUB_SHA::8} .
    - name: Publish Docker image
      run: docker push $REPO
```



4.2.2.2 Flujo release

Definición del flujo release

El flujo release está compuesta de un único job nombrado **build** compuesto de 5 steps

- **build**
 1. Checkout del proyecto.
 2. Cacheado del repositorio de maven.
 3. Descarga e instalación del JDK.
 4. Configuración del user y email de Github Actions.
 5. Publicación del artefacto en github.

```
name: Release

on:
  push:
    branches:
      - release

jobs:
  build:
    runs-on: ubuntu-18.04
    steps:
      - name: Checkout project
        uses: actions/checkout@v2
      - name: Cache local Maven repository
        uses: actions/cache@v2
        with:
          path: ~/.m2/repository
          key: ${ runner.os }-maven-${ hashFiles('**/pom.xml') }}
          restore-keys: ${ runner.os }-maven-
      - name: Setup Java JDK
        uses: actions/setup-java@v1.4.2
        with:
          java-version: 11.0.4
          server-id: github
      - name: Configure Git user
        run: |
          git config user.email "actions@github.com"
          git config user.name "GitHub Actions"
      - name: Publish JAR
        run: ./mvnw -B release:prepare release:perform
        env:
          GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }}
```




Universidad
Rey Juan Carlos

Para poder gestionar las releases de un proyecto y publicarlas en github, se necesita configurar distintos elementos que vamos a ver a continuación.

La gestión de release se realiza con el plugin maven release. Este plugin ejecuta dos fases antes de publicar el artefacto:

release:prepare

En esta fase se realizan las siguientes operaciones:

- Chequea que no haya cambios en los fuentes sin subir a Git
- Chequea que todas las dependencias del proyecto son releases (no hay SNAPSHOTS)
- Cambia la versión de todos los artefactos involucrados, quitando el SNAPSHOT. Ej: La versión 1.0-SNAPSHOT se convierte en la versión 1.0.
- Inserta en la sección SCM del POM la información del TAG que se va a realizar
- Ejecuta todos los test sobre el POM modificado, para asegurar que todo sigue funcionando correctamente
- Realiza el commit de los POM modificados
- Marca el código recién subido con una etiqueta que representa la versión que se ha generado
- Vuelve a modificar la versión en los POM para que sea la siguiente versión a desarrollar, seguida del tag SNAPSHOT. Ej: De la versión 1.0 podemos pasar a la versión 1.1-SNAPSHOT o a cualquier otra que queramos.
- Realiza el commit de los POM con la siguiente versión, de manera que los desarrolladores puedan empezar a trabajar en la nueva versión.

release:perform

En esta fase se publica en el repositorio de artefactos una versión previamente subida al control de versiones. Se realizan básicamente dos tareas:

- Se descarga del sistema de control de versiones el último TAG de versión
- Genera el artefacto y lo sube al repositorio de artefactos (ejecuta mvn deploy)

Configuración del proyecto y de maven

Requisito previo

- [Tener creado los secretos](#)



Para poder utilizar este plugin conjuntamente con el maven deploy plugin que permite publicar un artefacto en el SCM github, hay que:

- Declarar los plugins deploy plugin y release plugin en el pom del propio proyecto.

```
<plugin>
  <artifactId>maven-deploy-plugin</artifactId>
  <version>3.0.0-M1</version>
</plugin>
<plugin>
  <artifactId>maven-release-plugin</artifactId>
  <version>3.0.0-M1</version>
  <configuration>
    <tagNameFormat>v@{project.version}</tagNameFormat>
  </configuration>
</plugin>
```

- Configuración de la ruta del SCM en el fichero pom del proyecto.

```
<scm>
  <developerConnection>scm:git:https://github.com/${git.repository}.git</developerConnection>
</scm>
```

Para poder actuar contra el SCM github hay que:

- Declarar en el fichero pom del proyecto la propiedad con clave project.scm.id y el ID del servidor github, que corresponde a las credenciales de la cuenta github.

```
<project.scm.id>github</project.scm.id>
```

- Declarar en el fichero settings.xml de la herramienta maven las credenciales de la cuenta github.

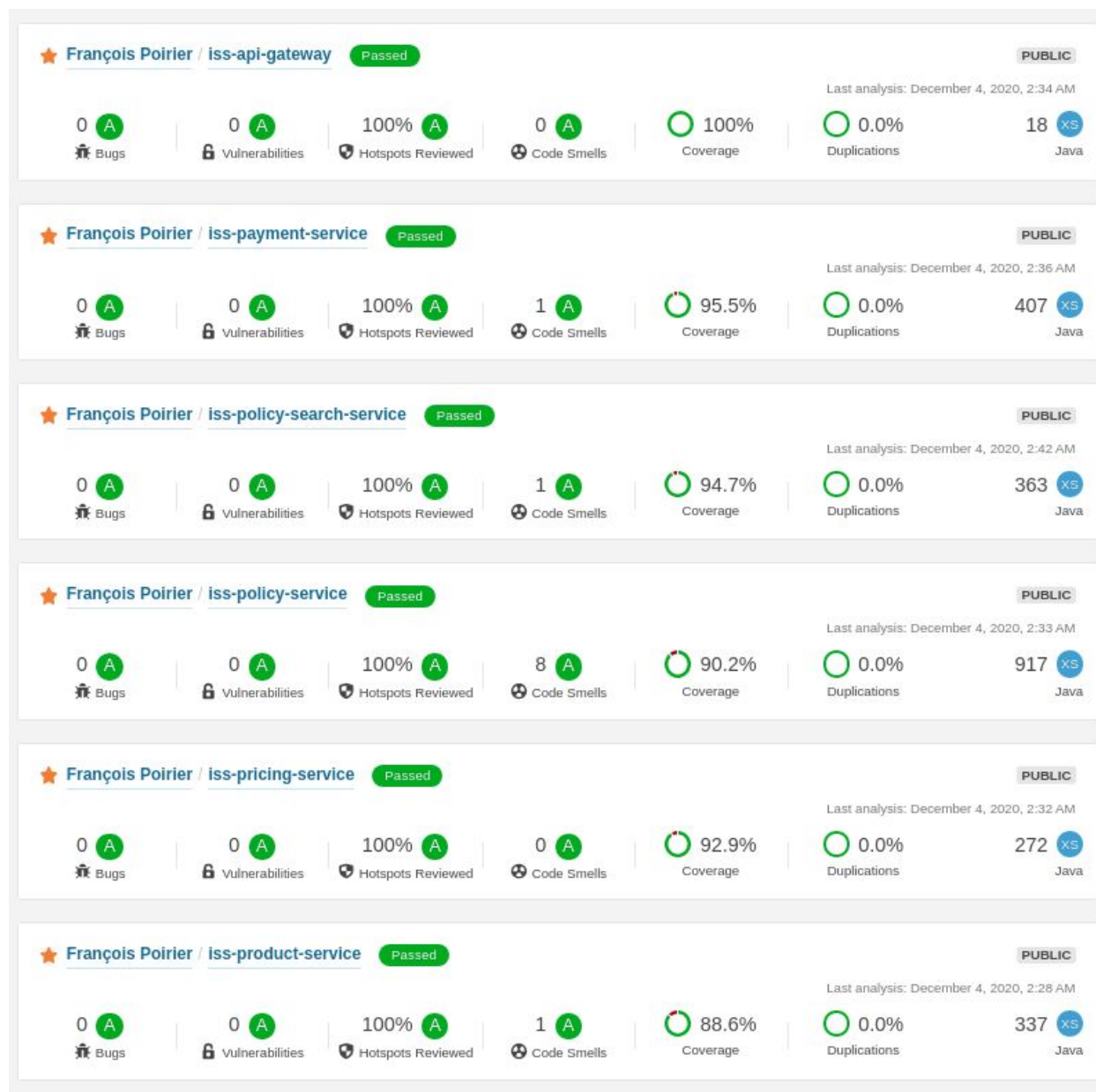
```
<server>
  <id>github</id>
  <username>my_username</username>
  <password>my_password</password>
</server>
```

- Declarar en el fichero pom del proyecto la ruta del proyecto en el repositorio SCM github.

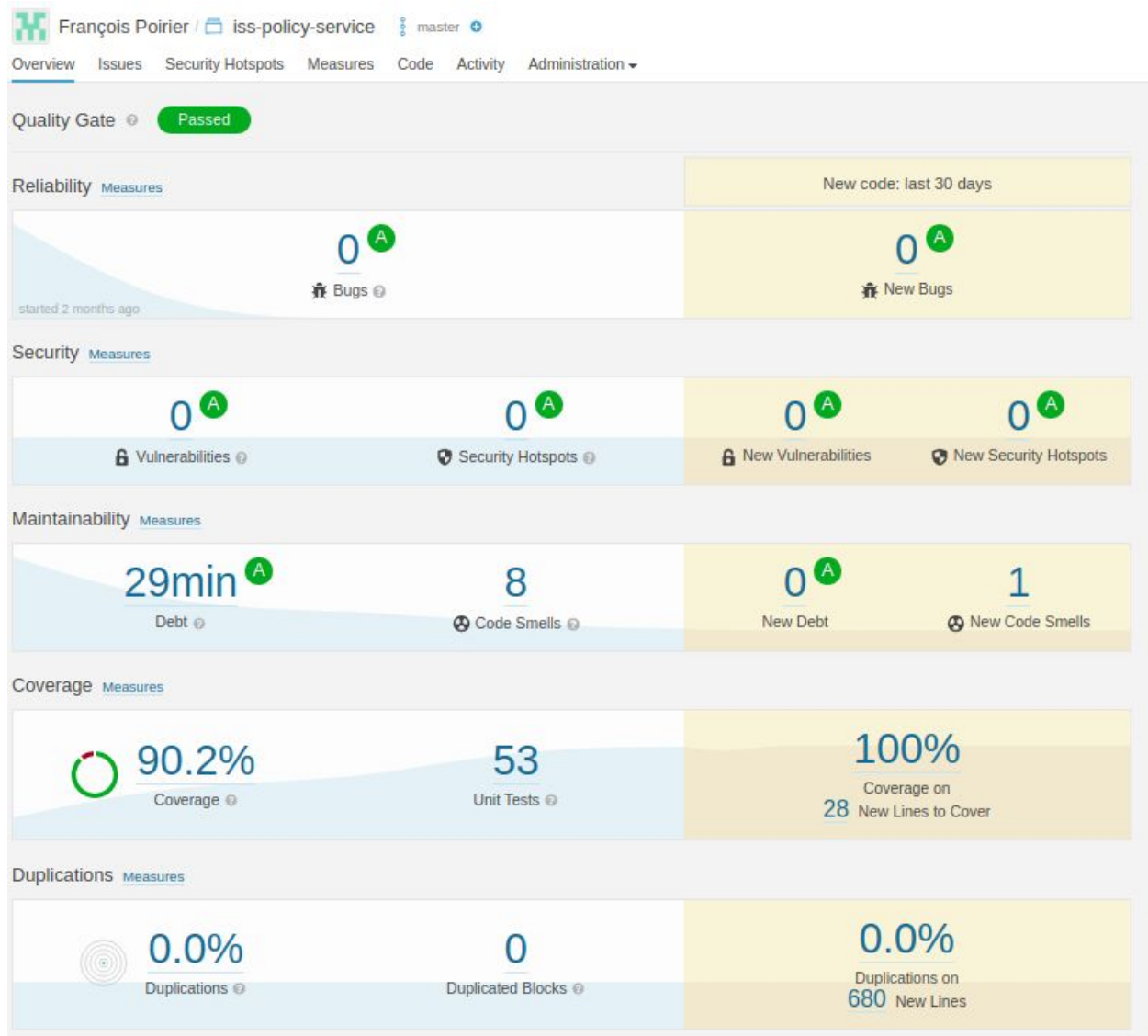
```
<distributionManagement>
  <repository>
    <id>github</id>
    <name>TFM MasterCloudApps</name>
    <url>https://maven.pkg.github.com/MasterCloudApps-Projects/iss-payment-service</url>
  </repository>
</distributionManagement>
```

4.2.3 Presentación de los informes

A continuación, el dashboard de Sonarcloud con un resumen del estado actual de los 6 microservicios.



Un Quality Gate es un conjunto de condiciones que ayuda a saber de inmediato si los proyectos están listos para salir a producción. Sonar proporciona una quality gate por defecto a todos los proyectos, que consta de 5 condiciones. A continuación un Quality Gate de un microservicio.



La cobertura actual de los distintos microservicios supera el 80% pedido por SonarCloud.

5. Despliegue en Kubernetes (minikube)

5.1 Instalar y configurar Minikube en Ubuntu

Primero de todo actualizamos nuestro sistema e instalamos algunos paquetes necesarios:

- **sudo apt-get update**
- **sudo apt-get install** apt-transport-https

Aquí podemos habilitar KVM o el hipervisor de VirtualBox. nos decantamos por el segundo.

- **sudo apt install** virtualbox virtualbox-ext-pack

Descarga de minikube

- **wget**
<https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64>
- **chmod +x** minikube-linux-amd64
- **sudo mv** minikube-linux-amd64 /usr/local/bin/minikube

Comprobamos la versión instalada:

- **minikube version**

```
minikube version: v1.15.1  
commit: 23f40a012abb52eff365ff99a709501a61ac5876
```

Instalar Kubectl

Necesitamos kubectl, que es una herramienta de línea de comandos utilizada para implementar y administrar aplicaciones en Kubernetes:

- **curl -LO** <https://storage.googleapis.com/kubernetes-release/release/>**`curl -s**
<https://storage.googleapis.com/kubernetes>

Marcamos el binario descargado como ejecutable:

- **chmod +x ./kubectl**

Lo movemos a la ruta correspondiente:

- **sudo mv ./kubectl /usr/local/bin/kubectl**

Comprobamos la versión instalada:

- **kubectl version -o json**

```
{
  "clientVersion": {
    "major": "1",
    "minor": "18",
    "gitVersion": "v1.18.3",
    "gitCommit": "2e7996e3e2712684bc73f0dec0200d64eec7fe40",
    "gitTreeState": "clean",
    "buildDate": "2020-05-22T10:45:08Z",
    "goVersion": "go1.14.3",
    "compiler": "gc",
    "platform": "linux/amd64"
  }
}
```

5.2 Despliegue del proyecto en minikube

Todos los descriptores de kubernetes se encuentran en el directorio k8s del proyecto iss-api-gateway.

Clonar el proyecto iss-api-gateway

- **git clone <https://github.com/MasterCloudApps-Projects/iss-api-gateway>**

Posicionarse en la carpeta raíz del proyecto

- **cd iss-api-gateway**

Arrancamos Minikube

- **minikube start --memory 8192**

Crear un **ClusterRoleBinding** para el namespace default

- **kubectl create clusterrolebinding admin --clusterrole=cluster-admin --serviceaccount=default:default**

Crear el Config Map de Postgresql

- **kubectl apply -f k8s/postgres/postgres-config.yaml**

Despliegue de Postgresql con un persistent volumen claim

- **kubectl apply -f k8s/postgres/volume.yaml**
- **kubectl apply -f k8s/postgres/postgres.yaml**

Crear el Config Map y el secret de MongoDB

- **kubectl apply -f k8s/mongodb/mongodb-config.yaml**
- **kubectl apply -f k8s/mongodb/mongodb-secret.yaml**

Despliegue de MongoDB con un persistent volumen claim

- **kubectl apply -f k8s/mongodb/volume.yaml**
- **kubectl apply -f k8s/mongodb/mongodb.yaml**

Crear el Config Map de Kafka service

- **kubectl apply -f k8s/kafka/kafka-config.yaml**

Despliegue de los cluster Zookeeper y Kafka

- **kubectl apply -f k8s/kafka/zookeeper-services.yaml**
- **kubectl apply -f k8s/kafka/zookeeper-cluster.yaml**
- **kubectl apply -f k8s/kafka/kafka-service.yaml**
- **kubectl apply -f k8s/kafka/kafka-cluster.yaml**

Despliegue del microservicio iss-pricing-service

- **kubectl apply -f k8s/iss-pricing-service-deployment.yaml**

Despliegue del microservicio iss-product-service

- **kubectl apply -f k8s/iss-product-service-deployment.yaml**

Despliegue del microservicio iss-policy-service

- **kubectl apply -f k8s/iss-policy-service-deployment.yaml**

Despliegue del microservicio iss-policy-search-service

- **kubectl apply -f k8s/iss-policy-search-service-deployment.yaml**

Despliegue del microservicio iss-payment-service

- **kubectl apply -f k8s/iss-payment-service-deployment.yaml**

Despliegue del microservicio iss-api-gateway

- **kubectl apply -f k8s/iss-api-gateway-deployment.yaml**

Activar el plugin ingress controller

- **minikube addons enable ingress**

Despliegue de una configuración ingress

- **kubectl apply -f k8s/ingress.yaml**

Definición del DNS local mastercloudapps.com

- **export MINIKUBE_IP=\$(minikube ip)**
- **echo \$MINIKUBE_IP mastercloudapps.com | sudo tee --append /etc/hosts >/dev/null**

Chequear el estatus de los pods y servicios desplegados

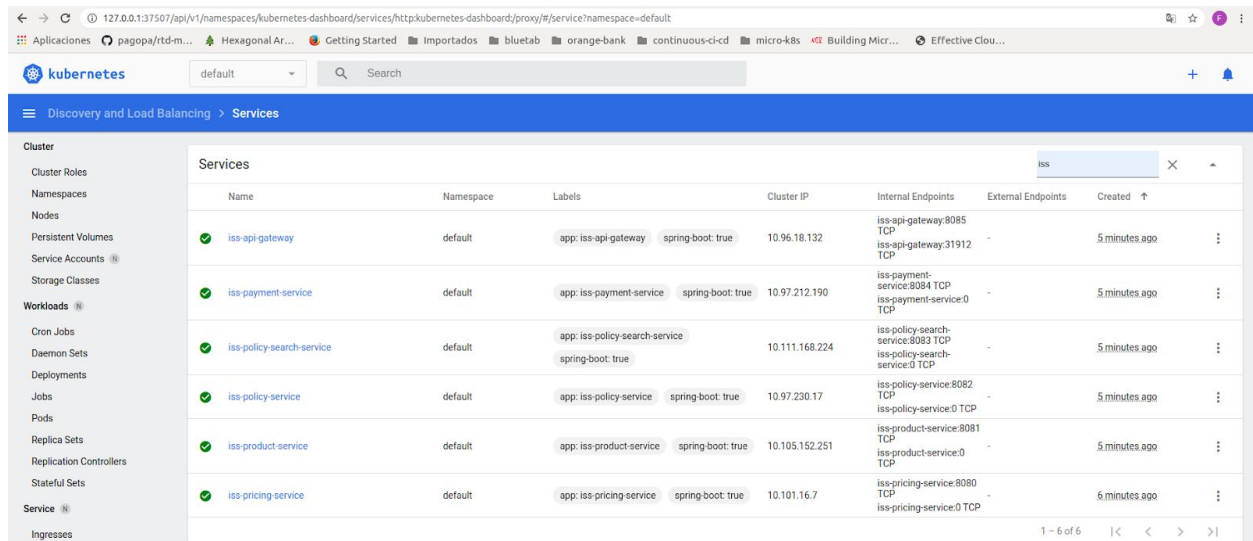
- **kubectl get pods,services**

```
francois@master-node:~/workspace/iss-api-gateway$ kubectl get pods,services
NAME                                     READY   STATUS    RESTARTS   AGE
pod/iss-api-gateway-75b9dc666d-kw8jd    1/1     Running   0           2m43s
pod/iss-payment-service-6bf87fd6b-g44rp 1/1     Running   0           3m2s
pod/iss-policy-search-service-69bb986964-g95dz 1/1     Running   0           3m11s
pod/iss-policy-service-7b8c9d5f9b-64w2m 1/1     Running   0           3m19s
pod/iss-pricing-service-86bcb77797-ntlfw 1/1     Running   0           3m45s
pod/iss-product-service-65b4f699c8-drj49 1/1     Running   0           3m30s
pod/kafka-deployment-6ff5b59ff9-dntwb   1/1     Running   0           4m29s
pod/mongodb-695f9fb84c-p2mhn            1/1     Running   0           7m29s
pod/postgres-deployment-799bf75756-7pnvd 1/1     Running   0           8m11s
pod/zookeeper-deployment-1-c6f9f84f-f6cs2 1/1     Running   0           4m56s
pod/zookeeper-deployment-2-8598d6879f-2l7kr 1/1     Running   0           4m56s
pod/zookeeper-deployment-3-7b9f67fd77-lzmxs 1/1     Running   0           4m56s

NAME                                     TYPE          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
service/iss-api-gateway                 NodePort      10.96.18.132    <none>           8085:31912/TCP   2m43s
service/iss-payment-service             ClusterIP     10.97.212.190   <none>           8084/TCP         3m2s
service/iss-policy-search-service        ClusterIP     10.111.168.224 <none>           8083/TCP         3m11s
service/iss-policy-service              ClusterIP     10.97.230.17    <none>           8082/TCP         3m18s
service/iss-pricing-service             ClusterIP     10.101.16.7     <none>           8080/TCP         3m45s
service/iss-product-service             ClusterIP     10.105.152.251 <none>           8081/TCP         3m30s
service/kafka-service                   ClusterIP     10.111.246.89   <none>           9092/TCP         4m44s
service/kubernetes                      ClusterIP     10.96.0.1       <none>           443/TCP          11m
service/mongodb                         ClusterIP     10.97.167.71    <none>           27017/TCP        7m29s
service/postgres-service                ClusterIP     10.111.70.118   <none>           5432/TCP         8m11s
service/zoo1                            ClusterIP     10.109.101.69   <none>           2181/TCP,2888/TCP,3888/TCP 5m15s
service/zoo2                            ClusterIP     10.108.80.116   <none>           2181/TCP,2888/TCP,3888/TCP 5m15s
service/zoo3                            ClusterIP     10.111.250.48   <none>           2181/TCP,2888/TCP,3888/TCP 5m15s
```

Arrancamos el dashboard de minikube para comprobar el estado de los servicios y consultar las trazas de los pods.

- **minikube dashboard**



Name	Namespace	Labels	Cluster IP	Internal Endpoints	External Endpoints	Created
iss-api-gateway	default	app: iss-api-gateway, spring-boot: true	10.96.18.132	iss-api-gateway:8085 TCP, iss-api-gateway:31912 TCP	-	5 minutes ago
iss-payment-service	default	app: iss-payment-service, spring-boot: true	10.97.212.190	iss-payment-service:8084 TCP, iss-payment-service:0 TCP	-	5 minutes ago
iss-policy-search-service	default	app: iss-policy-search-service, spring-boot: true	10.111.168.224	iss-policy-search-service:8083 TCP, iss-policy-search-service:0 TCP	-	5 minutes ago
iss-policy-service	default	app: iss-policy-service, spring-boot: true	10.97.230.17	iss-policy-service:8082 TCP, iss-policy-service:0 TCP	-	5 minutes ago
iss-product-service	default	app: iss-product-service, spring-boot: true	10.105.152.251	iss-product-service:8081 TCP, iss-product-service:0 TCP	-	5 minutes ago
iss-pricing-service	default	app: iss-pricing-service, spring-boot: true	10.101.16.7	iss-pricing-service:8080 TCP, iss-pricing-service:0 TCP	-	5 minutes ago

6. Conclusiones y próximos pasos

6.1 Beneficios de una arquitectura de microservicios

- **Desarrollo independiente:** los equipos pueden elegir las tecnologías que mejor se adapten a cada servicio utilizando varias pilas de tecnología y no están limitados por las elecciones realizadas al inicio del proyecto. Además, un solo equipo de desarrollo puede crear, probar e implementar un servicio, lo que permite una implementación más rápida y facilita la innovación continua. También es más difícil cometer errores ya que existen fuertes límites entre los diferentes servicios.
- **Implementación independiente:** los microservicios se implementan de forma independiente. Un servicio se puede actualizar sin tener que volver a implementar toda la aplicación, lo que facilita la gestión de las correcciones de errores y las nuevas funciones. Esto simplifica la implementación y la integración continua. En muchas aplicaciones tradicionales, si hay un error en una parte de la aplicación, puede bloquear todo el proceso de lanzamiento.
- **Escalado independiente:** los servicios se pueden escalar de forma independiente para adaptarse a las necesidades, optimizando los costes y el tiempo, ya que no es necesario escalar toda la aplicación como lo haría con un monolito.
- **Pequeños equipos especializados:** los equipos pueden centrarse en un solo servicio o dominio funcional, lo que hace que el código sea más fácil de entender y facilita que los nuevos miembros se unan al equipo, sin necesidad de pasar semanas averiguando cómo funciona un monolito complejo.
- **Base de código pequeño:** en una aplicación monolítica, las dependencias de código tienden a confundirse con el tiempo. Agregar nueva funcionalidad requiere cambiar el código en muchos lugares. En una arquitectura de microservicios, que no comparte código ni base de datos, estas dependencias se minimizan, lo que facilita la adición de nuevas funciones. También complementa el punto anterior de que facilita la comprensión del código y la introducción de nuevos miembros al equipo.
- **Aislamiento de datos:** en una arquitectura de microservicios, cada servicio tiene acceso privado a su base de datos (idealmente), luego podemos realizar una actualización del esquema de la base de datos sin afectar a los otros servicios. En una aplicación monolítica, las actualizaciones de esquemas pueden volverse muy difíciles y arriesgadas, ya que varias partes de la aplicación pueden utilizar los mismos datos.

- **Resiliencia:** con una arquitectura de microservicios, los puntos críticos de fallo se reducen considerablemente. Cuando un servicio deja de funcionar, toda la aplicación no deja de funcionar como lo hace con el modelo monolítico y, por lo tanto, el riesgo también se reduce cuando se desarrollan nuevas funciones. Los errores también están aislados y, por lo tanto, son más fáciles de corregir.
- **Avances tecnológicos:** los avances recientes en tecnologías en la nube y contenedorización hacen que la configuración de una arquitectura de microservicios sea cada vez más simple. Cada proveedor de nube tiene soluciones para este tipo de arquitectura para facilitar la vida de los desarrolladores.

6.2 Inconvenientes de una arquitectura de microservicios

- **Complejidad:** si bien cada servicio es más simple, el sistema en su conjunto es más complejo. Debido a que este es un sistema distribuido, se debe tener cuidado de seleccionar y configurar todos los servicios y bases de datos, y luego implementar cada uno de estos componentes de forma independiente. Se deben tener en cuenta todos los desafíos de un sistema distribuido.
- **Pruebas:** tener muchos servicios independientes puede hacer que la escritura de pruebas sea más compleja, especialmente cuando hay muchas dependencias entre servicio y terceros. Se debe utilizar herramientas impostoras tipo wiremock o mountebank en complementos de imágenes dockerrizadas de servicios de terceros para cada servicio dependiente para poder realizar las pruebas oportunas.
- **Integridad de los datos:** los microservicios tienen una arquitectura de base de datos distribuida, lo que representa un desafío para la integridad de los datos. Algunas transacciones, que requieren la actualización de múltiples funciones de la aplicación, necesitan actualizar múltiples bases de datos propiedad de diferentes servicios. Esto requiere plantear una solución de consistencia eventual de los datos, que obviamente es más compleja y menos intuitiva para los desarrolladores (Saga pattern).
- **Latencia de la red:** el uso de muchos servicios pequeños puede resultar en un aumento de las comunicaciones entre los servicios. Además, si tiene una cadena de dependencia entre servicios para realizar una transacción, la latencia adicional resultante puede convertirse en un problema. Se deben favorecer las comunicaciones asincrónicas



cuando el uso lo permita, pero esto agrega una vez más complejidad al sistema.

6.3 Próximos pasos

- Implementar con Spring Batch el batch de integración de los pagos mensuales de las pólizas.
- Implementar test de aceptación con Gherkin, Cucumber, TestContainers y Mountebank.
- Optimizar y Securizar la solución a distintos niveles de la arquitectura
 - Imagen Docker (<https://reflectoring.io/spring-boot-docker/>)
 - Cluster Kubernetes (namespace por microservicios, service account, etc ..)
 - Comunicaciones externas (token JWT) y internas (Basic Auth)
- Soporte de service Mesh con Istio para cubrir los aspectos de resiliencia:
 - Circuit-breaking,
 - Retries and timeouts,
 - Fault injection,
 - Fault handling,
 - Load balancing and failover.
- Integrar una solución de observabilidad.
- Integrar una solución de tracing tipo EFK.

7. Anexos

Los repositorios se encuentran en la cuenta github del máster en las siguientes url:

- <https://github.com/MasterCloudApps-Projects/iss-pricing-service>
- <https://github.com/MasterCloudApps-Projects/iss-policy-service>
- <https://github.com/MasterCloudApps-Projects/iss-policy-search-service>
- <https://github.com/MasterCloudApps-Projects/iss-payment-service>
- <https://github.com/MasterCloudApps-Projects/iss-product-service>
- <https://github.com/MasterCloudApps-Projects/iss-api-gateway>

Los informes de calidad se encuentran en SonarCloud en las siguientes url:

- <https://sonarcloud.io/dashboard?id=iss-pricing-service>
- <https://sonarcloud.io/dashboard?id=iss-policy-service>
- <https://sonarcloud.io/dashboard?id=iss-policy-search-service>
- <https://sonarcloud.io/dashboard?id=iss-payment-service>
- <https://sonarcloud.io/dashboard?id=iss-product-service>
- <https://sonarcloud.io/dashboard?id=iss-api-gateway>

Los imágenes docker de los microservicios se encuentran en Docker Hub en las siguientes url:

- <https://hub.docker.com/repository/docker/fpoirier2020/iss-pricing-service>
- <https://hub.docker.com/repository/docker/fpoirier2020/iss-policy-service>
- <https://hub.docker.com/repository/docker/fpoirier2020/iss-policy-search-service>
- <https://hub.docker.com/repository/docker/fpoirier2020/iss-payment-service>
- <https://hub.docker.com/repository/docker/fpoirier2020/iss-product-service>
- <https://hub.docker.com/repository/docker/fpoirier2020/iss-api-gateway>