Universidad Nacional de La Pampa
Facultad de Ingeniería

HS-API
Hardware Security API

PROYECTO FINAL DE CARRERA

Ingeniería en Sistemas

Alumno
Ramiro Higonet Lang

Tutor
Dra. Carolina Salto

Jurado
Dra. María de los Ángeles Martín
Dra. María Fernanda Papa
Ing. Ernesto Berges

General Pico, La Pampa
8 de abril de 2022

HS-API

Hardware Security API

Resumen

Las transacciones de pagos electrónicos tienen datos sensibles que necesitan ser protegidos al momento de ser impresos en registros de aplicación y/o almacenados, lo cual es posible de lograr mediante una encriptación de los datos utilizando alguna llave criptográfica digital. Una empresa que brinda soluciones de pago personalizadas y a medida de sus clientes, cuenta con varias aplicaciones que son las encargadas de gestionar cada una de las transacciones que los compradores llevan a cabo cuando utilizan sus servicios. Todas éstas aplicaciones tienen necesidades de seguridad muy similares y todas necesitan crear, almacenar y utilizar llaves criptográficas para proteger los datos sensibles.

Para cumplir con tales fines se creó **HS-API**, *Hardware Security* **API**, una aplicación java desarrollada utilizando el *framework* **Spring** que interactúa con módulos de seguridad por hardware (**HSM**) para exponer sus operaciones de una forma amigable al desarrollador. En este Proyecto Final se presenta cómo **HS-API** soluciona la gestión de llaves criptográficas **AES** (*Advanced Encryption Standard*) a distintas aplicaciones de pagos electrónicos, que utilizan dichas llaves en su administración de información de datos sensibles. Se describe a **Scrum** como la metodología utilizada para su organización, el flujo de trabajo seguido para su desarrollo, las funcionalidades de la aplicación y cada una de las operaciones que expone a través de su **API**, la arquitectura de microservicios a la que pertenece, la base de datos, las conexiones **TCP** (*Transmission Control Protocol*) con los módulos **HSM**, y también la gestión de llaves desde la perspectiva del cliente de **HS-API**.

Palabras claves: Módulos de seguridad por hardware (**HSM**), **AES**, gestión de llaves.

HS-API

Hardware Security API

Abstract

Digital payments transactions contain sensitive information that needs to be protected when it's being logged into application records or stored, which is possible to achieve by encrypting the data using a digital cryptographic key. A company that provides custom payment solutions to its clients has several applications that handle each one of the customer's payments transactions. All of these applications have very similar security needs, and they all need to create, store and use cryptographic keys to protect sensitive data.

To meet these goals, ***HS-API****, Hardware Security* ***API****, was built. A java* ***Spring*** *application that interacts with hardware security modules (****HSM****) to expose their operations in a developer-friendly way. This Final Project presents how* ***HS-API*** *solves the* ***AES*** *(Advanced Encryption Standard) key management for different electronic payment applications, which use those keys to handle the sensitive cardholder information. This project describes* ***Scrum*** *as the methodology used for its organization, the development workflow, the functionalities of the application and each one of the operations that exposes through its* ***API****, the microservices architecture to which it belongs, the database, the* ***TCP*** *(Transmission Control Protocol) connections to the* ***HSM*** *modules, as well as the key management from the perspective of the* ***HS-API*** *client.*

Key words: *Hardware security module (****HSM****),* ***AES****, key management.*

Índice general

Índice de figuras

Capítulo 1

Introducción

El Instituto Nacional de Estándares y Tecnología (*National Institute of Standards and Technology* - **NIST** [1]), departamento en Recursos de Seguridad Informática (*Computer Security Resource Center* - **CSRC** [2]) estableció la fecha de obsolescencia del algoritmo de encriptación **3DES** [3] (*Triple Data Encryption Algorithm*) y recomendó a todos los usuarios del algoritmo a migrar a la encriptación **AES** [4] (*Advanced Encryption Standard*) lo antes posible.

En la industria financiera, actualmente se encuentran invertidos millones de dólares en dispositivos que solo cuentan con soporte para tecnología **3DES**, como por ejemplo los cajeros automáticos (*Automated Teller Machine* - **ATM**). Esto indica que la transición entre los dos algoritmos va a llevar tiempo y además debe darse de manera ordenada.

Cada transacción de pago electrónico tiene datos sensibles que necesitan ser protegidos al momento de ser impresos en registros de aplicación y/o almacenados. La forma de protegerlos es mediante una encriptación de los datos utilizando alguna llave criptográfica digital. Como se crea, almacena y utilizan esas llaves digitales es algo que debe ser tratado con mucho cuidado, porque se puede utilizar el algoritmo de encriptación más fuerte del mundo, pero si se almacenan las llaves de una

manera insegura, de nada sirve.

El presente trabajo se llevó a cabo en una empresa que brinda soluciones de pago personalizadas y a medida de sus clientes, que tiene varias aplicaciones que son las encargadas de gestionar cada una de las transacciones de los compradores y llevar a cabo el pago electrónico. Todas éstas aplicaciones tienen necesidades de seguridad muy similares y todas necesitan crear, almacenar y utilizar llaves criptográficas para proteger los datos sensibles. Es por ello que se detecta la necesidad de una aplicación que centralice la gestión de éstas llaves criptográficas, y les permita a los programadores de éstas aplicaciones, una manera sencilla de realizar operaciones criptográficas sin perder bajo ningún punto de vista la seguridad.

La solución que se encontró al problema planteado fue llevar a cabo el desarrollo e implementación de la aplicación **HS-API**, *Hardware Security* ***API***, en un ambiente de microservicios que interactúe con módulos de seguridad por hardware y exponga sus operaciones de una forma amigable al desarrollador. Un módulo de seguridad por hardware, o mejor conocido como *Hardware Security Module*, **HSM**, es un dispositivo que administra de manera segura llaves criptográficas, realiza tareas de encriptación, desencriptación y otras funciones criptográficas, y cuenta con sólidos mecanismos de autenticación. Estos módulos se utilizan en la industria financiera porque ayudan a desarrollar software con niveles de seguridad más altos.

La aplicación **HS-API** cumple con las siguientes características:

- Está desarrollada en lenguaje JAVA utilizando el *framework Spring* [5] y sus herramientas.
- Permite realizar operaciones criptográficas que utilicen dispositivos **HSM** a través de una interfaz amigable al desarrollador.
    - Crear nuevas llaves digitales

    - Encriptar datos
    - Desencriptar datos
    - Otras operaciones pertinentes

- Centraliza las conexiones con los dispositivos **HSM**. Éstos son módulos que cuentan con licencia para poder usarse, y éstas limitan el número de conexiones simultáneas que soportan. Es decir, si se desea tener mayor número de conexiones simultáneas hacia éstos, se debe pagar más dinero. Por lo que, una aplicación que centralice éstas conexiones es la mejor forma de ser eficientes y ahorrar dinero.

- Está documentada correctamente siguiendo los estándares de la compañía.

- Provee una interfaz de alto nivel, que abstrae los conceptos internos de los dispositivos **HSM** y facilita la interacción con éstos y sus comandos por medio de operaciones en una **API REST** (*REpresentational State Transfer*).

- Adopta las prácticas y procesos **CI/CD** (Integración Continua y Entrega Continua) que se practican en la empresa.

El objetivo de este proyecto de trabajo final es contar la experiencia de desarrollo e implementación de la aplicación, pasando por puntos claves como la metodología utilizada para su organización, las funcionalidades de la aplicación, la arquitectura y base de datos, como también la gestión de llaves desde la perspectiva del usuario. Es por ello que el documento se estructura en los siguientes capítulos:

- **Capítulo 1 - Introducción:** se presenta la problemática a resolver y la solución encontrada, **HS-API**. Además se desarrollan conceptos necesarios para el entendimiento del proyecto como: los Módulos de Seguridad por Hardware (**HSM**), algunos definiciones claves de criptografía y microservicios, y por último, se introduce **Spring** como el *framework* de desarrollo utilizado.

- **Capítulo 2 - Metodología de desarrollo:** se explica la metodología de desarrollo de software utilizada en el proyecto adoptando a **Scrum** como *framework*. Así mismo se describe cómo fue la organización del flujo de trabajo diario para su desarrollo.
- **Capítulo 3 - Aplicación: HS-API:** se describe la aplicación, las funcionalidades que cumple, la arquitectura a la que pertenece, la base de datos, y las conexiones con los módulos **HSM**.
- **Capítulo 4 - Gestión de llaves AES:** se expone la gestión de llaves criptográficas **AES** desde la perspectiva del cliente de **HS-API**.
- **Capítulo 5 - Conclusión:** por último se enuncian las conclusiones sobre la experiencia del proyecto final y se describen algunas características proyectadas para el futuro de **HS-API**.

A continuación se introducirán los dispositivos **HSM**, algunos conceptos criptográficos importantes como **KEK** (*Key Encryption Key* - llave de encriptación de llave), **AES**, **3DES**, entre otros, para finalmente caracterizar y el marco de trabajo a utilizado, **Spring Boot**.

1.1. HSM: Módulos de Seguridad por Hardware

Un módulo de seguridad por hardware es un dispositivo de hardware resistente a la manipulación de externos, que se utiliza principalmente en la industria financiera para proporcionar altos niveles de protección a las claves criptográficas y a la información de las tarjetas de los clientes. Generalmente, éstos módulos se emplean para proporcionar llaves para funciones críticas como encriptación, desencriptación y autenticación.

Las siguientes son características de los **HSM** que contribuyen a su seguridad:

- **Diseño seguro:** utilizan hardware especialmente diseñado que adhiere a los estándares **FIPS** [6] (Estandarización Federal de Procesamiento de Información, *Federal Information Processing Standardization*), a los Criterios Comunes [7], y a los requisitos impuestos por la industria de pagos con tarjeta **PCI DSS** [8] (*Payment Card Industry Data Security Standard*).

- **Resistencia a la manipulación:** se someten a procesos que los vuelven resistentes a accesos malintencionados y manipulación ilícita.

- **Sistema operativo seguro:** su sistema operativo que fue construido basado en la seguridad.

- **Aislamiento:** se ubican físicamente en el área segura del centro de datos para evitar el acceso no autorizado.

- **Control de acceso:** controlan el acceso a sus datos, están diseñados para evidenciar los signos de manipulación y algunos tienen la funcionalidad de volverse inoperables o eliminar sus claves criptográficas si la detectan.

La función más importante de un módulo **HSM** es la de autenticación de llaves. Es decir, asegurarse de que sus operaciones solo se puedan realizar con llaves que hayan sido previamente autorizadas por un administrador de seguridad. ¿Cómo pueden asegurar esto? Para entender cómo funciona debemos hablar de un concepto muy importante, el de Clave Local Maestra (*Local Master Key* - ***LMK***). **LMK**, es la llave de encriptación maestra que utiliza el dispositivo **HSM** para proteger las demás llaves criptográficas. Por lo general, una **LMK** preserva todas las llaves de una institución.

Como podemos observar en la Figura 1.1, las claves **LMK** no son usadas para encriptar datos, son usadas para encriptar/desencriptar otras llaves criptográficas, las cuales se utilizan para realizar operaciones en los **HSM**. Las **LMK** aseguran

que por más de que el tráfico de entrada y salida sea interferido, el verdadero valor de las llaves no se vea comprometido.

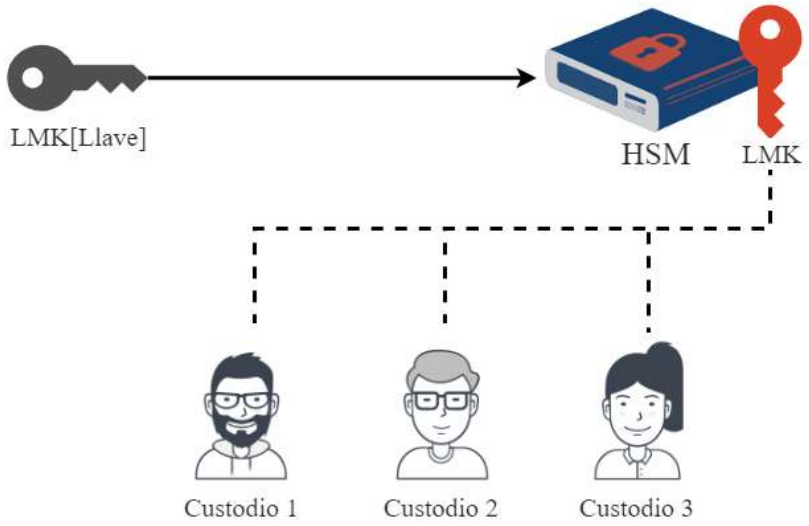

Figura 1.1: Concepto de Clave Local Maestra (**LMK**)

Cuando el dispositivo recibe una operación LMK[Llave][1], internamente sabe que necesita desencriptar el contenido con la **LMK** para poder acceder al verdadero valor de la llave. Es la única forma de acceder al valor original y esto es lo que vuelve a los dispositivos **HSM** tan seguros. Además, la **LMK** se instala bajo un procedimiento establecido y luego de esto se pierde el acceso a su valor. Una **LMK**, por lo general, está dividida en 3 componentes (componentes de clave), los cuales son resguardados por distintos custodios. Para guardar cada componente se suelen utilizar tarjetas inteligentes que preservan el valor de forma automática, eliminando el error humano de hacerlo manualmente (escribir en un documento físico el valor del componente).

[1]La notación LMK[Llave] es una forma estándar de indicar que una llave es protegida por otra. Llave1[Llave2] quiere decir que la Llave2 está protegida por Llave1 y que para poder ver el contenido original de Llave2, se necesita realizar una desencriptación con Llave1.

Es importante notar que si se pierde la **LMK**, se pierde el acceso a las llaves que se tienen resguardadas con ella, por eso que es muy importante que los custodios guarden los componentes. En el caso de cualquier eventualidad, se puede recuperar la **LMK** repitiendo el procedimiento de instalación en un dispositivo **HSM**, y se pueden seguir accediendo a las llaves que se encuentran protegidas por ésta.

1.2. Conceptos claves

Antes de empezar a dar más detalles el proyecto, se necesitan mencionar algunos conceptos claves que ayudarán a comprenderlo mejor. La idea no es profundizar en estos, sino darle al lector una noción para que pueda entender los distintos procesos.

- **KEK**: llave de encriptación de llave, como su nombre delata rápidamente, es una llave criptográfica que se utiliza para proteger otras llaves. En la industria financiera, es muy común que estas llaves se utilicen para proteger otras llaves al momento de almacenarlas en alguna base de datos.

- **AES**: *Advanced Encryption Standard*, es un algoritmo de encriptación estándar adoptado por el gobierno de los Estados Unidos. Precede al algoritmo **DES** [9] (*Data Encryption Standard*), es rápido tanto en software como en hardware, fácil de implementar y no requiere mucha memoria. El algoritmo descrito por **AES** es de claves simétricas, esto quiere decir que se utiliza la misma llave para encriptar y desencriptar el mensaje.

- **3DES**: **Triple DES** (**3DES** o **TDES**) es el algoritmo de encriptación de claves simétricas que realiza tres veces el cifrado **DES**. Esto es debido a que se descubrió que un solo cifrado **DES** es vulnerable a fuerza bruta. Igualmente esto no fue suficiente y en 2016 **NIST** encontró un problema de seguridad

mayor [10] en ambos algoritmos y determinó la obsolescencia en aplicaciones nuevas en 2017 y para las ya existentes en 2023.

- **Pod**: un pod se define como la unidad mínima desplegable que se puede crear y gestionar en **Kubernetes** [11]. Representa una instancia única de ejecución de un proceso o aplicación en el cluster. Por lo general un pod contiene una sola imagen pero pueden contener más. En el caso de múltiples imágenes éstas comparten los recursos de red, memoria y almacenamiento.

1.3. Introducción al *framework* Spring

Spring es un *framework* de código abierto para el desarrollo de aplicaciones en lenguaje Java, el cual proporciona una infraestructura de soporte al desarrollador. Se encarga de unir los componentes de la aplicación, manejar su ciclo de vida y la interacción entre estos.

Si bien las características fundamentales de **Spring** pueden ser usadas en cualquier aplicación desarrollada en Java, existen variadas extensiones para facilitar la construcción de aplicaciones web complejas. A pesar de que no impone ningún modelo de programación en particular, este *framework* se ha vuelto popular en la comunidad al ser considerado una alternativa, sustituto, e incluso un complemento al modelo **EJB** [12] (*Enterprise JavaBeans*).

Se puede ver a **Spring** como una fábrica de objetos, ya que permite la definición de *Java Beans* que relacionan a un nombre con una implementación. Así, al utilizar el contexto provisto por el *framework* (*Spring ApplicationContext*) se puede pedir el objeto por su nombre y obtener una instancia de la implementación asociada.

Entre sus características más importantes se pueden destacar las siguientes:

- Inyección de dependencias y control de versiones

- Provisión de servicios listos para producción, tales como métricas, monitoreo (*health checks*) y la posibilidad de externalizar la configuración
- Acceso a distintas bases de datos
- Gestión de transacciones
- Programación orientada a aspectos
- Modelo Vista Controlador
- *Framework* liviano, es decir no obliga al programador a heredar ninguna clase ni a implementar ninguna interfaz.

Y para entender lo amplio y potente que es Spring como *framework*, a continuación se enumeran todas las aplicaciones que se pueden construir con él:

- **Microservicios**: permite hacer entrega de microservicios independientes e iterables con grado de producción fácilmente.
- ***Reactive***: con sus características asíncronas y su arquitectura no bloqueante, permite hacer un uso más eficiente de los recursos.
- ***Cloud***: permite conectar y escalar los servicios con cualquier plataforma en la nube.
- **Aplicaciones Web**: permite construir aplicaciones web conectadas a cualquier almacenamiento de datos de una manera rápida, segura y responsiva.
- ***Serveless***: permite construir aplicaciones flexibles que escalen en demanda cuando se requiera y cuando no, que se apague para no consumir recursos.
- **Sistemas dirigidos por evento**s: aplicaciones de negocio que puedan responder a eventos, que actúen en tiempo real en base a distintas entradas de datos.

- ***Batch***: permite automatizar tareas y procesar los datos fuera de línea en el tiempo que se desee.

1.3.1. Componentes Spring

El *framework* está compuesto por una serie de módulos que brindan a los programadores los medios y servicios necesarios para desarrollar aplicaciones de acuerdo a sus necesidades. Es importante establecer que no necesariamente se tienen que utilizar todos los módulos, el programador tiene libre albedrío para decidir qué módulos utilizar en su aplicación.

En la Figura 1.2, extraída de la documentación oficial de **Spring**, podemos observar sus distintos componentes y cómo se agrupan.

De todos los módulos descritos en la Figura 1.2, solo los que se listan a continuación son utilizados en la construcción de la aplicación del proyecto:

- ***Core Container***: es el módulo que provee la funcionalidad fundamental de **Spring**. Contiene la fábrica de Beans (*BeanFactory*), que representa al contexto de **Spring** y la base de la inyección de dependencias.

- ***Data Access/Integration***: este es uno de los módulos más importantes de **Spring**, ya que libera al programador de la interacción con los gestores de base de datos.

- **Web**: módulo que brinda el soporte para la construcción de aplicaciones basadas en el patrón **MVC** [13] (Modelo Vista Controlador).

- ***Messaging***: este módulo introduce abstracciones claves para la construcción de mensajes y su manejo. Como por ejemplo mensajes **TCP** [14] (*Transmission Control Protocol*) y **UDP** (*User Datagram Protocol*).

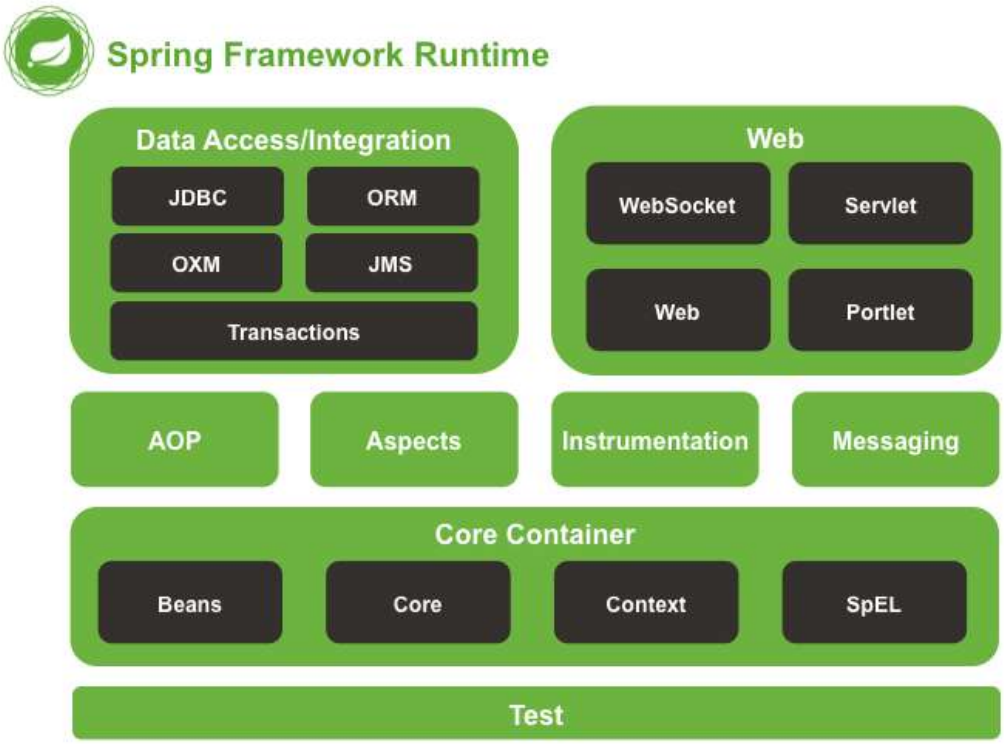

Figura 1.2: Componentes Spring

1.3.2. Spring Boot

Spring Boot es la forma amigable y rápida de crear una aplicación lista para producción basada en el *framework* **Spring**. Favorece las convenciones sobre la configuración y está diseñado para comenzar a trabajar lo más rápido posible. ¿Qué tan rápido? Como se muestra en el siguiente fragmento de código:

```
@SpringBootApplication
@RestController
public class DemoApplication {
```

```
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

    @GetMapping("/hola")
    public String holaMundo(@RequestParam(value = "name",
    defaultValue = "mundo") String name) {
        return String.format("Hola %s!", name);
    }
}
```

Este es un ejemplo de una aplicación lista para ejecutarse, que expone el método `holaMundo` a través de una operación **HTTP GET**.

1.3.3. Uso en el proyecto

Para construir la aplicación se utiliza **Spring Boot** en su versión **2.4** y **Spring** en su versión **5**.

Para agregar los distintos módulos de **Spring** y dependencias a librerías externas, se usa **Gradle** [15] escribiendo su archivo de configuración en lenguaje **Kotlin**. En el cual se destaca el siguiente fragmento que contiene los módulos **Spring** descritos previamente y otros que merecen ser mencionados.

```
// Interacción con la base de datos
implementation("org.springframework.boot:spring-boot-starter-data-
    jpa")
// Autenticación y Autorización de usuarios
implementation("org.springframework.boot:spring-boot-starter-
    security")
```

```
// OAuth estandar
implementation("org.springframework.security:spring-security-oauth2-
    resource-server")
// Módulo web - MVC
implementation("org.springframework.boot:spring-boot-starter-web")
// Trazabilidad de transacciones
implementation("org.springframework.cloud:spring-cloud-starter-
    sleuth")
// Cliente TCP
implementation("org.springframework.boot:spring-boot-starter-
    integration")
```

Solo eso fue necesario para empezar a utilizar **Spring** y al resto de las dependencias. A continuación se introducen los módulos que no se mencionaron anteriormente:

- **Sleuth** es una librería que al incluirla a la aplicación se encarga de generar un identificador de 64-bit aleatorio para la trazabilidad de transacciones.
- **Spring *Security*** es un poderoso y customizable *framework* de autenticación y autorización.
- **OAuth** [16] es un protocolo estándar sobre cómo brindar autorización a los usuarios para aplicaciones web, móviles y de escritorio.

Capítulo 2

Metodología de desarrollo

El desarrollo de software no es una tarea fácil, por ello es que existen numerosas propuestas metodológicas que inciden en distintas dimensiones del proceso de desarrollo. Una **metodología de desarrollo de software** es un marco de trabajo que se usa para estructurar, planificar y controlar el proceso de desarrollo de sistemas de información. Es un conjunto de técnicas y métodos organizativos que se aplican para diseñar soluciones de software informático, con el objetivo de intentar organizar los equipos de trabajo para que estos desarrollen las funciones de un programa de la mejor manera posible.

Existen dos grandes categorías de propuestas a elegir, las tradicionales y las ágiles, siendo éstas últimas las que han ganado mayor popularidad en la industria del software los últimos años y la escogida para el desarrollo de éste proyecto. Las metodologías ágiles son aquellas que permiten adaptar la forma de trabajo a las condiciones del proyecto, consiguiendo flexibilidad e inmediatez en la respuesta para amoldar el proyecto y su desarrollo a las circunstancias específicas del entorno. Y dentro de ésta gran categoría **Scrum** [17] fue el marco de trabajo seleccionado.

Se seleccionó **Scrum** porque es una metodología rápida, adaptativa, iterativa y flexible, que asegura transparencia en las comunicaciones y crea un ambiente de

responsabilidad y progreso continuo. Entre otras cosas, además, está centrada en el cliente, en la entrega de valor a etapas tempranas del desarrollo, y de una manera continua, iterativa e incremental, que gracias a su retroalimentacion constante, permite mejorar y corregir errores. En la Sección 2.1 se narrará la experiencia vivida al aplicar este *framework*.

Las herramientas que se utilizan en las tareas diarias del desarrollo del software impactan en el día a día y rendimiento del equipo, ya que contar con las herramientas correctas hará la labor mucho más fácil. Además de mencionar el marco de trabajo elegido, a continuación se listan las herramientas utilizadas para desarrollar las aplicaciones en la empresa y las seleccionadas para llevar a cabo **HS-API**, que luego en la Sección 2.2, se relatará como fue la interacción con ellas en los distintos pasos del desarrollo.

- **Scrum**: como *framework* de desarrollo.
- **Jira**: como herramienta de software para gestionar el desarrollo bajo el marco de **Scrum**.
- **GitHub**: como repositorio del código fuente de la aplicación y controlador de las distintas versiones utilizando **Git**.
- **TeamCity** [18]: como herramienta de integración e implementación continua.
- **Octopus Deploy** [19]: como herramienta de instalación de la aplicación en sus distintas versiones y en los diferentes ambientes.

2.1. Scrum

Scrum fue la metodología escogida para realizar el proyecto, es la metodología que se usa día a día en el trabajo y es la más usada por la industria del soft-

ware [20]. **Scrum**, como sus creadores lo definen, es un *framework* que permite a las personas abordar problemas complejos mientras productiva y creativamente entregan productos del mayor valor posible.

Esta sección no estará centrada en explicar en detalle los conceptos teóricos de **Scrum**, sino que estará enfocada en la experiencia vivida utilizando este *framework* para llevar a cabo el proyecto de este trabajo.

2.1.1. Equipo

El equipo **Scrum** estaba compuesto por cinco personas las cuales dividían sus responsabilidades en tres roles distintos:

- **Dueño del producto**: una persona, encargada de optimizar y maximizar el valor del producto. Se encarga de gestionar las peticiones de los clientes y generar las historias de usuario a implementar. Dicta las prioridades.
- **Scrum *master***: una persona, facilitadora del equipo. Tiene la tarea de ayudar a resolver cualquier problemática en la cual se encuentre el equipo, como así también de ayudar en la organización de reuniones y la utilización de distintas herramientas que **Scrum** provee.
- **Desarrolladores**: tres personas, encargadas de desarrollar el producto, auto-organizándose y auto-gestionándose para conseguir entregar un incremento de software al final de cada ciclo de desarrollo.

Cabe destacar que el equipo no solo estaba destinado a las tareas de la aplicación **HS-API**, sino que también tiene la tarea de desarrollar, mantener y monitorear los demás microservicios de la compañía. También, es importante resaltar que la escritura del código fuente de la aplicación estuvo a cargo exclusivamente por

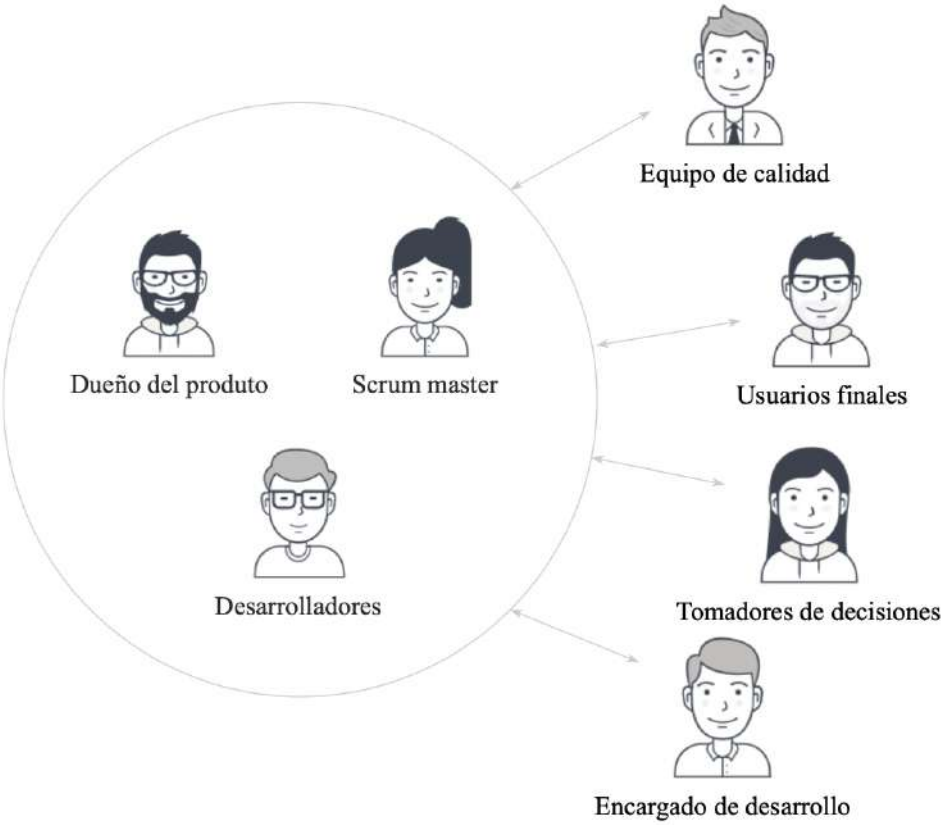

Figura 2.1: Equipo **Scrum** de la empresa a cargo del desarrollo de **HS-API**

el autor del presente trabajo. El resto de los desarrolladores estuvo abocado a codificar otros proyectos dentro de la empresa.

Aunque el equipo estaba compuesto por cinco personas, el grupo de trabajo incluía a otros que ayudaron a lograr el producto final, cómo se ve en la Figura 2.1. El equipo de calidad de la empresa impuso desde un principio el flujo de trabajo a seguir al momento de implementar los cambios en el código e instalar la aplicación en los ambientes de prueba y producción. Así mismo, los *stakeholders* (también conocidos como grupo de interés) fueron muy importantes en el desarrollo ya que ayudaron a tomar decisiones, su constante *feedback* enriqueció el proyecto y se aprendió mucho de su experiencia. En este caso los *stakeholders* fueron los usuarios

finales de la aplicación, personas con poder para tomar decisiones, y el encargado de desarrollo. Los usuarios finales de la aplicación tenían experiencia previa en algoritmos de encriptación y fueron los implementadores del obsoleto algoritmo **3DES** en las aplicaciones de pago, por lo que su antigüedad en la materia fue de gran utilidad. Las personas con poder para tomar decisiones dentro de la compañía trabajan en distintos proyectos y tienen una visión global distinta a la del equipo **Scrum**. Por último, el encargado de desarrollo fue una fuente de conocimiento a la hora de la implementación y ayudó a resolver dudas específicas en los procesos que involucran encriptación de datos.

2.1.2. Ceremonias de Scrum

Para poder decir que un equipo está aplicando **Scrum**, debe implementar las ceremonias obligatorias que el *framework* propone. Son cuatro tipos de reuniones, las cuales tienen una ocurrencia y duración recomendada por las buenas prácticas propuestas por **Scrum**. Ellas son:

- **Sprint Planning**
- **Daily Scrum**
- **Sprint Review**
- **Sprint Retrospective**

La Figura 2.2 muestra cómo funciona un **Sprint** del equipo. Los **Sprints** se ajustan a la duración recomendada de dos semanas comenzando un miércoles y terminando un martes. El comienzo del **Sprint** está marcado por la reunión de **Sprint Planning** donde, los miembros del equipo se reúnen a establecer cuáles serán los *tickets* que formarán parte del **Sprint**, cuáles son las prioridades, y en

Figura 2.2: Calendario de un **Sprint**

caso de que sea necesario, refinar y estimar los *tickets* que lo necesiten. Una vez que todo el equipo está de acuerdo con lo que se planeó, comienza el **Sprint**.

Durante el **Sprint**, todos los días el equipo se junta a realizar la **Daily Scrum**. Una reunión muy breve de quince minutos, que tiene características en común con la conocida *stand-up meeting*. Aquí el objetivo es que cada uno de los participantes

pueda contestar tres preguntas: "¿qué hice?", "¿qué estoy haciendo?" y "¿qué planeo hacer?". Es una reunión informal que dada su frecuencia diaria genera sinergia en el grupo para que todos los miembros estén actualizados con el progreso y estado del **Sprint**. Además es muy útil para identificar bloqueantes en alguna tarea y que el **Scrum Master** pueda realizar su labor y ayudar a eliminar los impedimentos.

Luego, los dos viernes del **Sprint** se tiene la reunión llamada **Backlog Grooming**, que si bien es muy recomendada por **Scrum** no está dentro de las reuniones obligatorias. Aquí el equipo se reúne para organizar el **Backlog**, esto significa refinar los *tickets* que lo necesiten y estimar los que aún no hayan sido estimados. Además, es una oportunidad para que todos los miembros del equipo o *stakeholders* propongan *tickets*, como por ejemplo un *refactor* que quieran hacer, alguna mejora en un proceso o reportar algún *bug*. La intención de estas reuniones es llegar a la **Sprint Planning** en una mejor posición con la mayor cantidad de *tickets* estimados y refinados posibles, para que organizar el próximo **Sprint** se vuelva una tarea más fácil.

Y por último, el **Sprint** cierra con dos reuniones, ellas son **Sprint Review** y **Sprint Retrospective**. El concepto de **Sprint Review** es muy simple, cada uno de los desarrolladores expone al resto del equipo y *stakeholders* en qué trabajó durante el *Sprint*. Explica uno por uno de los *tickets* trabajados y cuenta qué pasó, si fueron implementados en su totalidad o parcialmente, si se crearon nuevos *tickets* para continuar el trabajo en un futuro **Sprint**, si ocurrió algún cambio en la documentación, o si se desarrolló una característica que afecta al cliente. Es común hacer una demostración para que los *stakeholders* puedan verlo. En la reunión de **Sprint Retrospective** el equipo se junta para dejar escrito en un documento las respuestas a las preguntas: "¿Qué hicimos bien en este **Sprint**", "¿Qué podemos hacer mejor?", "¿Qué podríamos hacer para mejorar?". Además se deja asentada

una lista de las acciones a las que el equipo se compromete para seguir mejorando. La **Sprint Retrospective** es la retro-alimentación del proceso de **Scrum** en el equipo, es la forma que se tiene de ir afinando el proceso y la herramienta de desarrollo.

2.1.3. Organización del proyecto

Como ya se mencionó, **Scrum** es el marco de trabajo que se utilizó para realizar el proyecto. En este capítulo se describirá cuál fue el papel del *framework* durante el proceso del desarrollo.

Desarrollar una aplicación entera puede ser una labor muy tediosa si no se la divide correctamente en los *tickets* adecuados. Para empezar el desarrollo de **HS-API** se describió una épica, la cual es un tipo de *ticket* **Scrum** que agrupa historias de usuario. La épica se considera completa cuando todas sus historias de usuario hayan sido completadas. Éstas historias de usuario a su vez son dividas en tareas.

En la Figura 2.3 se puede observar un mapa de *tickets* que ayuda a comprender cómo fue dividido el trabajo. El mismo no fue posible de realizar sin antes conocer los requisitos de la aplicación. En la Sección 3.1 se encuentra una lista de los requisitos funcionales y no funcionales de **HS-API**. A partir de ellos, se crearon cuatro historias de usuario y cada una de ellas se dividió en distintas tareas.

Es importante entender que el mapa de *tickets* de la Figura 2.3 es algo que se armó al comienzo del proyecto. En el momento en que se gestionan los requerimientos y de divide el trabajo en *tickets* se trata de tener la vista global de todo el trabajo que hay que hacer. Aunque luego en la práctica van surgiendo algunos cambios en los requisitos originales, por lo que hay que realizar ajustes. Además de que en la práctica, por más que se trate de pensar globalmente y de tratar de gestionar todos los detalles, hay *tickets* que no fueron creados y que luego son necesarios para el proyecto y se deben ir agregando. Por suerte **Scrum** al ser una

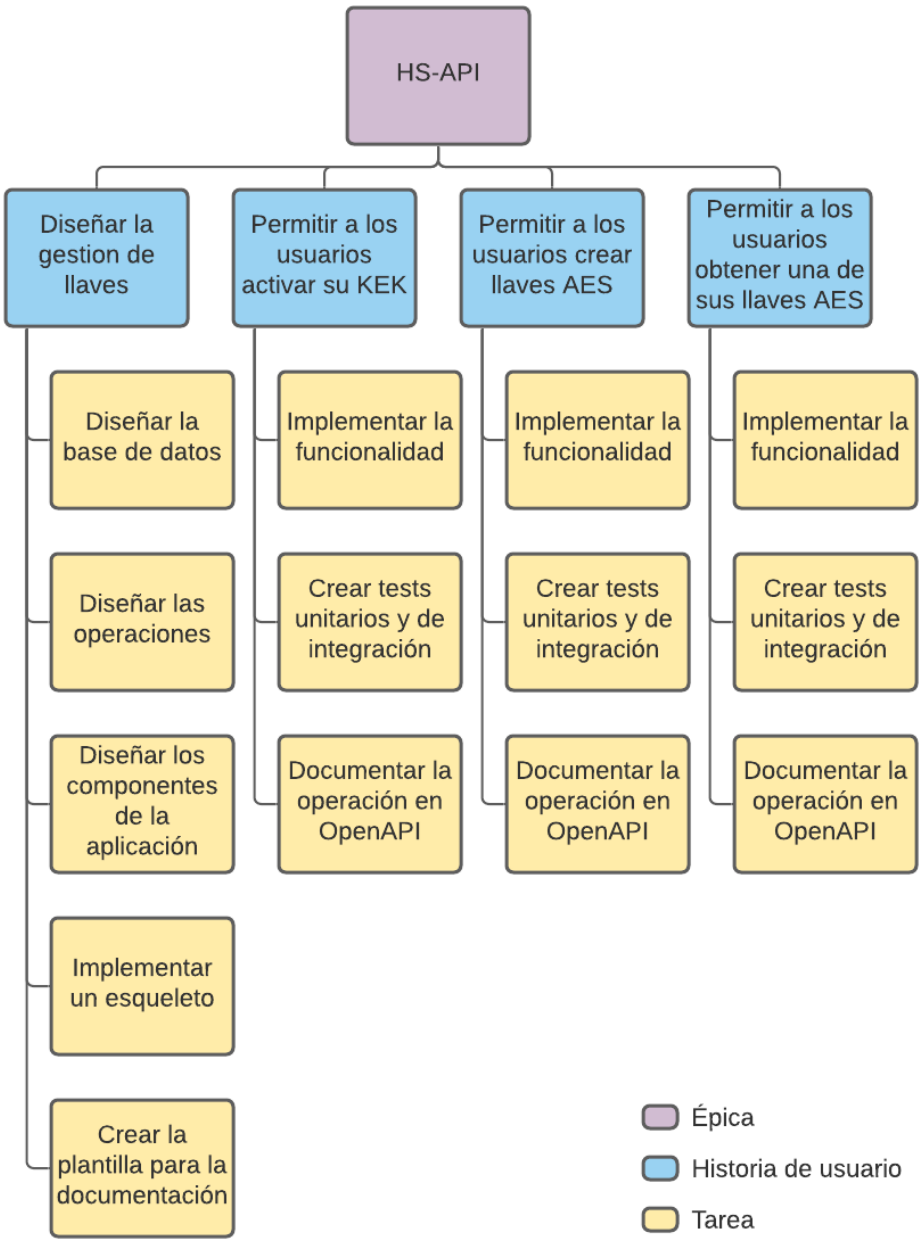

Figura 2.3: Mapa de *tickets*

metodología ágil contempla estas situaciones y al ir transcurriendo los distintos **Sprints**, la retroalimentación permite hacer los ajustes necesarios.

Como costumbre del equipo, siempre que hay una épica, hay una historia de usuario que le da valor al usuario final donde se incluyen las tareas de diseño, donde hay idas y vueltas entre los distintos miembros del equipo, con los clientes, usuarios y compañeros de más experiencia. Como también hay trabajo que está estandarizado y que le agrega enorme valor al producto, pero al ya estar hecho no se ve reflejado en los *tickets*, pero que terminan siendo parte de la aplicación.

2.2. Flujo de trabajo

En esta sección se narrará cómo fue el flujo de trabajo seleccionado para este proyecto y que se sigue actualmente día a día para construir, mejorar, diseñar y monitorear las distintas aplicaciones que se desarrollan en la empresa.

Para explicar todo el flujo desde comienzo a fin, como se puede observar en la Figura 2.4, se desarrolla todo el proceso desde que se selecciona un *ticket* del **Sprint** actual, hasta que es resuelto e instalado en el ambiente de pruebas con el cliente.

Se simulará que se está en medio del **Sprint** descrito en la Figura 2.5 y que se acaba de finalizar de trabajar un *ticket*. Por lo tanto se debe tomar otro para seguir con el trabajo y poder llegar al final del **Sprint** con todo lo planeado realizado. Los *tickets* que se encuentran en la lista para realizar son previamente ordenados por prioridad en el **Sprint Planning** y ya han sido refinados por completo, es decir que se conocen sus requerimientos y su resultado esperado. Por lo tanto, se podría tan solo tomar el primero de la lista.

Una vez que ya se tiene el *ticket*, se debe asegurar de entender todos los requisitos y resultados que se esperan de él, se analiza si hace falta dividirlo en tareas, en caso de ser una historia de usuario, o en sub-tareas en caso de una tarea. El tipo de trabajo a realizar puede ser diverso como: investigación de tecnologías,

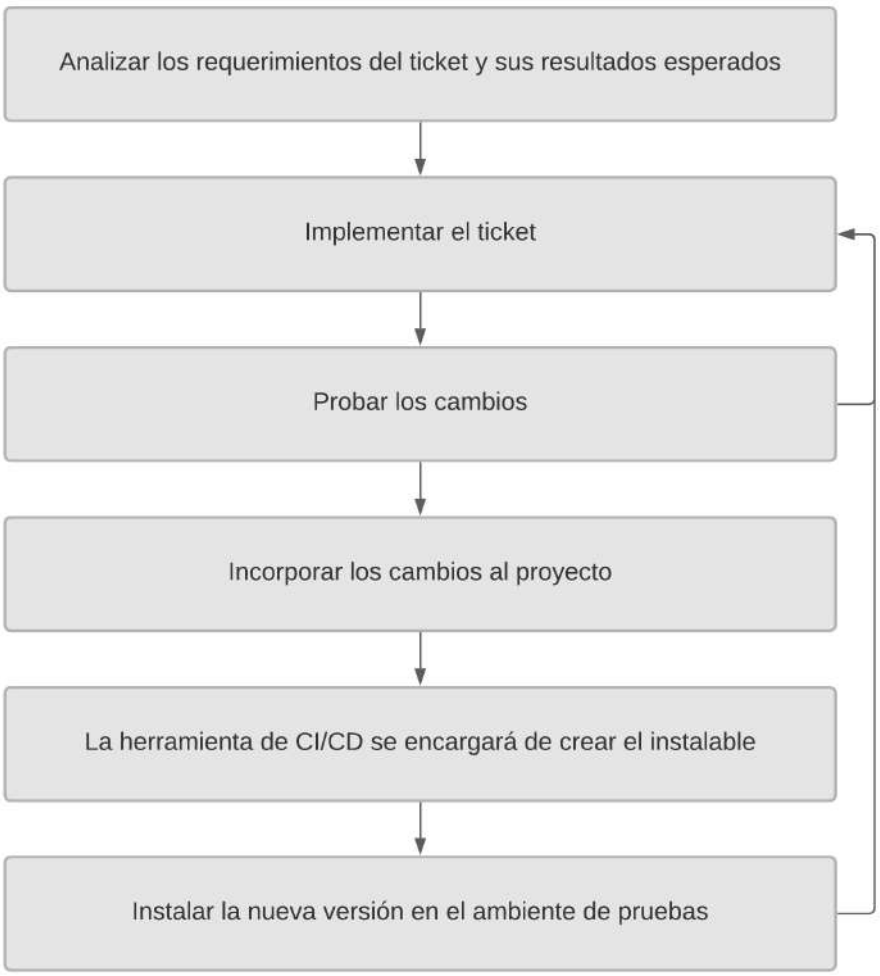

Figura 2.4: Flujo de trabajo

probar funcionalidades, hacer una prueba de concepto (*proof of concept* - **POC**), implementar una funcionalidad, diseñar un componente, etc. Lo más común en un proyecto de desarrollo es que los *tickets* sean de implementación, por lo tanto, para este ejemplo práctico se hará de cuenta que el *ticket* seleccionado requiere implementar una funcionalidad.

La Figura 2.6 presenta el *ticket* que se utilizará de ejemplo. Es una tarea muy simple, ya ha sido implementada la funcionalidad para obtener llaves **AES** exis-

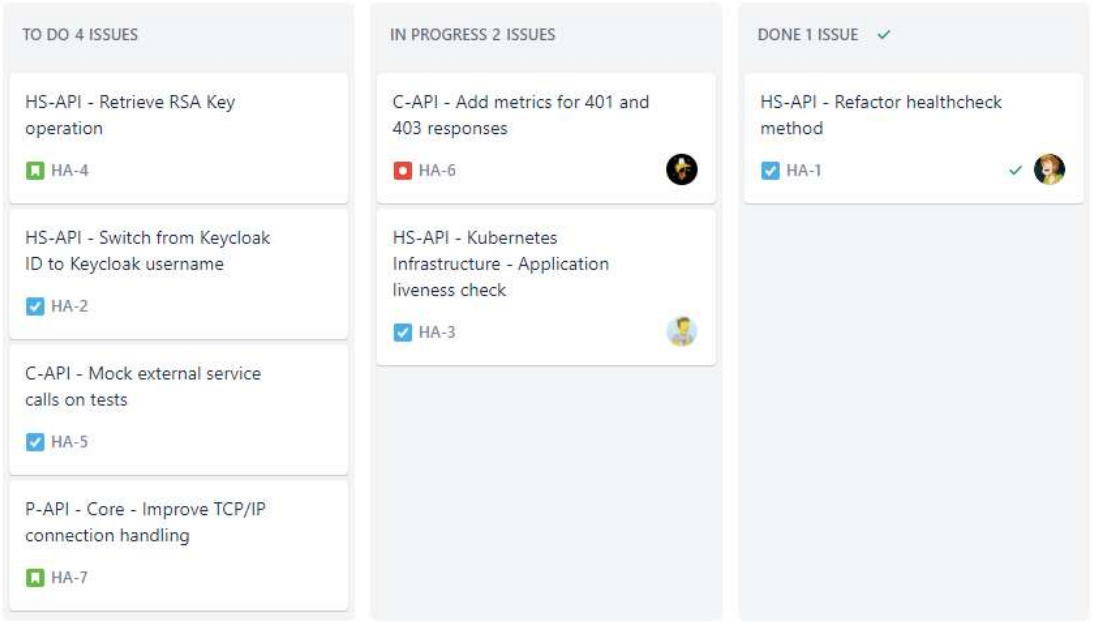

Figura 2.5: **Sprint** actual

tentes, solo se debe exponer la operación a través de la **API** de **HS-API**. Ahora que ya ha sido presentado el *ticket* a desarrollar, se enumerarán los pasos que se siguen para completarlo en el flujo de trabajo:

1. **Analizar los requerimientos del *ticket* y sus resultados esperados**: la teoría de **Scrum** establece que los *tickets* que se encuentran en el Sprint ya han sido refinados y estimados, por lo que se consideran listos para ser llevados a cabo, aunque en la práctica esto no siempre ocurre y lo primero que se debe hacer es analizar los requerimientos del mismo, ver que tengan sentido y que sean posibles de lograr. Esto es importante porque permite identificar bloqueantes antes de que sucedan y porque dan lugar a la mejora. Por ejemplo, en el transcurso del proyecto se dio que, una tarea proponía implementar un filtro de **Spring** para ciertas transacciones, y luego al momento de su implementación se sugirió un cambio en la tecnología para poder llegar

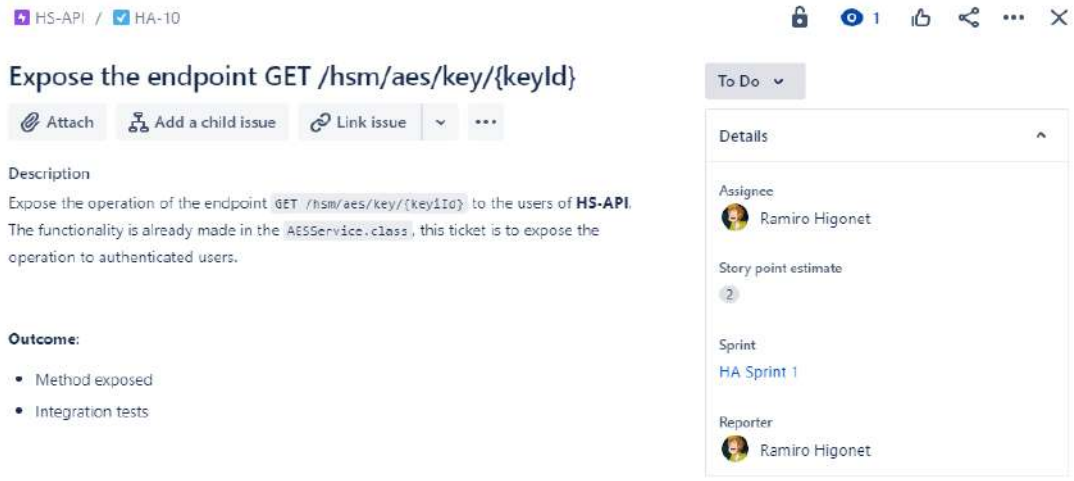

Figura 2.6: *Ticket* a implementar

al resultado esperado de una manera más eficiente, mantenible y escalable.

2. **Implementar el ticket**: para implementar el *ticket* hay que seguir las reglas de desarrollo, para lo cual se siguen los siguientes pasos:

   a) Es necesario tener el proyecto en la rama *develop* en su versión más reciente, para asegurarlo se ejecuta el comando `git pull`.

   b) Se crea una rama para desarrollar la característica, se ejecuta el comando `git checkout -b feature/HA-10`. Se sigue el estándar `feature/<identificador del ticket>` lo que permite generar una trazabilidad de los cambios en las distintas plataformas, tanto **Jira** como **GitHub**.

   c) Se implementa el código necesario para cumplir con lo solicitado.

3. **Probar los cambios**: es muy importante entender la importancia de las pruebas automatizadas. Generar los casos de prueba como parte del código de la aplicación y que estos puedan ejecutarse automáticamente, sin intervención humana, ahorran mucho tiempo y optimizan la productividad del

equipo. Cuando los sistemas se van volviendo más complejos es posible que un cambio genere impacto en componentes que no se tenían en cuenta. Las pruebas automatizadas permiten identificar este tipo de escenarios en una etapa temprana para poder resolverlos en el desarrollo y que no tengan un impacto negativo en el usuario final. En el caso del *ticket* seleccionado, que implica exponer una funcionalidad, se suelen crear pruebas de integración que ejecuten la aplicación y simulen la ejecución del método desde la perspectiva del usuario esperando una respuesta determinada. De este modo se prueba la funcionalidad y el contrato que se tiene con el cliente.

4. **Incorporar los cambios al proyecto**: una vez que los cambios del código han sido completados y tienen sus pruebas automatizadas listas, ya se pueden incorporar al proyecto. Para ello es necesario enviarlos al repositorio en **GitHub** con el comando `git push origin feature/HA-10`. Los cambios no son enviados a la rama de desarrollo principal, sino que son enviados a la nueva rama que hemos creado específicamente para el *ticket*, una vez que la rama se encuentra en el repositorio se puede crear un **pull request**, que es una petición de combinar los nuevos cambios con la rama *develop*.

   En la Figura 2.7 se aprecia como se ve un **pull request** desde **GitHub**. El proceso es simple, una vez que la rama `feature/HS-10` ya se encuentra en el repositorio, se procede a crear la petición de **pull request**, allí se respeta la nomenclatura `<identificador de ticket> - <nombre del ticket>` como su título y se agregan a los pares del equipo como revisores. Los revisores tienen la labor de comprobar que los cambios sean correctos. Pueden sugerir modificaciones, ya sea porque detectan un error o porque ven la posibilidad de mejorar el código. Para poder combinar los cambios de la nueva rama con la rama *develop*, es necesario que al menos uno de los revisores haya aprobado los cambios y que las pruebas automatizadas, que se ejecutan por

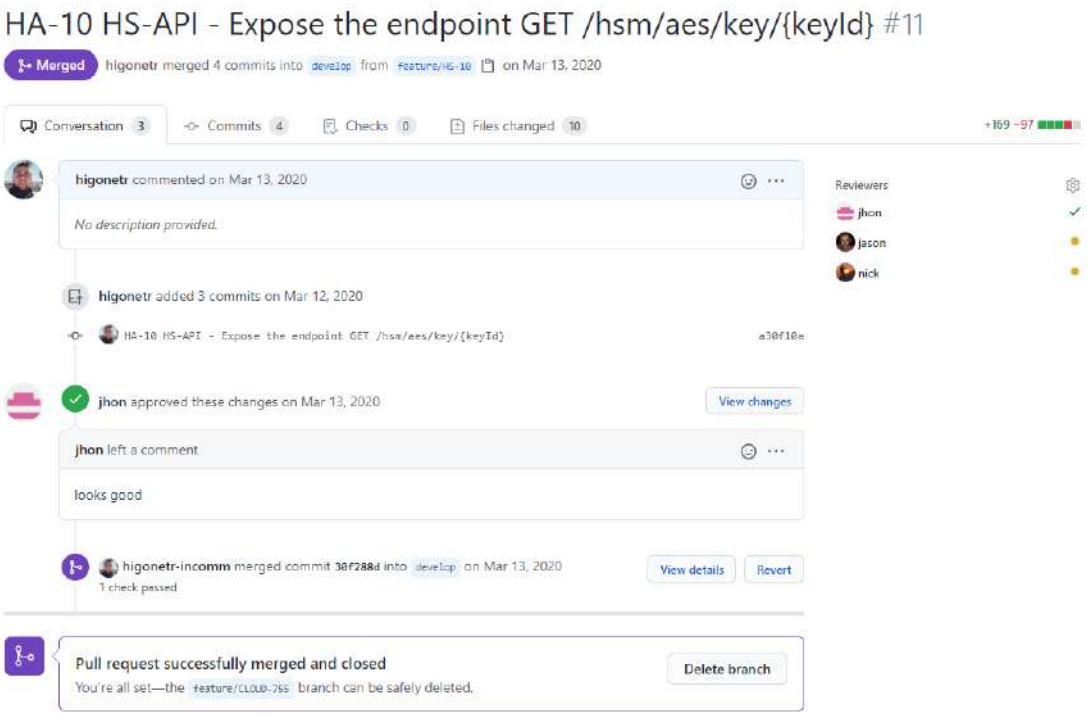

Figura 2.7: Pull request del ticket

TeamCity cada vez que se crea un **pull request**, se hayan desempeñado sin errores. Si se cumplen las condiciones, se pueden combinar los cambios y dar por finalizado el proceso de **pull request**.

5. **La herramienta de CI/CD se encargará de crear el instalable**: automáticamente **TeamCity** detectará que ha ocurrido un cambio en la rama de desarrollo. Como consecuencia se iniciarán dos procesos: el primero se encarga de construir una imagen **Docker** de la aplicación y enviarla al repositorio de imágenes **DockerHub** [21]; el segundo proceso se encargará de construir un `.zip` con los archivos **YAML** [22] que se necesitan para instalar una aplicación a un ambiente de **Kubernetes**. Todo esto será enviado a la herramienta de gestión de lanzamientos, **Octopus Deploy**.

6. **Instalar la nueva versión en el ambiente de pruebas**: una vez que se

tiene la versión instalable en **Octopus Deploy**, se procede a instalar la versión de la aplicación que se acaba de crear. Se puede instalar en el ambiente de pruebas, primero en el ambiente de pruebas del equipo de desarrollo y luego si se detecta que todo se encuentra bien, se instalará en el ambiente de pruebas contra el cliente, para que la característica este disponible para usarse por los usuarios finales.

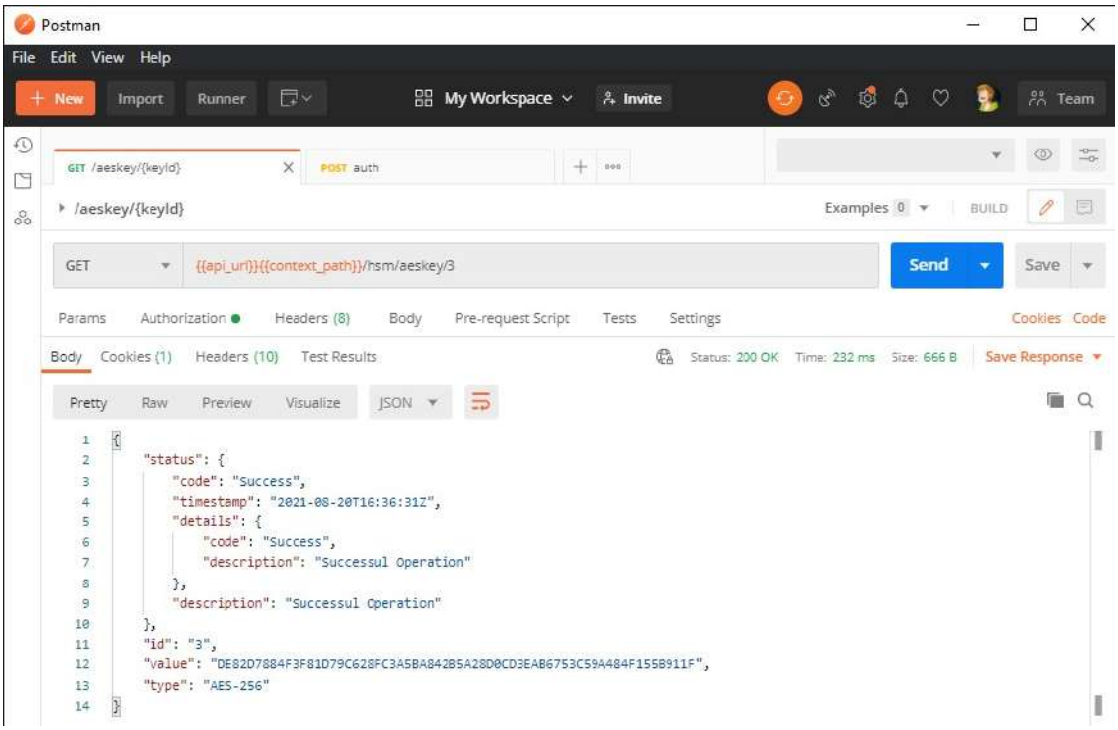

Figura 2.8: Prueba de la operación con Postman

La manera usual y sencilla de probar la funcionalidad agregada en el *ticket* es con una prueba manual hecha usando una herramienta como **Postman** para enviar una petición a la **API**. La Figura 2.8 ejemplifica como se ve una llamada a la aplicación a través de la herramienta. Una vez que el resultado es el esperado se pasa a instalar en el ambiente de pruebas contra el cliente. Sino, se repite el proceso implementando los cambios necesarios en la rama del *ticket*, arreglando las pruebas y creando el **pull request** nuevamente,

hasta validar que se cumplieron los requisitos pedidos en el *ticket*.

Capítulo 3

Aplicación: HS-API

HS-API es la aplicación que se desarrolló y la principal protagonista de este proyecto final. Se puede definir en pocas palabras como un microservicio realizado en **Spring Boot** que expone, a través de una **API**, operaciones criptográficas sobre módulos **HSM**.

Esta aplicación, como ya se explicó anteriormente, surgió ante la necesidad de migrar a un nuevo tipo de llaves criptográficas, debido a una obsolescencia por parte de los estándares de la industria que regulan las transacciones con tarjeta de crédito.

La empresa, en este escenario, vió una oportunidad de mejorar. En lugar de realizar el cambio en cada una de las aplicaciones encargadas de transaccionar las peticiones de los clientes, pensó en centralizar la responsabilidad del manejo de las llaves criptográficas que participaban en el proceso. Como respuesta a esta oportunidad surgió **HS-API** como un microservicio interno en la arquitectura existente de microservicios de la compañía. El mismo debía ofrecer una forma amigable de poder gestionar el ciclo de vida de las llaves criptográficas, y cualquier operación criptográfica pertinente.

Por cuestiones de seguridad informática, el gestor de estas llaves, el que las crea

y el que puede ver su contenido, es un dispositivo de encriptación por hardware, los llamados **HSM** definidos anteriormente.

En este capítulo se explicará cómo se diseñó la aplicación, cómo es su arquitectura, cómo se hizo para conectarla con los módulos **HSM**, cuáles son las operaciones que se pueden realizar con ella, cómo es su base de datos, cómo es el uso de la aplicación y el ciclo de vida de las llaves criptográficas desde la perspectiva de una aplicación que usa **HS-API**, y el trabajo que se tiene preparado para ella en el futuro.

3.1. Funcionalidades

Antes de describir de las funcionalidades que cumple la aplicación y de cuales expone a través de su **API**, es necesario mencionar que los requerimientos fueron brindados por el dueño del producto del equipo. Al momento de recibir una petición para el desarrollo de un nuevo sistema, aplicación o característica, por lo general a los desarrolladores les llega una idea previamente trabajada. Esto no quiere decir que este pensada al 100 % pero sí su idea general, o al menos es lo que se espera. Afortunadamente, los requisitos se expresaron en su totalidad y le permitió al equipo utilizarlos como guía para dividir el trabajo en *tickets* que hicieran posible el desarrollo de **HS-API**, lo cuál se describió en la Sección 2.1.3.

Requisitos funcionales:

- Crear claves criptográficas **AES 256** para un usuario determinado.
- Un usuario debe poder obtener las claves criptográficas **AES 256** que creó.
- Un usuario solo puede acceder a sus claves criptográficas, no a las de otros usuarios.

- Un usuario debe poder obtener las claves criptográficas encriptadas con la **KEK** que posee.
- Las conexiones con los dispositivos **HSM** deben ser reusables y no se debe crear una conexión nueva por cada petición que reciba la aplicación.
- Debe exponer las operaciones en una interfaz estándar **HTTPs API**.

Requisitos no funcionales:

- Los usuarios que hagan peticiones deben estar autenticados.
- La comunicación con los dispositivos **HSM** debe ser veloz, no superar los 500ms.
- La aplicación debe ser escalable.
- Su **API** debe estar documentada usando el estandar **OpenAPI** 3.0 [23].
- La aplicación no debe registrar, en ningún momento, información sensible en sus entradas de *logs*.
- La aplicación debe producir información suficiente en sus entradas de *logs* para poder encontrar problemas en tiempo de ejecución.
- La aplicación debe producir métricas para poder medir su comportamiento en tiempo de ejecución, y así crear alertas con ellas.
- La aplicación debe respetar los estándares de integración continua y despliegue continuo definidos por el equipo de calidad de la empresa.
- La aplicación debe asegurar la calidad de su software con pruebas unitarias y de integración.

- La aplicación debe soportar la condición de carrera sin ningún problema para el cliente.
- Las variables de la aplicación deben poder ser configuradas sin necesidad de tocar el código fuente.
- La aplicación debe poder ser empaquetada en una imagen **Docker**.
- La aplicación debe poder instalarse como un microservicio en un ambiente de **Kubernetes**.

Su construcción como microservicio y los requisitos no funcionales están relacionados. Al ser la aplicación un microservicio, puede escalar horizontalmente y crear tantas instancias de la misma como se necesite. Así mismo, al tenerla en una imagen **Docker**, se la puede ejecutar en un solo comando y en el ambiente que se deseé.

Esta sección se enfocará en los requisitos funcionales, aquellos que hacen a la función de la aplicación, aquellos que especifican los servicios que brinda al usuario.

3.1.1. Activar una KEK para su posterior uso

Ya se habló de lo que significa una clave criptográfica **KEK**, llave de encriptación de llave. En la aplicación para crear llaves, y que éstas se almacenen de una manera segura, es necesario tener previamente una **KEK** activada. Por lo que se crea una operación que permite a un usuario especificar cuál es su clave **KEK**, y de esta manera "activarla" para luego ser usada. La lógica de la operación es muy simple, como muestra la Figura 3.1, un usuario manda el valor de su LMK[KEK], la aplicación la recibe, chequea su integridad contra el dispositivo **HSM**, y la almacena. No hay ningún riesgo en almacenarla porque la misma ya está protegida bajo la **LMK** y solo puede ser usada contra el **HSM**, y debido a que es una **KEK** sus funcionalidades están limitadas a encriptar/desencriptar otras llaves.

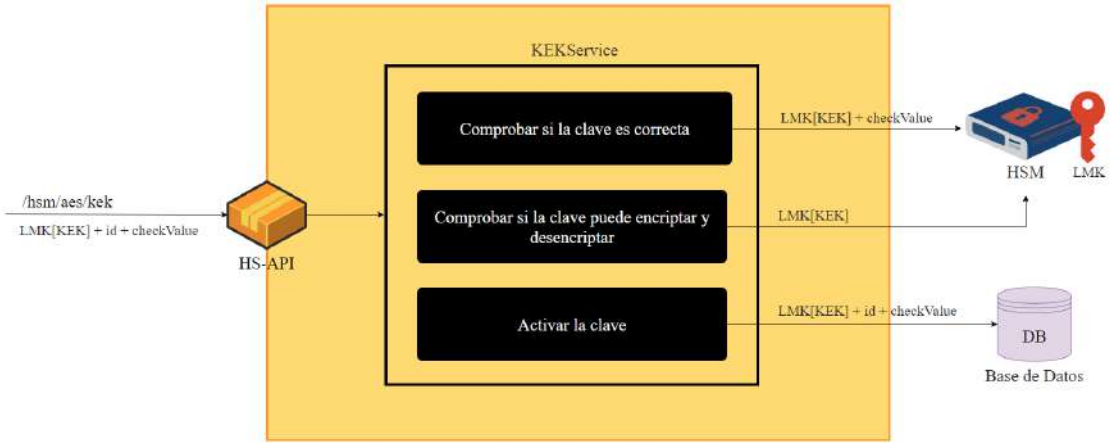

Figura 3.1: Proceso para activar una **KEK**

A continuación se presenta el proceso en más detalle, comenzando con los campos que la operación requiere como entrada:

- `id`: identificador con el cuál se guardará la clave y servirá para referenciarla luego. Al exponer la selección de los IDs al usuario siempre existe el conflicto de colisión, por lo tanto se decide hacer única la combinación usuario más ID. En consecuencia, puede haber claves con el mismo ID siempre y cuando no sean del mismo usuario. Si por algún motivo un usuario envía dos veces la solicitud con el mismo ID, se tomará a la segunda como una actualización.
- `value`: valor de la **KEK**, técnicamente LMK[KEK].
- `checkValue`: valor de comprobación de la clave, el cual sirve para comprobar que el valor de la clave es correcto y no ha sido alterado en tránsito. El concepto de `checkValue` es similar al concepto de *checksum*.

Cuando la aplicación recibe una petición de este tipo es manejada por la clase `KEKService`. Ésta, tiene la funcionalidad de activar las **KEK** y para ello divide su labor en tres pasos cómo se mostró en la Figura 3.1:

- **Comprobar si la clave es correcta**: usando los valores proporcionados `value` y `checkValue` se ejecuta un comando en el dispositivo **HSM** para verificar que el valor de la clave sea correcto.

- **Comprobar si la clave puede encriptar y desencriptar**: para asegurar el no activar claves que luego no puedan realizar las tareas necesarias, es decir, no puedan encriptar ni desencriptar, la aplicación realiza una verificación contra el dispositivo **HSM**, encriptando un valor de prueba y luego desencriptando el resultado, comprobando que se haya obtenido el valor original.

- **Activar la clave**: este es el paso más importante de la operación, donde realmente se "activa" la **KEK**. La activación consiste en almacenarla en la base de datos de **HS-API**. Se almacenan los campos proporcionados por el usuario en la petición (`id`, `value` y `checkValue`) junto con una referencia al usuario, de esta manera se asegura que los usuarios no puedan ver información de otros.

  En este paso del proceso es posible que ocurra el escenario de **condición de carrera**. Por el uso que tendrá la aplicación, que se expondrá más adelante en el Capítulo 4 donde se explica su función en la gestión de llaves **AES**, ocurrirán casos donde de manera simultanea se va a tratar de activar la misma **KEK**, por lo que se debe tener en cuenta que esto puede repercutir al momento de guardarla en la base de datos. ¿Por qué podría pasar esto? Si dos usuarios **A** y **B** en paralelo tratan de guardar una nueva entrada en la base de datos, suponiendo que **A** lo logre hacer antes, a **B** se le negará la petición porque ya existe una entrada en la tabla con el ID seleccionado.

  Como está situación no es posible de evitar, se construye la operación de una manera que si se diera la condición no afecte el resultado final de la operación de activación de **KEK**. Por lo que si por algún motivo al momento de insertar

una nueva entrada en la base de datos con la información de la clave, se obtiene un error porque ésta ya existe, se procede a intentar actualizar la entrada de la base de datos, con el valor `value` y una estampa de tiempo propia de la operación.

En la Figura 3.2 se puede ver cómo es la definición de la operación definida en **OpenAPI 3.0**, los campos que se solicitan en la entrada y un ejemplo de respuesta.

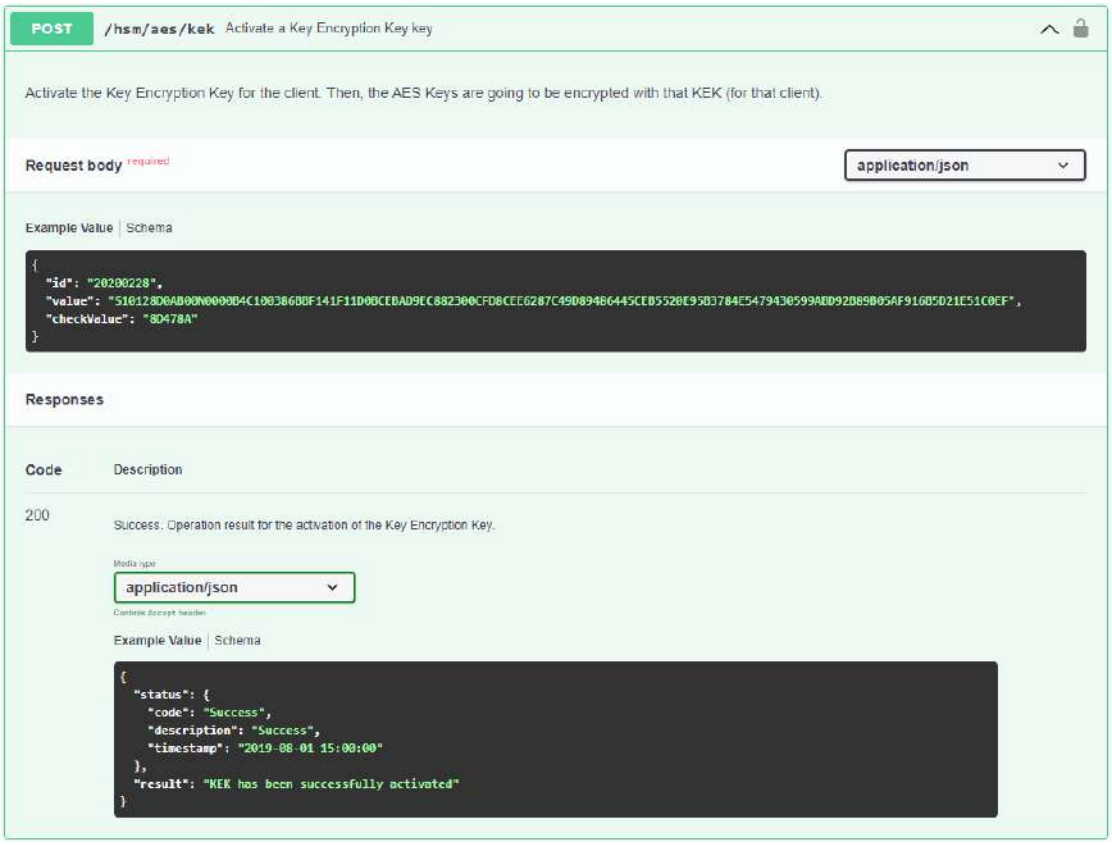

Figura 3.2: Definición de la operación activar una **KEK**

3.1.2. Crear llaves criptográficas AES 256

La aplicación ofrece la funcionalidad de crear llaves **AES 256** a través de su **API**, esto quiere decir que un usuario autorizado puede solicitar crear nuevas llaves para su uso. Como se mencionó anteriormente, un requisito es tener por lo menos una **KEK** activada. Por lo tanto en esta operación los pasos que se siguen son los que presenta la Figura 3.3. Se requiere solo un valor de entrada, `localMasterKey` el cuál hace referencia al identificador unívoco de la **LMK** en los dispositivos **HSM**.

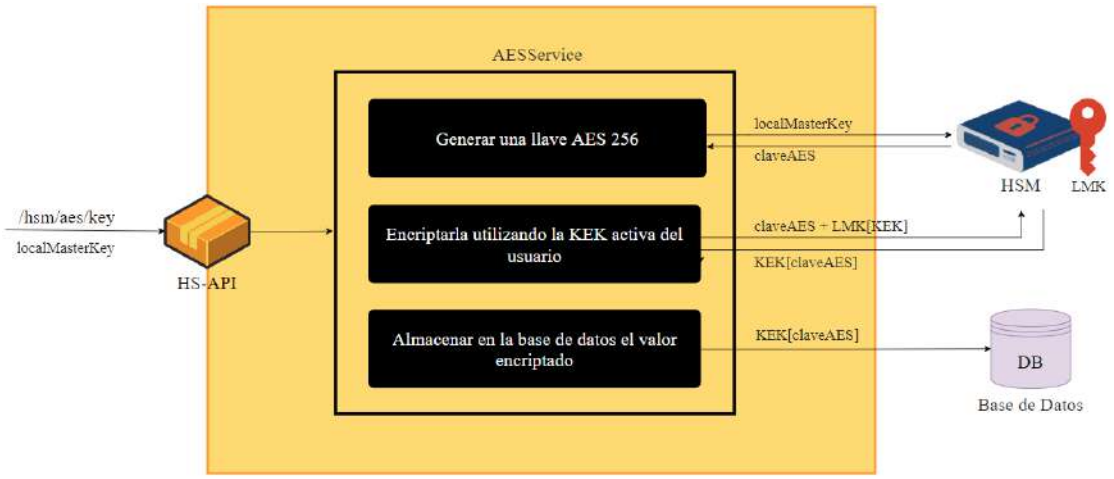

Figura 3.3: Proceso para crear una llave **AES 256**

- **Generar una llave AES 256**: con el valor de `localMasterKey` proporcionado por el usuario, se genera una llave **AES 256** utilizando al dispositivo **HSM** para ello. De esta operación sale como resultado una `claveAES` 256 sin ninguna encriptación.

- **Encriptarla utilizando la KEK activa del usuario**: con el valor de la `claveAES` resultante de la operación anterior y con la clave **KEK** anteriormente configurada por el usuario, la aplicación realiza una encriptación que obtiene como resultante `KEK[claveAES]`, valor de la nueva clave **AES** protegida por la **KEK**. Dicho criptograma se puede guardar en cualquier tipo

de almacenamiento dado a que no se considera sensible y la única forma de poder ver el verdadero valor de la llave es mediante un dispositivo **HSM**.

- **Almacenar en la base de datos el valor encriptado**: la aplicación almacena el valor `KEK[claveAES]` en su base de datos.

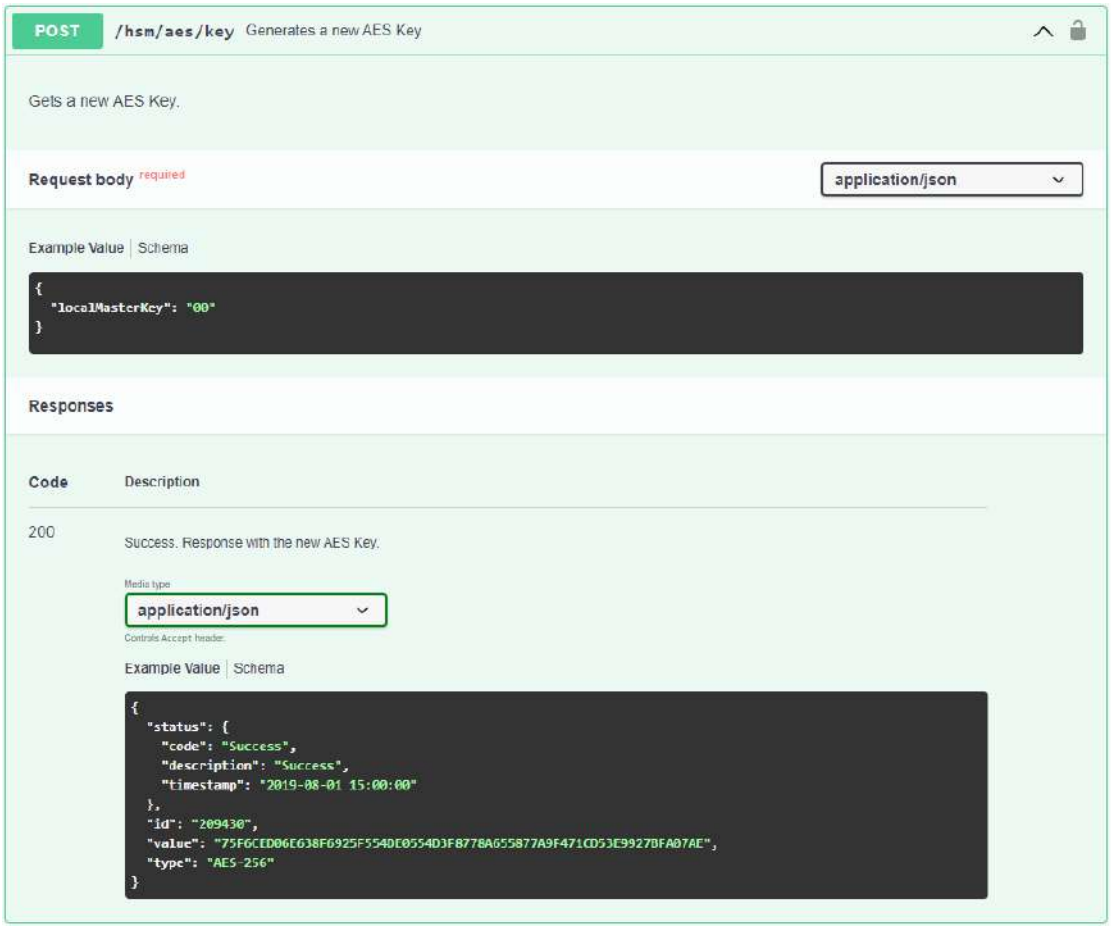

Figura 3.4: Definición de la operación crear una llave **AES 256**

Al usuario que realizó la operación se le devuelve el valor de la clave **AES** sin ningún tipo de encriptación, para que éste pueda utilizarla en sus aplicaciones.

En la Figura 3.4 se muestra cómo es la llamada de la **API**, parámetros necesarios para la consulta y su respuesta.

3.1.3. Obtener una llave criptográfica AES 256

Como típica operación **CRUD** (*Create*, *Read*, *Update*, *Delete*), la aplicación debe soportar la obtención de llaves previamente creadas. Dado un ID de llave en particular, debe ser capaz de devolver el valor de la clave sin ningún tipo de encriptación. Cuando un usuario envía una petición para obtener una llave **AES 256** con un ID específico, la aplicación busca si existe dicha clave para ese usuario, y obtiene la entrada necesaria de la base de datos. Pero como esa entrada contiene el valor KEK[AES] es necesario desencriptarlo utilizando la **KEK** con la que se encriptó el valor, en el dispositivo **HSM**. Para así poder devolverle al usuario el valor de la clave solicitada sin ningún tipo de encriptación.

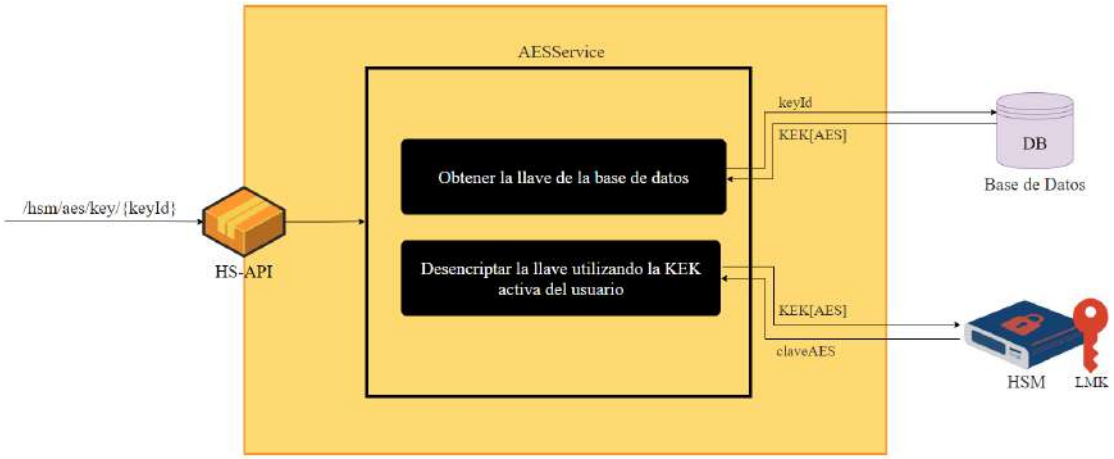

Figura 3.5: Proceso para obtener una llave **AES 256**

En la Figura 3.5 se ve cómo funciona este proceso. Los pasos son:

- **Obtener la llave de la base de datos:** con el valor de `keyId` proporcionado por el usuario, la aplicación busca la entrada de la base de datos que contiene el valor de la clave protegida bajo la **KEK**.
- **Desencriptar la llave utilizando la KEK activa del usuario:** luego, con la **KEK** del usuario y con la clave KEK[AES] se realiza una desencriptación

utilizando el dispositivo **HSM** para obtener el valor de la clave **AES 256** sin ningún tipo de encriptación.

La Figura 3.6 muestra la especificación de la operación en la API, el parámetro de entrada `keyId` y un ejemplo de respuesta.

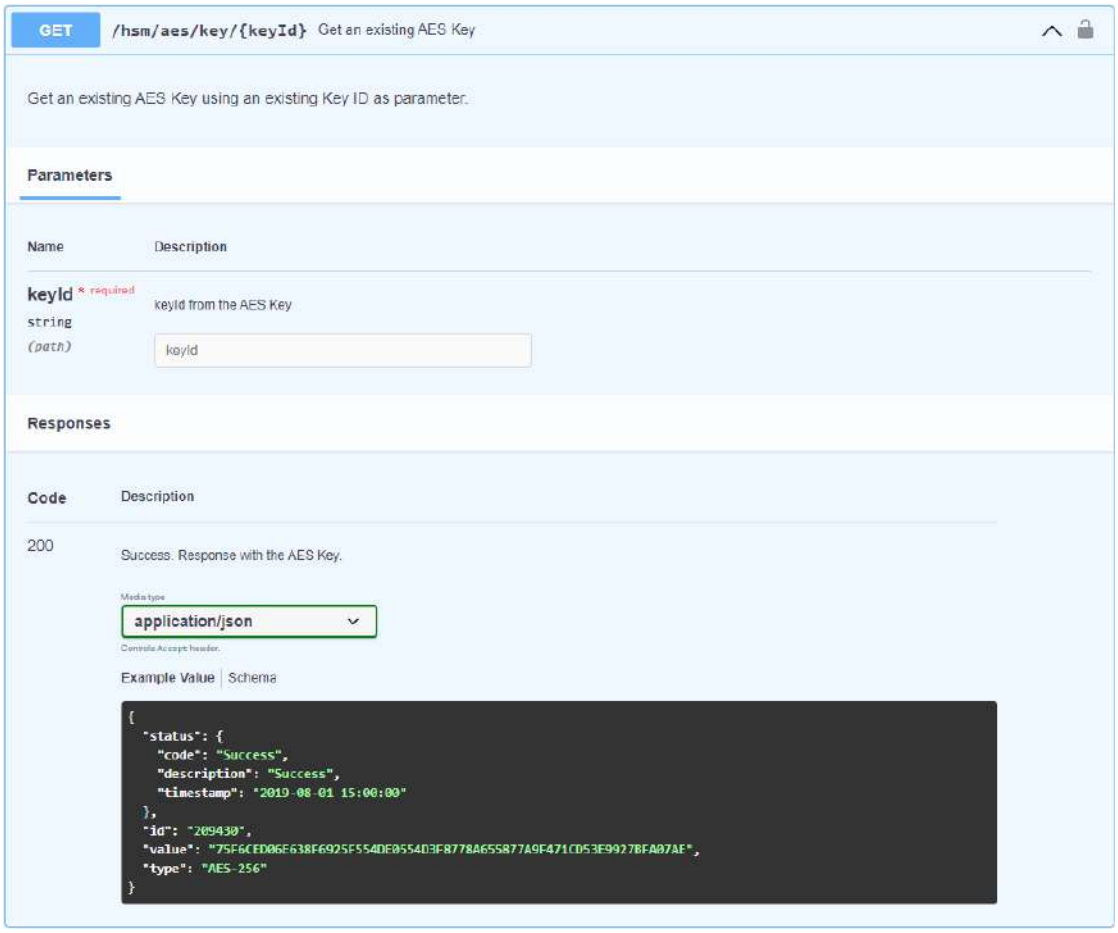

Figura 3.6: Definición de la operación obtener una llave **AES 256**

3.2. Arquitectura

HS-API es una aplicación que pertenece a la arquitectura de microservicios de la compañía, en esta sección se explica como se compone, y se señalan los elementos

que hacen posible que sea alcanzada por los usuarios.

El diagrama de arquitectura de la aplicación es el que se muestra en la Figura 3.7, en ella se evidencia dónde se encuentra la aplicación dentro de esta arquitectura, cuáles son los componentes con los que interactúa y su relación con ellos. A continuación se explicaran brevemente, a excepción del componente Base de Datos el cual se desarrollará en la Sección 3.3.

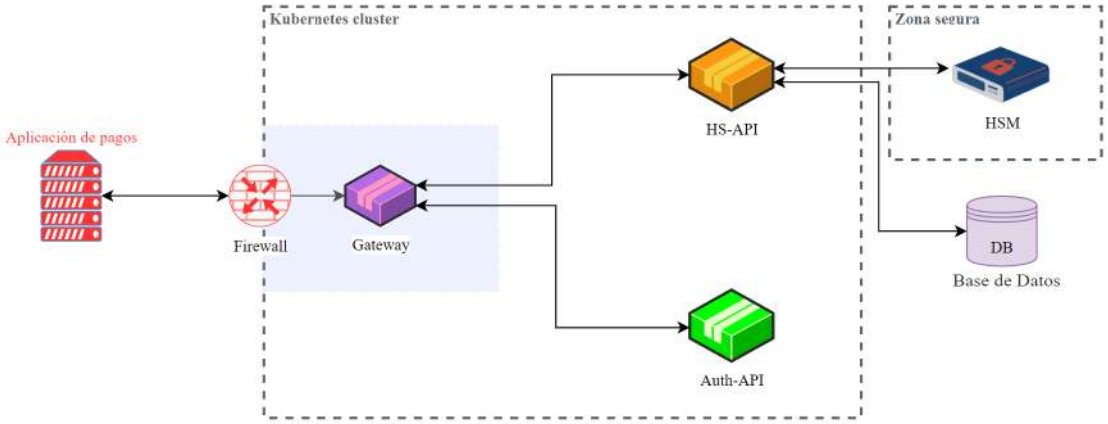

Figura 3.7: Diagrama de arquitectura de la aplicación

- **Aplicación de pagos:** como se mencionó anteriormente los usuarios de la aplicación son otras aplicaciones, las cuales procesan transacciones de pagos electrónicos como tarjetas de crédito, débito, *gift cards*, etc. Estas consumen las operaciones que expone **HS-API** para gestionar sus llaves **AES**, en el Capítulo 4 se explicará en detalle el flujo de la gestión de llaves y su ciclo de vida.

- **Firewall:** antes de entrar al ambiente de **Kubernetes**, se cuenta con un **Firewall** que actúa de intermediario en el tráfico que llegan a las aplicaciones. El mismo tiene la funcionalidad de resolución de **DNS**, es decir, de cruzar las peticiones que llegan al nombre del servicio (**URL**) con la dirección **IP** virtual (**VIP**) asignada al ambiente de microservicios. Además cuenta con

características de seguridad tales como: prevenir ataques mal intencionados como un ataque **DDoS** (*Distributed Denial of Service*), lista blanca de orígenes permitidos, otorgar encriptación **SSL** (*Secure Sockets Layer*) al tráfico, etc.

- **Gateway:** es una de las piezas más importantes de cualquier arquitectura de microservicios, ya que un *gateway* tiene la función de enrutador del tráfico entrante y se asegura que este llegue al microservicio que corresponda. Esto lo hace mediante filtros, normalmente en base a la **URL** y dirige cada una de las peticiones a su destino. Además, es el punto de entrada a los microservicios por lo tanto, es una práctica común delegar la responsabilidad de autenticar a los usuarios en él. Es decir, además de su función de dirigir tráfico, tiene la función de asegurarse de que el tráfico que dirija sea de un usuario autenticado. Notar que se habla de autenticación y no de autorización, la autorización queda a cargo de la aplicación que reciba la petición, aunque también se puede delegar esa tarea en él.

- **Auth-API:** microservicio que cumple el papel clave de ser el servidor de autorización cumpliendo el protocolo **OAuth 2.0**.

  OAuth 2.0 es un protocolo estándar para la autorización de usuarios que mediante la gestión de *tokens* asegura su autorización. No se verá en detalle esta aplicación o este protocolo ya que extiende los alcances del trabajo, aunque sí una breve explicación para comprenderlo mejor. La aplicación **Auth-API** expone una operación donde el usuario, proveyendo usuario y contraseña, obtiene un *token* **JWT** [24] (**JSON** Web *Tokens*) que luego usará en sus peticiones para acceder a las operaciones protegidas. Los microservicios que reciben un *token* junto a una petición, mediante una verificación de firma pueden asegurar la autenticidad del mismo.

- **HSM:** Módulo de Seguridad por Hardware, como ya se explicó en la Sección 1.1.

- **Kubernetes cluster:** ambiente de **Kubernetes** donde viven las aplicaciones. **Kubernetes** (**K8S**) es una plataforma *open source* que permite gestionar contenedores facilitando a los administradores configurar y automatizar el despliegue de las aplicaciones.

- **Zona segura:** Los dispositivos **HSM** se encuentran en otra zona de red, la cual la empresa denomina como "zona segura". Esta es una zona donde el tráfico está altamente restringido, y solo se permiten algunas conexiones, como las conexiones **TCP/IP** entre **HS-API** y los dispositivos **HSM**.

3.2.1. Arquitectura interna de la aplicación

En la sección anterior se puede observar el ambiente y los distintos elementos externos a la aplicación. Pero si se quiere ver cómo es la arquitectura interna de la aplicación, es decir sus distintos componentes y relaciones, se debe hacer zoom en la aplicación. Para ello en el diagrama de la Figura 3.8 se muestra el flujo de las transacciones que entran a la aplicación y cuáles son los componentes que recorre.

1. **AESManagementController**: una clase del tipo controlador en **Spring Boot** tiene la responsabilidad de procesar las solicitudes **HTTP** entrantes, procesarla y otorgarle una respuesta al usuario. La clase **AESManagementController** es un controlador que se encarga de realizar las funciones mencionadas para las peticiones relacionadas con la gestión de llaves **AES**.

2. **AESService**: clase que contiene la lógica necesaria para crear llaves **AES** y obtenerlas. Es una clase servicio de **Spring Boot**, las mismas tienen la intención de contener la lógica del negocio y separarla del controlador.

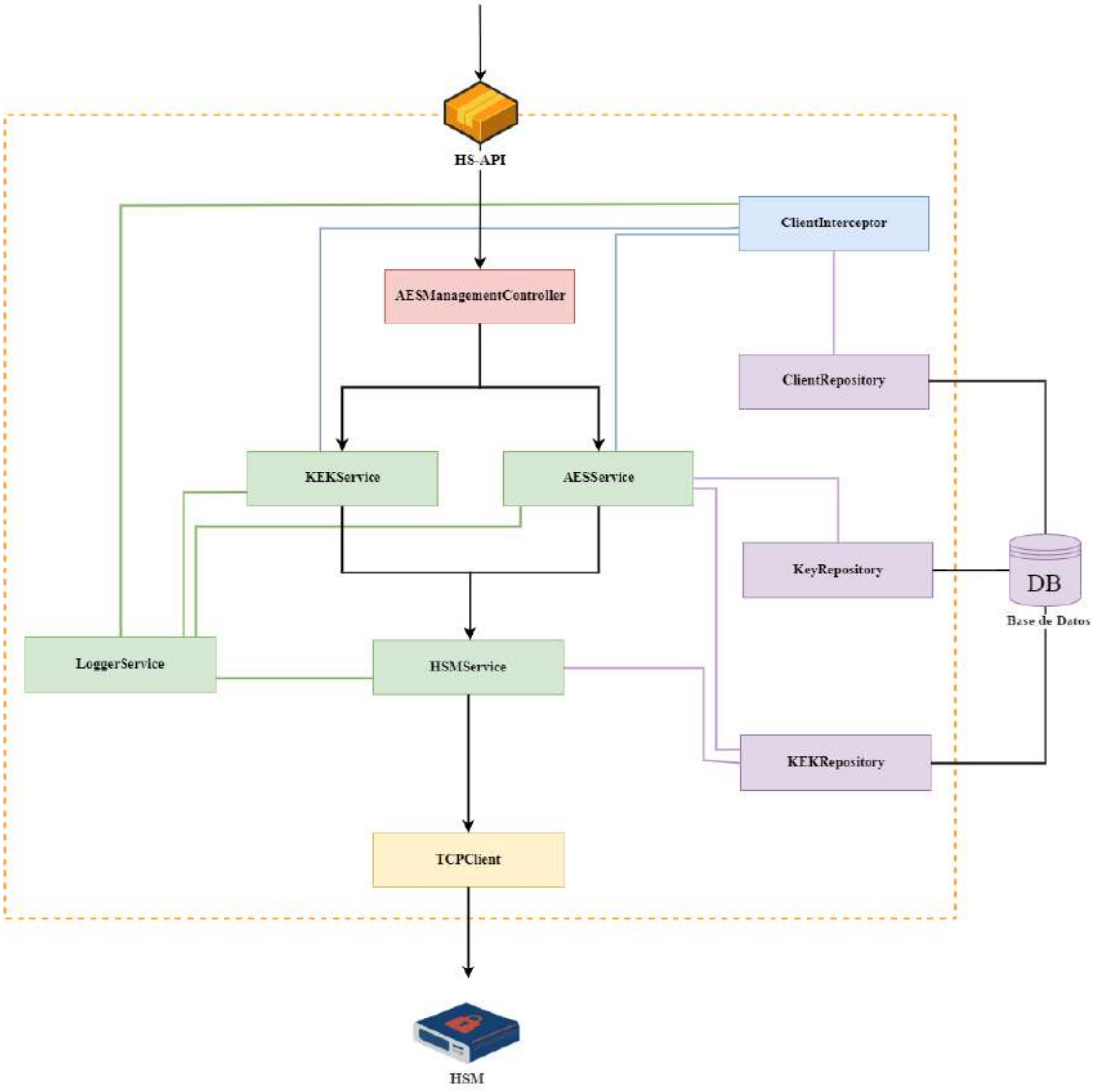

Figura 3.8: Diagrama de la arquitectura interna de la aplicación

3. **KEKService**: clase que posee los métodos requeridos para llevar a cabo la activación de una **KEK**.

4. **HSMService**: clase que centraliza todas las operaciones que se pueden realizar con los módulos **HSM**, es por ello que este servicio es utilizado por las clases **AESService** y **KEKService**.

5. **TCPClient**: esta es la clase que se encarga de gestionar las conexiones **TCP** con los módulos **HSM**, y tiene la responsabilidad de enviar/recibir los mensajes. Se explicará más en detalle todo lo relacionado a las conexiones con los módulos en la Sección 3.4.

En la Figura 3.8 se evidencia cómo los servicios de la aplicación hacen uso de distintos repositorios que se han creado para realizar las consultas pertinentes a la base de datos. En la Sección 3.3 se detallan sobre éstos y la estructura de la base de datos.

El componente **ClientInterceptor** es un componente muy importante de la aplicación. Un interceptor, como su nombre lo indica, tiene la capacidad de interceptar las peticiones del cliente y realizar algo con ello. En este caso, la clase **ClientInteceptor** tiene la responsabilidad de interceptar las solicitudes e identificar si el usuario es válido, y en caso de serlo, inyectar un cliente en el contexto de la petición **HTTP** para que luego pueda ser consumido donde se requiera. Inyectar un cliente, se refiere a obtener de la base de datos la entrada del mismo utilizando su nombre de usuario, asegurarse de que este autorizado a realizar la operación que solicita y guardar en el contexto una instancia del cliente utilizando la entidad `Client` (**Client** es una clase utilizada para representar una entrada de una tabla de base de datos, se detalla más sobre ella en la Sección 3.3.3).

Y por último, el servicio **LoggerService** es una clase que permite generar entradas de registro utilizando un *logger* personalizado en formato **JSON**. El mismo es utilizado por todas las instancias que requieran generar *logs*, es por ello que es requerida por varios de los componentes de la aplicación.

3.2.2. Diagrama de clases de HS-API

Para comprender aún mejor cómo se relacionan los distintos elementos de la aplicación, se presentará un diagrama de clases. Las secciones previas con sus

respectivos diagramas no explican cómo las distintas clases de la aplicación interactúan, por lo que esta sección se centrará en ello.

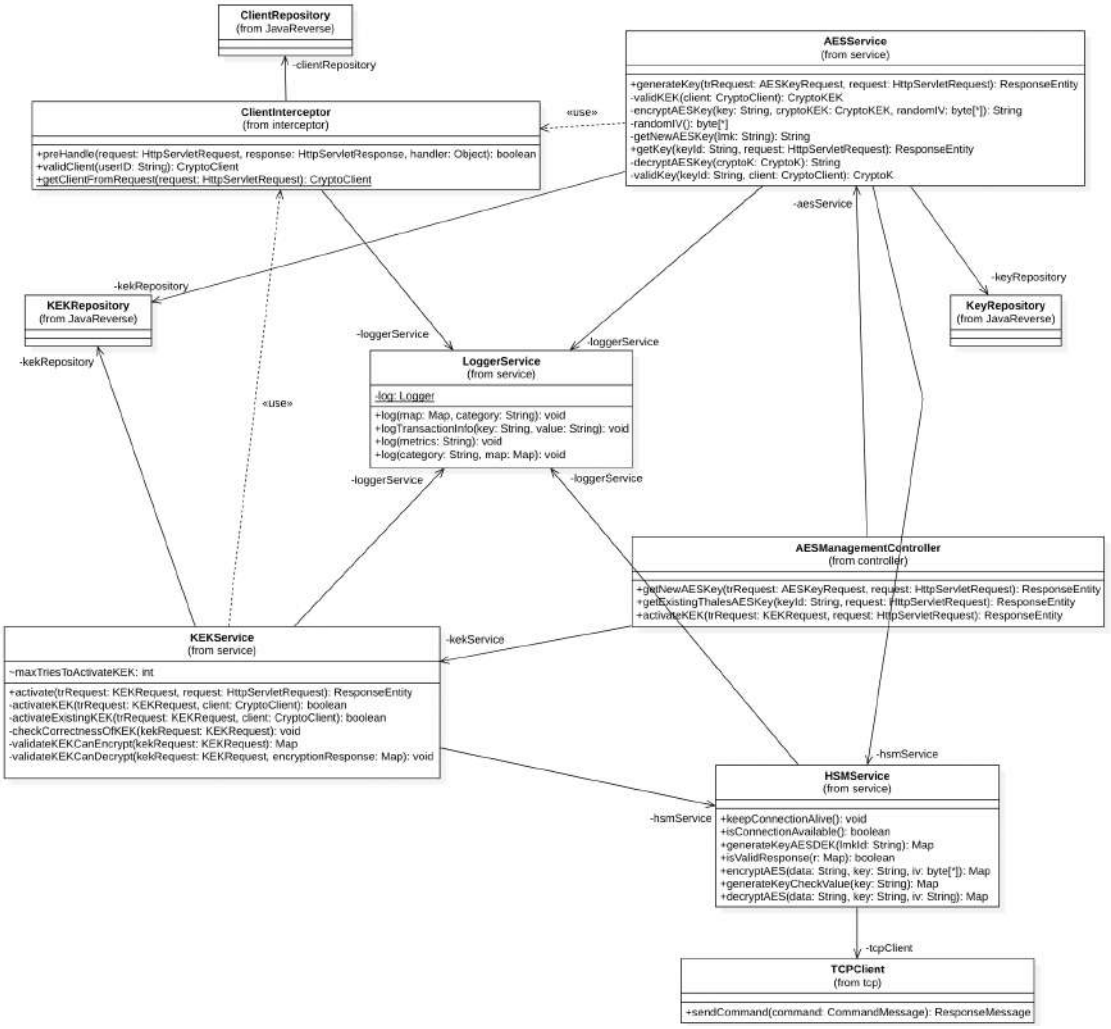

Figura 3.9: Diagrama de la clases de la aplicación

La Figura 3.9 muestra un diagrama de clases **UML** de un fragmento de la aplicación. Por cuestiones obvias, no se muestran todas las clases pero sí se evidencian las más importantes, las mismas que se mencionan en la sección anterior.

Mismas clases que las presentadas en la Figura 3.8 pero distinta perspectiva, aquí la idea es poder observar los distintos métodos de clase y atributos de ellas.

3.3. Base de datos

Se necesitaba una forma de persistir la información de la aplicación de una manera centralizada, para que cada una de las instancias de la aplicación que se ejecutaría en un **pod**, pudiera acceder y escribir los mismos datos. Es por ello que se decidió utilizar una base de datos. La compañía tiene la iniciativa de utilizar a **Microsoft SQL Server** [25] como sistema de gestión de base de datos relacional en los proyectos. Por lo que en el desarrollo de **HS-API** no fue la excepción y se construyó una base de datos **SQL** que se ejecutará sobre este. En esta sección se verá cómo una aplicación **Spring Boot** se conecta a una base de datos, cuáles son las entidades y relaciones que se construyeron, y cómo se realizan las consultas.

3.3.1. Conexión con la aplicación

Para conectar la aplicación con la base de datos se necesitó configurar las siguientes dependencias y propiedades:

- **Dependencias:** para poder conectar la aplicación con el motor de base de datos **Microsoft SQL Server**, se agregaron dos dependencias a la aplicación. Una de ellas ya se introdujo anteriormente en la Sección 1.3, el módulo `starter-data-jpa`, el cual tiene la funcionalidad de unificar y facilitar el acceso a distintos tipos de almacenamiento como base de datos relacionales y base de datos **NoSQL**. La otra dependencia específica del motor de base de datos que se seleccionó. Para poder integrarlas en la aplicación son necesarias las siguientes líneas en el archivo **Gradle**:

```
implementation("org.springframework.boot:spring-boot-starter-
    data-jpa")
runtimeOnly("com.microsoft.sqlserver:mssql-jdbc")
```

Una se agrega con la palabra clave `implementation` y la otra con `runtimeOnly`, porque la librería `spring-boot-starter-data-jpa` necesita ser implementada en la aplicación, ya que se hace uso de las clases que provee. En cambio la librería `mssql-jdbc`, solo se necesita presente al momento de ejecución por sus *drivers*.

- **Propiedades:** cuando se agrega uno de los componentes de **Spring** como dependencia, muchas veces éstos realizan su funcionalidad automáticamente en segundo plano sin que se pueda evidenciar con facilidad. Y en otras oportunidades requieren que se les configuren ciertas propiedades de aplicación para que con los valores proporcionados puedan funcionar de la manera esperada. En este caso se configuró el *driver* de la base de datos, su dirección, usuario y contraseña. El siguiente es un fragmento de las propiedades de la aplicación donde se muestran los valores proporcionados:

```
spring:
  jpa:
    hibernate:
      ddl-auto: validate
      show-sql: true
    properties:
      hibernate:
        format_sql: true
        dialect: org.hibernate.dialect.SQLServerDialect
  datasource:
    driverClassName:
        com.microsoft.sqlserver.jdbc.SQLServerDriver
    url:
        jdbc:sqlserver://192.168.99.100:1433;databaseName=
```

```
        hs_api;sendStringParametersAsUnicode=false;prepareSQL=3;
        maxStatements=2147483647;xaEmulation=false;TDS=8.0;
    username: sa
    password: adminadmin
```

3.3.2. Entidades y relaciones

Las entidades de base de datos creadas fueron tres: una para llevar un control de los usuarios clientes que utilizan la aplicación; una para registrar las **KEKs** que un usuario tiene y conocer cuál es la última activa; y la última para registrar todas las llaves criptográficas que la aplicación haya originado.

El diagrama entidad relación (**DER**) es el diagrama que ilustra las distintas entidades y cómo se relacionan entre sí. El **DER** de la aplicación se puede observar en la Figura 3.10, donde se evidencian las tres entidades, sus campos y relaciones de claves primarias y foráneas.

Por cada entidad del **DER** se generó una tabla definida en lenguaje **SQL**, y a continuación se encuentran los esquemas de cada una:

- **kek**:

```
CREATE TABLE hs_api.dbo.kek (
  id [numeric](19, 0) IDENTITY(1,1) NOT NULL,
  key_id varchar(20) NOT NULL,
  value varchar(MAX) NOT NULL,
  check_value varchar(MAX) NOT NULL,
  activation_time datetime NOT NULL,
  client_id [numeric](19, 0) NOT NULL,
  CONSTRAINT PK_KEK PRIMARY KEY (id),
  CONSTRAINT kek_UN UNIQUE (key_id,client_id),
```

```
  CONSTRAINT fk_kek_crypto_client_id FOREIGN KEY (client_id)
      REFERENCES hs_api.dbo.client(id)
)
```

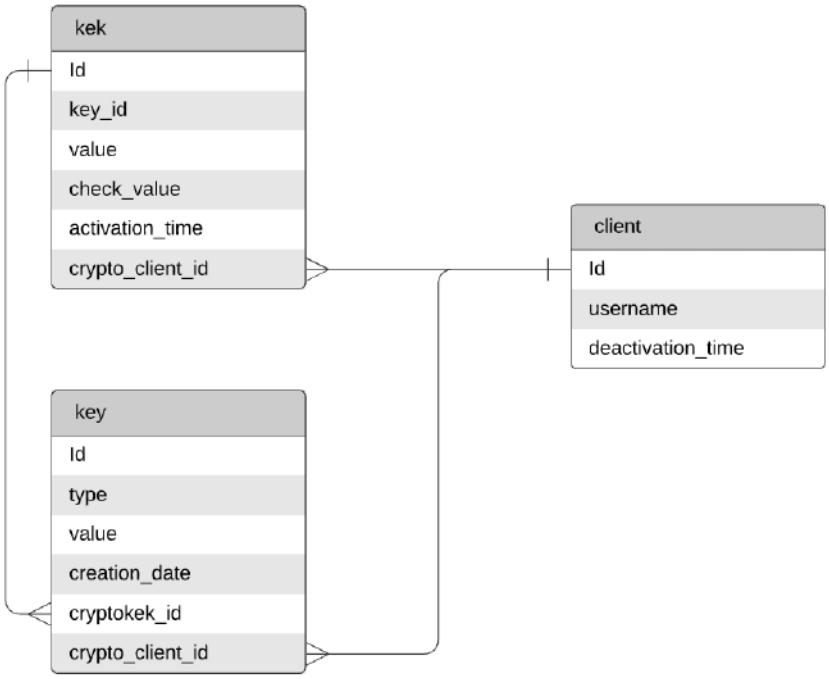

Figura 3.10: Diagrama entidad relación

- **client**:

```
CREATE TABLE hs_api.dbo.client (
  id [numeric](19, 0) IDENTITY(1,1) NOT NULL,
  username varchar(255) NOT NULL,
  deactivation_time datetime NULL,
  CONSTRAINT PK_CLIENT PRIMARY KEY (id),
  CONSTRAINT UC_username UNIQUE (username)
)
```

- **key**:

```
CREATE TABLE hs_api.dbo.key (
  id [numeric](19, 0) IDENTITY(1,1) NOT NULL,
  [type] varchar(20) NOT NULL,
  value varchar(MAX) NOT NULL,
  creation_date datetime NOT NULL,
  kek_id [numeric](19, 0) NOT NULL,
  client_id [numeric](19, 0) NOT NULL,
  CONSTRAINT PK_KEY PRIMARY KEY (id),
  CONSTRAINT fk_key_client_id FOREIGN KEY (client_id)
     REFERENCES hs_api.dbo.client(id),
  CONSTRAINT fk_key_kek_id FOREIGN KEY (kek_id) REFERENCES
     hs_api.dbo.kek(id)
)
```

3.3.3. Consultas desde la aplicación

Para poder realizar consultas y actualizaciones a una base de datos desde una aplicación **Spring Boot**, se deben tener en el código de la aplicación ciertas clases.

Por cada una de las tablas de la base de datos, debe existir una clase que la represente con sus mismos campos. Esta es la clase denominada **entidad** (*entity*) la cual va a representar cada una de las entradas de la tabla, es decir, los registros de la tabla se podrán instanciar usándola. Para poder indicarle a la aplicación que una clase es de tipo **entidad**, cuáles son sus relaciones de clave foránea, su clave primaria y su forma de generación, la librería que se necesita es `spring-boot-starter-data-jpa`, la cuál provee de anotaciones para facilitar su implementación, como se muestra en el ejemplo:

```java
@Entity
@Table(name = "client")
public class Client {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @NotNull
    @Column(name = "username", nullable = false)
    private String username;

    @Column(name = "deactivation_time")
    private ZonedDateTime deactivationTime;

    @OneToMany(mappedBy = "cryptoClient")
    private Set<CryptoKEK> keks = new HashSet<>();

    @OneToMany(mappedBy = "cryptoClient")
    private Set<CryptoK> keys = new HashSet<>();

    public CryptoClient() {
    }
}
```

Luego, para poder realizar las inserciones a la tabla, realizar una modificación de un registro o una consulta, se debe crear una clase especial que nos brinde esta funcionalidad. **Spring** tiene su utilidad de **ORM** (*Object–relational mapping*)

en las clases llamadas **repositorio** (*repository*). Al configurar una clase como **repositorio** se especifica la entidad sobre la cual va a operar y automáticamente, gracias a operaciones pre-definidas que otorga el *framework*, se obtienen métodos como: guardar un nuevo registro, buscar por clave primaria, etc. Se puede ver un ejemplo de **repositorio** en el siguiente fragmento de código:

```
@Repository
public interface ClientRepository extends JpaRepository<Client,
    Long> {
    CryptoClient findByUsernameAndDeactivationTimeNull(String
        username);
}
```

Por lo general, esta clase no va a contener ningún método en su definición, porque las operaciones pre-definidas son más que suficientes para cubrir la mayoría de los casos. Pero si por algún motivo en particular se necesita hacer una consulta sobre los campos de la entidad, por ejemplo buscar por `username` y además que no contenga ningún valor de `deactivationTime`, se puede definir un método como se ve en el ejemplo anterior, que al momento de ejecutarse se va a traducir en la siguiente consulta **SQL**:

```
SELECT
    client.id as id,
    client.deactivation_time as deactivation_time,
    client.username as username
FROM
    client
WHERE
```

```
client.username = client_username
AND (client.deactivation_time is null)
```

3.4. Conexiones TCP

La cantidad de conexiones **TCP** que permiten los módulos **HSM** es limitada por licencia, es decir que si se requiere una mayor cantidad de conexiones hay que adquirir una licencia que lo permita, y por supuesto requiere más dinero. Cuando surgió la necesidad de migrar de la encriptación con llaves **3DES** a **AES**, las plataformas de pagos que iban a recibir este impacto eran seis, cada una con seis instancias en producción. Por lo que automáticamente se supo que no iba a ser posible conectar cada una de las instancias a los módulos **HSM** para realizar la gestión **AES**. **HS-API** es el centralizador de conexiones con los módulos y por ello uno de sus requisitos funcionales es generar conexiones permanentes y reusables con los mismos.

TCP es uno de los principales protocolos de Internet el cual permite generar conexiones punto a punto y transmitir datos asegurando que se transmitan sin errores y en el orden correcto.

Implementar esta funcionalidad fue la parte más demandante del proyecto, ya que la aplicación está escrita en un *framework* moderno como **Spring**. Hoy en día las comunicaciones entre los servicios se da por medio de interfaces modernas como **APIs HTTP**, donde **Spring** resuelve la comunicación entre éstas de una manera fácil y accesible, que gracias a su popularidad está bien implementada y probada. En cambio, cuando mencionamos conexiones entre una aplicación **Spring** y un servidor **TCP** la situación cambia.

Por suerte el equipo de **Spring** elaboró una librería [26] para trabajar con conexiones **TCP** aunque su uso no fuera tan amigable, sirvió para resolver el problema planteado. Para hacer uso de ella, al igual que las librerías anteriores, bastó con

incluirla a lista de dependencias, con la siguiente línea:

```
implementation("org.springframework.integration:spring-integration-
    ip")
```

Antes de entrar más en detalle acerca de cómo se implementaron las conexiones entre la aplicación **HS-API** y los módulos **HSM**, es necesario entender el escenario inicial. Como presenta la Figura 3.11, cada uno de los **pods** de **HS-API**, es decir, cada una de sus instancias, debía poder usar los dos dispositivos **HSM** disponibles.

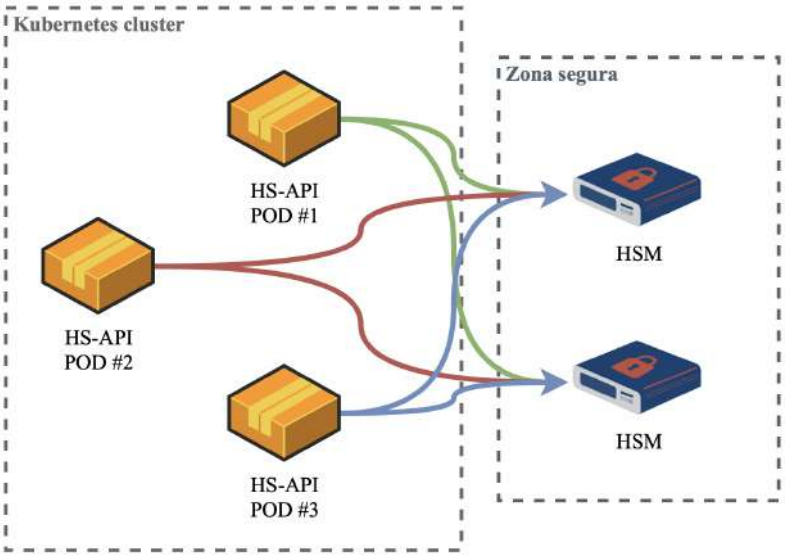

Figura 3.11: Conexiones **TCP**: escenario a resolver

Por ello se decidió que la mejor manera de gestionar las conexiones era implementándolas de manera tal que la aplicación tuviera la capacidad de hacer **round-robin** entre los dos dispositivos **HSM**, en caso de que la comunicación fallara, que pudiera hacer **fail-over** hacia el dispositivo contrario.

3.4.1. Implementación

Para implementar las conexiones utilizando la dependencia de **Spring**, se generó una clase de configuración en el código que permite establecer el cliente **TCP** a utilizar en la aplicación. Ésta se creó siguiendo las recomendaciones y buenas prácticas de la documentación oficial de **Spring** [26]. La Figura 3.12 es un diagrama de lo que se implementó, lo cuál permitió que la aplicación tenga las siguientes características: conexiones permanentes, conexiones reusables, **roun-robin** y **fail-over**.

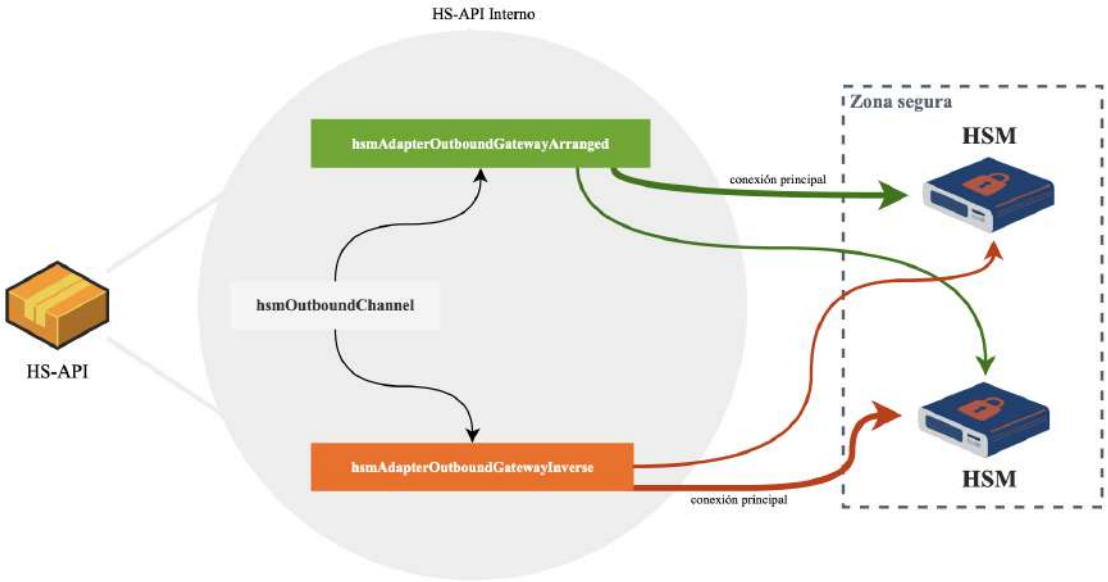

Figura 3.12: Diagrama de las conexiones **TCP**

Cuando se necesita operar con los módulos **HSM**, la aplicación, bajo las capacidades de la clase `TCPClient`, ejecuta una operación con `bytes[]` como parámetros. Ésta es recibida y procesada por `hsmOutboundChannel`, el canal de salida de las conexiones **TCP**. Se define como el canal de salida de los mensajes, que se encarga de separar a los productores de mensajes de los consumidores. En `hsmOutboundChannel` se configuraron dos posibles *gateways* de salida entre los cuales se realiza **round-robin**. En palabras simples, **round-robin** quiere decir que se van intercalando las

salidas, cuando llega una petición utiliza la salida **A**, cuando llega otra utiliza la salida **B**, al llegar otra utiliza la salida **A**, y así sucesivamente.

Los gateways de salida son `hsmAdapterOutboundGatewayArranged` y `hsmAdapter-OutboundGatewayInverse`, los mismos son *gateways* definidos por la librería de **Spring** y tienen la particularidad de realizar **fail-over** con las conexiones configuradas. Esto quiere decir que utilizan la conexión A siempre y cuando funcione bien pero, al momento de que esta falle al enviar o recibir un mensaje, se pasará a usar la siguiente conexión de la lista, y se utilizará esta hasta que falle y utilice la siguiente conexión de la lista. Ambos *gateways* de salida tienen la misma configuración, con la particularidad que la lista de conexiones de los mismos está invertida. En el primero tenemos la lista [**A**, **B**] y en el segundo tenemos la lista [**B**, **A**]. Por lo tanto de esta manera se asegura que cuando lleguen transacciones a **HS-API** y no se encuentren fallas en las conexiones ocurra la siguiente secuencia de uso en las conexiones: **G1-A**, **G2-B**, **G1-A**, **G2-B**. Siempre y cuando no ocurra un fallo en alguna de las conexiones, pues esta secuencia se puede alterar por las condiciones de **fail-over**.

Un punto interesante que ocurrió en el desarrollo de estás características, fue que al ser una librería poco utilizada no se encontraron buenas referencias a ejemplos de implementaciones similares. Por lo cual, se dio una conversación por la página de **Stack Overflow** con uno de los creadores de la librería de **Spring**. Las preguntas y respuestas que surgieron se pueden visitar aquí[1]. De lo que se destaca la recomendación de utilizar dos instancias de **fail-over** con los dispositivos **HSM** y unirlas utilizando un canal de salida, y que este brinde la función de **round-robin**.

[1]https://stackoverflow.com/questions/63195590/spring-integration-two-permanents-connections-with-roundrobin-and-failover

Capítulo 4

Gestión de llaves AES

La gestión de llaves criptográficas administra el ciclo de vida de las llaves, esto implica desde cómo se crean, usan, almacenan, acceden, eliminan y archivan. Básicamente todo lo que le puede ocurrir a una llave criptográfica. Es esencial la seguridad en la gestión de las llaves criptográficas, la misma no debe tener grietas de seguridad, y su acceso y almacenamiento deben estar controlados. En este proyecto la gestión se dio sobre claves criptográficas de tipo **AES**, y en este capítulo se explicará cómo funciona desde un punto de vista práctico desde la perspectiva del cliente de **HS-API**.

La gestión de llaves **AES** tiene como protagonistas a las aplicaciones de pagos que utilizan **HS-API**. En el caso estudiado, el ciclo de vida de las llaves tiene tres pasos los cuales contrastan con las operaciones que expone **HS-API**, ellos son: la activación de la **KEK** para la protección de las claves **AES** futuras, la creación de nuevas llaves **AES** y la obtención de una clave **AES** existente. A continuación se verán con más detalle y de la vista de los clientes que consumen la **API**.

4.1. Activación de la KEK para la protección de claves futuras

Cada uno de los nodos de las aplicaciones de pagos al momento de iniciación van a dar comienzo al proceso de activación de la clave **KEK** que tengan configurada. Como se mencionó anteriormente en la Sección 3.1.1, al estar en esta situación donde múltiples instancias intentan realizar la misma operación al mismo tiempo, se puede producir una condición de carrera. La Figura 4.1 muestra el proceso desde la perspectiva del cliente y debajo se desarrollarán cada uno de los pasos:

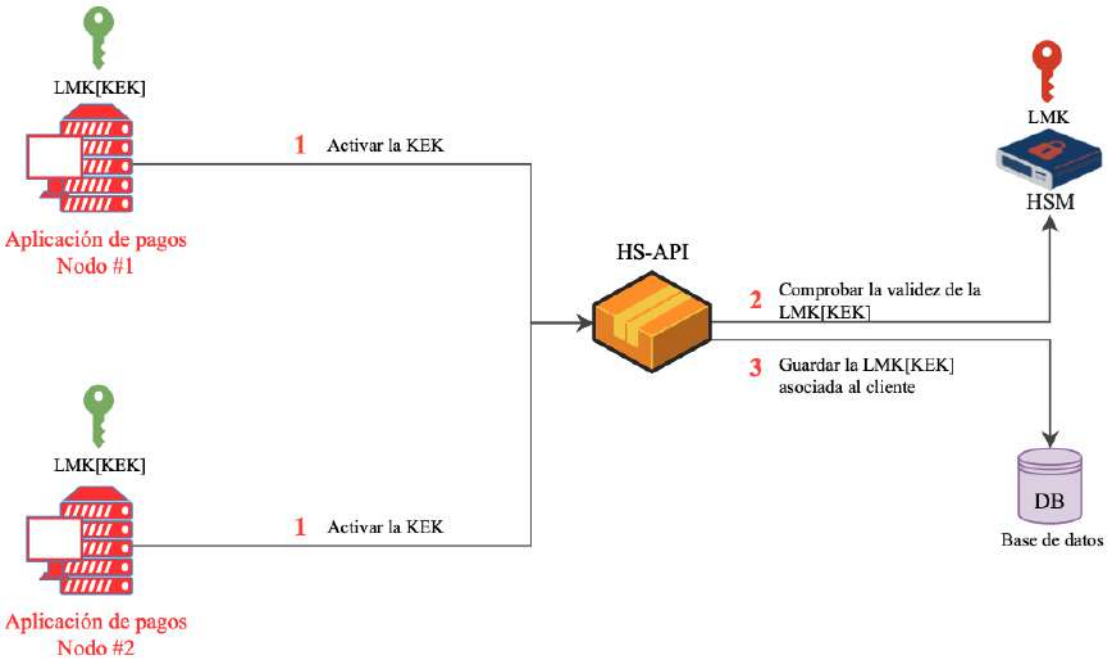

Figura 4.1: Activación de la **KEK** desde la plataforma de pagos

1. **Activar la KEK:** una instancia de la aplicación de pagos envía una petición **HTTP** `/hsm/aes/kek` enviando el valor de la clave **KEK** protegida bajo la **LMK** que tiene configurada como su clave de protección actual.

2. **Comprobar la validez de la LMK[KEK]**: la aplicación **HS-API** recibe esta petición y comprueba contra los módulos **HSM** que la LMK[KEK] pasada por parámetro sea válida.

3. **Guardar la LMK[KEK] asociada al cliente:** como último paso, **HS-API** almacena la clave protegida LMK[KEK] en su base de datos con una referencia al cliente que realizó la petición.

En el paso 3 se ve cómo funciona el almacenamiento de la **KEK** en la base de datos de **HS-API** cumpliendo con los requisitos de auditoría impuestos por **PCI**. La clave almacenada está protegida bajo la **LMK** de los módulos **HSM** y solo es posible ver su valor mediante la interacción con estos, por lo tanto no se compromete el valor de las claves.

4.2. Creación de nuevas llaves AES

Todos los días cada uno de los nodos de las aplicaciones de pagos va a generar una nueva clave **AES 256** con la que va a encriptar la información sensible de sus transacciones. Esta nueva clave que genera la mantiene en memoria por 24 horas y luego pasado este tiempo, genera una nueva que reemplaza la anterior. La Figura 4.2 presenta los pasos que se necesitan para este proceso.

1. **Crear una nueva llave AES:** una instancia de la aplicación de pagos envía una petición **HTTP** `/hsm/aes/key` para obtener una nueva llave **AES**.

2. **Buscar el cliente y la LMK[KEK]: HS-API** recibe la petición y busca en su base de datos al cliente que solicitó la petición y la **KEK** que tiene activada.

3. **Generar una nueva llave AES: HS-API** utiliza los módulos **HSM** para generar el valor de una nueva clave **AES**.

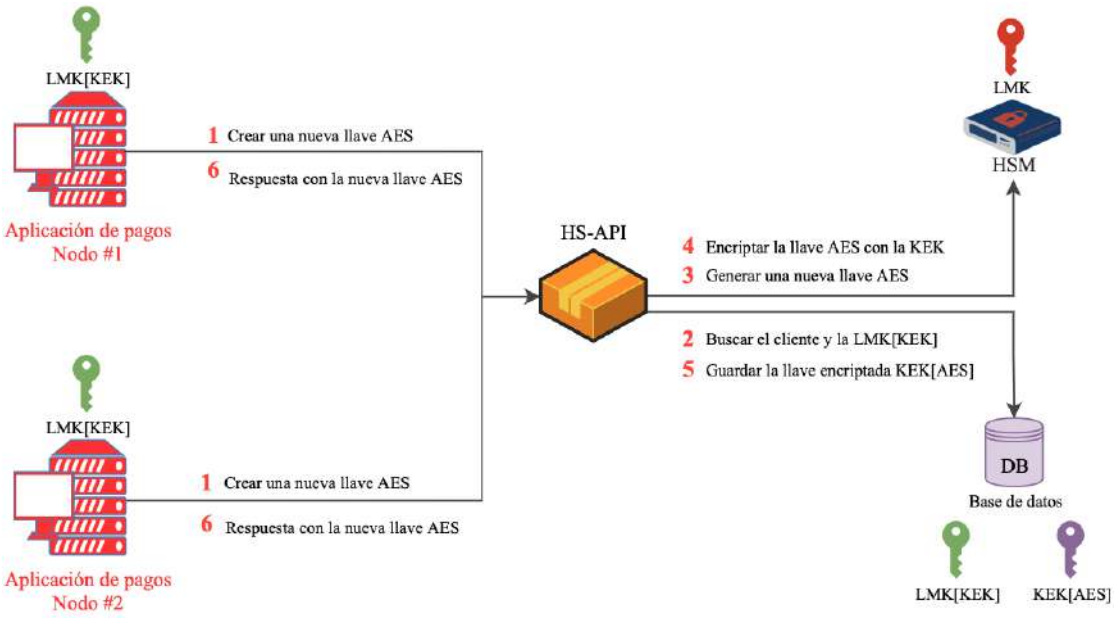

Figura 4.2: Creación de una nueva llave **AES** desde la plataforma de pagos

4. **Encriptar la llave AES con la KEK:** la aplicación encripta, con la **KEK** que el cliente tiene activada, el valor de la clave **AES** recién generada para obtener como resultado el valor de la clave **AES** protegido bajo la KEK: KEK[AES].

5. **Guardar la llave encriptada KEK[AES]:** ahora que **HS-API** tiene la clave **AES** protegida bajo la **KEK** del cliente, es seguro almacenarla en la base de datos. Por lo que la almacena junto a una referencia al cliente y a la clave **KEK** usada.

6. **Respuesta con la nueva llave AES:** a la aplicación de pagos se le devuelve el valor de llave **AES** sin ningún tipo de encriptación, ya que luego ésta necesita hacer uso de la misma para realizar encriptaciones.

En este proceso se puede observar la creación de las llaves **AES** y el almacenamiento de éstas, tanto en la memoria de la instancia de la aplicación de pagos,

como en la base de datos de **HS-API**. Nuevamente se cumplen los requisitos de seguridad, la clave en el almacenamiento persistente está protegida bajo la **KEK** y por lo tanto, la única manera de obtener su valor es con la misma clave que la encriptó, y haciendo uso de los módulos **HSM** para poder realizar la desencriptación. Por otro lado, la llave **AES** está siendo almacenada por las instancias de las aplicaciones de pagos en memoria, sin embargo es sobre memoria volátil y solo por 24 horas, hasta que se genere nuevamente otra clave **AES**. Además de que las distintas instancias de la aplicación van a tener en memoria una llave **AES** diferente.

4.3. Obtención de una clave AES existente

Cuando las aplicaciones de pagos reciben una petición que involucra ver los detalles de una transacción pasada, es necesario poder desencriptar los datos de la misma. Para ello se debe obtener la clave **AES** que se utilizó para realizar la encriptación. Cuando esto ocurre, la aplicación de pagos solicita a **HS-API** por la clave **AES**. La Figura 4.3 describe el proceso, que consta de los siguientes pasos:

1. **Obtener una llave AES existente:** una instancia de la aplicación de pagos envía una petición **HTTP** `/hsm/aes/key/{keyId}` para obtener una nueva llave **AES**.

2. **Obtener el Cliente, LMK[KEK] y KEK[AES]: HS-API** recibe la petición y busca en la base de datos al cliente, la **KEK** y la clave **AES**.

3. **Desencriptar la KEK[AES]: HS-API** desencripta la clave **AES** protegida con la **KEK** del cliente utilizando los módulos **HSM**. De esta manera, obtiene el valor de la llave **AES** sin protección.

4. **Devolver la clave AES:** la aplicación de pagos recibe la clave **AES** sin protección para poder usarla desde su lado.

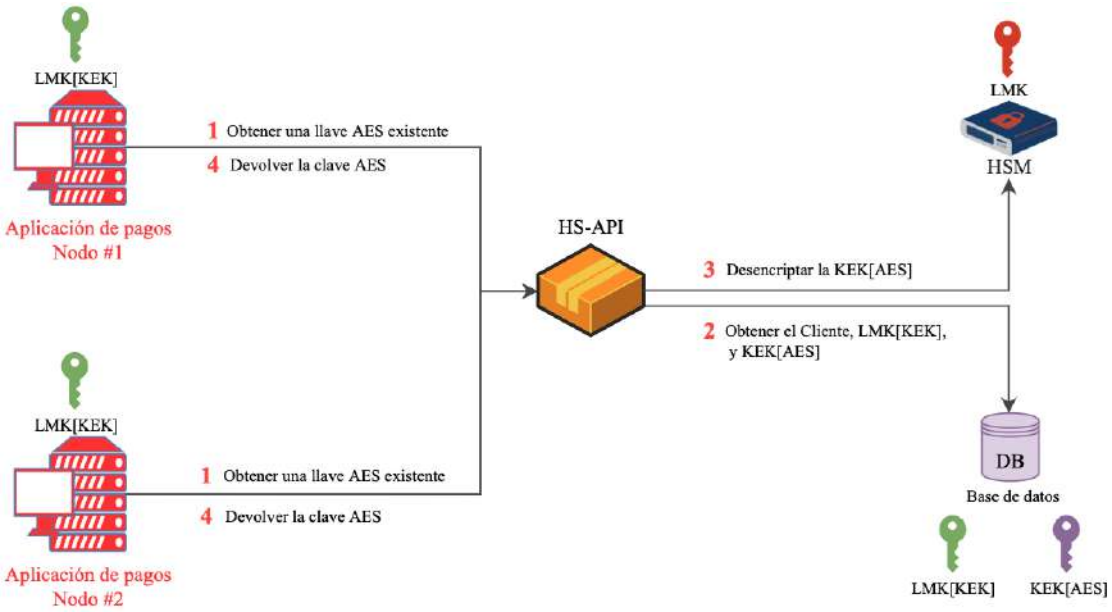

Figura 4.3: Obtención de una llave **AES** existente desde la plataforma de pagos

Este proceso contempla los accesos a claves **AES** existentes, los cuales están dentro de los requisitos de seguridad necesarios. La lectura de las claves **AES** solo está permitida para el mismo usuario que la generó, es por eso que **HS-API** guarda la referencia al cliente junto con las llaves. Por lo que solo se permiten accesos autorizados. Y la llave obtenida, desde la perspectiva de la aplicación de pagos, es utilizada para poder desencriptar los datos de la transacción a la que se necesita acceder y luego descartada, no queda almacenada en memoria. Otro punto fuerte de seguridad es que para poder obtener el valor de la clave es necesario tener la **KEK** y poder operar con los módulos **HSM**.

Capítulo 5

Conclusión

El proyecto realizado alcanzó a cumplir los objetivos propuestos y ahora las aplicaciones de pagos de la compañía no tienen que preocuparse por la obsolescencia del algoritmo **3DES**, ya que **HS-API** resuelve el ciclo de vida de las llaves **AES** por ellas. Sólo es cuestión de que estos realicen su implementación para comunicarse a través de la **API** que expone y ejecutar las operaciones siguiendo los procedimientos que se establecieron.

Luego de haber llevado a cabo el desarrollo de **HS-API** se puede concluir que la Facultad no te prepara específicamente para implementar un microservicio **Spring Boot** que gestione el ciclo de vida de llaves **AES** y que mediante conexiones **TCP** se comunique con módulos **HSM**. Pero sí que la Facultad te brinda todas las herramientas necesarias para poder aprender los conceptos necesarios y realizar las implementaciones. Lo importante es entender los conceptos básicos que hay por detrás de las tecnologías específicas.

- En el transcurso de la carrera uno aprende **JAVA** como lenguaje fuerte, y hoy en día es uno de los lenguajes más demandados. Por suerte el proyecto se implementó en ese mismo lenguaje. Y si bien no se aprendió ningún *framework* de desarrollo, la ausencia de este hizo que a lo largo de la carrera,

implementar sin ayuda de un marco de trabajo lograra un mayor entendimiento de las dependencias e interrelaciones entre los distintos componentes de una aplicación. Y gracias a esto, aprender un *framework* como **Spring Boot** se vuelve una tarea sencilla.

- La teoría de Ingeniería de Software incorporada ayudó a adaptarse a un marco de trabajo como **Scrum** y que este no presentara ningún tipo de dificultad. Los conceptos de historias de usuarios y tareas fueron muy útiles y permitieron aportar valor a la gestión de los **Sprints**.

- Los conceptos de Base de Datos aprendidos fueron más que suficientes para resolver los problemas planteados.

- Si bien el tema de microservicios no es algo que se aprende en la carrera, la teoría de Sistemas Distribuidos contribuyó en gran medida al entendimiento de conceptos claves como, que la misma aplicación corre en simultaneo en varias instancias y se debe asegurar no producir conflictos en operaciones coordinadas. Además de que las arquitecturas de microservicios proponen la comunicación entre los distintos microservicios y cómo la sinergia entre éstos sirve a un propósito en común.

- Implementar las conexiones **TCP** fue la tarea más complicada del proyecto, debido a las características esperadas de **round-robin** y **fail-over**. Aunque en la Facultad no se hayan creado conexiones hacia una interfaz **TCP/IP**, sí se adquirieron los conocimientos necesarios para entender cómo funcionan, lo cual fue de gran utilidad al momento de efectuar la tarea.

Para finalizar las conclusiones, es importante mencionar que hay mucho contenido que ha sido omitido en este Trabajo Final pero que ha sido muy importante. Muchos conceptos de microservicios, como por ejemplo dar a entender todo el

ambiente de **Kubernetes** en el cual se ejecuta la aplicación y las distintas características que la arquitectura de microservicios brinda, es algo de gran valor pero no se ha incluido en el informe con el fin de enfocarse en las funcionalidades de **HS-API** que aportan más valor al negocio. A continuación se verá cuál es el futuro que se espera del proyecto.

5.1. Futuro del proyecto

El proyecto ya tiene algunas características planeadas para su futuro. Una vez finalizado, se esperó a la implementación de los usuarios de **HS-API**, las aplicaciones de pagos, las cuales pudieron resolver su gestión de llaves **AES** sin ningún tipo de inconvenientes. Al tener una retroalimentación positiva, por parte de los usuarios, el equipo pudo evidenciar que el camino elegido era el correcto y se empezaron a trabajar nuevas características en la aplicación que sirvieran para ofrecer soluciones a distintos casos de uso sobre los módulos **HSM**. La siguiente es una lista de los posibles casos de uso en los que la aplicación **HS-API** participaría y con lo cual se daría lugar a mejoras y nuevas implementaciones en ella.

- Desencriptación con el algoritmo **3DES DUKPT**[1]: uno de los proyectos requiere una solución para desencriptar datos de tarjetas de los clientes, datos cuyo algoritmo y esquema de encriptación es **3DES DUKPT**. Que además por cuestiones de seguridad, dichas llaves de encriptación deben generarse por módulos **HSM**, protegidas por una **LMK** y una **KEK**. La idea en este proyecto, es que **HS-API** pueda almacenar las claves criptográficas de una manera segura, y además exponga bajo una operación **HTTP** la desencriptación de los datos.

[1] **DUKPT**, *Derived unique key per transaction*

- Interfaz de comandos **HSM**: así como el caso de uso anterior, consiste en exponer el comando de desencriptación de los módulos **HSM**. La intención es que **HS-API** sea el punto de acceso a los módulos y, por lo tanto, exponga todos los comandos/operaciones que se requieran de los módulos **HSM**. Los comandos pueden ser: crear llaves criptográficas, encriptar/desencriptar datos, etc.

- Gestor de llaves para procesos de tokenización: la tokenización de datos es una característica deseada por las empresas que brindan soluciones de seguridad informática. La tokenización, en su variedad más simple, es un proceso que consiste en generar una cadena de texto aleatoria que representa una referencia en un almacenamiento a datos de información sensibles que se han guardados encriptados. Si bien toda esta responsabilidad escapa del espíritu de **HS-API**, la idea planteada es que se desarrollen las características de tokenización en otro microservicio de la arquitectura y que el gestor de estas llaves sea **HS-API**, lo cual al tratarse de llaves creadas a través de módulos **HSM** incrementa la seguridad de toda la operación en general.

Además de los interesantes casos de usos planteados, el proyecto debe seguir siendo mantenido, por lo que se harán las actualizaciones necesarias para cumplir con los requisitos de calidad impuestos por el equipo de calidad. Y desde luego, la aplicación es constantemente monitoreada por lo que en caso de ocurrir algún error, este debe ser tratado y solucionado.

Bibliografía

[1] *National Institute of Standards and Technology* - **NIST** - 20/10/2021 - `https://www.nist.gov`

[2] *Computer Security Resource Center* - **CSRC** - 20/10/2021 - `https://csrc.nist.gov`

[3] **T**riple **DES** - 31/10/2021 - `https://en.wikipedia.org/wiki/Triple_DES`

[4] Advanced Encryption Standard - **AES** - 31/10/2021 - `https://en.wikipedia.org/wiki/Advanced_Encryption_Standard`

[5] **Spring** - 31/10/2021 - `https://spring.io/`

[6] Estandarización Federal de Procesamiento de Información, *Federal Information Processing Standardization* - 19/09/2021 - `https://www.nist.gov/standardsgov/compliance-faqs-federal-information-processing-standards-fips`

[7] Criterios Comunes, *Common Criteria* - 19/09/2021 - `https://www.commoncriteriaportal.org`

[8] *Payment Card Industry Data Security Standard* - **PCI DSS** - 31/10/2021 - `https://www.pcicomplianceguide.org/faq/#1`

[9] Data Encryption Standard - **DES** - 13/03/2022 - https://en.wikipedia.org/wiki/Data_Encryption_Standard

[10] Reporte de vulnerabilidad en algoritmos **DES** y **3DES** - *CVE-2016-2183 Detail* - 11/10/2021 - https://nvd.nist.gov/vuln/detail/CVE-2016-2183

[11] **Kubernetes** - 09/10/2021 - https://kubernetes.io

[12] *Enterprise JavaBeans* - **EJB** - 13/03/2022 - https://www.oracle.com/java/technologies/enterprise-javabeans-technology.html

[13] Modelo Vista Controlador - **MVC** - 13/03/2022 - https://developer.mozilla.org/en-US/docs/Glossary/MVC

[14] Transmission Control Protocol - **TCP** - 14/08/2021 - https://en.wikipedia.org/wiki/Transmission_Control_Protocol

[15] **Gradle** - 14/10/2021 - https://gradle.org

[16] **OAuth 2.0** - 09/10/2021 - https://oauth.net/2/

[17] **Scrum** - 31/10/2021 - https://www.scrum.org/

[18] **TeamCity** - 18/10/2021 - https://www.jetbrains.com/teamcity/

[19] **Octopus Deploy** - 18/10/2021 - https://octopus.com

[20] Al-Zewairi, M. , Biltawi, M. , Etaiwi, W. and Shaout, A. (2017) *Agile Software Development Methodologies: Survey of Surveys. Journal of Computer and Communications*, **5**, 74-97. doi: https://10.4236/jcc.2017.55007

[21] **DockerHub** - 20/10/2021 - https://hub.docker.com

[22] *YAML Ain't Markup Language* - **YAML** - 16/10/2021 - https://yaml.org

[23] **OpenAPI** Specification - 20/10/2021 - https://swagger.io/specification/

[24] **JSON** Web Tokens - **JWT** - 09/10/2021 - https://jwt.io

[25] Microsoft SQL Server - 09/10/2021 - https://es.wikipedia.org/wiki/Microsoft_SQL_Server

[26] Spring, TCP and UDP Support - 17/08/2021 https://docs.spring.io/spring-integration/docs/current/reference/html/ip.html