AAT 6356
BN 186806

UCAB Universidad Católica ANDRÉS BELLO

FACULTAD DE INGENIERÍA

ESCUELA DE INGENIERÍA INFORMÁTICA

PUNTOS DE VENTA CITYWALLET BASADOS EN PLATAFORMA INTERNET DE LAS COSAS DE AMAZON WEB SERVICES

TRABAJO INSTRUMENTAL DE GRADO

Presentado ante la

UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Como parte de los requisitos para optar al título de

INGENIERO EN INFORMÁTICA

REALIZADO POR	Manuel Alejandro Pacheco León
TUTOR ACADÉMICO	Ing. Biagio Cante
TUTOR EMPRESARIAL	Ing. Félix Fernández Pérez
FECHA	Septiembre 2017

TESIS
II 2017
P3

UCAB Universidad Católica ANDRÉS BELLO

FACULTAD DE INGENIERÍA

ESCUELA DE INGENIERÍA INFORMÁTICA

PUNTOS DE VENTA CITYWALLET BASADOS EN PLATAFORMA INTERNET DE LAS COSAS DE AMAZON WEB SERVICES

Este Jurado; una vez realizado el examen del presente trabajo ha evaluado su contenido con el resultado: Dieciocho (18) Puntos

JURADO EXAMINADOR

Firma:	Firma:	Firma:
Nombre: RAFAEL A. LARA CAMPOS	Nombre: _____	Nombre: P/ VALERIA LEÓN

REALIZADO POR	Manuel Alejandro Pacheco León
TUTOR ACADÉMICO	Ing. Biagio Cante
TUTOR EMPRESARIAL	Ing. Félix Fernández Pérez
FECHA	Septiembre 2017

DEDICATORIA

A mis padres, Jeanett León y Jose Luis Pacheco, y mi hermano Luis Pacheco. Sin ustedes este trabajo no hubiese sido posible.

A mis tías, Leivis León, Anelsy León, Yelithza León, Jacqueline Riera, mis abuelos Pedro León y María Arcia, su apoyo incondicional es invaluable.

A mí persona, que sirva como recordatorio que puedes con todos los obstáculos de la vida y con mucha más, cuando sientas que no puedes seguir la batalla, mira hacia atrás y recuerda todos los logros que has conseguido, en especial este, y todas las personas que te apoyan y aman incondicionalmente. Si puedes, siempre podrás.

AGRADECIMIENTOS

A mis padres, Jeanett León y Jose Luis Pacheco, que lo han dado todo por mí, su apoyo y amor incondicional, por ayudarme a lo largo de este proceso y en mi vida. A mi hermano Luis Pacheco, el cual siembre me ha brindado su amor y apoyo durante toda su vida.

A mis tías, Leivis León, Anelsy León, Yelithza León, Jacqueline Riera, por siempre estar pendientes de mi trabajo, mi vida y siempre estar ahí cuando las necesitaba, y a mis abuelos Pedro León y María Arcia por las mismas razones, las amo. A mi prima Anelsy Acosta, por acompañarme durante este proceso realizando un trabajo de grado al igual que yo.

A la directiva de CityWallet, Ramón Ginéz, Liesl Isler, Atilana Piñón, Hernán García y Felix Fernández, por darme la oportunidad de trabajar y dearrollarme como profesional y humano en su empresa, lo cual ha sido una experiencia invaluable. A mis amigos y compañeros de trabajo Carlos Unda, María Unda, Alejandro García, July Marval, Javier Pérez, Santiago Bernal y Graciela Sosa, por todos los momentos vividos tanto dentro como fuera de la oficina, por el apoyo y la ayuda incondicional.

A mis mejores amigos, Fabiola Yepez, Armando Muñoz y Ana Gabriela Sotelo, simplemente por el hecho de haber estado conmigo durante este proceso apoyándome, enseñándome que debo dar lo mejor de mí, los adoro, aprecio y amo.

Por ultimo a mis tutores empresarial y académico Félix Fernández y Biagio Cante por el apoyo y guía durante el desarrollo de este trabajo de grado.

INDICE

DEDICATORIA I
AGRADECIMIENTOS II
INDICE III
INDICE DE FIGURAS VI
SINOPSIS VIII
CAPÍTULO I VIII
1. Necesidades de la Empresa 1
1.1. Planteamiento del Problema 1
1.2. Solución Propuesta 4
2. Objetivos 10
2.1. Objetivo General 10
2.2. Objetivos Específicos 10
3. Aportes 11
3.1. Aportes Tecnológicos 11
3.2. Aportes Funcional 11
4. Alcance y Limitaciones 12
4.1. Alcance 12
1.1.1. Objetivos Específicos 12
1.1.2. Aportes Funcionales 16
1.1.3. Aportes Tecnológicos 17
4.2. Limitaciones 18
5. Justificación 19
CAPÍTULO II - MARCO TEÓRICO 21
1. CityWallet 21
2. FinTech 22
3. NFC 23
4. IoT 24
5. Arquitectura de software 25
5.1. Microservicios 25
6. AWS 26
7. Microservicios AWS 27

7.1. AWS Lambda 27
7.2. AWS IoT 27
7.3. AWS IoT Shadows 28
7.4. AWS SNS 29
8. Notificaciones Push 29
9. Google Firebase 29
10. POS 30
10.1. Check-in 31
10.2. Check-out 31
11. Geateway 31
12. Firma Digital 31
12.1. DSA 32
12.2. SHA 32
CAPÍTULO III - MARCO METODOLÓGICO 33
1. Método de Desarrollo 33
1.1. Artefactos 34
1.2. Eventos 35
1.3. Roles 36
CAPÍTULO IV - DESARROLLO 41
1. Introducción 41
2. Sprints 42
2.1. SPRINT 0 42
2.2. SPRINT 1 46
2.3. SPRINT 2 54
2.4. SPRINT 3 59
2.5. SPRINT 4 62
2.6. SPRINT 5 63
2.7. SPRINT 6 66
2.8. SPRINT 7 68
2.9. SPRINT 8 69
2.10. SPRINT 9 73
CAPÍTULO V - RESULTADOS 77
1. Resultados 77

2. Conclusiones 83
3. Recomendaciones 84
REFERENCIAS BIBLIOGRÁFICAS 86
APÉNDICES 92
1. Figuras 92
2. Sprint Backlogs 120
2.1. Sprint 1 120
2.2. Sprint 2 121
2.3. Sprint 3 121
2.4. Sprint 4 122
2.5. Sprint 5 122
2.6. Sprint 6 123
2.7. Sprint 7 124
2.8. Sprint 8 124
2.9. Sprint 9 125

INDICE DE FIGURAS

Figura 1. Arquitectura Actual de los Puntos de Servicio de CityWallet 2
Figura 2. Arquitectura de los POS con AWS IoT integrado. 5
Figura 3. Estructura de CityWallet con AWS IoT integrado desde el punto de vista de los POS. 9
Tabla 1. Product Backlog 43
Figura 4. Diagrama de Arquitectura POSv1 92
Figura 5. Diagrama de Arquitectura POSv2 92
Figura 6. Ejemplo perfil de un usuario guardado en la base de datos local del POS ... 93
Figura 7. Metodo IsValidUser del Gateway en Java 94
Figura 8. Metodo IsValidUser del Gateway en Python 96
Figura 9. Metodo UserHasPlan del Gateway en Java 97
Figura 10. Metodo UserHasPlan del Gateway en Python 98
Figura 11. Ejemplo de token 98
Figura 12. Metodo de createToken en Gateway de Java 99
Figura 13. Metodo de createToken en Gateway de Python 100
Figura 14. Método de isValidTransaction en Gateway de Java 101
Figura 15. Método de isValidTransaction en Gateway de Python 102
Figura 16. Ejemplo de una transacción almacenada en base de datos local del POS 102
Figura 17. Método createParkingTransaction desarrollado en el Gateway de Java ... 105
Figura 18. Método createParkingTransaction desarrollado en el Gateway de Python 107
Figura 19. Ejemplo de Logs en el POS 107
Figura 20. Pantalla de Dashboard de AWS IoT Console 107
Figura 21. Listado de POS registrados en la plataforma AWS IoT 108
Figura 22. Formulario para la creación de un “Thing 108
Figura 23. Listado de POs registrados en AWS IoT con el nuevo POS 109
Figura 24. Opcion de Connect en panel de AWS IoT 109
Figura 25. Selección del sistema operativo y lenguaje de programacion del POS 110
Figura 26. Seleccion del POS creado previamente 110
Figura 27. Descarga del Kit de Conexión para el POS 111
Figura 28. Interacción del POS con la plataforma AWS IoT 111

Figura 29. Pruebas QA IoT 112
Figura 30. Ejemplo del Shadow de un POS 112
Figura 31. Ejemplo de cambio en un atributo del Shadow de un POS 112
Figura 32. Pruebas QA del POS 113
Figura 33. Pantalla Inicio de Sesión 113
Figura 34. Pantalla Cambio de Contraseña 114
Figura 35. Pantalla de Dashboard 115
Figura 36. Pantalla de Perfil del POS 116
Figura 37. Politica iot-data-access 116
Figura 38. Ejemplo del JSON que recibe el microservicio 117
Figura 39. Listado de POS disponibles aplicación Admin 117
Figura 40. Listado de POS disponibles aplicación Merchant 118
Figura 41. Perfil del POS en aplicación Admin 118
Figura 42. Perfil del POS en aplicación Merchant 119
Figura 43. Proceso de Modificación de un Shadow Mediante aplicacion cliente 119
Tabla 2. Sprint Backlog correspondiente al Sprint 1 120
Tabla 3. Sprint Backlog correspondiente al Sprint 2 121
Tabla 4. Sprint Backlog correspondiente al Sprint 3 121
Tabla 5. Sprint Backlog correspondiente al Sprint 4 122
Tabla 6. Sprint Backlog correspondiente al Sprint 5 122
Tabla 7. Sprint Backlog correspondiente al Sprint 6 123
Tabla 8. Sprint Backlog correspondiente al Sprint 7 124
Tabla 9. Sprint Backlog correspondiente al Sprint 8 124
Tabla 10. Sprint Backlog correspondiente al Sprint 9 125

SINOPSIS

El presente Trabajo Instrumental de Grado, titulado "Puntos de Venta CityWallet Basados en Plataforma Internet de las Cosas de Amazon Web Services", se basa en el desarrollo de la segunda versión de los puntos de venta (POS) de CityWallet, que tiene como objetivo la migración del Gateway que provee al POS, el cual fue desarrollando en Java, a Python e integrarlo con el código existente del POS. También, la implementación e integración de la plataforma de Internet de las Cosas con los puntos de venta para generar una red virtual de estos puntos, que permita servir como soporte para el desarrollo de aplicaciones que consuman de este servicio, así como mejorar el monitoreo de los mismos. Para llevar a cabo el proyecto fue planteado el uso de la metodología de desarrollo ágil SCRUM, modificado para cubrir las necesidades y estándares de la empresa.

Añadido a esto, posee una aplicación móvil que consumirá del servicio de IoT para el monitoreo y control remoto de los puntos de venta. Las aplicaciones existentes de CityWallet (portales web para el departamento administrativo tanto de la empresa como de los comercios) también poseerán integración con esta plataforma con el objetivo de extender el soporte a usuarios dueños de los establecimientos que posean CityWallet.

Por último, se integraron los puntos de venta, la aplicación desarrollada para monitoreo de los POS y las aplicaciones administrativas de CityWallet con un microservicio que se encargará del manejo de las notificaciones push entre los POS y las aplicaciones clientes. Esto con el objetivo de generar un proceso de notificaciones de los POS en donde el usuario (administrador, ya sea del comercio o de CityWallet) pueda recibir mensajes referentes los diferentes estados en los que puede encontrarse un punto de venta perteneciente a su estacionamiento.

CAPÍTULO I

1. Necesidades de la Empresa

1.1. Planteamiento del Problema

CityWallet es una empresa start up que busca implementar pagos electrónicos de microtransacciones en la ciudad de Caracas a través de tecnología NFC. La empresa lleva casi un año de trayectoria enfocándose en desarrollar esta tecnología para pagos en donde los puntos de venta comunes no llegan. Actualmente han logrado implementar sus servicios de pago en estacionamientos, brindando a los mismos unos puntos de servicio (POS, o Point of Sale en inglés) los cuales sirven como lectores NFC. A su vez estos están conectados a un dispositivo Gateway que es el encargado de realizar las peticiones a los servicios en la nube que cumplen las distintas funcionalidades referentes a las microtransacciones. Un usuario puede adquirir una calcomanía CityWallet que trabaja con tecnología NFC, recargar saldo a través de la aplicación o página web, para luego ser consumido en los distintos establecimientos. El usuario presenta la calcomanía ante el lector del POS y este es el encargado de conectarse con el Gateway para el consumo de los servicios web en la nube.

A pesar de que en la actualidad el principal mercado de CityWallet son los estacionamientos, el pago por NFC puede

extenderse a otros establecimientos como quioscos y estaciones de servicio e incluso el transporte público, máquinas expendedoras, etc.

CityWallet cuenta con un sistema en producción que se encuentra activo en dos establecimientos de la ciudad de Caracas (estacionamiento de la Torre Xerox, y estacionamiento del Centro Comercial La Lagunita), sin embargo, contemplado la anterior idea de expandirse a otros mercados y la potencial expansión de la empresa a otros países, CityWallet se ve en la necesidad de mejorar sus procesos y tecnologías para brindar una plataforma y un servicio con mayor seguridad, facilidad de uso y control y confiabilidad.

Hoy en día la estructura de los puntos de servicio que brinda CityWallet es de la siguiente manera:

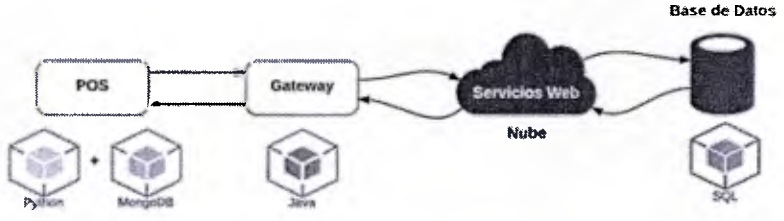

Figura 1. Arquitectura Actual de los Puntos de Servicio de CityWallet

Con esta estructura la empresa ha logrado exitosamente implementar la plataforma de pagos con NFC en dos

estacionamientos, sin embargo, se ha enfrentado a una serie de desventajas y problemas.

El primer problema y el más notorio es que para que el POS funcione necesita de una conexión constante a internet. Al presentar una calcomanía en el lector el POS hace conexión con el Gateway para consumir de los servicios web que se encuentran en la nube y que son solo accesibles cuando el Gateway posee conexión a internet. En Venezuela, es bien conocido el problema con la confiabilidad en las conexiones de internet, sean estas brindadas por empresas privadas o públicas. El tener una conexión intermitente o con periodos largos de caída del servicio compromete el funcionamiento normal del POS, ya que este será incapaz de subir transacciones a los servicios web para que estos guarden en base de datos o bajar información de perfiles de usuarios para el reconocimiento de las calcomanías.

Otro problema es el control y monitoreo de los POS. Los puntos de servicio están diseñados para trabajar las 24 horas del día, sin embargo, como todo dispositivo electrónico a veces presentan fallas, errores de lectura de calcomanía, errores en la conexión a internet o simplemente un apagado inesperado debido a alguna falla eléctrica o apagón. Estos distintos estados del POS no son monitoreables a distancia, ni tampoco existe algún tipo de contingencia rápida, aparte de trasladarse al lugar y arreglar la

falla estando físicamente donde se encuentra el punto, a la hora en que se presente alguno de los inconvenientes ya mencionados.

Una desventaja también de la estructura actual de los POS es que se necesitan de dos dispositivos Raspberry Pi, uno que actué como punto de servicio y otro como Gateway. A pesar de que varios POS se pueden conectar a un mismo Gateway, la inversión monetaria a la hora de implantar el sistema en un nuevo establecimiento siempre será mayor si se necesitan al menos dos Raspberries para que el sistema funcione, teniendo en cuenta de que es posible tener el Gateway y el POS en un mismo dispositivo.

1.2. Solución Propuesta

La solución que se propone ante los problemas anteriormente mencionados, es la integración de la plataforma de CityWallet con la de Amazon Web Services Internet of Things (IoT) y la integración del POS y el Gateway en un solo dispositivo. Amazon Web Services (AWS, o Servicios Web de Amazon) Internet of Things (IoT, o Internet de las Cosas en español) es una plataforma que permite conectar dispositivos a los servicios de AWS, a otros dispositivos, a datos e interacciones seguras, a procesos y acciones sobre esos datos, y permite que las

aplicaciones puedan interactuar con los mismos dispositivos, incluso cuando están fuera de línea.

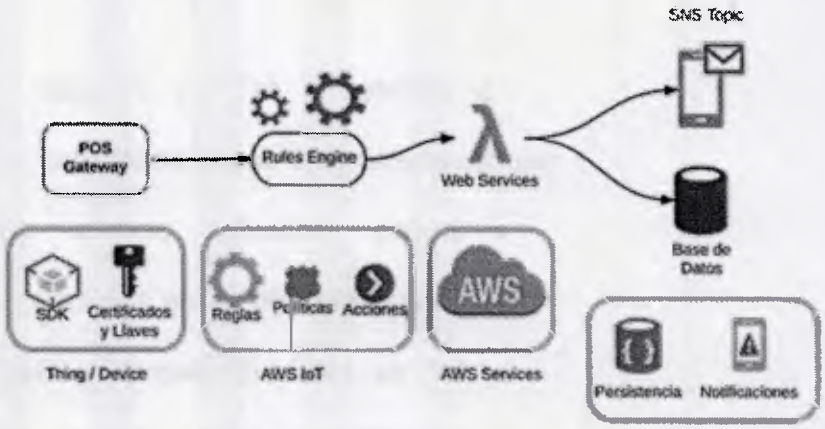

Figura 2. Arquitectura de los POS con AWS IoT integrado.

En la figura 2 se muestra cómo quedaría la estructura de los POS al ser integrado con la plataforma IoT. Como se puede ver la estructura se divide en 4 partes principales: 1) Los POS Gateway se comportan como "cosas" que formaran parte de la plataforma IoT. Estos usaran certificados para autenticarse contra los servicios y poder acceder a ellos. 2) Rules Engine, que es el encargado de ser puente entre los web services y los POS Gateway. Aquí se concentran las políticas, reglas y acciones que regirán la plataforma IoT de CityWallet. 3) Servicios web basados en AWS, los cuales serán en cargados de la lógica y procesamiento de datos en la nube. 4) Base de datos y tópicos SNS (Simple Notification Service), siendo estos la capa de

persistencia de los datos y las notificaciones entre dispositivos, respectivamente.

La plataforma IoT de Amazon Web Service se divide principalmente en 5 partes principales:

- Device Gateway: permite que los dispositivos se comuniquen de forma segura y eficiente con la plataforma IoT de AWS. El dispositivo Gateway puede intercambiar mensajes a través de un modelo de publicación / suscripción, que permite comunicaciones uno a uno y uno a muchos. Con este patrón de comunicación uno-a-muchos AWS IoT hace posible que un dispositivo conectado pueda transmitir datos a varios suscriptores para un tema determinado. El dispositivo Gateway soporta MQTT, WebSockets, y los protocolos HTTP 1.1 y se puede implementar fácilmente soporte para protocolos propietarios o heredados. (Amazon Web Services I., IoT How It Works, 2016)
- Autenticacion y Autorización: proporciona autenticación mutua y cifrada en todos los puntos de conexión, de modo que los datos nunca se intercambian entre los dispositivos IoT sin identidad probada. AWS IO soporta el método de autenticación de AWS (llamado 'SigV4'), así como

certificado X.509 autenticación basada. También puede asignar el tipo de rol y / o políticas para cada certificado, por lo que se puede autorizar a los dispositivos o aplicaciones para tener acceso, o cambiar de opinión y revocar el acceso por completo sin tener que tocar el dispositivo. (Amazon Web Services I., IoT How It Works, 2016)

- Registro: establece una identidad para los dispositivos y está encargado de hacerle seguimiento a los metadatos, como atributos y capacidades de los dispositivos. El Registro asigna una identidad única a cada dispositivo que tiene el formato consistente independientemente del tipo de dispositivo o cómo se conecta. También es compatible con metadatos que describen las capacidades de un dispositivo. (Amazon Web Services I., IoT How It Works, 2016)
- Device Shadow: los shadows o sombras son una versión persistente y virtual de cada dispositivo que incluye último estado del mismo para que las aplicaciones u otros dispositivos puedan leer mensajes e interactuar con él sin estar conectado directamente al mismo. Las sombras se mantienen el último estado informado y el estado deseado de cada dispositivo, incluso cuando el dispositivo está fuera

de línea. (Amazon Web Services I., IoT How It Works, 2016)

- Rules Engine: es el encargado de evaluar los mensajes entrantes publicados en la plataforma IoT. Transforma y entrega los mensajes a otro dispositivo o un servicio en la nube, basado en reglas de negocio definidas. Una regla puede aplicarse a los datos de uno o varios dispositivos, y puede tomar una o muchas acciones en paralelo. (Amazon Web Services I. , IoT How It Works, 2016)

Integrando estas cinco partes dentro de la plataforma de los puntos se servicios de CityWallet, se busca afrontar las problemáticas principales referentes a la baja disponibilidad del servicio de internet (con la implementación de sombras en los dispositivos) y la inexistencia de una interfaz por la cual se puedan monitorear y controlar los POS de forma remota. Sumado a esto, la plataforma IoT también ofrece integración con el servicio de notificaciones SNS, por el cual se pretende gestionar los mensajes que se envíen entre dispositivos y aplicaciones. (Amazon Web Services I., Amazon Simple Notification Service (SNS), 2016)

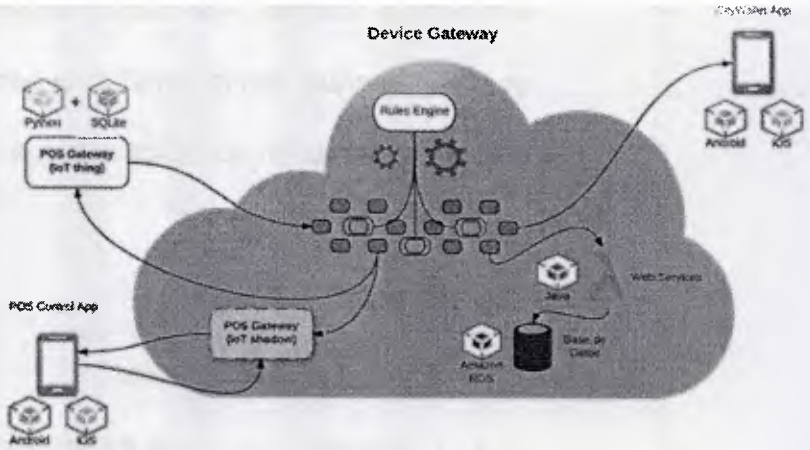

Figura 3. Estructura de CityWallet con AWS IoT integrado desde el punto de vista de los POS.

(Amazon Web Services I., 2016)

En la figura 3 se puede observar una vista general de cómo será la plataforma de IoT integrada con CityWallet. El flujo de la información en el diagrama va de la siguiente manera: del POS Gateway salen las peticiones hacia el Rules Engine, este es el encargado de repartir las peticiones entre los diferentes web services dependiendo del servicio que se requiera ser consumido. Los web services serán la conexión contra la base datos para la consulta y almacenamiento de información. La POS control app será la encargada del monitoreo y control de los POS, esta aplicación se conectará con el shadow del dispositivo para así enviar y recibir información sobre el mismo. El shadow o sombra del POS es una versión virtual en la nube que será capaz de responder por el POS físico en caso de que este no tenga

conexión a internet. Por último, tenemos la app de CityWallet la cual se integrará con la plataforma con la simple funcionalidad de obtener información sobre los estados de los puntos de servicio.

2. Objetivos

2.1. Objetivo General

- Desarrollar los puntos de servicio (POS) de CityWallet teniendo como base la plataforma de Amazon Web Services (AWS) Internet of Things (IoT).

2.2. Objetivos Específicos

1. Desarrollar la segunda versión los Puntos de Servicio (POS).
2. Implementar una configuración del POS que le permita conectarse a un gateway remoto.
3. Desarrollar la aplicación móvil para monitoreo de los POS integrados con IoT.
4. Integrar la aplicación móvil para monitoreo de los POS con la plataforma AWS IoT y SNS.
5. Desarrollar un microservicio para el manejo de las notificaciones push entre el POS y las aplicaciones clientes de CityWallet.
6. Integrar los web services necesarios de CityWallet con la nueva versión de los POS.

7. Integrar la plataforma AWS IoT con las aplicaciones administrativas de CityWallet.
8. Integrar el microservicio para manejo de notificaciones push con las aplicaciones administrativas de CityWallet.

3. Aportes

3.1. Aportes Tecnológicos

9. Integración de los POS con la plataforma AWS IoT.
10. Implementación de Shadows para el manejo de estados de los POS incluso en caso de que no posean conexión a internet.
11. Integración con plataforma de notificaciones AWS SNS y Google Firebase para la gestión de notificaciones push entre los puntos de venta y las aplicaciones cliente.

3.2. Aportes Funcional

12. Rediseño de la arquitectura de POS de CityWallet por medio de la integración del Gateway y el punto de servicio en un solo dispositivo e integrándolo con la plataforma AWS IoT.
13. Diseño y desarrollo del proceso de control y monitoreo remoto de los POS a través de una aplicación móvil.

4. Alcance y Limitaciones

4.1. Alcance

Objetivos Específicos

1. Desarrollar de segunda versión los Puntos de Servicio (POS): Los POS son los encargados de realizar las transacciones cada vez que una calcomanía es presentada en el lector NFC. Actualmente para que el POS pueda consumir de los servicios web de CityWallet, registre transacciones y valide usuarios, necesita conectarse a través de un Gateway que le proporcione las peticiones a la nube. Este Gateway es un dispositivo aparte del POS. La segunda versión de los puntos de servicios contempla unir estos dos dispositivos en uno solo. Actualmente el desarrollo del POS está en Python y MongoDB y la del Gateway en Java. Para la segunda versión los requerimientos son: 1) Que el POS y el Gateway se integren a base de un solo lenguaje de programación, que en este caso será Python. 2) El nuevo Gateway integrado con el POS será capaz de conectarse con los servicios web basados en AWS. La segunda versión del POS estará conectada con la plataforma de Internet de las Cosas de Amazon Web Services. Esto con el fin de generar una red de puntos de venta de CityWallet los cuales puedan utilizar las funciones de IoT para el manejo de estados del POS, integración con los Shadows (o sombras, en español) de IoT, publicación y

suscripción a tópicos para el intercambio de mensajes contra la plataforma de AWS y aplicaciones cliente.

2. Implementar una configuración del POS que le permita conectarse a un gateway remoto: Por requerimientos de los comercios y dependiendo de los recursos que ellos posean, el punto de venta o POS de CityWallet debe ser capaz de reconocer a qué tipo de Gateway se va a conectar (si es local o remoto) a través de un archivo de configuración. Este reconocimiento se realizará cada vez que se inicialice el POS, con el fin de establecer la ruta de conexión contra la que se conectará el POS al gateway. En caso de que la conexión sea contra un gateway remoto, se desarrollará un componente capaz de realizar peticiones del tipo HTTP para poder consumir de los métodos brindados por el gateway remoto.

3. Desarrollar la aplicación móvil para monitoreo de los POS integrados con IoT: La aplicación móvil será capaz obtener información actualizada conectándose a la sombra de cualquier POS a través de la plataforma de Amazon Web Services, así como también realizar funciones sobre el mismo tales como reinicio del mismo en caso de que se necesite.

4. Integrar la aplicación móvil para monitoreo de los POS con la plataforma AWS SNS para recepción de notificaciones push: La aplicación móvil debe ser capaz de recibir notificaciones push

cuando un POS cambie de estado. Para esto, se integrará con el SDK de AWS SNS con el objetivo de que la aplicación sea capaz de suscribirse a tópicos definidos por CityWallet, a donde el POS pueda publicar mensajes de interés del usuario.

5. Desarrollar un microservicio para el manejo de las notificaciones push entre el POS y las aplicaciones clientes de CityWallet: Las notificaciones push enviadas por el POS pueden ser recibidas por la aplicación móvil encargada de monitorear los puntos de venta o por cualquier aplicación administrativa de CityWallet (admin o merchant). Para esto, se requiere de un servicio que reciba el mensaje del POS y se encargue de distribuirlo a los receptores debidos. Este servicio se integrará con las librerías de AWS SNS (para el envío de notificación push a la aplicación móvil a través de publicación de mensajes a tipos definidos por CityWallet) y a la librería de Google Firebase Admin (para el envío de notificación push a aplicación web que se ejecuten en un navegador).

6. Integrar los web services necesarios de CityWallet con la nueva versión de los POS: Las nueva versión del POS debe ser capaz de interactuar con los servicios web de CityWallet. En este caso, se integrará con el microservicio para el manejo de las notificaciones push desarrollado en el objetivo número 6, esto con el fin de que el POS sea capaz de enviar un mensaje en forma de

JSON cuando lo necesite, para que este sea entregado en forma de notificación push a las aplicaciones clientes (aplicaciones administrativas de CityWallet y aplicación móvil encargada de monitorear los POS).

7. Integrar la plataforma AWS IoT con las aplicaciones administrativas de CityWallet: CityWallet hoy en día cuenta con 2 aplicaciones administrativas: un módulo merchant en donde los comercios pueden gestionar CityWallet en su establecimiento y un módulo Admin donde los administradores de CityWallet gestionan todos los comercios que han adquirido el servicio de la empresa. Estas tres aplicaciones deberán ser capaces de conectarse con los servicios de AWS IoT, así como también obtener información del comportamiento de los POS en los distintos establecimientos.

8. Integrar el microservicio para manejo de notificaciones push con las aplicaciones administrativas de CityWallet: Para que las aplicaciones administrativas de CityWallet puedan recibir notificaciones push, están primero deber ser integradas con la librería de Google Firebase para JavaScript. Una vez realizado esto, las aplicaciones podrán recibir notificaciones provenientes del POS a través del microservicio desarrollado en el objetivo número 6.

1.1.1. Aportes Funcionales

9. Integración de los POS con la plataforma AWS IoT: Los POS actualmente son dispositivos independientes por los cuales no se tiene control ni monitoreo remotamente. Con la integración de la plataforma IoT de Amazon Web Service se busca centralizar el control y monitoreo de los POS sin tener que estar físicamente en el lugar en donde están instalados. La Plataforma IoT ofrece 4 grandes funcionalidades que se pueden integrar con los POS y que cumplen con los siguientes requerimientos: 1) Centralización del control y monitoreo de los POS a través del Device Gateway de IoT. 2) Autenticación y autorización de un POS como un dispositivo individual para así poder distinguir y reconocer los dispositivos, así como también brindar una capa de seguridad a través de certificados y conexión cifrada contra el Device Gateway. 3) Acceso al POS incluso cuando este esté sin conexión a internet. Esto se logra a través de los "shadows" o sombras de los dispositivos. Cada POS contará con su propia sombra que estará alojada virtualmente en la nube y este será el encargado de contestar las peticiones (teniendo en cuenta el último estado del POS antes de perder la conexión a internet) que provengan de otros dispositivos o aplicación. También servirá como capa intermedia entre las aplicaciones móviles y el POS, ya que las aplicaciones solo accederán a la sombra y su estado y nunca al

POS físico en sí. 4) Proporcionar un motor de reglas que sea el encargado de decidir qué funciones acceder y a donde se tienen que enviar los mensajes cuando se trate de comunicación entre dispositivos.

10. Implementación de Shadows para el manejo de políticas offline de los POS en caso de que no posean conexión a internet: Todos los dispositivos POS Gateway deben tener dentro de la plataforma AWS IoT un "shadow" o sombra, el cual cumplirá la función de ser una representación virtual del POS físico para que cuando este no posea conexión a internet el shadow entre en acción como un respaldo del último estado en el que estuvo conectado el POS y así poder ser accedido desde las aplicaciones de monitoreo y control.

11. Integración con plataforma de notificaciones AWS SNS para la gestión de mensajes entre dispositivos: Las aplicaciones móviles de CityWallet deben ser capaces de recibir notificaciones sobre los estados de los POS, esto se logra con la plataforma de notificaciones de AWS SNS (Amazon Web Services Simple Notification Service) el cual será el encargado de enviar los mensajes de notificaciones hacia los dispositivos que lo necesiten.

1.1.2. Aportes Tecnológicos

12. Rediseño de la arquitectura de POS de CityWallet por medio de la integración del Gateway y el punto de servicio en un solo

dispositivo e integrándolo con la plataforma AWS IoT: Ese rediseño implica la transición desde la plataforma actual a la plataforma integrada con IoT (ver figuras 1 y 2). El rediseño implica la nueva versión del POS en donde se integra el punto de servicio con el Gateway y la integración del mismo con AWS IoT.

13. Diseño y desarrollo del proceso de control y monitoreo remoto de los POS a través de una aplicación móvil: Una de las ventajas de que los POS pasen a ser parte de la plataforma AWS IoT es que se puede consumir de los estados del mismo a través del shadow. Para esto se desarrollará una aplicación que sea capaz de conectarse contra la plataforma de IoT y consumir del shadow de dispositivo POS para el control y monitoreo de los estados del mismo.

4.2. Limitaciones

- El lenguaje de programación que se va a usar para el desarrollo de la segunda versión del POS es Python 2.7.
- La aplicación móvil será desarrollada con Ionic 2, y consumirá de los SDK de AWS IoT y SNS para JavaScript.
- La aplicacion móvil encargada del monitoreo de los POS y administrativa de CityWallet usarán AWS Cognito para la autenticación de los usuarios.

- El microservicio de notificaciones push para las aplicaciones administrativas de CityWallet será Google Firebase.
- Los POS deberán poseer certificados del tipo X.509 para poder autenticarse ante los servicios de AWS IoT.
- El microservicio desarrollado debe estar basado en la plataforma Amazon Web Service Lambda.
- El microservicio estará desarrollado en el entorno multiplataforma NodeJs.
- Para las notificaciones push, la aplicación móvil usará AWS SNS y las aplicaciones administrativas Google Firebase.
- El lenguaje de programación de la librería de Google Firebase a usar es JavaScript.

5. Justificación

CityWallet es una empresa que se encontraba migrado su plataforma actual a la plataforma de Amazon Web Services. Como AWS ofrece una gama amplia de servicios, CityWallet vio la oportunidad de mejora de sus puntos de venta a través de la integración de los mismos con la tecnología de Internet de Las Cosas que ofrece AWS. La principal razón de este desarrollo sería poder generar una aplicación por la cual se pudiera consultar información sobre los puntos de venta instalados en los estacionamientos y poder controlarlos remotamente, así como también poder recibir notificación sobre el estado de los mismos. Esto agregaría valor a la propuesta de CityWallet a los

estacionamientos ya que amplían la gama de servicios que ofrecen a los clientes al instalar CityWallet en sus establecimientos.

Al estar CityWallet soportado por la plataforma de AWS, gana reconocimiento y confianza ya que su plataforma ahora estará basada en nuevas tecnologías provenientes de una empresa de reconocimiento a nivel mundial, además que el mantenimiento de la plataforma pasa a ser ámbito de Amazon, ya que CityWallet solo estaría consumiendo de sus servicios como cliente, y su preocupación principal pasaría a ser solo en el mantenimiento de las funciones actuales y desarrollo de nuevas pero no de la plataforma en sí.

CAPÍTULO II - MARCO TEÓRICO

A continuación, se fundamentan los conceptos básicos que constituyen la base teórica del presente trabajo, el objetivo de este marco referencial es lograr el completo entendimiento del lector sobre el desarrollo y cumplimiento de los objetivos principales de la investigación.

1. CityWallet

Es una herramienta de pagos móviles que aporta a la construcción de ciudades inteligentes y sostenibles, dando la oportunidad a un mejor aprovechamiento del tiempo por parte de los ciudadanos e impactando positivamente su calidad de vida.

CityWallet es una empresa emergente cuyo propósito es construir una red de micropagos de fácil acceso para todos los sectores de la población (bancarizados o no), donde el usuario deberá crearse una cuenta online a través de la web o su teléfono inteligente, añadir saldo (mediante su tarjeta de débito/crédito, cuenta bancaria y/o efectivo) y realizar pagos en los comercios afiliados.

Actualmente, utiliza la tecnología NFC mediante calcomanías inteligentes como medio de pago en los principales estacionamientos de Caracas, Venezuela. Tras una extensa investigación de mercado a nivel nacional, "la calcomanía inteligente" fue identificada por los usuarios potenciales como el medio más seguro y confiable para ejecutar pagos cotidianos, siendo de fácil acceso a todos los sectores.

2. FinTech

La palabra Fintech es un término compuesto que sale de unir las palabras Finance y Technology, es decir, es una palabra que surge de la unión de dos y que hace referencia a todas aquellas empresas de servicios financieros que utilizan la última tecnología existente para ofrecer productos y servicios financieros innovadores en el mercado [1].

Las empresas Fintech se dedican a intermediar en el mundo de las finanzas en múltiples aspectos, en las transferencias de dinero, en los préstamos, en las compras y ventas de títulos o en el asesoramiento financiero y de inversiones; es por ello que las iniciativas Fintech enfocan sus esfuerzos a:

- Disponer de la comodidad de una aplicación o la atractiva interfaz similar a una red social para brindar a los clientes productos anteriormente reservados a la banca tradicional, utilizando los dispositivos inteligentes como intermediadores.
- Agregar valor a los servicios financieros actuales, personalizando el trato y haciéndolo más detallado y eficaz.
- Ofrecer productos financieros totalmente en línea, sin necesidad de desplazamiento del usuario, eliminando los recaudos físicos.
- Reducir de modo eficiente los costos de servicios actuales.
- Democratizar el acceso a una mayor cantidad de servicios financieros, así como favorecer una mayor transparencia en la oferta financiera.
- Ampliar el alcance de los servicios financieros a más sectores de la población.

- Utilizar la tecnología para dar mejores soluciones a los servicios financieros.

3. NFC

Near Field Communication (NFC) o comunicación de campo cercano, es una forma de comunicación sin contacto entre dispositivos que cuentan con un chip de esta tecnología, actualmente la mayor parte de teléfonos inteligentes o tabletas lo contienen [2]. El chip actúa como un enlace inalámbrico, el cual se activa al entrar en contacto con otro, por lo que no consume energía ni requiere una fuente de poder como una batería. Adicionalmente, cada etiqueta NFC posee un identificador único, basado en el estándar ISO 14443 [3].

La comunicación sin contacto permite al usuario con un dispositivo NFC enviar información sin necesidad de tocar los dispositivos juntos o pasar por varios pasos estableciendo una conexión rápida y conveniente.

Según la Organización Near Field Communication "la tecnología detrás de NFC permite a un dispositivo, conocido como lector, interrogador o dispositivo activo, crear una corriente de radiofrecuencia que se comunica con otro dispositivo compatible con NFC o una pequeña etiqueta NFC que contiene la información que el lector desea. Los dispositivos pasivos, como la etiqueta NFC en carteles inteligentes, almacenan información y se comunican con el lector, pero no leen activamente otros dispositivos. La comunicación peer-to-peer a través de dos dispositivos activos también es una posibilidad con NFC. Esto permite a ambos dispositivos enviar y recibir información" [4].

4. IoT

Con la llegada de Internet al mundo, la evolución ha sido constante y cada vez a mayor velocidad, desde los módems de 56KB y el internet dial-up hasta el día de hoy, donde es posible una conexión forma móvil desde cualquier lugar y a velocidades nunca antes pensadas.

Actualmente, es posible conectar a Internet teléfonos móviles, impresoras, Smart TV, cámaras vía IP, GPS, entre otros; pero, existe una nueva era llama IoT, siglas que provienen de la palabra Internet of Things o en español Internet de las cosas.

El Internet de las cosas se define como la consolidación a través de la red de redes de una "red" que aloja una gran multitud de objetos o dispositivos, es decir, poder tener conectada a esta todas las cosas de este mundo como podrían ser vehículos, electrodomésticos, dispositivos mecánicos, o simplemente objetos tales como calzado, muebles, maletas, dispositivos de medición, bio-sensores, o cualquier objeto [5]. En resumen, es un sistema de dispositivos de computación interrelacionados, máquinas mecánicas y digitales, objetos, animales o personas que tienen identificadores únicos y la capacidad de transferir datos a través de una red, sin requerir de interacciones humano a humano o humano a computadora [6].

IoT ha evolucionado desde la convergencia de tecnologías inalámbricas, sistemas micro-electromecánicos (MEMS), microservicios e internet. La convergencia ha ayudado a derribar las paredes de silos entre la tecnología operativa (OT) y la tecnología de la información (TI), permitiendo que los datos

no estructurados generados por máquinas sean analizados para obtener información que impulse mejoras [6]. Las capacidades que actualmente se están desarrollando en el área de IoT son [7]:

- Comunicación y cooperación: los objetos tienen la capacidad de conectarse a los servicios de Internet y/o entre sí; intercambiando, actualizando datos entre ellos y estableciendo comunicación con servidores.
- Capacidad de direccionamiento: esta clase de dispositivos pueden ser configurados y localizados desde cualquier lugar de la red.
- Identificación: los objetos pueden ser identificados mediante tecnologías tales como RFID (Radio Frecuency Identification), NFC (Near Field Communication), códigos de barra de lectura óptica, entre otras formas.
- Localización: tienen en todo momento conocimiento sobre su ubicación física, pudiendo saber dónde se encuentra en todo momento.
- Actuación: determinados objetos son capaces de manipular su entorno.

5. Arquitectura de software

5.1. Microservicios

Según Martin Fowler y James Lewis "La arquitectura de microservicio es un enfoque para desarrollar una sola aplicación como un conjunto de pequeños servicios, cada uno ejecutándose en su propio proceso y comunicándose con mecanismos ligeros, a menudo una API de recursos HTTP. Estos servicios se basan en capacidades empresariales y se pueden implementar de forma independiente

mediante maquinaria de despliegue completamente automatizada. Existe un mínimo de gestión centralizada de estos servicios, que puede escribirse en diferentes lenguajes de programación y utilizar diferentes tecnologías de almacenamiento de datos" [8].

Los pequeños servicios son definidos en función a las capacidades y necesidades de cada empresa, y su implementación garantiza despliegue completamente automatizado, siendo su gestión es centralizada y abierta tanto para escribirse en diversos lenguajes de programación como utilizar diferentes tecnologías de almacenamiento de datos [9].

6. AWS

Amazon Web Services (AWS) o en su término en español "Servicios Web Amazon" son un conjunto de productos globales basados en la nube, incluidas aplicaciones de informática, almacenamiento, bases de datos, análisis, redes, móviles, herramientas para desarrolladores, herramientas de administración, IoT, seguridad y empresariales. Estos servicios ayudan a las empresas a avanzar con mayor rapidez, reducir los costos de TI y escalar; respaldando una amplia variedad de cargas de trabajo, como las aplicaciones web y móviles, el desarrollo de juegos, el almacenamiento y procesamiento de datos, el almacenamiento en general, el archivado, entre otras [10].

7. Microservicios AWS

7.1. AWS Lambda

Es un servicio de informática sin servidores que ejecuta el código como respuesta a eventos y administra automáticamente los recursos informáticos subyacentes. Lambda ejecuta el código en una infraestructura informática de alta disponibilidad y ejecuta la administración integral de los recursos informáticos, incluido el mantenimiento del servidor y del sistema operativo, el aprovisionamiento de capacidad y el escalado automático, la implementación de parches de seguridad y código, así como la monitorización y los registros. Lo único que tiene que hacer es proporcionar el código [11].

Con Lambda, puede ejecutar código para casi cualquier tipo de aplicación o servicio back-end) y todo sin administrar nada, solo cargar previamente el código. Lambda se encargará de todo lo necesario para ejecutar y escalar el código con alta disponibilidad. Puede configurar el código para que se active automáticamente desde otros servicios de AWS o puede llamarlo directamente desde cualquier aplicación web o móvil [12].

7.2. AWS IoT

Es una plataforma de nube administrada que permite a los dispositivos conectados interactuar con facilidad y seguridad con las aplicaciones en la nube y otros dispositivos. AWS IoT admite

miles de millones de dispositivos y billones de mensajes, y es capaz de procesar y enrrutar dichos mensajes a puntos de enlace de AWS y a otros dispositivos de manera fiable y segura. Con AWS IoT, las aplicaciones pueden realizar un seguimiento de todos los dispositivos y comunicarse con ellos en todo momento, incluso cuando no están conectados [13].

AWS IoT facilita la utilización de servicios de AWS como AWS Lambda, Amazon Kinesis, Amazon S3, Amazon Machine Learning y Amazon DynamoDB, Amazon CloudWatch, AWS CloudTrail y Amazon Elasticsearch Service con la integración incorporada de Kibana para crear aplicaciones de IoT que recopilen, procesen, analicen y utilicen datos generados por dispositivos conectados sin necesidad de tener que administrar ninguna infraestructura [14].

7.3. AWS IoT Shadows

Una sombra de objetos (Things) o sombra de dispositivo, es un documento JSON que se utiliza para almacenar y recuperar información de estado actual de una cosa (dispositivo, aplicación, etc.). El servicio Thing Shadows, mantiene una sombra de objetos para cada cosa que conecte a AWS IoT utilizando sombras de objetos para obtener y establecer el estado de una cosa sobre MQTT o HTTP, independientemente de si la cosa está conectada

a Internet. Cada sombra de la cosa se identifica únicamente con su nombre [15]

7.4. AWS SNS

Amazon Simple Notification Service (Amazon SNS) es un servicio web que facilita las tareas de configuración, utilización y envío de notificaciones desde la nube. Ofrece a los desarrolladores una funcionalidad muy escalable, flexible y rentable para publicar mensajes desde una aplicación y entregarlos inmediatamente a suscriptores o a otras aplicaciones [16]. Amazon SNS sigue el paradigma de mensajería "publicación-suscripción" (pub-sub), con notificaciones que se envían a los clientes con un mecanismo "push" que elimina la necesidad de verificar o "sondear" regularmente información o actualizaciones nuevas [17].

8. Notificaciones Push

El servicio de notificaciones push es un sistema que te permite una comunicación instantánea con tu cliente a través de la entrega de mensajes instantáneos de hasta 255 caracteres hacia aplicaciones o usuarios [18].

9. Google Firebase

Consiste en una plataforma disponible para diferentes plataformas, la cual provee un API para guardar y sincronizar datos en la

nube en tiempo real. Sus características fundamentales están divididas en varios grupos, ellos son [19]:

- Analíticas: provee una solución gratuita para tener todo tipo de medidas (hasta 500 tipos de eventos), para gestionarlo todo desde un único panel.
- Desarrollo: permite construir mejores apps, permitiendo delegar determinadas operaciones en Firebase, para poder ahorrar tiempo, evitar bugs y obtener un aceptable nivel de calidad. Entre sus características destacan el almacenamiento, testeo, configuración remota, mensajería en la nube o autenticación, entre otras.
- Crecimiento: permite gestionar los usuarios de las aplicaciones, y además captar nuevos. Para ello dispondremos de funcionalidades como las de invitaciones, indexación o notificaciones.
- Monetización a través a AdMob.

10. POS

En términos internos de CityWallet, un POS (Point of Sell), o punto de venta en español, son los puntos instalados en las entradas de los estacionamientos en donde los usuarios pueden presentar su calcomanía de CityWallet para generar una transacción de check-in, o de check-out dependiendo del caso

10.1. Check-in

Tipo de transacción que se genera en un POS de entrada en un estacionamiento. Esta transacción representa una entrada al establecimiento al momento en que se presenta la calcomanía en el punto de CityWallet y este procesa exitosamente.

10.2. Check-out

Tipo de transacción que se genera en un POS de salida en un estacionamiento. Esta transacción representa una salida del establecimiento al momento en que se presenta la calcomanía en el punto de CityWallet y este procesa exitosamente.

11. Geateway

En términos internos de CityWallet, un Gateway es el software encargado de la generación de tokens (cadena de caracteres que se genera cada vez que se realiza una transacción) para las transacciones de estacionamiento de CityWallet, validación de usuarios, validación de tokens y registro de transacciones en base de datos local. También es el encargado de firmar los tokens que genere y de verificar los que ya se hayan generado.

12. Firma Digital

Es un método criptográfico que permite que un mensaje firmado digitalmente se le pueda identificar a la entidad de origen (autenticación), y confirmar que el mismo no ha sufrido alteraciones desde el momento en que fue firmado por el emisor (integridad). [20]

12.1. DSA

Es un estándar de criptografía de clave asimétrica propuesto por el Instituto Nacional de Normas y Tecnología de los Estados Unidos y sirve para firmar información. [21]

12.2. SHA

El SHA (Secure Hash Algorithm, Algoritmo de Hash Seguro) es una familia de funciones hash de cifrado publicadas por el Instituto Nacional de Estándares y Tecnología (NIST). En la actualidad existen cuatro versiones de este algoritmo las cuales son SHA-0, SHA-1, SHA-4 SHA-3. En caso del Gateway la versión usada es la SHA-1, la cual es una función de hash de 160 bits que se asemeja al algoritmo MD5, con DSA. [22]

CAPÍTULO III - MARCO METODOLÓGICO

1. Método de Desarrollo

Se eligió Scrum como marco de trabajo con el objetivo de garantizar que el desarrollo produjera incremento de forma constante. Scrum no es una metodología, sino un conjunto de reglas basadas en el empirismo, añadiendo un enfoque heurístico; en tal sentido, los equipos que trabajan con Scrum identifican el problema, lo definen, exploran estrategias viables para resolverlo y luego avanzan con dichas estrategias hasta lograr la solución. El proceso heurístico culmina evaluando el efecto de las actividades e incluye los aprendizajes para futuras planificaciones [23].

En Scrum los requerimientos y las soluciones evolucionan a través de la colaboración entre equipos auto organizados y multifuncionales [24]. Scrum forma parte de la ingeniería de desarrollo ágil, la cual empodera a través liderazgo colaborativo. De esta forma el equipo es quien identifica cómo abordar los retos que presenta tanto el desarrollo, como los clientes, el mercado o cualquier otra variable que pudiera afectar el producto final.

Adicionalmente, Scrum se enfoca en el incremento, lo que significa que el producto crece al final de cada período de trabajo. El objetivo del equipo de desarrollo es aumentar semana tras semana la cantidad de funcionalidades de la aplicación [25]. Los resultados de utilizar Scrum son ideales para las startups como CityWallet, pues la continua inspección del trabajo permite reducir el costo del retraso en la disponibilidad del producto. Consecuentemente, el proyecto debe ser gerenciado y/o revisado con constancia para que se cumplan

los tres pilares del framework: inspección, transparencia y adaptación. De esta forma, el trabajo siempre podrá ser adaptado para alcanzar los objetivos a tiempo y no retrasar la salida del producto al mercado.

1.1. Artefactos

Para facilitar la inspección de las tareas, Scrum contempla una serie de elementos llamados artefactos que dan orden y transparencia a la carga de trabajo de cada miembro del equipo; así la revisión se simplifica y cualquier adaptación necesaria se asimila con facilidad [26].

Los artefactos son:

- Backlog de Producto: es una lista ordenada de todo lo que el producto podría necesitar [27]. El Dueño del Producto es quien construye esta lista apoyándose en la inteligencia obtenida a través de distintas fuentes como los clientes y sus requerimientos; el mercado y sus necesidades; la empresa y sus objetivos, entre otros. Esta lista de tareas y requerimientos es dinámica, es decir, evoluciona a la par de los stakeholders previamente mencionados y es flexible a cambios. El equipo de desarrollo se planifica en función a lo expresado en el backlog de producto.
- Backlog del Sprint: es una selección de los elementos listados en el Backlog de Producto. Esta carga de trabajo se organiza tomando en cuenta qué tareas deben hacerse y completarse para que al final del período el equipo pueda añadir una nueva funcionalidad el producto. En este backlog se puede visualizar

con facilidad el trabajo que el equipo de desarrollo hace para alcanzar sus objetivos [28].

- El Incremento: pudiera decirse que este artefacto concreta los resultados del trabajo del Equipo de Desarrollo, ya que en cada al final de cada sprint se sumariza las partes del producto que fueron completamente construidas, completadas y que ya están operativas [29].

1.2. Eventos

Para darle ritmo al trabajo del equipo, Scrum celebra ciertos eventos que permiten aplicar los pilares de Scrum: inspección, transparencia y adaptación. La idea de estos eventos es evitar reuniones largas y tediosas que hagan perder el tiempo [30]. Los eventos se esbozan a continuación:

- Sprint: es el período de tiempo que enmarca el desarrollo del incremento de producto. Cada sprint tiene objetivos claramente definidos y es responsabilidad del Equipo de Desarrollo definir y ejecutar un plan para alcanzarlos. Un sprint empieza cuando otro termina. La ventaja de organizar los esfuerzos en períodos radica la previsibilidad, pues el proceso asegura la inspección del progreso hacia el objetivo del sprint [31].
- Planificación del Sprint: durante este evento, el Equipo de Desarrollo y el Maestro Scrum definen cuáles serán los incrementales del producto en ese período. Los requerimientos se

arrastran desde el Backlog de Producto hasta el Backlog del Sprint, en donde cada tarea es descrita con las especificaciones que deben tener para considerarse como terminadas. En este evento participan activamente tanto el equipo de desarrollo como el Maestro Scrum, siendo este último quien debe asegurar que todos los miembros comprendan lo que se espera alcanzar al final del sprint [32].

- Scrum Diario: es un proceso de inspección diario no mayor a 15 minutos, diseñado para sincronizar las actividades del equipo de desarrollo y determinar si existe algún bloqueo que adaptar [33].
- Revisión del Sprint: al culminar el sprint, el Equipo de Desarrollo se reúne con el Maestro del Scrum y con el Dueño del Producto para cuantificar el incremento del mismo, es decir, se revisa cuáles tareas fueron culminadas y cuáles no. Está reunión arroja como resultado el estatus de del producto funcional [34].
- Retrospectiva del Sprint: Scrum no sólo impulsa el incremento del producto, sino también el incremento de la eficiencia del desarrollo. Por ende, al final de cada período se discuten los aprendizajes y las fallas con el objetivo de incorporar las mejores prácticas en el próximo sprint.

1.3. Roles

- El Dueño del Producto: es el único responsable por manejar el backlog del producto, consecuentemente todos los requerimientos

y tareas son definidos y procesados por su persona. El dueño del producto debe asegurar que los requerimientos estén ordenados en una prioridad que optimice el desarrollo; además, la presentación de las tareas debe estar expresada de forma que todos los miembros del equipo puedan comprenderlas con facilidad [35]. El Ing. Ramón Ginéz fungió como Dueño del Producto para el desarrollo presentado en esta tesis.

- El Maestro Scrum: hace comprender los objetivos del sprint y asegura su ejecución, para ello, el Maestro Scrum se encarga de guiar al equipo de desarrollo en la aplicación del marco de Scrum. Por otro lado, este rol está al servicio del Equipo de Desarrollo, pues el Maestro Scrum es responsable de asegurar el flujo del trabajo y de sortear bloqueos e impedimentos [36]. El Ing. Félix Fernández desempeñó el rol de Maestro Scrum en cada planificación estructurada en CityWallet.
- El Equipo de Desarrollo: son los responsables de generar el incremento del producto. Este equipo se compone de individuos con distintos tipos de habilidades específicas que se complementan entre sí. La característica más importante de este grupo de personas es que manejan su propia organización y ello les aporta una sinergia independiente de cualquier otro elemento de la organización [37].

Internamente en CityWallet se maneja como marco de trabajo SCRUM, con la modificación de que una vez obtenido las historias de usuario se procede a establecer las épicas del proyecto. Se define como Épica a una gran carga de trabajo que puede agrupar una o varias historias de usuario, una épica puede durar uno o más Sprint. A su vez cada épica tiene hitos, los cuales son puntos en el proceso de desarrollo del proyecto en donde se logra culminar una o varias actividades de forma exitosa. Dicho lo anterior entonces se tienen Épicas, las cuales se dividen en hitos, los cuales a su vez se subdividen en actividades. Cada Épica puede cubrir una o más historias de usuario. [38]

Teniendo en cuenta lo explicado anteriormente, en el mapa a continuación se expone la lista de Sprints realizados en este trabajo instrumental de grado junto con los objetivos y aportes, funcionales y tecnológicos que abarcó cada uno:

- Sprint 0
  - Desarrollo de la segunda versión los Puntos de Servicio (POS)
  - Integración de los POS con la plataforma AWS IoT.
  - Desarrollar la aplicación móvil para monitoreo de los POS integrados con IoT.
  - Integrar el servicio de notificaciones AWS SNS (Simple Notification Service) para el manejo de notificaciones entre dispositivos.

- Sprint 1
  - Desarrollo de la segunda versión los Puntos de Servicio (POS).
- Sprint 2
  - Desarrollo de la segunda versión los Puntos de Servicio. (POS)
  - Integración de los POS con la plataforma AWS IoT.
  - Integrar los web services necesarios de CityWallet con la nueva versión de los POS.
- Sprint 3
  - Integración de los POS con la plataforma AWS IoT.
- Sprint 4
  - Integración de los POS con la plataforma AWS IoT.
- Sprint 5
  - Desarrollar la aplicación móvil para monitoreo de los POS integrados con IoT.
- Sprint 6
  - Desarrollar la aplicación móvil para monitoreo de los POS integrados con IoT.
- Sprint 7
  - Integrar la aplicación móvil para monitoreo de los POS con la plataforma AWS IoT y SNS.

  - Desarrollar un microservicio para el manejo de las notificaciones push entre el POS y las aplicaciones clientes de CityWallet.
- Sprint 8
  - Integrar la aplicación móvil para monitoreo de los POS con la plataforma AWS IoT y SNS.
  - Desarrollar un microservicio para el manejo de las notificaciones push entre el POS y las aplicaciones clientes de CityWallet.
- Sprint 9
  - Integrar la plataforma AWS IoT con las aplicaciones existentes de CityWallet

CAPÍTULO IV - DESARROLLO

1. Introducción

El desarrollo del presente Trabajo Instrumental de Grado inicia, siguiendo el framework de Scrum, con una primera fase llamada "Sprint 0". Este sprint tiene como objetivo principal listar los requerimientos e historias de usuarios las cuales serán la base para el desarrollo del producto en los siguientes Sprints. Para esto fue necesario reuniones con el gerente de producto y el arquitecto de CityWallet para logra identificar los procesos que serán cubiertos para dar solución a los objetivos específicos del presente trabajo.

Posterior al listado de los requerimientos e historias de usuarios, se procedió a definir las Épicas e hitos que conforman cada Sprint, y con ellos las tareas que se requieren para lograr completar la Épica. Cabe destacar que un Sprint puede tener más de una Épica dependiendo de a qué objetivo se esté enfocando.

Para CityWallet, los Sprints tienen una duración de 2 semanas (10 días de trabajo), y se consideró un esfuerzo base de 8 horas por día. El único Sprint distinto a los demás fue el Sprint 0 el cual solo duró una semana. En total, resultaron 9 Sprints de desarrollo en los cuales se cubrieron todos los objetivos específicos.

2. Sprints

2.1. SPRINT 0

Siguiendo el framework de Scrum, el objetivo de este Sprint fue de servir como semana inicial para reuniones con el Dueño del Producto con el fin de obtener un Backlog de Producto e historias de usuarios referentes al desarrollo de este trabajo de grado.

En la siguiente tabla (Tabla 1) se puede observar el Product Backlog resultante. Los actores a que intervienen el presente trabajo son:

- Usuario: Portador de la calcomanía, el cual visitará los distintos estacionamientos que posean CityWallet y realiza transacciones presentando la calcomanía en los POS.
- Administrador de CityWallet: Representante de CityWallet, que tiene como labor principal el monitoreo del funcionamiento de la plataforma de CityWallet en los distintos estacionamientos, y servir de soporte ante una eventual contingencia.
- Administrador del Establecimiento: Encargado(s) del estacionamiento, los cuales deben tener acceso a métricas e información general de la plataforma CityWallet en su establecimiento.

Tabla 1. Backlog de Producto

Fuente [Elaboración Propia]

ID	ACTOR	HISTORIA	ESTADO	VALOR	PRIORIDAD
POSv2-1	Usuario	Como usuario, quiero poder realizar una transacción (check-in, check-out)	Listo	6	Alta
POSv2-2		Como usuario, quiero que mi transacción sea segura	Listo	6	Alta
POSv2-3		Como usuario, quiero poder saber si la transacción fue exitosa o no (si hubo error cual fue)	Listo	5	Media
POSv2-4	Administrador de CityWallet	Como administrador de CityWallet, quiero que las transacciones realizadas se alojen localmente en el POS para posterior sincronización	Listo	5	Alta
POSv2-5		Como administrador de CityWallet, quiero que el POS genere un token único por transacción para luego ser escrito en la memoria de la calcomanía	Listo	6	Media
POSv2-6		Como administrador de CityWallet, quiero que el POS sea capaz de firmar el token creado para cada transacción	Listo	6	Alta
POSv2-7		Como administrador de CityWallet, quiero que el POS sea capaz de validar el token creado para cada transacción	Listo	5	Alta
POSv2-8		Como administrador de CityWallet, quiero poder verificar si el token fue escrito en el cashout device	Listo	4	Media
POSv2-9		Como administrador de CityWallet, quiero poder verificar si el token del cashout device fue autenticado	Listo	3	Baja

POSv2-10		Como administrador de CityWallet, quiero que el POS sea capaz de conectarse a un Gateway remoto aun teniendo Gateway propio	**Listo**	**2**	**Alta**
POSv2-11		Como administrador de CityWallet, quiero que el POS se conecte a la plataforma de AWS IoT	**Listo**	**5**	**Alta**
POSv2-12		Como administrador de CityWallet, quiero poder consultar un listado de POS actualmente registrados en CityWallet y sus estados	Listo	**5**	**Alta**
POSv2-13		Como administrador de CityWallet, quiero que el POS sea capaz de publicar sus estados en la plataforma AWS IoT	**Listo**	**6**	**Alta**
POSv2-14		Como administrador de CityWallet, quiero poder tener acceso al último estado publicado del POS incluso cuando el mismo se encuentra sin conexión (Shadows)	**Listo**	**5**	Media
POSv2-15		Como administrador de CityWallet, quiero poder realizar acciones remotas sobre el POS	**Listo**	**3**	Media
POSv2-16		Como administrador de CityWallet, quiero poder recibir notificaciones con información referente al estado del POS en caso de que este cambie	**Listo**	**4**	Media
POSv2-17	Administrador de Estacionamiento	Como Administrador, quiero que el POS sea capaz de validar a los usuarios	**Listo**	**6**	**Alta**
POSv2-18		Como Administrador, quiero que las transacciones realizadas en mi comercio se sincronicen con la plataforma de	**Listo**	**6**	**Alta**

		CityWallet			
POSv2-19		Como Administrador, quiero tener la información de los usuarios actualizada para validarlos	Listo	5	**Media**
POSv2-20		Como administrador, quiero poder consultar los estados de todos los POS registrados de CityWallet a través de una app o portal web	Listo	5	**Alta**
POSv2-21		Como administrador, quiero poder recibir notificaciones con información referente al estado del POS en caso de que este cambie	Listo	4	Media

Como se puede observar en la Tabla 1, se obtuvo un total de 21 historias de usuarios que fueron cubiertas durante los 9 Sprints anteriormente mencionados. En la tabla, aparte de las columnas de ID, actor y estado, se definen: valor, que es la importancia de la historia de usuario para el Dueño del Producto; Prioridad, preferencia que se tiene hacia una historia comparada con las demás; y Sprint, que es una referencia al número de sprint en que le dé abocó a realizar esa historia de usuario.

2.2. *SPRINT 1*

Objetivos cubiertos en este Sprint: Desarrollo de la segunda versión los Puntos de Servicio (POS).

Para este sprint se tuvo como base para el desarrollo el Sprint Backlog Tabla 2 del Apéndice 2, el cual cuenta con las siguientes actividades:

[E1H1A1] Integración de lógica de Gateway migrada de Java a Python con código de Python existente: en la versión inicial (1.0) del POS, la arquitectura planteada es como se observa en la figura 4 del Apéndice 1.

El POS se encargaba de leer las calcomanías de los usuarios, obtener el tag ID de dicha calcomanía para luego enviar una solicitud al gateway con ese identificador como parámetro principal. El gateway realizaba las validaciones necesarias y enviaba una respuesta al finalizar el proceso.

La lógica del Gateway, por requerimientos de la empresa, tenía que ser migrada a Python para así fusionar el POS con el Gateway. Para lograr esto se necesitó desarrollar las tareas **E1H2A1, E1H3A1, E1H4A1, E1H4A2, E1H5A1,** y luego integrarlas con el código existente del POS, generando así un punto de venta de CityWallet independiente de un Gateway centralizado, además, se simplifica la arquitectura y el proceso de generación de una transacción se vería modificado para ser un código más ligero y mantenible.

Como se observa en la figura 5 del Apéndice 1, ahora el POS es el encargado de todo el proceso de lectura de la calcomanía, validación del usuario y registro de la transacción, a diferencia de la versión 1.0, la cual solo se encargaba de la lectura de la calcomanía.

[E1H2A1] Migración de los métodos de Java a Python para el manejo del perfil del usuario de base de datos local: Los usuarios de plataforma de CityWallet poseen un perfil único al cual tienen su calcomanía asociada. Estos perfiles, además de almacenarse en la base de datos central de CityWallet, también se alojan localmente en los POS. Los perfiles son sincronizados a través de un cron que ejecuta un shell script para así mantener la información actualizada de los usuarios en cada punto de venta.

La base de datos local está basada en el sistema de base de datos orientado a documentos de MongoDB, la cual es una base de datos no relacional orientada a documentos que aloja la siguiente información del usuario:

- AccountId: identificador único que une al perfil del usuario con la cuenta del mismo.
- CreatedDate: fecha de creación del perfil.
- DocumentId: documento de identificación del usuario. En este caso es el número de cedula de identidad
- Metadata: objeto en forma de JSON que contiene el nombre, apellido del usuario y PIN y PAK de la calcomanía del usuario.

- ProfileId: identificador único por perfil de usuario
- ProfileStatus: estado actual del perfil, puede ser ACTIVE o INACTIVE, activo o inactivo respectivamente.
- ProfileType: Tipo del perfil del usuario, en este caso PARKING_CLIENT, el cual denota que el usuario es cliente de estacionamiento.
- TransactionToolId: identificador único que une al perfil del usuario con la el transaction tool del mismo.
- UpdatedDate: última fecha de modificación del perfil.
- UserExternalKey: TagId de la calcomanía del usuario

En la figura 6 del Apéndice 1, se puede observar un ejemplo de un perfil de usuario guardado en la base de datos local del POS, en forma de JSON.

La migración de Java a Python referentes al manejo del perfil del usuario abarcó los siguientes métodos:

- isValidUser: En este método se extrae de la base de datos el perfil del usuario que presentó la calcomanía en el lector, y se encarga de verificar que el usuario posea el estatus de ACTIVO, de lo contrario el usuario no es válido.
- userHasPlan: este método se encarga de verificar si el usuario tiene un plan de CityWallet asociado a tu cuenta para el ingreso al estacionamiento, de ser así, se le reduce un uso al número de

usos disponibles en su plan. De lo contrario se consultará contra el saldo disponible del usuario para determinar si puede entrar o no al estacionamiento.

En las figuras 7 y 8 del Apéndice 1 se puede observar el método isValidUser desarrollado en Java y Python respectivamente, así como en las figuras 9 y 10 del Apéndice 1 se muestran los códigos de los métodos userHasPlan en Java y Python respectivamente.

[E1H3A1] Migración de método de Java a Python de creación de token: Cada transacción registrada en la plataforma de CityWallet genera un token único que se guarda en la memoria de la calcomanía del usuario. Este token está conformado por los siguientes campos:

- TagId: serial alfanumérico que sirve como identificador único para cada calcomanía. Viene establecido por defecto en la memoria de cara calcomanía.
- PosId: serial alfanumérico que sirve de identificador único para cada POS.
- Secuencia: numero hexadecimal que se autoincrementa cada vez que un nuevo token es generado para la calcomanía. Una vez alcanzado el número 16 la secuencia se reinicia.
- Versión: número que identifica la versión del token que la calcomanía está usando.
- Fecha: fecha de creación del token.
- Firma: Firma digital generada por el Gateway.

Cada uno de estos valores separados por coma conforman el token que se genera cada vez que se procesa una transacción exitosa en el POS.

En la figura 11 del Apéndice 1, podemos ver un ejemplo de un token generado por un POS de entrada. La sección de color rojo representa el TagId de la calcomanía, la sección en naranja el PosId, la sección en amarillo la secuencia, la sección en verde la versión, la sección en azul la fecha y la sección en morado la firma. Esta cadena de caracteres es la que posteriormente de ser creada se escribirá en la memoria de la calcomanía. Cabe destacar que la cadena de caracteres no puede ser mayor a 130 bytes.

La migración de Java a Python referente a la creación del token abarcó los siguientes métodos:

- CreateToken: este método es el encargado de generar la cadena de caracteres que conformará el token que será escrito en la memoria de la calcomanía.

En las figuras 12 y 13 del Apéndice 1 se puede observar el método de creación de token en Java y Python respectivamente.

[E1H4A1] Migración de método de Java a Python de Validación del tipo de transacción: Cada vez que un usuario presenta una calcomanía en el lector del POS es necesario identificar qué tipo de transacción es la que se va a procesar. Esto se determina mediante el token que está escrito en la memoria de la calcomanía. Si un POS es de

entrada la única forma que registre una transacción exitosamente es que el token de la calcomanía provenga de un POS de salida, o no tenga token registrado aún (calcomanía sin uso). Para el caso de una transacción de salida, el token debió haber sido generado por un POS de entrada, y este pertenecer al mismo establecimiento que el POS de salida donde se está presentando la calcomanía. En cualquiera de los dos casos, si no se cumplen las condiciones, el POS se encarga de notificar al usuario mediante un audio el error correspondiente.

La migración de Java a Python referentes a la validación del tipo de transacción abarcó los siguientes métodos:

- isValidTransaction: Este método se encarga de validar si la transacción que va a realizar el usuario en el punto de venta es válida, es decir, si presenta la calcomanía en un POS de salida, esta función valida que tenga una entrada registrada, o si el usuario presenta la calcomanía en un POS de entrada, esta función verifica que el usuario no tenga ya una entrada registrada sin salida en el mismo estacionamiento.

En las figuras 14 y 15 del Apéndice 1, se puede observar el método de isValidTransaction desarrollado en Java y luego migrado a Python, respectivamente.

[E1H4A2] Migración de método de Java a Python de registro de una transacción en base de datos local: una vez generado el token, y escrito en memoria de la calcomanía se procede a registrar la

transacción en la base de datos local. Para esta tarea se usó la librería de mongo para Python PyMongo. Las transacciones registradas contienen la siguiente información:

- transaction_id: identificador único de la transacción.
- pos_id: identificador único del POS en donde se registró la transacción.
- Transaction_type: tipo de la transacción registrada, es decir, si es de entrada o de salida.
- date: fecha de la transacción.
- token: token perteneciente a la transacción.

En la figura 16 del Apéndice 1, podemos observar un ejemplo de una transacción guardada en la base de datos local

La migración de Java a Python referentes al registro de una transacción abarcó los siguientes métodos:

- CreateParkingTransaction: este método se encarga de registrar en la base de datos local del POS, las transacciones hechas por los usuarios al presentar la calcomanía en los puntos de venta.

En las figuras 17 y 18 se puede observar el método de CreateParkingTransaction desarrollado en Java y luego migrado a Python, respectivamente.

[E1H5A1] Registrar en logs el proceso completo del POS: esta tarea consistió en llamar a la escritura de un archivo de logs al inicio, al

final, y en los casos de excepciones de cada método. Esto con el objetivo de crear un archivo que contenga todo el proceso de creación de una transacción para así poder hacer trazable cualquier potencia error que pueda surgir.

El formato de la cada línea de log es el siguiente, en el mismo orden y separados por guiones:

- Fecha de escritura en log.
- Tipo de registro: Cada registro o línea escrita en logs puede ser de un tipo de los siguientes: INFO, si el registro a escribir es solo informativo; ERROR, si el registro a escribir es un error dado por el POS.
- Nombre del módulo al que pertenece la clase que se está ejecutando.
- Nombre de la función que se está ejecutando.
- Mensaje

En la figura 19 del Apéndice 1, podemos observar el contenido del archivo de los del POS. En este caso se muestran registros del tipo INFO.

2.3. *SPRINT 2*

Objetivos cubiertos en este Sprint: Desarrollo de la segunda versión los Puntos de Servicio (POS); Integración de los POS con la plataforma AWS IoT

Para este sprint se tuvo como base para el desarrollo el Sprint Backlog, Tabla 3 del Apéndice 2, el cual cuenta con las siguientes actividades:

[E2H1A1] Extracción de los métodos de firma y verificación de firmas del Gateway a un Java Jar aparte para su consumo individual (Token Signer): Cada token generado por un POS debe estar formado por el mismo para luego poder verificar que el token escrito en cada calcomanía lo genero en efecto, un POS perteneciente a la plataforma de CityWallet.

Para lograr esto el último campo del token es una firma DSA generada por el POS. EL objetivo principal de esta tarea era migrar el método de Java del Gateway a Python para integrarlo con el código del POS. Sin embargo, un requerimiento de CityWallet durante la realización de este trabajo instrumentar de grado, era que el POS 2.0 iba a convivir en producción junto con el 1.0. Es decir, dependiendo de las limitaciones del estacionamiento, CityWallet puede decidir que versión del POS instalar, si una con el gateway centralizado (1.0) o una con el Gateway integrado en el POS (2.0).

A la hora de migrar el método de firma de token se instó por usar librerías de firmas para Python. El inconveniente que surgió fue que los tokens formados por un POS 2.0 no podían ser verificados por un POS 1.0, y viceversa. Creando un conflicto de versiones que dejaría al usuario sin poder entrar a ciertos estacionamientos (dependiendo de con qué versión se le haya firmado el token).

Como solución a eso se planteó generar un JAR solo con los métodos de firma y verificación de token en Java para este ser consumido por el POS (desde Python) con una librería "puente" entre ambos. Los métodos de firma de token fueron los únicos métodos de Java pertenecientes al Gateway que no fueron migrados.

[E2H2A1] Integración del código del POS en Python con la librería Py4j para generar un puente entre Python y la librería de firma de tokens Token Signer: LA librería Py4J permite que los programas Python tengan acceso dinámico a objetos Java en una Máquina Virtual Java. Los métodos se llaman como si los objetos de Java residían en el intérprete de Python y las colecciones de Java se pueden acceder a través de métodos de colección estándar de Python. Py4J también permite a los programas Java llamar a objetos Python.

Con esta librería se crea un puente entre el código del POS en Python y la librería de firma de tokens en Java.

[E3H1A1] Creación del Thing en la plataforma de AWS IoT: Para poder conectar un POS con la plataforma IoT de AWS es necesario

registrar el POS como un Thing o "cosa" en español en el panel de control de AWS IoT. Para eso se llevaron a cabo los siguientes pasos:

Al iniciar sesión con la cuenta de AWS, nos dirigimos a la opción de IoT, en donde encontramos con el panel principal (figura 20 del Apéndice 1).

En la opción de Registry -> Things en encuentra la opción de Create (Figura 21 del Apéndice 1).

Al clickear Create, la consola de AWS nos lleva a introducir la siguiente información (Figura 22 del Apéndice 1):

- Nombre del Thing.
- Tipo de Thing: Cada Thing puede pertenecer a un tipo específico. Cada tipo provee un máximo de tres atributos por los cuales se pueden filtrar los Things. En este caso, se creó anteriormente el tipo CWParkingPOS, el cual tiene como atributos un id que sirve como identificador único, "type" para denotar si es un POS de salida o, de entrada, y "merchant" para denotar a qué estacionamiento pertenece el POS.
- Atributos buscables: que son los valores de los atributos que se definieron anteriormente en el tipo.
- Atributos no buscables: que son atributos extras. En este caso, los POS solo poseen un solo atributo extra que es la descripción.

Al llenar toda la información y crear el Thing, ya tenemos el POS registrados dentro de la plataforma de AWS IoT (figura 23 del Apéndice 1).

[E3H1A2] Creación de los certificados del Thing: Una vez creado el Thing, se necesita descargar los certificados que se guardaran dentro del POS para que se pueda autenticar contra la plataforma de AWS IoT. Para ello se siguen los siguientes pasos:

- Se selecciona la opción de "Connect" en la barra izquierda (figura 20 del Apéndice 1).
- Luego se selecciona la primera opción donde dice configurar un dispositivo, después se selecciona "get started" (figura 24 del Apéndice 1).
- En la ventana siguiente se selecciona el sistema operativo y el lenguaje de programación que usara el Thing. En el caso de CityWallet los POS usan sistema operativo Linux/OSX y lenguaje de programación Python (figura 25 del Apéndice 1).
- ¿En el siguiente paso se selecciona "Choose an existing thing instead?" y luego se escoge el POS que se acaba de crear en la consola (Figura 26 del Apéndice 1).
- En el siguiente paso se descarga el kit de conexión (Figura 27 del Apéndice 1).

Una vez descargado el kit de conexión, se descomprime su contenido. Dentro del kit se encuentran un certificado, una clave pública

y una clave privada las cuales serán usadas por el POS para autenticarse contra la plataforma IoT.

[E3H1A3] Desarrollo del Módulo de IoT en Python: Para esta actividad fue necesario el SDK de AWS IoT para Python. El módulo de IoT se desarrolló para contener una clase con los métodos necesarios para consumir del SDK previamente mencionado. Los métodos desarrollados son:

- Connect: método por el cual el POS se autentica y crea una conexión contra la plataforma AWS IoT.
- Disconnect: este método desconecta al POS de la plataforma AWS IoT en caso de que exista una conexión abierta.
- Subscribe: siendo el nombre de un tópico el parámetro para la invocación de este método, tiene como función hacer del POS un receptor de todos los mensajes que lleguen al tópico al cual está suscrito.
- Unsubscribe: este método le permite al POS cancelar la suscripción a un tópico al que previamente se haya suscrito, con el fin de no recibir más mensajes sobre ese tópico.
- Configure Last Will: Este método establece un mensaje que será enviado automáticamente una vez que la plataforma de AWS IoT detecte que el POS no posee conexión a internet.

En la figura 28 del Apéndice 1, podemos apreciar un diagrama de interacción entre un POS y la plataforma AWS IoT. El POS al conectarse

contra la plataforma de AWS IoT, es capaz de recibir y publicar mensajes a la misma, así como obtener y actualizar su sombra.

2.4. SPRINT 3

Objetivos cubiertos en este Sprint: Integración de los POS con la plataforma AWS IoT

Para este sprint se tuvo como base para el desarrollo el Sprint Backlog, Tabla 4 del Apéndice 2, el cual cuenta con las siguientes actividades:

[E4H1A1] Prueba de conexión con plataforma AWS IoT: Para esta actividad se tomó en como base el documento de pruebas figura 29 del Apéndice 1. Cada prueba contuvo un input inicial, un output esperado y una respuesta final. La prueba se consideró exitosa si el output esperado era igual a la respuesta dada por el POS, de lo contrario la prueba se consideró fallida.

[E4H2A1] Desarrollo de métodos de obtención y actualización de Shadow en Módulo de IoT: Una de las propiedades más importantes de la plataforma de IoT es que cada Thing tiene registrado un shadow, o sombra en español, el cual es un JSON con atributos referentes al Thing que queda almacenado en la plataforma de AWS. Esto con la finalidad que cada Thing, en este caso cada POS de CityWallet, actualice periódicamente su estado contra la plataforma de IoT y esta sombra pueda ser accedida aun si el POS está desconectado del internet.

En la figura 20 del Apéndice 1, se puede observar un ejemplo del shadow inicial de un POS. En él se encuentran los siguientes atributos:

- Status: este atributo muestra si el POS esta en modo "running" o "rebooting", los cuales significan que el POS o se está ejecutando de forma normal o se está reiniciando, respectivamente.
- syncProfiles: este atributo puede tomar dos valores, true o false. Si es true el POS ejecutará un comando para actualizar la base de datos local de los usuarios de CityWallet.
- isEnabled: este atributo puede tomar dos valores, en caso de ser true el POS funciona de forma normal, de ser false el POS queda inhabilitado para su uso.
- Connected: este atributo puede tomar dos valores, "true" si el POS está conectado a internet, y "false" si no lo está.

  Esta actividad abarcó el desarrollo de los siguientes métodos:

- getThingShadow: método que obtiene el shadow del POS a través de la plataforma AWS IoT.
- updateThingShadow: método que actualiza uno o más atributos del Shadow del POS

[E4H2A2] Desarrollo del método de publicación de estados del POS: Ya teniendo un POS conectado con la plataforma de IoT, y capaz

de acceder a su shadow, es posible usar este último como medio para realizar acciones remotas sobre el POS.

Por medio del método de suscripción de IoT, un POS puede suscribirse a un tópico reservado "$aws/things/[nombre del POS]/shadow/update/delta", el cual envía un mensaje al suscriptor cada vez que el shadow del POS se ha modificado. Esto fue integrado con el módulo de IoT de modo que, si una aplicación cliente conectada a la plataforma de IoT hiciera modificaciones sobre el shadow, el POS dueño de ese shadow pueda recibir un mensaje con el delta (o cambio) que hubo sobre el JSON.

El objetivo de esta funcionalidad es que, mediante el shadow, se le puedan mandar señales al POS para que ejecute una acción definida. Dentro de las acciones definidas por CityWallet se encuentran:

- Actualización de perfiles de usuario
- Reinicio del POS
- Activación o desactivación del POS

Como vimos en la figura 30 del Apéndice 1, el estado inicial del shadow del POS tiene todos los valores establecidos en false, sin embargo, si alguno de estos valores llega a cambiar a true, por modificación de una aplicación cliente, el POS recibirá la notificación de este cambio causando que ejecute ciertos comandos dependiendo de que se le haya pedido hacer. En la figura 31 del Apéndice 1, podemos

ver como se ha actualizado el shadow del POS para que el valor del atributo “syncProfiles” cambie a true.

Una vez ejecutados los comandos necesarios en el POS, el mismo se encarga de volver a establecer el valor en false, para notificar a la aplicación cliente que en efecto fue realizada la acción.

2.5. SPRINT 4

Objetivos cubiertos en este Sprint: Integración de los POS con la plataforma AWS IoT

Para este sprint se tuvo como base para el desarrollo el Sprint Backlog, Tabla 5 del Apéndice 2, el cual cuenta con las siguientes actividades:

[E5H1A1] Prueba completas del POS: Para esta actividad se tomó en como base el documento de pruebas figura 32 del Apéndice 1. Cada prueba contuvo un input inicial, un output esperado y una respuesta final. La prueba se consideró exitosa si el output esperado era igual a la respuesta dada por el POS, de lo contrario la prueba se consideró fallida.

2.6. SPRINT 5

Objetivos cubiertos en este Sprint: Desarrollar la aplicación móvil para monitoreo de los POS integrados con IoT

Para este sprint se tuvo como base para el desarrollo el Sprint Backlog, Tabla 6 del Apéndice 2, el cual cuenta con las siguientes actividades:

Para las actividades **[E6H1A1], [E6H1A2], [E6H1A3], [E6H1A4]** Cabe destacar que para cada actividad se tomó Ionic como framework de desarrollo para aplicaciones móviles.

[E6H1A1] Desarrollo de la pantalla de inicio de sesión: La pantalla de inicio de sesión como se puede observar en la figura 33 del Apéndice 1, consiste en un formulario con tres campos:

- Una lista desplegable para que el usuario escoja que tipo de usuario es: admin (administrador de CityWallet) o merchant (administrador del estacionamiento)
- Nombre de usuario
- Contraseña

[E6H1A2] Desarrollo de la pantalla de cambio de clave: La pantalla de cambio de contraseña como se puede observar en la figura 34 del Apéndice 1, consiste en un formulario con un campo:

- Nueva contraseña
- Repetir contraseña

El usuario es llevado a esta pantalla cada vez que inicia sesión por primera vez en la aplicación. Como las cuentas de los administradores de los estacionamientos son creadas por los administradores de CityWallet, esto provee claves temporales que luego deben ser cambiadas cuando el usuario inicia sesión.

[E6H1A3] Desarrollo de la pantalla de Dashboard (lista de POS disponibles según el usuario con sesión iniciada); La pantalla de dashboard como se puede observar en la figura 35 del Apéndice 1, consiste en un listado con todos los pos disponibles del establecimiento (si el usuario es un administrador de estacionamiento) o todos los POS registrados en la plataforma de CityWallet (si el usuario es una administrador de CityWallet). Cada ítem de la lista es clickeable y redirecciona al usuario al perfil del POS seleccionado.

[E6H1A4] Desarrollo de la pantalla del perfil del POS (información general referente al estado del mismo): La pantalla de perfil del pos como se puede observar en la figura 36 del Apéndice 1, en información general sobre el estado del POS como: Nombre, estacionamiento al que pertenece, estado y tipo (si es de entrada o salida). Sumado a esto en la parte inferior se encuentra una serie de botones que se encargan de accionar el método de updatethingShadow. Cada botón modifica un atributo distinto del shadow del POS.

[E7H2A1] Creación de los Pool de Usuarios de Cognito: Para que los usuarios puedan consumir de algún recurso de AWS, es necesario

que los mismos cuenten con las credenciales necesarias. En este caso por ser una aplicación móvil, los usuarios inician sesión para obtener estas credenciales a través del servicio de Cognito de AWS.

AWS Cognito permite la creación de pool de usuarios. Tanto las aplicaciones administrativas como la aplicación que monitorea los POS, autentican los usuarios contra dichos. Cada usuario posee su nombre de usuario y su contraseña, y al iniciar sesión, Cognito se encarga de enviar las credenciales necesarias para el consumo de los métodos (ya sean de SNS o IoT) desarrollados en las aplicaciones.

[E7H2A2] Desarrollo del método para iniciar sesión contra un pool de usuarios de Cognito en la aplicación controladora de los POS: Una vez creado el pool de usuarios, se integró la aplicación con el SDK de AWS Cognito. En esta actividad se concentró el esfuerzo en desarrollar un método que recibiera como parámetros el tipo de cuenta del usuario (para saber contra cuál pool de usuarios autenticarse), el nombre de usuario y la contraseña. El método se encarga enviar una solicitud de inicio de sesión a Cognito a través del SDK con los atributos mencionados anteriormente.

[E7H3A1] Desarrollo del método para cerrar sesión contra pool de usuarios de Cognito en la aplicación controladora de los POS: En esta actividad se concentró el esfuerzo en desarrollar un método que recibiera como parámetro la sesión del usuario activo, para luego enviar una solicitud de cierre de sesión a través del SDK.

[E7H4A1] Desarrollo del método para cambiar de clave contra pool de usuarios de Cognito en la aplicación controladora de los POS; En esta actividad se concentró el esfuerzo en desarrollar un método que recibiera como parámetro la nueva contraseña del usuario, para luego enviar una solicitud de cambio de clave a través del SDK.

2.7. SPRINT 6

Objetivos cubiertos en este Sprint: Desarrollar la aplicación móvil para monitoreo de los POS integrados con IoT

Para este sprint se tuvo como base para el desarrollo el Sprint Backlog Tabla 7 del Apéndice 2, el cual cuenta con las siguientes actividades:

[E8H1A1] Integración con el SDK de AWS IoT: Una vez que se tienen las credenciales necesarias, el usuario registrado en los pool de usuarios de Cognito puede consumir del SDK de AWS IoT. Esta tarea se enfocó en el desarrollo del método que consume del SDK de AWS IoT, el cual es:

- listThings: método que retorna una lista todos los POS registrados en la plataforma AWS IoT.

Como se observa en la figura 35 del Apéndice 1, se presenta una captura de pantalla de lo que representa la ventana de dashboard en la aplicación controladora de POS. En ella se listan los POS disponibles que posee el usuario con sesión iniciada.

[E8H2A1] Integración con el SDK de AWS IoT Shadow: Esta tarea se enfocó en el desarrollo del método que consume del SDK de AWS IoT, el cual es:

- getThingShadow: método que obtiene el shadow del POS a través de la plataforma AWS IoT.
- updateThingShadow: método que actualiza uno o más atributos del Shadow del POS

Cabe destacar que para que un usuario pudiera consumir de los métodos para shadows del SDK se le tienen que dar permisologías adicionales a las dadas por el rol de Cognito. Esto se logra con el método "attachPrincipalPolicy", el cual, a un usuario antes de consumir de cualquier método relacionado con los shadow del SDK de IoT, se le adjunta una política creada previamente en la consola de IoT con la permisología que se necesite. En este caso se usó la siguiente política, como se observa en la figura 37 del Apéndice 1.

Como se puede apreciar en la figura 36 del Apéndice 1, esta es una captura de pantalla en la ventana de la aplicación perteneciente al perfil de un POS seleccionado de la lista en la figura 35. En ella se presenta información básica del POS así como las diferentes acciones que se pueden realizar sobre él.

2.8. *SPRINT 7*

Objetivos cubiertos en este Sprint: Integrar la aplicación móvil para monitoreo de los POS con la plataforma AWS IoT y SNS; Desarrollar un microservicio para el manejo de las notificaciones push entre el POS y las aplicaciones clientes de CityWallet.

Para este sprint se tuvo como base para el desarrollo el Sprint Backlog, Tabla 8 del Apéndice 2, el cual cuenta con las siguientes actividades:

[E9H1A1] Desarrollo del método para la obtención del device token: Cada dispositivo móvil posee un serial identificador único que se usa para el sistema de notificaciones. Esta actividad se centró en el desarrollo del método para la obtención del device token, esto con la finalidad de una vez obtenido el identificador, seria publicado en la plataforma de SNS para registrar el dispositivo y poder recibir notificaciones push.

[E9H2A1] Integración con el SDK de AWS SNS: Esta tarea se enfocó en el desarrollo del método que consume del SDK de AWS SNS, el cual es:

- Subscribe: método que permite que la aplicación se suscriba a un tópico específico para recibir notificaciones push enviadas a dicho tópico.
- updateCognitoUser: método que actualiza el atributo “device” del usuario en el pool de usuarios con el device ID obtenido.

Cada vez que un usuario inicia sesión se suscribe a uno de los dos canales existentes de CityWallet en plataforma AWS SNS. Si el usuario es administrador de CityWallet, se suscribe al tópico admin, el cual recibe notificaciones de todos los POS sin importar a qué establecimiento pertenecen. Si el usuario es administrador de un estacionamiento, se suscribe al tópico /merchant, el cual solo recibe notificaciones push de los POS pertenecientes a su establecimiento

2.9. SPRINT 8

Objetivos cubiertos en este Sprint: Integrar el servicio de notificaciones AWS SNS (Simple Notification Service) para el manejo de notificaciones entre dispositivos; Desarrollar un microservicio para el manejo de las notificaciones push entre el POS y las aplicaciones clientes de CityWallet.

Para este sprint se tuvo como base para el desarrollo el Sprint Backlog, Tabla 9 del Apéndice 2, el cual cuenta con las siguientes actividades:

[E10H1A1] Integración con SDK de Firebase para Web Browser: Para las notificaciones push hacia las aplicaciones merchant y admin se usó la plataforma de Google Firebase, esto debido a que AWS SNS, para el momento en que se desarrolló este trabajo instrumental de grado, no es capaz de mandar notificaciones a navegadores web. Esta actividad consistió en las siguientes tareas:

- Registro de las aplicaciones en la consola de Google Firebase
- Integración de librería JavaScript de Firebase para aplicaciones web

[E10H2A1] Desarrollo del método para la obtención del device token: Así como cada dispositivo móvil posee un device token único el cual es usado para publicar notificaciones push a los mismos, las aplicaciones web también requieren de uno para poder recibir notificaciones. Sin embargo, este código no viene dado por el dispositivo sino por la plataforma de Google Firebase. Una vez integrada a la librería, se desarrolló el método de petición de id, el cual retorna una cadena de caracteres creada por Firebase que se puede usar posteriormente para publicar mensajes hacia esa aplicación web, en ese dispositivo específico.

[E11H1A1] Desarrollo del método para la obtención de todos los usuarios con device token registrado en los pool de usuarios de Cognito: Una vez se tuvo las aplicaciones admin y merchant registradas en Google Firebase, se procedió en el desarrollo de un microservicio el cual es el encargado de manejar todas las notificaciones push generadas por el POS. Este microservicio luego tiene como objetivo, mandar dichas notificaciones los recipientes necesarios dependiendo del caso.

El primer método desarrollado de este microservicio fue el de la obtención de todos los usuarios pertenecientes a los pool de usuarios de merchant y admin que contarán con un device token registrado. Cada

vez que un usuario de merchant o admin inicia sesión, un device token es generado y guardado en el atributo personalizado de "device" que le pertenece a su usuario en el pool de usuarios de Cognito. Así mismo, cuando el usuario cierra sesión, el valor de este atributo es borrado. Cuando se manda una notificación desde un POS, este mensaje es enviado primero a este microservicio y el primer paso es saber a quién se le enviará la notificación.

Como se ve en la figura 38 del Apéndice 1, es un ejemplo de un JSON que envía el POS al microservicio. Este contiene los atributos:

- Pos_id: identificador único alfanumérico del POS.
- Pos_name: nombre del POS con el cual se creó en la plataforma AWS IoT.
- Pos_type: Tipo del pos, si es de entrada o de salida.
- Message: Mensaje a enviar a las aplicaciones cliente.
- Merchant_name: nombre del estacionamiento al que pertenece.

Al saber a qué estacionamiento pertenece el POS (merchant_name), se procede a realizar una búsqueda en el pool de usuarios de merchant para corroborar que tiene una sesión iniciada, y en caso de que lo tenga, obtener el atributo "device" de ese usuario y mandar la notificación a ese device ID.

Sumado a esto, también se busca en el pool de usuarios de admin, ya que todos los usuarios pertenecientes a ese pool reciben notificaciones de todos los POS registrados en la plataforma CityWallet.

Cabe destacar que para esta actividad fue necesario usar el SDK de AWS para la conexión del microservicio con la los pool de usuarios de Cognito. El método que se usó del UDK fue ListAllUsers, el cual devuelve todos los usuarios pertenecientes a un pool de usuarios de Cognito.

[E11H2A1] Integración con SDK de Firebase para NodeJs: Firebase cuenta con una librería administradora para JavaScript. Con esta librería se pueden enviar notificaciones push a los usuarios dado un "device ID" (identificador único del dispositivo) o un conjunto de "device ID". Esta actividad marcó una limitación para la realización del microservicio, ya que solo está disponible para NodeJs, por esta razón, el microservicio fue desarrollado en el mismo entorno. El método desarrollado en esta actividad fue el de publicación de notificaciones a través de la librería de Firebase de Google. Una vez obtenidos los "device ID" por el método desarrollado en la actividad anterior **[E11H1A1]**, se procede a llamar al método de publicación de mensajes, el cual se encarga de consumir del SDK de Firebase y publica el mensaje recibido por el POS (ver figura 38 del Apéndice 1, atributo "message"), a sus destinatarios. Esto para servir las notificaciones a los clientes merchant y admin en la web.

[E11H3A1] Integración con SDK de AWS SNS: El método desarrollado en esta actividad fue el de publicación de notificaciones a través del SDK de AWS. Una vez obtenidos los device tokens por el método desarrollado en la actividad **[E11H1A1]**, se procede a llamar al método push, el cual se encarga de consumir del SDK de AWS y publica el mensaje recibido por el POS (ver figura 38 del Apéndice 1, atributo message) a dos tópicos, el primero el tópico admin, el cual recibe todas las notificaciones independientemente de a qué estacionamiento pertenece el POS, y el tópico de del estacionamiento (cada uno tiene un tópico único), el cual es un tópico único por estacionamiento y solo recibe notificaciones de POS pertenecientes al mismo.

2.10. SPRINT 9

Objetivos cubiertos en este Sprint: Integrar la plataforma AWS IoT con las aplicaciones existentes de CityWallet; Implementar una configuración del POS que le permita conectarse a un gateway remoto.

Para este sprint se tuvo como base para el desarrollo el Sprint Backlog, Tabla 10 del Apéndice 2, el cual cuenta con las siguientes actividades:

[E12H1A1] Implementación de la configuración del POS que le permita conectarse a un gateway remoto: Esta actividad consistió en agregar en el archivo de configuración del POS una variable booleana la cual si es establecida en el valor “true” el POS haría sus procesos como

la versión 1.0, es decir iría contra un gateway remoto para la creación de un token de una transacción. Para esto se modificó el método: createToken. En estos métodos se introdujo un condicional el cual, si era verdadero, el POS no generaría un token si no que realizará una petición contra un gateway centralizado para que este le respondiera con un token. En caso de que el valor de la variable del archivo de configuración sea "false", todo el proceso del POS seria como el planteado en la 2.0, es decir, el POS seria cargo de generar un token para cada transacción sin necesidad de ir contra un gateway externo.

Esta actividad tuvo como objetivo principal cubrir una necesidad que presentaba la empresa de CityWallet en la cual independientemente de la arquitectura que se fuera a escoger para la instalación en el estacionamiento, el POS fuera capaz de adaptarse.

En las figuras 4 y 5 del Apéndice 1, podemos observar la diferencia entre los dos tipos de arquitectura que pueden ser instaladas en un estacionamiento. Cabe destacar que se escoge una con respecto a la otra dependiendo de las necesidades y limitaciones del estacionamiento. En la figura 4 todos los POS están conectados contra aun gateway remoto que es el encargado de suministrarles los tokens necesarios para cada transacción, y de sincronizar los perfiles de usuarios y las transacciones. En la figura 5 el POS es el encargado de realizar todas estas tareas.

[E12H2A1] Desarrollo del método en Python encargado hacer peticiones a Microservicio de notificaciones push: Una vez desarrollado el microservicio que funciona como manejador de las notificaciones, el POS tiene que ser capaz de hacer peticiones contra él. Para esto se desarrolló un método usando la librería por defecto de Python "request", el cual hace una petición del tipo POST al microservicio desarrollado, teniendo como parámetro un JSON como se presenta en la figura 38 del Apéndice 1.

[E13H1A1] Integración con el SDK de AWS IoT: Con el código desarrollado de la actividad **[E8H1A1],** esta tarea se enfocó en la integración de ese código resultante con las aplicaciones web de admin y merchant. Los métodos integrados fueron los siguientes:

- listThings: método que retorna un listado de todos los POS registrados en la plataforma de AWS IoT

Como se observa en la figura 39 y 40, se presenta una captura de pantalla de la opción POS del menú lateral en la aplicación admin y merchant respectivamente. En ellas se listan los POS disponibles que posee el usuario con sesión iniciada.

[E13H2A1] Integración con el SDK de AWS IoT Shadow: Con el código desarrollado de la actividad **[E8H2A1],** esta tarea se enfocó en la integración de ese código resultante con las aplicaciones web de admin y merchant. Los métodos integrados fueron los siguientes:

- getThingShadow: método encargado de obtener el shadow de un POS.
- updateThingShadow: método encargado de actualizar uno o más atributos del shadow de un POS.

Como se observa en la figura 41 y 42 del Apéndice 1, se presenta una captura de pantalla de del perfil del POS seleccionado de la lista de POS disponibles. En ella se presenta información básica del POS así como las diferentes acciones que se pueden realizar sobre él.

CAPÍTULO V - RESULTADOS

1. Resultados

Una vez culminada la etapa del desarrollo, se obtuvieron los siguientes productos los cuales serán listados por cada objetivo que tuvo este trabajo instrumental de grado:

- *Desarrollar la segunda versión los Puntos de Servicio (POS).*

  POS completamente funcional capaz de realizar registro de una transacción (ya sea de entrada o de salida), validación de usuario, creación, firma y verificación de token y. Todo esto sin requerir de un gateway externo. Este resultado se obtuvo en el desarrollo realizado en los Sprints 1, 2, 3 y 4 (ver puntos 2.2, 2.3, 2.4, y 2.5 en el Capitulo IV).

- *Implementar una configuración del POS que le permita conectarse a un gateway remoto.*

  POS capaz de adaptarse a cualquiera de las arquitecturas de instalación actuales ofrecidas por CityWallet, ya sea funcionando solo como lector de calcomanías y dependiendo de un gateway remoto para la generación de tokens para cada transacción, o como POS independiente capaz de realizar todas las funciones que desempeña un gateway. Este resultado se obtuvo en el desarrollo realizado en el Sprint 9 (ver punto 2.10 en el Capitulo IV).

- *Desarrollar la aplicación móvil para monitoreo de los POS integrados con IoT.*

Aplicación móvil capaz de correr en dispositivos con sistema operativo Android o iOS, la cual es capaz de consumir de los servicios de AWS IoT brindándole al usuario un acceso a los estados de los puntos de venta de CityWallet pertenecientes a su establecimiento (en caso de ser un usuario administrador de un estacionamiento) o de los puntos de venta pertenecientes a la red completa de CityWallet (en caso de ser un usuario administrador de CityWallet). Este resultado se obtuvo en el desarrollo realizado en los Sprints 5 y 6 (ver puntos 2.6 y 2.7 en el Capitulo IV).

- *Integrar la aplicación móvil para monitoreo de los POS con la plataforma AWS IoT y SNS.*

Aplicación móvil capaz de correr en dispositivos con sistema operativo Android o iOS, la cual es capaz de consumir de los servicios de AWS SNS, brindándole al usuario la capacidad de recibir notificaciones push cada vez que un POS haga modificaciones en su Shadow, es decir modifique atributos de este JSON. Este resultado se obtuvo en el desarrollo realizado en los Sprints 7 y 8 (ver puntos 2.8 y 2.9 en el Capitulo IV).

- *Desarrollar un microservicio para el manejo de las notificaciones push entre el POS y las aplicaciones clientes de CityWallet.*

De este objetivo se obtuvo un microservicio basado en la tecnología AWS Lambda, el cual es el responsable de distribuir los mensajes entrantes del POS hacia las diferentes aplicaciones clientes. Si

los mensajes se dirigen hacia los usuarios que han iniciado sesión en las aplicaciones web de merchant y admin, el microservicio se encarga de utilizar la librería de Google Firebase para distribuir los mensajes en forma de notificación push hacia estas aplicaciones. Si los mensajes se dirigen a la aplicación móvil que monitorea a los POS, el microservicio se encarga de consumir de la librería de AWS SNS para publicar el mensaje en dos tópicos fijos, uno perteneciente a los administradores de CityWallet y el otro a el tópico perteneciente al estacionamiento en donde se encuentra el POS instalado. Este resultado se obtuvo en el desarrollo realizado en los Sprints 7 y 8 (ver puntos 2.8 y 2.9 en el Capitulo IV).

- *Integrar los web services necesarios de CityWallet con la nueva versión de los POS.*

Un POS capaz de consumir de los servicios en la nube de CityWallet para ampliar sus funcionalidades básicas más allá del registro de transacciones. En este caso, capaz de consumir del microservicio desarrollado en el objetivo anterior, el cual recibirá los mensajes que el POS quiera publicar hacia las aplicaciones clientes de CityWallet. Este resultado se obtuvo en el desarrollo realizado en el Sprint 2 (ver punto 2.3 en el Capitulo IV).

- *Integrar la plataforma AWS IoT con las aplicaciones administrativas de CityWallet.*

Dos aplicaciones web (merchant y admin) las capaces de consumir de los servicios de AWS IoT brindándole al usuario un acceso a los estados de los puntos de venta de CityWallet pertenecientes a su establecimiento, es decir, capaces de consultar el shadow del cualquier POS pertenecientes a la red de CityWallet. Este resultado se obtuvo en el desarrollo realizado en el Sprint 9 (ver punto 2.10 en el Capitulo IV).

- *Integrar el microservicio para manejo de notificaciones push con las aplicaciones administrativas de CityWallet.*

Dos aplicaciones web (merchant y admin) las capaces de consumir de la librería de Google Firebase para la recepción de notificaciones push. Esto con el fin de brindarle al usuario una aplicación web en donde pueda recibir notificaciones provenientes del POS. Este resultado se obtuvo en el desarrollo realizado en el Sprint 9 (ver punto 2.10 en el Capitulo IV).

- *Integración de los POS con la plataforma AWS IoT.*

Con la versión 2.0 POS desarrollado totalmente funcional y capaz de consumir del SDK de AWS IoT, so obtiene un punto de venta que pertenece a una red de POS basados en el Internet de las Cosas de AWS, los cuales son capaces de publicar estados contra un shadow que puede ser consumido y modificado a través de una aplicación cliente. Con IoT, los POS son capaces de recibir señales para que ejecuten

funciones específicas dependiendo del atributo del shadow que se haya modificado. Este resultado se obtuvo en el desarrollo realizado en los Sprints 2, 3, 4 (ver puntos 2.3, 2.4 y 2.5 en el Capitulo IV).

- *Implementación de Shadows para el manejo de estados de los POS incluso en caso de que no posean conexión a internet.*

Con IoT integrado con el POS, también trajo la ventaja de que es posible acceder al último estado actualizado del POS a través de su Shadow. Cada POS, una vez registrado en la red IoT de AWS, crea automáticamente un JSON en la plataforma de IoT al cual se le denomina Shadow. Este documento contiene una serie de atributos los cuales pueden ser modificados por el POS y accedidos por las aplicaciones clientes. Cada Shadow posee información relevante al POS, y la cual puede ser consultada en cualquier momento, incluso si el POS no posee conexión a internet. Este resultado se obtuvo en el desarrollo realizado en los Sprints 3, 4, 5, 6 (ver puntos 2.4, 2.5, 2.6 y 2.7 en el Capitulo IV).

- *Integración con plataforma de notificaciones AWS SNS y Google Firebase para la gestión de notificaciones push entre los puntos de venta y las aplicaciones cliente.*

Teniendo las aplicaciones clientes de CityWallet conectadas con el microservicio de notificaciones push, los usuarios ahora son capaces de recibir mensajes provenientes del POS en forma de notificaciones, ya sea en el navegador web o en el teléfono, dependiendo de qué

aplicación el usuario esté usando, las aplicaciones administrativas o la aplicación móvil encargada de monitorear los POS respectivamente. Este resultado se obtuvo en el desarrollo realizado en los Sprints 7, 8, 9 (ver puntos 2.8, 2.9 y 2.10 en el Capitulo IV).

- *Rediseño de la arquitectura de POS de CityWallet por medio de la integración del Gateway y el punto de servicio en un solo dispositivo e integrándolo con la plataforma AWS IoT.*

La arquitectura inicial de los POS sufrió una modificación en la versión 2.0. Mientras los POS en la versión 1.0 dependen de un gateway centralizado para poder realizar el registro de transacciones y validación de usuarios, el POS 2.0 es capaz de realizar todo esto. Como se ven en las figuras 4 y 5 del Apéndice 1, se puede observar que el Gateway fue eliminado de la arquitectura, y cada POS pasó a ser POS Gateway, es decir, cada POS individualmente puede realizar las mismas funciones que hace un Gateway de CityWallet. Este resultado se obtuvo en el desarrollo realizado en los Sprints 0, 1, 2, 3, 4 (ver puntos 2.1, 2.2, 2.3, 2.4 y 2.5 en el Capitulo IV).

- *Diseño y desarrollo del proceso de control y monitoreo remoto de los POS a través de una aplicación móvil.*

En la figura 2 se puede observar la arquitectura que poseen los POS contra la plataforma IoT y las aplicaciones clientes. En esencia lo POS actualizan a través de IoT su shadow, y estos son consumidos por las aplicaciones clientes. Las aplicaciones clientes a su vez pueden tomar

control remoto del POS, como se aprecia en la figura 43, se muestra el proceso de cómo un usuario desde la aplicación (ya se las administrativas o la móvil) pueden modificar el Shadow del POS, el cambio de algún atributo específico dentro del Shadow acciona una función específica dentro del POS, tales como reiniciar el dispositivo, habilitar o deshabilitarlo, o sincronizar los perfiles de los usuarios. Este resultado se obtuvo en el desarrollo realizado en el Sprints 0 (ver punto 2.1 en el Capitulo IV).

2. Conclusiones

La culminación de este Trabajo Instrumental de grado tiene como resultado final el rediseño de los puntos de venta de CityWallet, lo cual significa una nueva arquitectura de instalación en los estacionamientos, ya que los POS ahora que son independientes del gateway pueden realizar todas las funcionalidades que este antes cumplía.

Sumado a lo anterior CityWallet ahora posee una nueva aplicación que facilita el monitoreo de los puntos de venta, lo cual simplifica el trabajo del administrador de la empresa y agrega valor sobre la solución que se le ofrece a los estacionamientos ya que esta aplicación también puede ser utilizada por los administradores del estacionamiento.

También, conjunto con la plataforma de AWS SNS, se crea una interacción automática entre los puntos de venta y las aplicaciones

clientes la cual permite recibir notificaciones sobre los estados de los POS, y acceder a los mismos aun si estos están desconectados a internet.

Gracias a todo lo anterior, se crea una red de puntos de venta unificados bajo la misma plataforma, que en este caso es la de AWS IoT, y la cual al mismo tiempo posee un nivel de monitoreo mayor al que se tenía en la versión anterior de los puntos de venta.

3. Recomendaciones

Dentro de las oportunidades de mejora cabe destacar que se puede desarrollar un sistema aún más automatizado, en el cual a través de una aplicación cliente se agregue un punto de venta a un estacionamiento y este proceso a su vez, cree automáticamente el POS en el registro de cosas de AWS IoT, genere las políticas y certificados necesarios, los envíe al usuario CityWallet que creo el punto de venta en la aplicación y que solo quede al final, integrar los certificados con el POS correspondiente previamente a su instalación.

También, dentro de las oportunidades de mejora, se puede realizar un desarrollo extra que permita que un POS que esté conectado contra un Gateway remoto, pueda responder en el caso de fallo que este Gateway sea incapaz de responder conectándose contra su propio Gateway, es decir, como el POS 2.0 posee una variable de configuración la cual dicta contra qué tipo de Gateway se conectara (ya sea remoto o propio), se puede desarrollar un failover que detecte el fallo de cualquier

tipo de gateway al cual el POS se esté conectando, y en caso de que este falle el POS sea capaz de conectarse con un Gateway de respaldo (si el POS está configurado para ir contra un Gateway remoto, el Gateway de respaldo sería su propio Gateway, y viceversa).

REFERENCIAS BIBLIOGRÁFICAS

[1] A. Nuñez-Torrón, «¿Qué es Fintech? Todas las claves para entenderlo,» Ticbeat España, Julio 2016. [En línea]. Disponible en: http://www.ticbeat.com/innovacion/que-es-fintech-claves-concepto/ [Último acceso: 27 de octubre de 2016].

[2] A. Ruben, «Qué es el NFC Móvil, cómo funciona y que puedes hacer con él,» Computer Hoy, 8 de febrero de 2017. [En línea]. Disponible en: http://computerhoy.com/noticias/life/que-es-nfc-movil-para-que-sirve-como-funciona-24207 [Último acceso: 27 de octubre de 2016].

[3] S. Bernal (2016). Plataforma de Pagos Basado en Tecnología NFC y Monederos Digitales (trabajo instrumental de pre-grado). Universidad Católica Andrés Bello. Caracas, Venezuela.

[4] Nearfieldcommunication.org, «About Near Field Communication,» Nearfieldcommunication.org, 2012. [En línea]. Disponible en: http://nearfieldcommunication.org/about-nfc.html. [Último acceso: 28 de octubre 2016].

[5] ReQuality, «¿Qué es el IoT (el internet de las cosas)?,» ReQuality Sistemas, 27 de noviembre de 2015. [En línea]. Disponible en: https://www.reqquality.com/que-es-iot-el-internet-de-las-cosas/. [Último acceso: 23 de noviembre de 2016].

[6] TechTarget, «Internet de las cosas (IoT),» TechTarget, enero 2017. [En línea] Disponible en:

http://searchdatacenter.techtarget.com/es/definicion/Internet-de-las-cosas-IoT. [Último acceso: 23 de noviembre de 2016].

[7] DomoDesk; «¿Qué es IoT (el internet de las cosas)?,» DomoDesk, 2015. [En línea]. Disponible en: http://www.domodesk.com/a-fondo-que-es-el-internet-de-las-cosas. [Último acceso: 23 de noviembre de 2016].

[8] J. Lewis y M. Fowler, «Microservices,» 25 de marzo de 2014. [En línea]. Disponible en: https://martinfowler.com/articles/microservices.html#footnote-etymology. [Último acceso: 27 de noviembre de 2016].

[9] Microsoft Azure, «¿Por qué usar un enfoque de microservicios para crear aplicaciones?,» Microsoft, 6 de febrero de 2017. [En línea]. Disponible en: https://docs.microsoft.com/es-es/azure/service-fabric/service-fabric-overview-microservices. [Último acceso: 27 de noviembre de 2016].

[10] Amazon Web Services, «Productos de la nube,» Amazon, 2017. [En línea]. Disponible en: https://aws.amazon.com/es/products/. [Último acceso: 27 de noviembre de 2016]

[11] Amazon Web Services, «Detalles del producto AWS Lambda,» Amazon, 2017. [En línea]. Disponible en: https://aws.amazon.com/es/lambda/details/. [Último acceso: 27 de noviembre de 2016]

[12] Amazon Web Services, «Preguntas frecuentes sobre AWS Lambda,» Amazon, 2017. [En línea]. Disponible en:

https://aws.amazon.com/es/lambda/faqs/. [Último acceso: 27 de noviembre de 2016]

[13] Amazon Web Services, «Preguntas frecuentes de qué es Amazon IoT,» Amazon, 2017. [En línea]. Disponible en: https://aws.amazon.com/es/iot-platform/faqs/. [Último acceso: 27 de noviembre de 2016].

[14] Amazon Web Services, «Características de Amazon IoT,» Amazon, 2017. [En línea]. Disponible en: https://aws.amazon.com/es/iot-platform/how-it-works/. [Último acceso: 27 de noviembre de 2016].

[15] Amazon Web Services, «Preguntas frecuentes sobre Amazon IoT Shadows» Amazon, 2017. [En línea]. Disponible en: http://docs.aws.amazon.com/iot/latest/developerguide/iot-thing-shadows.html. [Último acceso: 28 de noviembre de 2016].

[16] Amazon Web Services, «Preguntas frecuentes sobre Amazon SNS» Amazon, 2017. [En línea]. Disponible en: https://aws.amazon.com/es/sns/faqs/. [Último acceso: 28 de noviembre de 2016].

[17] Amazon Web Services, «Amazon Simple Notification Service (SNS),» Amazon, 2017. [En línea]. Disponible en: https://aws.amazon.com/es/sns/. [Último acceso: 28 de noviembre de 2016].

[18] Upplication, «¿Qué son las notificaciones push y para qué sirven?,» Upplication, 16 de abril de 2004. [En línea]. Disponible en: https://www.upplication.com/blog/que-son-las-notificaciones-push/. [Último acceso: 11 de enero de 2017].

[19] Google, «Firebase te ayuda a crear mejores apps para dispositivos móviles y hacer crecer tu empresa,» Google, 2017. [En línea]. Disponible en: https://firebase.google.com/. [Último acceso: 11 de enero de 2017].

[20] Wikipedia, «Firma Digital,» Wikipedia, 2017. [En línea]. Disponible en: https://es.wikipedia.org/wiki/Firma_digital. [Último acceso: 24 de marzo de 2017].

[21] Wikipedia, «DSA,» Wikipedia, 2016. [En línea]. Disponible en: https://es.wikipedia.org/wiki/DSA. [Último acceso: 11 de septiembre de 2016].

[22] Wikipedia, «Secure Hash Algorithms» Wikipedia, 2017. [En línea]. Disponible en: https://en.wikipedia.org/wiki/Secure_Hash_Algorithms. [Último acceso: 13 de julio de 2017].

[23] Interciencia, «¿Qué es el Método Heurístico?», Interciencia, [En línea]. Disponible en: https://inteciencia.wordpress.com/2013/01/22/que-es-el-metodo-heuristico/ . [Último acceso: 05 de Mayo de 2017].

[24] CPrime, «What is Agile? What is Scrum?», CPrime, [En línea]. Disponible en: https://www.cprime.com/resources/what-is-agile-what-is-scrum/. [Último acceso: 05 de Mayo de 2017].

[25] Scrum Alliance, «The Manifesto», Scrum Alliance, Inc. 2017 [En línea]. Disponible en: https://www.scrumalliance.org/why-scrum/core-scrum-values-roles [Último acceso: 05 de Mayo de 2017].

[26] Scrum Org, «What is Scrum», Scrum.Org, 2017 [En línea]. Disponible en: https://www.scrum.org/resources/what-is-scrum [Último acceso: 05 de Mayo de 2017].

[27] Scrum Org, «What is the Product Backlog?», Scrum.Org, 2017 [En línea]. Disponible en: https://www.scrum.org/resources/what-is-a-product-backlo. [Último acceso: 06 de Mayo de 2017].

[28] Scrum Org, «What is the Product Backlog?», Scrum.Org, 2017 [En línea]. Disponible en: https://www.scrum.org/resources/what-is-a-sprint-backlog [Último acceso: 04 de Mayo de 2017].

[29] Scrum Manager, «Incremento», Scrum Manager, 2014, [En línea]. Disponible en: https://www.scrummanager.net/bok/index.php?title=Incremento [Último acceso: 07 de Mayo de 2017].

[30] Scrum Org, «What is Scrum?», Scrum.Org, 2017 [En línea]. Disponible en: https://www.scrum.org/resources/what-is-scrum [Último acceso: 05 de Mayo
de 2017].

[31] Scrum Org, «What is a Sprint in Scrum?», Scrum.Org, 2017 [En línea]. Disponible en: https://www.scrum.org/resources/what-is-a-sprint-in-scrum [Último acceso: 05 de Mayo de 2017].

[32] Scrum Org, «What is Sprint Planning?», Scrum.Org, 2017 [En línea]. Disponible en: https://www.scrum.org/resources/what-is-sprint-planning sprint [Último acceso: 05 de Mayo de 2017].

[33] Scrum Org, «What is a Daily Scrum?», Scrum.Org, 2017 [En línea]. Disponible en: https://www.scrum.org/resources/what-is-a-daily-scrum [Último acceso: 05 de Mayo de 2017].

[34] Scrum Org, «What is a Sprint Review?», Scrum.Org, 2017 [En línea]. Disponible en: https://www.scrum.org/resources/what-is-a-sprint-review [Último acceso: 05 de Mayo de 2017].

[35] Scrum Org, «What is a Product Owner?», Scrum.Org, 2017 [En línea]. Disponible en: https://www.scrum.org/resources/what-is-a-product-owner [Último acceso: 05 de Mayo de 2017].

[36] Scrum Org, «What is a Scrum Master?», Scrum.Org, 2017 [En línea]. Disponible en: https://www.scrum.org/resources/what-is-a-scrum-master: [Último acceso: 05 de Mayo de 2017].

[37] Scrum Org, «What is a Scrum Development Team?», Scrum.Org, 2017 [En línea]. Disponible en: https://www.scrum.org/resources/what-is-a-scrum-development-team: [Último acceso: 05 de Mayo de 2017].

[38] Confluence, «Working with Epics», Atlassian Documentation, 2017 [En línea]. Disponible en: https://confluence.atlassian.com/agile/jira-agile-user-s-guide/working-with-epics: [Último acceso: 01 de Febrero de 2017].

APÉNDICES

1. Figuras

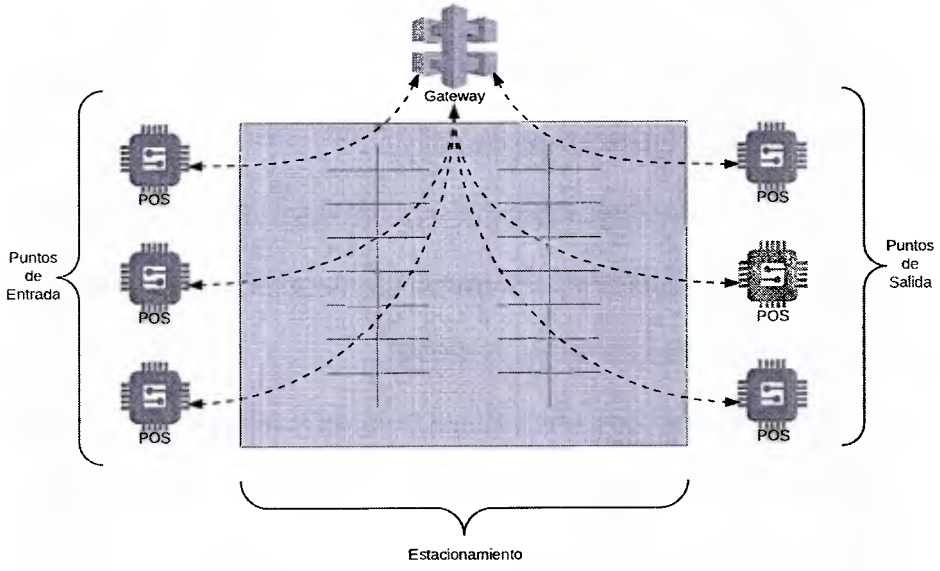

Figura 4. Diagrama de Arquitectura POSv1
Fuente [Elaboración Propia]

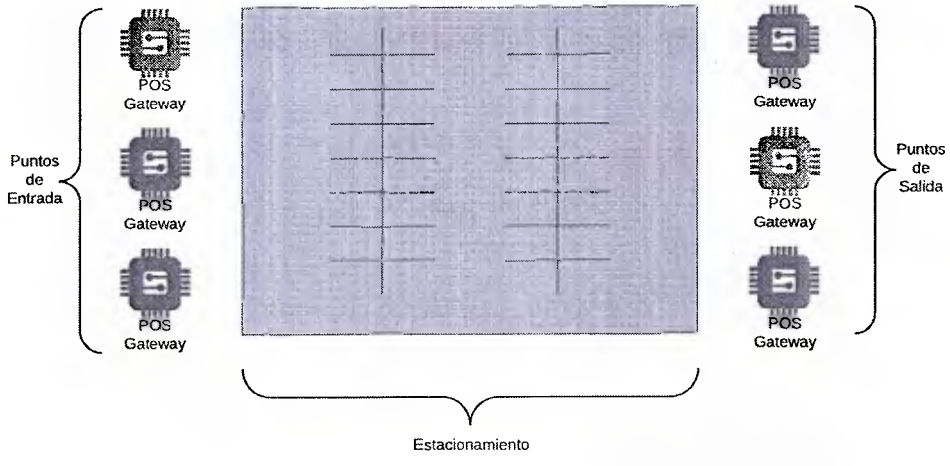

Figura 5. Diagrama de Arquitectura POSv2
Fuente [Elaboración Propia]

```
{
    "_id" : "59729a2e6ed1376219771cce",
    "profileId" : "b9921a69-7c3b-48eb-8789-31faf6a12984",
    "accountId" : "06275f67-f95f-4410-acb4-e9d10842a7a2",
    "transactionToolId" : "6fd6d3a0-d755-4c52-bec7-f78eeb858aae",
    "userExternalKey" : "04F34C42D44880",
    "profileType" : "PARKING_CLIENT",
    "profileStatus" : "INSUFFICIENT_BALANCE",
    "createdDate" : 1499094291202,
    "updatedDate" : 1499094291202,
    "metadata" :
"{\"firstName\":\"manuel\",\"lastName\":\"pacheco\",\"pin\":\"AAAp\",\"pak\":\"1208\
"}",
    "documentId" : "V-21438184"
}
```


Figura 6. Ejemplo perfil de un usuario guardado en la base de datos local del POS
Fuente [Elaboración Propia]

```
public boolean isValidUser(Transaction transaction) throws GatewayException {
        String stickerSerial = transaction.getSticker();
        FindIterable<Document> iterable = _mongoDatabase
                .getCollection(MongoConstants.COLLECTION_PROFILES)
                .find(new
Document(MongoConstants.PROFILE_DOCUMENT_FIELD_STICKER_SERIAL,
                        stickerSerial));

        if(iterable == null || iterable.first() == null)
        {
            throw new GatewayException(GatewayException.NON_EXISTING_TAG_ID_CODE,
                    GatewayException.NON_EXISTING_TAG_ID_MSG);
        }

        boolean sufficient_balance = true;
        String status = "";
        Date plan_date = null;
        Document document_profile = iterable.first();

        status =
document_profile.getString(MongoConstants.PROFILE_DOCUMENT_FIELD_STATUS);

        FindIterable<Document> iterable_merchant = _mongoDatabase
                .getCollection(MongoConstants.COLLECTION_MERCHANT)
                .find();

        if(iterable_merchant != null || iterable_merchant.first() != null){

            String merchant_id = iterable_merchant.first().

get(MongoConstants.MERCHANT_DOCUMENT_FIELD_MERCHANT_ID).toString();

            List<Document> profile_plans = (List<Document>)document_profile.get(
                    MongoConstants.PROFILE_DOCUMENT_FIELD_PLANS_ARRAY);

            if (profile_plans != null){
                for (Document doc_plans : profile_plans){
```

```
                    if
(doc_plans.get(MongoConstants.MERCHANT_DOCUMENT_FIELD_MERCHANT_ID).toString()
                            .equals(merchant_id)){
                        plan_date = Utils.stringToDate(
doc_plans.getString(MongoConstants.PROFILE_DOCUMENT_FIELD_PLAN_FINISH_DATE));
                    }
                }
            }
        }else{
            throw new GatewayException(GatewayException.INTERNAL_ERROR_CODE,
                    GatewayException.INTERNAL_ERROR_MSG);
        }

        if (!status.equals(GatewayStatus.ACTIVE)){
            if (status
                    .equals(GatewayStatus.INACTIVE_ACCOUNT))
            {
                throw new GatewayException(GatewayException.INACTIVE_USER_CODE,
                        GatewayException.INACTIVE_USER_MSG);
            }

            if (status
                    .equals(GatewayStatus.INACTIVE_STICKER))
            {
                throw new GatewayException(GatewayException.INACTIVE_STICKER_CODE,
                        GatewayException.INACTIVE_STICKER_MSG);
            }

            if (status
                    .equals(GatewayStatus.INSUFFICIENT_BALANCE)
                    && (plan_date == null || !plan_date.before(new Date()))) {

                sufficient_balance = false;
            }
        }

        Date today = new Date();

        if(plan_date != null)
        {
            if(plan_date.before(today) && !sufficient_balance &&
transaction.getTransactionType().equals(Constants.PARKING_TRANSACTIDN_IN))
            {
                throw new GatewayException(GatewayException.NO_PLAN_NO_BALANCE_CODE,
                        GatewayException.NO_PLAN_NO_BALANCE_MSG);
            }
        }

        if (!sufficient_balance &&
transaction.getTransactionType().equals(Constants.PARKING_TRANSACTION_IN)){
            throw new GatewayException(GatewayException.INSSUFICIENT_BALANCE_CODE,
                    GatewayException.INSSUFICIENT_BALANCE_MSG);
        }
        return true;
    }
```

Figura 7. Metodo IsValidUser del Gateway en Java
Fuente [Elaboración Propia]

```
def isValidUser(self, transaction):
        stickerSerial = transaction.getSticker()
        logging.info("isValidUser tagId = " + stickerSerial)

        documentProfile = self.db.profile.find_one(
            {mongoConstants.PROFILE_DOCUMENT_FIELD_EXTERNAL_KEY: stickerSerial})

        if documentProfile is None:
            logging.error("Non existing user")
            raise GatewayException(
                GatewayException.NON_EXISTING_TAG_ID_CODE,
GatewayException.NON_EXISTING_TAG_ID_MSG)

        if not self.validateUserInAcl(transaction):
            logging.error("Access denied for user: "+ transaction.getSticker())
            raise GatewayException(
                GatewayException.ACCESS_DENIED_CODE,
GatewayException.ACCESS_DENIED_MSG)


        sufficient_balance = True
        status = ""
        plan_date = None

        status = documentProfile[mongoConstants.PROFILE_DOCUMENT_FIELD_STATUS]

        documentMerchant = self.db.merchant.find_one()

        if documentMerchant is not None:
            merchant_id = str(

documentMerchant[mongoConstants.MERCHANT_DOCUMENT_FIELD_MERCHANT_ID])

            try:
                profile_plans = documentProfile[
                    mongoConstants.PROFILE_DOCUMENT_FIELD_PLANS_ARRAY]
                for plan in profile_plans:
                    if str(plan[mongoConstants.MERCHANT_DOCUMENT_FIELD_MERCHANT_ID])
== merchant_id:
                        plan_date = util.stringToDate(

plan[mongoConstants.PROFILE_DOCUMENT_FIELD_PLAN_FINISH_DATE])
            except KeyError:
                logging.info("No plans")
        else:
            raise GatewayException(
                GatewayException.INTERNAL_ERROR_CODE,
GatewayException.INTERNAL_ERROR_MSG)

        print "users status: " + status
        if status != gatewayStatus.ACTIVE:
            if status == gatewayStatus.INACTIVE_ACCOUNT:
                raise GatewayException(
                    GatewayException.INACTIVE_USER_CODE,
GatewayException.INACTIVE_USER_MSG)
            if status == gatewayStatus.BLOCKED:
                raise GatewayException(
                    GatewayException.BLOCKED_USER_CODE,
GatewayException.BLOCKED_USER_MSG)
            if status == gatewayStatus.INSUFFICIENT_BALANCE and (
```

```
                    plan_date is None or plan_date < datetime.datetime.now()):
                sufficient_balance = False

            if plan_date is not None:
                if (plan_date < datetime.datetime.now() and sufficient_balance ==
False and
                        transaction.getTransactionType() ==
gatewayConstants.PARKING_TRANSACTION_IN):
                    raise GatewayException(GatewayException.NO_PLAN_NO_BALANCE_CODE,
                                           GatewayException.NO_PLAN_NO_BALANCE_MSG)

        print "transactionType: " + transaction.getTransactionType()

        if sufficient_balance == False and transaction.getTransactionType() ==
gatewayConstants.PARKING_TRANSACTION_IN:
            raise GatewayException(GatewayException.INSUFFICIENT_BALANCE_CODE,
                                   GatewayException.INSUFFICIENT_BALANCE_MSG)
        print "retorno TRUE"
        return True
```

Figura 8. Metodo IsValidUser del Gateway en Python
Fuente [Elaboración Propia]

```
public boolean userHasPlan(String stickerSerial) throws GatewayException{

        FindIterable<Document> iterable = _mongoDatabase
                .getCollection(MongoConstants.COLLECTION_PROFILES)
                .find(new
Document(MongoConstants.PROFILE_DOCUMENT_FIELD_STICKER_SERIAL,
                        stickerSerial));

        if(iterable == null || iterable.first() == null)
        {
            throw new GatewayException(GatewayException.NON_EXISTING_TAG_ID_CODE,
                    GatewayException.NON_EXISTING_TAG_ID_MSG);
        }

        Date plan_date = null;
        Document document_profile = iterable.first();
        FindIterable<Document> iterable_merchant = _mongoDatabase
                .getCollection(MongoConstants.COLLECTION_MERCHANT)
                .find();

        if(iterable_merchant != null || iterable_merchant.first() != null){

            String merchant_id = iterable_merchant.first().

get(MongoConstants.MERCHANT_DOCUMENT_FIELD_MERCHANT_ID).toString();
            List<Document> profile_plans = (List<Document>)document_profile.get(
                    MongoConstants.PROFILE_DOCUMENT_FIELD_PLANS_ARRAY);
            if (profile_plans != null) {
                for (Document doc_plans : profile_plans) {
                    if
(doc_plans.get(MongoConstants.MERCHANT_DOCUMENT_FIELD_MERCHANT_ID).toString()
                            .equals(merchant_id)) {
                        plan_date = Utils.stringToDate(
```

```
doc_plans.getString(MongoConstants.PROFILE_DOCUMENT_FIELD_PLAN_FINISH_DATE));
                    }
                }
            }else{
                return false;
            }
        }else{
            throw new GatewayException(GatewayException.INTERNAL_ERROR_CODE,
                    GatewayException.INTERNAL_ERROR_MSG);
        }

        Date today = new Date();

        if(plan_date != null)
        {
            if(plan_date.before(today))
            {
                return false;
            }else {
                return true;
            }
        }else{
            return false;
        }

    }
```

Figura 9. Metodo UserHasPlan del Gateway en Java
Fuente [Elaboración Propia]

```
def userHasPlan(self, stickerSerial):
        # get one profile given an sticker serial
        document_profile = self.db.profile.find_one(
            {mongoConstants.PROFILE_DOCUMENT_FIELD_EXTERNAL_KEY: stickerSerial})

        if document_profile is None:
            raise GatewayException(GatewayException.NON_EXISTING_TAG_ID_CODE,
                                   GatewayException.NON_EXISTING_TAG_ID_MSG)

        plan_date = None
        # get one merchant (all info)
        merchant_profile = self.db.merchant.find_one()
        if merchant_profile is not None:
            # get one merchant id
            merchant_id = merchant_profile[
                mongoConstants.MERCHANT_DOCUMENT_FIELD_MERCHANT_ID]
            # get plans of a client
            try:
                profile_plans = document_profile[
                    mongoConstants.PROFILE_DOCUMENT_FIELD_PLANS_ARRAY]
```

```
                if profile_plans is not None:
                    for doc_plan in profile_plans:
                        if
str(doc_plan[mongoConstants.MERCHANT_DOCUMENT_FIELD_MERCHANT_ID]) == merchant_id:
                            plan_date = util.stringToDate(

str(doc_plan[mongoConstants.PROFILE_DOCUMENT_FIELD_PLAN_FINISH_DATE]))
                else:
                    return False
            except KeyError:
                logging.info("No Plans")
        else:
            raise GatewayException(GatewayException.INTERNAL_ERROR_CODE,
                                   GatewayException.INTERNAL_ERROR_MSG)

        today = datetime.datetime.now()

        if (plan_date is not None):
            if (plan_date < today):
                return False
            else:
                return True
        else:
            return False
```

Figura 10. Metodo UserHasPlan del Gateway en Python
Fuente [Elaboración Propia]

0457AF5AD84981,AAAA11,1A,1,1612160923,MC0CFQCV738rNxWRof/8uGEKIxg6jug8fwIU
bilsCzn1ou7mOjS7hyOumaE97IQ=

Figura 11. Ejemplo de token
Fuente [Elaboración Propia]

```
public String createToken(String tagId, String currentToken, String posId, String
isIn) throws Exception{
        JsonObject error = new JsonObject();
        JsonObject response = new JsonObject();
        JsonObject wrap = new JsonObject();
        String newToken = "";
        try{
            _logger.info("incomming token: " + currentToken);

            validateCreateTokenInput(tagId, currentToken, posId);
```

```
            String transactionType = "";

            if (isIn == null || isIn.equals(Constants.TRUE) || isIn.isEmpty() )
            {
                transactionType = Constants.PARKING_TRANSACTION_IN;
            }
            else if (isIn.equals(Constants.FALSE))
            {
                transactionType = Constants.PARKING_TRANSACTION_OUT;
            }
            Transaction = new Transaction();
            transaction.setSticker(tagId);
            transaction.setTransactionType(transactionType);
            transaction.setToken(currentToken);
            if(_gManager.userIsValid(transaction) &&
_gManager.isValidTransaction(transaction))
            {
                newToken = _gManager.createNewToken(tagId,
                        currentToken, posId);
                _logger.info("Sended token: " + newToken);
                _logger.info("Token length: " + newToken.length());
                response = _jManager.buildCreateToken(newToken);
                error = _jManager.buildOkError();
            }
        }
        catch(GatewayException ex){
            _logger.debug("Error: " + ex.getMsg());
            error = _jManager.buildErrorFromGatewayException(ex);
        }
        catch(Exception ex){
            _logger.debug("Error ", ex);
            error = _jManager.buildInternalError(ex);
        }
        wrap.add(Constants.JSON_FIELD_RESPONSE, response);
        wrap.add(Constants.JSON_FIELD_ERROR, error);
        return _gson.toJson(wrap);
    }
```

Figura 12. Metodo de createToken en Gateway de Java
Fuente [Elaboración Propia]

```
def createToken(self, tagId, currentToken):
        logging.info('-------------------------------------------------------------
-------------------------------------------------------------')
        if tagId == Settings.previousTagId and Settings.previousToken is not None:
            return Settings.previousToken
        else:
            if Settings.gatewayMode == 'True':
                    json =
self.gatewayManager.createToken(str(tagId),currentToken,str(Settings.posId),str(Sett
ings.isIn))

                    if json["error"]["code"] == 0:
                        logging.info('Returning token: ' +
```

```
json["response"]["token"])
                        token = json["response"]["token"]
                        self.setGlobalVar(tagId, token)
                      Settings.passToken = currentToken
                      return token
                    else:
                        subprocess.Popen(["pkill", "-f", "lednfc.py"],
stdout=subprocess.PIPE)
                        subprocess.call('python ledRed.py &', shell=True)
                        raise
CityWalletException(json["error"]["code"],json["error"]["msg"] + " TagId: " +
self.nfcTag)
            else:
                token = self.coreManager.createToken(str(tagId), currentToken,
str(Settings.posId), str(Settings.isIn))
                self.setGlobalVar(tagId, token)
               Settings.passToken = currentToken
                return token
```

Figura 13. Metodo de createToken en Gateway de Python
Fuente [Elaboración Propia]

```
public boolean isValidTransaction(Transaction transaction) throws GatewayException {

        if (isProfileWithInvalidToken(transaction)){
            return true;
        }
        else {
            if (transaction.getToken() != null && !transaction.getToken().isEmpty())
{

                String tagId = transaction.getSticker();
                String type = transaction.getTransactionType();
                String incomingToken = transaction.getToken();
                String tokenGwId = incomingToken.split(",")[4];
                Calendar tokenDate =
Util.tokenDateToCore(incomingToken.split(",")[5]);
                Calendar today = Calendar.getInstance();
                today.setTime(new Date());
                String gatewayId = Integer.toString(getGwId());
                _logger.debug("TRANSACTION TYPE: " + type);
                _logger.debug("TOKEN GW ID:" + tokenGwId);
                _logger.debug("LA LAGUNITA IN POS: " +
Constants.LA_LAGUNITA_IN_PDS);
                _logger.debug("TODAY: " + today.get(Calendar.DAY_OF_MONTH));
                _logger.debug("TOKEN DAY: " + tokenDate.get(Calendar.DAY_OF_MONTH));

                if (type.equals(Constants.PARKING_TRANSACTION_OUT)
                        && !Arrays.asList(getInGateways()).contains(tokenGwId)) {
                    throw new
GatewayException(GatewayException.MISSING_CHECKIN_CODE,
                            GatewayException.MISSING_CHECKIN_MSG);

                } else if (type.equals(Constants.PARKING_TRANSACTION_IN)
                        && tokenGwId.equals(gatewayId)) {
                    throw new
```

```
GatewayException(GatewayException.PENDING_CHECKOUT_CODE,
                            GatewayException.PENDING_CHECKOUT_MSG);
                }
            }
        }
        return true;
    }
```

Figura 14. Método de isValidTransaction en Gateway de Java
Fuente [Elaboración Propia]

```
def isValidTransaction(self, transaction):
        logging.info("isValidTransaction with transaction: " +
                     str(transaction))

        if self.isProfileWithInvalidToken(transaction):
            logging.info("Profile with invalid token")
            return True
        else:
            print "not transaction.getToken(): " + str(not transaction.getToken())
            print " transaction.getToken().isspace(): " +
str(transaction.getToken().isspace())
            if (not transaction.getToken() or transaction.getToken().isspace()) and
transaction.getTransactionType() == gatewayConstants.PARKING_TRANSACTION_IN:
                print "ENTRA AQUI, PRIMERA VEZ RETORNO TRUE"
                return True
            if (not transaction.getToken() or transaction.getToken().isspace()) and
transaction.getTransactionType() == gatewayConstants.PARKING_TRANSACTION_OUT:
                logging.error("Mising checkin for transaction: " + str(transaction))
                raise GatewayException(GatewayException.MISSING_CHECKIN_CODE,
                                       GatewayException.MISSING_CHECKIN_MSG)

            if transaction.getToken() and transaction.getToken() != "":
                transactionType = transaction.getTransactionType()
                incomingToken = transaction.getToken()
                tokenGwId = incomingToken.split(",")[1]
                today = datetime.datetime.now()
                gatewayId = str(Settings.posId)
                inPosPrefix = Settings.parkingPosPrefix + "1"
                logging.info("tokengwid: " + str(tokenGwId))
                logging.info("inPosPrefix: " + str(inPosPrefix))

                if transactionType == gatewayConstants.PARKING_TRANSACTION_OUT:
                    if not tokenGwId.startswith(inPosPrefix):
                        logging.error(
                            "Mising checkin for transaction: " + str(transaction))
                        raise
GatewayException(GatewayException.MISSING_CHECKIN_CODE,
                                                 GatewayException.MISSING_CHECKIN_MSG)
                else:
                    if transactionType == gatewayConstants.PARKING_TRANSACTION_IN:
                        if tokenGwId.startswith(inPosPrefix):
                            logging.error(
                                "Pending checkout for transaction: " +
str(transaction))
                            raise
```

```
GatewayException(GatewayException.PENDING_CHECKOUT_CODE,

GatewayException.PENDING_CHECKOUT_MSG)
                return True
```

Figura 15. Método de isValidTransaction en Gateway de Python
Fuente [Elaboración Propia]


```
{
        "_id" : ObjectId("58a61ba95a5c4a044b738103"),
        "sticker" : "04EA4C42D44880",
        "transactions" : [
                {
                        "date" : "2017-02-16 17:37:45",
                        "payment_type" : "BALANCE",
                        "pos_id" : 13,
                        "status" : "OK",
                        "timestamp" : "2017-02-16 17:37:45",
                        "token" :
"04EA4C42D44880,247,11,1,6,1602171737,MCwCFEFZuETovF2MvBGdc7wN6biZhWyQAhQHzIJ3CuThIw
i7JpYsUe/N/lQwaQ==,0000000000000000000000000000000000000000",
                        "transaction_id" : 648,
                        "transaction_type" : "IN"
                }
        ]
}
```


Figura 16. Ejemplo de una transacción almacenada en base de datos local del POS
Fuente [Elaboración Propia]

```
public PosTransaction createParkingTransaction(Transaction transaction) throws
GatewayException {

        if(userIsValid(transaction))// &&
                //isValidTransaction(transaction))
        {
            FindIterable<Document> counter =
                    _mongoDatabase.getCollection(MongoConstants.COLLECTION_COUNTER)
                            .find(new Document(MongoConstants.COUNTER_FIELD_ID,

MongoConstants.COUNTER_FIELD_ID_TRANSACTION_VALUE));

            int transId = 0;

            if(counter == null || counter.first() == null)
            {
                transId = 1;
                createCounterCollection();
            }
            else
            {
                transId = getNextSequence(counter.first());
```

```
            }

            Date today = new Date();
            Date startDate = Utils.stringToDate(transaction.getDate());

            if(today.before(startDate))
            {
                throw new
GatewayException(GatewayException.INVALID_TRANSACTION_CODE,
                        GatewayException.INVALID_TRANSACTION_MSG);
            }


if(!transaction.getTransactionType().equals(Constants.PARKING_TRANSACTION_IN) &&

!transaction.getTransactionType().equals(Constants.PARKING_TRANSACTION_OUT))
            {
                throw new
GatewayException(GatewayException.INVALID_TRANSACTION_CODE,
                        GatewayException.INVALID_TRANSACTION_MSG);
            }

            PosTransactionImpl pos = new PosTransactionImpl();
            pos.setName(getProfileName(transaction.getSticker()));

            if (userHasPlan(transaction.getSticker())){
                pos.setPaymentType(Constants.PAYMENT_TYPE_PLAN);
            }else{
                pos.setPaymentType(Constants.PAYMENT_TYPE_BALANCE);
            }

            FindIterable<Document> iterable =

_mongoDatabase.getCollection(MongoConstants.COLLECTION_TRANSACTION)
                            .find(
                                    new
Document(MongoConstants.TRANSACTION_DOCUMENT_FIELD_STICKER,
                                            transaction.getSticker()));

            if(iterable == null || iterable.first() == null)
            {
                _mongoDatabase.getCollection(MongoConstants.COLLECTION_TRANSACTION)
                        .insertOne(
                                new Document()

.append(MongoConstants.TRANSACTION_DOCUMENT_FIELD_STICKER,
                                                transaction.getSticker())

.append(MongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_ARRAY,
                                                asList(
                                                        new Document()

.append(MongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_ID,
                                                                        transId)

.append(MongoConstants.TRANSACTION_DOCUMENT_FIELD_POS_ID,

Integer.valueOf(transaction.getPosId()))

.append(MongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_TYPE,
```

```
transaction getTransactionType())

.append(MongoConstants.TRANSACTION_DOCUMENT_FIELD_DATE,

transaction.getDate())

.append(MongoConstants.TRANSACTION_DOCUMENT_FIELD_TOKEN,

transaction.getToken())

.append(MongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_TIMESTAMP,

Utils.dateToString(new Date()))

.append(MongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_STATUS,

MongoConstants.TRANSACTION_STATUS_WAITING)

.append(MongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_PAYMENT_TYPE,

pos.getPaymentType())
                                                    ))
                        );
            }
            else
            {
                _mongoDatabase.getCollection(MongoConstants.COLLECTION_TRANSACTION)
                        .updateOne(
                                new
Document(MongoConstants.TRANSACTION_DOCUMENT_FIELD_STICKER,
                                        transaction.getSticker()),
                                new Document(MongoConstants.MONGO_OPERATOR_PUSH,
                                        new
Document(MongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_ARRAY,
                                                new Document()

.append(MongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_ID, transId)

.append(MongoConstants.TRANSACTION_DOCUMENT_FIELD_POS_ID,

Integer.valueOf(transaction.getPosId()))

.append(MongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_TYPE,

transaction.getTransactionType())

.append(MongoConstants.TRANSACTION_DOCUMENT_FIELD_DATE,

transaction.getDate())

.append(MongoConstants.TRANSACTION_DOCUMENT_FIELD_TOKEN,

transaction.getToken())

.append(MongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_TIMESTAMP,

Utils.dateToString(new Date()))

.append(MongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_STATUS,
```

```
MongoConstants.TRANSACTION_STATUS_WAITING)

.append(MongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_PAYMENT_TYPE,

pos.getPaymentType())
                                        ))
                        );
            }
            pos.setId(String.valueOf(transId));
            return pos;
        }
        return null;
    }
```

Figura 17. Método createParkingTransaction desarrollado en el Gateway de Java Fuente [Elaboración Propia]

```
def createParkingTransaction(self, transaction):
    logging.info("createParkingTransaction transaction = " +
                    str(transaction))

    if self.isValidUser(transaction):
        today = datetime.datetime.today()
        counter = self.db.counter.find_one(
                {mongoConstants.COUNTER_FIELD_ID:
mongoConstants.COUNTER_FIELD_ID_TRANSACTION_VALUE})
        transId = 0
        if counter is None:
            transId = 1
            self.createCounterCollection()
        else:
            transId = self.getNextSequence(counter)

        if today < transaction.getDate():
            raise GatewayException(GatewayException.INVALID_TRANSACTION_CODE,
                                   GatewayException.INVALID_TRANSACTION_MSG)

        if transaction.getTransactionType() is not
gatewayConstants.PARKING_TRANSACTION_IN and transaction.getTransactionType() \
                is not gatewayConstants.PARKING_TRANSACTION_OUT:
            raise GatewayException(GatewayException.INVALID_TRANSACTION_CODE,
                                   GatewayException.INVALID_TRANSACTION_MSG)
        pos = PosTransaction()
        pos.setName(self.getProfileName(transaction.getSticker()))
        pos.setTransactionType(transaction.getTransactionType)
        if self.userHasPlan(transaction.getSticker()):
            pos.setPaymentType(gatewayConstants.PAYMENT_TYPE_PLAN)
        else:
            pos.setPaymentType(gatewayConstants.PAYMENT_TYPE_BALANCE)

        document_transaction = self.db.sticker_transaction.find_one(
            {mongoConstants.TRANSACTION_DOCUMENT_FIELD_STICKER:
transaction.getSticker()})

        if document_transaction is None:
```

```
            self.db.sticker_transaction.insert({
                mongoConstants.TRANSACTION_DOCUMENT_FIELD_STICKER:
transaction.getSticker(),
                mongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_ARRAY: [
                    {
                        mongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_ID:
transaId,
                        mongoConstants.TRANSACTION_DOCUMENT_FIELD_POS_ID:
transaction.getPosId(),
                        mongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_TYPE:
str(
                            transaction.getTransactionType()),
                        mongoConstants.TRANSACTION_DOCUMENT_FIELD_DATE:
util.dateToString(transaction.getDate()),
                        mongoConstants.TRANSACTION_DOCUMENT_FIELD_TOKEN:
str(transaction.getToken()),

mongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_TIMESTAMP: util.dateToString(
                            datetime.datetime.now()),

mongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_STATUS:
mongoConstants.TRANSACTION_STATUS_WAITING,

mongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_PAYMENT_TYPE: str(
                            pos.getPaymentType())
                    }
                ]
            })

        else:
            self.db.sticker_transaction.update_one(
                {mongoConstants.TRANSACTION_DOCUMENT_FIELD_STICKER:
transaction.getSticker()},
                {
                    '$push':
{mongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_ARRAY: {
                        mongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_ID:
transaId,
                        mongoConstants.TRANSACTION_DOCUMENT_FIELD_POS_ID:
transaction.getPosId(),
                        mongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_TYPE:
str(
                            transaction.getTransactionType()),
                        mongoConstants.TRANSACTION_DOCUMENT_FIELD_DATE:
util.dateToString(transaction.getDate()),
                        mongoConstants.TRANSACTION_DOCUMENT_FIELD_TOKEN:
str(transaction.getToken()),

mongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_TIMESTAMP: util.dateToString(
                            datetime.datetime.now()),

mongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_STATUS:
mongoConstants.TRANSACTION_STATUS_WAITING,

mongoConstants.TRANSACTION_DOCUMENT_FIELD_TRANSACTION_PAYMENT_TYPE: str(
                            pos.getPaymentType())
                    }}
                })

        pos.setId(str(transId))
```

```
        return pos
    else:
        return None
```

Figura 18. Método createParkingTransaction desarrollado en el Gateway de Python
Fuente [Elaboración Propia]

```
2017-07-19 08:43:28 - INFO - timeout - startTimer - starting timeout error for writing error
2017-07-19 08:43:29 - INFO - ControllerModule - processTag - Written token verified successfully
2017-07-19 08:43:29 - INFO - ControllerModule - registerParkingTransaction - Registering parking transaction for Tag :044E5552A84A81
```

Figura 19. Ejemplo de Logs en el POS
Fuente [Elaboración Propia]

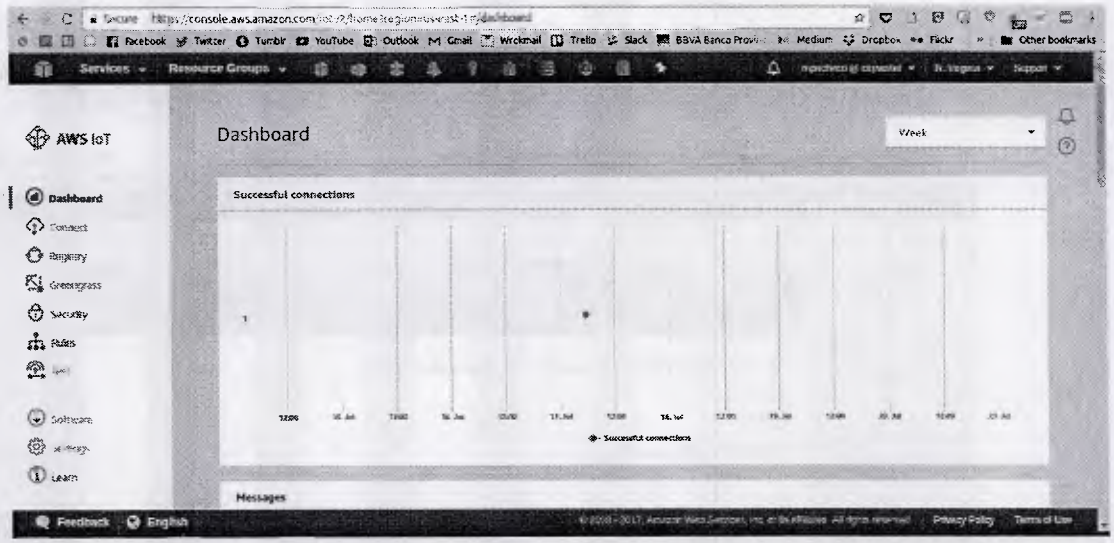

Figura 20. Pantalla de Dashboard de AWS IoT Console
Fuente [Elaboración Propia]

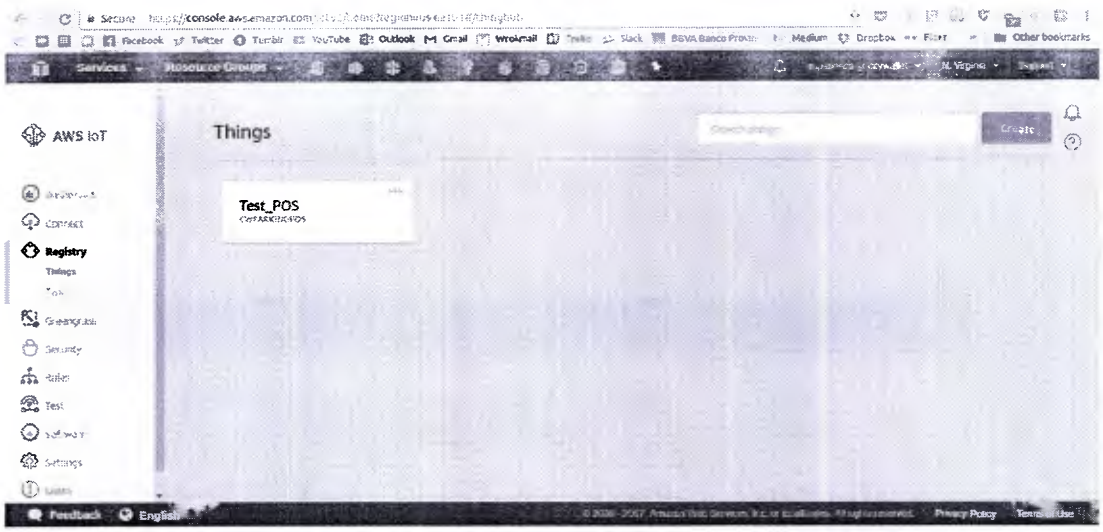

Figura 21. Listado de POS registrados en la plataforma AWS IoT
Fuente [Elaboración Propia]

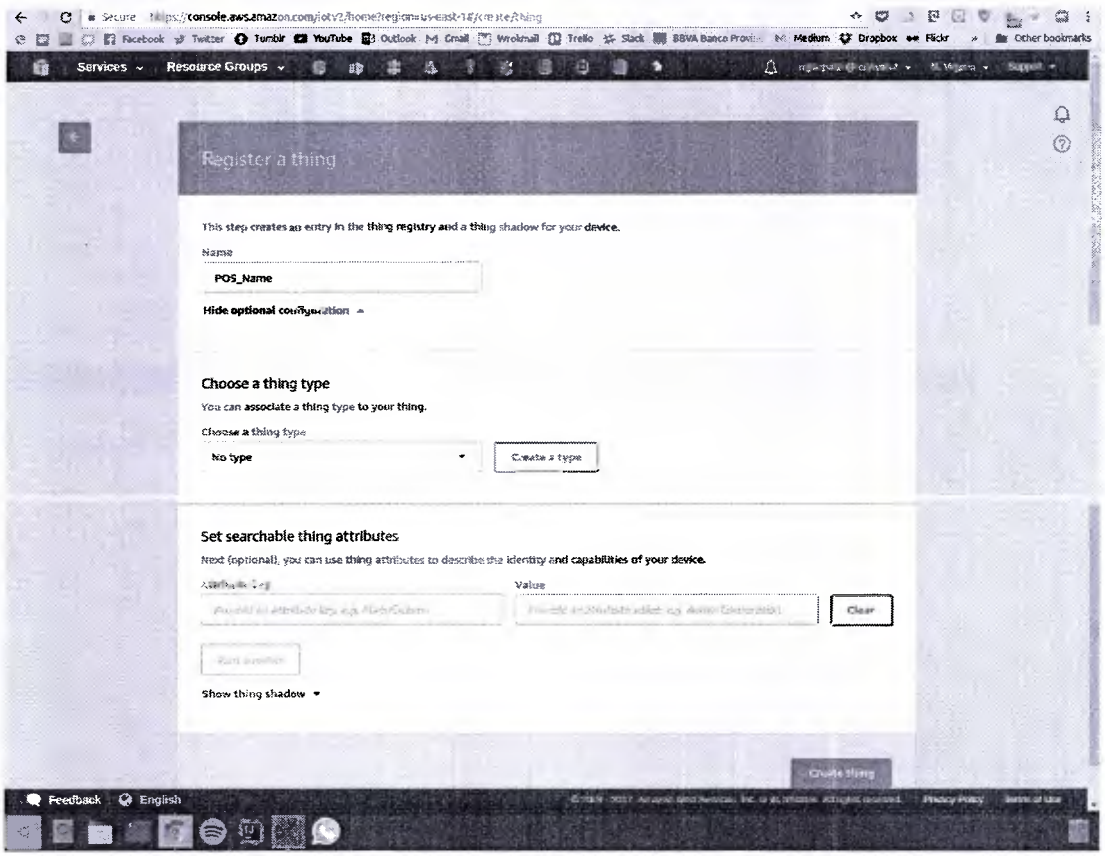

Figura 22. Formulario para la creación de un "Thing"
Fuente [Elaboración Propia]

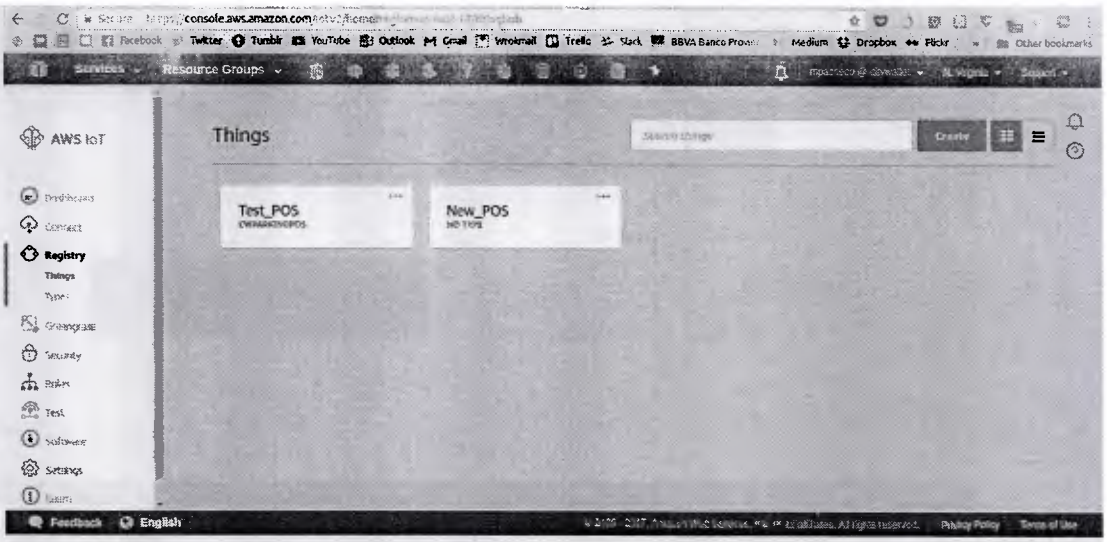

Figura 23. Listado de POs registrados en AWS IoT con el nuevo POS
Fuente [Elaboración Propia]

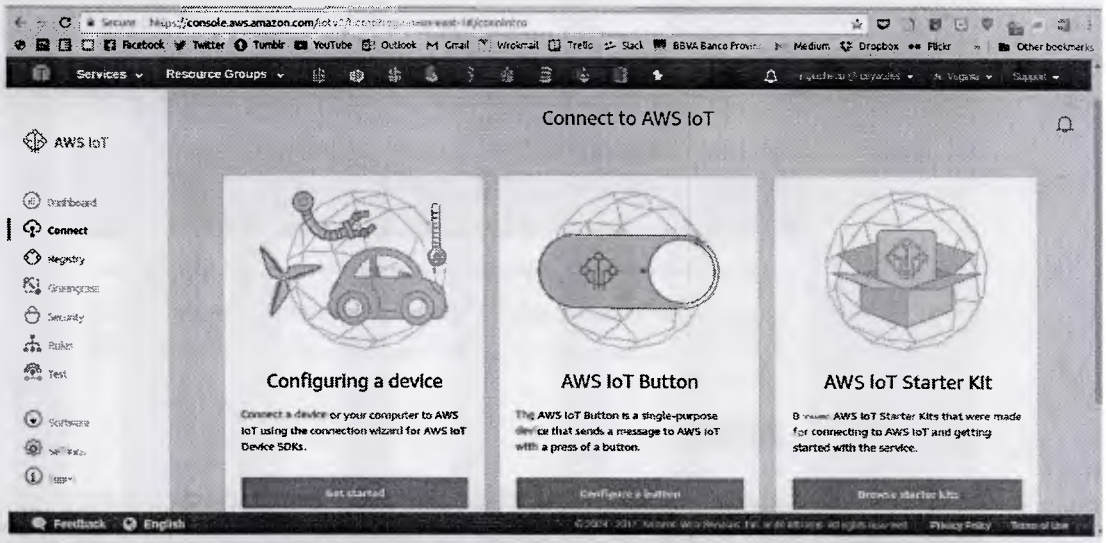

Figura 24. Opcion de Connect en panel de AWS IoT
Fuente [Elaboración Propia]

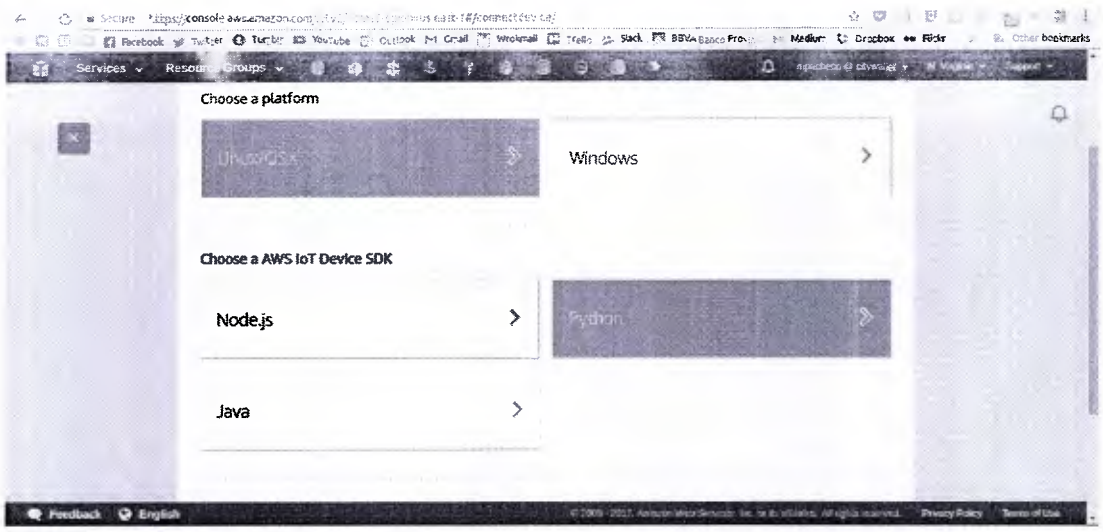

Figura 25. Selección del sistema operativo y lenguaje de programacion del POS
Fuente [Elaboración Propia]

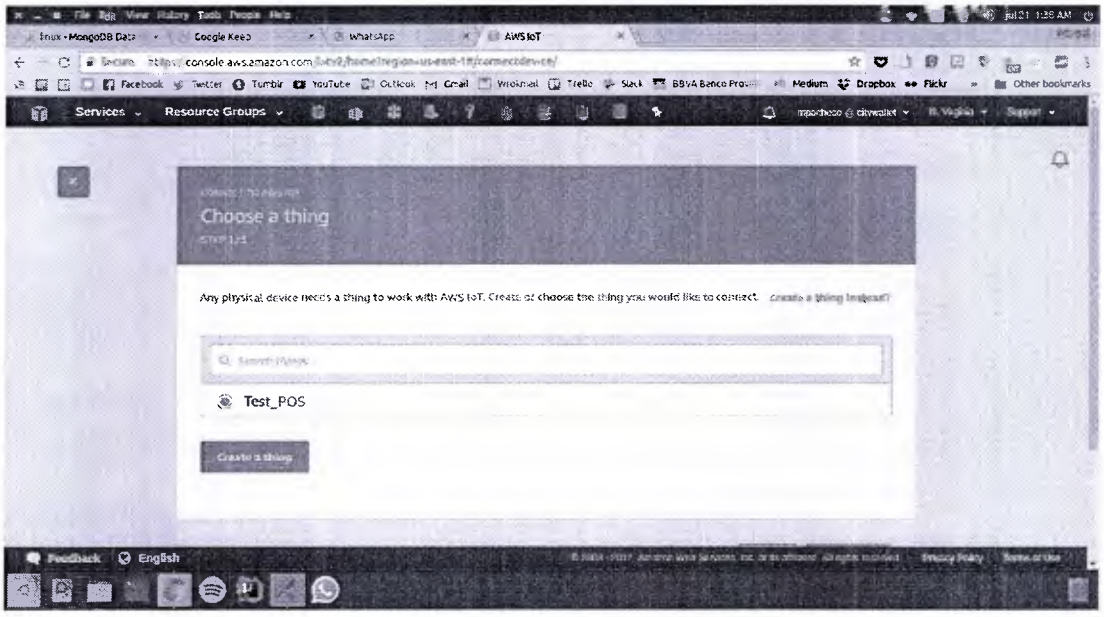

Figura 26. Seleccion del POS creado previamente
Fuente [Elaboración Propia]

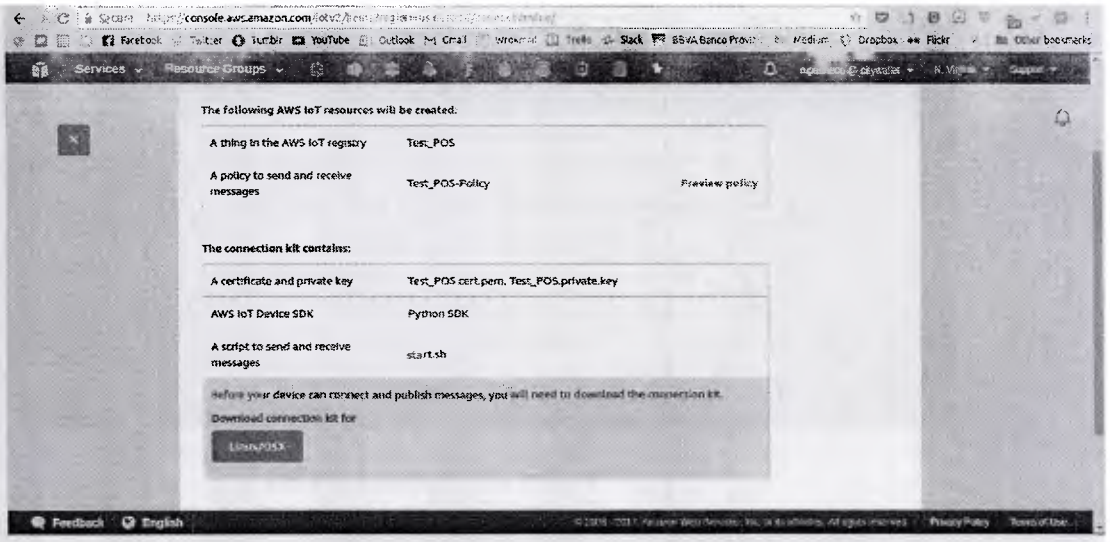

Figura 27. Descarga del Kit de Conexión para el POS
Fuente [Elaboración Propia]

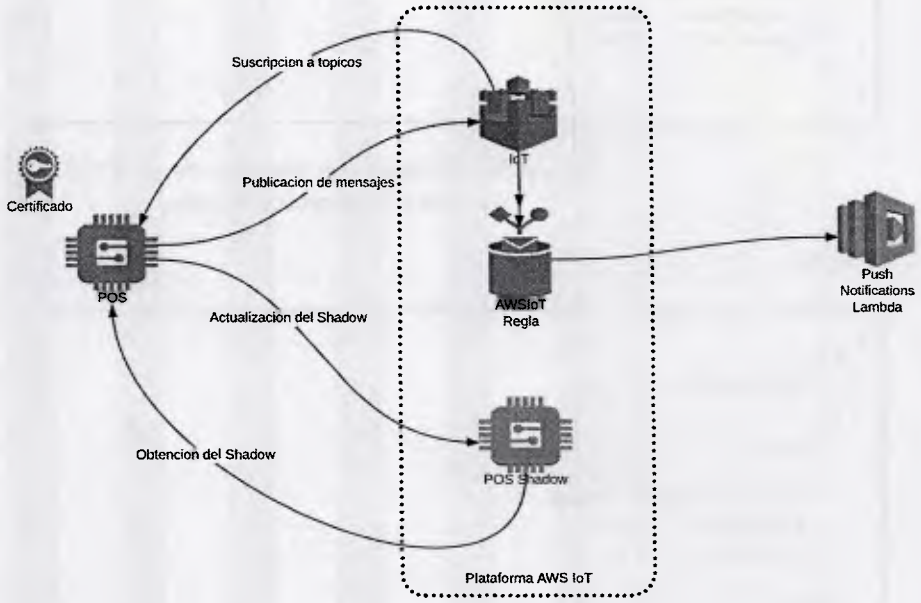

Figura 28. Interacción del POS con la plataforma AWS IoT
Fuente [Elaboración Propia]

Código	Caso	Consideraciones	Input	Output	Observaciones
1	Conexión contra AWS IoT	POS posee certificados	Ninguno	Exito	
2	Conexión contra AWS IoT	POS no posee certificados	Ninguno	Error	Al POS no poseer los certificados requeridos pos AWS IoT, no puede conectarse contra la plataforma
3	Publicación aun topico	Se publicó a un tópico cualquiera de ejemplo	/test, "Este es un mensaje de prueba"	Success	
4	Suscripción a un topico	Se suscribió a un tópico cualquiera de ejemplo	/test	"Este es un mensaje de prueba"	
5	Cancelar la suscripcion de un topico	Se canceló la suscripcion del mismo topico de ejemplo	/test	Success	Al ya no estar suscrito no se recibió más mensajes publicados al topico
6	Desconexión con AWS IoT	Después de desconectase se intento realizar alguna funcion sobre la plataforma AWS IoT	Ninguno	Success	
7	Configuración del Last Will	Se envió un JSON en forma de string	{"state":{"reported":{"state": "running", "syncTransactions": true, "syncMerchant": true, "syncProfiles": true }}}	Success	

Figura 29. Pruebas QA IoT
Fuente [Elaboración Propia]

```
{
  "reported": {
    "status": "running",
    "syncProfiles": false,
    "isEnabled": true,
    "connected": true
  }
}
```

Figura 30. Ejemplo del Shadow de un POS
Fuente [Elaboración Propia]

```
{
  "desired": {
    "syncProfiles": true,
  },
  "reported": {
    "status": "running",
    "syncProfiles": false,
    "isEnabled": true,
    "connected": true
  },
  "delta": {
    "syncProfiles": true,
  }
}
```

Figura 31. Ejemplo de cambio en un atributo del Shadow de un POS
Fuente [Elaboración Propia]

Código	Caso	Consideraciones	Input	Output	Observaciones
1	Check in	Usuario con saldo	Se presenta la calcomnia en el lector del POS	Success	Se reproduce el audio de exito en el POS
2	Check in	Usuario sin saldo	Se presenta la calcomnia en el lector del POS	Error	Se reproduce el audio de saldo insuficiente en el POS
3	Check in	Usuario con calcomania bloqueada	Se presenta la calcomnia en el lector del POS	Error	Se reproduce el audio de eror en el POS
4	Check in	Usuario con perfil bloqueado	Se presenta la calcomnia en el lector del POS	Error	Se reproduce el audio de error en el POS
5	Check in	Usuario con un check in existente	Se presenta la calcomnia en el lector del POS	Error	Se reproduce el audio de chack out faltante en el POS
6	Check out	Usuario con check in registrado	Se presenta la calcomnia en el lector del POS	Success	Se reproduce el audio de exito en el POS
7	Check out	Usuario con sin check in registrado	Se presenta la calcomnia en el lector del POS	Error	Se reproduce el audio de check in faltante en el POS
8	Check out	Usuario con calcomania bloqueada	Se presenta la calcomnia en el lector del POS	Error	Se reproduce el audio de error en el POS
9	Check out	Usuario con perfil bloqueado	Se presenta la calcomnia en el lector del POS	Error	Se reproduce el audio de error en el POS

Figura 32. Pruebas QA del POS
Fuente [Elaboración Propia]

Figura 33. Pantalla Inicio de Sesión
Fuente [Elaboración Propia]

Figura 34. Pantalla Cambio de Contraseña
Fuente [Elaboración Propia]

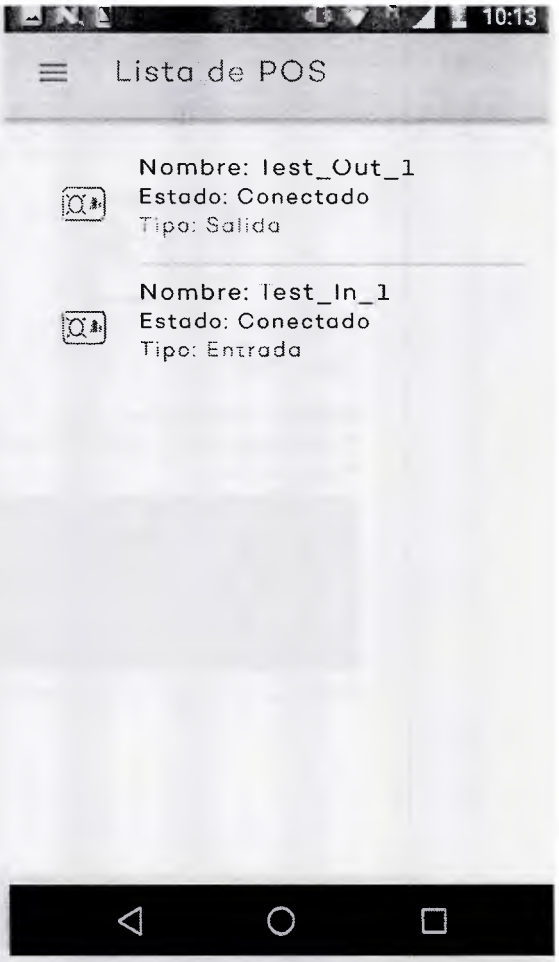

Figura 35. Pantalla de Dashboard
Fuente [Elaboración Propia]

Figura 36. Pantalla de Perfil del POS
Fuente [Elaboración Propia]

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Connect",
        "iot:UpdateThingShadow",
        "iot:GetThingShadow"
      ],
      "Resource": "*"
    }
  ]
}
```

Figura 37. Política iot-data-access
Fuente [Elaboración Propia]

```
{
  "pos_id": 1,
  "pos_name": "Test_POS",
  "pos_type": "in",
  "message": "This is a test push notification",
  "merchant_name": "tolon",
}
```

Figura 38. Ejemplo del JSON que recibe el microservicio
Fuente [Elaboración Propia]

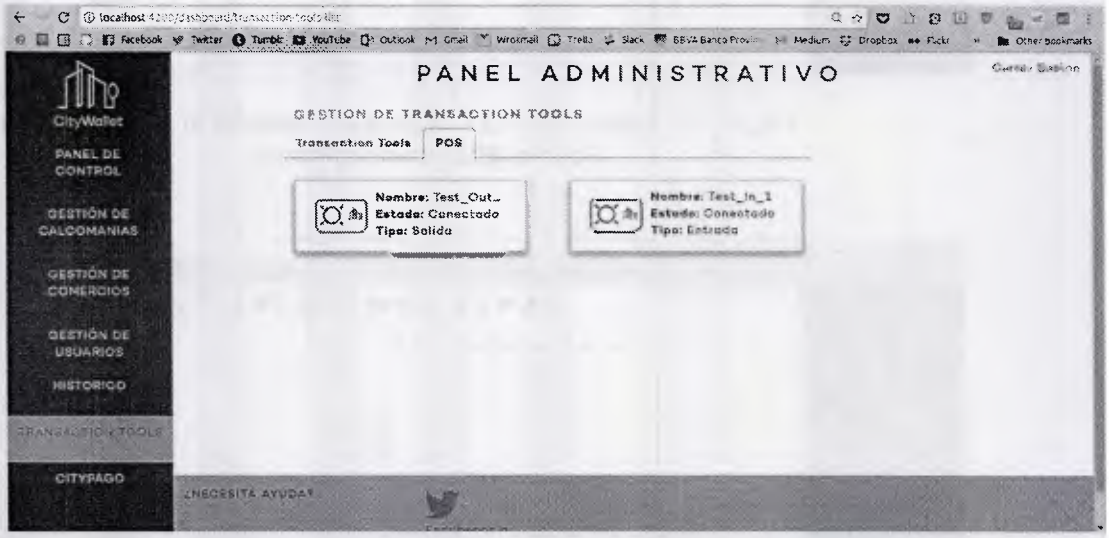

Figura 39. Listado de POS disponibles aplicación Admin
Fuente [Elaboración Propia]

Figura 40. Listado de POS disponibles aplicación Merchant
Fuente [Elaboración Propia]

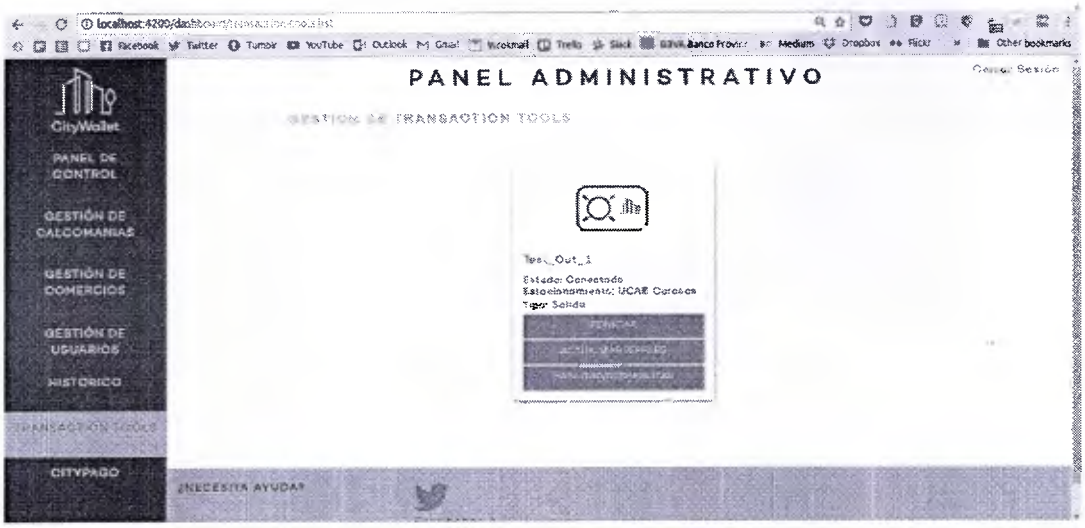

Figura 41. Perfil del POS en aplicación Admin
Fuente [Elaboración Propia]

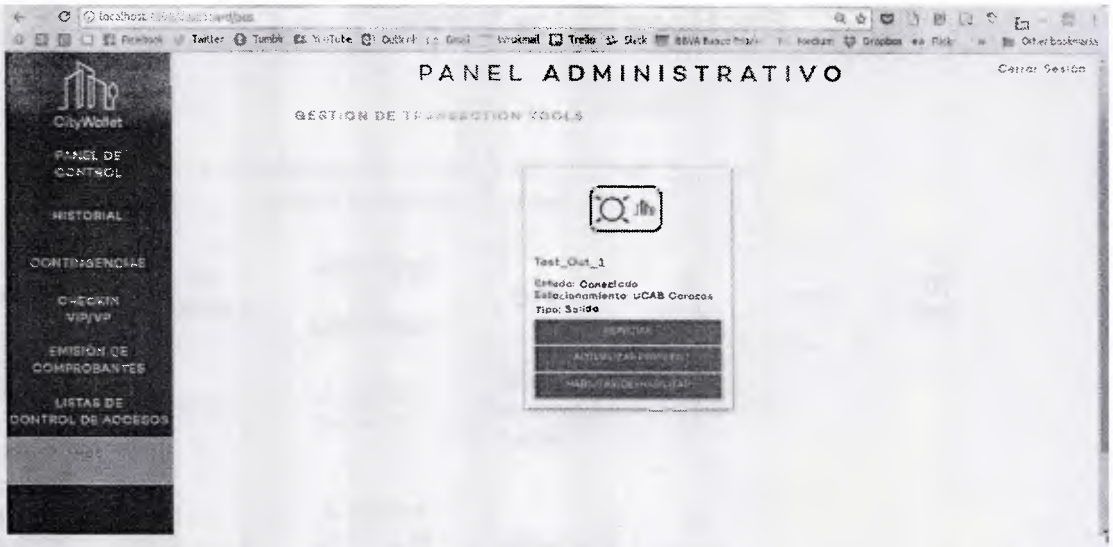

Figura 42. Perfil del POS en aplicación Merchant
Fuente [Elaboración Propia]

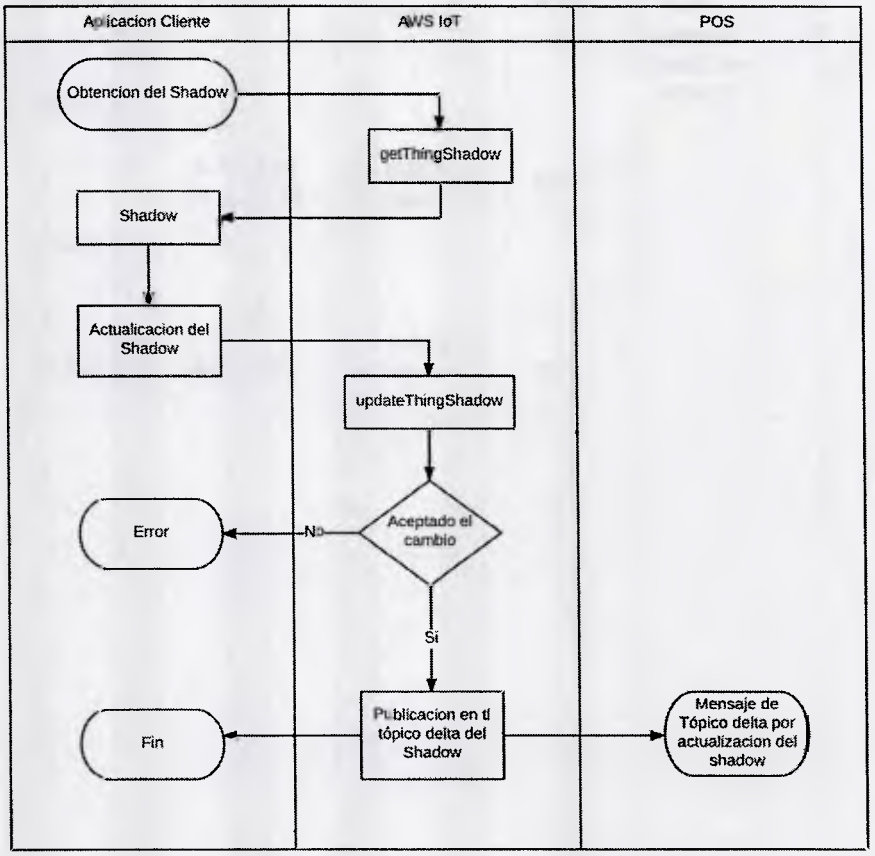

Figura 43. Proceso de Modificación de un Shadow Mediante aplicacion cliente
Fuente [Elaboración Propia]

2. Backlogs de los Sprints

2.1. *Sprint 1*

Tabla 2. Backlog del Sprint correspondiente al Sprint 1
Fuente [Elaboración Propia]

ID EPIC	EPIC	ID HITO	HITO	HISTORIA DE USUARIO	ID ACTIVIDAD	ACTIVIDADES
E1	Migración del Gateway de Java a Python	E1H1	Check in / Check out exitoso	*POSv2-1, POSv2-8*	E1H1A1	Integración de lógica migrada de Java a Python con código de python existente
		E1H2	Validación de usuario	POSv2-3, POSv2-18, POSv2-20, POSv2-17	E1H2A1	Migración de los métodos de Java a Python para el manejo del perfil del usuario de base de datos local
		E1H3	Creación de Token	*POSv2-5*	E1H3A1	Migración de método de Java a Python de creación de token
		E1H4	Registro de Transacción	*POSv2-4, POSv2-19*	E1H4A1	Migración de método de Java a Python de Validación del tipo de transacción
					E1H4A2	Migración de método de Java a Python de registro de una transacción en base de datos local
		E1H5	Registro del Proceso del POS en logs	*POSv2-9*	E1H5A1	Registrar en logs el proceso completo del POS

2.2. *Sprint 2*

Tabla 3. Backlog del Sprint correspondiente al Sprint 2
Fuente [Elaboración Propia]

ID EPIC	EPIC	ID HITO	HITO	HISTORIA DE USUARIO	ID ACTIVIDAD	ACTIVIDADES
E2	Firma y validación del token	E2H1	Generación del JAR que firmara los tokens creados por el POS	*POSv2-2, POSv2-6, POSv2-7*	E2H1A1	Extracción de los métodos de firma y verificación de firmas del Gateway a un Java Jar aparte para su consumo individual (Token Signer)
		E2H2	Integración entre Python y Java con Py4j	*POSv2-2*	E2H2A1	Integración del POS en python con la librería Py4j para generar un puente entre python y la librería de firma de tokens Token Signer
E3	Integración del POS con AWS IoT	E3H1	Módulo de IoT en Python	*POSv2-13*	E3H1A1	Creación del Thing en la plataforma de AWS IoT
					E3H1A2	Creación de los certificados del Thing
					E3H1A3	Desarrollo del Módulo de IoT en python

2.3. *Sprint 3*

Tabla 4. Backlog del Sprint correspondiente al Sprint 3
Fuente [Elaboración Propia]

ID EPIC	EPIC	ID HITO	HITO	HISTORIA DE USUARIO	ID ACTIVIDAD	ACTIVIDADES
E4	Integración con IoT Shadow	E4H1	Conexión contra la plataforma AWS IoT	*POSv2-11*	E4H1A1	Prueba de conexión con plataforma AWS IoT
		E4H2	Obtención y actualización del shadow	*POSv2-14*	E4H2A1	Desarrollo de métodos de obtención y actualización de Shadow en Módulo de IoT

			POS signals	*POSv2-15*	E4H2A2	Desarrollo del método de publicación de estados del POS

2.4. Sprint 4

Tabla 5. Backlog del Sprint correspondiente al Sprint 4
Fuente [Elaboración Propia]

ID EPIC	EPIC	ID HITO	HITO	HISTORIA DE USUARIO	ID ACTIVIDAD	ACTIVIDADES
E5	POS funcional con IoT Integrado	E5H1	POS completamente funcional con IoT y shadow integrado	*POSv2-13, POSv2-18, POSv2-19*	E5H1A1	Prueba completas del POS

2.5. Sprint 5

Tabla 6. Backlog del Sprint correspondiente al Sprint 5
Fuente [Elaboración Propia]

ID EPIC	EPIC	ID HITO	HITO	HISTORIA DE USUARIO	ID ACTIVIDAD	ACTIVIDADES
E6	Aplicación Controladora de los POS	E6H1	Pantallas y diseño de la aplicación	*POSv2-20*	E6H1A1	Desarrollo de la pantalla de inicio de sesión
					E6H1A2	Desarrollo de la pantalla de cambio de clave
					E6H1A3	Desarrollo de la pantalla de Dashboard (lista de POS disponibles según el usuario con sesión iniciada)
					E6H1A4	Desarrollo de la pantalla del perfil del POS (información general referente al estado del mismo)
E7	Integración con Cognito	E7H2	Iniciar Sesión	*POSv2-20*	E7H2A1	Creación de los Pool de Usuarios de Cognito

					E7H2A2	Desarrollo del método para iniciar sesión contra pool de usuarios de Cognito en la aplicación controladora de los POS
		E7H3	**Cerrar Sesión**	*POSv2-20*	E7H3A1	Desarrollo del método para cerrar sesión contra pool de usuarios de Cognito en la aplicación controladora de los POS
		E7H4	Cambio de Clave	*POSv2-20*	E7H4A1	Desarrollo del método para cambiar de clave contra pool de usuarios de Cognito en la aplicación controladora de los POS

2.6. *Sprint 6*

Tabla 7. Backlog del Sprint correspondiente al Sprint 6
Fuente [Elaboración Propia]

ID EPIC	EPIC	ID HITO	HITO	HISTORIA DE USUARIO	ID ACTIVIDAD	ACTIVIDADES
E8	Integración de IoT Shadow con POS app	E8H1	Listar POS disponibles dependiendo del usuario con sesión iniciada	*POSv2-12*	**E8H1A1**	Integración con el SDK de AWS IoT
		E8H2	Obtención y actualización del Shadow de un POS	*POSv2-13, POSv2-14, POSv2-20*	E8H2A1	Integración con el SDK de AWS IoT Shadow

2.7. Sprint 7

Tabla 8. Backlog del Sprint correspondiente al Sprint 7

Fuente [Elaboración Propia]

ID EPIC	EPIC	ID HITO	HITO	HISTORIA DE USUARIO	ID ACTIVIDAD	ACTIVIDADES
E9	Integración de SNS con POS app	E9H1	Obtención y registro del device ID en SNS	*POSv2-21*	E9H1A1	Desarrollo del método para la obtención del device token
		E9H2	Suscripción a tópicos dependiendo del usuario que haya iniciado sesión	*POSv2-16, POSv2-21*	E9H2A1	Integración con el SDK de AWS SNS

2.8. Sprint 8

Tabla 9. Backlog del Sprint correspondiente al Sprint 8
Fuente [Elaboración Propia]

ID EPIC	EPIC	ID HITO	HITO	HISTORIA DE USUARIO	ID ACTIVIDAD	ACTIVIDADES
E10	Integración de notificaciones push con aplicaciones Merchant y Admin	E10H1	Obtención y registro del browser ID en Google Firebase	*POSv2-16, POSv2-21*	E10H1A1	Desarrollo del método para la obtención del device token
		E10H2	Integración de las aplicaciones Merchant y Admin con Google Firebase	*POSv2-16, POSv2-21*	E10H2A1	Integración con SDK de Firebase para Web Browser
E11	Lambda handler entre las aplicaciones POS, Merchant y Admin app y el servicio de notificacione	E11H1	Obtención de los device IDs de los usuarios con sesión iniciada en las aplicaciones Merchant y Admin app	*POSv2-16, POSv2-21*	E11H1A1	Desarrollo del método para la obtención de todos los usuarios con device token registrado en los pool de usuarios de Cognito

	s push	E11H2	Integración con Google Firebase NodeJS Admin SDK para publicación de notificaciones a aplicaciones Merchant y Admin	*POSv2-16, POSv2-21*	E11H2A1	Integración con SDK de Firebase para NodeJs
		E11H3	Integración con SNS para publicación de notificaciones a POS app	*POSv2-16, POSv2-21*	E11H3A1	Integración con SDK de AWS SNS

2.9. Sprint 9

Tabla 10. Backlog del Sprint correspondiente al Sprint 9
Fuente [Elaboración Propia]

ID EPIC	EPIC	ID HITO	HITO	HISTORIA DE USUARIO	ID ACTIVIDAD	ACTIVIDADES
E12	Configuración del POS para consumo de APIs de CityWallet	E12H1	Realizar request contra servicios de Gateway Remoto	*POSv2-10*	E12H1A1	Implementación de la configuración del POS que le permita conectarse a un gateway remoto
		E12H2	Realizar request contra Microservicio de notificaciones push	*POSv2-16, POSv2-21*	E12H2A1	Desarrollo del método en python encargado de realizar peticiones al microservicio de notificaciones push
E13	Integración de IoT Shadow con aplicaciones Merchant y Admin	E13H1	Listar POS disponibles dependiendo del usuario con sesión iniciada	*POSv2-12, POSv2-20*	E13H1A1	Integración con el SDK de AWS IoT
		E13H2	Obtención y actualización del Shadow de un POS	*POSv2-12, POSv2-20*	E13H2A1	Integración con el SDK de AWS IoT Shadow