UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

INTEGRACIÓN DE UN PROTOTIPO QUE PERMITA LA AUTENTICACIÓN PRESENCIAL CON COMPATIBILIDAD DE MÚLTIPLES DISPOSITIVOS DESARROLLADA EN MOQUI

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

LUKAS EDWIN GRIBBELL LAGOS

PROFESOR GUÍA:
Andrés Muñóz Ordenes

MIEMBROS DE LA COMISIÓN:
Alejandro Hevia Angulo
Patricio Inostroza Fajardin

SANTIAGO DE CHILE
2022

Resumen

En el presente informe se explicará el desarrollo de un prototipo que permita la autenticación presencial utilizando Moqui como *framework* seleccionado para su implementación.

Como introducción se presentan los sistemas de control de acceso existentes en el mercado, los cuales pueden ser contratados como servicio para proteger un establecimiento de accesos no deseados. Como las empresas no muestran a detalle las metodologías utilizadas al desarrollar sus sistemas, en este informe se ideará un método que, tomando un conjunto de componentes, entregue el servicio de controlar los accesos a un establecimiento tomando todas las precauciones de seguridad pertinentes para este tipo de aplicaciones. Evaluando cada paso tales como la identificación, autenticación, autorización y finalmente registro, se investiga cómo será la mejor manera de replicar un sistema que ejecute toda esa secuencia con la mejor metodología posible.

El inicio de todo este proyecto se centra en la investigación de las tecnologías a utilizar, tanto sobre Moqui como los dispositivos que se fueron seleccionando a medida que se iba investigando más sobre el tema. Uno de los puntos que se privilegian para el diseño de este proyecto es entregar una interfaz simple para su uso diario, y también agilizar la autenticación del usuario. Por lo que la selección del diseño privilegia la usabilidad y la versatilidad que tendrá el sistema al adecuar su método de autenticación según las necesidades del administrador. Utilizando esta filosofía, Moqui llega como pilar fundamental al facilitar la implementación de un diseño amigable para el usuario. Además, se buscará simplificar la instalación del sistema en el establecimiento, por lo que a la par con Moqui, se seleccionó un dispositivo que ofrece fácil portabilidad y compatibilidad con múltiples componentes orientados a la autenticación, este dispositivo corresponde a la *Raspberry*.

Este tipo de sistemas deben garantizar la seguridad tanto de la información manejada como de la protección física al establecimiento. De esta manera se planteó un método que se centra en evitar que la comunicación lector-servidor sea interceptada por algún tercero. Se decide utilizar la encriptación de los mensajes como método de seguridad, la cual también es utilizada para proteger toda la información almacenada en la base de datos del servidor de Moqui, y además este mismo solicitará credenciales cada vez que un dispositivo intente extraer información confidencial de la empresa.

Finalmente se ejecuta la implementación física del sistema, momento en el cual toda la teoría planteada se puso en práctica. Las hipótesis del memorista fueron llevadas a cabo físicamente, que luego de cambios minúsculos, permitieron alcanzar un buen producto final, concluyendo con su respectivo análisis de los resultados obtenidos.

Para toda mi familia,
gracias por apoyarme y motivarme
a hacer lo que realmente me gusta.

Agradecimientos

Partir agradeciendo a mi padre y madre que siempre me apoyaron en todas mis decisiones, sin ese soporte nada de esto habría sido posible. También a mis hermanas que apenas entré a estudiar esta carrera, me mantuvieron con los conocimientos fresquitos cada vez que necesitaban ayuda relacionada a la informática. En general, agradecer a toda mi familia que siempre ha estado conmigo deseándome todo el éxito del mundo, bueno, miren dónde estoy escribiéndoles los agradecimientos.

También a todos mis compañeros, futuros colegas y amigos, quienes con todo su apoyo me motivaron a seguir por el mejor camino que pude haber elegido. Sin dejar afuera todos los profesores que me orientaron en toda mi carrera universitaria, funcionarios de la universidad que me posibilitaron disfrutar de cada espacio, tanto para el estudio como el ocio, todos esos momentos que nunca se olvidarán.

Gracias por todo.

Tabla de Contenido

Índice de Ilustraciones

Capítulo 1

Introducción

1.1. Antecedentes Generales

Los sistemas de control de acceso [1][2] se encargan de la autenticación y autorización del usuario que interactúa con un dispositivo. Éste solicita las credenciales de la persona para poder trabajar con ellas y verificar toda la información, éstas pueden ser contraseñas, números de identificación personal (PINs), huellas dactilares, tokens de seguridad o algún otro tipo de factor de autenticación.

El objetivo de este tipo de sistemas es minimizar el riesgo de accesos no autorizados tanto de la parte física, como de la lógica. El control de acceso es fundamental en la mayoría de las organizaciones que trabajan tanto con información delicada, sistemas computacionales, aplicaciones, información personal de los trabajadores o propiedad intelectual. Toda organización deberá proteger todo lo mencionado tanto físicamente como a nivel de red.

Los beneficios de los sistemas de control de acceso modernos son:

- **Permisos definidos para los empleados:** Los roles definen los cargos de cada empleado, por lo que desde un inicio sabe los limites de acceso que tiene en el establecimiento.
- **No son necesarias llaves tradicionales:** Las llaves tradicionales tienen una gran cantidad de desventajas. Como por ejemplo si se tiene distintas puertas que limitan el acceso a zonas de la oficina, el empleado debe tener una llave para cada entrada, lo cual a gran escala terminaría con un llavero gigante. Con un sistema moderno el empleado puede autenticarse sin objeto alguno (en el caso de dispositivos con PIN o lector biométrico por ejemplo) o bien funcionar con objetos *RFID* que podrá ser ingresado en cualquier lector del establecimiento.
- **Seguimiento de quién entra o sale del lugar:** El historial de los accesos realizados al recinto deberá ser documentado y almacenado en la base de datos de la empresa. Y su acceso debe estar restringido para administradores que posean las credenciales necesarias, ya que contiene información de los usuarios e instancias de interacción con cada lector de la empresa.
- **Protección contra accesos no deseados:** Cada dispositivo debe estar conectado al servidor de la empresa, donde se verifica si el usuario en cuestión tiene los permisos necesarios para tener acceso al lugar, de lo contrario se le negará el ingreso.

- **Libertad de los empleados:** Como el sistema nunca se apaga, cada empleado puede entrar a la hora que quiera y seguirá funcionando el seguimiento de los accesos totales.
- **Protección contra robo de información:** El servidor tiene protección en contra de accesos con malas intenciones. De esta manera se protegerá la información de los empleados que es almacenada por el sistema.

1.1.1. Componentes principales

Cada sistema de control de acceso está compuesto por:

- **Lectores:** Dispositivos encargados de procesar la credencial ingresada por el usuario y enviarla hacia el *Software* para que realice la autenticación.
- **Entrada:** Objeto que actúa como bloqueo de la entrada. Puede ser una puerta, barrera de un estacionamiento, etc.
- **Hardware de bloqueo:** Dispositivo encargado de asegurar que la entrada está bloqueada. Pueden ser seguros magnéticos, barras de empuje, etc.
- **Software:** Componente encargado de realizar toda la lógica. Tanto como la autenticación o autorización del usuario. También se encarga de almacenar toda la información de los perfiles.

1.1.2. Funcionamiento del sistema

Como primera acción del proceso se tiene a una persona ingresando una credencial en el dispositivo lector, paso que inicia la **Identificación** del usuario. El dispositivo lector deberá recibir la credencial de la persona, verificar que la credencial presenta un formato correcto y luego proceder con la consulta a los datos almacenados para saber si la persona puede ser reconocida por el sistema. Aquí es donde comienza el segundo paso denominado **Autenticación**, por lo que el lector debe enviar la información recogida de la persona (PIN, código encriptado de una tarjeta, etc.) y enviarla a la aplicación que se encarga de verificar si la credencial es reconocida por el sistema. Si todo sale correctamente el sistema debe haber obtenido los datos de esta persona, la cual previamente había sido ingresada al sistema.

Luego de reconocer a la persona como registrada, se debe verificar si tiene los permisos de acceso hacia el lugar determinado, este paso se conoce como **Autorización**. Si toda la información es correctamente procesada y se tienen los permisos, la persona tendrá acceso al lugar. En este paso (**Acceso**), la aplicación debe enviar la señal al seguro de la puerta para que este se libere y permita el acceso.

El siguiente paso es el **Registro**, el cual se lleva a cabo cuando toda la secuencia anterior ya se ha completado. La aplicación almacenará toda la información de las acciones anteriormente realizadas y, de esta manera, se tendrá el registro de lo sucedido junto con el perfil de la persona involucrada en la acción.

Finalmente tenemos el paso de **Revisión**, donde se puede hacer un análisis con los datos obtenidos. Aquí se podrá ver un resumen de los movimientos de las personas y los distintos intentos de acceso al lugar. Es posible que la aplicación genere automáticamente este paso

luego de recolectar cierta cantidad de información a lo largo de un día, semana o mes.

1.1.3. Moqui

Moqui[3] será el *framework* a utilizar en el desarrollo de este sistema, el cual permitirá implementar una interfaz de administración. Aprovechando que es un *framework open-source* se puede considerar el apoyo en diversos módulos ya existentes que puedan simplificar el desarrollo de funciones del sistema.

Una de las principales características de Moqui corresponde a que es modular, es decir, cuenta con diversos módulos que cumplen tareas definidas y que en un futuro pueden trabajar en conjunto para entregar un servicio nuevo. Tomando en cuenta esa característica, el producto a crear en esta memoria podría ser parte de un conjunto de soluciones que Moit podría proveer a un cliente o también operar como una solución de acceso físico independiente.

1.1.4. Escenario actual

Moit[4] es una empresa que se enfoca en la venta de *softwares* desarrollados en Moqui tomando el modelo *SaaS* (*Software as a Service*). Al ser parte de los proveedores principales de este tipo de servicios, cuentan con bastante experiencia en el desarrollo de aplicaciones con Moqui, permitiendo así una buena guía e introducción sobre este *framework* al momento de comenzar el análisis previo al desarrollo.

El lugar de trabajo designado para todos los trabajadores de la empresa es un establecimiento enfocado en el *Cowork*. Denominación utilizada para los ambientes que permiten el uso de sus dependencias sin importar la empresa. Ofrecen un ambiente laboral variado para toda persona que requiera de instalaciones para ejercer sus labores en un lugar enfocado en el trabajo y a la vez cómodo para separar la rutina laboral del hogar.

En estos espacios comparten trabajadores de múltiples empresas, por lo que es esencial hacer un filtrado de las personas que pueden acceder a ciertos espacios y quienes no. Incluso permitir accesos temporales es importante para este tipo de establecimientos. Actualmente Moit está utilizando, como dispositivo de acceso, un lector de huellas que permite el ingreso a los integrantes de esta compañía. Este sistema tiene la información de las personas que debe dejar entrar al lugar, aunque no entrega otro dato más que permitir el acceso o no. El sistema existente tiene 2 entes principales, el lector y la aplicación que abre la puerta, el primero envía solamente la señal para que la aplicación abra la puerta, sin confirmar que la persona que quiere entrar es quien dice ser y tampoco registrando el acceso. Este sistema almacena un conjunto de huellas las cuales no identifica como tal dentro de una base de datos, por lo que solo se encarga de verificar si el código (huella digital en este caso) coincide con alguna de las ingresadas anteriormente.

La implementación que se busca en esta memoria es incluir un sistema más elaborado que maneje perfiles de las personas que tengan acceso al lugar, entregando así información como ¿Quién es? y ¿A qué hora ingresó? Todo esto manejado a través de una plataforma de administración donde se habilitará o se removerá el acceso a ciertas personas.

Además, múltiples empresas requieren este tipo de sistemas para controlar el ingreso al lugar y guardar el respectivo registro, es por esto que el manejo de datos debe tener la capacidad de adaptarse a distintos escenarios y permitir la administración de una gran cantidad de perfiles, ofreciendo así un servicio más completo.

1.2. Objetivos

1.2.1. Objetivo general

Desarrollo de un prototipo que realice la autenticación presencial de personas utilizando un dispositivo determinado implementado en Moqui. Al momento en el que ocurre esta acción deberá confirmar si la persona tiene los permisos de acceso y luego debe almacenar la información recopilada en la base de datos del sistema.

1.2.2. Objetivos específicos

1. Investigar el nivel de seguridad y estructura de los diferentes tipos de dispositivos utilizados en cada método de autenticación.
2. Manejar las distintas implementaciones ya existentes para conocer más sobre la arquitectura del sistema.
3. Investigar sobre Moqui y el desarrollo de este tipo de sistemas en el *framework*.
4. Determinar cuáles son los tipos de dispositivos de autenticación capaces de enviar información a un sistema desarrollado en Moqui.
5. Entender la estructura del dispositivo especifico para hacer compatible toda la información a manejar con respecto a la identificación del usuario.
6. Establecer estructura, arquitectura y dispositivo óptimos para el desarrollo del sistema.
7. Implementar el prototipo en Moqui y vincularlo con el dispositivo elegido.
8. Generalizar la base de datos para que tenga la posibilidad de almacenar una gran cantidad de perfiles.
9. Realizar la comprobación del cifrado de la información y la exposición a posibles hackeos al sistema.
10. Implementar todos los tests que garanticen la seguridad y correcto funcionamiento del sistema.
11. Investigar sobre posible cambio del dispositivo de autenticación, con el fin de aumentar las funcionalidades del sistema y dando libertad de cambiar de método de autenticación.

Capítulo 2

Preparación

2.1. Definición de requerimientos

Las primeras semanas de trabajo consistieron en reuniones orientadas a definir cómo se comenzaría el desarrollo del sistema. Como primera decisión se definió que se implementaría la autenticación que funciona con códigos QR, método utilizado cada vez más por la simpleza de su utilización y además su sencilla portabilidad. En simultaneo se debía ir conociendo más sobre la utilización de Moqui y comenzar el desarrollo del módulo.

La interfaz del sistema es crucial para la administración de perfiles y vista de todos los datos almacenados en la base de datos. Ésta debe contar con un ingreso simple de usuarios, a los cuales deberá ser asignada una clave (código QR en este caso) para permitir el acceso. Cada usuario y puertas deberán contar con roles definidos, de esta manera se le permitirá al administrador filtrar ciertas zonas para permitir el ingreso de usuarios con privilegios distintos dentro del establecimiento.

Cada puerta deberá tener una *Raspberry* conectada a la misma red en la que está desplegada la aplicación de Moqui. Cada uno de los dispositivos poseerá una *key* que se utilizará para verificar si la respuesta del servidor proviene realmente del mismo. El punto más importante de este proceso es la seguridad del sistema, cada *key* manejada en la base de datos será encriptada y solo podrá ser desencriptada por los poseedores de la llave maestra, conocida solo por el servidor. El sistema operativo en cada Raspberry será encriptado para no permitir el acceso de terceros. Además, Moqui funciona con un sistema de usuarios propio, al cual solo tendrá acceso el administrador del sistema. La idea principal es emplear la mayor cantidad de métodos que aumenten la seguridad del sistema con tal de evitar posibles filtraciones de información personal de los usuarios ingresados en el sistema.

El servidor de Moqui deberá trabajar como creador y distribuidor de las llaves, de esta manera, al momento de ingresar nuevos perfiles en la interfaz de administración se deberán enviar, mediante correo electrónico, las llaves asignadas a los usuarios creados. Además cabe destacar que para la seguridad del sistema, cada puerta trabajará con una *key* específica, por lo que el administrador deberá incluir en el código de la *Raspberry* esta llave que permitirá la conexión a la API de Moqui y además como identificador de la puerta en el sistema.

Cada lector asignado a una puerta estará compuesto de 3 dispositivos esenciales: la *Rasp-*

berry, componente codificador de la credencial del usuario (en este caso será una cámara), un relé y un seguro eléctrico para la puerta. Este conjunto de componentes será crucial para establecer una comunicación lector-servidor y llevar a cabo la acción física de abrir y cerrar la puerta para dar ingreso al usuario.

2.2. Estudio Moqui Framework

Moqui será la herramienta a utilizar en el desarrollo de este sistema, permitiendo la implementación de la interfaz de administración. Aprovechando que es un *framework open-source* se puede considerar el apoyo en diversos módulos ya existentes que puedan simplificar el desarrollo de funciones del sistema.

Una de las principales características de Moqui corresponde a que es modular, es decir, cuenta con diversos módulos que cumplen tareas definidas y que en un futuro pueden trabajar en conjunto para entregar un servicio determinado. Tomando en cuenta esa característica, la solución a crear en esta memoria podría ser parte de un conjunto de soluciones que Moit podría proveer a un cliente o también operar como una solución de acceso físico independiente.

Moqui basa su funcionamiento en módulos, los cuales interactúan entre si brindando funcionalidades distintas para lograr un resultado final en conjunto. Estos componentes tendrán tareas distintas tanto como para la lógica como para la interfaz de la aplicación.

Los módulos base son:

- `mantle-usl`: Brinda las funcionalidades básicas del *framework*. Es la base para los servicios de Moqui.
- `mantle-udm`: Entrega las herramientas para crear un modelo de datos en la aplicación y además cargarlo de información.
- `SimpleScreens`: Provee plantillas para el *front-end* de las aplicaciones.

De la misma manera que Moqui tiene sus módulos esenciales para el desarrollo, Moit también tiene los propios:

- `Moquichile`: Brinda las herramientas que realizan las conversiones tanto de moneda, idioma, documentos nacionales, etc. Todo esto dirigido al desarrollo de aplicaciones que deben incluir información de Chile.
- `moit-utils`: Servicios solicitados por la mayoría de soluciones brindadas por Moit, de esta manera solo se incluye este módulo y se agiliza el desarrollo.

Los módulos que se desarrollarán tienen una estructura definida, las cuales a nivel general son las siguientes:

- `data`: Directorio donde se almacena la información tanto de la carga inicial de la aplicación, como los datos de prueba. Archivos externos a la estructura común de un proyecto de Moqui deben ser inicializados en esta carpeta (el ejemplo son las *templates* para correos electrónicos nombradas en este informe a continuación).

- `entity`: Contiene la estructura y modelos de datos que se cargarán durante el proceso de ejecución de Moqui.
- `screen`: Directorio que contiene todas las pantallas que forman la interfaz de la aplicación (*front-end*)
- `service`: Cada uno de los servicios que en conjunto formarán toda la lógica del proyecto deben estar implementados en esta carpeta (*back-end*).

2.3. Estudio sistemas existentes de control de acceso con administración de usuarios

Existen múltiples alternativas que ofrecen un servicio similar al sistema desarrollado en esta memoria, aunque es necesario señalar las características para hacer una comparativa y reconocer las ventajas de este sistema.

Sabemos que un sistema de control de acceso para personas es aquél que permite o restringe la entrada a una persona en determinado recinto mediante su identificación, es por esto que cada una de las empresas que ofrece el servicio proporciona una cantidad de lectores limitada para realizar la autenticación. Además la seguridad utilizada en estos sistemas no es para nada transparente con el cliente, por lo que nunca será conocedor del funcionamiento de esta parte del sistema.

IDenticard[5] ofrece servicios de control de automóviles, personas y/o pacientes. Cada una de estas opciones tiene distintos métodos de funcionamiento, tanto como utilizando chips *RFID* o lectores biométricos, por lo que su catálogo es bastante amplio y permite al cliente optar por su lector más cómodo. Sin embargo, todos estos servicios no cuentan con una aplicación que cuente con el historial de los accesos de usuarios, por lo que sería una de las grandes faltas del producto de esta empresa.

BeePro[6] ofrece un servicio de autenticación con distintos planes para contratar. El plan más completo permite administrar el acceso mediante un software accesible desde el ordenador o el móvil, además posee una cámara web que dejará registro fotográfico de las personas que están siendo registradas. Además cuenta con un historial en la nube de hasta 12 meses de registros, por lo que permitirá hacer análisis de la información recopilada durante bastante tiempo. La mayor falta de este servicio es que su único método de autenticación es mediante un escáner de cédula de identidad y pasaporte, por lo que limita al cliente que se recopile la información y permita el acceso a los usuarios utilizando su propio documento sin dar opción al administrador de elegir el método de autenticación más conveniente según el establecimiento.

SALTO Systems[7] es la empresa que ofrece el servicio más completo en comparación a las anteriores. Ellos fabrican sus propios lectores que funcionan con *RFID* o *PIN*, además cuentan con cerraduras electrónicas, candados e incluso taquillas con la marca de *SALTO*. Todos estos dispositivos estarán conectados a la misma red y cada uno de los lectores o cerraduras funcionarán con pilas y tendrán conexión directa con *SALTO Space*, plataforma de gestión integral de accesos inteligente bastante completa que ofrece todo el historial de acceso de los dispositivos accionados por algún usuario, incluso los intentos fallidos serán

registrados en la plataforma. En resumen, es el sistema que más destaca, aunque tiene la gran limitación de que solo será compatible con los productos fabricados por ellos mismos y que además trabajan con dispositivos *RFID* o que funcionan mediante *PIN's*, lo cual limita bastante las elecciones del cliente.

Considerando cada uno de los puntos tratados en los resúmenes de cada empresa, el nuevo sistema a desarrollar contará con libertad de elección del cliente sobre cuál método le parece más cómodo para realizar la autenticación (códigos *QR*, dispositivos *RFID* e incluso lectores biométricos) dado que se trabajará con un generador de *key's* si es necesario proporcionarle alguna al usuario. En el caso de que la *key* se proporcione mediante el lector biométrico se podrá utilizar esta como la llave vinculada a los usuarios. Los dispositivos permitidos deberán ser compatibles con la *Raspberry Pi* asignada a esa puerta, por lo que se tiene una amplia gama de posibilidades.

Además la aplicación de *Moqui* permitirá a los administradores contar con un historial amplio de los accesos efectivos o intentos de acceso en cada una de las puertas vinculadas al sistema. La información máxima tendrá como limite la capacidad de la base de datos, por lo que estará definida según el almacenamiento disponible del servidor.

Y finalmente el punto más fuerte del sistema es que será *open-source*, por lo que el cliente tendrá la libertad de poder modificar el sistema como estime conveniente, es decir, si hay algún punto de la seguridad del sistema o pantalla de la aplicación que no le parezca correcto, podrá modificarlo. O si tan solo quiere entender el funcionamiento de algún aspecto del sistema, tiene acceso a todo lo desarrollado y que se especifica en esta memoria.

Capítulo 3

Diseño

3.1. Arquitectura

La seguridad del sistema fue el punto más importante al plantear la arquitectura de comunicación entre los dispositivos, es por esto que se decidió trabajar con encriptación tanto en la comunicación servidor-*Raspberry* como en el sistema operativo *Raspbian* (instalado en las *Raspberry*).

Tomando en cuenta que Moqui permite trabajar con servicios y ejecutar consultas a la *API* dentro de la misma red, se decidió trabajar con una *Raspberry* por cada puerta que se desea agregar al sistema. Cada uno de estos dispositivos tendrá su propia *key* que le permitirá hacer las consultas pertinentes para autenticar al usuario, por lo que se permitirá al administrador poseer la cantidad de dispositivos que estime conveniente según el establecimiento.

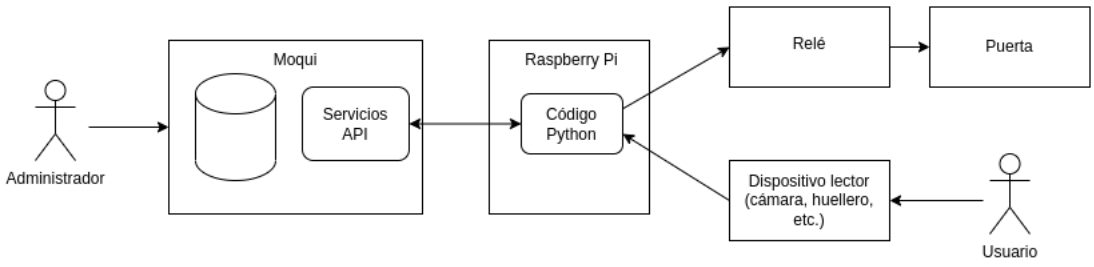


Figura 3.1: Arquitectura del sistema

3.2. Base de Datos

La base de datos a utilizar para almacenar los datos de cada usuario será proporcionada por *Moqui*, por lo que dentro del mismo *framework* se establecieron las entidades, llaves y servicios del sistema. Las entidades principales son *users*, *roles* y *doors*, cada una de estas tendran la información de los usuarios, roles y puertas respectivamente. Además se utilizarán entidades relacionales para establecer los vínculos entre valores de las entidades principales (estas solo contarán con el par de *id's* de dos entidades).

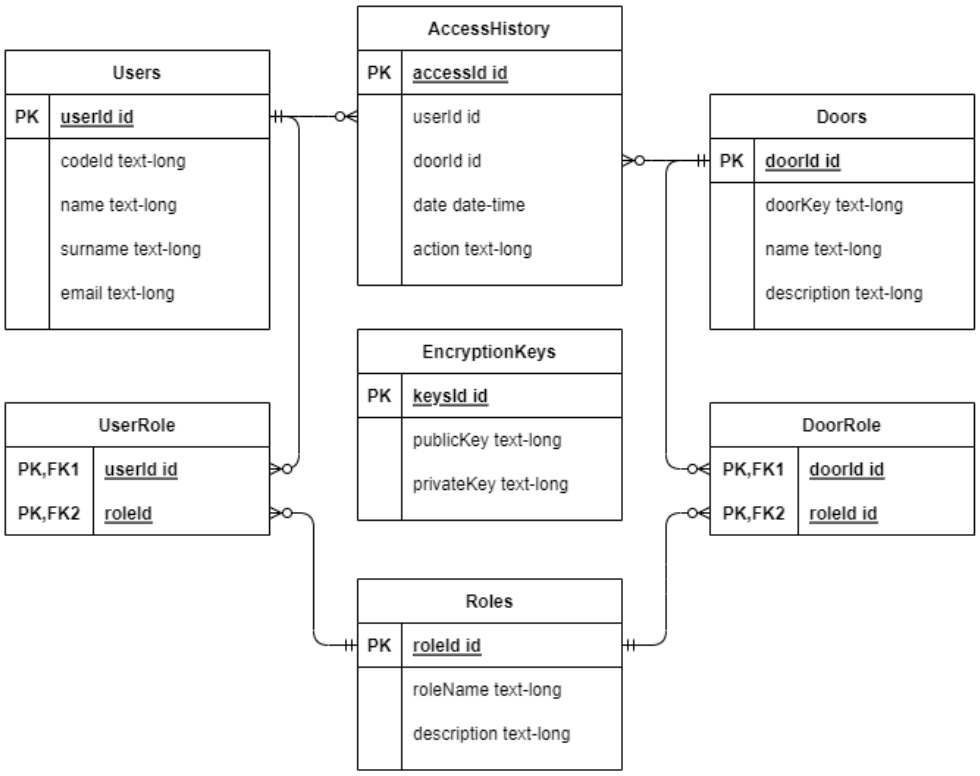


Figura 3.2: Diagrama de entidades

Las entidades principales tendrán una llave primaria asignada y las entidades relacionales tendrán referencia a esos identificadores mediante llaves foráneas. La única entidad que no tendrá ninguna referencia será la que contiene las llaves del sistema de encriptación, y sus únicos valores serán la llave pública y la privada. Estas llaves serán únicas en el sistema, por lo que cuando un servicio implementado las solicite se utilizarán las mismas llaves para autenticar al usuario y verificar que el dispositivo que se quiere comunicar con la *API* posea las credenciales correctas.

3.3. Programas

3.3.1. Interfaz

La aplicación web para administrar el sistema posee una pantalla designada a cada entidad para ver cada detalle de los valores, de esta manera se podrán agregar nuevos valores, editar existentes y eliminar datos. Es bastante simple realizar las acciones mencionadas, ya que tienen un diseño bastante estándar y guiado como se muestra en la Figura 3.3.

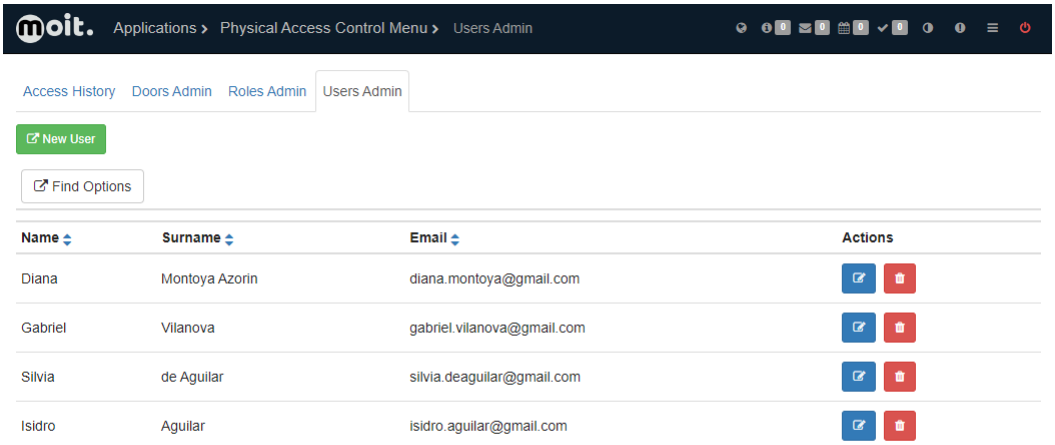


Figura 3.3: Ejemplo de tabla en la interfaz

En la zona superior de cada pantalla se presentan pestañas que permitirán al usuario acceder a todas las opciones que brinda el sistema al administrador. Las opciones son:

3.3.1.1. *Access History* (Historial de accesos)

Pantalla principal en la cual está la tabla de historial de interacciones con el sistema, lugar donde se encontrarán los intentos de acceso erróneos y los que se realizaron satisfactoriamente (ejemplo en Figura 5.8). La información en cada acceso será:

- **Nombre usuario**: Nombre de la persona que intentó ingresar al lugar. En el caso de que no se haya podido autenticar a la persona se indicará como "*Sin autenticar*".
- **Puerta**: Nombre de la puerta por la cual intentó ingresar el usuario. Este apodo debe ser asignado por el administrador al ingresar la puerta al sistema.
- **Hora**: Tiempo exacto en el que el usuario interactuó con el sistema (solo puede hacerlo mediante algún lector vinculado).
- **Acción**: Indica lo ocurrido en la autenticación. Las opciones serán:
  - *Ingreso satisfactorio*: Usuario ingresa y cumple con los requisitos.
  - *No ingresado al sistema*: La llave ingresada no corresponde a ninguna de las existentes en el sistema.
  - *No posee los permisos*: Usuario no tiene asignados los roles que permiten el acceso por esa puerta.
  - *Código de puerta incorrecto*: La puerta que intentó hacer la consulta no tiene una llave que coincida con alguna existente en la base de datos.

3.3.1.2. *Doors Admin* (Administrador de puertas)

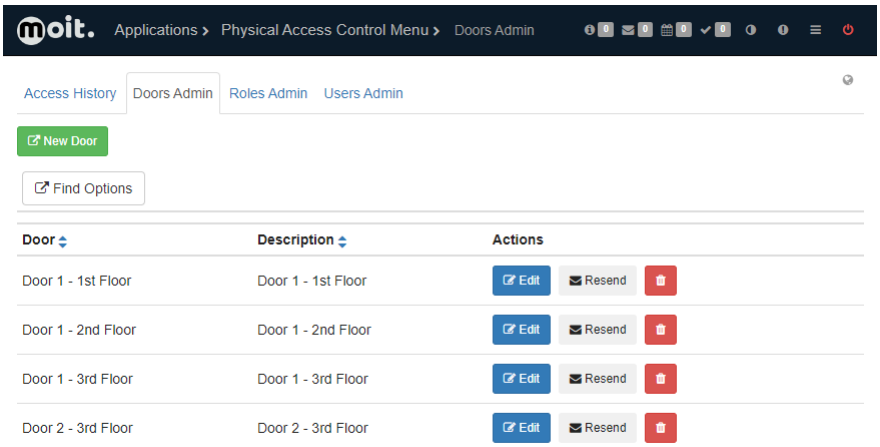


Figura 3.4: Administrador de puertas

Pantalla que permitirá administrar todas las puertas ingresadas en el sistema. Como botón principal se presenta *New Door*, el cual sirve para ingresar una nueva puerta al sistema. Los parámetros solicitados serán los que se muestran en la Figura 3.5. Cabe destacar que el correo electrónico requerido servirá para enviar las credenciales asignadas a esa nueva puerta (correo que se envía al momento de enviar el formulario).

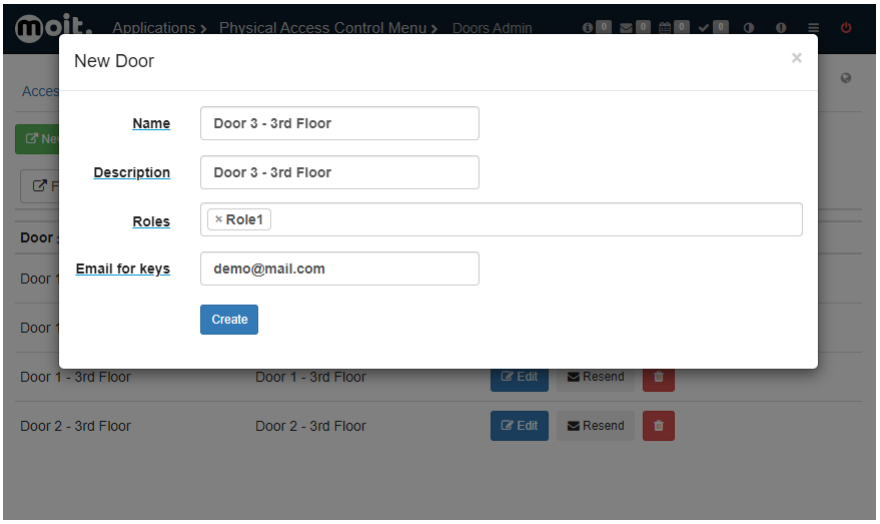


Figura 3.5: Ventana para crear una nueva puerta

Para los valores presentes en la tabla se muestra una columna con 3 botones distintos, los cuales permitirán editar la fila respectiva a alguna puerta, reenviar el correo electrónico con las credenciales o eliminar la puerta del sistema. Al presionar el botón para editar se accionará una ventana que solicitará los campos de la puerta en cuestión (Figura 3.6). En el caso de que se quiera reenviar el correo con las credenciales o eliminar el valor del sistema, se le preguntará al usuario con una ventana de confirmación si realmente quiere realizar lo solicitado.

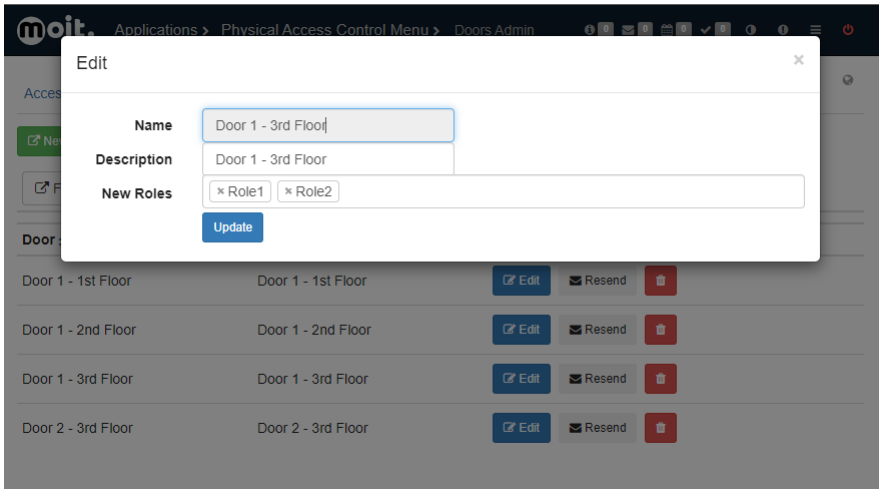


Figura 3.6: Ventana para editar valores de una puerta

Las puertas no tendrán asignado un correo electrónico en la base de datos, por lo que cada vez que se quiera obtener las credenciales se le solicitará al administrador ingresar una dirección de correo electrónico para hacerle llegar las credenciales.

3.3.1.3. *Roles Admin* (Administrador de roles)

En esta pestaña se podrán administrar todos los roles ingresados al sistema, la cual, de manera similar a las puertas, tiene un botón *New Role* que acciona la ventana que solicitará los valores de un rol para ingresarlo al sistema. Además cada fila tiene los botones de edición y eliminación respectivos (con una interfaz similar a la Figura 3.6 y además, incluyendo la ventana de confirmación por parte del administrador si quiere eliminar un rol).

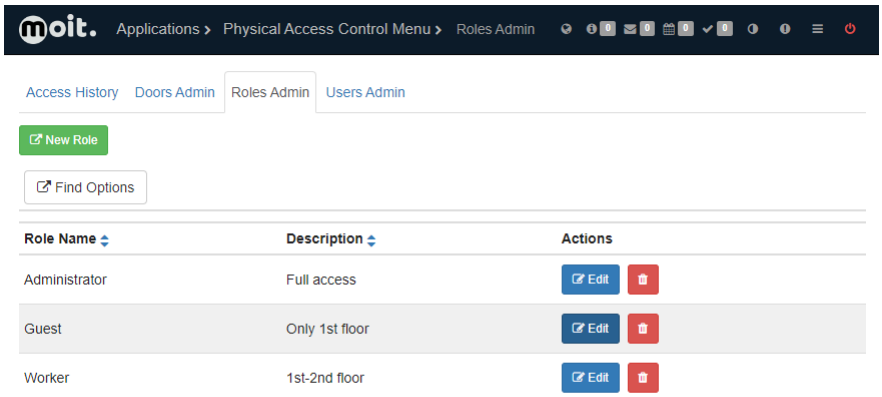


Figura 3.7: Administrador de roles

3.3.1.4. *Users Admin* (Administrador de usuarios)

Finalmente se presenta la pestaña que permite la administración de los usuarios. De igual manera contiene un botón para crear un usuario nuevo y además editar o eliminar alguna fila de la tabla. Sin embargo, se agrega un botón extra junto a la dirección de correo electrónico del usuario, el cual servirá para reenviar el *mail* con las credenciales del usuario. El destinatario será la dirección almacenada en los valores del usuario en cuestión.

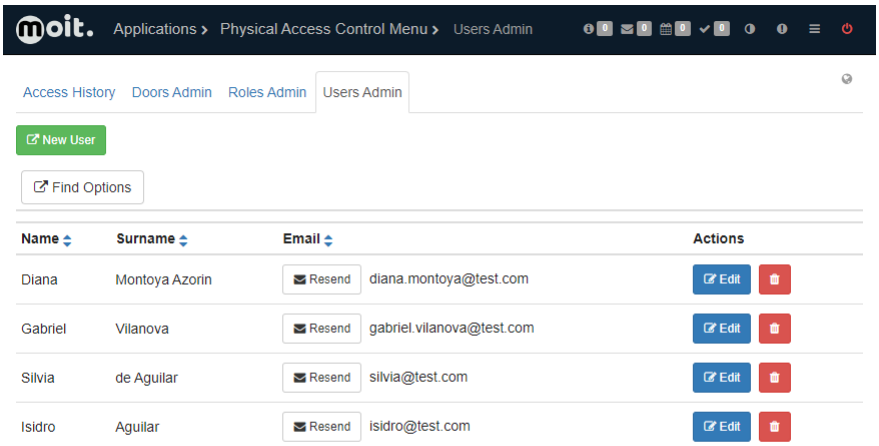


Figura 3.8: Administrador de usuarios

3.3.2. Correos electrónicos con credenciales

Existen 2 correos que se enviarán por parte del sistema: el dirigido al usuario (con el código QR) y el que contiene las credenciales de una puerta.

3.3.2.1. Código QR

En este correo se indicará con un breve mensaje las intenciones de ese correo y adjunto a este irá un archivo `.jpeg` con el código QR.

(a) Mensaje en correo electrónico de entrega de código QR

(b) Ejemplo de código QR (archivo `.jpeg`)

Figura 3.9: Ejemplo de imagen múltiple.

3.3.2.2. Credenciales para una puerta

Muy similar al anterior correo electrónico descrito, se señala un breve mensaje para describir las intenciones del correo. La gran diferencia reside en que las credenciales para la puerta se encuentran escritos en el mismo mensaje como se muestra en la Figura 3.10.

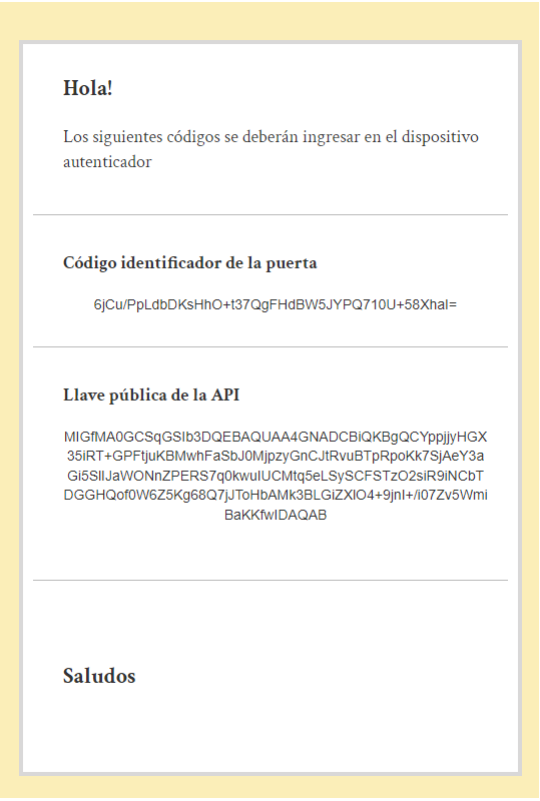


Figura 3.10: Correo electrónico con credenciales para la puerta

3.3.3. Lector de QR en *Raspberry*

Este método de autenticación fue el seleccionado para la realización de esta memoria por la simple manipulación y envío de credenciales asignadas a cada usuario. La idea principal es abstraer el manejo de llaves para que sean utilizadas con otros métodos de autenticación.

En este caso se facilita un código *Python* para cada *Raspberry*, punto importante para identificar cada puerta y además brindarle el permiso para consultar la credencial y roles del usuario. En el código se tendrá que ingresar la llave asignada desde el servidor (creada al momento de ingresar la puerta en la base de datos) y la llave pública señaladas en el correo electrónico enviado desde el servidor (Figura 3.10).

El programa lector de códigos QR da el *feedback* de la lectura mediante la terminal de cada *Raspberry*, en la cual se notificará cada acción efectuada con el servidor apenas el usuario interactúe con el dispositivo. Como siguiente mensaje aparecerá la notificación si el usuario tiene los permisos correspondientes para acceder por esa puerta. Luego de mostrar el mensaje volverá a solicitar el siguiente código QR para solicitar el acceso.

Todo el proceso es realizado por los siguientes archivos implementados en *Python*:

- `constants.py`: Aquí se deben ingresar todas las variables necesarias para el funcionamiento del código en la *Raspberry*. Tales como url de la API, llave de la puerta, llave pública del servidor (ligado a la encriptación), usuario para consultar a la *API*, contraseña para consultar a la *API* y el canal *GPIO* asignado al relé.
- `main.py`: Archivo principal del programa. Llama a la funciones definidas en los otros archivos para realizar la completa autenticación del usuario.
- `qrScanner.py`: Ingresando una imagen realiza el escaneo y entrega el código QR decodificado en caso de encontrar alguno en la imagen.
- `relay.py`: Permite activar y desactivar el relé según un canal determinado en el archivo `constants.py`.
- `verifier.py`: Utilizando la llave de la puerta, la llave pública del servidor y la firma entregada como respuesta de la *API*, verifica si esa firma es proveniente del servidor correspondiente.

3.3.4. Servicios del servidor

El servidor de Moqui tiene múltiples servicios implementados, los cuales son utilizados en cada una de las pantallas para administrar todos los parámetros de la base de datos. La mayoría corresponden a servicios utilizados para obtener, determinar o actualizar los valores de las entidades.

Para la autenticación de un usuario se utiliza una *API REST*, la cual se despliega utilizando el mismo servidor de moqui y permite a los dispositivos de la misma red acceder a esta consulta. Esta *API* da acceso a solo un servicio, `userHasAccess`, el cual determinará mediante múltiples verificaciones si el usuario ingresado corresponde a uno ya existente en la base de datos (se profundizará más en los métodos utilizados en la siguiente sección). Cabe

destacar que a este servicio solo pueden acceder los dispositivos que sean conocedores del usuario y contraseña de administrador, de lo contrario no podrán hacer la consulta y recibirán un error como respuesta.

3.3.5. Seguridad y encriptación de contraseñas

Si bien todo el sistema está pensado para restringir el acceso a personas sin los permisos a un lugar determinado, no hay que esperar que todo funcione sin tomar las precauciones pertinentes. Una de los mejores métodos es ponerse en el peor caso posible, y en este caso sería que gente con mayor conocimiento de informática intente intervenir la comunicación entre la *Raspberry* y el servidor, con tal de obtener datos personales de los usuarios y las llaves necesarias para abrir la puerta. Es por esto que se diseñó un método que aumenta un gran porcentaje la seguridad en esta comunicación con la *API*.

Uno de los métodos más utilizados en la informática para evitar que un mensaje sea interpretado por otros usuarios es la **encriptación**. Procedimiento mediante el cual los archivos, o cualquier tipo de documento, se vuelven completamente ilegibles gracias a un algoritmo que desordena sus componentes.

En criptografía, una clave es un dato que se utiliza para codificar los datos de manera que parezcan aleatorios; suele ser un número grande, o una cadena de números y letras. Cuando los datos sin encriptar, también llamados texto plano, se introducen en un algoritmo de encriptación utilizando la clave, el texto plano aparece en el otro lado como datos de aspecto aleatorio. Sin embargo, cualquiera que tenga la clave adecuada para desencriptar los datos puede volver a ponerlos en forma de texto plano.

El método de encriptación seleccionado para esta memoria se denomina **encriptación asimétrica**, el cual permite encriptar datos con dos claves diferentes y hace que una de las claves, la pública, esté disponible para que cualquiera pueda utilizarla. La otra clave se conoce como clave privada. Los datos encriptados con la clave pública solo pueden desencriptarse con la clave privada, y los datos encriptados con la clave privada solo pueden desencriptarse con la clave pública.

Utilizando esta filosofía podemos llevarlo a nuestro sistema, donde tenemos un servidor, conocedor de las claves de cada usuario en texto plano, y el lector de claves en la puerta, quien no necesariamente debe conocer las claves de cada usuario. Como el lector solo actúa como un intermediario en la comunicación usuario-servidor, no necesita conocer realmente la clave de este usuario, es por esto que el servidor, al ingresar cada usuario, creará una clave para este y además encriptará este texto para finalmente enviar el Código QR al usuario con la clave ya encriptada (Figura 3.11).

Como el usuario solo deberá manipular la imágen del código QR, tampoco necesita conocer su propia clave, es por esto que el único conocedor de todas las claves siempre será el servidor.

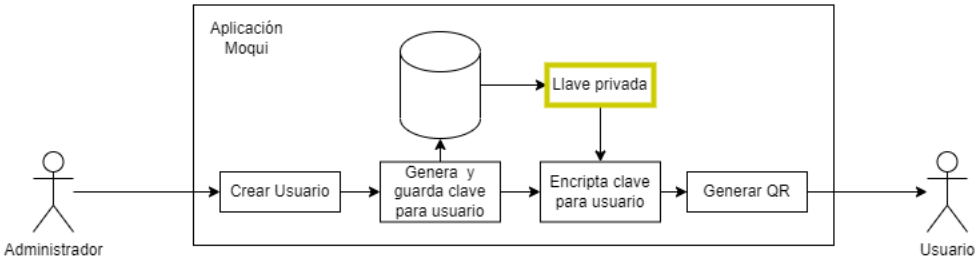


Figura 3.11: Diagrama de flujo creación de usuario

En el momento de que el usuario quiera autenticarse e intentar ingresar al establecimiento, ingresará su código encriptado al lector, el cual se comunicará con el servidor enviando el código mediante la *API* y además el propio código de la puerta para identificar sus permisos. Cabe destacar que Moqui limita el acceso a su *API* con la ayuda de un usuario y contraseña establecidos anteriormente por el administrador, por lo que para hacer cualquier consulta al servidor se necesitan estas credenciales.

Cuando el servidor reciba la consulta de alguno de los lectores del establecimiento desencriptará el código recibido y consultará en su base de datos si la clave coincide con alguna de los usuarios ingresados. Luego de verificar su existencia y además consultar si los roles de la puerta en cuestión coinciden con los que posee el usuario, el servidor creará una **firma**.

En la encriptación asimétrica, como se mencionó anteriormente, se puede desencriptar con una llave lo encriptado con la contraria, por lo que el lector, luego de hacer la consulta al servidor, esperará la firma de este para verificar si realmente la respuesta de otorgar acceso al usuario viene realmente del servidor de Moqui designado para el sistema.

Si el usuario finalmente si tiene los permisos y entregó el código correcto, el servidor enviará su firma al lector, para que este, utilizando la llave pública, confirme la veracidad de esa firma criptográfica y ejecute los comandos para abrir la puerta asignada a ese lector.

Toda la secuencia se puede interpretar, sin tanto detalle, como es descrito en la figura 3.12, donde se muestran todas los pasos que deben pasar las credenciales ingresadas para poder garantizar el acceso al usuario. Si al menos una de estas consultas no está correcta no será posible que la persona pueda entrar por esa puerta.

Analizando las Figuras 3.11 y 3.12, se tiene que el servidor de Moqui utiliza la llave privada almacenada en la base de datos para encriptar, y el dispositivo autenticador necesita la llave pública para verificar la procedencia de la respuesta de la *API* y confirmar que viene del servidor correspondiente.

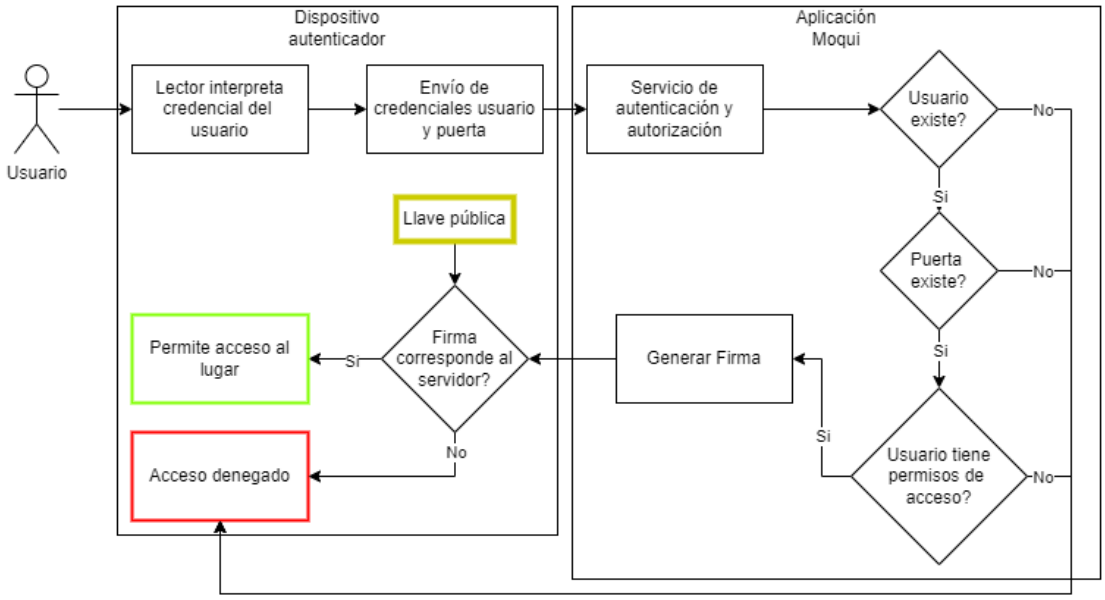


Figura 3.12: Diagrama de flujo autenticación y autorización de usuario

Capítulo 4

Implementación

4.1. Instalación inicial Moqui Framework

Comenzando con la preparación de los prerrequisitos tenemos la instalación de *JDK 11*, *IntelliJ IDEA* y *GIT*. Ya con esto listo se procede con la preparación de la llave SSH que permitirá clonar el repositorio `moit-framework` que se encuentra en *GitLab*, esta parte consistió en generar esta llave mediante el comando `ssh-keygen -t rsa -b 2048 -C «comment>"`, el cual generará la llave que sería posteriormente ingresada en la plataforma de *GitLab* y que ayudará con la autenticación solicitada por el *software*. Con todos los pasos listos se clonó finalmente el repositorio señalado anteriormente.

Para continuar con la realización del `build` de la aplicación fue necesario utilizar la rama `moit-devel`, la cual contiene la última versión para desarrollo creada por Moit. Ya con esto listo, el siguiente paso fue utilizar *Gradlew*, un sistema de automatización de construcción de código, el cual ayudaría con la descarga de todos los componentes solicitados para crear el `build` de la aplicación. En este paso se importan algunos repositorios extra que la aplicación necesita, como el `moqui-runtime`.

Finalmente es posible obtener el `build` de la aplicación, nuevamente se utilizó *Gradlew* para generar todos los archivos, dentro de los cuales está el archivo `.war`, con el cual se podrá ejecutar la aplicación.

Para concluir se utilizó `gradlew load` para poblar la base de datos local y luego `gradlew run` para iniciar el servidor y empezar a utilizar la aplicación. Finalmente, con todo lo anterior realizado con éxito, se tiene acceso a todas las plataformas de ejemplo.

4.2. Instalación de otros servicios

4.2.1. Raspbian OS

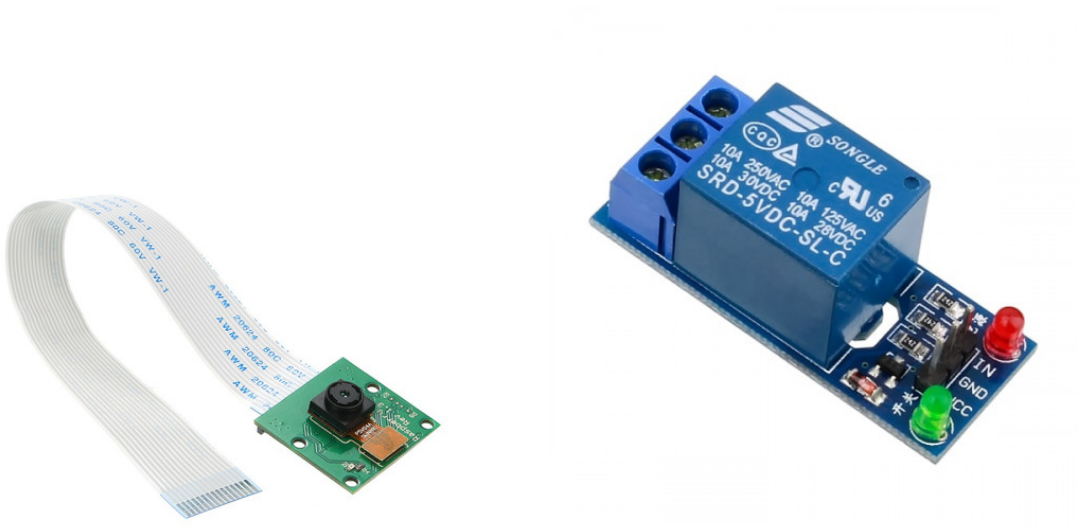
El dispositivo principal asignado a cada puerta es una *Raspberry Pi*, la cual estará funcionando bajo el sistema operativo *Raspbian*. Este sistema operativo estará encriptado para evitar posibles filtraciones de información delicada, para lo cual se utilizó *LUKS*[8], un método que permite encriptar la partición `root` de la *Raspberry*. Este paso crucial para la seguridad del sistema dado que estos dispositivos manejarán *keys* que dan acceso a la API de *Moqui*.

4.2.2. Conexión de dispositivos en *Raspberry*

Si bien el sistema desarrollado en esta memoria busca proporcionar un sistema de control de seguridad que funcione con el método de autenticación que el administrador del establecimiento estime necesario, se implementó todo el sistema con un lector específico para luego abstraer la utilización de llaves a los otros autenticadores.

El sistema seleccionado es la autenticación mediante códigos QR, por lo que el dispositivo *Raspberry* de cada puerta debe escanear el código que ingresarán los usuarios. El escaneo será hecho digitalmente utilizando una cámara *Raspberry* conectado mediante un cable *Flex*.

Además la misma *Raspberry* debe estar conectada al seguro de la puerta mediante un relé. Este será accionado desde el código *Python* del sistema y al activarse, el seguro de la puerta se abrirá. En el caso de que se corte el circuito por parte del relé, el seguro se cerrará.



(a) Cámara *Raspberry* (b) Relé de 5 *Volts*

Figura 4.1: Ejemplo de imagen múltiple.

Cada uno de estos dispositivos tienen puertos específicos donde conectarlos. Para este proyecto se utilizaron los siguientes conectores para el funcionamiento del sistema. En el caso del relé tenemos 3 conectores, *VCC*, *GND* e *IN*, los cuales corresponden a la fuente de poder, tierra y conector al *GPIO* respectivamente. Estos pines se conectaron mediante cables *Dupont* hembra-hembra.

Para el caso de la cámara se debe conectar su cable *Flex* directamente al puerto señalado en la Figura 4.2.

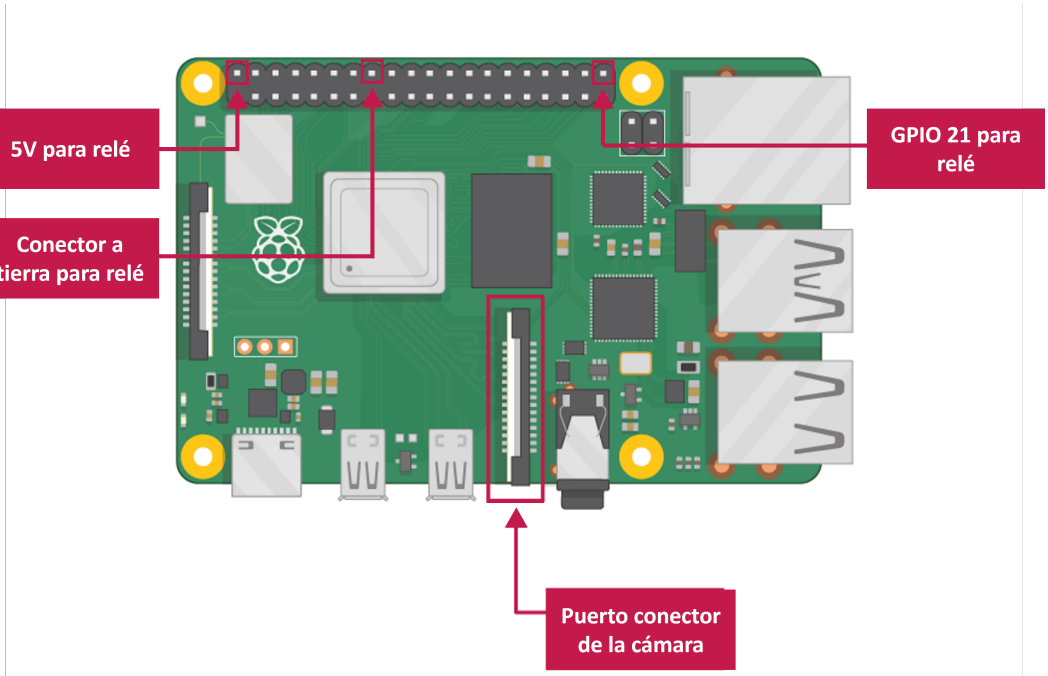


Figura 4.2: Diagrama completo de conexiones en *Raspberry*

4.3. Generadores de llaves y códigos QR

Para llevar a cabo la encriptación y desencriptación de las claves que maneja el sistema se utilizó la librería *Cryptography* y *Security* de *Java*. Además, para generar códigos QR se importó la librería *QRGen*[9] implementada por Ken Gullaksen y publicada en *GitHub*.

KeyHelper

Clase encargada de generar claves para puertas y usuarios, además es permite encriptar estas mismas utilizando las librerías mencionadas anteriormente.

Para generar las claves se utiliza `keyGenerator` de *Java*, al cual se le ingresa el algoritmo *AES* (*Advanced Encryption Standard*).

En los años '90, *DES* era el algoritmo más utilizado para la creación de claves, pero a medida que la tecnología iba progresando se fue volviendo inseguro dado a que las llaves estaban limitadas a un tamaño de 56-bit, lo cual es bastante pequeño en comparación a lo existente hoy en día y es bastante inseguro. Es por esto que el Instituto Nacional de Estándares y Tecnología (*NIST*), en el año 2000 presentó *AES*, un nuevo algoritmo generador de llaves que funciona con bloques de 128 bits y permite un largo de 128, 192 y 256 bits. En la actualidad se ha vuelto uno de los algoritmos criptográficos más utilizados.

Por lo tanto, se seleccionó *AES* para generar las llaves asignadas a los usuarios y las puertas. Estas mismas solo deben ser conocidas por el servidor, es por esto que esta misma clase

ofrece funciones para encriptar y desencriptar estas claves para permitir la manipulación de estas por parte de los usuarios (y evitar que caigan en manos de terceros).

Código 4.1: Generador de claves

```
1       ...
2       keyGenerator = KeyGenerator.getInstance("AES");
3       keyBitSize = 256;
4       keyGenerator.init(keyBitSize, random);
5       ...
6       return keyGenerator.generateKey();
7
```

Utilizando la misma librería de *Java* se solicitará un `KeyPairGenerator`, el cual, utilizando el algoritmo *RSA*, entregará el par de llaves para llevar a cabo el método criptográfico asimétrico. Método el cual es explicado su funcionamiento y porqué se seleccionó para este sistema en la sección 3.3.5.

Código 4.2: Generador de llaves para sistema de encriptación

```
1       ...
2       keyGen = KeyPairGenerator.getInstance("RSA");
3       keyGen.initialize(1024);
4       ...
5       KeyPair pair = keyGen.generateKeyPair();
6       PrivateKey privateKey = pair.getPrivate();
7       PublicKey publicKey = pair.getPublic();
8       ...
9
```

Finalmente para encriptar utilizando las llaves pública y privada, se crea un objeto de la clase `Cipher`, el cual, ingresando la transformación que el programador quiera, podrá encriptar y desencriptar un `string` y un arreglo de bytes respectivamente. Para este proyecto se define *RSA* como algoritmo y el modo *ECB* en conjunto de *PKCS1Padding*. Para este método de encriptación se solicita ingresar un tipo de *padding*, dado que se solicita que el *input* (en este caso las claves) sea un múltiplo exacto del tamaño de los bloques del algoritmo definido.

Finalmente como todas las llaves creadas por *Java* son de tipo `byte array`, se deberá utilizar la librería `Base64`. Esquema utilizado en la criptografía para tomar información binaria y transformarla en datos fáciles de interpretar, como por ejemplo, un *string*. De esta manera permitirá el almacenamiento de las claves en la base de datos evitando errores de formato.

QrHelper

La librería implementada por *kenglxn* tiene diversas funciones muy útiles para generar códigos QR. Lo principal es especificar los siguientes parámetros para el generador:

- Formato de destino

- Tamaño en píxeles del código a generar
- Información que se busca transformar

En este proyecto se define el formato de destino como un archivo de imagen `.jpg` con la finalidad de adjuntar esta misma al correo electrónico y simplificar la acción de poder abrir el archivo por parte del usuario. El tamaño considerado para una buena lectura por parte del escáner es 250x250. Y finalmente la información que se busca cifrar será un `string`.

Código 4.3: Generador de códigos QR

```
...
QRCode.from(stringCode).to(ImageType.JPG).withSize(250, 250).stream();
...

```

4.4. Autenticador en *Raspberry*

Como ya fue indicado anteriormente, el código de `main.py` se encarga primero de inicializar el escáner de códigos QR (`qrScanner.py`) y comunicarse con la *API* del servidor, luego dependiendo de la respuesta del mismo se encarga de verificar si el mensaje de respuesta es proveniente del servidor de Moqui (`verifier.py`). Finalmente en el caso de que el usuario se haya autenticado satisfactoriamente, se acciona el relé para dar acceso al establecimiento (`relay.py`).

qrScanner.py

Para decodificar los códigos QR se utilizará `OpenCV`, una librería de visión artificial y *machine learning*, la cual permitirá el procesamiento en cada *frame* de la imagen obtenida desde la cámara.

La segunda librería será `pzybar`, la cual permite la lectura de códigos de barra y QR de una imagen dada.

Código 4.4: Pseudocódigo lector códigos QR Python

```
# qrScanner.py
def escanearQR(image):
    imagenDecifrada = pyzbar.decodificar(image)

    for obj in codigo:
        codigo = obj.data.decode("utf-8")
        return codigo

# main.py
camara = conectarCamara()
while True:
```

```
12          frame = camara.obtenerFrame()
13          codigo = qrScanner.escanearQR(frame)
14
```

El procedimiento consiste en utilizar `OpenCV` para obtener cada *frame* de la cámara conectada a la *Raspberry*, luego con `pyzbar` se decodifica cada *frame* para concluir con el `string` equivalente a lo presentado por el usuario.

relay.py

Este código entrega 2 funciones que conectarán al programa con el GPIO indicado por el administrador en las constantes. Este canal debe ser exactamente al que está conectado físicamente el relé, para obtener así un correcto manejo de este dispositivo por parte del programa. Para establecer la conexión con el puerto GPIO se utilizó la librería `RPi.GPIO`, la cual permite acceso directo a estos puertos y su manejo.

verifier.py

El punto más importante para la autenticación está en este archivo, el cual se encarga de dar el *check* final y comprobar si realmente la llave de respuesta de la API corresponde a la firma creada por el servidor con las llaves de encriptación establecidas. Para lograr este procedimiento se necesitan 3 parámetros, la firma, la llave pública y la llave de la puerta, con estos valores y utilizando la librería `Crypto` de *Python* se puede importar una llave *RSA*, ingresar la llave de la puerta con *SHA256* (conjunto de funciones hash criptográficas) y finalmente utilizar un *verifier* creado con la función `PKCS1v15` (la cual permite establecer nuestro string de llave pública como *Public-Key Cryptography Standards*), finalmente utilizando todos estos valores, la librería indicará la veracidad de la firma ingresada.

Código 4.5: Verificador de firma en programa de Python

```
1
2       key = RSA.importKey(public_key_decoded)
3       hasher = SHA256.new(doorKey.encode())
4       verifier = PKCS1_v1_5.new(key)
5       return verifier.verify(hasher, signb64decode)
6
```

4.5. Inicialización de parámetros para el lector en la *Raspberry*

Las *Raspberry* estarán ejecutando un código implementado en el lenguaje *Python* que efectuará la consulta a la API para solicitar los permisos del usuario y dar acceso al recinto. Además el dispositivo lector (cámara, huellero, etc.) estará siendo utilizado desde este mismo código para realizar la autenticación.

Para ejecutar correctamente el programa en cada *Raspberry* se cuenta con el archivo `constants.py`, en el cual deben ser ingresadas las credenciales de cada puerta.

Código 4.6: Constantes establecidas en `constants.py`

```

    API_URL = 'http://<serverHost>:<serverPort>/rest/s1/physical-control-access/
↪ userhasaccess'
    DOOR_KEY = '<doorKey>'
    PUBLIC_KEY = '<publicKey>'
    USER_SERVER = '<userServer>'
    PASS_SERVER = '<passServer>'
    RELAY_CHANNEL = <relayGPIOChannel>

```

Para el administrador que está configurando una nueva puerta en el sistema, solo deberá llenar los campos señalados en el Código 4.6, luego bastará con ejecutar `main.py` para que funcione el programa correctamente.

Además de los parámetros para el programa se necesitarán las librerías `opencv-python`, `pyzbar` y `pycrypto`. Cabe destacar que en algunos casos hay que actualizar la librería `numpy` ya instalada en el SO *Raspbian*.

4.6. Servicios y Módulo de Moqui

Moqui permite establecer servicios que siguen ciertos procedimientos determinados por el programador, los cuales pueden ser llamados desde múltiples partes del código con tal de evitar tener código repetido. Los servicios principales creados para este sistema son los que se utilizan para crear, editar o eliminar parámetros de cada entidad. Cada uno de los archivos de la carpeta `services` contiene los servicios dirigidos a la entidad estipulada en el nombre.

UsersServices.xml

Para la creación de esta entidad se utiliza la función `getRandomKey()` de la clase `KeyHelper`, el cual proporcionará una clave aleatoria para asignarla a este nuevo usuario.

Código 4.7: Generador de claves para un usuario

```
    import cl.pca.KeyHelper

    def keyHelper = new KeyHelper()

    def newKey = keyHelper.getRandomKey()
    def stringFromKey = keyHelper.getStringFromKeyForDB(newKey)

```

La clave obtenida es un `byte array`, es por esto que se le solicitará a la misma clase

mencionada anteriormente que proporcione el `string` de esa clave, con tal de facilitar su manipulación. Esto se logra mediante el codificador `Base64`, el cual permite la conversión de información almacenada en formatos binarios a datos que podrán ser interpretados como texto.

La llave generada por `getRandomKey()` será la almacenada en la base de datos, por lo que se añadirá como parámetro de este nuevo usuario. Además en el formulario que debe llenar el administrador en la interfaz, deberá ingresar los roles asignados a este usuario (si es que le decide asignar alguno), entonces, cada uno de los roles serán agregados en conjunto con el `ID` de este nuevo usuario a la entidad `userRole`, de esta manera se establecerá que ese usuario tiene ciertos roles asignados.

Finalmente se ejecutará el servicio `SendQREmail`, lo que concluye con el envío de la credencial de ese usuario (se profundizará más sobre este servicio en su propia explicación a continuación).

El único parámetro necesario para ejecutar el servicio `SendQREmail.xml` es la `ID` de algún usuario ya presente en la base de datos. El cual será utilizado para obtener sus datos y enviar el código QR utilizado como credencial de esta persona.

Comienza con la obtención de los datos del usuario en cuestión, para luego ejecutar el servicio `EncryptForQR` de los servicios de autenticación, el cual entregará la llave de este usuario ya encriptada. Este nuevo parámetro será codificado como código QR para hacer la entrega al usuario. Este paso se logra ejecutando la función `generateQRCodeImage` de la clase `QrHelper`. Este método entregará el archivo `.jpeg` que contiene el código generado.

Como último paso del servicio estará el envío del correo electrónico llenando el destinatario con el correo asignado al usuario y adjuntando el código QR generado.

Código 4.8: Generador de código QR para un usuario

```
1   import cl.pca.QrHelper
2
3   def qrHelper = new QrHelper()
4
5   def qrCode =
    ↪ qrHelper.generateQRCodeImage(encryptedKeyValue.encryptedDataStringOut)
6
7   def qrAttach = []
8   qrAttach.add([filename:"QRCode.jpeg", contentType:"image/jpeg",
    ↪ contentDisposition:"attachment", contentBytes:qrCode.toByteArray()])
9
10
```

Para el servicio `RemoveUser` se obtienen todos los roles asignados al usuario en cuestión y se eliminan, concluyendo con la eliminación de los datos de la persona en la entidad de `users`.

Siguiendo con el servicio `UpdateUser`, se actualizan los valores ya existentes en la entidad

`users` y para los roles asignados se eliminan todos los que ya estaban asignados al usuario y se asignan los ingresados en el formulario de creación de usuarios.

Para el servicio de eliminar un rol del servidor (`RemoveRole`) se eliminan todas las relaciones con usuarios o puertas de este rol y finalmente se borra de la entidad `roles`.

Finalmente está el servicio `CreateRoleRel`, el cual, con un arreglo de roles y un usuario, creara los valores que conectan al usuario con cada rol presente en la lista.

DoorsServices.xml

De manera similar a lo mencionado anteriormente en los servicios para usuarios, el crear, editar y eliminar puertas es ejecutado por `CreateDoor`, `UpdateDoor` y `RemoveDoor`. También verifican que se eliminen las llaves foráneas de las entidades relacionales antes de remover una puerta. La escepción es que al crear la puerta el correo electrónico tiene distinto contenido. Para este caso se asigna una llave generada con la función `getRandomKey()` a la puerta nueva para enviarla en el correo. Pero se agregará también la llave pública obtenida de los servicios de autenticación.

Por lo tanto el correo con las credenciales de la puerta contiene la llave generada para la puerta y también la llave pública generada por el servidor para el sistema de encriptación.

Código 4.9: Creación de credenciales para una puerta nueva

```
1     <script><![CDATA[
2
3           import cl.pca.KeyHelper
4           import cl.pca.QrHelper
5
6           def keyHelper = new KeyHelper()
7           def qrHelper = new QrHelper()
8
9           def newKey = keyHelper.getRandomKey()
10          def stringFromKey = keyHelper.getStringFromKeyForDB(newKey)
11
12          def qrAttach = []
13
14    ]]></script>
15
16    ...
17
18    <service-call name="AuthServices.get#GetSetEncryptionKeys" out-map="
    ↪ encryptionKeys"/>
19
20    <set field="publicKeyDB" from="encryptionKeys.publicKey"/>
21
22
```

AuthServices.xml

Toda la manipulación del sistema de encriptación se encuentra en este archivo, el cual tendrá acceso a la llave pública y la llave privada.

Para el servicio `EncryptForQR` se necesita como parámetro una clave (que en este caso será una clave asignada a un usuario almacenada en la base de datos), el cual será utilizado en la función `encrypt()` de la clase `KeyHelper`. Lo importante en este paso es tomar en cuenta que la clave del usuario solo debe ser leída por el mismo servidor, y utilizando las propiedades de la encriptación asimétrica, se encriptará esta clave utilizando la llave pública, ya que el servidor es el único conocedor de la llave privada, por lo tanto **será el único capaz de desencriptar la clave del usuario** generada por `encrypt()`.

Por lo tanto se utilizará la llave pública y la clave del usuario almacenada para encriptar la misma y retornar el string que será utilizado por otro servicio para generar el código QR.

Código 4.10: Encriptación de clave de un usuario

```
1     <service-call name="AuthServices.get#GetSetEncryptionKeys" out-map="
    ↪ encryptionKeys"/>
2
3     <set field="publicKeyDB" from="encryptionKeys.publicKey"/>
4     <set field="privateKeyDB" from="encryptionKeys.privateKey"/>
5
6     <script><![CDATA[
7
8        import cl.pca.KeyHelper
9
10       def keyHelper = new KeyHelper()
11
12       def encryptedData = keyHelper.encrypt(data, publicKeyDB)
13       def encryptedDataString = keyHelper.getStringFromEncodedData(encryptedData)
14
15    ]]></script>
16
17
```

Para la acción contraria está `DecryptFromQr`, servicio el cual desencriptará el string encriptado de la clave de algún usuario. Para llevar a cabo esta acción utiliza la función `decrypt()` de la clase `KeyHelper` y en conjunto con la llave privada podrá desencriptar la clave del usuario.

Para la obtención de las llaves de encriptación se utiliza el servicio `GetSetEncryptionKeys`, el cual verificará si ya existen un par de llaves en la base de datos, en el caso contrario utiliza la función `generatePrivPublKeysDB()` y guardará esas claves generadas en la base de datos. De esta manera si el servidor crea las llaves una vez, siempre utilizará esas mismas llaves hasta que sean eliminadas las anteriores.

Uno de los servicios más importantes de este proyecto es `UserHasAccess`, el cual deter-

minará si el usuario ingresado, primero existe en la base de datos y si es así, si posee los permisos para ingresar por esa puerta.

Primero se desencriptará el código obtenido utilizando `DecryptFromQR`, para luego consultar a la base de datos si el código coincide con alguno de los usuarios existentes. Además será necesario verificar si el código de la puerta ingresado también coincide con alguno de los códigos existentes en la base de datos. Esto también funcionará como identificador de la puerta para ver si los roles usuario-puerta coinciden en alguno. A medida que se van verificando los parámetros, se van rechazando todos los casos que retornen error, tales como que el usuario no existe, la puerta no existe o no tengan los roles. Si el usuario tiene el permiso para acceder por la puerta en cuestión se generará la firma del servidor con la función `sign()`, de esta manera el lector podrá verificar la procedencia de el mensaje y accionar la puerta. En el caso contrario el servidor no genera ninguna firma y el lector no ejecutará acción alguna.

En paralelo a todo esto se documentarán todas las interacciones con este servicio en la entidad *AccessHistory*, donde se almacenará el usuario, la puerta, hora de ingreso y el resultado de la autenticación. Tanto como los fallos como los ingresos correctos serán documentados en esta entidad.

`physical-access-control.rest.xml`

El servidor del sistema tendrá una única manera de comunicarse con los dispositivos de cada puerta. En este archivo se encontrará el servicio al que tendrán acceso (solicitando previamente las credenciales necesarias), el cual será `UserHasAccess` de los servicios de autenticación mencionados anteriormente. Esta API solicitará los datos del administrador para tener acceso a la consulta y la respuesta será el resultado del servicio asignado para realizar la autenticación del usuario.

Capítulo 5

Evaluación

Para continuar con la evaluación del funcionamiento del sistema se realizará un paso a paso de cómo el administrador deberá configurar usuarios, puertas y roles para configurar el sistema de control de acceso correctamente. Todos los datos ingresados a continuación serán creados y utilizados solamente para esta demostración del sistema.

Cabe destacar que se mostrará la utilización de cada uno de los servicios mencionados en este informe anteriormente funcionando en conjunto para enseñar un posible caso de uso del sistema.

5.1. Presentación de resultados

Para contextualizar este caso se ingresarán los datos de 3 trabajadores de una empresa y un invitado, cada uno con distintos cargos que determinarán los permisos y accesos a distintas zonas del establecimiento. Los roles a designar serán (Figura 5.1):

- **Administrador/a**: Usuario con acceso a todas las puertas del recinto y además al servidor mismo del sistema.
- **Emplead@**: Tendrá acceso al área ejecutiva de planteamiento e implementación del producto que ofrece la empresa.
- **Invitad@**: En la situación de que se requiera el ingreso de gente externa a la empresa, se le entregará la credencial de un usuario con acceso a la sala de espera del establecimiento.

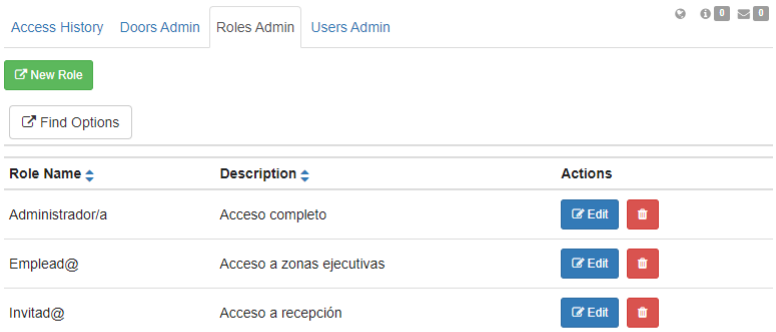


Figura 5.1: Roles ingresados al sistema

Cada puerta del establecimiento tendrá su respectivo nombre de identificación y además una breve descripción definida por el administrador. Las cuales serán (Figura 5.2):

- **Puerta exterior**: Acceso principal al recinto.
- **Oficina administrador**: Zona de trabajo para dueño de la empresa.
- **Sala de reuniones**: Sala para realizar las reuniones con clientes o con la misma emrpesa.
- **Área de desarrollo - Norte**: Acceso norte a la zona designada para los trabajadores/desarrolladores de la empresa.
- **Área de desarrollo - Sur**: Acceso sur a la zona designada para los trabajadores/desarrolladores de la empresa.
- **Sala de servidores**: Lugar de almacenaje físico de los servidores de la empresa.

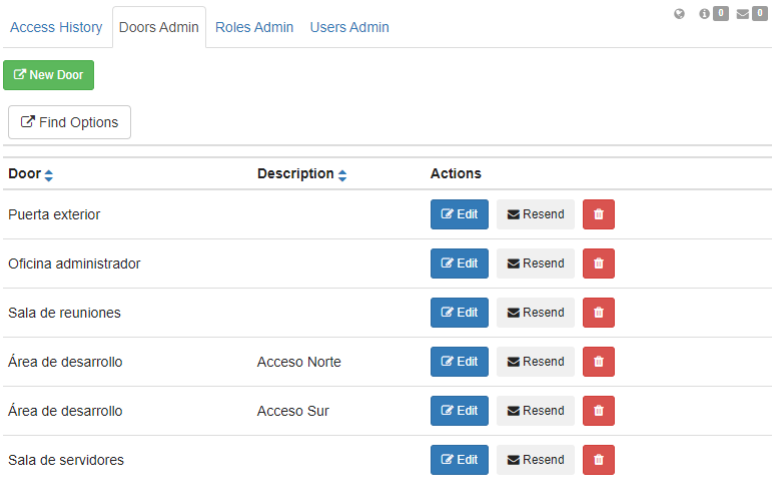


Figura 5.2: Puertas ingresadas al sistema

Finalmente los trabajadores serán ingresados al sistema con sus respectivos roles. Es por esto que se definirán las personas (Figura 5.3) asignadas a cada rol.

- **Administrador/a**:
  - Orlando Real
- **Emplead@**:
  - Sol Ledesma
- **Invitado**:
  - Eva Arias

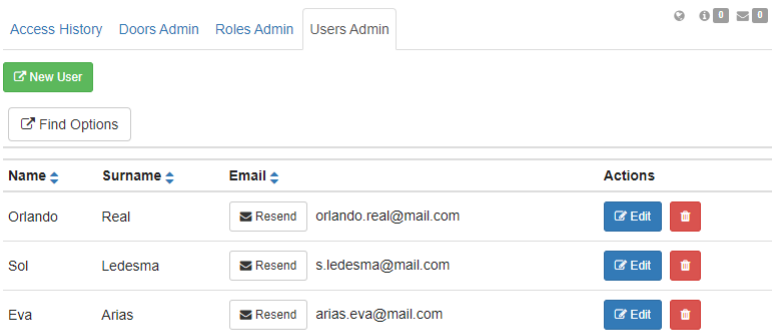


Figura 5.3: Usuarios ingresados al sistema

De la misma manera cada una de las puertas tendrá roles asignados con respecto a la seguridad que le quiera dar las zonas que buscan restringir.

- **Puerta exterior**:
  - Administrador/a
  - Emplead@
  - Invitad@
- **Oficina administrador**:
  - Administrador/a
- **Sala de reuniones**:
  - Administrador/a
  - Emplead@
- **Área de desarrollo - Norte**:
  - Administrador/a

  - Emplead@

- **Área de desarrollo - Sur**:
  - Administrador/a
  - Emplead@
- **Sala de servidores**:
  - Administrador/a

5.1.1. Inicialización del sistema

Todo lo planteado teóricamente podrá ser ingresado en el sistema mediante la interfaz. Usuarios y puertas tendrán asignados los roles mencionados anteriormente. Al ingresar los valores se enviarán los correos electrónicos con las credenciales, tanto para usuarios como puertas.

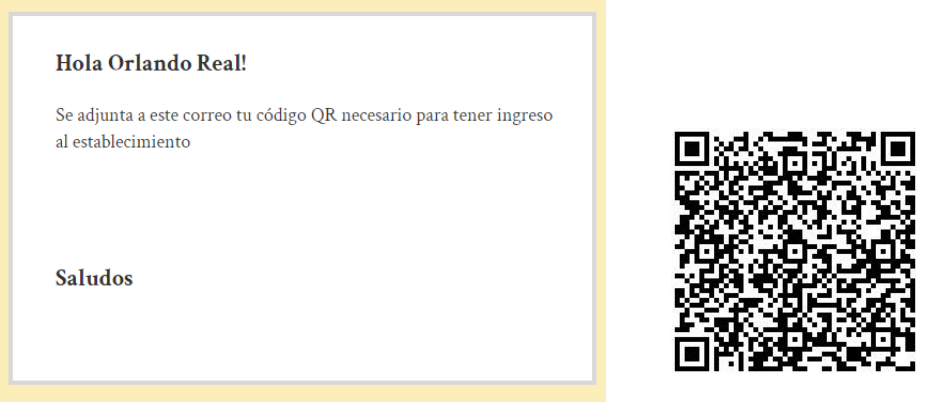
(a) Correo electrónico puerta exterior

(b) Correo electrónico puerta acceso norte del área de desarrollo

Figura 5.4: Ejemplos de correos electrónicos con credenciales de las puertas

Como se puede ver en las Figuras 5.4.a y 5.4.b la llave pública entregada es exactamente la misma dado que el sistema de encriptación utilizará la misma llave para verificar la procedencia de la respuesta del servidor. En cambio, el código identificador de cada puerta será generado al momento de su ingreso al sistema.

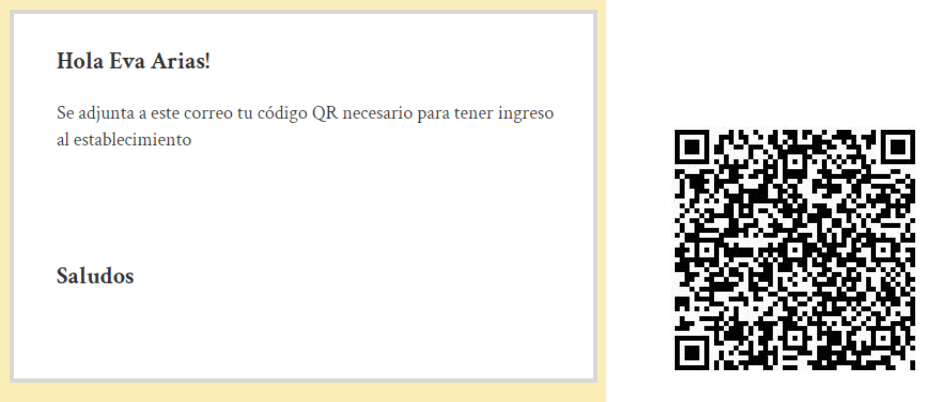
En el caso de los usuarios tenemos los siguientes ejemplos:



(a) Correo electrónico enviado

(b) Código QR asignado para este usuario

Figura 5.5: Correo electrónico para el administrador Orlando Real



(a) Correo electrónico enviado

(b) Código QR asignado para este usuario

Figura 5.6: Correo electrónico para la invitada Eva Arias

Cada usuario recibirá el mismo mensaje en el correo electrónico, a excepción del nombre de la persona involucrada. Además el código QR generado para funcionar como credencial se enviará adjunto al correo electrónico. Incluso el lector puede comprobar, escaneando los códigos QR mostrados para verificar que realmente son códigos imposibles de interpretar y corresponden a información encriptada.

En paralelo a la creación de usuarios y puertas, el servidor inicializó las llaves que se utilizarán para el encriptado de información. Luego de generar estas llaves se almacenan en la base de datos para poder acceder en múltiples servicios ejecutados por el sistema. Para el ejemplo de este informe las llaves creadas son las siguientes:

Código 5.1: Llaves de encriptación asimétrica utilizados para la sección de evaluación

```
1
2   # ----------------------PRIVATE KEY---------------------------
3
4   MIICdQIBADANBgkqhkiG9w0BAQEFAASCAl8wggJbAgEAAoGBAJl4S5mMGIPnSqE
5   RGSPvUM5NPBG8e9ClcpcsFXvyyvCgx8OQwsmwLgesj/xNIvcdnAgNEanZQFiNGF
```

```
rwh8JOJ7RmviGk/Gj0G4r41HMmRyhgOi2eZWHYX+HxDS6ZRsQOgcBqttF8aH1DO
MWhEjmeLPZRcT8PwMJZ/miXw5fESxbTAgMBAAECgYBiAFl2centwTL866g8NERL
mTJ0uQHv2Nb2BiaqAf7p7iHilnxswt6B9AX2PWPIndXXplDTP3JBmcgluldXEwV
yxqyONZ9qCtBsyk15VnBk0HPoSLKyyBRSCF5y14u4pMMHrfexubgOAL0tD7aBHW
krqxBEz1i4C5xCMZapMjEhAQJBAMxhBeQgAe4+cEPdp9tiEt7XTiO7bH4PZImZC
Rh88TW6QJBjjRJ4Ey+jpLj/022iYvU3eWfnu3tY3FgOgj8CU0ECQQDAO4hOWfa5
XE4vDzc5EkHTGybZf0bNmjyblJ7tYBtkeJYZxob4puznYrAzLhrBFhZmJeFOprx
i1cOPeyzI6akTAkBNIjj09deEsO/xzQ/yzjrM+3k0JyI1GV4aSDWUlmL6ogaH9+
pgFA05p4sdfv+umuC6J2uceWrajJjsat1hJmGBAkAX6tBIv1S9N0ja/g+m4Cikd
WpeAUHz7s6/pLwpWJ3P4Fj8mADPCyXKnDLEC2Z0w6TBzNC9km059tLAFVk7HidV
AkAPvgXpIkk1XoOLngnTPciDhW6D+G1qQ9qzP0Db0ab1R6hFxaNETMSyijEaug6
9K7g1LFxLx2HqZJ3PWrxPb4NQ

# ----------------------PUBLIC KEY---------------------------

MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCZeEuZjBiD50qhERkj71DO
TTwRvHvQpXKXLBV78srwoMfDkMLJsC4HrI/8TSL3HZwIDRGp2UBYjRha8IfC
Tie0Zr4hpPxo9BuK+NRzJkcoYDotnmVh2F/h8Q0umUbEDoHAarbRfGh9QzjF
oRI5niz2UXE/D8DCWf5ol8OXxEsW0wIDAQAB
```

5.1.2. Intentos de acceso

Como primera secuencia de acciones tenemos que el administrador ingresará al establecimiento hasta su oficina, pasando por la puerta exterior y finalmente la puerta que le permitirá el ingreso a su lugar de trabajo. En paralelo tendremos que la empleada hará ingreso hacia su lugar de trabajo, pero luego de haber ingresado, intentará hablar con su jefe, por lo que intentará ingresar a su oficina, pero al no tener los permisos se le negará el acceso (Figura 5.7).

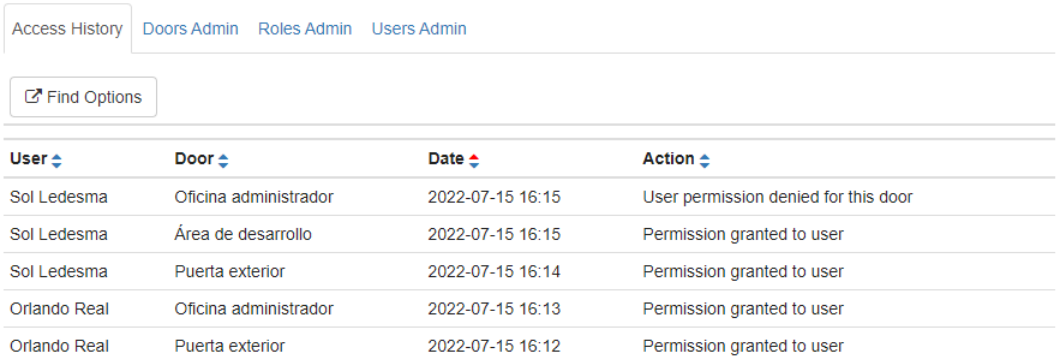
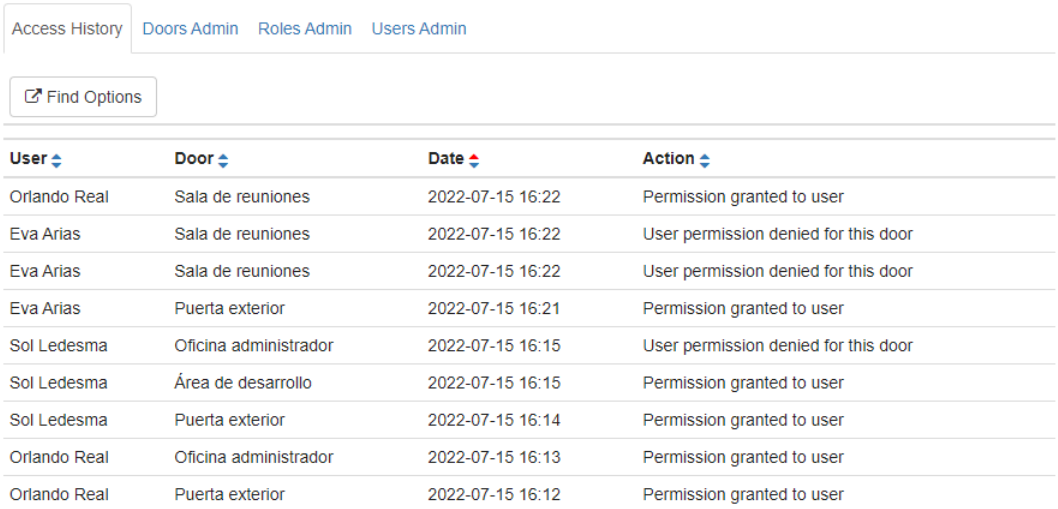


Figura 5.7: Movimientos de la empresa. Administrador y empleada.

Para la segunda secuencia tenemos que hay una usuaria invitada a una reunion con el

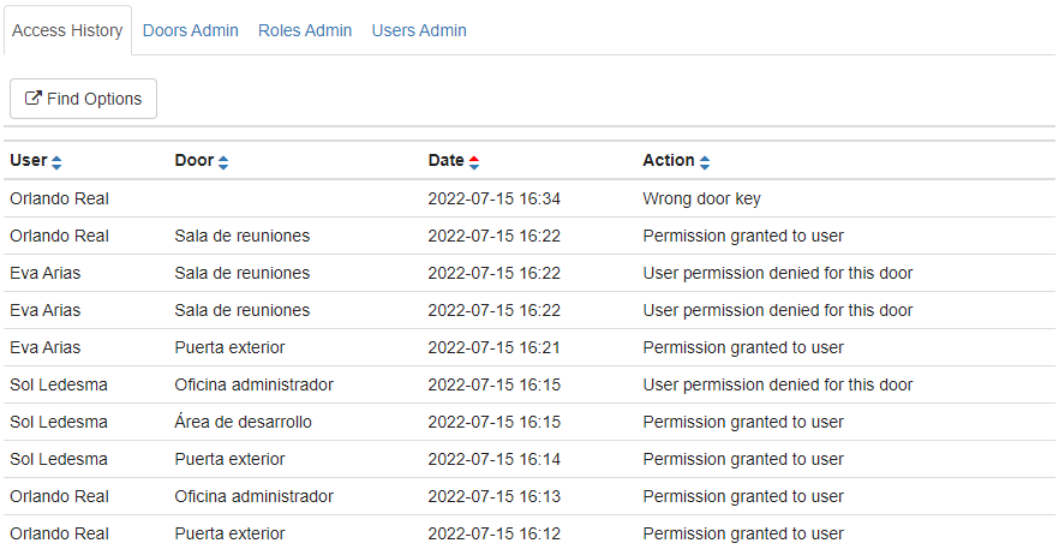
administrador, para esto deberá acceder a la sala de reuniones. El problema es que su rol no tendrá acceso a esta sala, por lo que su credencial solo le permitirá abrir la puerta exterior. Si un usuario piensa que hubo alguna falla de lectura intentará ingresar más de una vez su credencial, lo cual quedará documentado en la base de datos. Finalmente llegará el administrador para la reunión y le permitirá el ingreso a la sala (Figura 5.8).



User	Door	Date	Action
Orlando Real	Sala de reuniones	2022-07-15 16:22	Permission granted to user
Eva Arias	Sala de reuniones	2022-07-15 16:22	User permission denied for this door
Eva Arias	Sala de reuniones	2022-07-15 16:22	User permission denied for this door
Eva Arias	Puerta exterior	2022-07-15 16:21	Permission granted to user
Sol Ledesma	Oficina administrador	2022-07-15 16:15	User permission denied for this door
Sol Ledesma	Área de desarrollo	2022-07-15 16:15	Permission granted to user
Sol Ledesma	Puerta exterior	2022-07-15 16:14	Permission granted to user
Orlando Real	Oficina administrador	2022-07-15 16:13	Permission granted to user
Orlando Real	Puerta exterior	2022-07-15 16:12	Permission granted to user

Figura 5.8: Movimientos de la empresa. Invitada intenta ingresar a sala de reuniones.

La secuencia de acciones mostradas anteriormente serían las más normales en la utilización del sistema de control de acceso. Ahora, puede ocurrir que el administrador falla en ingresar la llave correcta a alguna puerta del recinto, y esta puerta corresponde al área de trabajo del administrador. En este caso tendremos lo siguiente en el historial de accesos.

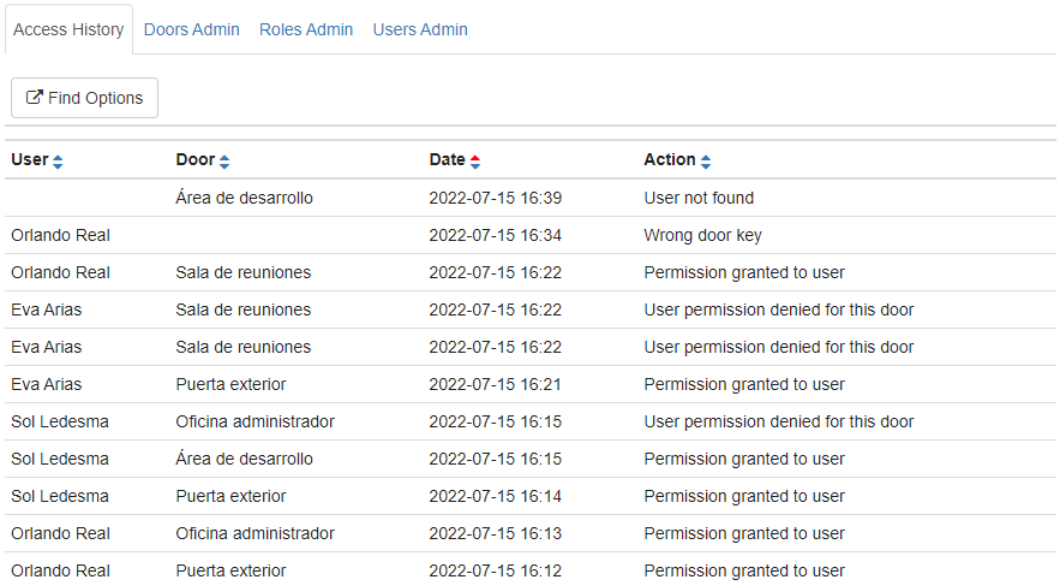


User	Door	Date	Action
Orlando Real		2022-07-15 16:34	Wrong door key
Orlando Real	Sala de reuniones	2022-07-15 16:22	Permission granted to user
Eva Arias	Sala de reuniones	2022-07-15 16:22	User permission denied for this door
Eva Arias	Sala de reuniones	2022-07-15 16:22	User permission denied for this door
Eva Arias	Puerta exterior	2022-07-15 16:21	Permission granted to user
Sol Ledesma	Oficina administrador	2022-07-15 16:15	User permission denied for this door
Sol Ledesma	Área de desarrollo	2022-07-15 16:15	Permission granted to user
Sol Ledesma	Puerta exterior	2022-07-15 16:14	Permission granted to user
Orlando Real	Oficina administrador	2022-07-15 16:13	Permission granted to user
Orlando Real	Puerta exterior	2022-07-15 16:12	Permission granted to user

Figura 5.9: Llave mal ingresada en lector

Al no poseer una llave correcta es imposible para el sistema identificar la puerta a la cual corresponde dicha acción, es por esto que ese campo no se puede llenar. En el caso del usuario si se pudo identificar dado que el administrador sí ingresó su credencial correctamente.

En el caso de que un usuario utilice una credencial expirada o simplemente nunca fue ingresado al sistema se verá de la siguiente forma (Ver última acción del historial en la Figura 5.10). Similar a lo mencionado en el párrafo anterior, es imposible identificar a la persona si no tiene una credencial válida, por lo que el campo del mismo quedará vacío.



User	Door	Date	Action
	Área de desarrollo	2022-07-15 16:39	User not found
Orlando Real		2022-07-15 16:34	Wrong door key
Orlando Real	Sala de reuniones	2022-07-15 16:22	Permission granted to user
Eva Arias	Sala de reuniones	2022-07-15 16:22	User permission denied for this door
Eva Arias	Sala de reuniones	2022-07-15 16:22	User permission denied for this door
Eva Arias	Puerta exterior	2022-07-15 16:21	Permission granted to user
Sol Ledesma	Oficina administrador	2022-07-15 16:15	User permission denied for this door
Sol Ledesma	Área de desarrollo	2022-07-15 16:15	Permission granted to user
Sol Ledesma	Puerta exterior	2022-07-15 16:14	Permission granted to user
Orlando Real	Oficina administrador	2022-07-15 16:13	Permission granted to user
Orlando Real	Puerta exterior	2022-07-15 16:12	Permission granted to user

Figura 5.10: Usuario con credencial invalida intenta entrar

5.1.3. *Feedback* de la autenticación en el lector

El código implementado utiliza la librería OpenCV para capturar la imagen de la cámara y mostrarla en pantalla, por lo que es posible compartir esta imagen con el usuario para que él mismo sepa cuando es realmente leída su credencial.

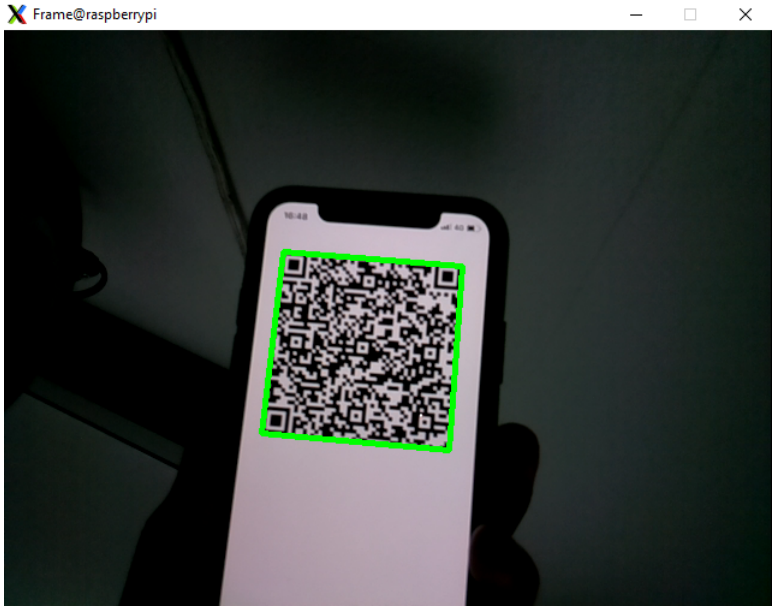


Figura 5.11: Ejemplo del escaneo de un código QR

Al momento de realizar el escaneo de la credencial del usuario, el lector intenta comunicarse inmediatamente con la API de autenticación del servidor. La respuesta de esta consulta está compuesta por 2 valores almacenados en formato *JSON*. El primero sería el valor `result`, el cual entregará la respuesta del servidor con respecto a los valores emitidos desde el lector. Y el segundo valor será la **firma** del servidor, parámetro que solo será enviado en caso de que el acceso haya sido otorgado al usuario con la finalidad de verificar la procedencia de esta respuesta y la *Raspberry* abra la puerta verificando la procedencia de este mensaje.

Código 5.2: Ejemplo de respuesta de la API con acceso confirmado

```
{"result": "Permission granted to user",
"sign": "Y3G3Kjb9XNL4xiPi7jFatd9cua7GoJEChX5H0byh3HSJrlOPEDPVNM7LVWqkh9
 ↪ nknAVqG0pq8KJVOXh4k/X2slEoiReFc6v0Kx7NIlfNirzOIVt/dAzPe5n+/y9DTXt3q9
 ↪ bX5NiQpHZL9tKUCdINSKtA5sij4lUmMve3xKUjYHs="}

```

Código 5.3: Ejemplo de respuesta de la API con acceso denegado

```
{"result": "User not found",
"sign": ""}

```

Dependiendo de esta respuesta de la API el lector deberá tomar la decisión de con cuál

acción continuar. En el mejor de los casos accionará el relé, y por consiguiente se abrirá la puerta.

5.2. Análisis del cumplimiento de los objetivos

Los sistemas de control de acceso modernos buscan facilitar el trabajo del usuario, es decir, siempre se preferirá un sistema que requiera menos exigencia de los usuarios para su funcionamiento. Llevando este aspecto al sistema explicado en este informe, la utilización de un código QR como credencial cae dentro de los métodos de autenticación más simples del mercado. En comparación con los sistemas que requieren un dispositivo *RFID* (identificación por radiofrecuencia) tales como tarjetas o llaveros, no se deberá fabricar un objeto asignado a cada usuario, en cambio se utilizará una herramienta indispensable para las personas en el mundo moderno, el celular. Incluso si el usuario presenta la dificultad de manipular un celular en el momento de la autenticación, se permite llevar una impresión de la credencial, aunque esto quedará en manos de la misma persona.

Desde un inicio la intención de este prototipo fue permitir la compatibilidad del sistema con distintos métodos de autenticación, es por esto que el funcionamiento de todo el sistema se implementó en base a claves y encriptación. Los lectores existentes siempre solicitan un código del usuario para llevar a cabo la autenticación, sin embargo, la diferencia está en cómo el usuario comunica el código al lector. Por ejemplo, tenemos que el lector biométrico escanea la huella digital del usuario, la cual es transformada desde una imagen a un código definido para efectuar su manipulación en el código. También tenemos los objetos que funcionan con *RFID*, los cuales a grandes rasgos son similares a lo implementado con códigos QR, solo que estos requieren un objeto físico que almacene este código. Si bien el sistema desarrollado cumple con la autenticación utilizando códigos QR, las pruebas con otros tipos de lectores no se realizaron. No obstante, el manejo de claves y llaves efectuado por el sistema funciona correctamente, por lo que solo debe cambiarse el momento en el cual estas llaves son ingresadas al sistema y posteriormente facilitadas al usuario. En el caso del lector, tenemos que el dispositivo designado para esta tarea, la *Raspberry*, debe ser conectada al lector que se requiera y solo interpretar el código proporcionado por el usuario, dado que la comunicación con el servidor y la verificación de la respuesta del mismo esta implementada. La elección de la *Raspberry* para el desarrollo de este proyecto se debe, en gran parte, a la simplicidad a la hora de conectar este pequeño dispositivo a un circuito y además la compatibilidad con una gran gama de lectores.

La investigación para el diseño de la seguridad del sistema fue primordial, ya que es el punto que determina qué tan bueno es el sistema, y si este falla, no estaría cumpliendo la labor principal del producto. Cabe destacar que dentro de la informática es imposible elaborar un sistema que sea 100 % impenetrable, dado que gran parte de los métodos de seguridad pueden ser neutralizados utilizando ingeniería inversa para saber cómo están implementados, y además a medida que la programación sigue avanzando, en simultáneo el ataque informático también encuentra nuevas metodologías para lograr su objetivo, obtener información confidencial. Es por esto que para determinar que un sistema es seguro, se implementa una serie obstáculos que dificulten el acceso de terceros a la información manipulada por el sistema, aspecto el cual fue cumplido satisfactoriamente utilizando el método de seguridad explicado

en el informe. Gracias al cual se obtuvo una comunicación lector-servidor que enfocada en su correcto funcionamiento tomando las precauciones necesarias para evitar que un tercero tenga acceso sin tener la autorización de la administración del producto.

Las aplicaciones más utilizadas por los usuarios hoy en día enfocan gran parte del desarrollo de su interfaz en la simplicidad de su uso, tomando en cuenta que el usuario siempre tendrá preferencia por la aplicación más cómoda y fácil de utilizar. Es por esto que la interfaz de administración presenta un diseño sencillo y que puede ser fácilmente entendido sin explicación previa. Moqui permitió con gran éxito presentar una interfaz clara y limpia para prestar el servicio al administrador, quien hará uso recurrente de esta aplicación.

Y si eso no es suficiente, el *framework* utilizado permitió satisfactoriamente un excelente almacenamiento de los datos manejados por el sistema. Entidades funcionando en conjunto permitieron una buena administración de los datos. Además gracias a su complicidad con el lenguaje de programación *Groovy*, permitió manejar y modificar valores para evitar problemas de compatibilidad con el almacenamiento de la *data*.

El diseño de la arquitectura del sistema en su totalidad fue pensada con el fin de cumplir su objetivo principal en una oficina, lugar en el cual hay mucho flujo de personas. Es por esto que se decidió incorporar una *Raspberry* como dispositivo a cargo de cada acceso al recinto, de esta manera el cliente (quien solicitará el servicio de este sistema), tiene la posibilidad de adaptar la cantidad de lectores a su propio establecimiento, de esta manera se tendrá un servicio muy cómodo para quien lo utilice.

Enfocando el desarrollo a la simplicidad de su utilización, se logró obtener un producto simple y limpio en su interfaz, y que además ofrece un sistema de seguridad funcional que evitará accesos de terceros en el lugar de instalación del sistema.

Capítulo 6

Conclusión

6.1. Retrospectiva

La elaboración de este proyecto estuvo motivada en un inicio por la utilización de dispositivos *Raspberry* como controladores de cada puerta, idea proporcionada por el profesor guía. En conjunto el descubrimiento del *framework* Moqui fue una sorpresa para el memorista, ya que en un inicio se vio bastante complicado dada su lenguaje basado en *Groovy* y *XML* (nunca antes utilizado por el memorista), pero al avanzar con el desarrollo del sistema, Moqui presentó muchas utilidades que facilitaron la implementación, tanto de la interfaz como la base de datos desplegada por el mismo *framework*.

La investigación previa al desarrollo fue de lo más importante, ya que hubo que seleccionar la mejor opción considerada en virtud de la información obtenida. La decisión final planteada en este informe fue enfocada en qué tan amigable será la plataforma con el usuario y además su simple instalación por parte del administrador. Si bien la *Raspberry* dio muchos puntos a favor a la arquitectura del sistema escogida, tanto como su portabilidad, compatibilidad con múltiples lectores y además, su simple conexión con dispositivos electrónicos para controlar circuitos (en este caso orientados a seguros electrónicos para puertas), la elección del escaneo de códigos QR utilizando una cámara presentó diversas limitaciones tomando en cuenta el tiempo de procesamiento para escanear los códigos. En el caso de haber elegido directamente un dispositivo que funcione con luz láser (similar a los que escanean códigos de barras), se habría optimizado la ejecución y sobrecarga de la *Raspberry*.

Idear la compatibilidad entre el servidor de Moqui y la *Raspberry* fue de lo más interesante, ya que utilizando toda la información investigada, se logró desarrollar un producto bueno con tecnologías desconocidas por el memorista. En base a la teoría de la investigación se diseñó el sistema mostrado en el informe que pudo ser luego elaborado físicamente con los dispositivos facilitados por Moit y por consiguiente probar la ejecución planteada, en un principio, como una mera hipótesis del memorista. El buen funcionamiento de lo diseñado fue bastante satisfactorio luego de tanta investigación.

Importante mencionar que a medida que se fueron presentando las dificultades en la implementación en Moqui, el equipo de Moit siempre estuvo dispuesto a la ayuda. Cabe destacar al profesor guía de esta memoria, quien respondía todo tipo de dudas hechas por el grupo de memoristas trabajando con él, tanto de la implementación, como consultas sobre todo el

proceso de la elaboración de la memoria.

6.2. Trabajo futuro

Como fue señalado en los objetivos de este informe, el sistema está orientado a funcionar con dispositivos que utilicen múltiples métodos de autenticación. Es por esto que el sistema de autenticación implementado puede ser reutilizado para funcionar con lectores biométricos, de *RFID* e incluso códigos *PIN*. Como siguiente paso de este proyecto sería entregar un catálogo de dispositivos compatibles para que el administrador pueda seleccionar según su preferencia sin cambiar el programa, e incluso permitir que combine distintos lectores en el sistema.

Una de las ideas más interesantes para desarrollar y complementar el sistema, es sacandole provecho a todas las herramientas ofrecidas por la *Raspberry*, en especifico al *Bluetooth*. Tomando en cuenta que el sistema será tomado como preferencia de los clientes mientras más simple sea su uso sin perder la calidad en su seguridad, es posible implementar una aplicación que almacene la credencial de cada usuario y la comunicación con el lector será mediante *Bluetooth*. Por ejemplo, si un usuario quiere acceder a un la puerta de la sala de reuniones, simplemente debe accionar el botón en la aplicación para que esta autentique al usuario y le permita el acceso (siguiendo los pasos de autentiación y autorización de la persona). Siguiendo la misma filosofía de que el sistema debe tener una interfaz simple, estos botones deberán ser de fácil acceso y la aplicación debe tener una comunicación instantánea con el celular del usuario mediante *Bluetooth*.

Otro aspecto a tomar en cuenta para preferir este sistema en comparación a la competencia será lo económico que son sus componentes. Si bien el modelo de *Raspberry* utilizado para el desarrollo de esta memoria es la *3 Model B+*, la misma empresa ofrece modelos más económicos que son compatibles con lo implementado en el sistema presentado. Por lo que como siguiente paso también se tiene las pruebas con otros modelos más económicos de *Raspberry* para seguir optimizando recursos del producto final.

Bibliografía

[1] ElProCus, "Access control systems.", https://www.elprocus.com/understanding-about-types-of-access-control-systems/.

[2] CIEGroup, "Access control systems.", https://cie-group.com/how-to-av/videos-and-blogs/access-control-systems.

[3] Moqui, "Sitio web.", https://www.moqui.org/.

[4] Moit, "Sitio web.", https://moit.cl/.

[5] Identicard, "Sitio web.", https://www.identicard.cl/.

[6] BeePro, "Sitio web.", https://www.beepro.cl/.

[7] Systems, S., "Sitio web.", https://saltosystems.com/es-cl/.

[8] rr developer, "Luks on raspberry pi.", https://rr-developer.github.io/LUKS-on-Raspberry-Pi/.

[9] Gullaksen, K., "Qrgen: a simple qrcode generation api for java.", https://github.com/kenglxn/QRGen.

[10] Moqui, "Security.", https://www.moqui.org/docs/framework/Security.

[11] RedIRIS, "Autenticación de usuarios.", https://www.rediris.es/cert/doc/unixsec/node14.html.

[12] Isonas, "The evolution of access control.", https://www.isonas.com/news-education/the-evolution-of-access-control/.

[13] TechTarget, "Access control systems.", https://www.techtarget.com/searchsecurity/definition/access-control.

[14] Watson, A., "How to securely store a password in java.", https://dev.to/awwsmm/how-to-encrypt-a-password-in-java-42dh.

[15] Schooneveld, J. V., "Python and rest apis: Interacting with web services.", https://realpython.com/api-integration-in-python/.

[16] ZYMR, "Managing api security by using tokenization.", https://www.zymr.com/managing-api-security-by-using-tokenization/.

[17] PreVeil, "Public – private key pairs how they work.", https://www.preveil.com/blog/public-and-private-key/.

[18] Cloudflare, "¿cómo funciona la encriptación de clave pública? | criptografía de clave pública y ssl.", https://www.cloudflare.com/es-es/learning/ssl/how-does-public-key-encryption-work/.

[19] Ray, D., "Rsa encryption and decryption in java.", https://www.devglan.com/java8/rs

a-encryption-decryption-java.

[20] Simplilearn, "What is data encryption: Types, algorithms, techniques and methods.", https://www.simplilearn.com/data-encryption-methods-article.

[21] Dommerholt, N., "Rsa signing and encryption in java.", https://niels.nu/blog/2016/java-rsa.

[22] Unlayer, "Html email templates.", https://unlayer.com/templates.

[23] Educative, "What is the rsa algorithm?.", https://www.educative.io/answers/what-is-the-rsa-algorithm.

[24] OpenPath, "Access control systems.", https://www.openpath.com/the-ultimate-guide-to-access-control-systems.

Anexo

A. Código fuente

A.1. Groovy

Código A.1: Administrador de claves del sistema

```
1
2   class KeyHelper {
3
4   KeyGenerator keyGenerator;
5   KeyPairGenerator keyGen;
6   SecureRandom secureRandom;
7   int keyBitSize;
8
9   String privateKeyString;
10  String publicKeyString;
11
12  public KeyHelper() throws NoSuchAlgorithmException {
13     SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
14
15     keyGenerator = KeyGenerator.getInstance("AES");
16     keyBitSize = 256;
17     keyGenerator.init(keyBitSize, random);
18
19     keyGen = KeyPairGenerator.getInstance("RSA");
20     keyGen.initialize(1024, random);
21  }
22
23  public String getStringFromEncodedData(byte[] data) {
24     return Base64.getEncoder().encodeToString(data);
25  }
26
27  public byte[] getEncodedDataFromString(String data) {
28     return Base64.getDecoder().decode(data);
29  }
30
31  public static PublicKey getPublicKey(String base64PublicKey){
32     PublicKey publicKey = null;
33     try{
34        X509EncodedKeySpec keySpec = new
 ↪ X509EncodedKeySpec(Base64.getDecoder().decode(base64PublicKey.getBytes()));
```

```
35          KeyFactory keyFactory = KeyFactory.getInstance("RSA");
36          publicKey = keyFactory.generatePublic(keySpec);
37          return publicKey;
38       } catch (NoSuchAlgorithmException e) {
39          e.printStackTrace();
40       } catch (InvalidKeySpecException e) {
41          e.printStackTrace();
42       }
43       return publicKey;
44    }
45
46    public static PrivateKey getPrivateKey(String base64PrivateKey){
47       PrivateKey privateKey = null;
48       PKCS8EncodedKeySpec keySpec = new
     ↪ PKCS8EncodedKeySpec(Base64.getDecoder().decode(base64PrivateKey.getBytes()));
49       KeyFactory keyFactory = null;
50       try {
51          keyFactory = KeyFactory.getInstance("RSA");
52       } catch (NoSuchAlgorithmException e) {
53          e.printStackTrace();
54       }
55       try {
56          privateKey = keyFactory.generatePrivate(keySpec);
57       } catch (InvalidKeySpecException e) {
58          e.printStackTrace();
59       }
60       return privateKey;
61    }
62
63    public static String sign(String plainText, String privateKey) throws Exception {
64       Signature privateSignature = Signature.getInstance("SHA256withRSA");
65       privateSignature.initSign(getPrivateKey(privateKey));
66       privateSignature.update(plainText.getBytes("UTF-8"));
67
68       byte[] signature = privateSignature.sign();
69
70       return Base64.getEncoder().encodeToString(signature);
71    }
72
73    public static boolean verify(String plainText, String signature, String publicKey) throws
     ↪ Exception {
74       Signature publicSignature = Signature.getInstance("SHA256withRSA");
75       publicSignature.initVerify(getPublicKey(getPublicKey(publicKey)));
76       publicSignature.update(plainText.getBytes("UTF-8"));
77
78       byte[] signatureBytes = Base64.getDecoder().decode(signature);
79
80       return publicSignature.verify(signatureBytes);
81    }
82
83    public static byte[] encrypt(String data, String publicKey) throws BadPaddingException,
     ↪ IllegalBlockSizeException, InvalidKeyException, NoSuchPaddingException,
```

```
    ↪ NoSuchAlgorithmException {
84      Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
85      cipher.init(Cipher.ENCRYPT_MODE, getPublicKey(publicKey));
86      return cipher.doFinal(data.getBytes());
87  }
88
89  public static byte[] encryptPrivate(String data, String privateKey) throws
    ↪ BadPaddingException, IllegalBlockSizeException, InvalidKeyException,
    ↪ NoSuchPaddingException, NoSuchAlgorithmException {
90      Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
91      cipher.init(Cipher.ENCRYPT_MODE, getPrivateKey(privateKey));
92      return cipher.doFinal(data.getBytes());
93  }
94
95  public static String decryptPublic(byte[] data, PublicKey publicKey) throws
    ↪ NoSuchPaddingException, NoSuchAlgorithmException, InvalidKeyException,
    ↪ BadPaddingException, IllegalBlockSizeException {
96      Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
97      cipher.init(Cipher.DECRYPT_MODE, publicKey);
98      return new String(cipher.doFinal(data));
99  }
100
101 public static String decryptPublic(String data, String base64PublicKey) throws
    ↪ IllegalBlockSizeException, InvalidKeyException, BadPaddingException,
    ↪ NoSuchAlgorithmException, NoSuchPaddingException {
102     return decryptPublic(Base64.getDecoder().decode(data.getBytes()),
    ↪ getPublicKey(base64PublicKey));
103 }
104
105 public static String decrypt(byte[] data, PrivateKey privateKey) throws
    ↪ NoSuchPaddingException, NoSuchAlgorithmException, InvalidKeyException,
    ↪ BadPaddingException, IllegalBlockSizeException {
106     System.out.println("se inicia el decrypt en groovy");
107     Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
108     cipher.init(Cipher.DECRYPT_MODE, privateKey);
109     System.out.println(new String(cipher.doFinal(data)));
110     return new String(cipher.doFinal(data));
111 }
112
113 public static String decrypt(String data, String base64PrivateKey) throws
    ↪ IllegalBlockSizeException, InvalidKeyException, BadPaddingException,
    ↪ NoSuchAlgorithmException, NoSuchPaddingException {
114
115     return decrypt(Base64.getDecoder().decode(data.getBytes()),
    ↪ getPrivateKey(base64PrivateKey));
116 }
117
118 public void generatePrivPublKeysDB() throws NoSuchAlgorithmException {
119     KeyPair pair = keyGen.generateKeyPair();
120     PrivateKey privateKey = pair.getPrivate();
121     PublicKey publicKey = pair.getPublic();
122
```

```
123         privateKeyString = Base64.getEncoder().encodeToString(privateKey.getEncoded());
124         publicKeyString = Base64.getEncoder().encodeToString(publicKey.getEncoded());
125     }
126 
127     public SecretKey getRandomKey() throws NoSuchAlgorithmException {
128         return keyGenerator.generateKey();
129     }
130 
131     public String getStringFromKeyForDB(SecretKey key) throws
        ↪ NoSuchAlgorithmException {
132         return Base64.getEncoder().encodeToString(key.getEncoded());
133     }
134 
135 }
136 
```

Código A.2: Generador de códigos QR

```
1 
2           class QrHelper {
3 
4     public static ByteArrayOutputStream generateQRCodeImage(String barcodeText) throws
        ↪ Exception {
5         return QRCode.from(barcodeText).to(ImageType.JPG).withSize(250, 250).stream();
6     }
7 }
8 
```

A.2. Servicios de Moqui

A.2.1. Servicios de autenticación

Código A.3: Encriptar información para código QR

```
1 
2           <service verb="get" noun="EncryptForQR" authenticate="anonymous-all" allow-
    ↪ remote="false" transaction-timeout="300000">
3         <in-parameters>
4             <parameter name="data"/>
5         </in-parameters>
6         <out-parameters>
7             <parameter name="encryptedDataStringOut"/>
8         </out-parameters>
9         <actions>
10 
11            <service-call name="AuthServices.get#GetSetEncryptionKeys" out-map="
    ↪ encryptionKeys"/>
12 
13            <set field="publicKeyDB" from="encryptionKeys.publicKey"/>
14            <set field="privateKeyDB" from="encryptionKeys.privateKey"/>
15 
```

```
        <script><![CDATA[

            import cl.pca.KeyHelper

            def keyHelper = new KeyHelper()

            def encryptedData = keyHelper.encrypt(data, publicKeyDB)
            def encryptedDataString = keyHelper.getStringFromEncodedData(
 ↪ encryptedData)

        ]]></script>

        <set field="encryptedDataStringOut" from="encryptedDataString"/>

    </actions>
</service>

```

Código A.4: Desencriptar información desde código QR

```

        <service verb="get" noun="DecryptFromQR" authenticate="anonymous-all">
    <in-parameters>
        <parameter name="data"/>
    </in-parameters>
    <out-parameters>
        <parameter name="dataOut"/>
    </out-parameters>
    <actions>

        <service-call name="AuthServices.get#GetSetEncryptionKeys" out-map="
 ↪ encryptionKeys"/>

        <set field="publicKeyDB" from="encryptionKeys.publicKey"/>
        <set field="privateKeyDB" from="encryptionKeys.privateKey"/>

        <script><![CDATA[

            import cl.pca.KeyHelper

            def keyHelper = new KeyHelper()
            def decryptedData = null
            try {
                decryptedData = keyHelper.decrypt(data, privateKeyDB)
            } catch (Exception e) {
                print 'Error al desencriptar\n'
            }
        ]]></script>

        <set field="dataOut" from="decryptedData"/>

```

```
    </actions>
  </service>

```

Código A.5: Obtener/Definir las llaves de encriptación para el sistema

```

    <service verb="get" noun="GetSetEncryptionKeys" authenticate="anonymous-all"
↪ allow-remote="false" transaction-timeout="300000">
    <out-parameters>
      <parameter name="privateKey"/>
      <parameter name="publicKey"/>
    </out-parameters>
    <actions>

      <entity-find entity-name="access.control.encryptionKeys" list="eKeysList"/>

      <if condition="eKeysList.size() > 0">
        <set field="keyPair" from="eKeysList[0]"/>
        <set field="privateKey" from="keyPair.privateKey"/>
        <set field="publicKey" from="keyPair.publicKey"/>
        <return/>
      </if>

      <script><![CDATA[

        import cl.pca.KeyHelper

        def keyHelper = new KeyHelper()

        keyHelper.generatePrivPublKeysDB()

        def newPrivateKey = keyHelper.privateKeyString
        def newPublicKey = keyHelper.publicKeyString

      ]]></script>
      <service-call name="create#access.control.encryptionKeys" in-map="[publicKey:
↪ newPublicKey, privateKey:newPrivateKey]"/>
      <set field="privateKey" from="newPrivateKey"/>
      <set field="publicKey" from="newPublicKey"/>
    </actions>
  </service>

```

Código A.6: Autenticación del usuario

```

    <service verb="get" noun="UserHasAccess" authenticate="anonymous-all" allow-
↪ remote="true" transaction-timeout="300000">
    <in-parameters>
      <parameter name="codeId" required="true"/>
```

```
        <parameter name="doorKey" required="true"/>
    </in-parameters>
    <out-parameters>
        <parameter name="result"/>
        <parameter name="sign"/>
    </out-parameters>
    <actions>

        <set field="openKey" value=""/>

        <service-call name="AuthServices.get#DecryptFromQR" in-map="[data:codeId]"
↪ out-map="codeIdDecrypted"/>
        <set field="decryptedUserKey" from="codeIdDecrypted.dataOut"/>

        <if condition="codeIdDecrypted == null">
            <set field="action" value="User key don't match with security system"/>
            <service-call name="create#access.control.accessHistory" in-map="[date:date,
↪ action:action]" out-map="newAccess"/>
            <set field="result" from="action"/>
            <return/>
        </if>

        <set field="date" from="ec.user.nowTimestamp"/>

        <entity-find-one entity-name="access.control.users" value-field="userValues">
            <field-map field-name="codeId" from="decryptedUserKey"/>
        </entity-find-one>

        <entity-find-one entity-name="access.control.doors" value-field="doorValues">
            <field-map field-name="doorKey" from="doorKey"/>
        </entity-find-one>

        <if condition="userValues == null">
            <if condition="doorValues == null">
                <set field="action" value="User not found and wrong door key"/>
                <service-call name="create#access.control.accessHistory" in-map="[date:date,
↪  action:action]" out-map="newAccess"/>
                <set field="result" from="action"/>
                <return/>
            </if>
            <set field="action" value="User not found"/>
            <service-call name="create#access.control.accessHistory" in-map="[doorId:
↪ doorValues.doorId, date:date, action:action]" out-map="newAccess"/>
            <set field="result" from="action"/>
            <return/>
        </if>

        <if condition="doorValues == null">
            <set field="action" value="Wrong door key"/>
            <service-call name="create#access.control.accessHistory" in-map="[userId:
↪ userValues.userId, date:date, action:action]" out-map="newAccess"/>
            <set field="result" from="action"/>
```

```
            <return/>
        </if>

        <entity-find-one entity-name="access.control.userDoorRole" value-field="fullRel">
            <field-map field-name="userId" from="userValues.userId"/>
            <field-map field-name="doorId" from="doorValues.doorId"/>
        </entity-find-one>

        <if condition="fullRel == null">
            <set field="action" value="User permission denied for this door"/>
            <service-call name="create#access.control.accessHistory" in-map="[userId:
    ↪ userValues.userId, doorId:doorValues.doorId, date:date, action:action]" out-map="
    ↪ newAccess"/>
            <set field="result" from="action"/>
            <return/>
        </if>

        <set field="action" value="Permission granted to user"/>
        <service-call name="create#access.control.accessHistory" in-map="[userId:
    ↪ userValues.userId, doorId:doorValues.doorId, date:date, action:action]" out-map="
    ↪ newAccess"/>

        <service-call name="AuthServices.get#GetSetEncryptionKeys" out-map="
    ↪ encryptionKeys"/>

        <script><![CDATA[

            import cl.pca.KeyHelper

            def keyHelper = new KeyHelper()

            def signValue = keyHelper.sign(doorKey, encryptionKeys.privateKey)

        ]]></script>

        <set field="result" from="action"/>
        <set field="sign" from="signValue"/>
    </actions>
</service>

```

A.2.2. Servicios para las puertas

Código A.7: Creación de una puerta con envío de correo electrónico

```

            <service verb="create" noun="CreateDoor" authenticate="anonymous-all" allow-
    ↪ remote="true" transaction-timeout="300000">
    <description></description>
    <in-parameters>
        <auto-parameters entity-name="access.control.doors" include="nonpk"/>
```

```
        <parameter name="name" required="true"/>
        <parameter name="description" required="false"/>
        <parameter name="roles" type="List"/>
        <parameter name="email" required="true"/>
      </in-parameters>
      <out-parameters>
        <parameter name="doorId"/>
      </out-parameters>
      <actions>

        <script><![CDATA[

            import cl.pca.KeyHelper
            import cl.pca.QrHelper

            def keyHelper = new KeyHelper()
            def qrHelper = new QrHelper()

            def newKey = keyHelper.getRandomKey()
            def stringFromKey = keyHelper.getStringFromKeyForDB(newKey)

            def qrAttach = []

        ]]></script>
        <set field="doorKey" from="stringFromKey"/>

        <service-call name="create#access.control.doors" in-map="context+[doorKey:
    ↪ doorKey]" out-map="newDoor"/>

        <if condition="roles">
          <iterate list="roles" entry="role">
            <service-call name="create#access.control.doorRole" in-map="context+[
    ↪ doorId:newDoor.doorId ,roleId:role]"/>
          </iterate>
        </if>

        <service-call name="AuthServices.get#GetSetEncryptionKeys" out-map="
    ↪ encryptionKeys"/>

        <set field="publicKeyDB" from="encryptionKeys.publicKey"/>

        <set field="emailTemplateId" value="DoorKeyTemplate"/>
        <set field="fromName" value=""/>
        <set field="fromAddress" from="org.moqui.util.SystemBinding.getPropOrEnv('moit
    ↪ -erp.notification.emailFromAddress')"/>
        <set field="toAddresses" from="email"/>
        <set field="title" value="Test Correo Door Key"/>
        <set field="bodyParameters" from="[replyToAddresses:fromAddress, title:title,
    ↪ fromName:fromName, fromAddress:fromAddress, newDoorKey: doorKey,
    ↪ newDoorName: name, newDoorDesc: description, publicKey: publicKeyDB]"/>
        <set field="attachmentQrEmail" from="qrAttach"/>
```

```
        <service-call name="org.moqui.impl.EmailServices.send#Email" out-map="sendOut
    ↪ ">
            <field-map field-name="emailTypeEnumId" value="EMT_NOTIFICATION"/>
            <field-map field-name="emailTemplateId" from="emailTemplateId"/>
            <field-map field-name="toAddresses" from="toAddresses"/>
            <field-map field-name="replyToAddresses" from="fromAddress"/>
            <field-map field-name="fromName" from="fromName"/>
            <field-map field-name="fromAddress" from="fromAddress"/>
            <field-map field-name="attachments" from="attachmentQrEmail"/>
            <field-map field-name="bodyParameters" from="bodyParameters"/>
        </service-call>
        <if condition="sendOut.messageId == null">
            <set field="result" from="false" />
            <return error="true" message="Error sending mail, rolling back"/>
        </if>
    </actions>
</service>

```

Código A.8: Envío de credenciales de la puerta

```

        <service verb="get" noun="SendDoorKeysToEmail" authenticate="anonymous-all"
    ↪ allow-remote="true" transaction-timeout="300000">
    <description></description>
    <in-parameters>
        <parameter name="doorId" required="true"/>
        <parameter name="email" required="true"/>
    </in-parameters>
    <out-parameters>
        <parameter name="doorId"/>
    </out-parameters>
    <actions>

        <entity-find-one entity-name="access.control.doors" value-field="doorValues">
            <field-map field-name="doorId" from="doorId"/>
        </entity-find-one>

        <service-call name="AuthServices.get#GetSetEncryptionKeys" out-map="
    ↪ encryptionKeys"/>

        <set field="publicKeyDB" from="encryptionKeys.publicKey"/>

        <set field="emailTemplateId" value="DoorKeyTemplate"/>
        <set field="fromName" value=""/>
        <set field="fromAddress" from="org.moqui.util.SystemBinding.getPropOrEnv('moit
    ↪ -erp.notification.emailFromAddress')"/>
        <set field="toAddresses" from="email"/>
        <set field="title" value="Test Correo Door Key"/>
        <set field="bodyParameters" from="[replyToAddresses:fromAddress, title:title,
    ↪ fromName:fromName, fromAddress:fromAddress, newDoorKey: doorValues.doorKey,
```

```
    ↪ newDoorName: name, publicKey: publicKeyDB]"/>
27          <set field="attachmentQrEmail" from="qrAttach"/>
28          <service-call name="org.moqui.impl.EmailServices.send#Email" out-map="sendOut
    ↪ ">
29              <field-map field-name="emailTypeEnumId" value="EMT_NOTIFICATION"/>
30              <field-map field-name="emailTemplateId" from="emailTemplateId"/>
31              <field-map field-name="toAddresses" from="toAddresses"/>
32              <field-map field-name="replyToAddresses" from="fromAddress"/>
33              <field-map field-name="fromName" from="fromName"/>
34              <field-map field-name="fromAddress" from="fromAddress"/>
35              <field-map field-name="attachments" from="attachmentQrEmail"/>
36              <field-map field-name="bodyParameters" from="bodyParameters"/>
37          </service-call>
38          <if condition="sendOut.messageId == null">
39              <set field="result" from="false" />
40              <return error="true" message="Error sending mail, rolling back"/>
41          </if>
42      </actions>
43  </service>
44
```

A.2.3. Servicios para los usuarios

Código A.9: Creación de un usuario con asignación de claves

```
1
2           <service verb="create" noun="CreateUser" authenticate="anonymous-all" allow-
    ↪ remote="true" transaction-timeout="300000">
3       <description></description>
4       <in-parameters>
5           <auto-parameters entity-name="access.control.users" include="nonpk"/>
6           <parameter name="name" required="true"/>
7           <parameter name="surname" required="true"/>
8           <parameter name="email" required="true"/>
9           <parameter name="roles" type="List"/>
10      </in-parameters>
11      <out-parameters>
12          <parameter name="userId"/>
13      </out-parameters>
14      <actions>
15
16          <script><![CDATA[
17
18              import cl.pca.KeyHelper
19
20              def keyHelper = new KeyHelper()
21
22              def newKey = keyHelper.getRandomKey()
23              def stringFromKey = keyHelper.getStringFromKeyForDB(newKey)
24
25          ]]></script>
```

```
26
27          <set field="codeId" from="stringFromKey"/>
28          <entity-find-one entity-name="access.control.users" value-field="userValues">
29             <field-map field-name="codeId" from="codeId"/>
30          </entity-find-one>
31
32          <if condition="userValues != null">
33             <return error="true" message="User already exists"/>
34          </if>
35
36          <service-call name="create#access.control.users" in-map="context" out-map="
    ↪ newUser"/>
37
38          <if condition="roles">
39             <iterate list="roles" entry="role">
40                <service-call name="create#access.control.userRole" in-map="context+[userId
    ↪ :newUser.userId ,roleId:role]"/>
41             </iterate>
42          </if>
43
44          <service-call name="UsersServices.get#SendQREmail" in-map="[userId:newUser.
    ↪ userId]"/>
45
46       </actions>
47    </service>
48
```

Código A.10: Envío de código QR al correo electrónico del usuario solicitado

```
1
2          <service verb="get" noun="SendQREmail" authenticate="anonymous-all" allow-
    ↪ remote="true" transaction-timeout="300000">
3       <description></description>
4       <in-parameters>
5          <parameter name="userId" required="true"/>
6       </in-parameters>
7       <out-parameters>
8          <parameter name="userId"/>
9       </out-parameters>
10      <actions>
11
12         <entity-find-one entity-name="access.control.users" value-field="userValues">
13            <field-map field-name="userId" from="userId"/>
14         </entity-find-one>
15
16         <if condition="userValues == null">
17            <return error="true" message="User not found"/>
18         </if>
19
20         <service-call name="AuthServices.get#EncryptForQR" in-map="[data:userValues.
    ↪ codeId]" out-map="encryptedKeyValue"/>
```

```
        <script><![CDATA[

            import cl.pca.QrHelper

            def qrHelper = new QrHelper()

            def qrCode = qrHelper.generateQRCodeImage(encryptedKeyValue.
    ↪ encryptedDataStringOut)

            def qrAttach = []
            qrAttach.add([filename:"QRCode.jpeg", contentType:"image/jpeg",
    ↪ contentDisposition:"attachment", contentBytes:qrCode.toByteArray()])

        ]]></script>
        <set field="emailTemplateId" value="QrCodeTemplate"/>
        <set field="fromName" value=""/>
        <set field="fromAddress" from="org.moqui.util.SystemBinding.getPropOrEnv('moit
    ↪ -erp.notification.emailFromAddress')"/>
        <set field="toAddresses" from="userValues.email"/>
        <set field="title" value="Test Correo"/>
        <set field="bodyParameters" from="[replyToAddresses:fromAddress, title:title,
    ↪ fromName:fromName, fromAddress:fromAddress, userName:userValues.name,
    ↪ userSurname:userValues.surname]"/>
        <set field="attachmentQrEmail" from="qrAttach"/>
        <service-call name="org.moqui.impl.EmailServices.send#Email" out-map="sendOut
    ↪ ">
            <field-map field-name="emailTypeEnumId" value="EMT_NOTIFICATION"/>
            <field-map field-name="emailTemplateId" from="emailTemplateId"/>
            <field-map field-name="toAddresses" from="toAddresses"/>
            <field-map field-name="replyToAddresses" from="fromAddress"/>
            <field-map field-name="fromName" from="fromName"/>
            <field-map field-name="fromAddress" from="fromAddress"/>
            <field-map field-name="attachments" from="attachmentQrEmail"/>
            <field-map field-name="bodyParameters" from="bodyParameters"/>
        </service-call>
        <if condition="sendOut.messageId == null">
            <set field="result" from="false" />
            <return error="true" message="Error sending mail, rolling back"/>
        </if>
      </actions>
    </service>

```