ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DEL SOFTWARE

Sistema de gestión hotelera para plataformas iOS y Android

(Hotel management system for iOS and Android platforms)

Realizado por
Miguel Álvarez de Perea Castro
Tutorizado por
Eduardo Guzmán de los Riscos
Cotutorizado por
Carlos García Vega
Departamento
Lenguaje y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DE 2017

Fecha defensa:
El Secretario del Tribunal

Resumen: Este trabajo de fin de grado está realizado dentro del programa *Impulso TFG* promovido por el *Vicerrectorado de Proyectos Estratégicos* de la Universidad de Málaga en colaboración con la empresa malagueña *EasyStay Technologies S.L.* cuya principal actividad consiste en proporcionar al sector hotelero soluciones tecnológicas que mejoren la experiencia del huésped durante su estancia, así como mejorar los procesos de gestión realizados por los empleados.

El producto desarrollado por *EasyStay* consta de dos partes diferenciadas: la primera, una aplicación web destinada a la gestión para empleados donde reciben y procesan las peticiones de los huéspedes y, por otro lado, una aplicación móvil para los clientes que les permite estar en contacto continuo con recepción así como acceso a realizar pedidos, reservas u obtener información sobre el hotel o sus eventos.

La finalidad de este trabajo de fin de grado es adaptar la aplicación web a la aplicación móvil inicialmente destinada únicamente a los clientes, permitiendo así que un empleado pueda gestionar solicitudes desde un dispositivo móvil.

Palabras clave: *aplicación móvil híbrida, aplicación web, ionic framework, ruby on rails, Android, iOS, hoteles*

Abstract: This project is performed under the Impulso TFG program in collaboration with EasyStay Technologies S.L. a company focused on provide technologies solutions to hotel sector and improve the guest experience.

Their product has two different parts: in one hand, a web application whose purpose is to manage requests made by the guests such as orders or reservations and, in the other hand, a mobile application that customers use to make those requests.

The main goal of this project is to adapt the management web application into the mobile one in order to let employees to manage requests in a mobile device.

Keywords: *hybrid mobile application, web application, ionic framework, ruby on rails, Android, iOS, hotels*

Índice general

Capítulo 1. Introducción 13

1.1 Motivación 13

1.2 Objetivos 15

1.3 Materiales y tecnologías 17

1.3.1 Ruby on Rails 17

1.3.2 Ionic Framework 17

1.3.3 Pusher 19

1.3.4 Pushwoosh 19

1.4 Fases del proyecto 21

Capitulo 2. Análisis 23

2.1 Requisitos funcionales 23

2.2 Requisitos no funcionales 25

2.3 Casos de uso 26

Capítulo 3. Diseño y modelado del sistema 33

3.1 Autenticación 33

3.2 Accesos 34

3.3 Chats 35

3.4 Gestión de pedidos y reservas 35

3.5 Eventos 36

3.6 Incidencias 36

Capítulo 4. Desarrollo 37

4.1 Estructura de las aplicaciones 38

4.1.1 Servidor 38

4.1.2 Aplicación móvil 39

4.2 Autenticación 41

4.2.1 Servidor 41

4.2.2 Aplicación móvil 41

4.3 Accesos 43

4.3.1 Mostrar accesos 43

4.3.2 Aceptar y rechazar peticiones de acceso 47

4.3.3 Notificar una nueva petición de acceso 50

4.4 Reservas 52

4.4.1 Mostrar reservas 52

4.4.2 Aceptar y rechazar reservas 55
4.4.3 Notificar una nueva reserva 56
4.5 Pedidos 58
4.5.1 Mostrar pedidos 58
4.5.2 Aceptar pedidos 61
4.5.3 Notificar de un nuevo pedido 64
4.6 Chats 65
4.6.1 Mostrar listado de chats 65
4.6.2 Enviar y recibir mensajes 69
4.7 Eventos 74
4.7.1 Mostrar listado de eventos 74
4.7.2 Mostrar información de un evento 77
4.7.3 Notificar nueva inscripción a un evento 79
4.8 Incidencias 80
4.8.1 Mostrar incidencias 80
4.8.2 Solucionar incidencias 83
4.8.3 Notificar nueva incidencia 85
Capítulo 5. Conclusiones 87
5.1 Valoraciones finales 87
5.2 Trabajo futuro 89
Apéndices 90
Apéndice A 91
Instalación de los entornos 91
A.1 Aplicación web 91
A.2 Aplicación móvil 92
Bibliografía 94

Capítulo 1

Introducción

1.1 Motivación

En una sociedad cada vez más acostumbrada a usar la tecnología para interactuar con su entorno, es normal que surjan proyectos cuya finalidad es hacer accesible toda esa información que nos rodea.

Actualmente, hay sectores cada vez más concienciados con la importancia de adaptar sus negocios a las nuevas formas de consumo que los usuarios demandan.

Uno de esos sectores que está apostando por la modernización y actualización de su modelo de negocio es el *sector hotelero*. La cada vez mayor presencia de los *smartphones* en nuestra vida diaria ha acelerado este proceso de cambio con el que el sector espera alcanzar una posición de referencia tecnológica.

EasyStay Technologies S.L. tiene como principal objetivo ayudar a hoteleros y gestores de apartamentos a conseguir esa modernización tecnológica tan solicitada por el sector. Hoteles independientes, cadenas hoteleras y gestores de apartamentos (nacionales e internacionales) han apostado por las soluciones tecnológicas de la empresa.

Actualmente, el principal producto consiste en una aplicación móvil orientada a los clientes del hotel donde estos pueden registrar su estancia y acceder a todos los servicios que el hotel o apartamento pone a su disposición (véase reserva de productos, servicio de habitaciones, recomendaciones, información turística, chat con el personal…)

Por otro lado, el hotel o el gestor del apartamento dispone también de una aplicación web que le permite gestionar las solicitudes de registro, reservas, gestión de contenidos o notificaciones y chats, todo esto centralizado en una aplicación que mejora la eficiencia y el trato con el huésped.

Como se ha comentado anteriormente, la aplicación móvil está orientada a los clientes, mientras que la aplicación web está destinada a recepcionistas o encargados de apartamentos. Sin embargo, a diferencia de los recepcionistas en los hoteles, los gestores de apartamentos no suelen disponer de una recepción o espacio que les permita estar constantemente frente a un ordenador atendiendo las necesidades de sus clientes.

Este proyecto nace con la intención de desarrollar una aplicación móvil que permita a los gestores atender cierto tipo de solicitudes sin la necesidad de estar frente a un ordenador, una demanda que han transmitido numerosos hoteleros y dueños de apartamentos que buscan poder ofrecer a sus huéspedes una atención directa y personalizada.

1.2 Objetivos

Actualmente, cada hotel o apartamento dispone de una aplicación individual desde la cual los clientes acceden a los servicios del mismo. El acceso al módulo que desarrollaremos estará restringido a los gestores, que serán los únicos capaces de acceder al mismo.

Las funcionalidades y módulos que busca implementar este proyecto son las siguientes:

- *Accesos*: gestionar peticiones de accesos realizadas por los clientes, permitiendo al gestor confirmar o cancelar los mismos.

- *Gestión de reservas:* recibir en tiempo real las reservas de los clientes, pudiendo estas ser aceptadas o rechazadas, así como adjuntar mensajes.

- *Pedidos:* recibir notificaciones de los pedidos, pudiendo aceptarlos.

- *Chats:* permitir al gestor recibir y enviar mensajes en tiempo real con los usuarios que utilicen la aplicación móvil que tengan un acceso activo al hotel o apartamento.

- *Eventos:* mantener un control de los huéspedes que han solicitado acudir a un evento organizado por el hotel o apartamento.

- *Incidencias:* poder recibir alertas de incidencias transmitidas por los clientes, permitiendo así a los gestores acelerar el proceso de tramitación de las mismas y aumentando la satisfacción del huésped, que verá su solicitud atendida inmediatamente.

Una breve descripción de las tecnologías que conforman el sistema actual así como los cambios que realizaremos en cada uno de las plataformas se describe a continuación:

- Actualmente se utiliza *Ruby On Rails* para la *API REST*, por lo que las modificaciones necesarias se realizarán usando esta tecnología.

- *Ionic Framework* que permite la realización de aplicaciones híbridas para *iOS* y *Android* utilizando tecnologías web (*HTML*, *CSS*, *JavaScript* y *AngularJS*). La realización de una

aplicación híbrida permite que, con el desarrollo de un único proyecto y código, generar aplicaciones para ambas plataformas sin necesidad de hacer desarrollos independientes en cada uno de los lenguajes nativos de cada plataforma.

- *Pusher* como servicio *WebSocket* utilizado para el chat en tiempo real. Es necesario crear un canal que conecte ambos interlocutores. Estos canales deben ser únicos para cara uno de los hoteles.

- *PushWoosh* para proveer a la aplicación de notificaciones *push.* Este tipo de notificaciones son aquellas que el usuario recibe cuando la aplicación no está en primer plano.

1.3 Materiales y tecnologías

Para el desarrollo haremos uso de los siguientes recursos:

1. Ordenador *MacBook Pro* para el desarrollo del código y la memoria.
2. *RubyMine* como *IDE* para el desarrollo en *Ruby on Rails*.
3. *WebStorm* como *IDE* para el desarrollo en *Ionic*.
4. *Sequel Pro* como gestor de base de datos para *MySQL*.

El los siguientes apartados explicamos más en profundidad las tecnologías implicadas en el desarrollo del proyecto.

1.3.1 Ruby on Rails

La aplicación web está desarrollada utilizando *Ruby on Rails*, un *framework* para el desarrollo de aplicaciones web escrito en el lenguaje de programación *Ruby* que sigue las directrices del patrón Modelo-Vista-Controlador y es conocido por estar detrás de grandes aplicaciones como *GitHub*, *Shopify*, *Airbnb* o *Twitch*.

Además de *Ruby*, *Ruby on Rails* también hace uso de otras tecnologías altamente conocidas en el desarrollo web como *HTML*, *CSS* o *JavaScript* para las interfaces y vistas así como *XML* o *JSON* para la transferencia de datos. Este último tiene especial importancia en la interacción entre la aplicación web y la aplicación móvil ya que es el estándar utilizado por la *API* para el envío de información entre ambas plataformas.

Otra de los atractivos de *Ruby on Rails* son las llamadas *gems* (gemas), *plugins* que se añaden a los proyectos dotándoles de nuevas funcionalidades o herramientas. Por mencionar algunas, durante el desarrollo de este proyecto se han utilizado *Devise* o *Serializer*, de las que hablaremos más en profundidad más adelante.

1.3.2 Ionic Framework

Antes de hablar de la aplicación móvil es interesante introducir el concepto de aplicación híbrida, así como diferenciarlas de las aplicaciones nativas.

Cuando desarrollamos una aplicación para móviles, ya sea *Android* o *iOS*, suele optarse por desarrollarlas en los lenguajes nativos de cada plataforma, *Java* o *Swift* respectivamente. Este tipo de desarrollo tiene como gran ventaja el acceso de forma nativa a elementos del dispositivo como pueden ser la cámara o los sensores. A su vez ofrecen un aspecto más cuidado que siguen los estilos de diseño de cada una de las plataformas para las que son desarrolladas gracias a las herramientas que los *SDK* y las herramientas de desarrollo proveen a los desarrolladores.

Sin embargo, tienen una gran desventaja y es la necesidad de disponer de un equipo de desarrollo para cada plataforma lo que multiplica por dos los costes de desarrollo y mantenimiento de las aplicaciones y esto, en empresas con pocos recursos, puede suponer un gran problema.

Otra de las opciones es desarrollar una aplicación híbrida. Este tipo de aplicaciones hace uso de tecnologías web como *HTML*, *CSS* y *JavaScript* que posteriormente son son encapsuladas y ejecutadas en las distintas plataformas. La ejecución de estas aplicaciones se realiza sobre un navegador web en lugar del *framework* nativo de la propia plataforma.

Esto supone una gran ventaja respecto a las aplicaciones nativas ya que reduce el desarrollo a un único proyecto utilizando tecnologías tan asentadas y conocidas en el desarrollo web.

En empresas como *EasyStay* con recursos muy limitados esto es un aspecto crítico ya que el producto tiene que estar disponible para el mayor número de usuarios posibles, por lo que su presencia en ambas plataformas es más que obligada y diversificar el desarrollo para dos plataformas distintas no es una opción viable. Es por eso que el desarrollo de una aplicación híbrida es, sin duda, la opción que más se adapta a las necesidades de la empresa.

Dentro de las aplicaciones híbridas existen diferentes *frameworks* que facilitan la tarea de desarrollo añadiendo estilos y componentes que imitan el comportamiento y diseño de las aplicaciones nativas. Uno de ellos es *Ionic*, el utilizado para la realización de la aplicación móvil.

Ionic es un *framework* de código abierto que ofrece una gran cantidad de componentes, desde el estilo de los botones hasta los iconos, que se adaptan a las guías y estilos de diseño que las distintas plataformas definen para el desarrollo de sus aplicaciones.

1.3.3 Pusher

Pusher es un servicio que permite el envío y recepción de eventos en tiempo real, esto facilita que, por ejemplo, una vez se recibe un pedido el recepcionista reciba una notificación, así como notificar a los clientes que sus peticiones han sido aceptadas o rechazadas.

Pusher ofrece una librería en *JavaScript* que se adapta perfectamente a ambas aplicaciones y su funcionamiento es muy sencillo. Una vez que el servidor recibe un evento, ya sea enviar un mensaje por chat, aceptar una solicitud o solucionar una incidencia, el servidor dispara el evento correspondiente a través de un canal de comunicación que conecta el emisor y el receptor. Una vez el receptor detecta un evento, procede a realizar una acción como mostrar una ventana emergente, incrementar un indicador de mensajes no leídos o mostrar un mensaje.

1.3.4 Pushwoosh

Pushwoosh dota a la aplicación de la capacidad de mostrar notificaciones a los clientes cuando tienen la aplicación cerrada. Esta funcionalidad es muy importante ya que permite notificar a los usuarios de los cambios en sus solicitudes así o avisar de la recepción de nuevos mensajes a través del chat.

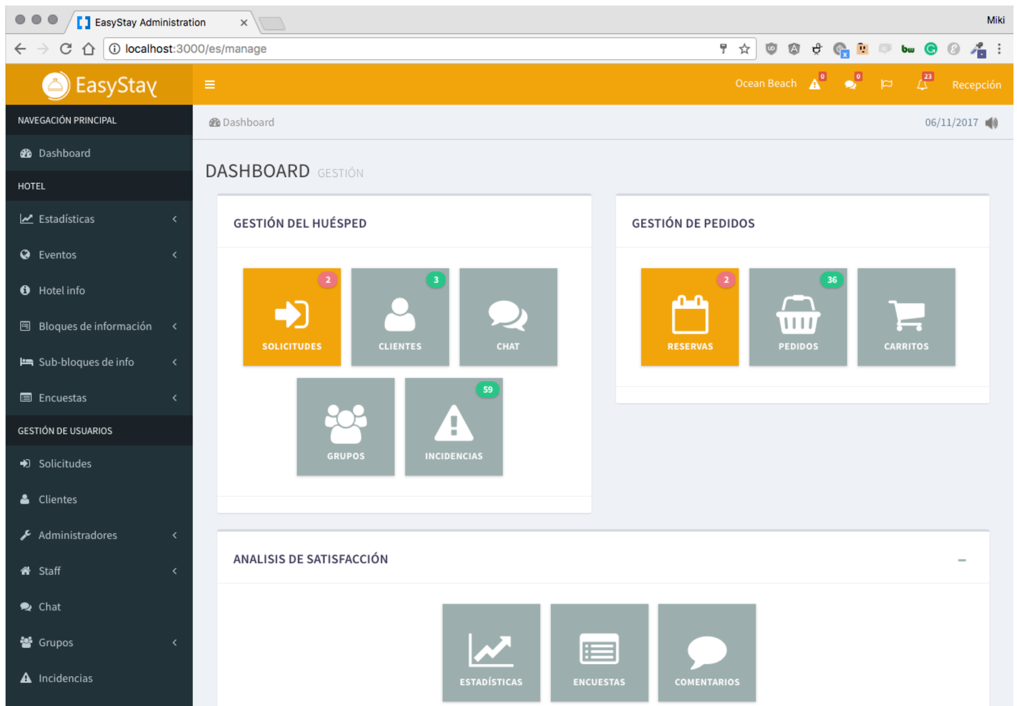


Figura 1.1 Aplicación web

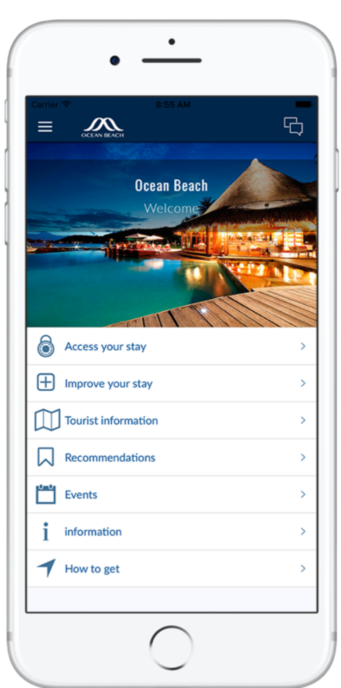


Figura 1.2 Aplicación móvil

1.4 Fases del proyecto

Durante la realización del proyecto, el desarrollo de cada uno de los módulos ha consistido en las siguientes fases:

1. Documentación.

Estudio del proceso de implementación de nuestra aplicación en el proyecto actual así, incluyendo un nuevo tipo de usuario que no estaba contemplado inicialmente en la aplicación.

2. Análisis de requisitos.

Realización de un estudio sobre los requisitos funcionales y no funcionales que determinan el comportamiento de la aplicación.

3. Diseño y modelado del sistema.

Estudiar y planificar el modelo que vamos a utilizar para dar forma al sistema, creando así la estructura en la que basaremos el funcionamiento de nuestra aplicación y la conexión entre las tecnologías que la compondrán.

4. Desarrollo.

Desarrollo los módulos de la aplicación híbrida usando *Ionic Framework* así como las diversas API en *Ruby On Rails* que la aplicación consumirá para su funcionamiento.

Para la aplicación será necesaria el desarrollo de:

- Un flujo para el nuevo tipo de usuario que accederá a la aplicación.
- Vistas, controladores y servicios para cada uno de los módulos: gestión de accesos, gestión de reservas, chats, eventos e incidencias.

- Una nueva API que será necesaria para consumir los datos que los nuevos módulos.
- Conexiones con servicios externos (*Pusher* y *PushWoosh*).

5. Pruebas y carga de información.

Realización de pruebas en un entorno real, sometiendo a la aplicación a una batería de situaciones cuya superación satisfactoria determinará que realiza correctamente sus funciones.

6. Redacción de la memoria.

Explicación del proceso de desarrollo de las aplicaciones y la documentación técnica asociada.

Capítulo 2

Análisis

En la etapa de análisis realizamos la captura de requisitos funcionales que definirán el comportamiento de nuestra aplicación. Para cada módulo a desarrollar definiremos requisitos específicos que variarán según la funcionalidad final del mismo.

A su vez, especificaremos los requisitos no funcionales que definirán el correcto funcionamiento del sistema haciendo referencia principalmente a requisitos de rendimiento o usabilidad.

2.1 Requisitos funcionales

Identificador	**Nombre**	**Descripción**
RF01	Acceder a la aplicación de gestión	La aplicación mostrará la interfaz de gestión si se introducen las credenciales correctas.
RF02	Mostrar accesos	La aplicación permitirá al usuario ver el estado de las peticiones de acceso
RF03	Aceptar accesos	El usuario podrá aceptar accesos pendientes
RF04	Rechazar accesos	El usuario podrá rechazar accesos pendientes
RF05	Enviar mensajes al rechazar accesos	Al rechazar un acceso, el usuario podrá enviar un mensaje que indique el motivo.
RF06	Recibir peticiones de accesos	La aplicación notificará al usuario de la recepción de nuevas peticiones de acceso

RF07	Mostar reservas	La aplicación mostrará al usuario la información de las reservas y sus estados
RF08	Aceptar reservas	El usuario podrá aceptar las reservas pendientes
RF09	Rechazar reservas	El usuario podrá rechazar reservas pendientes
RF11	Enviar mensajes al rechazar reservas	La aplicación permitirá al usuario mandar un mensaje al rechazar una reserva
RF12	Recibir peticiones de reserva	La aplicación notificará al usuario cuando una nueva petición sea recibida
RF13	Mostrar pedido	El usuario podrá ver la información de los pedidos y sus estados
RF14	Aceptar pedidos	El usuario podrá aceptar los pedidos pendientes
RF15	Recibir pedidos nuevos	La aplicación notificará al usuario de la recepción de nuevos pedidos
RF16	Mostrar listado de chats	La aplicación mostrará al usuario el listado de conversaciones
RF17	Buscar conversaciones	El usuario podrá buscar un chat utilizando el nombre del cliente
RF18	Enviar mensajes	El usuario podrá enviar mensajes a través de la aplicación
RF19	Recibir mensajes	El usuario podrá recibir mensajes a través de la aplicación
RF20	Recibir notificaciones de nuevos mensajes	La aplicación notificará al usuario de la recepción de nuevos mensajes
RF21	Mostrar listado de eventos	El usuario podrá ver un listado de eventos

RF22	Mostrar información de un evento	La aplicación mostrará los detalles de un evento
RF23	Recibir notificaciones de eventos	La aplicación notificará al usuario cuando se inscriban a un evento
RF24	Mostrar listado de incidencias	El usuario podrá ver un listado de las incidencias
RF25	Solucionar incidencias	La aplicación permitirá marcar incidencias como solucionadas
RF26	Recibir notificaciones de incidencias	La aplicación notificará al usuario de la recepción de nuevas incidencias

2.2 Requisitos no funcionales

Identificador	**Nombre**	**Descripción**
RNF01	Conexión con el servidor	La aplicación realizará peticiones al servidor, que responderá con los datos solicitados
RNF02	Diseño adaptativo	El diseño de la aplicación se adaptará a las distintas resoluciones de los dispositivos
RNF03	Autenticación basada en *tokens*	Se utilizará un sistema de autenticación basado en *tokens* para aumentar la seguridad del sistema

2.3 Casos de uso

CU01: autenticarse en el sistema.

- Descripción: el usuario introduce sus credenciales de tipo "*ContentManager*"
- Precondición: el usuario tiene una cuenta de tipo "*ContentManager*"
- Escenario principal:
    1. El usuario introduce su email
    2. El usuario introduce su contraseña
    3. El usuario pulsa el botón iniciar sesión
    4. El sistema verifica que se trata de un "*ContentManager*"
    5. El sistema carga la vista principal del módulo de gestión
- Postcondición: el usuario está identificado en el sistema
- Escenario alternativo

    4b. El sistema determina que no se trata de un "*ContentManager*"

    5b. El sistema carga la vista principal para los usuarios del hotel.

CU02: ver el listado de peticiones de acceso

- Descripción: el usuario podrá ver una vista con todas las peticiones de acceso.
- Precondición: CU01
- Escenario principal:
    1. El usuario pincha sobre el icono del módulo de gestión de accesos
    2. El sistema hace una petición solicitando la lista de peticiones de acceso
    3. El sistema carga la vista mostrando la lista de peticiones de acceso
- Postcondición: el usuario puede consultar la lista de peticiones de acceso
- Escenario alternativo

    3b. El sistema falla y no muestra las peticiones de acceso

    4b. El sistema muestra una ventana emergente indicando que ha habido un error

CU03: aceptar una peticiones de acceso

- Descripción: el usuario podrá aceptar una petición de acceso
- Precondición: CU01, CU02
- Escenario principal:
    1. El usuario pincha sobre el botón "*ACEPTAR*" de una petición de acceso
    2. El sistema muestra una venta de confirmación

3. El usuario confirma la acción
4. El sistema realiza la petición al servidor y modifica la petición de acceso
5. El sistema recarga la vista actual

- Postcondición: la petición de acceso ha sido confirmada
- Escenario alternativo

3b. El usuario cancela la acción en la ventana emergente

4b. El sistema cierra la ventana emergente y vuelve a mostrar el listado de peticiones de acceso.

CU04: rechazar una peticiones de acceso

- Descripción: el usuario podrá rechazar una petición de acceso
- Precondición: CU01, CU02
- Escenario principal:
    1. El usuario pincha sobre el botón "*RECHAZAR*" de una petición de acceso
    2. El sistema muestra una venta de confirmación
    3. El usuario introduce un mensaje
    4. El usuario confirma la acción
    5. El sistema realiza la petición al servidor y modifica la petición de acceso
    6. El sistema recarga la vista actual
- Postcondición: la petición de acceso ha sido rechazada
- Escenario alternativo

3b. El usuario cancela la acción en la ventana emergente

4b. El sistema cierra la ventana emergente y vuelve a mostrar el listado de peticiones de acceso.

CU05: ver el listado de reservas

- Descripción: el usuario podrá ver una vista con todas las reservas.
- Precondición: CU01
- Escenario principal:
    1. El usuario pincha sobre el icono del módulo de gestión de reservas
    2. El sistema hace una petición solicitando la lista de reservas
    3. El sistema carga la vista mostrando la lista de reservas
- Postcondición: el usuario puede consultar la lista de reservas
- Escenario alternativo

3b. El sistema falla y no muestra reservas

4b. El sistema muestra una ventana emergente indicando que ha habido un error.

CU06: aceptar una reserva

- Descripción: el usuario podrá aceptar una reserva
- Precondición: CU01, CU05. El estado de la reserva debe ser "*pending*"
- Escenario principal:
  1. El usuario pincha sobre el botón "*ACEPTAR*" de una reserva
  2. El sistema muestra una venta de confirmación
  3. El usuario confirma la acción
  4. El sistema realiza la petición al servidor y modifica la reserva
  5. El sistema recarga la vista actual
- Postcondición: la reserva ha sido confirmada
- Escenario alternativo

  3b. El usuario cancela la acción en la ventana emergente

  4b. El sistema cierra la ventana emergente y vuelve a mostrar el listado de reservas.

CU07: rechazar una reserva

- Descripción: el usuario podrá rechazar una reserva
- Precondición: CU01, CU05. El estado de la reserva debe ser "*pending*"
- Escenario principal:
  1. El usuario pincha sobre el botón "*RECHAZAR*" de una reserva
  2. El sistema muestra una venta de confirmación
  3. El usuario introduce un mensaje
  4. El usuario confirma la acción
  5. El sistema realiza la petición al servidor y modifica la reserva
  6. El sistema recarga la vista actual
- Postcondición: la reserva ha sido rechazada
- Escenario alternativo

  3b. El usuario cancela la acción en la ventana emergente

  4b. El sistema cierra la ventana emergente y vuelve a mostrar el listado de reservas

CU08: ver el listado de pedidos

- Descripción: el usuario podrá ver una vista con todos los pedidos.
- Precondición: CU01
- Escenario principal:
    1. El usuario pincha sobre el icono del módulo de gestión de pedidos
    2. El sistema hace una petición solicitando la lista de pedidos
    3. El sistema carga la vista mostrando la lista de pedidos
- Postcondición: el usuario puede consultar la lista de pedidos
- Escenario alternativo

    3b. El sistema falla y no muestra el listado de pedidos

    4b. El sistema muestra una ventana emergente indicando que ha habido un error

CU09: aceptar un pedido

- Descripción: el usuario podrá aceptar un pedido
- Precondición: CU01, CU08. La variable *status_id* del pedido debe ser 2 o 3
- Escenario principal:
    1. El usuario pincha sobre el botón "*ACEPTAR*" de un pedido
    2. El sistema muestra una venta de confirmación
    3. El usuario confirma la acción
    4. El sistema realiza la petición al servidor y modifica el pedido
    5. El sistema recarga la vista actual
- Postcondición: el pedido ha sido confirmada
- Escenario alternativo

    3b. El usuario cancela la acción en la ventana emergente

    4b. El sistema cierra la ventana emergente y vuelve a mostrar el listado de pedidos

CU10: ver el listado de chats

- Descripción: el usuario podrá ver una vista con todos los chats activos
- Precondición: CU01
- Escenario principal:
    1. El usuario pincha sobre el icono del chat en la barra superior
    2. El sistema hace una petición solicitando el listado de chats
    3. El sistema carga la vista mostrando la lista de chats

- Postcondición: el usuario puede consultar la lista de chats
- Escenario alternativo

    3b. El sistema falla y no muestra el listado de chats

    4b. El sistema muestra una ventana emergente indicando que ha habido un error

CU11: buscar una conversación

- Descripción: el usuario podrá buscar una conversación usando el nombre del cliente.
- Precondición: CU01, CU10
- Escenario principal:
    1. El usuario pincha sobre la caja de búsqueda
    2. El sistema muestra el teclado
    3. El usuario introduce el nombre del cliente
    4. El sistema filtra las conversaciones
    5. El sistema muestra únicamente la conversación del cliente buscado
- Postcondición: el listado muestra únicamente la conversación del cliente buscado
- Escenario alternativo

    5b. El sistema no encuentra una conversación y muestra un listado vacío

CU12: ver conversación

- Descripción: el usuario podrá ver una conversación
- Precondición: CU01, CU10
- Escenario principal:
    1. El usuario pincha sobre una conversación
    2. El sistema muestra la conversación
- Postcondición: el usuario puede ver la conversación seleccionada
- Escenario alternativo

    2b. El sistema falla y muestra un mensaje de error

CU13: enviar mensaje

- Descripción: el usuario podrá enviar un mensaje en una conversación
- Precondición: CU01, CU14
- Escenario principal:
    1. El usuario escribe el mensaje
    2. El usuario pulsa el icono de enviar

3. El sistema manda el mensaje
4. El sistema muestra el mensaje enviado en la conversación

- Postcondición: el usuario ha enviado correctamente el mensaje
- Escenario alternativo

3b. El sistema falla y no envía el mensaje

CU14: ver el listado de eventos

- Descripción: el usuario podrá ver una vista con todos los eventos.
- Precondición: CU01
- Escenario principal:

1. El usuario pincha sobre el icono del módulo de gestión de eventos
2. El sistema hace una petición solicitando la lista de eventos
3. El sistema carga la vista mostrando la lista de eventos

- Postcondición: el usuario puede consultar la lista de eventos
- Escenario alternativo

3b. El sistema falla y no muestra el listado de eventos

4b. El sistema muestra una ventana emergente indicando que ha habido un error

CU15: ver información de un evento

- Descripción: el usuario podrá ver la información de un evento
- Precondición: CU01, CU14
- Escenario principal:

1. El usuario pincha sobre uno de los eventos
2. El sistema hace una petición solicitando la información del evento
3. El sistema carga la vista mostrando la información del evento

- Postcondición: el usuario puede consultar la información del evento
- Escenario alternativo

3b. El sistema falla y no muestra el evento

4b. El sistema muestra una ventana emergente indicando que ha habido un error

CU16: ver el listado de incidencias

- Descripción: el usuario podrá ver una vista con todas las incidencias.
- Precondición: CU01
- Escenario principal:

1. El usuario pincha sobre el icono del módulo de gestión de incidencias
2. El sistema hace una petición solicitando la lista de incidencias
3. El sistema carga la vista mostrando la lista de incidencias

- Postcondición: el usuario puede consultar la lista de incidencias
- Escenario alternativo

  3b. El sistema falla y no muestra el listado de incidencias

  4b. El sistema muestra una ventana emergente indicando que ha habido un error

CU17: solucionar una incidencia

- Descripción: el usuario podrá marcar una incidencia como solucionada
- Precondición: CU01, CU16. El estado de la incidencia debe ser "*pending*" o "checked"
- Escenario principal:
  1. El usuario pincha sobre el botón "*MARCAR COMO SOLUCIONADO*" de una incidencia
  2. El sistema muestra una venta de confirmación
  3. El usuario confirma la acción
  4. El sistema realiza la petición al servidor y modifica la incidencia
  5. El sistema recarga la vista actual
- Postcondición: la incidencia ha sido marcada como solucionada
- Escenario alternativo

  3b. El usuario cancela la acción en la ventana emergente

  4b. El sistema cierra la ventana emergente y vuelve a mostrar el listado de incidencias

Capítulo 3

Diseño y modelado del sistema

La aplicación móvil estaba inicialmente pensada para ser utilizada únicamente por los clientes, por lo que toda la estructura de la propia aplicación y la *API* encargada de suministrar la información consumida por la misma estaba desarrollada en torno a este principio.

Antes de empezar con el desarrollo fue necesario estudiar el funcionamiento actual del sistema y su estructura . En este capítulo explicaremos este proceso e introduciremos alguno de los *plugins* utilizados por el servidor. Como hemos comentado anteriormente, en el lenguaje *Ruby* estos *plugins* reciben el nombre de *gemas* y utilizaremos algunas de ellas para realizar la autenticación de usuarios o responder a las peticiones a través de la *API*.

La complejidad del sistema se basa principalmente en la gran cantidad de modelos de los que dispone. Gracias a la versatilidad del producto, un hotel tiene la posibilidad de introducir información de todo tipo, por ejemplo: rutas, puntos de interés, recomendaciones o promociones para los productos.

Para este proyecto, muchas de estos modelos no tienen interés, por lo que los obviaremos y nos centraremos en aquellos que tienen relación directa con el desarrollo del mismo.

Dividiremos esta sección en cada una de las funcionalidades a implementar, explicando en cada una de ellas los modelos involucrados y sus relaciones.

3.1 Autenticación

Para explicar el proceso de autenticación de los usuarios es necesario hablar de *Devise*, una gema para *Ruby On Rails* que simplifica todos los procesos que rodean a la gestión de usuarios tales como creación de usuarios, almacenamiento seguro de contraseñas, cambio de contraseñas, uso de sesiones o incluso la creación de unas vistas básicas con los formularios para todas estas operaciones.

Una vez instalada la gema en el proyecto, es necesario un único comando para que *Devise* genere todo lo necesario para incluir la autenticación de usuarios en nuestra aplicación:

rails generate devise MODEL

Donde sustituimos MODEL por el nombre del modelo utilizado en nuestro proyecto para referenciar a los usuarios. En nuestro caso, el modelo se llama *User* y contiene, aparte de los campos generados automáticamente por *Devise*, campos como *device_token, authentication_token o type*.

Este último, el campo *type*, nos permite distinguir entre distintos tipos de usuarios. Esto es especialmente útil en el proyecto porque permite diferenciar entre los usuarios y los trabajadores del hotel, direccionando en cada caso a las vistas concretas y con la posibilidad de realizar las funciones acordes a cada rol. Para nuestro desarrollo, nos interesa saber que el campo *type* puede tomar las cadenas de caracteres "*Customer*" o "*ContentManager*".
Por otra parte tenemos el proceso de autenticación desde la aplicación móvil. En este caso se trata de una autenticación basada en *tokens* utilizando un módulo para *AngularJS* llamado *Satellizer* que soporta servicios como Google, Facebook o Twitter entre otros. Además permite el uso de email y contraseña como métodos de identificación.

Cuando el usuario introduce sus credenciales, *Satellizer* genera un *token* que es cifrado y añadido a la cabecera de cada una de las peticiones que se realiza para verificar así la autenticidad del usuario.

3.2 Accesos

El módulo de accesos permite a los trabajadores del hotel recibir las peticiones de acceso a la aplicación que los huéspedes realizan. Este acceso es imprescindible para poder disfrutar de servicios como el chat o la reserva y pedido de productos y servicios.

Una petición de acceso puede realizarse de dos maneras: solicitando un *preacceso* o un solicitando un *precheckin*. La diferencia entre ambas radica en que, en el caso del *precheckin*, tienes que introducir los datos personales de las personas que vayan a hospedarse. La idea de

realizar un *precheckin* es evitar tener que realizar el proceso al llegar al hotel, evitando así tiempos de espera en recepción.

Una vez realizada una petición de acceso, los recepcionistas pueden aceptar o rechazar las peticiones. En caso de ser aceptada la petición, al usuario se le asigna un acceso.

Todo este proceso queda reflejado en el siguiente diagrama, donde se muestra la relación entre los modelos *Hotel*, *User*, *Precheckin*, *Preaccess* y *Access*

3.3 Chats

Los huéspedes pueden comunicarse con la recepción mediante un chat integrado en la aplicación. Para utilizar el chat, los usuarios deben de tener previamente un acceso activo, evitando así que usuarios que no están actualmente en el hotel puedan comunicarse con recepción, por lo que cada chat estará relacionado con un acceso.

Por otro lado, los mensajes también tienen su propio modelo en la base de datos, existiendo una relación entre estos, un chat y un usuario.

3.4 Gestión de pedidos y reservas

Cuando un usuario solicita o reserva un producto, la petición llega a los trabajadores de recepción para que gestionen esa solicitud.

El modelo *Product* almacena la información referente a cada producto del hotel. Estos productos pueden ser de tres tipos: "*on_sold*", "*reservation*" o "*read_only*". Este tipo queda determinado por la columna *type_product*.

Los productos *on_sold* son aquellos que pueden ser añadidos al carrito de los clientes, mientras que aquellos de tipo *reservation* necesitan una aprobación previa por parte de la recepción del hotel.

Por otro lado, los de tipo *read_only* simplemente son mostrados en la aplicación, sin permitir que los clientes puedan reservarlos o solicitarlos.

3.5 Eventos

El hotel tiene la posibilidad de anunciar eventos a los que los clientes pueden asistir, confirmando su asistencia desde la aplicación. Para la creación de estos eventos y la asistencia de los clientes quedan especificadas mediante los modelos *Events* y *Events*_users, donde el primero contiene la información referente al evento y el segundo relaciona los eventos con los usuarios que han indicado su asistencia. La relación entre estos modelos, el hotel y los usuarios es la siguiente:

3.6 Incidencias

La última funcionalidad a implementar es la capacidad de gestionar incidencias desde la aplicación móvil. Estas incidencias son interpuestas por los clientes y es muy importante para los hoteles ser capaces de gestionarlas en la mayor brevedad posible.

Las incidencias se modelan utilizando el modelo *Incidents*, relacionada con un usuario y un hotel, conteniendo campos como el estado de la incidencia, el mensaje que el usuario ha indicado o el tipo de incidencia.

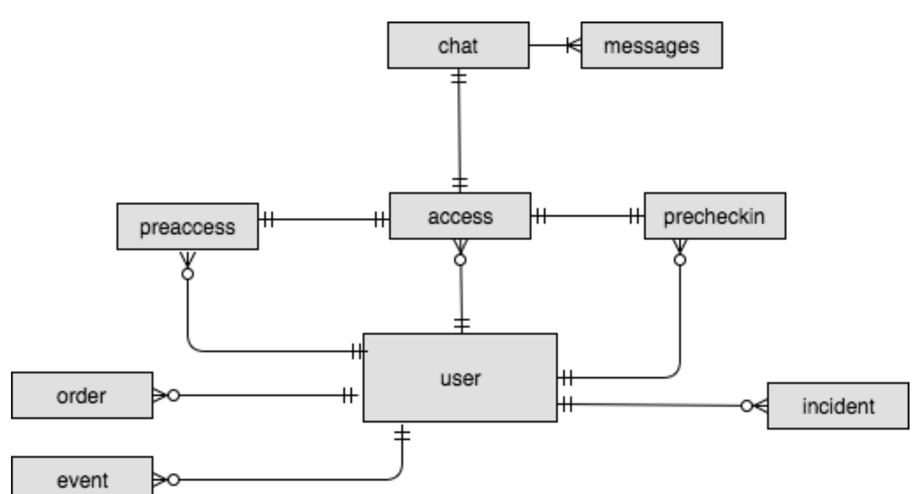


Figura 3.1 Diagrama del sistema

Capítulo 4

Desarrollo

La arquitectura del proyecto se basa en una aplicación cliente-servidor donde la aplicación móvil realiza peticiones a la aplicación web. Estas peticiones son procesadas por *Ruby On Rails* que, una vez procesada, la aplicación web devolverá la respuesta serializada en formato JSON que será tratada por la aplicación móvil para ser mostrada a los usuarios.

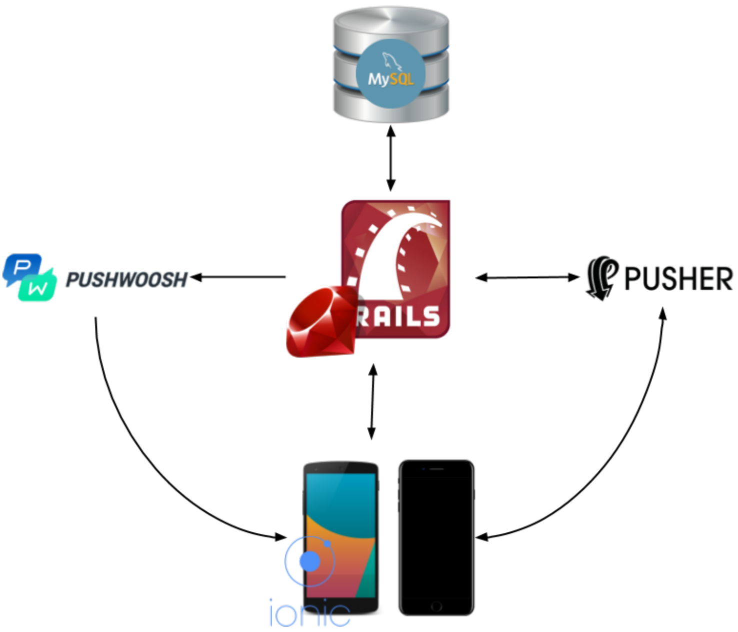


Figura 4.1 Estructura cliente servidor del proyecto

En esta sección profundizaremos en el desarrollo realizado en ambas aplicaciones para alcanzar los objetivos definidos anteriormente. Todos los casos tendrán dos partes diferenciadas: el desarrollo en la parte móvil y la creación de la API correspondiente en la parte del servidor.

4.1 Estructura de las aplicaciones

En este apartado comentaremos la estructura básica de ambas aplicaciones. Todos los módulos desarrollados siguen la misma estructura, por lo que es interesante dedicar una sección para definir los distintos ficheros y componentes que participan en cada uno de los procesos.

4.1.1 Servidor

Cuando la aplicación móvil solicita información al servidor lo hace mediante una petición http a una dirección concreta mediante métodos GET, PUT o POST. *En Ruby On Rails*, estas direcciones quedan definidas en el archivo *routes.rb* que es el encargado de asignar cada petición a la función del controlador correspondiente. Un ejemplo de la creación de las rutas podemos verlo a continuación, donde observamos la línea que debemos añadir al fichero *route.rb* y las rutas que genera, enlazando las URL a los controladores y sus funciones.

```
resources :access_requests, only: [:index, :update]
```

Como hemos comentado anteriormente, estas respuestas son serializadas en formato *JSON*. Este proceso de *serialización* de los objetos se realiza utilizando una gema llamada *Serializer*.

Serializer nos permite crear un fichero donde definimos la estructura del *JSON* que queremos devolver especificando los atributos del objeto o definir nuestros propios atributos.

Supongamos que hemos recibido una petición solicitando la información de un evento, por lo que tendremos el controlador de eventos y una función *show* que nos devuelve el evento que hemos solicitado.

```
class EventsController < ApiController

  def show
    @event = @current_hotel.events.find(params[:id])
    render json: @event, serializer: ManageEventSerializer
  end

end
```

Al final de la función especificamos que la respuesta va a ser un *JSON* y que utilizaremos el fichero *ManageEventSerializer* para serializar el objeto, que contiene tanto atributos propios del objeto como un atributo definido por nosotros mismos.

```
class ManageEventSerializer < ActiveModel::Serializer
  attributes :id, :hotel_id, :name, :description, :begin_hour

  def begin_hour
    if object.begin_hour
      object.begin_hour.strftime '%Y-%m-%d %H:%M:%S'
    end
  end
end
```

Para cada uno de los módulos que vamos a implementar deberemos crear tanto un controlador, que responderán a las peticiones que se realicen a la API por parte de la aplicación móvil, como los archivos para serializar a *JSON* los objetos que tengamos que devolver.

4.1.2 Aplicación móvil

Para el desarrollo de la aplicación móvil tendremos que desarrollar los controladores, servicios y las vistas de cada uno de los módulos.

Como hemos explicado anteriormente, *Ionic* encapsula la aplicación en un navegador web por lo que debemos especificar rutas como si de una aplicación web se tratara. Estas rutas se definen en el fichero routes.js y en él se indican la *URL*, la vista y su controlador asociado.

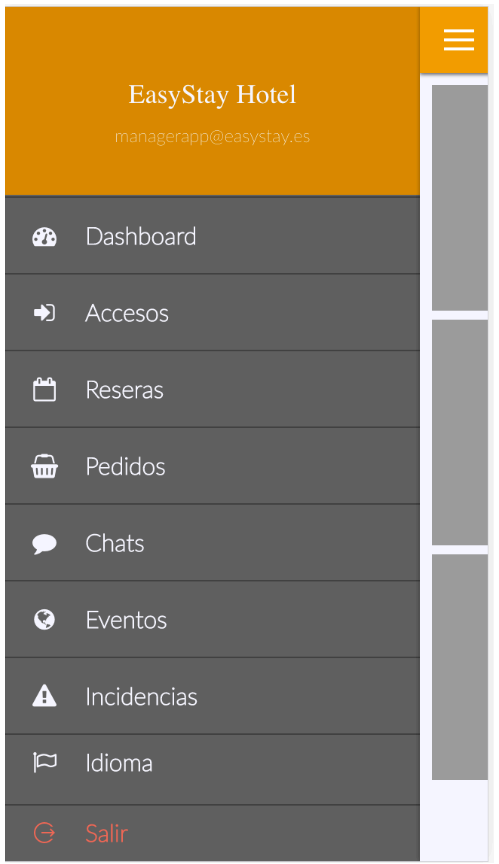
```
$stateProvider.state('app.manage_incidents', {
  url: '/manage/incidents',
  cache: false,
  views: {
    layoutContent: {
      templateUrl: 'manage/incidents/incidents.view.html',
      controller: 'ManageIncidentsCtrl'
    }
  }
});
```

Una vez que se accede a la *URL* especificada, se crean una instancia del controlador y se muestra la vista con la información recogida anteriormente por el controlador. Para recoger los datos, los controladores hacen uso de los servicios que realizan las peticiones al servidor y esperan la respuesta.

La ventana principal de la apartado de gestión contiene los botones que redireccionan a cada uno de los módulos que hemos desarrollado. Cada uno de estos botones tiene una ruta específica a la que redirecciona, cargando el controlador y la vista especificados en el route.js mencionado anteriormente

```
<div class="col manage-button" ng-class="events_active ? 'manage-orange-background' : "" ui-
sref="app.manage_events({hotel_id: hotel.id})">
  <span><i class="icon fa fa-globe"></i></span>
</div>
```

A los módulos también es posible acceder desde un menú lateral que se despliega y permite al usuario navegar por la aplicación.



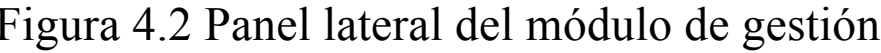


Figura 4.2 Panel lateral del módulo de gestión

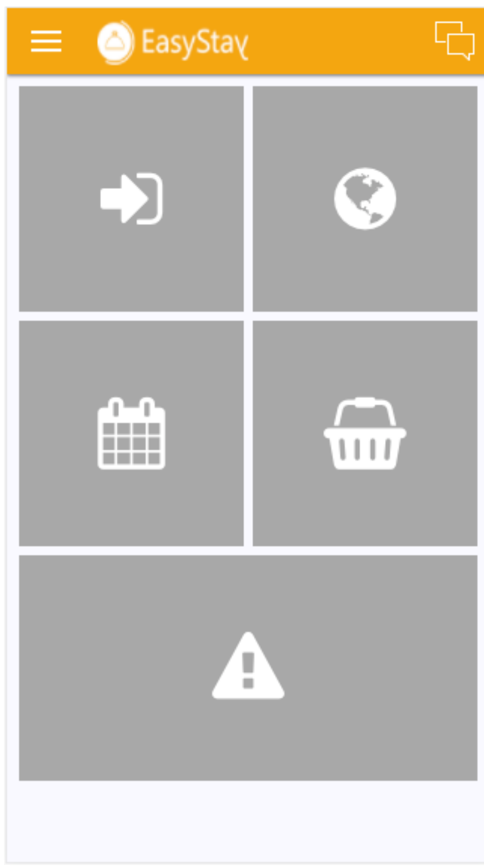


Figura 4.3 Vista principal

4.2 Autenticación

En apartados anteriores hemos explicado el funcionamiento del sistema de autenticación, introduciendo *Devise* y *Satellizer*. En este apartado explicaremos las modificaciones realizadas referente al proceso de autenticación.

4.2.1 Servidor

Como se ha comentado en varias ocasiones, la aplicación móvil estaba destinada exclusivamente a los usuarios del hotel. Debido a esto, el servidor buscaba en la base las credenciales usando el modelo *Customer*, que hereda de *User*, por lo que fue necesario modificar esta consulta. A su vez, añadimos el tipo de usuario al fichero *serializer* para poder así identificar qué tipo de usuario está accediendo a la aplicación móvil.

```
class AuthenticateApiRequest

  def user
    @user ||= User.find_by(authentication_token: decoded_auth_token[:user]) if
     decoded_auth_token && !decoded_auth_token[:user].nil?
    @user || errors.add(:authentication, I18n.t('api.invalid_token')) && nil
  end

end

class CustomerSerializer < ActiveModel::Serializer
  attributes :id, :email, :authentication_token, :type

  has_one :user_profile, root: 'profile'

end
```

4.2.2 Aplicación móvil

Desde la aplicación móvil, el usuario introduce sus credenciales en el formulario de ingreso y se realiza la petición al servidor para que verifique que son correctos. Como hemos comentado, esta petición se realiza añadiendo a la cabecera el *token* generado y cifrado por *Satellizer*.

Una vez obtenida una respuesta satisfactoria se realizan una serie de procesos como almacenar localmente la información del usuario, registrar el identificador del dispositivo (necesario para los notificaciones mediante *PushWoosh*), registrar el canal de comunicación que responderá a los eventos lanzados por *Pusher* y finalmente redireccionar al usuario a la vista correspondiente.

El canal de *Pusher* al que debe registrarse un cliente del hotel es distinto al que deben hacerlo los trabajadores, por eso debemos hacer distinción entre ambos tipos de usuarios. Esta

distinción la hacemos utilizando el atributo *type* que nos devuelve el servidor. En caso de que sea "*ContentManager*" registramos en *Pusher* el canal propio del hotel y *redireccionamos* al usuario a la vista para la gestión del hotel.

```
$rootScope.current_user = data;
// If user is a ContentManager, redirect to Manager App
if ($rootScope.current_user.type === 'ContentManager'){
  managerGoAfterSignIn();
  return $state.go('app.manage', {hotel_id: $rootScope.hotel.id});
}else{
  customerGoAgterSignIn();
}


function managerGoAfterSignIn(){
  //Get chat channels and subscribe
    var channels = PusherService.getChannels();
    var channel = APPCONFIG.hotel.id + '_hotel_channel';

    if(_.isEmpty(channels) || !_.contains(_.pluck(channels,'name'), channel)){
      PusherService.setManagerChannel(channel);
    }

  }
```

4.3 Accesos

4.3.1 Mostrar accesos

4.3.1.1 Servidor

Los usuarios pueden solicitar acceso realizando dos tipos de peticiones, un *preacceso* o un *precheckin*, por lo que en la API debemos devolver el conjunto que forman ambos modelos. A pesar de ser distintos, comparten los mismos atributos básicos que necesitamos para mostrar en la aplicación móvil, por lo que podemos utilizar el mismo *serializer* para los objetos de ambos tipos.

```
//Función que devuelve el conjunto de preaccess y precheckin
def index
  preaccess = Preaccess.where(hotel_id: @current_hotel.id).reverse_order
  precheckin = Precheckin.where(hotel_id: @current_hotel.id).reverse_order

  access_requests = preaccess + precheckin
  render json: access_requests, each_serializer:ManageAccessRequestSerializer
end

//Serializer para las peticiones de acceso
class ManageAccessRequestSerializer < ActiveModel::Serializer
  attributes :id, :arrival_date, :departure_date, :state, :comment, :type
  has_one :customer, root: "customer"

  def type
    object.class.name
  end
end
```

4.3.1.2 Aplicación móvil

Cuando el usuario accede al módulo de accesos inicialmente se le muestra un listado con todos las peticiones realizadas. En este listado aparecen los accesos que han sido aceptados, rechazados y los que están pendiente de ser respondidos.

El sistema de rutas de *AngularJS* carga la vista y el controlador referente a las peticiones de acceso una vez que el usuario ha pinchado sobre el icono en el menú principal. Al cargar el controlador, hace uso del servicio para realizar la petición a la *API* descrita en el punto anterior que devuelve el listado de accesos que, una vez que se recibe satisfactoriamente, son almacenados en la variable *access_requests* sobre la que posteriormente iteraremos y usaremos para mostrar la vista.

```
// Función del controlador que solicita al servicio el listado de peticiones
function activate(){
  LoaderService.show();
  ManageAccessRequestsService.all().then(
    function (data) {
      $scope.access_requests = data;
      LoaderService.hide();
    },
    function (error) {
      LoaderService.hide();
    }
  );

}

//Función del servicio que hace la petición al servidor
function all() {
  var deferred = $q.defer();
  var _url = APPCONFIG.api.url + 'manage/access_requests';

  $http({
    method: 'GET',
    url: _url
  })
    .success(function(resp) {
      deferred.resolve(resp);
    })
    .error(function(error) {
      deferred.reject(error);
    });

  return deferred.promise;
}
```

Para la vista hacemos uso de la directiva *ng-repeat* que nos permite repetir un bloque de código iterando sobre un *array* de objetos en este caso de *access_requests*. Dentro de este bloque definimos la apariencia de la tarjeta que contendrá la información de la petición de acceso, compuesta de una cabecera que tendrá un color descriptivo en función del estado de la petición. En caso de que la petición esté pendiente de confirmar, la tarjeta mostrará dos botones cuyo funcionamiento explicaremos en los siguientes apartados.

```
//Código HTML con la directiva ng-repeat de AngularJS
<div ng-repeat="access_request in access_requests | filter: filterByState">

    // ng-class determina el estilo de la cabecera en función del estado
  <div ng-class="{ 'checked' : access_request.state == 'pending', 'completed' : access_request.state == 'confirmed',
'error-order' : access_request.state == 'rejected'}" >

      //Ng-show determina que titulo poner en la cabecera según el estado
    <h1 ng-show="access_request.state == 'pending'" class="title">
                  {{'manage.pending' | translate}}
```

```
    </h1>

   <h1 ng-show="access_request.state == 'confirmed'" class="title">
                {{'manage.confirmed' | translate}}
    </h1>

   <h1 ng-show="access_request.state == 'rejected'" class="title">
                {{'manage.rejected' | translate}}
    </h1>
  </div>

  //ng-if mostrará los el bloque si se cumple la condición booleana
  <div ng-if="access_request.state == 'pending'>
   //ng-click ejecuta la función cuando el botón es pulsado
   <button ng-click="rejectRequest(access_request.id,access_request.type)">
    {{'manage.reject_text' | translate }}
   </button>

   <button ng-click="acceptRequest(access_request.id,access_request.type)">
    {{'manage.accept_text' | translate }}
   </button>

  </div>

</div>
```

El usuario también puede filtrar el listado en función del estado del mismo utilizando los botones superiores. Cada uno de los botones cambia el valor de la variable *filter.state* definida en el controlador. Como se puede observar anteriormente, dentro del *ng-repeat* hemos añadido el componente *filter*. Este componente mostrará aquellos elementos que cumplan la condición booleana de la función *filterByState* definida en el controlador.

```
<div class="row">

    //ng-click cambia el estado de la variable para realizar el filtrado
  <button ng-click="filter.state='all'">
   {{'manage.all' | translate }}
  </button>

  <button ng-click="filter.state='pending'">
   {{'manage.pending' | translate }}
  </button>

  <button ng-click="filter.state='confirmed'">
   {{'manage.confirmed' | translate }}
  </button>

</div>

//Función del controlador que filtra los elementos que cumplen la condición
function filterByState(item) {
  return item.state === $scope.filter.state || $scope.filter.state === 'all'
}
```

El funcionamiento de este filtro será parecido en otros modelos en los que también se permite el filtrado del listado.

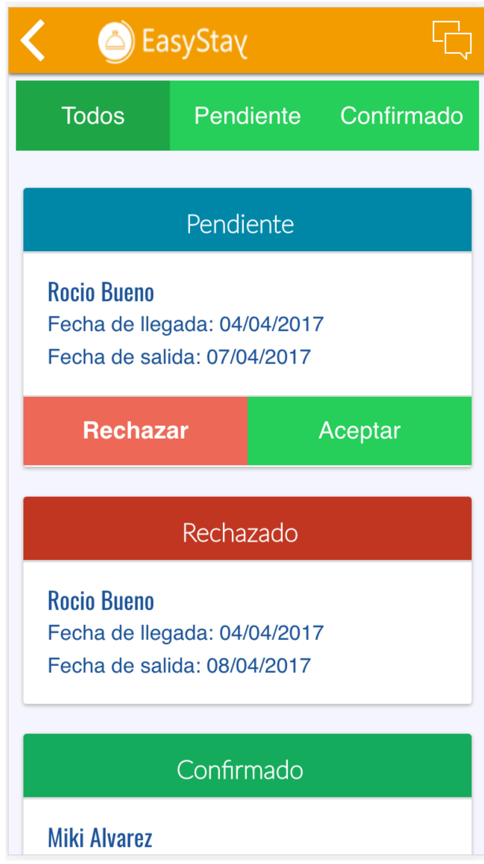


Figura 4.4 Vista de peticiones de acceso

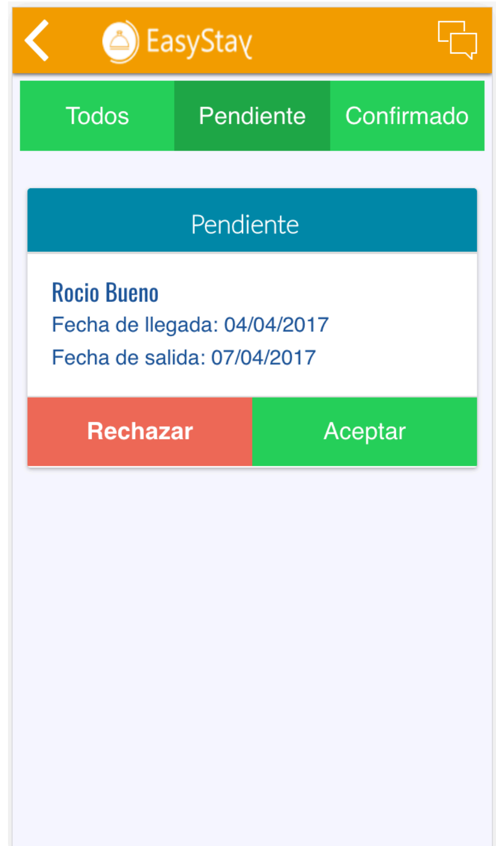


Figura 4.5 Vista de peticiones de acceso pendientes

4.3.2 Aceptar y rechazar peticiones de acceso

4.3.2.1 Servidor

Para aceptar o rechazar un *preacceso* debe hacerse una petición *PUT* indicando el id de la petición, el tipo y si ha sido aceptada o rechazada. Esta petición está asociada a la función *update* donde, en función del tipo, buscaremos y actualizaremos el objeto con el estado que se nos haya indicado en la petición, ya sea aceptada o rechazada. En caso de ser aceptada, se crea un acceso utilizando la información del *preaccess* o *precheckin* correspondiente

En ambos casos el proceso es muy similar: se busca en la base de datos el objeto mediante el id que hemos pasado por parámetro en la *URL* y lo actualizamos con su nuevo estado. En este momento se envían las notificaciones *push* al usuario, tanto las notificaciones de *Pusher* que el usuario recibirá si tiene la aplicación como las de *Pushwoosh*, que recibirá si, por el contrario, tiene la aplicación cerrada.

```
def update

  if params[:type] == 'Preaccess'
    update_preaccess
  elsif params[:type] == 'Precheckin'
    update_precheckin
  else
    render json: {error: "Error"}
  end
end

def update_preaccess

  @preaccess = Preaccess.find_by(id: params[:id], hotel_id: @current_hotel.id)

  if @preaccess.update(state: params[:state])

      //Enviar notificación PushWoosh
    new_notification = SendStateAccess.new
    new_notification.send_state_access(params[:state])

      //Enviar notificación Pusher
      pusher = PusherNotifications.new
    pusher.access_notification @preaccess, params[:state]

    render json: @preaccess, each_serializer: ManageAccessRequestSerializer

  else
    render json: {error: "Error"}

  end
end
```

4.3.2.2 Aplicación móvil

Otra de las funcionalidades de este módulo es poder aceptar y rechazar peticiones de acceso. Cuando una de las peticiones tiene "*pending*" como *state*, la interfaz muestra los botones para cada una de las acciones. Para ocultar o mostrar estos botones hacemos uso de la directiva *ng-if* de *AngularJS*, cuyo funcionamiento básicamente consiste en mostrar aquellos elementos que cumplen la condición booleana indicada. Cada uno de estos botones tiene asociado una función del controlador que es ejecutada cuando el botón es pulsado. Esta asignación se realiza con la directiva *ng-click*.

En el caso de que el usuario acepte la petición, una ventana emergente aparecerá para que confirme su decisión. Una vez aceptada hace uso del servicio indicará el identificador y el tipo, *precheckin* o *preaccess*, para mandar que sean mandados junto a la petición. Una vez recibida una respuesta satisfactoria, la vista se actualiza y se muestra el nuevo estado de la petición.

Por el contrario, si el usuario decide rechazar la petición, una ventana emergente solicitará indicar un mensaje que será enviado al cliente como motivo del rechazo. El proceso es similar al caso anterior con la única diferencia de que añadimos el mensaje a la petición.

Para mostrar las ventanas emergentes usamos la librería *ionicPopUp* predefinida de *Ionic* que permite mostrar distintos tipos de ventanas en función de lo que queramos transmitir, ya sea una alerta o una ventana de confirmación.

```
<div ng-if="access_request.state == 'pending'">

  <button ng-click="rejectRequest(access_request.id, access_request.type)">
    {{'manage.reject_text' | translate }}
  </button>

  <button ng-click="acceptRequest(access_request.id, access_request.type)">
    {{'manage.accept_text' | translate }}
  </button>

</div>


function acceptRequest(id, type){
  var confirmPopup = $ionicPopup.confirm({
    title: $translate.instant('confirmation'),
    template: $translate.instant('manage.request_confirmation'),
    cancelText: $translate.instant('answers.no'),
    okText: $translate.instant('answers.yes')
  });

  confirmPopup.then(function(res) {
    if (res) {
```

```
        LoaderService.show();
        ManageAccessRequestsService.accept_request(id, type).then(
          function (data) {
            LoaderService.hide();
            $state.go($state.current, {}, {reload: true});
          },
          function (error) {
            $ionicPopup.alert({
              title: $translate.instant('alertService.titleError'),
              template: $translate.instant('alertService.templateTryAgain')
            });
            LoaderService.hide();
          }
        );

      } else {

      }
    });
  }
```

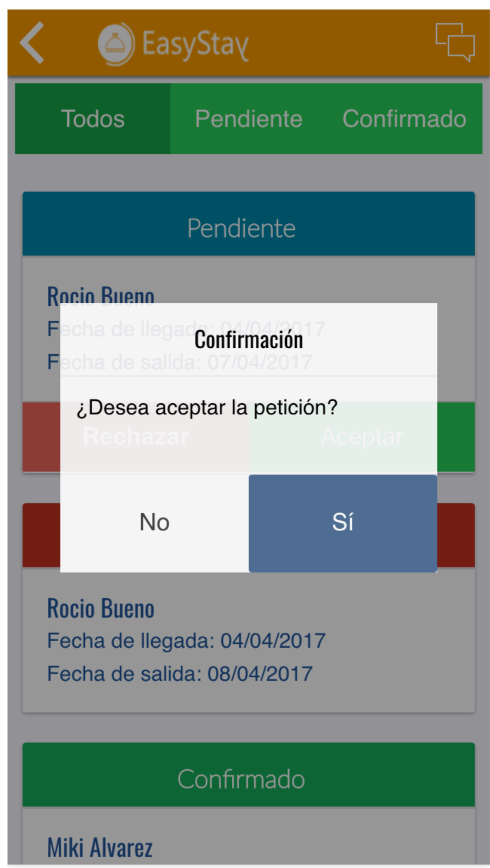


Figura 4.6 Ventana emergende de confirmación

Figura 4.7 Ventana emergente para rechazar acceso

4.3.3 Notificar una nueva petición de acceso

Cuando un usuario realiza una petición de acceso, desde el servidor se lanza un evento a través del canal del hotel que es recogido por la aplicación web para mostrar una notificación.

Para recibir ese evento también en la aplicación móvil debemos enlazar el evento con la función que se lanzará cuando sea recibido. Este proceso se realiza durante el proceso de autenticación explicado anteriormente, donde se enlazan los eventos con las funciones correspondientes. En este caso, los eventos son '*new_preaccess*' y '*new_precheckin*', enlazados con la función *newAccessRequest* que actualiza la variable *access_request_active* al valor booleano true. Esta variable la utilizamos en el menú principal del módulo de gestión para cambiar el color de fondo de los iconos que representan cada uno de los módulos mediante una clase *CSS* que cambia el color de fondo a naranja. Para esto usamos la directiva *ng-class* que nos permite indicar una clase si se cumple una condición booleana.

```
channel.bind('new_preaccess', newAccessRequest);
channel.bind('new_precheckin', newAccessRequest);

function newAccessRequest(data){
  $rootScope.access_requests_active = true;
}

//ng-class añade la clase si se cumple la condición booleana
<div class="col manage-button" ng-class="access_requests_active ? 'manage-orange-background' : ''"
  <span><i class="icon fa fa-sign-in"></i></span>
</div>
```

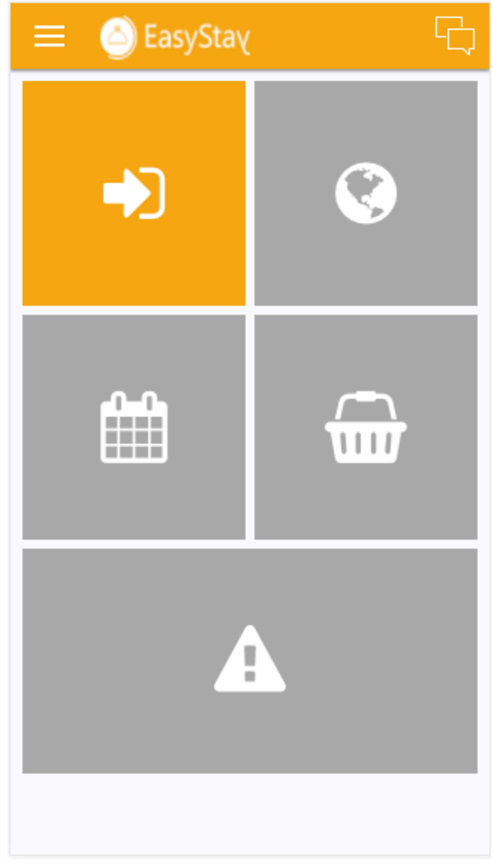


Figura 4.8 Icono notificando un nuevo acceso

4.4 Reservas

4.4.1 Mostrar reservas

4.4.1.1 Servidor

Para recibir el listado de reservas, la *API* proporciona una ruta que devuelve las reservas realizadas. Esta ruta, como hemos explicado anteriormente, se define en el fichero *route.rb*. En este caso, la petición se realizará a la *URI "/manage/reservations"* mediante el método *GET*.

Como en otros casos, debemos también especificar el *serializer* que usaremos. En el *serializer* de las reservas podemos indicar que también incluya el producto y el cliente que ha realizado la solicitud mediante haciendo uso de *"has_one"*

```
def index
  @reservations = Reservation.where(hotel_id: @current_hotel.id).order(created_at: :desc)

  render json: @reservations, each_serializer: ManageReservationSerializer
end

class ManageReservationSerializer < ActiveModel::Serializer
  attributes :id, :people, :date, :hour, :comment, :state, :state_comment
  has_one :customer, root: "customer"
  has_one :product, root: "product"

end
```

4.4.1.2 Aplicación móvil

Una vez pulsado el botón en la vista principal, el usuario es redireccionado al listado de reservas donde se mostrarán los elementos previamente almacenados tras realizar la petición al servidor. En el proceso intervienen, como en el caso de las peticiones de acceso, tanto el controlador como el servicio.

```
function activate(){
  LoaderService.show();
  ManageReservationsService.all().then(
    //success
    function (data){
      $log.debug(data);
      $scope.reservations = data;
      LoaderService.hide();
    },
    //error
    function(error){
      $log.debug(error);
      LoaderService.hide();
```

```
    }
  );
}

function all(hotel_id) {
  var deferred = $q.defer();
  var _url = APPCONFIG.api.url + 'manage/reservations';

  $http({
    method: 'GET',
    url: _url
  })
  .success(function(resp) {
    deferred.resolve(resp);
  })
  .error(function(error) {
    deferred.reject(error);
  });

return deferred.promise;
}
```

La interfaz es similar a la vista anteriormente en la lista de peticiones de acceso. Dispone de los botones para filtrar las reservas así como tarjetas que muestran la información de cada una de las solicitudes. En caso de que el estado sea "*pending*" mostramos los botones que nos permitirán aceptar o rechazar la reserva.

```
<div ng-if="reservation.state == 'pending'">

  <button ng-click="rejectReservation(reservation.id)">
    {{'manage.reject_text' | translate }}
  </button>

  <button ng-click="acceptReservation(reservation.id)">
    {{'manage.accept_text' | translate }}
  </button>

</div>

function filterByState(item) {
  return item.state === $scope.filter.state || $scope.filter.state === 'all'
}
```

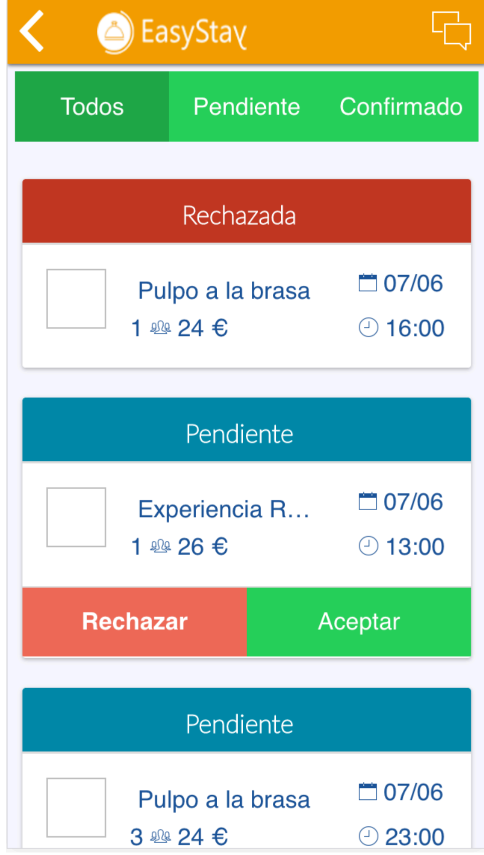


Figura 4.9 Listado de reservas

4.4.2 Aceptar y rechazar reservas

4.4.2.1 Servidor

El servidor proporciona una ruta para modificar una reserva. Esta petición debe hacerse a la *URI "manager/reservation/:id"* y el método *PUT*, donde id coincide con el identificador de la reserva a modificar. En este caso, la única modificación posible es la del estado de la reserva, que será "*accepted*" o "*rejected*" en función de la decisión tomada por el trabajador de recepción que tramite la petición. Esta ruta apunta a la función update del controlador de las reservas donde, en caso de modificar satisfactoriamente la reserva, también se envían las notificaciones push destinadas al usuario.

```
def update
  @reservation = Reservation.find_by(id: params[:id], hotel_id: @current_hotel.id)
  if @reservation.update(state: params[:state])

    #Send pusher notification
    pusher = PusherNotifications.new
    pusher.reservation_notification @reservation
    # Send push mobile notification
    new_notification = SendStateReservation.new
    new_notification.send_state_reservation(params[:state])

    render json: @reservation, serializer: ManageReservationSerializer
  else
    render json: {error: "Error"}
  end
end
```

4.4.2.2 Aplicación móvil

El proceso para aceptar o rechazar reservas sigue la misma estructura que la explicada en el proceso análogo para una petición de acceso, donde el usuario, al aceptar o rechazar una reserva, realiza una petición al servidor mediante un servicio indicando el *id* de la reserva y la acción, así como un mensaje en caso de que la reserva haya sido rechazada.

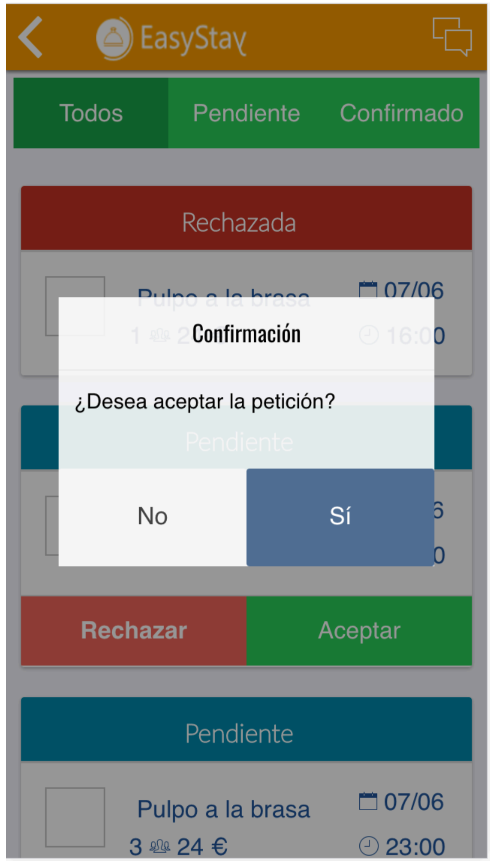


Figura 4.10 Venta emergente para aceptar reserva

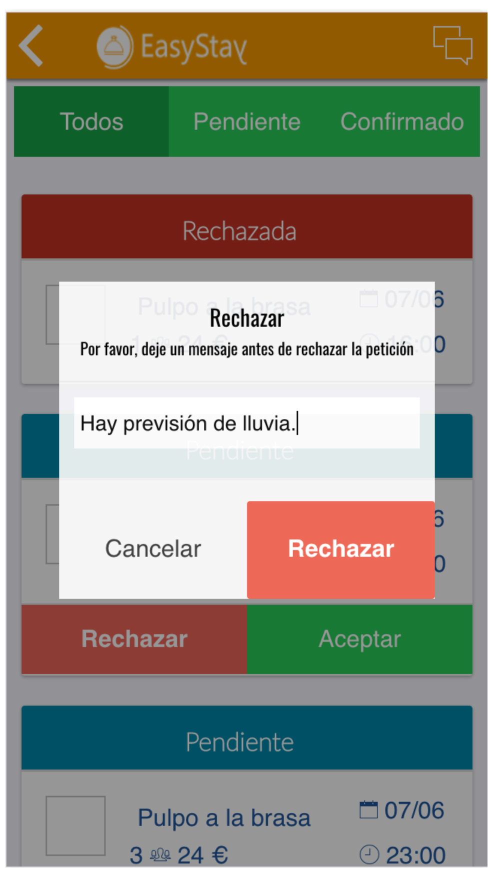


Figura 4.11 Ventana emergente para rechazar reservas

4.4.3 Notificar una nueva reserva

Para notificar de la recepción de una nueva reserva se asocia el evento *"new_reservation"* a la funcion *newReservation* que asigna a la variable reservations_active el valor true, activando así la clase que cambia el color de fondo del icono a naranja.

```
channel.bind('new_reservation', newReservation);
function newReservation(data){
  $rootScope.reservations_active = true;
}

<div ng-class="reservations_active ? 'manage-orange-background' : ''">
  <span><i class="icon fa fa-calendar"></i></span>
</div>
```

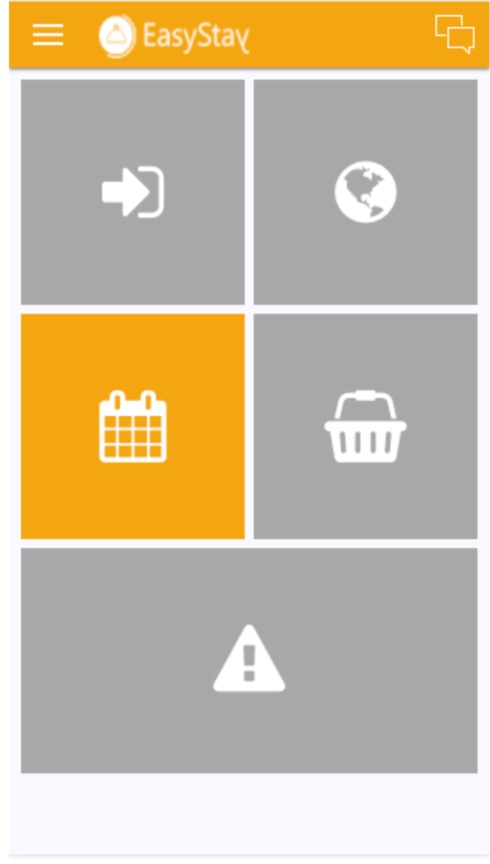


Figura 4.12 Icono indicando
nuevas reservas

4.5 Pedidos

4.5.1 Mostrar pedidos

4.5.1.1 Servidor

Para recibir el listado de pedidos de un hotel la *API* proporciona la ruta *"/manage/orders"* que, mediante el método *GET*, devuelve un listado de pedidos serializado utilizando el serializer destinado para ello. Los pedidos están relacionados con un "*carrito*", un objeto de tipo "*Cart*" que es el que realmente está relacionado con los productos. Debido a esto, es necesario especificar en el serializer un atributo para obtener los productos mediante esta relación. A su vez, en la respuesta se devuelve el usuario que ha relizado el pedido.

```
def index
  orders = Order.where(hotel_id: @current_hotel.id).order(created_at: :desc)
  render json: orders, each_serializer: ManageOrderSerializer
end

class ManageOrderSerializer < ActiveModel::Serializer
  attributes :id, :status_id, :amount, :products, :created_at
  has_one :customer, root: "customer"

  def products
    object.cart.products
  end

end
```

4.5.1.2 Aplicación móvil

Cuando el usuario pulsa en el icono de los pedidos en el menú principal, el controlador de pedidos hace la petición a la *API* haciendo el uso del servicio. Una vez recibida la respuesta es almacenada en la variable *orders*, sobre la que iteraremos posteriormente en la vista, mostrando la información de cada uno de los pedidos.

```
function activate(){
  LoaderService.show();
  ManageOrdersService.all().then(
    //success
    function (data){
      $log.debug(data);
      $scope.orders = data;
      LoaderService.hide();
    },
    //error
    function(error){
      $log.debug(error);
```

```
      LoaderService.hide();
    }
  );
}
```

```
function all(hotel_id) {
  var deferred = $q.defer();
  var _url = APPCONFIG.api.url + 'manage/orders';

  $http({
    method: 'GET',
    url: _url
  })
    .success(function(resp) {
      deferred.resolve(resp);
    })
    .error(function(error) {
      deferred.reject(error);
    });

  return deferred.promise;
}
```

A diferencia de las reservas, el estado de los pedidos se determina utilizando códigos que corresponden a cada uno de los estados:

Identificador	**Estado**	**Descripción**
1	Shopping	El usuario ha metido un producto en el carrito pero no ha procesado el pedido
2	Paid	El usuario realizado el pago de los productos solicitados
3	Checked	El pedido ha sido revisado por el hotel
4	Completed	El pedido ha sido aceptado.
5	Error	Ha habido un error en el producto

Utilizaremos estos códigos para determinar el estado del pedido y realizar los filtros en función del estado seleccionado en los botones superiores.

```
<div class="item item-divider bar bar-header"
    ng-class="{ 'paid': order.status_id == '2',
              'checked': order.status_id == '3',
              'completed' : order.status_id == '4',
              'error-order' : order.status_id == '5'}" >

  <h1 ng-show="order.status_id == '2'" class="title">
                {{'status.paid' | translate}}
    </h1>
```

```
<h1 ng-show="order.status_id == '3'" class="title">
                {{'status.checked' | translate}}
  </h1>

<h1 ng-show="order.status_id == '4'" class="title">
                {{'status.completed' | translate}}
  </h1>

<h1 ng-show="order.status_id == '5'" class="title">
                {{'status.error' | translate}}
  </h1>

function filterByStatus(item) {
 return item.status_id === $scope.filter.status_id || $scope.filter.status_id === 0 || (item.status_id === 3 &&
$scope.filter.status_id === 2)
}
```

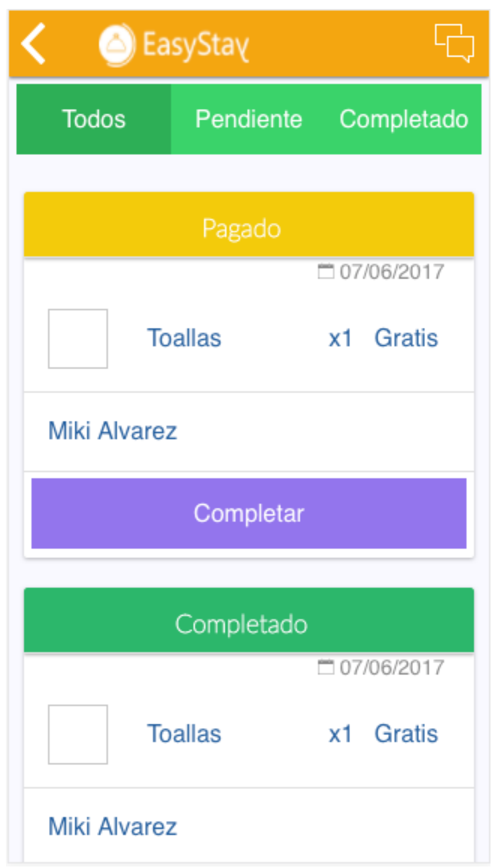


Figura 4.13 Listado de todos los pedidos

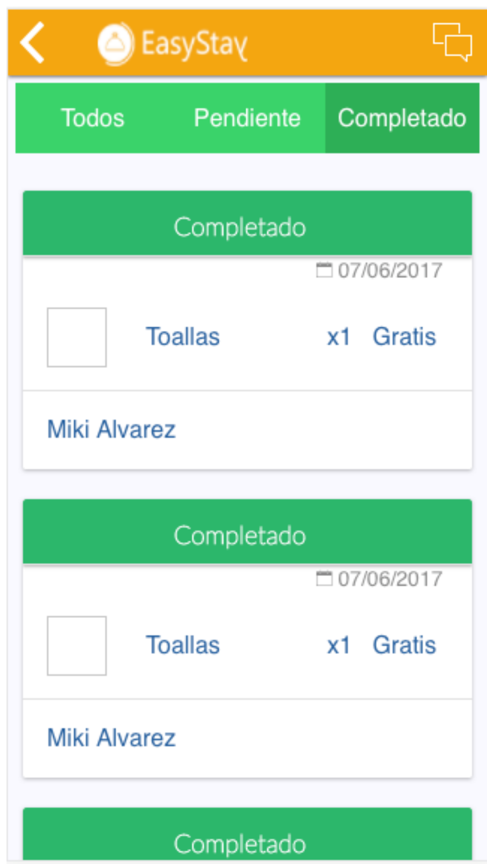


Figura 4.14 Pedidos completados usando el filtro

4.5.2 Aceptar pedidos

4.5.2.1 Servidor

Para completar un pedido mediante la *API* debe hacer una petición *PUT* a la *URI "manager/orders/:id"* donde id es el identificador del pedido a modificar. Una vez recibida la petición, se busca en la base de datos el pedido y se modifica el estado del mismo asignando un 4 al campo *status_id* que, como hemos comentado anteriormente, es el identificador del estado que indica que un producto ha sido completado. Una vez que se ha guardado satisfactoriamente el cambio, se envían las notificaciones de *Pusher* y *PushWoosh* y se devuelve una respuesta satisfactoria. En caso de fallo, se devuelve una respuesta errónea.

```
def update
  @order = Order.find_by(id: params[:id])

  @order.status_id = 4
  if @order.save

    #Send pusher notification
    pusher = PusherNotifications.new
    pusher.instant_order_notification @order

    new_notification = SendStateOrder.new
    new_notification.send_state_order(@order.access.customer)

    render json: { resp: 'action successfully' }
  else
    render json: { error: 'Error' }
  end

end
```

4.5.2.2 Aplicación móvil

Si la variable *status_id* del pedido es 2 o 3, la interfaz muestra un botón para completar el pedido. Este botón tiene asociada una función mediante la directiva *ng-click* que se ejecuta una vez pulsado el botón. Esta función muestra una ventana en la que se solicita la confirmación del usuario para realizar la acción. Si es aceptada, el controlador llama al servicio encargado de realizar la petición al servidor indicando el identificador del producto que, si es satisfactoria, recargará la vista mostrando el producto con su nuevo estado. En caso de que la petición falle, el sistema mostrará una ventana de alerta indicando del error.

```
function acceptOrder(id) {
  var confirmPopup = $ionicPopup.confirm({
    title: $translate.instant('confirmation'),
    template: $translate.instant('manage.order_confirmation'),
    cancelText: $translate.instant('answers.no'),
    okText: $translate.instant('answers.yes')
  });

  confirmPopup.then(function(res) {
    if (res) {

      LoaderService.show();
      ManageOrdersService.accept_order(id).then(
        function (data) {
          LoaderService.hide();
          $state.go($state.current, {}, {reload: true});
        },
        function (error) {
          $ionicPopup.alert({
            title: $translate.instant('alertService.titleError'),
            template: $translate.instant('alertService.templateTryAgain')
          });
        }
      );

    }
  });
}

function accept_order(id){
  var deferred = $q.defer();
  var _url = APPCONFIG.api.url + '/manage/orders/' + id;

  $http({
    method: 'PUT',
    url: _url,
    data: {state: 'confirmed'}
  })
    .success(function(resp) {
      deferred.resolve(resp);
    })
    .error(function(error) {
      deferred.reject(error);
    });

  return deferred.promise;
}

<div ng-if="order.status_id == '2' || order.status_id == '3'" class="row">
  <button ng-click="acceptOrder(order.id)">
    {{'manage.complete_text' | translate }}
  </button>
</div>
```

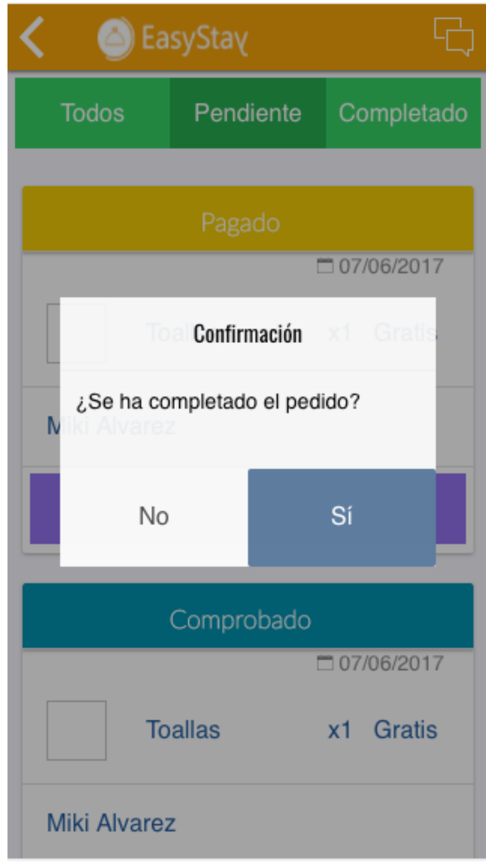


Figura 4.15 Venta emergente para completar un pedido

4.5.3 Notificar de un nuevo pedido

Como en casos anteriores, el icono de los pedidos situado en el menú principal cambiará su color de fondo una vez recibido un pedido nuevo. En este caso, recibiremos *“new_instant_product”,* que es el evento que lanza el servidor cuando recibe un nuevo pedido por parte de un cliente, mientras que la función asociada asignará *true* a la variable *orders_active.*

```
channel.bind('new_instant_product', newOrder);

function newOrder(data){
  $rootScope.orders_active = true;
}
```

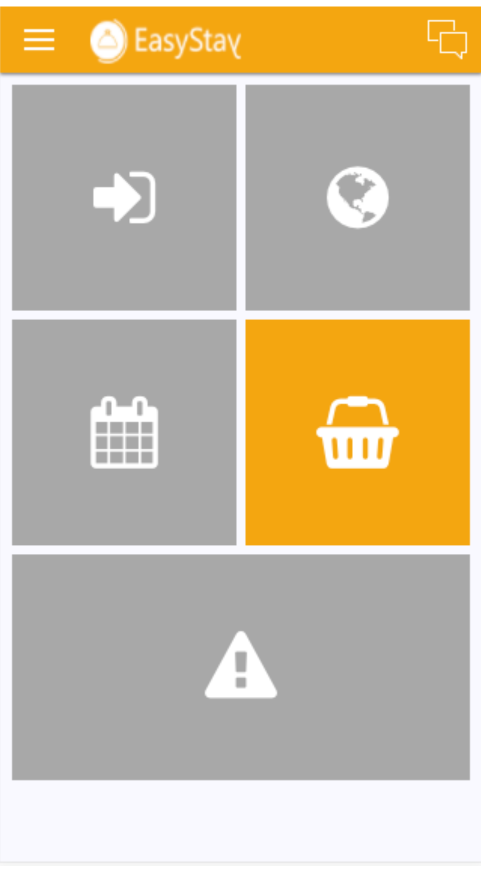


Figura 4.16 Icono notificando de un nuevo pedido

4.6 Chats

4.6.1 Mostrar listado de chats

4.6.1.1 Servidor

Para recibir el listado de chats la *API* ofrece la ruta *"/manage/chats"* que, mediante el método *GET*, devuelve un listado de chats serializados. En este caso, el *serializer* contiene varios atributos que utilizaremos para añadir detalles estéticos en la vista de la aplicación, como el último mensaje o el número de mensajes que tiene ese chat sin leer.

```
def index
  chats = Chat.active_chats @current_hotel.id
  render json: chats, each_serializer: ManageSimpleChatSerializer
end

class ManageSimpleChatSerializer < ActiveModel::Serializer
  attributes :id, :customer_name, :end_date, :last_message, :unread_messages

  def customer_name
    object.customer.full_name
  end

  def end_date
    object.access.end_date
  end

  def last_message
    object.messages.reverse.first
  end

  def unread_messages
    object.messages.where(readed: false, user_id: object.access.customer.id).count
  end

end
```

4.6.1.2 Aplicación Móvil

Para mostrar el listado de chats el usuario debe pulsar en el icono situado en la esquina derecha de la barra de navegación. Una vez pulsado, el controlador de los chats realizará una petición al servidor haciendo uso del servicio asociado que, tras una respuesta satisfactoria, almacenará la respuesta en la variable chats y cargará la vista.

La vista itera sobre cada uno de los chats y crea una componente con la información básica del chat: nombre del usuario, último mensaje, numero de mensajes sin leer, etc.

```
function activate() {

  LoaderService.show();

  ManageChatsService.all().then(
    //success
    function (data) {
      $log.debug(data);

      $scope.chats = data;
      LoaderService.hide();
    },
    // error
    function (error) {
      console.log(error);
      LoaderService.hide();
    }
  );

}

function all() {
  var deferred = $q.defer();
  var _url = APPCONFIG.api.url + '/manage/chats';

  $http({
    method: 'GET',
    url: _url
  })
    .success(function(resp) {
      deferred.resolve(resp);
    })
    .error(function(error) {
      deferred.reject(error);
    });

  return deferred.promise;
}
```

El usuario también tiene la posibilidad de filtrar los chats mediante el uso del cajón de búsqueda situado en la parte superior. Este cajón funciona de forma similar a los filtros usados en casos anteriores. Durante el proceso de iteración sobre la variable chats hemos añadido el componente *filter* asociado a la función *filterByName* del controlador de chats. Cuando el usuario empieza a escribir en el cajón de búsqueda almacena la cadena de caracteres en la variable *filter.customer_name*, utilizada por la función *filterByName* para determinar los objetos que cumplen la condición del filtro.

```
function filterByName(item) {
  return item.customer_name.toLowerCase().match($scope.filter.customer_name.toLowerCase()) ||
$scope.filter.customer_name === ''
}

<ion-item
  ng-repeat="chat in chats | orderBy: '-last_message.created_at' | filter: filterByName">


<input type="search" ng-model="filter.customer_name" placeholder="{{'manage.search_customer' | translate}}">

<a class="clear_button" ng-click="filter.customer_name=''">
  <span class="ion-android-close"></span>
</a>
```

Figura 4.17 Icono de chats mostrando
una conversación sin leer

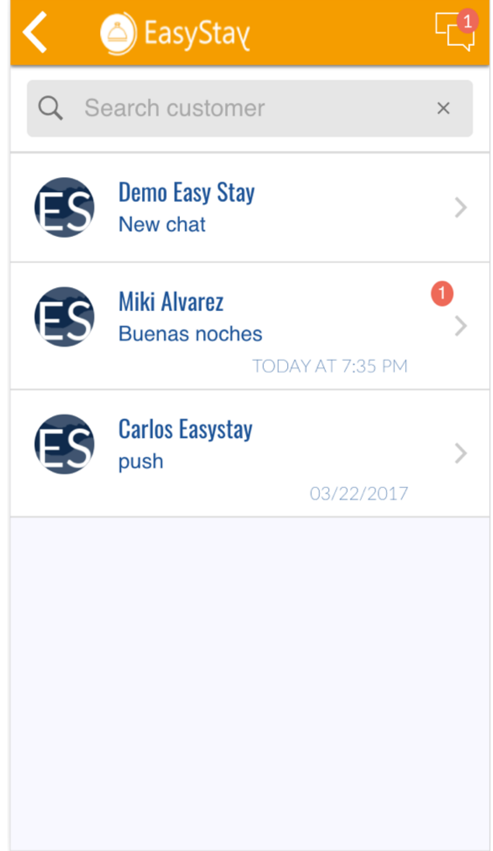


Figura 4.18 Listado de chats

4.6.2 Enviar y recibir mensajes

4.6.2.1 Servidor

Cuando un usuario escribe un mensaje, este se asociado a un chat y a un usuario. La *API* ofrece una ruta para la creación de un mensaje en un nuevo chat. La API acepta peticiones a la *URI “manage/chats/:id/messages”* mediante el método *POST* donde el id indica el chat al que pertenece el mensaje, el cual se envía en la cabecera de la petición. Una vez recibida la petición y procesada satisfactoriamente, se envían dos eventos de *Pusher* distintos. El primero de ellos manda el evento *“new_chat”* y es importante porque permitirá que la aplicación móvil se registre al canal del chat por el que se mandarán y enviarán mensajes. El segundo evento es *“new_chat_message”,* donde se añade el mensaje y dos variables que indican si el mensaje ha sido enviado por un usuario o por un trabajador del hotel. Estas variables serán utilizadas por la aplicación móvil para determinar el autor del mensaje una vez que es recibido.

La API también especifica una ruta para indicar que un mensaje ha sido leído mediante el método PUT, cuya *URI* es *“/manage/chats/:chat_id/messages/:id”*. Una vez recibida esta petición, asigna el valor true a la variable *read* del mensaje y envía el evento de *Pusher “read_chat_message”*.

```
def create
  @chat = Chat.find(params[:chat_id])
  @message = @chat.messages.build(message_params)
  @message.user_id = @current_user.id
  @message.readed = false
  @message.content_manager_id = @current_user.id
  @message.hotel_id = @current_user.hotel_id

  if @message.save
    # Send Notification to channel
    pusher = PusherNotifications.new
    pusher.new_chat @chat
    pusher.new_chat_message_from_api @message, false, true

    new_notification = SendNewMessage.new
    new_notification.send_new_message(current_user.hotel, @message)

    render json: { resp: 'action successfully' }
  else
    render json: { error: t('api.chat_message_fail') }
  end
end

def new_chat(chat)
  @pusher_client.trigger(chat.hotel.channel, 'new_chat', {
      channel: chat.channel
  })
end
```

```
def new_chat_message_from_api(message, user = true, content_manager = false)
  m = message.as_json
  m.merge!({'is_mine' => user, 'is_content_manager' => content_manager})
  @pusher_client.trigger(message.chat.channel, 'new_chat_message', {
      chat: message.chat,
      message: m
  })
end
```

4.6.2.2 Aplicación móvil

Cuando el usuario está dentro de una conversación, tiene la posibilidad de enviar un mensaje usando el campo de texto para tal fin. Una vez pulsado el icono de envío, el controlador y el servicio realizan una petición a la *API* en la que especifica la id del chat y añaden el mensaje a la cabecera. Como hemos explicado anteriormente, una vez procesada la petición el servidor manda dos eventos, *"new_chat"* y *"new_chat_message"*.

El primero, *"new_chat"* es enviado por el canal propio del hotel y contiene el canal por el que se envían y reciben los mensajes al que la aplicación debe subscribirse.

El segundo evento, *"new_chat_message"* es enviado a través del canal al que la aplicación se ha subscrito anteriormente. Cuando se recibe este evento, la función asociada utiliza una librería de *AngularJS* para notificar la recepción de un nuevo mensaje. Esta notificación es recibida por el controlador de los chat y ejecuta la función *reciveMessage*. Esta función comprueba si el usuario tiene abierta el chat, en cuyo caso añade el mensaje recibido a la variable chats para que este sea mostrado por la vista y manda una petición al servidor indicando que el mensaje ha sido leído. Esta petición envía el evento *"read_chat_message",* evento que es capturado por la aplicación móvil y cuya función enlazada actualiza la variable *read* a true del mensaje que corresponda. Esta variable es utilizada por la vista para mostrar si un mensaje ha sido leído o no.

```
//Código HTML del botón con la función asociada al ser pulsado
<button ng-click="sendMessage()"></button>

//Función del servicio que realiza la petición
function send(chatId, body) {
  var deferred = $q.defer();
  var _url = APPCONFIG.api.url + 'manage/chats/' + chatId + '/messages';

  var data = {
    message: {
      body: body
    }
  };

  $http({
    method: 'POST',
    url: _url,
    data: data
  })
    .success(function(resp) {
      deferred.resolve(resp);
    })
    .error(function(error) {
      deferred.reject(error);
    });
  return deferred.promise;
}
```

```
//Subscribción de los eventos de Pusher
function newChat(data) {
    channel.bind('new_chat_message', newChatMessage);
    channel.bind('read_chat_message', readChatMessage);
}

function newChatMessage(data) {
  $rootScope.$broadcast('pusher_events: new_chat_message', data);
}

function readChatMessage(data) {
  $rootScope.$broadcast('pusher_events: read_chat_message', data);
}

$scope.$on('pusher_events: new_chat_message', reciveMessage);
$scope.$on('pusher_events: read_chat_message', reciveReadMessage);

function reciveMessage(event, data) {
  if(data.chat.id === $scope.chat.id) {
    $scope.chat.messages.push({
      body: data.message.body,
      created_at: data.message.created_at,
      is_content_manager: data.message.is_content_manager,
      id: data.message.id
    });
    $timeout(function() {
      $ionicScrollDelegate.$getByHandle('chatScroll').scrollBottom(true);
    }, 200);

    if(!data.message.is_content_manager) {
      ManageChatsService.read(data);
    }

  }
}

function reciveReadMessage(event, data) {
  if(data.chat.id === $scope.chat.id) {
    var index = _.findIndex($scope.chat.messages, {id: data.message.id});
    if(index != -1) {
      $scope.chat.messages[index].readed = true;
    }
  }
}
```

Figura 4.19 Vista del chat con el cliente

4.7 Eventos

4.7.1 Mostrar listado de eventos

4.7.1.1 Servidor

La *API* proporciona una ruta para obtener el listado de eventos de un hotel. Para obtener el listado es necesario hacer una petición a la *URI "/manager/events"* con el método *GET* que responderá con un listado de eventos serializado. En este caso, a la *serialización* hemos añadido atributos que no son propios del modelo que utilizaremos para añadir elementos en la vista de la aplicación móvil. Uno de los atributos añadidos es *user_enrolled* que contendrá el numero de usuarios que han confirmado su asistencia al evento.

```
def index
  @events = Event.where(hotel_id: @current_hotel.id).order(start_date: :desc)

  render json: @events, each_serializer: ManageEventSerializer
end

class ManageEventSerializer < ActiveModel::Serializer
  attributes :id, :hotel_id, :name, :description, :short_description, :price, :start_date, :end_date, :begin_hour,
:end_hour, :poi, :picture, :is_reserved_from_user, :enroll, :users_enrolled

    def users_enrolled
    customers = EventUser.where(event_id: object.id).pluck(:user_id)
    users = UserProfile.where(user_id: customers)
  end

end
```

4.7.1.2 Aplicación móvil

Cuando el usuario accede al módulo de eventos se realiza una petición al servidor solicitando el listado de eventos del hotel. Esta petición es realizada por el controlador y el servicio de eventos que recibe el listado y lo almacena en la variable *events* que posteriormente será utilizada para iterar y mostrar el listado en la vista.

En este caso, la vista es más sencilla que en módulos anteriores porque sólo nos interesa mostrar al usuario el nombre del evento y el número de usuarios inscritos.

```
function activate(){
  LoaderService.show();

  ManageEventsService.all().then(
    //success
    function (data) {

      $scope.events = data;
      LoaderService.hide();
    },
    // error
    function (error) {
      console.log(error);
      LoaderService.hide();
    }
  );
}


function all() {
  var deferred = $q.defer();
  var _url = APPCONFIG.api.url + 'manage/events';

  $http({
    method: 'GET',
    url: _url
  })
    .success(function(resp) {
      deferred.resolve(resp);
    })
    .error(function(error) {
      deferred.reject(error);
    });

  return deferred.promise;
}

<ion-list>
  <ion-item ng-repeat="event in events" ui-sref="app.manage_event({id: event.id})">
    {{event.name}}
    <span ng-if="event.users_enrolled.length">
      {{event.users_enrolled.length}}
    </span>
  </ion-item>
</ion-list>
```

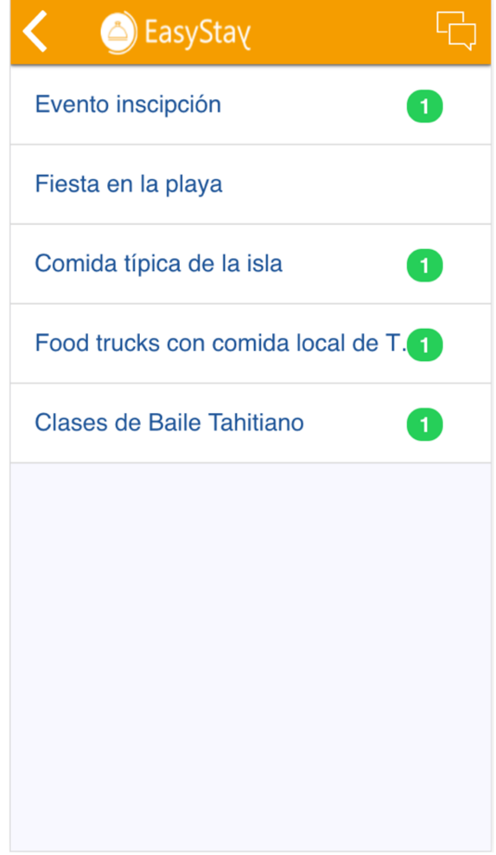


Figura 4.20 Listado de eventos

4.7.2 Mostrar información de un evento

4.7.2.1 Servidor

La *API* proporciona una ruta para obtener la información de un evento concreto. Para obtener el evento se realiza una petición a la *URI "/manager/events/:id"* donde id representa el identificador del evento. Esta petición se hace mediante el método *GET*.

Como en el caso anterior, la respuesta es serializada en *JSON* con los atributos especificados en el *serializer*.

```
def show
  @event = @current_hotel.events.find(params[:id])
  render json: @event, serializer: ManageEventSerializer
end
```

4.7.2.2 Aplicación móvil

Cuando el usuario pulsa uno de los eventos de la lista, accede a una vista donde se muestra la información detallada del evento. Para ello realiza una petición al servidor cuya respuesta almacena en la variable *events*. Esta vista muestra una foto, las fecha y el horario, así como el listado de usuarios inscritos en el evento.

```
<div class="head-event">
  <img class="full-image" ng-src="{{event.picture}}">
  <div class="title-wrapper">
    <h3>{{event.name}}</h3>
  </div>
</div>

<div class="list list-info-events">
  <div class="item item-icon-left item-event">
    <span>
      {{'event.startDate' | translate }}
      </span>
    <br>
    <span>
        {{'event.endDate' | translate }}
    </span>
  </div>

  <div class="item item-icon-left item-event">
    <span>
        {{'event.startHour' | translate}}
      </span>

    <span>
      {{'event.endHour' | translate }}
```

```
      </span>
  </div>

  <ul class="list">
    <li class="item" ng-repeat="user in event.users_enrolled">
      {{user.first_name}} {{user.last_name}}
    </li>
  </ul>
</div>
```

Figura 4.21 Información detallada de un evento

4.7.3 Notificar nueva inscripción a un evento

El sistema notificará al usuario cuando un cliente se subscriba a un evento. Cuando esto ocurre, el servido utiliza *Pusher* para mandar el evento *"new_event_subscription"*. A este evento está subscrito previamente la aplicación y ejecutará la función *newEventSubscription* cuando el evento sea recibido, momento en el cual asignará true a la variable *events_active* y el icono cambiará de color.

```
channel.bind('new_event_subscription', newEventSubscription);

function newEventSubscription(data){
  $rootScope.events_active = true;
}

<div class="col manage-button" ng-class="events_active ? 'manage-orange-background' : "" ui-sref="app.manage_events({hotel_id: hotel.id})">
  <span><i class="icon fa fa-globe"></i></span>
</div>
```

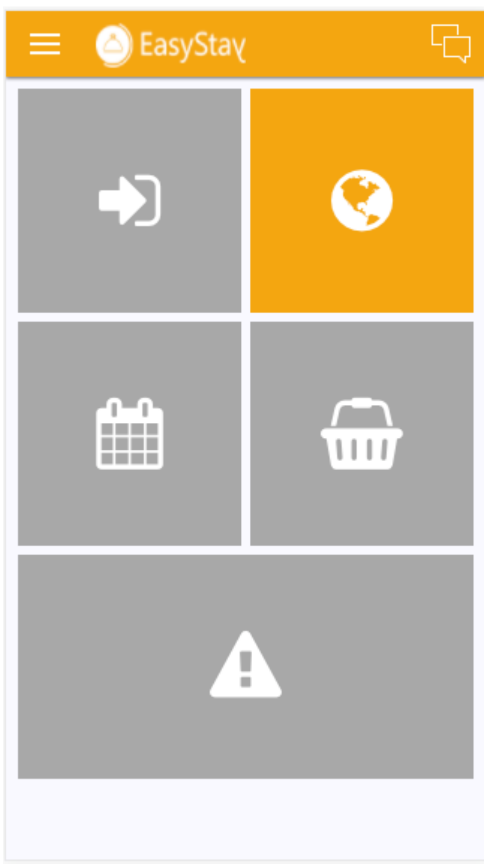


Figura 4.22 Icono notificando de una nueva subscripción

4.8 Incidencias

4.8.1 Mostrar incidencias

4.8.1.1 Servidor

Para recibir el listado de incidencias debe hacerse una petición a la *URI "/manage/incidents"* que devolverá el listado de incidencias serializadas. Cada incidencia estará asociada a un cliente, por lo que debemos especificar el atributo en el fichero de *serialización*.

```
def index
  incidents = Incident.where(hotel_id: @current_hotel.id).order(created_at: :desc)

  render json: incidents, each_serializer: ManageIncidentSerializer
end

class ManageIncidentSerializer < ActiveModel::Serializer
  attributes :id, :message, :incident_type, :state, :created_at

  has_one :user

end
```

4.8.1.2 Aplicación móvil

El usuario puede ver un listado de incidencias. Este listado se recibe como respuesta al realizar la petición descrita en el punto anterior utilizando su controlador y el servicio asociado, que será almacenada en la variable *incidents*, sobre la que iteraremos posteriormente.

El listado de incidencias mostrará la información de cada una de las incidencias, mostrando su estado, el mensaje del cliente explicando la incidencia y el *tipo*. A su vez, en caso de que la incidencia esté en los estados *"pending" o "checked"* muestra un botón que permite al usuario marcar una incidencia como solucionada.

A su vez, como en casos anteriores, permite al usuario filtrar las incidencias por estados, cuyo funcionamiento es igual a filtros explicados anteriormente.

```
function activate(){
  LoaderService.show();
  ManageIncidentsService.all().then(
    //success
    function (data){
      $log.debug(data);
      $scope.incidents = data;
      LoaderService.hide();
    },
    //error
    function(error){
      $log.debug(error);
      LoaderService.hide();
    }
  );
}

function all(hotel_id) {
  var deferred = $q.defer();
  var _url = APPCONFIG.api.url + 'manage/incidents';

  $http({
    method: 'GET',
    url: _url
  })
    .success(function(resp) {
      deferred.resolve(resp);
    })
    .error(function(error) {
      deferred.reject(error);
    });

  return deferred.promise;
}

<div class="row">
  <button ng-click="filter.state='all'">
    {{'manage.all' | translate }}
  </button>

  <button ng-click="filter.state='pending'">
    {{'manage.pending' | translate }}
  </button>

  <button ng-click="filter.state='fixed'">
    {{'manage.fixed' | translate }}
  </button>
</div>

<div ng-repeat="incident in incidents | filter: filterByState">
  <div class="item item-divider bar bar-header"
       ng-class="{ 'paid': incident.state == 'pending',
                   'checked': incident.state == 'checked',
                   'completed' : incident.state == 'fixed'}" >

    <h1 ng-show="incident.state == 'pending'" class="title">
        {{'manage.pending' | translate}}
      </h1>

    <h1 ng-show="incident.state == 'checked'" class="title">
        {{'manage.checked' | translate}}
      </h1>

    <h1 ng-show="incident.state == 'fixed'" class="title">
        {{'manage.fixed' | translate}}
      </h1>
```

```
  </div>

  <div class="item item-text-wrap">
    {{ incident.user.profile.first_name}} {{ incident.user.profile.last_name}}
  </div>

  <div ng-if="incident.message" class="item item-text-wrap">
    <i class="icon ion-ios-chatbubble-outline">{{incident.message}}</i>
  </div>

  <div ng-if="incident.state == 'pending' || incident.state == 'checked'">
    <button ng-click="completeIncident(incident.id)">
      {{'manage.fixed_button' | translate}}
    </button>
  </div>
</div>

function filterByState(item) {
  return item.state === $scope.filter.state || $scope.filter.state === 'all' || (item.state === 'checked' &&
$scope.filter.state === 'pending')
}
```

Figura 4.23 Listado de incidencias

4.8.2 Solucionar incidencias

4.8.2.1 Servidor

La ruta que ofrece la *API* para modificar una incidencia es *"/manager/incidents/:id",* que buscará en la base de datos aquella cuyo identificador recibe por parámetros y actualizará su estado con el valor *"fixed"*. Una vez almacenado, envía una notificación *push* al usuario y devuelve una respuesta satisfactoria.

```
def update
  @incident = Incident.find_by(id: params[:id])
  if @incident.state =='checked' || @incident.state == 'pending'
    @incident.state = 'fixed'
    if @incident.save

      unless @incident.user.get_device_token(@incident.hotel.chain.id).blank?
        new_notification = SendStateIncident.new
        new_notification.send_state_incident(@incident.user)
      end

      render json: { resp: 'action successfully' }
    else
      render json: { error: 'Error' }
    end
  end
end
```

4.8.2.2 Aplicación móvil

Cuando un usuario quiere marcar una incidencia como solucionada pulsa el botón que aparece junto a la incidencia. Este botón está asociado con la función *completeIncident* del controlador que hará la llamada a la *API* haciendo uso del servicio. En caso de recibir una respuesta satisfactoria la aplicación recargará el listado de incidencias.

```
<div ng-if="incident.state == 'pending' || incident.state == 'checked'">
  <button ng-click="completeIncident(incident.id)">
    {{'manage.fixed_button' | translate}}
  </button>
</div>

function completeIncident(id) {
  var confirmPopup = $ionicPopup.confirm({});

  confirmPopup.then(function(res) {
    if (res) {
      LoaderService.show();
      ManageIncidentsService.complete_incident(id).then(
        function (data) {
          LoaderService.hide();
          $state.go($state.current, {}, {reload: true});
        },
        function (error) {
```

```
        $ionicPopup.alert({});
      }
    );

  }
 });
}
function complete_incident(id){
 var deferred = $q.defer();
 var _url = APPCONFIG.api.url + '/manage/incidents/' + id;

 $http({
  method: 'PUT',
  url: _url
 })
  .success(function(resp) {
    deferred.resolve(resp);
  })
  .error(function(error) {
    deferred.reject(error);
  });

 return deferred.promise;
}
```

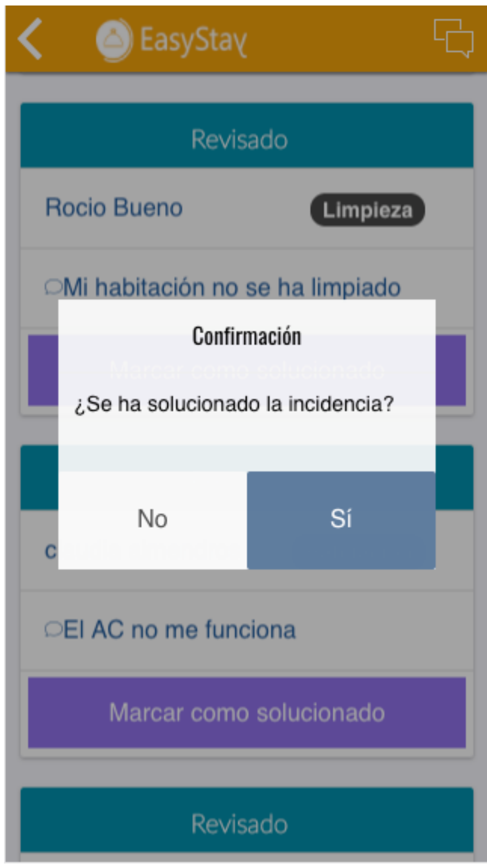


Figura 4.24 Venta emergente para solucionar una incidencia

4.8.3 Notificar nueva incidencia

Cuando un usuario envía una incidencia el servidor lanza el evento *"new_incident"* a través del canal del hotel. La aplicación móvil, previamente subscrita a este evento y enlazado a la función *newIncident*, captura el evento y cambiará a naranja el fondo del icono de incidencias, indicando así que se ha recibido una nueva incidencia. Para ello hará uso de la variable *incidents_active* que cambiará su valor a true una vez recibido el evento.

```
channel.bind('new_incident', newIncident);

function newIncident(data){
  $rootScope.incidents_active = true;
}

<div ng-class="incidents_active ? 'manage-orange-background' : ''">
  <span><i class="icon fa fa-exclamation-triangle"></i></span>
</div>
```

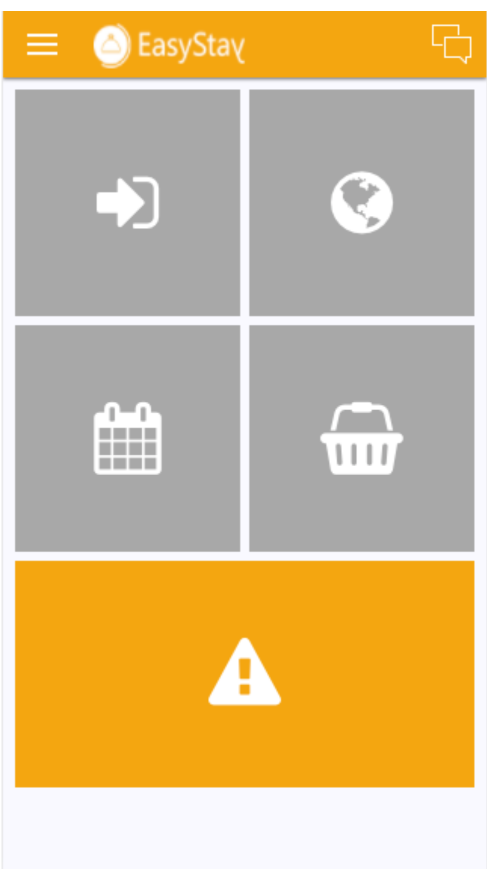


Figura 4.25 Icono notificando de una nueva incidencia

Capítulo 5

Conclusiones

5.1 Valoraciones finales

En este documento se ha abordado todo el proceso llevado a cabo para adaptar dos plataformas distintas y dotarlas de funcionalidades para las que no fueron desarrolladas inicialmente.

Este proyecto parte de un desarrollo ya realizado, por lo que la metodología es distinta a la mayoría de trabajos de fin de grado. La principal complicación de este tipo de desarrollos es estudiar y comprender el estado actual del sistema para poder modificarlo y adaptarlo a las nuevas funcionalidades. Estas modificaciones no pueden comprometer el comportamiento actual del sistema.

Todo el proceso de documentación y desarrollo queda recogido en este documento donde se explican los módulos que hemos desarrollado y las modificaciones que hemos tenido que aportar al código existente. En cada uno de los módulos se ha realizado tanto el desarrollo de la *API* en la parte del servidor como la adaptación del mismo a una aplicación móvil utilizando tecnologías híbridas, cada vez más presentes en entornos de desarrollo software y cuya importancia aumentará sin duda en los próximos años.

Este TFG me ha permitido trabajar en el proyecto de una empresa cuyo producto estaba ya en producción, siendo utilizado por usuarios reales. Esto me ha permitido aprender como funciona realmente el desarrollo software en un ambiente real donde cobra especial importancia la capacidad de entender y trabajar sobre un proyecto ya desarrollado.

Todo este proceso ha aumentado mis conocimientos en áreas de desarrollo que desconocía, desde establecer una comunicación entre dos aplicaciones utilizando una *API* hasta el desarrollo de aplicaciones móviles, sin olvidar que, al haber trabajado simultáneamente con dos plataformas distintas, me ha permitido adquirir conocimientos en lenguajes y *frameworks* distintos, véase *Ruby On Rails, Ionic Framework, HTML, CSS, AngularJS o Pusher*;

conocimientos que sin duda han hecho que adquiera un perfil más completo como desarrollador.

Además de los conocimientos técnicos, trabajar en el entorno real de una empresa pequeña te permite ser partícipe del día a día de la misma, participando en la toma de decisiones y aportando ideas que ayuden a la mejora del producto. Esto ha hecho que, a parte de los conocimientos técnicos, también haya adquirido conocimientos del funcionamiento interno de una empresa.

5.2 Trabajo futuro

A pesar de que la aplicación es funcional, solo ha sido adaptada para su uso en hoteles, por lo que sería necesario realizar modificaciones para que los gestores de apartamentos puedan hacer uso del módulo de gestión.

Otro de los cambios es adaptar la aplicación a nuevas versiones de *Ionic Framework* que aportarían mejoras estéticas y de rendimiento. Estas mejoras son notables en el aspecto estético. Como hemos comentado en el documento, es importante que una aplicación híbrida consiga adaptarse a la plataforma sobre la que se ejecuta copiando los estilos y componentes del sistema, mejoras que el *framework* ha adquirido en sus nuevas versiones.

Apéndices

Apéndice A

Instalación de los entornos

A.1 Aplicación web

Para poder desplegar la aplicación web es necesario tener instalado *Ruby On Rail*s. A continuación explicaremos el proceso necesario para instalar el entorno en el sistema operativo *Mac OS X*, sistema elegido para el desarrollo.

Mac OS X ya dispone de una versión de *Ruby* instalada por defecto. Sin embargo nuestro proyecto está desarrollado bajo una versión determinada, más concretamente la 2.1.4, por lo que es necesario instalar un gestor de versiones que nos permita especificar qué número de compilación queremos usar en nuestro proyecto. Para ello usaremos *RVM* (*Ruby Version Manager*) utilizando el siguiente comando:

```
\curl –sSl https://get.rvm.io | bash –s stable --ruby
```

Una vez terminado, procedemos a instalar la versión 2.1.4 y establecerla como predeterminada.

```
rvm install 2.1.4
rvm use 2.1.4 --default
```

Para instalar *Ruby On Rails* hacemos uso del sistema de gemas de *Ruby.*

```
gem install Rails
```

Una vez tenemos Ruby y Ruby On Rails instalamos accedemos a la carpeta del proyecto e instalamos las gemas y dependencias. Una vez completado el proceso, creamos la base de datos.

```
bundle install
rake db:create
rake db:migrate
```

Finalmente, iniciamos el servidor y accedemos a la aplicación.

```
rails s
http://localhost:3000
```

A.2 Aplicación móvil

La instalación de *Ionic Framework* se hace mediante *npm* (*Neutered Paranoid Meerkat*), un gestor de paquetes para desarrollos basados en JavaScript, como el caso de *Ionic*. Para instalar *npm* lo más sencillo es instalar *Node.js* descargándolo desde su página web, que a su vez incluye una versión de *npm.*

Una vez instalado *npm* en nuestro sistema, podemos instalar *Ionic* con el siguiente comando:

```
npm install –g ionic
```

Tras esto, podemos acceder a la carpeta de nuestro proyecto e iniciarlo accediendo en el navegador al puerto 8100:

```
ionic serve
http://localhost:8100
```

Bibliografía

[1] Moreno, S. (2013). *Ruby on Rails: Desarrollo práctico de aplicaciones web*. San Fernando de Henares, Madrid: RC Libros.

[2] Ionic. (2013-2017). *Ionic Framework.* Obtenido de Ionic: http://ionicframework.com/

[3] Papa, J. (2014-2016). *Angular Style Guide.* Obtenido de GitHub: https://github.com/johnpapa/angular-styleguide

[4] Simpsons, K. (2015). *You Don't Know JS: Async & Performance*. Sebastopol, CA: OReilly

[5] Google. (2010-2017). *AngularJS*. Obtenido de AngularJS: https://angularjs.org/

[6] Pusher. (2017). *Pusher*. Obtenido de Pusher: https://pusher.com/