ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA INFORMÁTICA

Sistema para la gestión del reconocimiento de equivalencias entre asignaturas: aplicación móvil

System for managing the approval of equivalence between subjects: mobile application

Autor:
Nicolás Vargas Ortega
nvortega92@gmail.com
Universidad de Málaga

Tutor:
José del Campo Ávila
jcampo@lcc.uma.es
Universidad de Málaga

Co-tutor:
Mónica Trella López
trella@lcc.uma.es
Universidad de Málaga

Tutor coordinador:
José del Campo Ávila

UNIVERSIDAD DE MÁLAGA
Málaga, Diciembre 2016

Fecha de defensa:
El Secretario del Tribunal

Resumen

Las tareas administrativas referentes a los planes de movilidad estudiantil (p. ej. Erasmus, SICUE y Única), son un trabajo arduo y costoso que actualmente no se apoya en ninguna herramienta software. La aplicación móvil del proyecto "Willyfog" intenta subsanar los problemas referentes a, las búsquedas de equivalencias por parte de los estudiantes, la manera en la que reciben las notificaciones y las consultas del estado de sus peticiones. El conjunto de servicios de Willyfog, está formado por varias aplicaciones que conforman el proyecto completo, éstas son: a) Willyfog-API, la interfaz de aplicación que se comunica con la base de datos y realiza toda la lógica de negocio de las aplicaciones a las que sirve; b) Willyfog-OpenId, que gestiona la autenticación de usuarios y maneja las sesiones; c) Willyfog-móvil, aplicación móvil tratada en este TFG; d) Willyfog-Web, la aplicación web que se encarga de poner a disposición los formularios a rellenar por los estudiantes y las tablas de equivalencias aceptadas por los profesores reconocedores y coordinadores de los centros, este último apartado se desarrolla en otro TFG que se ha realizado de forma coordinada con este. Ofrecemos una arquitectura orientada a servicios la cual sirva como apoyo para futuras implementaciones en otras plataformas y tecnologías.

Palabras clave

Erasmus, SICUE, Única, equivalencias, aplicación móvil, Android, OAuth2, OpenId Connect, MySQL, API, PHP, Java, SBT, Maven, Play framework, Slim framework

Abstract

The administrative tasks concerning student mobility plans (e.g. Erasmus, SICUE and Única), are an intensive and expensive job that currently are not supported by any kind of software tool. The "Willyfog" mobile application attend to solve the problems of the equivalences searches by the students, the way that they can receive notifications and the status queries about their equivalence requests. The "Willyfog" ecosystem, not only consists of a mobile application, if not, are formed of various applications that make up the whole project, they are: a) Willyfog-API, an application interface that connect with the database and performs all the business logic of the applications it serves; b) Willyfog-OpenId, that manages the user authentication and sessions; c) Willyfog-mobile, mobile application exposed here; d) Willyfog-Web, the web application that is responsible of the forms to be filled by students and the equivalence tables accepted by the recognizers professors and centre coordinators, this last point is developed in another TFG that has been done in a coordinated way with this. We offer a service-oriented architecture which serves as support for future implementations on other platforms and technologies.

Keywords

Erasmus, SICUE, Única, equivalences, mobile application, Android, OAuth2, OpenId Connect, MySQL, API, PHP, Java, SBT, Maven, Play framework, Slim framework

Índice

Resumen 1
Palabras clave 1

Abstract 2
Keywords 2

1. Introducción 5
1.1 Explicación del problema 5
1.2 Estudiantes, profesores y coordinadores 6
1.3 Flujo de trabajo para la petición de equivalencias 7
1.4 Objetivos 8
1.5 Estado del arte 9

2. Requisitos 11
2.1 Requisitos API 11
2.1.1 Usuarios 11
2.1.2 Peticiones de equivalencia 12
2.1.3 Asignaturas 12
2.1.4 Universidades y centros 13
2.1.5 Países y ciudades 13
2.1.6 Documentación 13
2.1.7 Notificaciones 13
2.2 Requisitos aplicación móvil 14
2.2.1 Perfil de usuario 14
2.2.2 Roles de la aplicación móvil 14
2.2.3 Búsqueda de equivalencias 14
2.2.4 Índice de peticiones 15
2.2.5 Sistema de notificaciones 15

3. Diseño e Implementación 17
3.1 Arquitectura 17
3.1.1 Base de datos 18
3.1.2 API RESTful 23
3.1.3 Aplicación móvil 24
3.1.4 OpenID 24
3.1.5 Integración con servicio gravatar 24
3.2 Tecnologías usadas 24
3.2.1 Buscando las tecnologías 25
3.3 Dependencias del proyecto 26
3.3.1 Dependencias de la API 26
3.3.2 Dependencias de la aplicación móvil 28
3.3.3 Dependencias del servidor OpenID 28
3.4. Diseño API 28

3.4.1 ¿Por qué REST y no SOAP? 29
3.4.2 Versionado 29
3.4.3 Rutas 29
3.4.4 Protección de la API con OAuth2 30
3.5. Diseño móvil 31
3.5.1 MVC 31
3.5.2 Inicio de sesión OpenID 32
3.5.3 Diseño de la interfaz de usuario 32
3.5.4 Integración con la API 34
3.5.5 Integración con OpenID 34

4. Metodologías **37**
4.1. Planificación 37
4.2. Pruebas 37
4.3. Documentación 38
4.3.1 ¿Swagger o RAML? 38
4.4. Entorno de desarrollo 39
4.4.1 Vagrant 39
4.4.2 Docker 39
4.4.3 Desarrollo y Producción 40

5. Conclusión **41**

Referencias **43**

A. Anexo **45**
A.1 Manual de usuario 45
A.2 Manual de despliegue 58
1. Entorno Vagrant 58
2. willyfog-api 58
3. willyfog-openid 60
4. willyfog-web 60
5. willyfog-mobile 60
6. Dominios 61
A.3 Manual API 61

1. Introducción

En la Universidad de Málaga existen actualmente varios planes de movilidad estudiantil (Erasmus, SICUE, etc.) [1]. Cuando un estudiante solicita un plan de movilidad, necesita conocer qué asignaturas de la universidad de destino son equivalentes a las que tendría que cursar en su universidad de origen para, una vez superadas, poder recibir el reconocimiento correspondiente.

Por tanto, es el estudiante el que se encarga de recopilar la información sobre el plan de estudios de la universidad destino para transmitírselo a su tutor académico. A su vez, el tutor revisa la información y la facilita al coordinador del centro que elevará la solicitud a la comisión de reconocimiento. Aunque pueda parecer un proceso único, varía según el centro de la universidad.

En este proceso intervienen una serie de usuarios con diferentes roles cada uno. Todos estos usuarios siguen una estricta cadena de mando, que a su vez se puede extrapolar a una serie de *fases* que mantienen una prioridad. Esto significa que a lo largo del proceso, una petición puede llegar a ser rechazada antes de que llegue a manos del último eslabón. Además, durante el curso del procedimiento no se dispone de herramientas específicas que lo simplifiquen.

Son los problemas que intentamos solucionar con la creación del sistema. Los estudiantes normalmente no están al tanto del estado de sus solicitudes, los profesores no tienen por qué avisar al estudiante. Si una solicitud es rechazada el estudiante podrá verlo en las tablas de equivalencias que se publican periódicamente, si su petición no aparece se entiende que ha sido rechazada. En este caso el estudiante tendrá que volver a enviar la petición cumpliendo los requisitos especificados por el coordinador o directamente su solicitud no es válida.

Dado que el teléfono móvil es una herramienta usada ampliamente entre la comunidad estudiantil, es ideal para que el estudiante esté continuamente informado de como están sus solicitudes y pueda ser capaz de buscar entre toda la base de datos de equivalencias ya aceptadas por la comisión. Esto ahorrará mucho trabajo tanto a la parte de los estudiantes, como a la parte de los profesores.

Aunque la aplicación está pensada para cubrir las necesidades de la *Universidad de Málaga*, se ha construido todo el conglomerado de herramientas pensando en dar servicio y soporte a más de una universidad, a más planes de movilidad y a todos los dispositivos, ya sean móviles, *tablets*, aplicaciones webs o incluso aplicaciones de escritorio, posibles.

1.1 Explicación del problema

Desde el inicio de una petición hasta que es resuelta, se ven involucrados varios procesos los cuales son, sin duda, manuales y bastante costosos. Actualmente, el alumno elige una (o varias) asignaturas las cuales cree que son equivalentes a las asignaturas de su grado de

origen. Estas asignaturas tienen que corresponder en materia y número de créditos para que puedan ser equivalentes. Al tenerlas elegidas, el alumno se lo comunica a su tutor mediante un email o tarea del campus virtual, pero los estudiantes no tienen ninguna forma de mandar esta información de una manera centralizada.

Los estudiantes envían información errónea con facilidad ya que los recursos de las asignaturas extranjeras pueden estar muy dispersos, en otros idiomas que no sean el inglés o que ni siquiera existan. Puede tener varias consecuencias como por ejemplo, que la universidad cambie la URI, o que no sea posible mandar una URI.

Los estudiantes no tienen ninguna forma de estar al tanto de sus peticiones de una manera sencilla, excepto esperar a que se publiquen las tablas de equivalencias.

Los profesores, tienen una hoja de cálculo donde van apuntando a sus alumnos tutorizados, a las universidades que van y las asignaturas que han elegido. **Aquí nos encontramos la primera herramienta** que los profesores usan por motivos de agilizar el proceso. Estas hojas de cálculo van mutando conforme el proceso se va completando y hay un gran componente manual que podría automatizarse.

1.2 Estudiantes, profesores y coordinadores

En el flujo de trabajo que sigue una petición de equivalencia, podemos destacar tres roles principales:

1. Estudiante
2. Profesor (reconocedor)
3. Coordinador

Cada uno de ellos tiene definido un ámbito de trabajo dentro del sistema. El **estudiante**, por ejemplo, es el productor principal, encargado de realizar las peticiones de las asignaturas que quiera reconocer y de suministrar la información necesaria para que se pueda validar correctamente su petición.

El **profesor** de la Comisión de Convalidaciones, Adaptaciones y Equivalencias (en adelante usaremos el término "profesores reconocedores"), hace de primer filtro antes de que la petición llegue al coordinador del centro. Se encarga de pedir información al estudiante si hiciera falta, pudiendo rechazar la petición hasta que no se cumplan las condiciones, también pueden tener el rol de tutor académico (que no tiene un papel activo en la herramienta que se esta desarrollando), que simplemente asesora a los alumnos y los ayuda en el proceso. Finalmente será el que le haga llegar la petición al coordinador del centro. Este proceso dependerá del centro, por ejemplo, en el centro de Comercio y Gestión es justo al revés, es decir, el coordinador recoge directamente la petición pero en la escuela de Politécnica Superior ocurre lo explicado anteriormente, del alumno al profesor y del profesor al coordinador.

Una vez que la solicitud llega al **coordinador**, éste se encarga de consultar a la Comisión de Convalidaciones, Adaptaciones y Equivalencias, cuyos profesores (reconocedores) se encargan de recoger sus informes y de generar la tabla de equivalencias.

Aunque en todo el proceso tenemos 3 claros actores, en el sistema realmente aparece otro cuarto rol que también participa. El **administrador** es el encargado de dar de alta/baja a los coordinadores de los distintos centros de una universidad. Y el coordinador será el encargado de dar de alta a los profesores reconocedores.

1.3 Flujo de trabajo para la petición de equivalencias

Primero vamos a explicar el flujo de trabajo completo que tiene que seguir una petición, desde que hace la petición un alumno hasta que se publican las tablas de equivalencias. Veremos en que parte de éste se centra la aplicación móvil y cual es el que cubre la API.

Lo primero que ocurre cuando un alumno recibe una beca de movilidad, es la elección de asignaturas en la universidad de destino que debe cursar para que en la universidad de origen sean equivalentes, es más, lo más común es que un alumno pregunte primero por las asignaturas que pueden ser reconocidas para elegir el destino. También podrá verlas en las tablas de equivalencias publicadas. En el caso de que ya sepa el destino, lo primero que tiene que hacer el alumno es una petición en el sistema y poner como equivalentes las asignaturas que crea convenientes en la petición (punto 1 de la Figura 1).

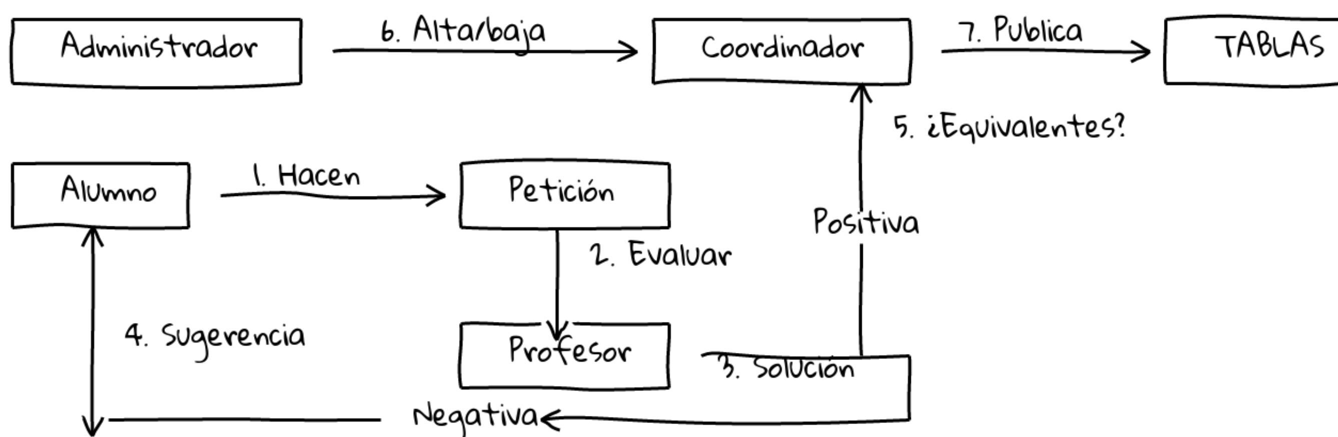

Figura 1: Flujo de trabajo de una petición

Ahora entran en juego los profesores reconocedores, que reciben la petición y la evalúan (punto 2 de la Figura 1). En este punto pueden pasar varias cosas. Si el profesor considera que la petición no cumple unos requisitos mínimos, él mismo puede gestionar la petición y dar una solución negativa, esto conlleva a una retroalimentación por parte del profesor al alumno dándole alguna sugerencia de información faltante o errónea (punto 3 y 4 de

la Figura 1). Por otro lado, si todo marcha bien, a la petición se le dará una solución positiva y será aceptada (punto 3 Figura 1).

Aunque en el proceso actual, normalmente el coordinador recoge y filtra las peticiones antes de mandarlas a la Comisión de Convalidaciones, Adaptaciones y Equivalencias, se podría proceder al envío de las peticiones directamente para su evaluación.

Al coordinador le llegarán las peticiones aceptadas las cuales tendrá que validar para realizar las tablas de equivalencias de asignaturas para la movilidad estudiantil que se publican en el centro correspondiente a éste.

1.4 Objetivos

Desarrollar una aplicación móvil que se apoye en una API la cual le de las herramientas necesarias para buscar, comprobar y estar al tanto de todo lo referente a sus solicitudes de equivalencias. Conviene destacar que cuando hablamos de la API, ésta tiene que ser capaz de dar servicio a más de un cliente. Es por esto, que nos hemos decantado por una solución RESTful.

Uno de los principales objetivos de este proyecto era aprender tecnologías nuevas. Hemos pasado por varias de ellas hasta dar con una que se nos ha hecho cómoda. Así fuimos eligiendo hasta cubrir todas las partes que componían el proyecto *Willyfog*.

Proteger la API es esencial, ya que, contendrá información personal y puede ser objeto de ataques fácilmente. Los recursos que ofrece la API, además de estar protegidos, están restringidos de manera que los diferentes roles de usuario de la aplicación sólo puedan usar lo que les corresponde. Este era un objetivo esencial que queríamos cubrir desde el principio.

Tener un entorno de desarrollo homogéneo que facilite el trabajo en equipo ha sido un punto crucial. El objetivo es tener el mismo entorno de desarrollo que el de producción. Esto facilita el despliegue de las aplicaciones, hace más fácil el mantenimiento y cualquier miembro del equipo puede tener todo funcionando en su ordenador local en menos tiempo.

Por tanto los objetivos ordenados por importancia quedarían así:

- Realizar una API para ofrecer una arquitectura orientada a servicios (SOA [15]) que resuelva los problemas de gestión derivados de los diferentes planes de estudio a la hora de reconocer asignaturas entre centros de destino y origen.
- Conocer y utilizar alguna manera de proteger la API.
- Construir una aplicación móvil que se sirva de la API.
- Homogeneizar el entorno de desarrollo y el de producción.

1.5 Estado del arte

Aunque no existe algo que dé solución de una manera directa a nuestro problema, existen varios *workarounds* que se utilizan actualmente en todo el proceso de reconocer asignaturas en los diferentes planes de movilidad. Algunos de ellos ni siquiera entran en el ámbito por el que aquí estamos, pero sirven de ejemplo para ilustrar la idea de lo que sería un sistema completo.

- **Ícaro**: La primera herramienta de la que vamos a hablar es Ícaro [10]. Una plataforma web donde los estudiantes y las empresas pueden ponerse de acuerdo y ofrecer contratos de prácticas para recién titulados o alumnos en proceso de titularse. Hemos querido mencionar esta herramienta porque plasma a la perfección la idea básica que nosotros hemos estudiado.

  Existe un problema entre los profesores, alumnos y las empresas a la hora de coordinar las prácticas de empresa. Por esta razón, la *Universidad de Almería* (autora de la plataforma) se puso manos a la obra y centró todos estos roles en una misma plataforma donde se pudieran poner de acuerdo todos. Centralizan la información, reducen el coste de tiempo y dinero, y dejan cubierta una necesidad que a todos nos beneficia.

  Algo parecido queremos ofrecer nosotros, una forma de centralizar la información, reducir costes y poner a todos los actores en común.

- **EVE**: El EVE o Espacio Virtual Erasmus [11], es un portal de la *Universidad de Málaga* donde los alumnos que se van de erasmus tienen que acceder para rellenar una tabla (entre otras muchas cosas que aparecen) con las asignaturas escogidas en la universidad de destino y que serán reconocidas en la universidad de origen.

  Esta herramienta, cubre una parte que nuestra aplicación no hace. Por tanto, podríamos decir que esta herramienta se complementaría con la nuestra que se desarrolla en este TFG.

- **Acuerdo académico**: El acuerdo académico [12], no es una herramienta software en sí misma, pero es lo que en última instancia un alumno debe rellenar para la equivalencia entre asignaturas de una universidad de destino y la de origen. Se trata de un formulario donde como cabecera hay que rellenar la facultad/escuela a la que se pertenece en la *Universidad de Málaga*, la titulación y la universidad de destino.

  Luego hay que poner los datos del alumno, y rellenar una tabla con los nombre de las asignaturas de origen a la izquierda y las asignaturas de destino a la derecha.

  Este formulario es firmado tanto por el alumno, como por el coordinador del centro y el tutor académico (figura que no aparece en nuestra arquitectura web), y el coordinador de la universidad de destino.

- **Campus virtual**: Por último, una herramienta que sirve de intercambio de información a la hora de presentar los planes de estudios de las asignaturas de destino, o de informar a los profesores reconocedores es el campus virtual de la UMA. Cuando un alumno decide las asignaturas que va a elegir, lo que haces es recoger toda la información de éstas (planes de estudio, créditos, étc) y se lo envía al profesor a través de una tarea.

  Las tareas son abiertas por el profesor reconocedor o por el coordinador, y gestionadas por ellos mismos, teniendo que estar al tanto los alumnos y los propios profesores por si reciben alguna información. Esto provoca pérdidas de información, problemas de comunicación entre las partes y no cumple con los requisitos específicos de cada tipo de movilidad estudiantil. Aún siendo una opción bastante rudimentaria, es una de las utilizadas en el centro de Comercio y Gestión de la UMA.

  Otra razón por la cual era necesario realizar una aplicación software que dé servicio a este problema de descentralización de información.

2. Requisitos

Puesto que la aplicación tiene unos requisitos generales, hemos querido separarlos tanto en los requisitos de la API como los de la aplicación móvil. Desde el inicio de las reuniones con nuestros tutores, hemos ido recabando información hasta elaborar una lista de requisitos que debía cumplir el sistema. A grosso modo hemos explicado por encima los requisitos que necesitamos cubrir y los roles que se involucran en el sistema.

2.1 Requisitos API

La API debe ser la que contiene toda la lógica de negocio de la aplicación, la interfaz que interactúa con la base de datos y da servicio a todos los clientes posibles. El propósito de desarrollar *willyfog-api* es la de poder ofrecer como servicio todas las acciones que conllevan la realización de una petición de equivalencia entre universidades.

2.1.1 Usuarios

Este módulo de la API es el encargado de realizar todas las operaciones con los usuarios del sistema. Desde listarlos, hasta poder conseguir la información asociada a un usuario, añadir/quitar que un profesor pueda reconocer asignaturas y listar las notificaciones asociadas a un usuario. La API **no es la encargada de registrar los usuarios**, para ello se utiliza un servicio externo con OpenID Connect.

Será necesario:

- Conseguir el centro al que pertenece un usuario.
- Recuperar el grado en el que esta dado de alta el usuario.
- Recuperar los datos personales (DNI, nombre, apellidos,...) del usuario.
- Buscar un usuario en la base de datos por su *access token*.
- Listar las notificaciones no leídas de un usuario.
- Asignar a un usuario con el rol de profesor las asignaturas que es capaz de reconocer.
- Quitar las asignaturas que reconoce un usuario con el rol de profesor.

Estas son los requisitos que la API cubre respecto a los usuarios, pero debemos destacar también lo que el servidor OpenID Connect ofrece en este campo. El servidor OpenID Connect debe:

- Se debe poder crear usuarios.
- Para crear un usuario se necesita un nombre, apellidos, DNI, email y contraseña.
- Las contraseñas se guardan en la base de datos con un hash MD5.
- Al crear un usuario se le debe asignar uno de los roles.

- Al crear un usuario se le asigna un grado (si es estudiante).
- Al crear un usuario se le asigna un centro al que pertenece.
- Un usuario al ser autenticado, se le asigna un token de acceso a la aplicación.
- Debe de ofrecer un formulario de login a todos los clientes.

Se puede intuir que tanto la API como el servidor OpenID se conectan a la misma base de datos, y se nutren de las mismas tablas. Hay otras formas de hacerlo pero, esto facilita bastante las cosas a la hora de desarrollar el módulo de autenticación de la aplicación y permite que sólo en un sitio se creen los usuarios. La API lee de la tabla usuarios, y OpenID escribe en la tabla.

2.1.2 Peticiones de equivalencia

Esta funcionalidad es el corazón de la API junto con los usuarios. Trata de realizar todas las funciones que una petición puede tener en el sistema, desde la creación hasta comentar en una petición o notificar las actualizaciones de la misma. Las peticiones en el sistema deben cumplir:

- En el sistema las peticiones se diferencian por un estado, las pendientes y las cerradas. Una petición si se encuentra en el estado cerrada, puede ser porque haya sido rechazada o aceptada.
- Listar las peticiones, sea cual sea su estado.
- Listar las peticiones de un usuario en concreto.
- Las peticiones se pueden rechazar sólo por profesores reconocedores o coordinadores.
- Las peticiones se pueden aceptar sólo por profesores reconocedores o coordinadores.
- Comentar en las peticiones para posibles problemas o sugerencias.
- Listar todos los comentarios de una petición.
- Se pueden crear peticiones. Sólo los alumnos pueden crear peticiones.

2.1.3 Asignaturas

La API debe de tener una ruta donde se puedan listar las asignaturas que hay en el sistema. Esta función cubre los siguientes puntos:

- Listar todas las asignaturas que existen en el sistema.
- Buscar una asignatura por su identificador.
- Buscar las asignaturas equivalentes unas con otras.

- Listar todas las asignaturas que hay de un grado en concreto.

Una de las cosas importantes que realiza la API es el poder hacer una búsqueda para encontrar las asignaturas que ya han sido reconocidas en el sistema. Así el estudiante antes de hacer una petición puede saber si las asignaturas que ya va a pedir han sido reconocidas, ahorrando así trabajo a los profesores y coordinadores.

2.1.4 Universidades y centros

Para realizar correctamente una petición de reconocimiento, el sistema tiene que poder ofrecer un listado completo de las universidades dadas de alta en el sistema y a qué centro pertenece. La función de universidades y centros cumple los siguientes requisitos:

- Listar todas las universidades del sistema.
- Mostrar todos los centros de una universidad.
- Ver todos los grados pertenecientes a un centro.
- Sacar todos los profesores reconocedores de un centro.

2.1.5 Países y ciudades

Puesto que una universidad pertenece a una ciudad en concreto y necesita por ende un país, se han creado unas cuantas funciones que nos permiten listar las ciudades y los países que hay en el sistema. Este requisito es necesario para poder hacer las peticiones.

2.1.6 Documentación

Uno de los requisitos que la API cumple es el de la documentación online. Para ello se necesita una ruta en la API que ofrezca a todo el usuario que lo solicite la documentación de manera online para que se pueda consultar desde cualquier parte.

2.1.7 Notificaciones

Las notificaciones en el sistema las maneja la API. Teniendo como requisitos principales:

- Se debe poder listar todas notificaciones de un usuario.
- Cuando un usuario ve sus notificaciones, éstas se marcan como vistas.
- Las notificaciones sólo se pueden ver si se recarga la aplicación cliente (web o móvil).

2.2 Requisitos aplicación móvil

Dado que trabajar desde el móvil puede resultar incómodo, la idea principal de la aplicación móvil es la de servir de punto informativo para los estudiantes. Un estudiante puede recibir en su móvil notificaciones de cuándo una de sus peticiones se ha actualizado, saber en qué estado se encuentran sus peticiones y poder buscar rápidamente en el sistema por equivalencias ya resueltas.

2.2.1 Perfil de usuario

Una vez que un usuario se ha autenticado contra el servicio de OpenID y entra en la aplicación móvil, lo primero que se encuentra es su perfil. Éste cumple las siguientes exigencias que se pidieron:

- Ver el nombre completo de un estudiantes.
- Saber la universidad a la que pertenece un estudiantes.
- Mostrar el grado en el un estudiante está dado de alta.
- Cambiar la foto de perfil de un usuario.

El último requisito que el perfil de usuario cumple en el móvil se hace a través de un servicio externo a la API, llamado Gravatar [23]. Ya que el propósito de la API y de la aplicación móvil no es el de tratar con imágenes, hemos decidido optar por una solución de terceros para subsanar esta carencia.

2.2.2 Roles de la aplicación móvil

Uno de los requisitos más explícitos y exigentes de la aplicación móvil es el de que **sólo sea accesible para los estudiantes**. Ya que ninguna de las funcionalidades que cubre la aplicación móvil es la de trabajar con peticiones, si no, meramente informativa, el acceso se ha restringido a los estudiantes.

El servidor OpenID autenticará a todos los usuarios que existan en el sistema, pero será la aplicación cliente (en esta caso móvil) la que se encargue de redireccionar a los usuarios que no tengan este rol hacia una vista vacía donde se le indique que no pueden acceder a la aplicación.

2.2.3 Búsqueda de equivalencias

El principal pilar de la aplicación móvil es el de poder ofrecer al usuario una búsqueda de las asignaturas ya reconocidas en el sistema. Para ello se definieron los siguientes requisitos:

- Se debe poder buscar cualquier asignatura en el sistema ya reconocida.
- La búsqueda se hace con cualquier tamaño de palabra, es decir, con cualquier subcadena que empiece por la palabra que se quiera buscar.
- Sólo se debe mostrar el nombre de la asignatura de origen y la asignatura de destino.

2.2.4 Índice de peticiones

La aplicación móvil cumple los siguientes requisitos en cuanto al índice de peticiones que el usuario puede realizar:

- Ver sus peticiones pendientes.
- Seleccionar una petición pendiente y ver su información con más detalle.
- Poder ver sus peticiones cerradas.
- Si el usuario selecciona una petición cerrada del listado mostrado, debe poder ver la información detallada del estado de la misma.

2.2.5 Sistema de notificaciones

Por último, la aplicación móvil cubre la exigencia de poder ver las notificaciones relacionadas con un usuario y sus peticiones. Los requisitos son:

- Se debe poder ver el listado de notificaciones de un usuario.
- Si un usuario no tiene ninguna notificación se le informa con un mensaje y una lista vacía.
- Un usuario tiene que entrar al menú de notificaciones para poder ver si tiene notificaciones pendientes.

3. Diseño e Implementación

En general, *Willyfog* se basa en una arquitectura basada en servicios. La cual nos permite poder hacer actualizaciones del corazón del sistema (API) y que todos los clientes esten siempre a la última. Permite un desarrollo incremental y un modelo de negocio bastante configurable.

3.1 Arquitectura

A continuación se muestra la imagen de la arquitectura completa que conforma el proyecto. Aunque nosotros nos centraremos en la parte móvil y dejaremos a un lado la aplicación web, podemos ver a simple vista todas las partes que componen *Willyfog* y cómo están conectadas entre sí.

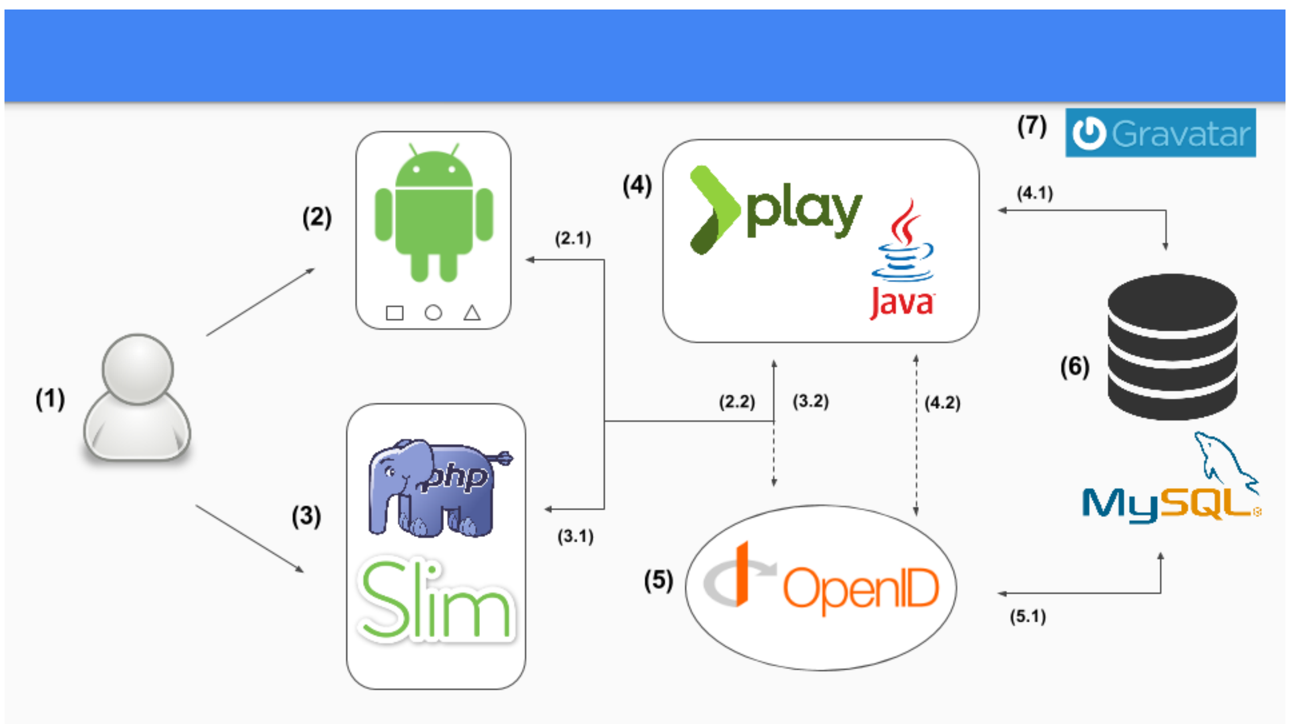

Figura 2: Arquitectura de Willyfog

Vamos a explicar un pequeño caso de uso donde daremos unas pinceladas de cómo se comporta la aplicación y cuál es el flujo de trabajo que un estudiante hace cuando accede a la aplicación móvil.

Empezando por el punto 1 de la Figura 2, podemos ver que el usuario tiene la posibilidad de acceder a la web (punto 3 de la Figura 2) o a la aplicación móvil (punto 2 de la Figura 2) indistintamente. Esto realmente no es así, sólo los estudiantes pueden acceder a la aplicación móvil y cualquier rol puede acceder a la aplicación web.

Una vez que el estudiante decide acceder al móvil, el móvil lo primero que hace es ponerse en contacto con el servicio OpenID (punto 2.1 de la Figura 2) y pide la clave pública, la clave pública se necesita para descifrar el token que OpenID le devolverá cuando el estudiante se autentique con toda la información del usuario. Ahora el usuario será enviado a un web view donde tendrá que rellenar un formulario servido por OpenID con su email y contraseña. Se autenticará y el móvil comenzará la comunicación con la API (punto 2.2 de la Figura 2) de una manera segura.

3.1.1 Base de datos

La base de datos fue el primer paso en el desarrollo de Willyfog. MySQL permite gestionar las bases de datos muy cómodamente, así que fue la opción elegida. En cuanto al diseño de la base de datos vamos a explicar un poco las principales entidades de nuestro modelo y a mostrar unas cuantas imágenes para que se vean las relaciones entre sí.

Puesto que la aplicación móvil no tiene manera de gestionar la base de datos ya que, no existe un SGBD completo que sea capaz de correr con los recursos tan limitados del móvil, nace la necesidad de una interfaz de aplicación que funcione como una capa que habla con la base de datos. Es por esto, que el diseño de la base de datos pertenece al ámbito de la API.

Se necesitan entidades para guardar tanto los países, como las ciudades. Cada ciudad tiene una o más de una universidad y a su vez una universidad tiene uno o más centros donde se ofrecen uno o más grados. Todas estas entidades contienen un nombre y un código que las identifica, además de un identificador numérico único que se utiliza de forma interna. Otra de las normas que hemos seguido a la hora de diseñar la base de datos es la de hacer tablas que contengan enumerados, por ejemplo, cuando detectamos que existe un enumerado en el sistema, lo registramos en una tabla, por ejemplo, los planes de movilidad. Este patrón nos beneficia para que no puedan existir valores inconsistentes a la hora de hacer peticiones de equivalencias.

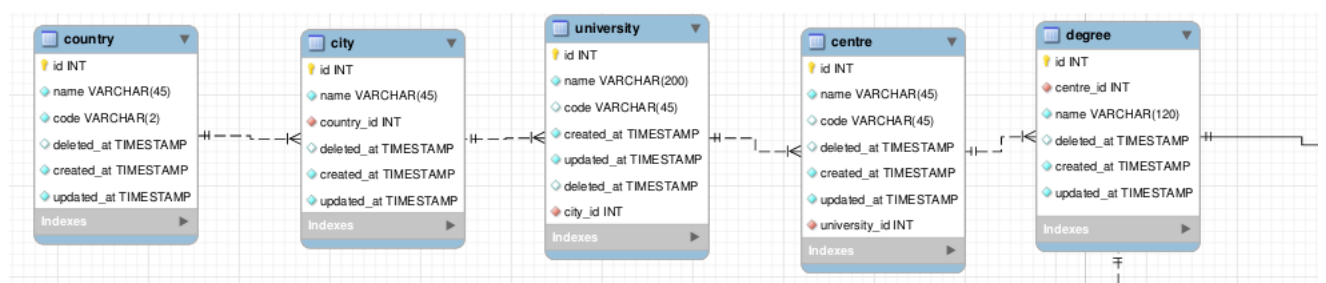

Figura 3: Entidades que forman los grados

Como podemos ver en la Figura 3, todas las entidades tienen un *timestamp* donde se guarda la hora en la que una fila fue creada, actualizada y borrada. Esto nos da mucha información a la hora de hacer auditorías a la base de datos. También todas las entidades

de nuestra base de datos tienen en común un identificador numérico único que se utiliza internamente por la API para acelerar el indexado.

Una de las entidades más importantes de la base de datos es la entidad asignatura. Una asignatura tiene un código, un nombre y un número de créditos y pertenece a un grado. Alrededor de esta entidad nacen otras relaciones con la entidad equivalencia y la entidad “asignaturas parecidas”. En la Figura 4 podemos ver estas relaciones más claramente:

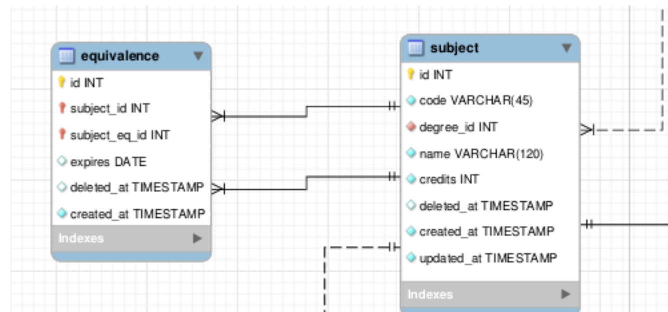

Figura 4: Entidades de equivalencia y asignaturas

Para modelar que una asignatura es equivalente a otra, lo que tenemos es una relación de muchos a muchos con la entidad equivalencia. Una equivalencia es la relación que hay entre una asignatura A con una asignatura B que son reconocidas, donde puede existir una fecha de expiración de manera opcional. En estas dos entidades es donde se centra toda la lógica de negocio a la hora de reconocer asignaturas en el sistema.

La siguiente entidad en la que nos vamos a centrar es en la petición. Una petición se registra en el sistema como una tabla que relaciona un estudiante, una asignatura de origen con una de destino y un tipo de movilidad. Otro patrón que hemos usado en este punto ha sido el de separar en entidades diferentes tanto las peticiones aceptadas como rechazadas. Esta mejora hace que la lógica de negocio se simplifique muchísimo, pudiendo, de un vistazo, ver las peticiones aceptadas y las rechazadas. En la Figura 5 podemos observar esta relación.

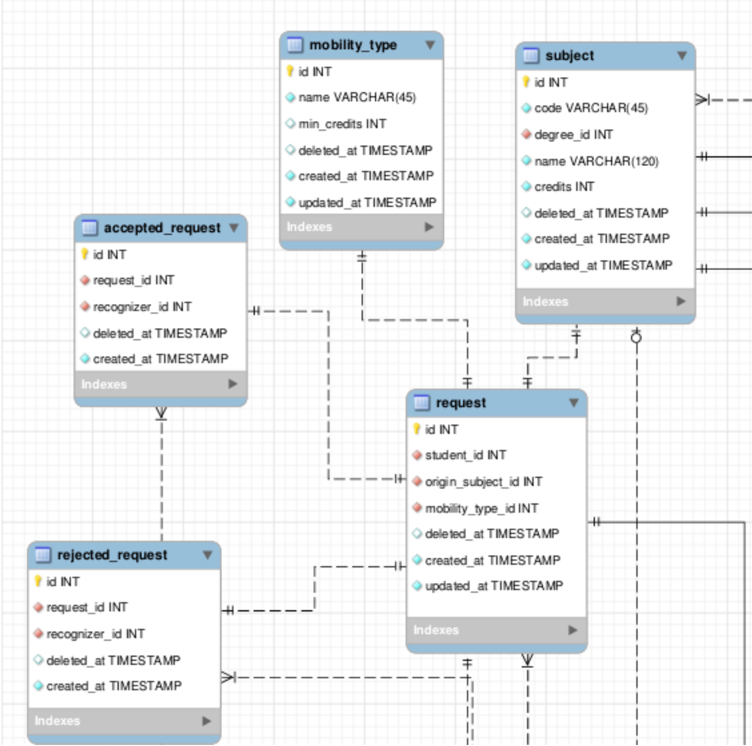

Figura 5: Entidades de peticiones

Se puede ver claramente que lo único que nos falta en el sistema son los usuarios y los roles. En la Figura 5 podemos ver que las entidades de peticiones aceptadas y rechazadas tienen una relación con el profesor reconocedor.

La siguiente entidad es la encargada de guardar toda la información que un estudiante rellena en el formulario de equivalencias. Antes de mostrar como hemos modelado tanto los roles, como los usuarios, sus notificaciones y comentarios, vamos a comentar esta entidad, muy importante en el sistema. Para permitir que una asignatura sea equivalente a varias (con un máximo de 3) hemos creado una entidad que, con una relación de muchos a muchos, relaciona una petición con una asignatura. La Figura 6 muestra esta relación.

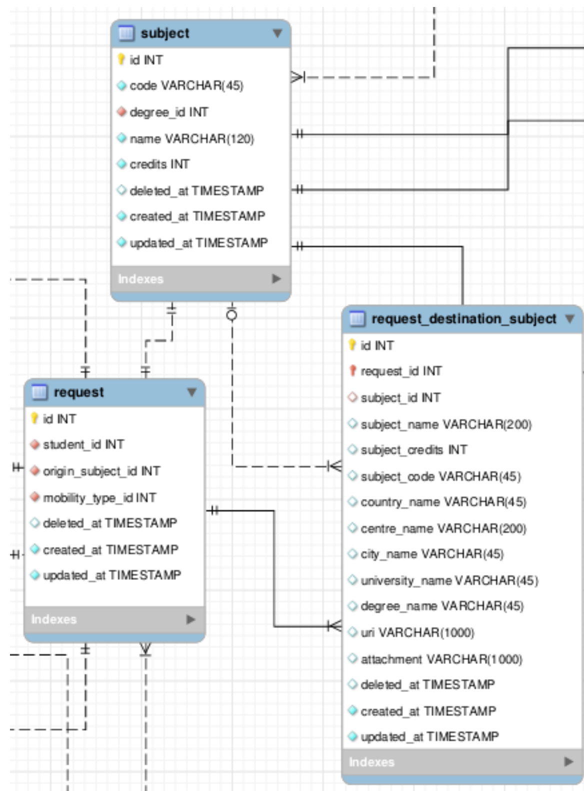

Figura 6: Entidad de asignaturas por petición

Como podemos ver en la Figura 6, de una petición de equivalencias de asignaturas se guardan los identificadores de la petición y de las asignaturas, el nombre de la asignatura, los créditos, el código de la asignatura, el país, la ciudad, el centro y al grado al que pertenece esa asignatura y la URI o el path de los archivos que adjunta el estudiante.

Ahora pasamos a los usuarios. Los usuarios en el sistema tienen la misma forma de autenticarse, es por esto que no hemos querido separar en varias entidades los diferentes roles. Un usuario está formado por un nombre, un apellido, DNI, email y un *hash* de la contraseña. Para modelar los roles hemos hecho una relación de muchos a muchos entre la entidad rol y usuario. Un rol está formado por un nombre y un permiso.

Los permisos en la aplicación son tratados a nivel de aplicación y no a nivel de base de datos. Es por esto que el atributo de permiso que tiene el rol está puesto para futuras mejoras que se añadan al sistema, para dar flexibilidad.

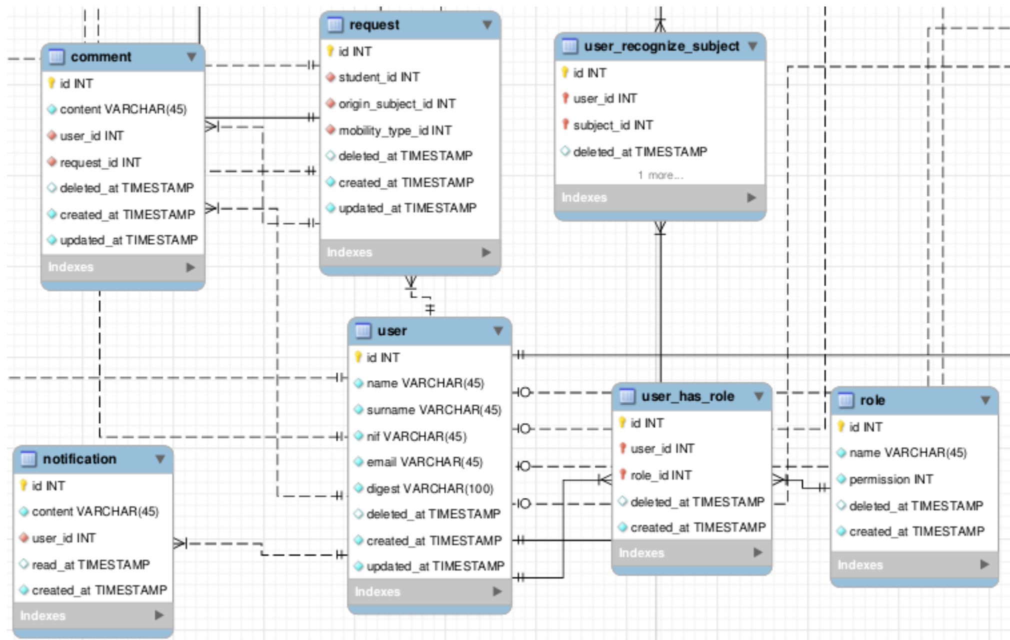

Figura 7: Entidades de usuario, notificaciones, comentarios y roles

En la Figura 7, podemos ver cómo se relacionan los usuarios y los roles. También se observa, que las notificaciones pertenecen a un usuario y no son más que un contenido y si estan leídas o no. Los comentarios estan formados también por un contenido y relacionan a un usuario con una petición (a la que se comenta).

Por último, tenemos que tener alguna forma de relacionar un usuario con un token de acceso que OpenID nos provee. Para ello necesitamos una serie de tablas que la librería de OpenID nos da. Todas estas entidades están relacionadas con un identificador de usuario, y son necesarias para completar todo el proceso de autenticación en la aplicación. En la Figura 8 podemos verlo.

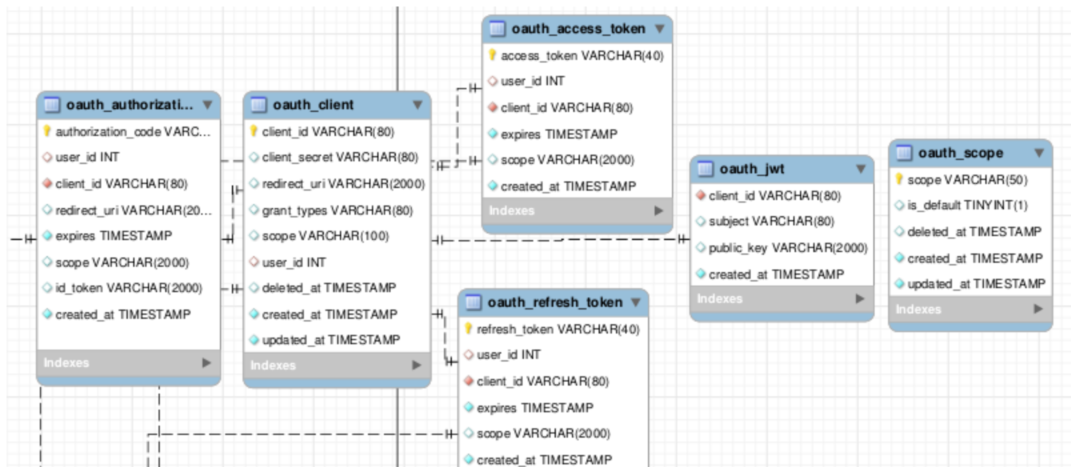

Figura 8: Entidades de OpenID

3.1.2 API RESTful

Una vez establecido el diseño de la base de datos, modelar la arquitectura de la API es bastante intuitivo. Antes que nada tenemos que pensar en qué tipo de mensajes va a recibir e intercambiar la API, qué tipo de servicios ofrece y a quién los ofrece. Ya que un usuario normal no puede acceder a los recursos que ofrece la API si no esta autenticado en el sistema. Por ello primeramente teníamos que elegir la arquitectura a usar en la API, sin duda alguna fue RESTful.

Los servicios que la API ofrece son exactamente los mismos que definimos en la sección de requisitos, por tanto lo único que tenemos que tener en cuenta a la hora de construir la API es que ella, es la encargada de tratar los datos, guardarlos en la base de datos y servirlos a todos sus clientes en formato JSON.

Proteger la API de peticiones externas no es más que pensar en una especie de proxy o *middleware* que actúe justo enfrente de las peticiones a los recursos de ésta. Por tanto, necesitamos algo que nos permita desarrollar este tipo de lógica.

Por tanto, el patrón que nuestra arquitectura sigue es el de una aplicación normal MVC pero sin la parte de la V (vista). Es decir, como si de una aplicación java clásica se tratase, con sus controladores, sus modelos (meros POJOs [24]), y sus capas de acceso a la base de datos donde separamos los modelos de las consultas los *daos*. Con esto dejamos

listo la arquitectura de la aplicación, los diferentes paquetes que definen el proyecto y podemos comenzar a trabajar.

3.1.3 Aplicación móvil

Una de las cosas más importantes de la aplicación móvil era cómo iba a ser estructurada. Inicialmente se pensó como una aplicación basada en actividades, es decir, cuando el usuario navegara por la aplicación iba a ir saltando de actividad en actividad hasta completar los diferentes flujos de trabajo. No obstante, esto provocaba que la aplicación fuera ralentizada, ya que en cada actividad se lanzaban varias tareas de forma asíncrona.

Esto provocó que dieramos otro enfoque al diseño de la aplicación móvil. Se definieron sólo tres actividades principales, dos de ellas eran las encargadas de mostrar el formulario de autenticación, y de establecer la conexión segura con la API. A partir de aquí, sólo iba a existir una actividad en memoria la cual iba a ser la encargada de orquestar todo el menú y los flujos de la aplicación. Sin más, la aplicación son tres actividades y todo lo demás fragments.

3.1.4 OpenID

Este servicio tenía que tener el mínimo impacto en nuestro diseño. No conocemos la especificación completa del protocolo, y tampoco es objetivo de este proyecto el de hacer una implementación desde cero de OpenID Connect. Por tanto el diseño de OpenID está en manos de la librería que hemos usado.

Cierto es que, OpenID en el caso de la aplicación móvil se trata de una manera especial. Hemos hablado de que tenemos tres actividades principales y dos de ellas son para la autenticación. De estas dos últimas actividades una de ellas es sólo para OpenID. Se trata de un WebView de android que carga el formulario que OpenID sirve a sus clientes.

3.1.5 Integración con servicio gravatar

Ya que uno de los requisitos que teníamos en el móvil era ofrecer perfiles de usuario, necesitábamos tratar con imágenes. Esto no era tampoco uno de los objetivos del proyecto, así que el diseño de gravatar consiste en implementar su servicio. Sólo era necesario en el diseño del perfil de usuario, y es la API la encargada de ofrecer el link hacia el cliente. Esto se consigue haciendo un hash md5 del correo electrónico del usuario autenticado y en el caso de que tenga cuenta en gravatar, su foto aparecerá.

3.2 Tecnologías usadas

Hemos hablado de que hemos estado haciendo un estudio de que tecnologías usar, pero no hemos explicado el camino que hemos seguido hasta quedarnos con la que más se nos ha ajustado a nuestras necesidades.

3.2.1 Buscando las tecnologías

Inicialmente la API la íbamos a construir con PHP, concretamente con un framework llamado Phalcon [16]. Esta propuesta la desechamos, debido a las dificultades que nos ofrecía el framework para poder adaptarlo a nuestrar necesidades. Phalcon es una extensión para PHP construida en el lenguaje de programación C. Tanto es así que esta escrito con un lenguaje compilado llamado Zephir [17]. Esta dependencia es la que nos hizo rechazar esta propuesta, nos ofrecía un alto rendimiento a cambio de no poder moldearlo a nuestras exigencias.

Tanto **Adrián González Leiva** como yo, somos unos apasionados de la programación funcional, de los lenguajes compilados y si pueden ser de tipado fuerte y estático, mejor aún. El siguiente paso que dimos fue en esa dirección. El candidato perfecto fue Scala.

Scala es un lenguaje compilado que nace de Java. De hecho es compilado al mismo bytecode que Java y es interpretado por la misma JVM.

Después de muchos estudios y esfuerzo, conseguimos tener un framework que se adaptara a nuestras necesidades y cumpliera el objetivo de darnos lo mínimo necesario para enrutar y mapear objetos a consultas de base de datos. Finch [8] un *microframework* construido encima del framework de Twitter, Finagle [9]. Esta propuesta fue arriesgada, dura y no fuimos capaces de desarrollar con la velocidad que esperábamos.
La falta de conocimientos en Scala, estar tan a bajo nivel de la arquitectura cliente-servidor y nuevamente otro fallo en el framework, nos hizo dar media vuelta y retornar al mismo punto desde el que iniciamos la búsqueda.

Habíamos probado un lenguaje de scripting, un lenguaje funcional y ya solo nos quedaba Java. Cuando empezamos a buscar sobre frameworks y librerías para Java, teníamos una cosa bastante clara, **no queríamos Java EE**. No queríamos realizar una aplicación monolítica, no queríamos ORM pero si queríamos protección, no queríamos gestor de plantillas (no tiene sentido para una API) y no queríamos tener un servidor de aplicaciones como *JBoss, Glassfish o Tomcat.*

De esta manera, nos dimos cuenta del mundo que desconocíamos de Java SE. Java tiene numerosas librerías y frameworks con los que poder hacer aplicaciones web sin tener que desplegar la aplicación en un servidor web, dispone de una pequeña librería llamada *JDBC* que nos provee de conexión con la base de datos y nos da protección contra inyecciones SQL, y podemos trocear la API hasta el punto que quisiéramos. Sin mencionar que Java SE 8 dispone de una API que permite programación funcional de una manera no muy bonita, pero cogimos el ritmo bastante rápido.

Finalmente, Play framework [18] y unas cuantas librerías fueron la elección de tecnologías para la API.

Ahora nos quedaba encontrar una solución para el servidor de OAuth2 [19]. Queríamos

algo que diera tokens de acceso a los usuarios registrados de la aplicación, es decir, a nivel de usuario y no de cliente de la API. Conocimos el protocolo OpenID Connect que ofrecía justo lo que estábamos buscando.

OpenID Connect esta construido encima de OAuth2 y sirve para manejar la autenticación de los usuarios de un servidor de autenticación. Para ello se usa un servidor el cual se encarga de asociar tokens de acceso por cada usuario registrado en el sistema. OpenID Connect además de asociar usuarios con tokens, también ofrece información del perfil de los usuarios.

Así pues, montamos un servidor OpenID Connect que diera servicio a nuestra API.

En la aplicación móvil, ya que teníamos Java, tener conocimientos sobre Android no era más que leer la documentación y practicar. Es por esto que llegamos a plantear las opciones que teníamos a nuestro alcance y Android salió como vencedora. El rendimiento que ofrece, no tiene rival frente a Ionic[14] o Cordova[13]. Frameworks que utilizan JavaScript para la realización de aplicaciones móviles. Por tanto, la versión de Android era lo único que nos faltaba por decidir, al final se opto por la versión 21 de la API de Android (5.1 Lollipop).

En cuanto a las herramientas que íbamos a usar para el desarrollo y producción del sistema, fueron más que directas. Vagrant [21] como máquina virtual y Docker [22] como contenedor para simular el servidor de producción y servidor de base de datos. Para ver más información sobre estas herramientas mirar la sección 4.4.

3.3 Dependencias del proyecto

Saber cuáles son las dependencias del proyecto es fundamental para el mantenimiento y desarrollo del mismo. Vamos a listar las dependencias de cada parte para que sea fácil consultar las mismas. Casi siempre, en un proyecto java se utiliza un sistema gestor de dependencias muy famoso, Maven[25], pero al conocer Scala y el framework Play descubrimos otro gestor de dependencias bastante potente llamado SBT[26]. Simplemente lo que nos hizo decantarnos por uno y no por otro fue, la *compilación lazy* o *compilación en caliente*. Esto significa que conforme estábamos desarrollando la API, desde el navegador podíamos ver los cambios simplemente refrescando sin tener que parar el servidor, compilar y lanzar de nuevo el servidor.

SBT se nutre de los mismos repositorios que Maven, esto nos da la ventaja de poder importar cualquier librería java a nuestro proyecto sin ningún problema. Uno de los pocos inconvenientes de SBT es que es demasiado lento a la hora de crear proyectos, pero una vez que se pone en marcha es bastante útil.

3.3.1 Dependencias de la API

Básicamente ya hemos hablado de las dependencias de la API, pero ahora las vamos a listar y vamos a hablar un poco de ellas con más detalle. Como era de esperar, no

queríamos muchas dependencias en nuestro proyecto, ya que, esto hace que el mantenimiento sea bastante complicado y difícil de realizar. Sin embargo, la consolidación de las dependencias que hemos elegido y el estudio que hicimos para elegirlas nos hicieron ver que, iban a ser dependencias con un soporte bastante prolongado y con una documentación bastante extensa.

- **Play framework**: Este framework es soportado por la empresa Lightbend, una de las empresas autora de varios frameworks muy exitosos para Scala, sin duda, una garantía. De Play sólo usamos unas cuantas cosas que nos han hecho falta, como: el enrutador, Play ofrece un enrutador bastante simple y potente que permite tipar las rutas, es decir, podemos decirle a Play el tipo de dato que espera una ruta en la URL; el middleware, Play tiene un sistema de middleware que nos permite poner antes de la llamada a un método un cierto predicado que tiene que cumplirse antes de que ese método se ejecute, con esto conseguimos que las rutas se protejan con los tokens que OpenID nos da; la inyección de dependencias, esto es una doble dependencia ya que, para que Play funcione con inyección de dependencias necesita otra librería que listaremos más adelante.

  La inyección de dependencias básicamente nos permite tener disponible la implementación de una interfaz instanciada en un objeto en cualquier parte del código. Esto es posible ya que Play por debajo tiene una factoría que va instanciando todos los objetos inyectables. Sin esto no sería posible usar una única conexión de base de datos de manera cómoda, por ejemplo, para poder usar una única conexión de base de datos tendríamos que tener una clase singleton o estática. Esto no es muy recomendable cuando quieres hacer pruebas y tests, y generalmente provoca errores.

  Otra de las cosas que Play nos permite es el **no tener servidor de aplicaciones**, ya que Play tiene un servidor integrado bastante potente llamado netty [27] el cual nos da la posibilidad de desplegar la API como un mero proceso más en el servidor.

- **Guice**: La siguiente dependencia viene de la mano de Google Inc. Guice [28] es una librería que permite inyección de dependencias en Java y es la librería que hemos usado en nuestro proyecto. Gracias a ella, Play es capaz de instanciar los objetos con sus factorías. Básicamente Guice tiene varios tipos de instanciación de objetos, los cuales nos permiten dar a una interfaz una implementación, instaciar un objeto directamente con su respectiva clase o hacer instanciaciones como objetos singleton y Guice los maneja por nosotros.

  Lo que normalmente se hace es, a nivel de código, definir las dependencias de todo el proyecto en una clase llamada *Module.java* que debe estar en nuestra raíz. Esta clase Guice la maneja y desde el inicio del arranque de nuestra aplicación la ejecuta para que, antes de nada, se resuelvan las dependencias de todo el proyecto. Como si de un *factory* se tratara, va resolviendo todas las dependencias e instanciando los objetos que se necesitan en el código.

- **Gson**: De nuevo Google Inc. nos trae una de sus librería estrella. Aunque Play ya ofrece un serializador de JSON, elegimos esta librería por la facilidad que nos daba a la hora de parsear objetos java planos (POJOs) a JSON y viceversa.
- **Sql2o**: Ya que no queríamos un ORM, por múltiples razones, buscamos una librería que fuera una fina capa de abstracción a la hora de mapear objetos java de las entidades de la base de datos. Sql2o[29] es una librería que escribiendo las consultas en MySQL, nos devuelve un objeto que podemos parsear con mucha facilidad a JSON gracias a la librería Gson.

  El hecho de no usar un ORM era porque ni queríamos un *query builder*, ni porque nos interesaba la idea de cambiar de sistema gestor de base de datos. Un ORM nos iba a dar una latencia importante a la hora de hacer consultas a la base de datos desde java que no necesitábamos. Simplemente con el mapeo nos bastaba.

3.3.2 Dependencias de la aplicación móvil

Las dependencias del móvil son bastante más reducidas que las de la API. Ya que sólo necesitábamos un cliente HTTP con el que hacer peticiones GET y POST cómodamente, y serializar/deserializar mensajes JSON a objetos java. A continuación las listamos:

- **API Android**: Aunque parezca mentira, la versión de la API usada en Android importa. La que se ha utilizado en este caso es la versión 21 de la API. Una de las cosas que nos ofrecía esta versión era poder hacer un diseño *material* fácilmente. Muchas de las opciones que esta versión de Android ofrece a la hora de construir las interfaces de usuario las anteriores no las tienen.
- **Gson**: Como librería para parser JSON a objetos.
- **OkHttp**: Una de las librerías estrella de Android que nos da un cliente de HTTP completo y fácil de usar.

3.3.3 Dependencias del servidor OpenID

El servidor OpenID está construido por una librería bastante famosa de PHP, bshaffer [30] es el autor del servidor OAuth2, que entre otras muchas cosas tiene OpenID Connect implementado. Este autor es trabajador de Google Inc., una de las razones por lo que lo elegimos fue por la facilidad que nos daba la librería a la hora de poner en marcha un servidor de autenticación con OpenID.

3.4. Diseño API

Hemos hablado de la arquitectura de la API y de su diseño, pero ahora vamos a hablar de las decisiones en el diseño que han marcado la diferencia en todo el proyecto.

3.4.1 ¿Por qué REST y no SOAP?

Cuando uno construye APIs tiene que decidir varias cosas antes de poder continuar. ¿Va a ser utilizada por más clientes?, en caso afirmativo, ¿Cómo va a ser protegida?, ¿Necesita mantener el estado?, ¿Qué tipo de mensajes intercambia?. Todas estas preguntas hacen decantarte por una arquitectura u otra.

SOAP es un protocolo que funciona intercambiando mensajes XML, estos mensajes comunmente viajan a través de HTTP aunque se puede utilizar cualquier otro protocolo de transporte. A diferencia de REST, SOAP define de una manera formal el tipo de mensajes que el servicio recibe mediante un archivo llamado *WSDL*, permite la autenticación de usuarios mediante usuario y contraseña, y en general una forma más compacta de definir un servicio web. SOAP no fue elegido por el hecho de tener que definir el WSDL y tener que intercambiar mensajes XML.

Sin embargo, REST nos permite trabajar con mensajes JSON, no tenemos que definir un archivo WSDL aunque existe algo parecido llamado WADL ya que, no tenemos que definir de manera formal ningún contrato con nuestros clientes, y trabajamos a través de POST, GET, PUT y DELETE, algo a lo que estamos acostumbrados a trabajar. Uno de los inconvenientes de REST es la dificultad a la hora de proteger de los recursos, que teniendo un protocolo como OAuth2 se soluciona fácilmente, y que si este servicio tuviera que hacer *streaming* de datos no sería la mejor solución.

3.4.2 Versionado

Una de las partes importantes a la hora de diseñar la API es el versionado. Aunque parezca una mala práctica, mantener varias versiones del servicio web a veces es necesario, ya sea por el tipo de clientes que tengas, o porque quieras ir poco a poco migrando tu servicio de una versión a otra.

La forma que hemos seguido a la hora de versionar la API es en las rutas de la misma. Básicamente un cliente de nuestra API tiene que saber que las rutas que tiene son del estilo:

http://www.example.com/api/v1/request

Lo que hacemos es poner v1, v2, v3,... para diferenciar una versión de la otra.

3.4.3 Rutas

Las rutas de la API se construyen de una forma muy similar al ejemplo anterior que hemos visto. Cuando se trata de una petición GET la ruta tiene que llevar en la URL los parámetros necesarios para realizar la acción necesaria. En cuanto a las peticiones POST podemos tener ciertas variaciones en las rutas, por ejemplo, una ruta POST que necesita el identificador único de una entidad de la base de datos pero también necesita un objeto

completo de una petición de equivalencia, lleva en la propia URL el identificador y en el cuerpo el objeto serializado a JSON de la petición en sí.

Para las peticiones PUT, que las usamos a la hora de actualizar datos, seguimos el mismo criterio que las peticiones POST. El identificador en la URL y en el cuerpo el objeto completo que se quiere modificar. Para las peticiones DELETE lo mismo.

3.4.4 Protección de la API con OAuth2

Para proteger la API hemos diseñado un flujo que nos permita autenticar a los usuarios. Debido a que REST no tiene ninguna forma de protegerse, tenemos que buscar otro tipo de soluciones, aquí es donde entra OAuth2.

OAuth2 es un protocolo para la protección de APIs que nos ofrece restringir el uso de los recursos de nuestra API a cualquier nivel, mediante el uso de acceso por roles, nos permite, no tener que compartir las credenciales de acceso a los recursos protegidos con ninguna aplicación cliente y lo más importante, todo este proceso va a través del protocolo de transporte HTTP/s.

Este tipo de acceso que damos a los clientes de nuestros recursos protegidos se hace mediantes *access token*, cadenas de texto arbitrario que representan la autorización del uso de esos recursos a un cliente en concreto. Normalmente este token se limita por un tiempo para evitar posibles problemas de seguridad, y se le provee al cliente con otro tipo de token llamado *refresh token*, otro tipo de token que permite al cliente volver a pedir otro token de acceso.

Durante todo el documento hemos estado hablando sobre OpenID Connect, pero ahora estamos hablando sobre OAuth2. Realmente OpenID Connect es OAuth2 pero con algunas mejoras para poder dar token de acceso no sólo a clientes (aplicaciones de terceros), si no, a usuarios finales con nombre de usuario y contraseña y permitir saber la información relacionada con un token de acceso y un usuario.

Una de las cosas más importantes de OAuth2 son los *scopes*. Los *scopes* nos permiten restringir los recursos por niveles de acceso. Esto es interesante en nuestra arquitectura a la hora de poder dar acceso a unos recursos a los estudiantes y otro a los profesores por ejemplo. Los *scopes* se definen libremente y a nivel de aplicación se le indica a los recursos que si el cliente que hace la petición no tiene el *scope* esperado, rechace la petición.

Los tipos de acceso que se conceden en OAuth2 que nos son útiles son:

1. **Client Credentials**: Básicamente este tipo de concesión de acceso es bastante simple. El cliente se autentica contra el servidor con sus credenciales (un identificador único y una cadena de texto secreta) y el servidor al compobar que el cliente es legítimo, le entrega un token de acceso válido por un tiempo. Este tipo de autenticación siempre se hace contra clientes que tienen un acceso privado y único a nuestros recursos, es decir, estos clientes se han dado de alta en nuestro servidor OAuth2 de una

manera confidencial y privada por ejemplo, contactando con nosotros directamente.

Este tipo de acceso en nuestro proyecto se usa a nivel de aplicación para las acciones que requieren hacer peticiones a la API pero no tenemos ningún usuario registrado, por ejemplo, en el formulario web de registro de nuevos usuarios.

2. **Authorization code**: Este tipo de concesión de acceso es la que usamos a nivel de usuario con OpenID Connect. El cliente pide al servidor de autenticación un acceso, en el cuerpo de la petición envía su identificador único, su scope, un estado arbitrario (un código aleatorio para demostrar la autenticidad de la petición que el servidor entrega al cliente igual que se le envío, así evitamos redirecciones malintencionadas) y una URI que servirá para redireccionar al usuario en caso afirmativo o negativo de autenticación.

   Una vez que el servidor ha comprobado que el cliente es legítimo lo manda a la URI dada y le proporciona un token de autorización. El cliente usará este token contra el servicio de autorización del servidor y así conseguirá un token de acceso a los recursos protegidos.

3.5. Diseño móvil

Anteriormente hemos hablado sobre el diseño móvil de manera general. Ahora vamos a explicar cómo esta hecha la aplicación móvil y qué patrones hemos seguido, hasta llegar a las especificaciones dadas.

3.5.1 MVC

El patrón de diseño que la aplicación móvil ha que hemos seguido es el MVC. Es un patrón clásico que nos permite que los controladores sean los que dinámicamente cambian las vistas y los modelos son simples POJOs que se parsean con los mensajes JSON que recibe de la API.

Los **controladores** los hemos separado en las clases que acaban en "Content", y en clases privadas dentro de las actividades principales de la aplicación. Debido a que los *Fragments* son simplemente controladores que se limitan a sustituir el contenido de la actividad principal por el que corresponda.

Para los **modelos** se ha seguido algo muy parecido a la API y en base a ella. En vez de diseñar los modelos, los modelos se definen directamente de los mensajes JSON que la API nos ofrece, de esta manera tenemos una manera directa de mapear las respuestas de la API a objetos Java y poder trabajar con ellos desde el móvil.

Finalmente, las **vistas** las podemos separar en dinámicas y estáticas. De hecho, sólo hay una vista estática y las demás son todas dinámicas. La única vista que es estática

es la principal, encargada de renderizar el menú y la interfaz que el usuario siempre tiene disponible en todas partes de la aplicación. Las vistas dinámicas se tratan desde las acciones del usuario, y son tablas y listas donde se renderizan cada vez que el usuario realiza una operación en la aplicación, por ejemplo, cuando pulsa en ver sus peticiones pendientes, o quiere ver sus notificaciones, en ese momento el fragment asociado a esa operación se activa y cambia el contenido principal de la vista.

3.5.2 Inicio de sesión OpenID

Lo primero que el usuario hace antes de entrar en la aplicación es pulsar un botón de acceso a la aplicación. Este bóton hará una petición asíncrona a la API, a una ruta desprotegida de la misma, que ofrece la clave pública. Esta clave pública es utilizada por el móvil para descifrar más adelante el mensaje JSON que la API ofrece con toda la información del usuario.

Una vez pasada esta vista, el móvil pide al servicio OpenID un formulario de acceso con su *Authorization code* (ver sección 3.4.4). Este formulario de acceso autenticará al usuario y en caso afirmativo el usuario accederá a la aplicación móvil a la sección de su perfil. Durante este último paso, el WebView que el móvil genera para el renderizado del formulario web, se destruye de forma mientras que el servidor está trabajando y se muestra un mensaje de *Loading...* hasta que recibe la respuesta del servidor.

3.5.3 Diseño de la interfaz de usuario

La interfaz de usuario se ha diseñado siguiendo el modelo que Google ofrece para diseños *material design*[31]. Principalmente todo gira en torno al menú, éste está formado por una cabecera la cual muestra en la sección que el usuario se encuentra, a la derecha se da la opción de salir de la aplicación y a la izquierda se encuentra el botón que despliega el menú.

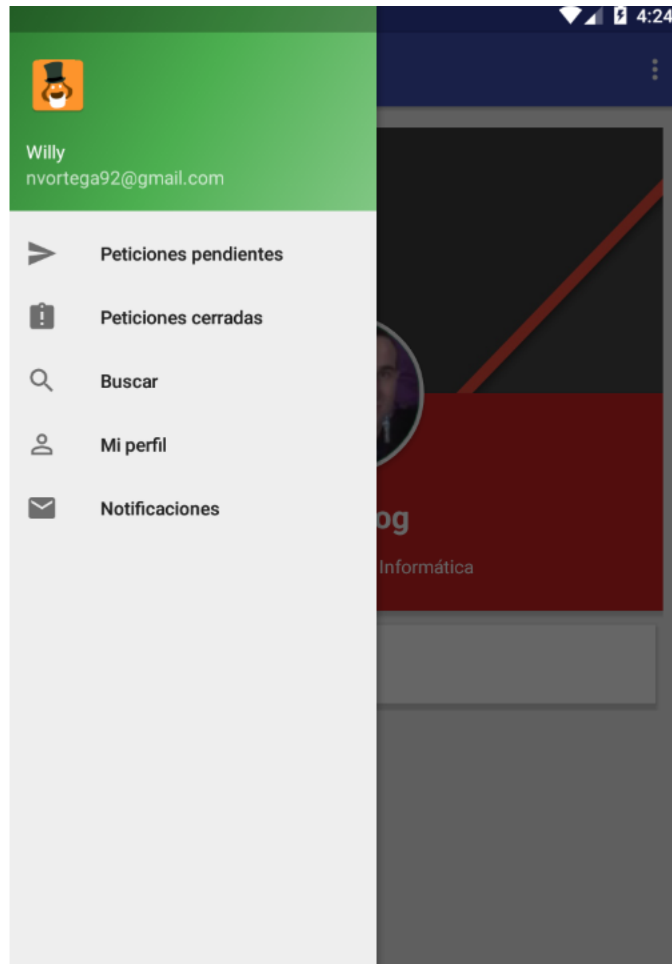

Figura 9: Pantalla de menú

Una vez que el usuario despliega el menú en la cabecera del menú desplegado se encuentra su nombre y una foto por defecto. A continuación se listan las diferentes opciones que el menú tiene:

- **Peticiones pendientes**: En este apartado se muestra una lista de todas las peticiones pendientes que tiene un estudiante. Si pulsa en alguna de la lista, podrá ver los detalles de la misma. Habrá un mensaje que indicará que la petición esta pendientes.
- **Peticiones cerradas**: Aquí saldrán las peticiones que han sido evaluadas por un profesor reconocedor. Este apartado tiene la misma similitud que el anterior, una lista donde aparecerán las asignaturas con su código y si se pulsa en la deseada, se podrán ver sus detalles. En este caso, se ha optado por otro tipo de diseño a la hora de mostrar al usuario si su petición ha sido aceptada o rechazada. Se muestra con un tono de transparente si la petición ha sido aceptada, en color verde y un check, o rechazada en color rojo y con una aspa.

- **Buscar**: En este apartado primeramente se muestra un campo para rellenar la asignatura que se quiera buscar en el sistema. En caso de que una asignatura no haya sido encontrada, se mostrará un mensaje de que la asignatura no existe en el sistema. En el caso de que sí exista en el sistema, se mostrarán todas las equivalencias que esa asignatura tiene, en forma de una lista.

- **Mi perfil**: En esta sección un estudiante puede ver su foto de perfil (si tuviera gravatar), su nombre y apellidos, su universidad y el grado en el que esta dado de alta. En esta sección se ha optado por un diseño *material design*.

- **Notificaciones**: Aquí simplemente se muestran las notificaciones que el estudiante todavía no ha leído. Se muestra en forma de lista y sólo informa de que usuario ha comentado o ha hecho alguna acción en las peticiones del estudiante.

3.5.4 Integración con la API

En la integración con la API, se han usado llamadas asíncronas de Android. Una de las cosas importantes que hay que tener en cuenta a la hora de desarrollar en Android es que el hilo principal de la aplicación tiene que ser lo más estable posible, si por ejemplo se quiere hacer una petición a un servicio y que la aplicación no quede congelada, se debe hacer de manera asíncrona.

Por tanto, la integración con la API se hace mediante peticiones asíncronas. Se tratan como clases privadas en cualquier lugar del proyecto Android donde tienen los siguientes métodos, *onPreExecute(), doInBackground(), onPostExecute().*

1. **onPreExecute()**: Este método generalmente gestiona alguna tarea que se debe hacer **antes** de que la petición asíncrona se ejecute. En nuestro caso, se utiliza para mostrar *dialogs* al usuario para mostrar que la aplicación está cargando.

2. **doInBackground()**: Esta acción es la petición asíncrona en sí. En nuestro caso, aquí va la petición HTTP al servicio de nuestra API que corresponda, recibirá el resultado y lo pasará al siguiente método.

3. **onPostExecute()**: Cuando la petición acaba, llega a este método el cual generalmente se utiliza para cerrar el *dialog* abierto en el método *onPreExecute()* y guardar la petición en la caché del móvil. Esto ahorra tiempo de respuesta de la aplicación y hace que se pueda utilizar la información en cualquier lugar del móvil. No obstante, no todas las peticiones del móvil se cachean.

3.5.5 Integración con OpenID

La integración con OpenID se hace de una manera similar a la de la API. Mediante una petición POST se autentica contra el servidor OpenID, mandando todos los datos necesarios, y la respuesta que se recibe se guarda en la caché del móvil. La actividad que maneja esta petición, es una de las principales de la aplicación llamada *OpenIdWebViewActivity*.

En esta actividad se realizan varias operaciones. La primera es hacer la petición a la ruta que el servidor de autenticación nos ofrece. En esta ruta se mandan los parámetros por GET, es decir, en la propia URL. Se manda el identificador del cliente, la URI de redirección, el tipo de respuesta que se quiere (en este caso un código de acceso), el *scope* una cadena siempre igual con el valor **openid** y el estado. La URL quedaría:

```
http://popokis.com:9000/authorize?client_id=<clientId>&redirect_uri=<redirectUri>
&response_type=<responseType>&scope=<scope>&state=<state>
```

Con todos estos datos, el servidor nos devuelve un formulario que el estudiante tendrá que rellenar con sus datos, esto lo mostraremos por un WebView.

Ahora sólo falta que el usuario introduzca las credenciales de acceso. En este momento, cuando el usuario pulsa el botón de *Authorize* se hace un POST a la ruta del servidor OpenID que nos dará un token de acceso si todo marcha bien.

En el cuerpo de esta petición POST pediremos al servidor que nos de acceso del tipo *Authorization code*, esto se especificará con una cadena de texto como clave *grant_type*, se mandará el identificador del cliente, la clave secreta del cliente, el código y la URI de redireccionamiento.

Si todo ha ido correcto, el servidor nos dará un token de acceso que durará por 24 horas, y que la aplicación móvil guardará en la caché durante ese tiempo. Y para ahorrar tiempo, el móvil en este momento es cuando usará la clave pública de la API para descifrar el JSON cifrado que el servidor de autenticación nos ha devuelto con la información del usuario. Así podrá ser guardada en caché y poder renderizar correctamente el perfil del usuario.

4. Metodologías

4.1. Planificación

En cuanto a la planificación del proyecto se ha seguido la metodología ágil mediante Scrum. Se han utilizado *sprints* de dos semanas, los cuales los hemos ido enumerando con las letras del abecedario. En total han sido 5 *sprints* y medio. Una de las formas que hemos tenido de motivarnos en el trabajo, ha sido creando una nomenclatura específica a la hora de ir tachando *sprints* en el calendario.

Cuando estás trabajando en un proyecto software, debes de tener todo perfectamente planificado, nosotros teníamos un calendario el cual mostraba todos los *sprints* e íbamos marcándolos con una triángulo, un aspa y un círculo respectivamente. El triángulo significaba que no habíamos llegado a la planificación de ese *sprint*, cosa que nos ha pasado sólo una vez. El siguiente símbolo es el de un aspa (representa la neutralidad), es muy típico ir tachando días en un calendario, así íbamos tachando algunos *sprints* en concreto 4 de ellos. Y por último el círculo, que representaba que un *sprint* había ido tan bien que nos había sobrado tiempo.

Para la planificación hemos usado pizarras de Scrum con la aplicación *Jira*[32] de *Atlassian*. Esta aplicación nos ha permitido llevar de una manera completa nuestro proyecto, asignar tareas y tiempos e ir haciendo un seguimiento bastante completo del desarrollo del proyecto. Jira permite la realización de informes en forma de gráficas para, de una manera visual, ver cómo ha ido el último *sprint*.

Las pizarras que usábamos contenían cuatro secciones. La sección de *ToDo* donde se listaban todas las tareas previstas para ese *sprint*. La sección de *In Progress* donde se ponían las tareas en las que se estaba trabajando en ello. La sección de *Testing* donde se dejaban las tareas hasta que se hacían las pruebas necesarias. Y por último, la sección de *Done*, donde se dejaban las tareas que habían completado el proceso de desarrollo y testeo.

4.2. Pruebas

Las pruebas que se han realizado en el proyecto han sido de una forma un tanto especial. Al darnos cuenta de que TDD no era una posible metodología ha seguir dado el tiempo y el desconocimiento en este ámbito, se decidió dedicar los últimos días de cada *sprint* (normalmente los dos últimos días) ha realizar las pruebas a mano.

El procedimiento era el siguiente:

1. Se desarrollaba una nueva funcionalidad.
2. Se dejaba en la pizarra Scrum, en la sección de *Testing*.
3. Llegado el día, se probaba la funcionalidad.

4. ¿Funciona como se esperaba?

    4.1 Sí, se pasaba a la pizarra de *Done*.

    4.2 No, volvíamos al punto 1 arreglando los fallos.

Así fuimos haciendo el testeo de todos los componentes del proyecto Willyfog. Como resultado final teníamos funcionalidades probadas por nosotros mismos y en última instancia por nuestros tutores.

4.3. Documentación

En cuanto la documentación de este proyecto la hemos querido separar en varias secciones que se encuentran en el anexo de esta misma memoria. Las secciones que podemos encontrar son:

1. **Manual de usuario**: Aquí se explica mediante fotos y comentarios como se pueden realizar todas las acciones disponibles en la aplicación móvil. Servirá para los estudiantes que deseen utilizar la aplicación móvil.

2. **Manual de despliegue**: Este manual muestra la forma de hacer un despliegue completo de todas las aplicaciones del proyecto Willyfog. Además de este manual, esta disponible un script que de forma automática hace un despliegue de la API.

3. **Manual de la API**: En este manual se muestra como hacer llamadas a la API, cuales son sus rutas principales y donde encontrar toda la documentación posible de la misma.

4.3.1 ¿Swagger o RAML?

A la hora de documentar la API nos hemos encontrado con un gran problema. ¿Cómo documentamos una API? De forma que se vayan explicando cada ruta, lo que recibe y lo que devuelve, ¿cómo se explican los mensajes de error?

Investigando sobre el tema nos topamos con varias herramientas que se utilizan para este fin. Dado que los servicios web REST no tienen la obligación de realizar ningún documento que especifique la forma en la que se reciben los mensajes, y las peticiones que aceptan, se han creado varias herramientas que permiten de una manera cómoda realizar esta tarea.

Swagger[33] y RAML[34] son dos herramientas que permiten no solo documentar una API REST y generar páginas estáticas para que un navegador las renderice, si no, especificar todos las rutas, los objetos que reciben e incluso permiten al usuario probar la API en el mismo momento que están leyendo la documentación.

Swagger además de una herramienta, ha sido diseñada para ser una forma de especificación de APIs que quiere ser estandarizada. Recientemente ha sido donada a la organización OAI [35], donde las empresas más importantes del mundo en el sector tecnológico dan apoyo a esta iniciativa.

Por tanto, teníamos varias opciones donde elegir, el caso era elegir la correcta. Primero se eligió RAML, la cual nos permitía escribir la documentación a través de un archivo de configuración YAML, una forma *human-writable* bastante más amigable que la forma en la que lo hacía Swagger (versión 1.0). Swagger permitía realizar la documentación de la API en el mismo código mediante una serie de anotaciones, pero esta forma nos causó bastantes problemas y era muy lenta, lo que nos hizo decantarnos por RAML.

Sin embargo, cuando estábamos escribiendo la documentación en RAML, se publicó una nueva versión de Swagger. Esta versión nos daba la posibilidad de escribir la documentación en un archivo YAML y además nos daba una herramienta que nos permitía ir visualizando la documentación conforme íbamos escribiéndola. Esto nos hizo cambiar rápidamente a Swagger y escribir la documentación de la API con esta herramienta.

4.4. Entorno de desarrollo

Uno de los pilares más importantes de este proyecto ha sido la forma que hemos tenido a la hora de establecer un entorno de desarrollo, qué IDE escoger y cómo trabajar para que el despliegue de la aplicación sea lo más rápido y parecido al entorno de desarrollo. Para ello hemos usado varias herramientas que hemos conocido durante el proyecto y que hemos utilizado dada la gran facilidad que nos han dado para montar todo el entorno.

4.4.1 Vagrant

Vagrant es una herramienta que permite configurar entornos de desarrollo en máquinas virtuales a través de "recetas" o *scripts* de provisionamiento de la máquina. Básicamente, Vagrant descargará una máquina virtual de su repositorio central, con un sistema operativo elegido, instalará las dependencias que indiques en tu *script* y enlazará una carpeta en tu máquina *host* con la máquina *guest* para que puedas desarrollar desde tu IDE en la máquina host y los cambios surtan efecto automáticamente en la máquina virtual. Sin duda, una herramienta imprescindible en nuestro día a día de desarrollo.

4.4.2 Docker

Por otro lado, Docker nos ofrece contenedores de herramientas software ya instaladas. Docker **no es una máquina virtual**, simplemente **virtualiza un sistema de ficheros** y aisla una o varias herramientas en una imagen. Docker corre como un proceso más en nuestra máquina host. A diferencia de una máquina virtual que es gestionada por un hipervisor (encargado de gestionar la memoria y las llamadas al sistema del sistema operativo host), Docker lo gestiona el propio motor de Docker (valga la redundancia). Esto significa, por ejemplo, que si tenemos un contenedor de Ruby on Rails de Docker,

y éste comienza a hacer forks, estos forks tendrán repercusión en nuestra máquina host pudiendo dejarnos sin poder crear hilos en nuestra máquina real.

4.4.3 Desarrollo y Producción

De esta forma podíamos montar el mismo entorno que el servidor de producción, y con el simple hecho de ejecutar el script de aprovisionamiento de la máquina vagrant, el entorno de producción estaría listo. Dado que las herramientas que usábamos en docker pueden ser instaladas en la máquina del servidor o podemos dejar instalados los contenedores en el servidor de producción y el cambio no se notaría.

5. Conclusión

La realización de este proyecto nos ha enriquecido como futuros profesionales en el sector, como personas. Han hecho que sepamos administrar nuestro tiempo, con planificaciones reales, pasos pequeños y metas fácilmente alcanzables que poco a poco han ido creando el proyecto Willyfog.

El camino que hemos seguido hasta darnos cuenta de que necesitábamos un lenguaje compilado con un tipado estático y fuerte, nos ha servido para aprender qué tecnología elegir para el desarrollo e implementación de un proyecto software. Un ingeniero debe tomar con precaución el etiquetar las cosas como "malas prácticas" de forma general. Hay que tomar el problema, ver las herramientas a tu disposición, elegir la apropiada (en algunos casos habrá que usarla previamente) y construir.

Los lenguajes dinámicos tienen un uso muy extendido y frecuente a la hora de desarrollar aplicaciones web, pero se quedan en la capas más altas de la arquitectura de las aplicaciones, normalmente en las vistas. Cuando hablamos de las capas más bajas, donde se realiza la lógica de negocio, debemos de saber elegir tecnologías que puedan ofrecer el mismo servicio cuando se tiene un bajo número de usuarios y éste crece hasta un número sustancial, sin tener que ampliar el *hardware*.

Cuando nos encontramos a nivel de datos, uno de los niveles que tanto nos apasiona, tomar las decisiones correctas es crucial, ya que, cometer un fallo de diseño hace que tengamos un lastre que acarrear en las demás capas, teniendo que subsanar con prácticas no muy buenas en las capas superiores. La base de datos es crucial, en este lado, podemos ver las aplicaciones como finas capas por encima que hacen que la lectura y la escritura de los datos sea más "humana", por tanto, la base de datos se convierte en **el pilar** de una aplicación software.

Gracias a la preparación que hemos recibido, hemos tenido la capacidad de poder adquirir nuevos conocimientos con manuales y recursos encontrados en la red, cosa que nos ha ayudado mucho a la hora de desarrollar este proyecto. Estamos seguros de que los conocimientos que hemos adquirido realizando este proyecto nos han dado una base más sólida para la realización de aplicaciones empresariales en un futuro.

Poder realizar una herramienta software que dé solución a un problema real, que puede ser usada por alumnos de la UMA y que sin duda va a ayudar a mucha gente, nos enorgullece. Demostrar de lo que somos capaces, asumir responsabilidades y estar a la última es una tarea que requiere mucho esfuerzo, trabajo y sobre todo compañerismo. Trabajar en equipo nos ha hecho crecer más rápido, aprender y "tirar del carro" mutuamente.

Un buen equipo, un buen puñado de tecnologías y herramientas nuevas y las ganas de hacer lo que más nos gusta, son el principal aliciente para motivarnos al máximo en nuestro desarrollo del trabajo de fin de grado. El camino que hemos seguido desde que empezamos el proyecto y como hemos acabado, ha sido sin duda una experiencia enriquecedora y nos

ha hecho aprender métodos que podemos aplicar en el día a día en nuestro trabajo.

Referencias

[1] http://www.uma.es/relacionesinternacionales/cms/menu/movilidadestudiantes/

[2] Desarollo ágil, *extreme programming*. https://es.wikipedia.org/wiki/Programaci%C3%B3n_extrema

[3] Metodología *Scrum*. https://es.wikipedia.org/wiki/Scrum

[4] *Android*. https://www.android.com/intl/es_es

[5] *Android Studio* IDE. https://developer.android.com/sdk/index.html

[6] *Git*. https://gitscm.com/

[7] *GitHub*. https://github.com/

[8] https://github.com/finagle/finch

[9] https://github.com/twitter/finagle

[10] https://icaro.ual.es/

[11] http://www.uma.es/programa_movilidad/outgoing_pmovilidad/index.php

[12] *Acuerdo Académico (Única)* http://www.uma.es/relaciones-internacionales/info/71003/conv-unica-paso-8/

[13] https://cordova.apache.org/

[14] http://ionicframework.com/

[15] https://es.wikipedia.org/wiki/Arquitectura_orientada_a_servicios

[16] *Phalcon PHP*. https://phalconphp.com/es/

[17] *Zephir*. https://github.com/phalcon/zephir

[18] *Play framework*. https://www.playframework.com/

[19] *OAuth2*. https://oauth.net/2/

[20] *OpenID Connect*. http://openid.net/connect/

[21] *Vagrant*. https://www.vagrantup.com/

[22] *Docker*. https://www.docker.com/

[23] *Gravatar*. https://en.gravatar.com/

[24] *Plain Old Java Object*. https://es.wikipedia.org/wiki/Plain_Old_Java_Object

[25] *Sistema gestor de dependencias Maven*. https://maven.apache.org/

[26] *Sistema gestor de dependencias SBT*. http://www.scala-sbt.org/

[27] *Netty server*. http://netty.io/

[28] *Guice*. https://github.com/google/guice

[29] *Sql2o*. https://github.com/aaberg/sql2o

[30] *Librería OAuth2*. https://github.com/bshaffer/oauth2-server-php

[31] *Material design*. https://material.google.com/#introduction-goals

[32] *Jira*. https://es.atlassian.com/software/jira

[33] *Swagger*. http://swagger.io/

[34] *RAML*. http://raml.org/

[35] *Open API Initiative (OAI)*. https://www.openapis.org/

A. Anexo

A.1 Manual de usuario

Antes de instalar la aplicación móvil asegúrese de que su móvil tiene la versión Android 5.1 o superior. Para poder instalar la aplicación se debe pedir una apk al correo que se muestra a continuación: *nvortega92@gmail.com*, ya que la aplicación no esta subida a ninguna tienda online y sólo esta disponible su código fuente en `www.github.com/popokis`, si se opta por bajar el código fuente se necesitará un IDE [5] para generar el archivo apk. Una vez instalada la aplicación y acceda a ella por primera vez, encontrará la siguiente pantalla:

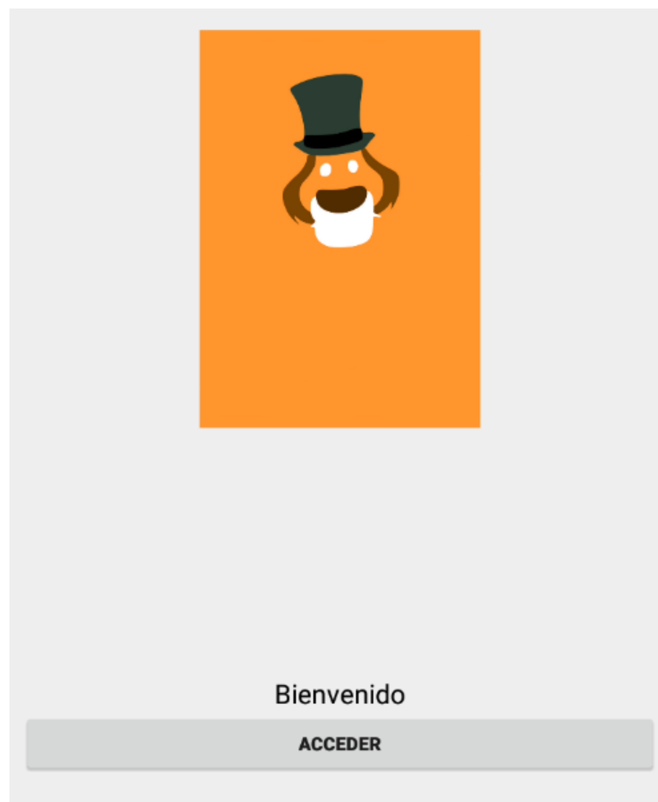

Figura 10: Pantalla de bienvenida

Tendrá que pulsar el botón **ACCEDER** y le enviará a la siguiente pantalla donde tendrá que ingresar sus datos de acceso a la aplicación. Si usted no está dado de alta, por favor, hágalo a través de la web que ofrecemos.

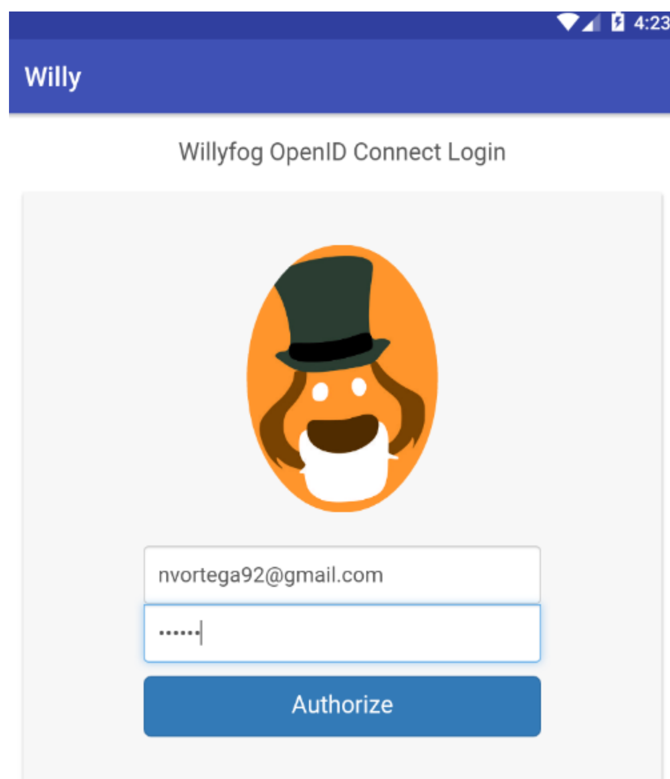

Figura 11: Pantalla de login

Pulse el botón **Authorize** para autenticarse y acceder a la aplicación. Por favor, espere unos instantes mientras la aplicación trabaja hasta llegar a la pantalla de su perfil de usuario.

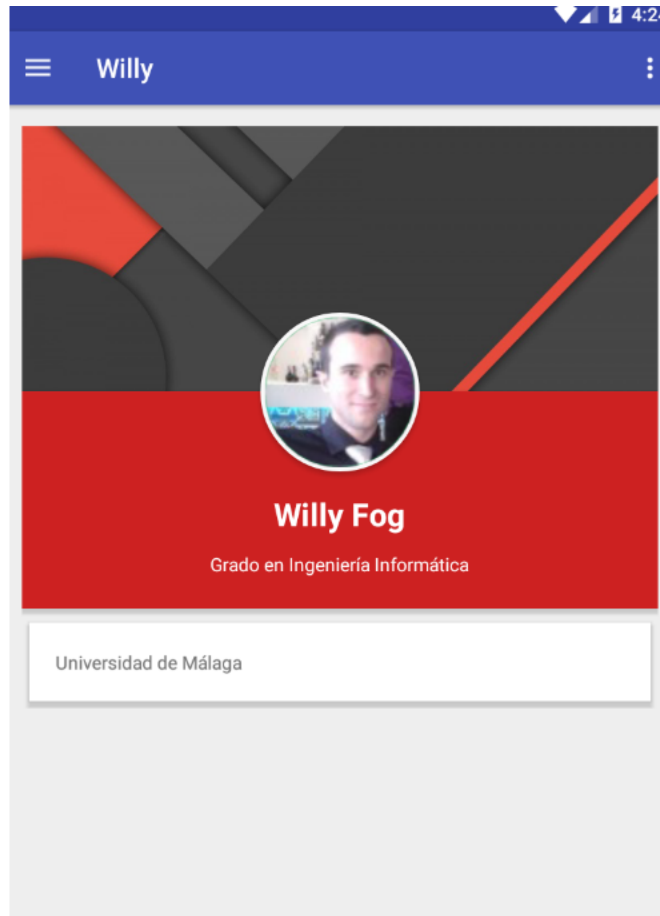

Figura 12: Pantalla de perfil de usuario

En la pantalla de perfil de usuario, podrá ver su información general sobre su nombre, el grado en el que está dado de alta y la universidad a la que pertenece. **Si esta información es errónea, por favor, contacte con el administrador del sistema**.

Para poder cambiar la imagen de su perfil, debe de darse de alta una cuenta en el servicio gratuito gravatar. **Si ya dispone de una cuenta en este servicio, por favor, omita este paso**.

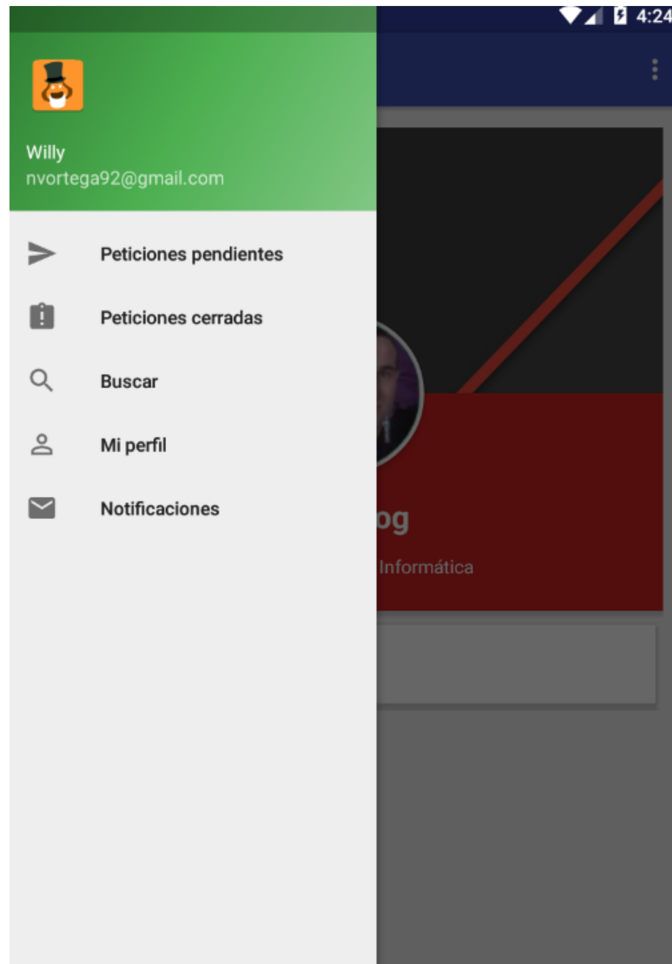

Figura 13: Pantalla de menú

En la Figura 13 se muestra el menú que tiene disponible. Aquí podrá navegar por la aplicación y poder aprovechar los servicios que ésta le ofrece. Ahora vamos a ir listando los diferentes elementos del menú y qué es lo que le ofrecen.

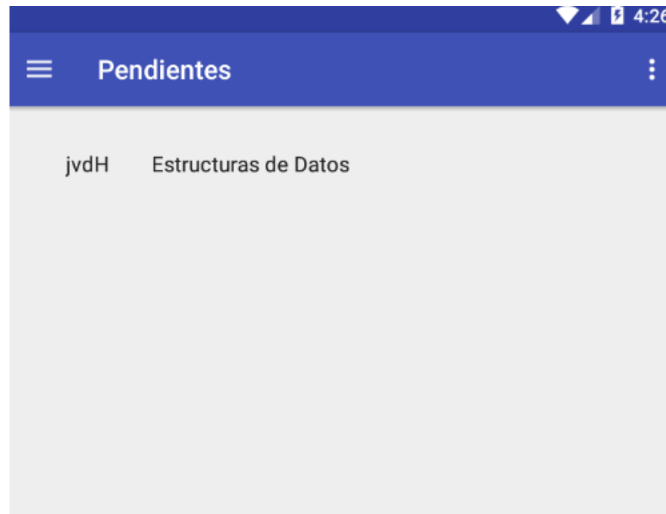

Figura 14: Pantalla general de peticiones pendientes

En este apartado del menú, se listan las peticiones que usted tiene pendiente de revisión por parte de la Comisión de Reconocimiento. A la izquierda se lista el código identificativo de la asignatura solicitada, y a la derecha el nombre de la misma. Si pulsa en una de la lista aparecerá la pantalla de detalles de la petición (Figura 15).

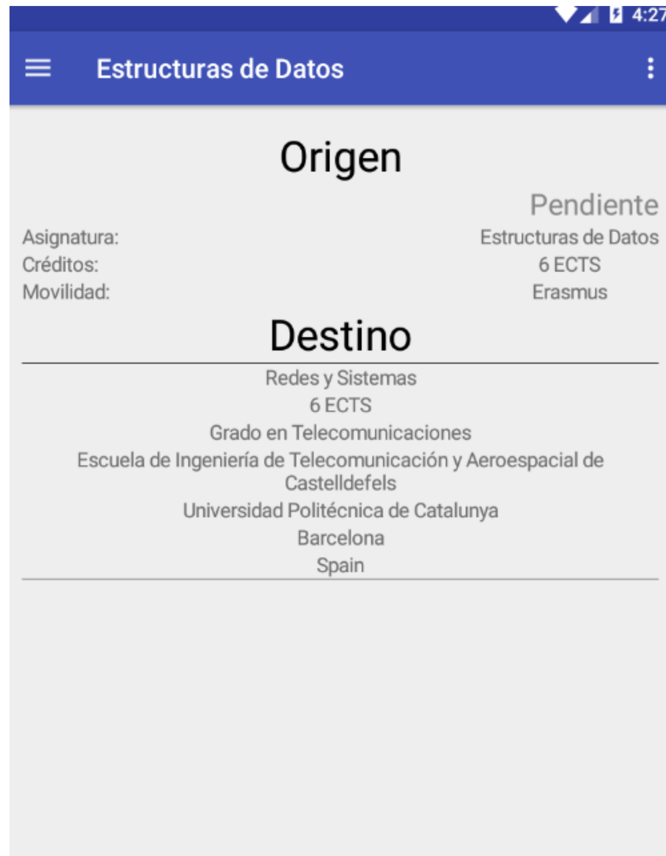

Figura 15: Pantalla de detalles de petición

En este apartado podrá ver la información de su petición, el estado en el que se encuentra en la esquina superior derecha y la información resumida de la asignatura/as de destino que usted eligió en la petición.

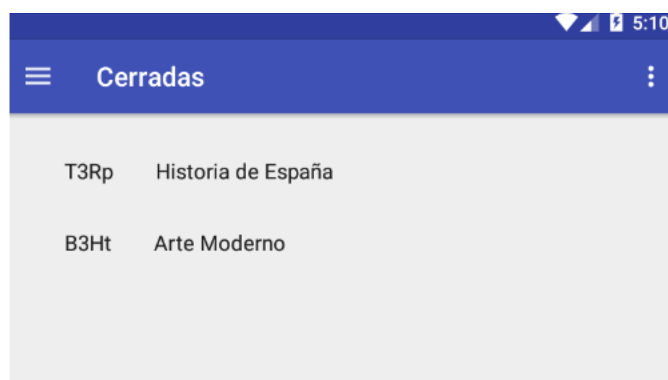

Figura 16: Pantalla de peticiones cerradas

Al igual que la sección de peticiones pendientes, en el apartado de peticiones cerradas podrá ver las peticiones que ya han sido revisadas y cerradas por parte de la comisión. Si pulsa en una de las peticiones podrá ver el estado en el que están, si han sido aceptadas o rechazadas (Figura 17 y Figura 18 respectivamente).

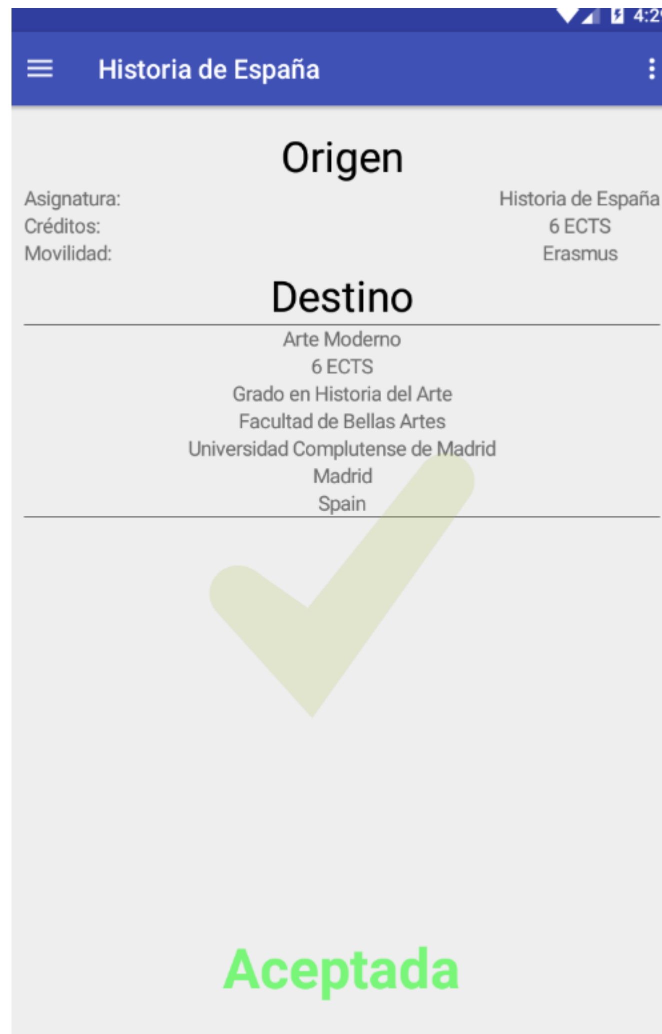

Figura 17: Pantalla de detalles de petición aceptada

Las peticiones aceptadas serán marcadas con una imagen de *check* verde y se informará de su estado abajo de la pantalla.

Figura 18: Pantalla de detalles de petición rechazada

Las peticiones rechazadas serán marcadas con una imagen de un *aspa* roja y se informará de su estado abajo de la pantalla.

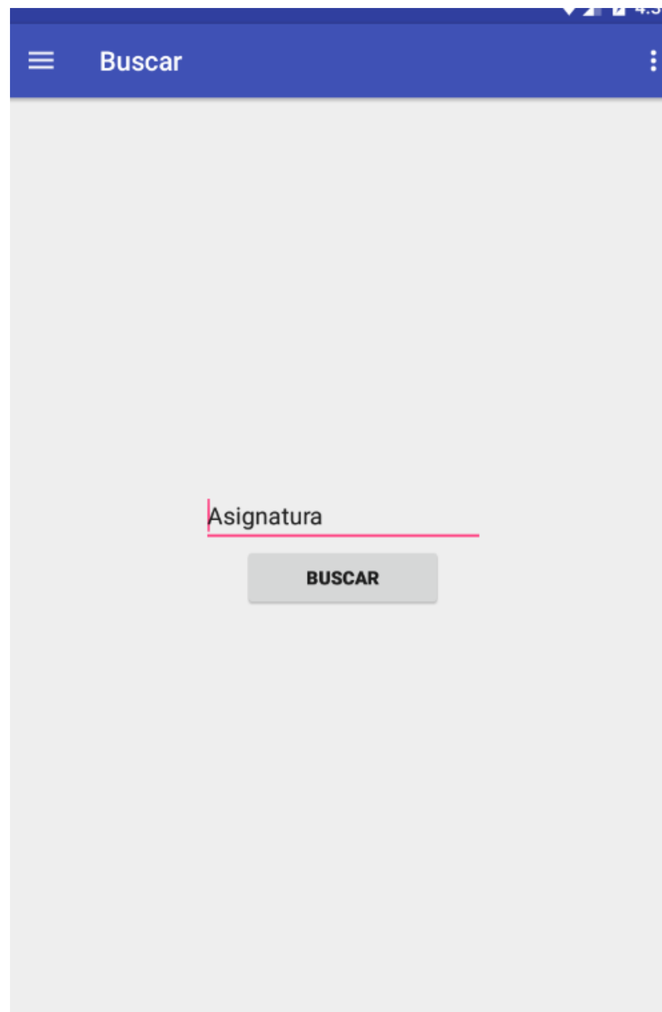

Figura 19: Pantalla de búsqueda

En la pantalla de búsqueda, se podrá buscar en todo el sistema de convalidaciones de Willyfog. Podrá poner el nombre de la asignatura que desea encontrar equivalencias y pulsando en el botón **BUSCAR** podrá ver los resultados (Figura 20), en caso de que no exista ninguna coincidencia en el sistema, se le enviará a una pantalla en blanco.

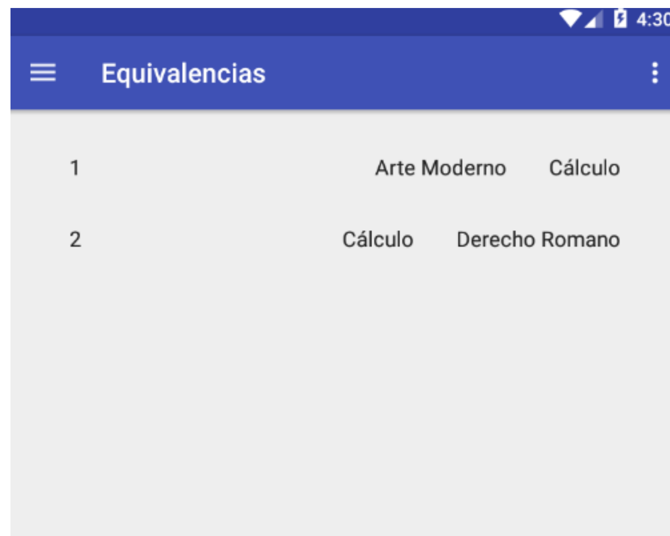

Figura 20: Pantalla de detalles de búsqueda

En la pantalla de detalles de búsqueda se listarán las asignaturas reconocidas para la búsqueda elegida. La asignatura buscada podrá aparecer indistintivamente a la izquierda o derecha de la lista.

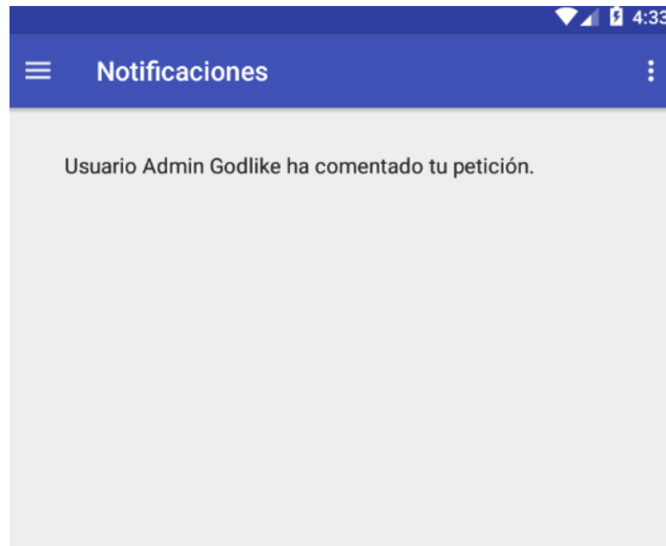

Figura 21: Pantalla de notificaciones

En la pantalla de notificaciones podrá ver si ha habido actividad en sus peticiones y que usuario ha sido. En el caso de que acceda a este menú y no tenga notificaciones, le aparecerá la pantalla que se muestra en la Figura 22.

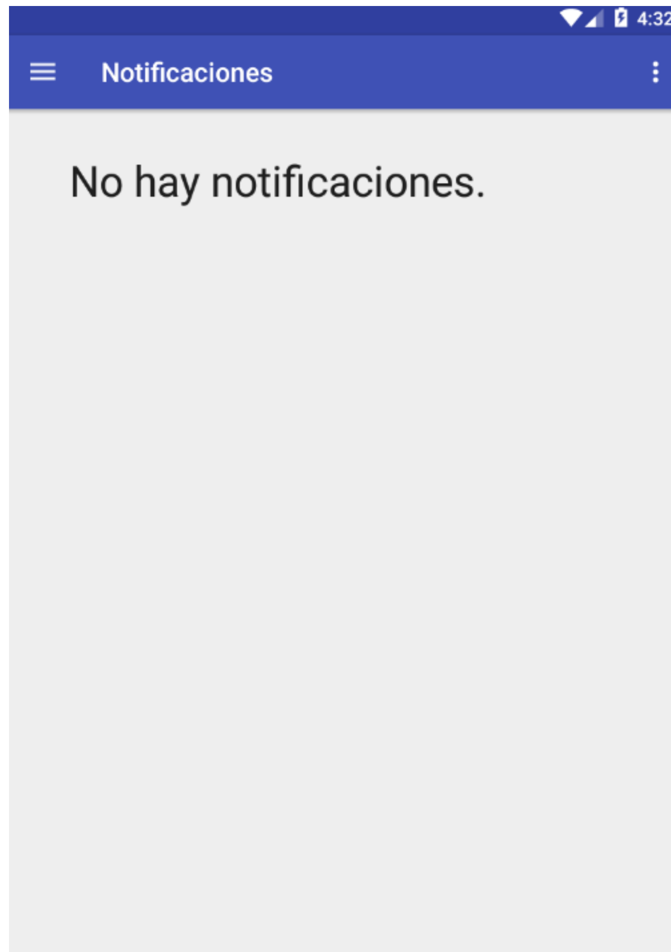

Figura 22: Pantalla de notificaciones vacía

A.2 Manual de despliegue

La jerarquía de repositorios es la siguiente: hay un primer repositorio llamado *willyfog* el cual contiene documentación, *scripts* y en definitiva los archivos necesarios para establecer el entorno de desarrollo. Dentro, podemos encontrar la carpeta *projects* donde tendremos que situar el resto de repositorios para que el despliegue sea exitoso.

Una vez comentado esto, pasamos a detallar las instrucciones:

1. Entorno Vagrant

1. *git clone* el repositorio principal:

```
$ git clone https://github.com/popokis/willyfog.git ~/willyfog
$ cd ~/willyfog
```

2. Posteriormente, *git clone* el resto de repositorios que conforman el proyecto:

```
$ git clone https://github.com/popokis/willyfog-api.git
    projects/willyfog-api
$ git clone https://github.com/popokis/willyfog-openid.git
    projects/willyfog-openid
$ git clone https://github.com/popokis/willyfog-web.git
    projects/willyfog-web
```

3. Por ultimo, arrancar la máquina *Vagrant* y conectarnos a ella por *ssh*:

```
$ vagrant up
  [...]
$ vagrant ssh
  [...]
```

Una vez establecida la maquina de desarrollo, vamos a proceder con el despliegue de cada una de las partes del sistema.

2. willyfog-api

Conectando a la base de datos

Para conectar a la base de datos puede que te encuentres con dos opciones:

- Si despliegas la API con un *jar* (usando alguna herramienta como *sbt-assembly* y ejecutándolo con *java* **dentro del entorno *Vagrant***. En este caso tendrás el servidor *MySQL* de manera local, así que no tendrás problema en conectarte.

- El caso en el que ejecutes el *jar* desde **fuera del entorno *Vagrant***, o más probablemente, porque estés desarrollando con un IDE como *IntelliJ* el cual se encarga de desplegar el archivo *jar* y lanzarlo por si mismo. En este caso tendrás problemas para alcanzar el servidor *MySQL* ya que **no lo tendrás de manera local**.

Creando un tunel SSH

No hay problema, vamos a crear un tunel *ssh* de manera que puedas conectarte a la base de datos desde fuera del entorno *Vagrant*. Ejecute este comando **desde la máquina host**:

```
$ ssh -f vagrant@192.168.33.10 -L 3307:localhost:3306 -N
```

Así, a partir de este momento tendrás al servidor *MySQL* escuchando localmente en el puerto 3307. Si quieres conectarte desde el *cli* de *MySQL*, tan solo tendrás que ejecutar **desde la máquina host**:

```
$ mysql -h 127.0.0.1 -P 3307 -uroot -proot
```

Usando *sbt dist*

Ya que es realmente simple, usamos *sbt dist* para construir nuestro ejecutable:

```
$ cd ~/willyfog/projects/willyfog-api
$ sbt
  [...]
  > dist
  [...]
  [success] Total time: XXXX s, completed Xxx XX, XXXX X:XX:XX PM
  > exit
```

Y una vez que ha terminado, tendrás un archivo *zip* que contiene los ejecutables en la ruta *target/universal/willyfog-api-1.0.zip*:

```
$ cd target/universal
$ unzip willyfog-api-1.0.zip
$ cd willyfog-api-1.0
```

Pudiendo ejecutarlo con tu *application secret* preferido:

```
$ bin/willyfog-api -Dplay.crypto.secret=abcdefghijk -Dhttp.port=7000 &
```

3. willyfog-openid

1. Instalar las dependencias.

```
$ cd ~/willyfog/projects/willyfog-openid
$ composer install
```

2. Establecer el archivo *constants.php*. Si no estas en producción, simplemente puedes renonmbrar el fichero *constants.php.example* a *constants.php*:

```
$ cp app/constants.php.example app/constants.php
```

3. Generar las claves públicas y privadas:

```
$ openssl genrsa -out data/privkey.pem 4096
$ openssl rsa -in data/privkey.pem -pubout -out
    data/pubkey.pem
```

4. willyfog-web

1. Instalar las dependencias.

```
$ cd ~/willyfog/projects/willyfog-web
$ composer install
```

2. Establecer el archivo *constants.php*. Si no estas en producción, simplemente puedes renonmbrar el fichero *constants.php.example* a *constants.php*:

```
$ cp app/constants.php.example app/constants.php
```

3. Enlazar la clave pública del servidor OpenID (para poder manejar los JWT):

```
$ cd data
$ ln -s ../../willyfog-openid/data/pubkey.pem
```

5. willyfog-mobile

Dado que desplegar la aplicación móvil para un usuario normal no es una tarea sencilla, se ha decidido que sea el usuario el que pida la aplicación y recibirá un APK firmado para que lo instale en su dispositivo. Teniendo en cuenta que esta aplicación es libre y que no se va a lucrar nadie de ella, se permite la libre difusión de la misma.

6. Dominios

Recuerda añadir los distintos dominios al archivo */etc/hosts*:

```
$ echo "192.168.33.10   willyfog.com api.willyfog.com
    openid.willyfog.com" | sudo tee -a /etc/hosts
```

A.3 Manual API

En cuanto al manual de la API, se encuentra disponible de forma online en la siguiente URL,

`http://popokis.github.io/willyfog-api/`

Dado que habíamos usado Swagger como herramienta para generar la documentación, y poder poner a disposición de todo el mundo las posibles funcionalidades de la API, hemos querido que no este incrustada en esta memoria y pueda ser accesible desde cualquier dispositivo. Se ha puesto en una página estática en la raíz del repositorio del proyecto *Willyfog-API*, así además de poder estar al alcance de cualquiera, también animamos a los usuarios a colaborar y hacer *pull requests* para ir mejorando la documentación e ir ampliándola.