EDUCAPP

Aplicación de gestión educativa

Proyecto de fin de ciclo.
C.F.G.S. Desarrollo de Aplicaciones Multiplataforma

Harold Hormaechea Garcia
Harold.hormaechea@gmail.com

Contenido

Propuesta

Este proyecto propone la creación de una herramienta de gestión de centros de estudios de fácil manejo por parte tanto de responsables de centro, profesores, alumnos, y tutores legales de éstos. El objetivo es la creación de un conjunto cliente – servidor encargado de mantener de forma persistente los detalles tanto de los cursos impartidos por parte del centro que gestione el servicio, como de las evaluaciones de los alumnos y sus controles de presencia y calificaciones.

Nuestros responsables de centros tendrán la posibilidad de registrar los cursos que sus instituciones imparten. Esto implica poder tanto otorgarles identificadores internos, como los nombres oficiales y las descripciones de dichos cursos. También se considera de importancia los aforos máximos de los cursos y las fechas de inicio y finalización, por lo que ellos serán los responsables de informarlas adecuadamente mediante una interfaz en el cliente apropiada para ello. Además, ellos (o personal autorizado por ellos) podrán incluir las matrículas de aquellos alumnos que sean asignados a los cursos, de modo que en cualquier momento se pueda identificar qué alumnos pertenecen a qué cursos en un año dado, pudiendo así acceder a todos sus datos de forma inmediata y sin ambigüedades.

Por otra parte, los profesores tendrán dos posibilidades principales. La primera, es pasar controles de presencia a diario. No será una función obligatoria a desempeñar por el profesorado, pero de ser empleada, permitirá generar alertas en caso de faltas de asistencia tanto a padres (o tutores legales) registrados en la aplicación como al propio alumno que se ausente. Además, también se les brindará la posibilidad de incluir las correcciones a entregas o exámenes en la aplicación, junto con archivos adjuntos. Entre estos archivos podrían encontrarse imágenes (o documentos) con las correcciones de exámenes, documentos de apoyo a alumnos, etcétera.

Los alumnos tendrán acceso a su lista de exámenes y cursos en los que hayan sido registrados. Recibirán notificaciones cuando las correcciones de sus exámenes estén en línea (tras haber sido subidas al servicio por los profesores), y también cuando tengan alguna falta de asistencia. Esto permitirá al alumno ser consciente de cualquier notificación generada por el profesorado hacia él, y mantener el listado de sus calificaciones del año accesible.

El último de los perfiles de usuario con el que nos encontramos es el de los tutores legales. Éstos recibirán las mismas alertas que los alumnos que estén registrados 'bajo su cargo', de modo que de forma inmediata puedan recibir tanto las alertas de calificaciones subidas, como aquellas de absentismos injustificados, lo cual les permitirá tomar medidas inmediatas ante cualquier tipo de problema disciplinario o educativo de sus hijos o estudiantes tutelados.

Finalmente, todos los usuarios podrán disponer de perfiles personales que incluirán (de querer el usuario) una foto de perfil, entre otros detalles. Inicialmente, esos usuarios deben ser registrados manualmente en el sistema por un administrador, pero en el futuro el registro se realizará por parte de los mismos usuarios y un gestor de centro podrá otorgar los privilegios pertinentes.

Por último, todos los usuarios tendrán un perfil de usuario público accesible por los demás, con una pequeña descripción personal que podrán modificar.

Resumen de funcionalidad y tecnologías empleadas

La implementación del proyecto es, necesariamente, en dos partes. La primera es un servidor encargado de gestionar, de forma concurrente y segura, las comunicaciones y solicitudes creadas en relación a las operaciones permitidas para los cuatro tipos de usuarios principales del servicio: gestores de centros, profesores, alumnos, y tutores legales. La segunda será un cliente orientado inicialmente a la plataforma Android que gestionará las operaciones de cara al usuario.

Nos encontramos, debido a los datos sensibles potencialmente visibles de los usuarios, ante una situación en la que la seguridad será algo crítico. Por este motivo se ha decidido la implementación de las comunicaciones mediante el protocolo HTTPS sobre un servicio REST-like, de modo que ataques que impliquen la lectura de paquetes en transporte no sean eficaces. El resto de características son propias de los dos elementos, como se detalla a continuación.

Para el cliente se ha empleado Android, con un min-sdk para la API19. Se utiliza RetroFit para adaptación de la interfaz EducappAPI en forma de API REST que es empleada por un servicio en segundo plano para la conexión y transferencia de información entre cliente y servidor. Respecto a las interfaces, una Activity se encargará de realizar tareas de conexión, y otra del resto de tareas. Esa segunda tarea (MainActivity) se compone de un DrawerLayout con una lista de acciones realizables por el usuario, y fragmentos intercambiables en el panel de contenido a la derecha.

El servidor, basado también en Java, emplea el framework Spring para la creación de un servicio en nube que se encarga de autenticar a usuarios y clientes, almacenar datos, y proveer la interfaz pública mediante su API para las conexiones mediante protocolos seguros (HTTPS, OAuth2).

Cliente

El cliente que operará con nuestro servicio es una aplicación para Android con la API 19 como API mínima durante el desarrollo. Para las comunicaciones con el servidor, se aprovechará la API pública de este junto con las librerías RetroFit que nos permitirán emplear la API como una interfaz estándar de Java, para que cualquier operación con el servidor se realice de forma estandarizada, coherente, y segura, siguiendo los protocolos HTTPS.

El cliente dispone de una serie de pantallas que permitirán al usuario acceder a las diferentes funciones expuestas por el servicio. La pantalla principal será de acceso al servicio, y presentará al usuario con la opción tanto de acceder al servicio introduciendo sus credenciales, como de registrarse como nuevo usuario *(pendiente de implementación)*. El resto de funciones dependerán del perfil de acceso que tenga el usuario.

Al introducir sus credenciales, la actividad de Login transmite sus datos a un *BoundService* a través de *Messenger* como enlace IPC. En ese momento, el servicio hace intento de conexión para obtener el *token* de sesión, y de ser exitoso, recupera la lista de privilegios para el usuario, que es devuelta a la actividad de Login. En ese momento, esta finaliza.

Una vez se ha conectado, al usuario se le presenta un *DrawerLayout* lateral con su información básica en la cabecera (que incluye la foto de perfil y nombre, actualmente) junto con una lista de botones en función de sus privilegios de acceso, tal y como son informados por el servidor a través de la petición realizada previamente por el servicio. Cada uno de estos elementos (*DrawerItem*) contienen información tanto del recurso textual que debe emplearse en ellos (un vínculo a R.string para facilitar la localización en idioma según el idioma del dispositivo). Por otra parte, la clase contiene un método estático que permite generar los *Fragment* asociados a cada *DrawerItem* creado.

Cada vez que el usuario pulsa en uno de esos elementos del menú, se crea y puebla un fragmento asociado en la ventana de contenido. Algunos de estos fragmentos tienen funcionalidades adicionales según los privilegios del usuario: por ejemplo, un usuario al que se le permita generar controles de presencia tendrá acceso a un menú para ello al hacer una pulsación larga sobre un curso de la lista, y un usuario con privilegios para generar nuevos cursos tendrá acceso a un botón especial para ello que sólo le aparecerá a él.

Métricas:

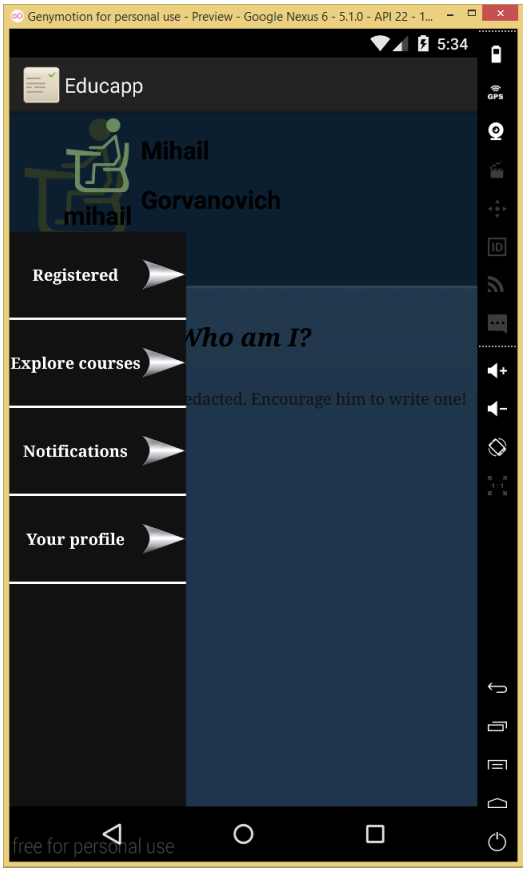
- Número de clases: 71
- Lineas de código no autogeneradas: 3480
- Métodos no autogenerados: 310

Interfaces

Todas las interfaces son de cara al usuario a través del cliente Android. Esto ha implicado el uso de objetos *Activity*, *Fragment*, *DrawerLayout*, interfaces en XML, y diversos elementos personalizables. A continuación explicaré el contenido de las interfaces de usuario implementadas en la aplicación hasta el momento, con una breve descripción de su funcionalidad y características.

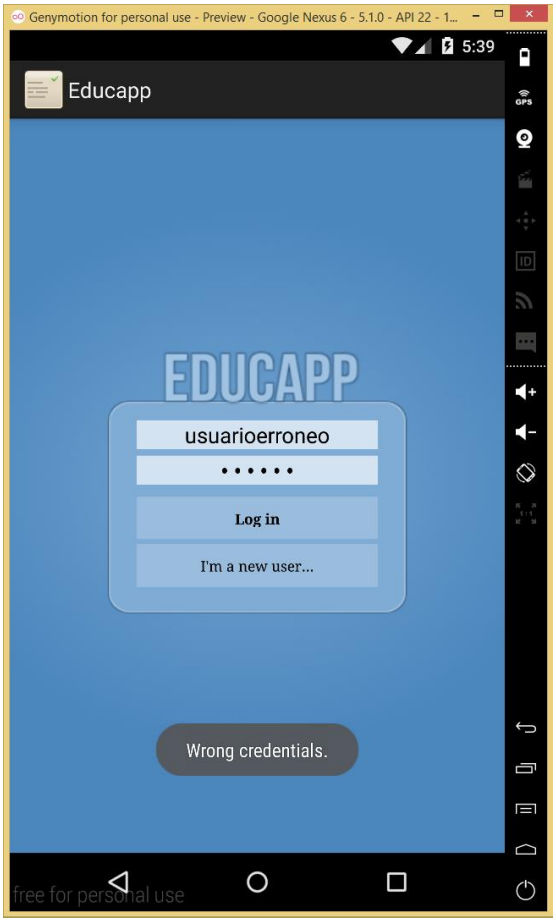
Actividad Principal (MainActivity.java)

Esta actividad está encargada de decidir si es necesario acceder a la pantalla de conexión de usuario, o si ya está conectado uno y sólo hay que recuperar sus vistas y datos. En una primera ejecución, se cargará *LoginActivity.java* con startActivityForResult() esperando como respuesta unos parámetros (lista de elementos de autenticación, nombre completo..). En caso de que ya haya un usuario conectado, o se haya regresado satisfactoriamente de la pantalla de Login, se vinculará al servicio en segundo plano (*NetworkOperationsService.java*) y se preparará para mostrar la primera pantalla que presentamos: el perfil personal del usuario. Además, en el *Drawer* lateral, se muestra la imagen de perfil (en este caso, un monigote), y el nombre del usuario, junto con la lista de actividades que puede realizar.



Actividad de Login (LoginActivity.java)

Esta actividad gestiona únicamente las operaciones de conexión con el servicio Spring remoto. Es lanzada por MainActivity, y pide un nombre de usuario y contraseña, con los que genera una petición síncrona al servidor mediante un *Message* y un *Handler* local de respuestas para IPC (Inter-Process communication). En función de si la conexión es exitosa o errónea, pasa de vuelta a MainActivity los datos del usuario o muestra un *Toast* con el error detectado: credenciales erróneas o errores de conexión. Esta deducción se realiza mediante las respuestas HTTP configuradas en el servidor Spring durante su desarrollo.



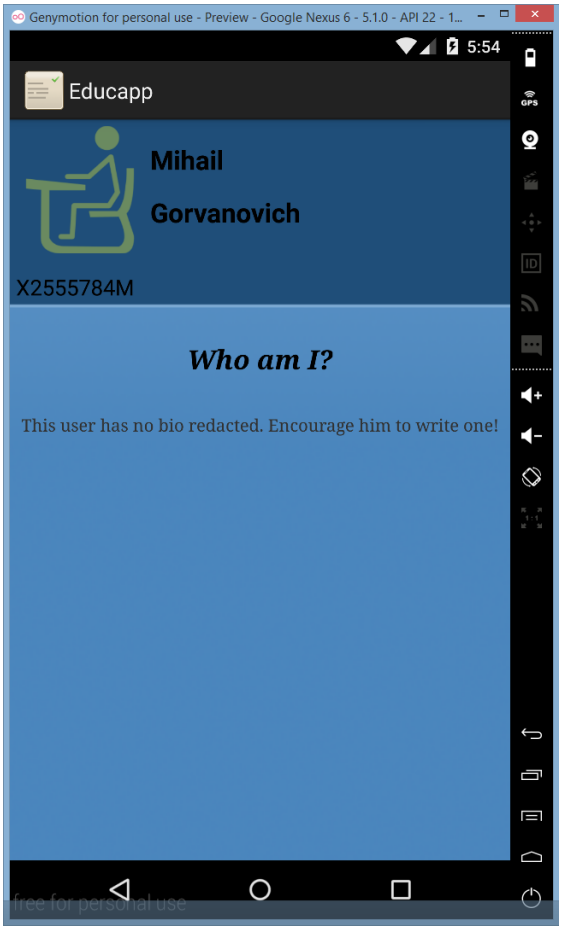
Fragmentos asociados a la Actividad Principal

Por cuestiones de eficiencia y usabilidad, se han empleado Fragments para la creación de vistas de contenidos en la aplicación. Esto permite un mejor tiempo de respuesta, además de una interfaz más fluida durante su manejo gracias al DrawerLayout facilitado por Android. Además, permitirá una vez implementado por completo la persistencia del estado del Fragment durante cambios de configuración. Esto no está actualmente configurado, pero es sencillo y ayudará a establecer, por ejemplo, diferentes Layouts para vistas horizontales o verticales en tablets, o cambios de resolución en las nuevas posibilidades de Android N respecto a vistas de aplicaciones en paralelo.

Estos Fragment definen la mayoría de actividades permisibles para el usuario, y son configurables en función de algunos criterios como los privilegios del usuario conectado actualmente. Además, contienen en algunos casos la lógica necesaria para conectar con el servicio (referencia al Binder IPC facilitada por la actividad principal) y obtener respuestas solo a las peticiones que ellos pueden realizar.

Fragment Perfil de Usuario (ProfileViewerFragment.java)

Esta interfaz es empleada tanto para mostrar el perfil de usuario cuando acaba de conectarse o lo solicita, como el de otros usuarios. El perfil público contiene la foto, nombre, apellidos, DNI y una pequeña descripción personal opcional. En caso de que el usuario no tenga una añadida, se muestra un mensaje estándar.

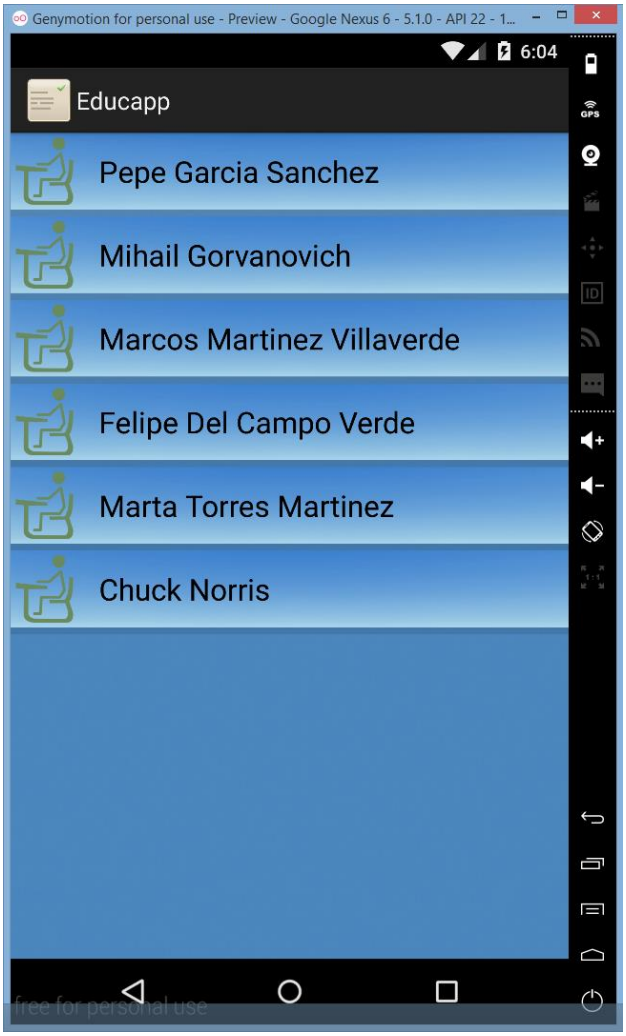


Esta clase tiene además dos posibles formas de ser cargada: Una es facilitándosele el *UserPublicProfile* que contiene la información que deseemos mostrar, y otra es facilitando un ID de perfil que solicitará al servidor a través del servicio en segundo plano. Emplea por tanto un Messenger local, junto con el Messenger del servicio provisto por *MainActivity*, y un *Message* pre-construido por la clase *ServiceMessageBuilder*.java con las constantes adecuadas.

Fragment Lista de Usuarios (UserListFragment.java)

Esta interfaz es empleada tanto para mostrar la lista de usuarios registrados a aquellos que puedan verla por sus privilegios. Además, mediante un click permite el acceso al perfil completo de dicho usuario (usando ProfileViewerFragment).

La lista de usuarios es descargada síncronamente a través de su adaptador personalizado *UserListAdapter.java* mediante el servicio en segundo plano (*NetworkOperationsService*), usando mensajes y handlers tal y como hace el resto de fragmentos. Esta lista contiene una versión simplificada de los datos contenidos en los perfiles (sólo imagen, nombre y apellidos). También emplea el patrón *ViewHolder* para facilitar un scrolling continuo y fluido mediante vistas reciclables.

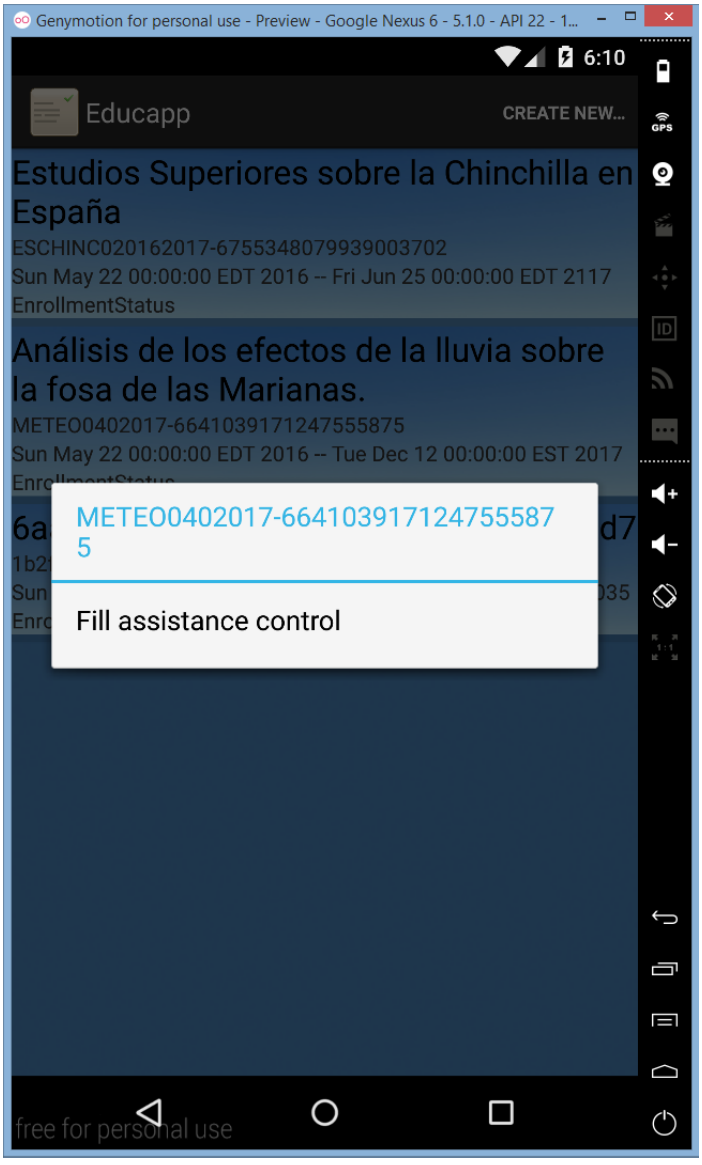


Fragment Lista de Cursos (CourseListFragment.java)

Esta interfaz es empleada tanto para mostrar la lista de cursos disponibles en el servicio, como facilitar las operaciones adicionales que algunos usuarios puedan realizar. Estas operaciones son, actualmente, la creación de nuevos cursos y la generación de controles de presencia.

El funcionamiento es similar a la interfaz que descarga las listas de usuarios. Un adaptador (*CourseListAdapter.java*) se encarga de alimentar de datos a esta lista mediante comunicación IPC con el servicio en segundo plano, y ésta emplea un *ViewHolder* para reciclar vistas y agilizar el funcionamiento. Además, adicionalmente, solicita a la actividad principal la lista de privilegios del usuario conectado. Si este puede añadir cursos nuevos, se habilita un botón en el menú superior para ello. Si, en cambio o adicionalmente, lo que puede es generar controles de presencia, al hacer un click largo en uno de los cursos se activará la opción para ello.

En el futuro se podrá incorporar el estado de matriculación en el curso junto con notas provisionales en esta vista.

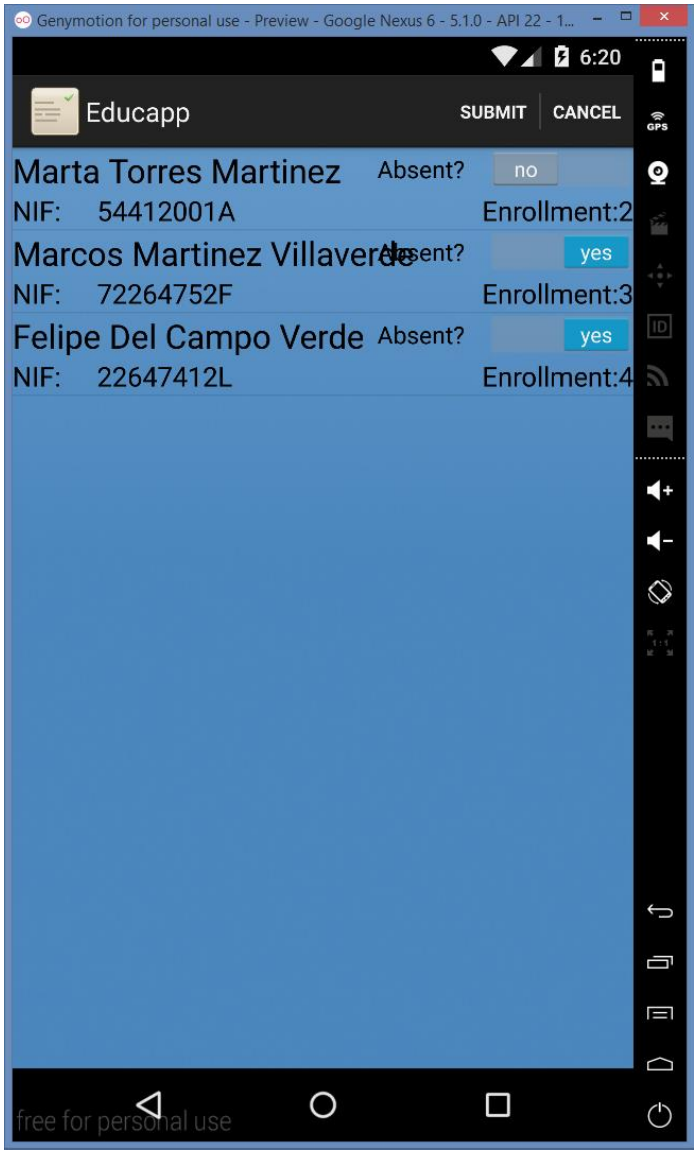


Fragment Control de Presencia (AssistanceFragment.java)

Esta interfaz es usada para solicitar al servidor modelos de controles de presencia para el curso seleccionado y subirlos posteriormente.

Se accede a ella a través de un menú contextual de la lista de usuarios (usando click largo). Descarga mediante el servicio en segundo plano la lista de matrículas asociadas al curso, y muestra una entrada por cada una de esas matrículas. Un Switch permite seleccionar si el alumno se encuentra o no en esa clase en el momento de realizar el control de presencia. Una vez se pulsa el botón inferior, esa lista actualizada es enviada para ser almacenada en el servicio*.

*El servicio genera notificaciones a los tutores legales del alumno en caso de absentismos injustificados.

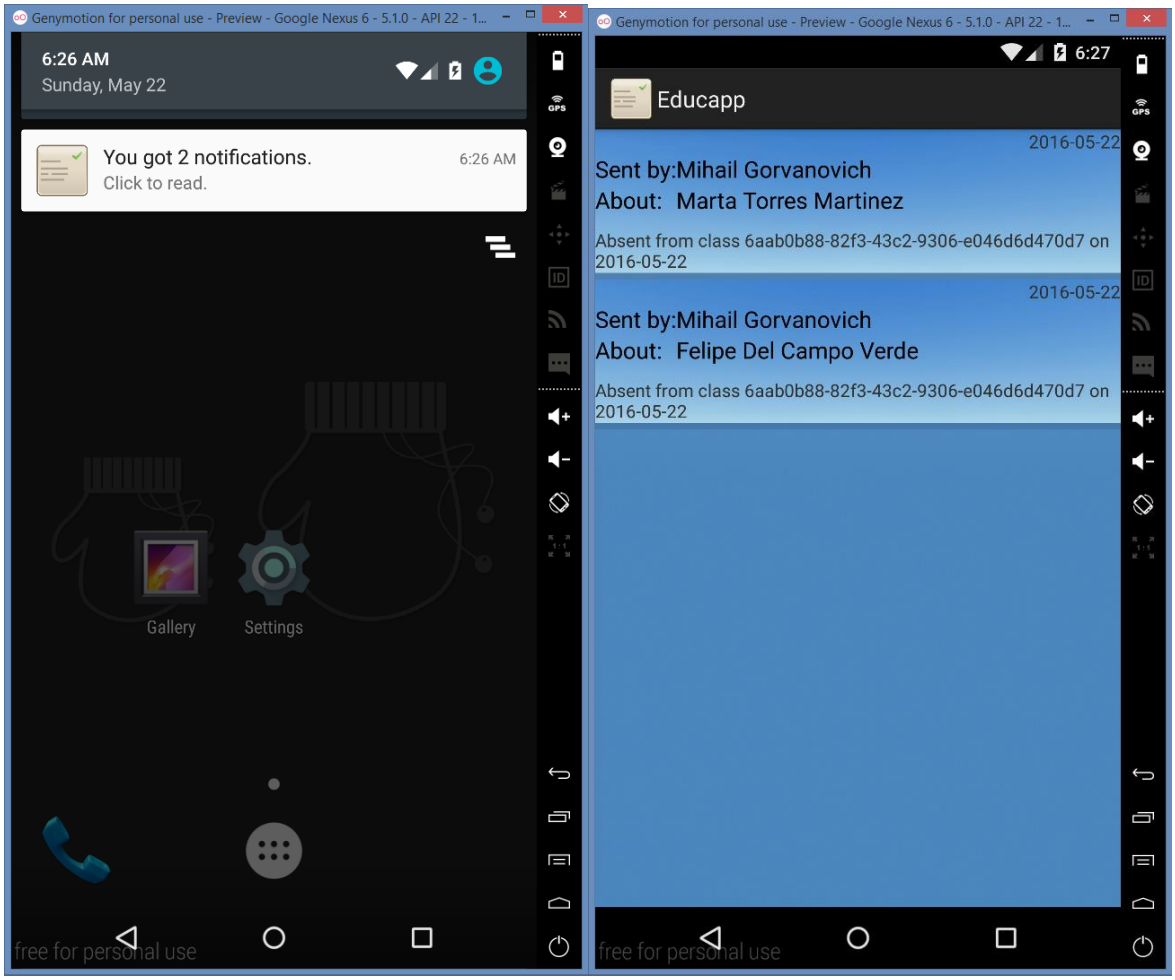


Fragment Notificaciones (NotificationListFragment.java)

Este último fragmento implementado se encarga de permitir al usuario ver la lista de notificaciones que tenga del profesorado, sean automáticas o personales. Por ejemplo, cuando un alumno tiene un absentismo injustificado, una notificación es generada a cualquier padre registrado en la aplicación como tal y vinculado a él.

Las notificaciones son recibidas por el servicio en segundo plano y mostradas. En el futuro se podrán marcar algunas como leídas para mejorar la usabilidad, pero de momento de leen todas en esta pantalla.

Por último, no sólo se puede acceder a ellas a través de aquí. En el caso de que se haya conectado un usuario a la aplicación y el servicio esté activo, se realiza un polling cada minuto al servidor (modo test) pidiendo la lista de notificaciones. De haber alguna, se envía una "notificación Android" al teléfono, que aparecerá en la barra superior. Pulsando en ella se abrirán las notificaciones nuevas.



Servicio en segundo plano

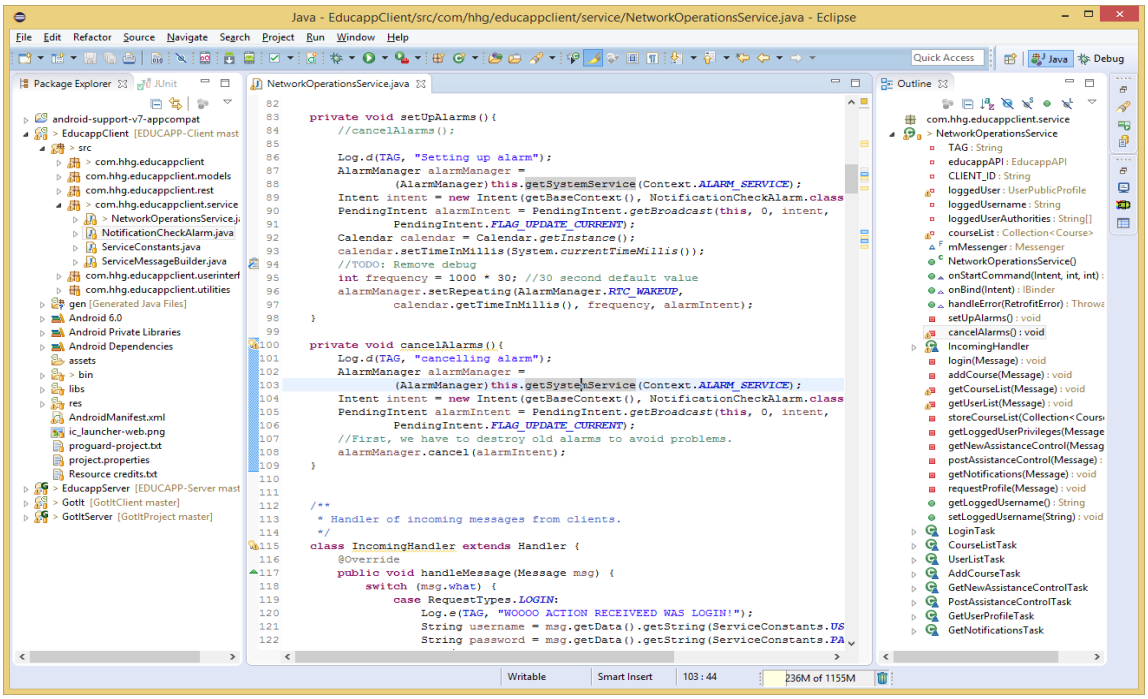
Android imposibilita la realización de operaciones de red en primer plano para evitar bloqueos en el hilo de ejecución de la interfaz de usuario que produzcan una sensación al usuario de ralentización o falta de respuesta. Por eso se obliga a la ejecución síncrona de operaciones de red y largas en muchos de sus componentes.

En este caso, el servicio en segundo plano se ha implementado en forma de *BoundService* al que las actividades se conectan y desconectan cuando inician y detienen su ejecución respectivamente. Una serie de tareas asíncronas en el emplean la API *EducappAPI.java* para conectarse con el servidor, y devolver los datos necesarios. *Messenger* vinculados a *Handler* son empleados en cada clase que haga uso del servicio (además del servicio en sí mismo) para gestionar la comunicación bidireccional.

Dado que se transfieren objetos pesados, y Android no permite esto en la barrera IPC mediante mensajes, se utiliza serialización mediante JSON para transmitir los datos en forma de Strings y arrays de Strings. Otra opción que sí permite la transferencia de objetos complejos es mediante AIDL, pero su funcionamiento es innecesariamente complejo y sobrecargado para este proyecto y fue descartado. Una opción futura es utilizar un almacenaje temporal en forma de base de datos o archivo temporal y enviar únicamente una URI a ese recurso.

El servicio tiene una función adicional, que es el configurar alarmas para el polling constante al servidor pidiendo notificaciones. De este modo el usuario recibe avisos en su teléfono de eventos que puedan ser de su interés sin necesidad de estar comprobando constantemente la aplicación. Estas alarmas son recibidas por un *BroadcastReceiver* ligero (NotificationCheckAlarm.java) que crea la notificación en caso de que se hayan recibido notificaciones nuevas.

Por último, se ha provisto de una clase estática utilitaria, ServiceMessageBuilder para construir mensajes *Message* al servidor facilitando parámetros sencillos. El servidor emplea internamente *Flags* (disponibles en *ServiceConstants.java*) para identificar y usar de clave en diccionarios de datos entrantes y de respuesta.



Servidor

Para facilitar la creación del servidor y poder cumplir todos los requisitos de seguridad y autenticación necesarios para una aplicación de este tipo, se ha decidido emplear java junto al framework Spring, que nos facilitará la labor de generación del servidor y la infraestructura y API REST que necesitamos para gestionar las conexiones entre cliente y servidor.

La seguridad será implementada mediante un contenedor basado en *Tomcat* incorporado en Spring, que se configurará para admitir sólo conexiones HTTPS mediante certificado firmado. Además, la autenticación de usuarios registrados se realizará mediante OAuth2. Esto último nos permite mantener un listado de usuarios junto con sus privilegios y estados de cuentas, junto al acceso mediante *tokens* de sesión, fácilmente gestionable y obtenible para poder garantizar que los datos sólo serán accesibles por aquellos usuarios y clientes (aplicaciones) que tengan los privilegios necesarios para ello. Spring, además, provee de anotaciones que nos permitirán realizar esto de forma sencilla, limitando el acceso a métodos concretos a determinados perfiles de usuario sin necesidad de alterar nuestra lógica de procesos.

El Servidor declarará una API que deberá ser respetada por cualquier cliente que desee hacer operaciones sobre él. La API será declarada en un POJO java a compartir con los clientes que se creen para interactuar con él, y definirá los diferentes REST *endpoints* accesibles por ellos.

Finalmente, para la persistencia de información se emplea JPA (Java Persistence API, Java EE) sobre una base de datos MySQL. JPA es un encapsulador sobre Hibernate que facilita las operaciones y transacciones sobre datos almacenados de forma segura y sencilla.

Métricas:

- Número de clases: 47
- Lineas de código no autogeneradas: 2893
- Métodos no autogenerados: 291

Adaptaciones sobre Spring realizadas

Spring es un framework muy potente que nos permite, rápidamente, crear servicios en red. No obstante hay muchas posibilidades de configurarlo, y son en gran parte necesarias para la generación de un servicio robusto.

En este caso se han realizado diversas modificaciones que facilitan el trabajo con el mismo y la interacción cliente-servidor con sus respectivas peculiaridades, que son descritas a continuación:

Autenticación según privilegios en métodos:

```
@EnableGlobalMethodSecurity(prePostEnabled = true, jsr250Enabled = true)
```

Al tener perfiles de usuarios con diferentes privilegios, se ha tenido que activar la funcionalid de Spring que permite definir el acceso a métodos en función de ellos (con métodos de apoyo en las clases User.java para acceder a estos datos). Esto supone una línea de control que evita que un usuario pueda acceder a un método (por ejemplo, el que provee la lista de cursos, o notificaciones) sin tener privilegios para ello.

OAuth2 para logado de usuarios y tokens de accesos

```
@Import(OAuth2SecurityConfiguration.class)
```

Uno de los requisitos básicos es la autenticación de usuarios y clientes de forma segura. Esta anotación nos permite definir una clase que cree una configuración de seguridad a aplicar. Además, configura un servidor embebido Tomcat con certificado de seguridad para HTTPS.

Persistencia mediante JPA

```
@EnableJpaRepositories(basePackageClasses = Repository.class)
```

Empleamos JPA, por lo que hay que definir las interfaces que se usarán para acceder a las tablas de la base de datos. Esto nos ayuda a definir un paquete desde el cual crearemos todas esas definiciones.

Personalización de controladores de usuarios y clientes

```
CustomizedCombinedDetailsManager.class
```

En este caso, tenemos que dejar claro a Spring como gestionamos usuarios y clientes, cuales tienen acceso a qué puntos, y en qué casos un usuario puede acceder a una ruta REST sin conexión (sólo aplicable, actualmente, a aquellos que quieren registrarse, que evidentemente necesitan poder acceder al endpoint de registro).

Puntos de acceso y controlador

Se han enlazado las diferentes rutas definidas en la API con métodos declarados en *BaseController*. Este controlador se encarga de comprobaciones iniciales, y deriva las operaciones sobre datos a un controlador de bases de datos declarado como *PersistentDataManager*. Por conveniencia, se han declarado dos controladores auxiliares: Uno para notificaciones (*NotificationManager*) y otro para datos de usuarios (*UserDataManager*). De este modo se separan componentes que no tienen por qué estar vinculados y se facilita que en el futuro puedan ser modificados o sustituidos por completo sólo conservando una interfaz común básica.

Un ejemplo de método que requiere de privilegios de acceso para su ejecución, utiliza variables en ruta HTTP, y devuelve una respuesta con código HTTP y un cuerpo con un objeto serializado es el siguiente:

```
/**
 * Retrieves the current profile of an user.
 */
@RequestMapping(value=EducappAPI.GET_PROFILES_SINGLE, method = RequestMethod.GET)
@PreAuthorize(value = "hasRole('"+EducappAPI.AUTH_RETRIEVE_OTHERS_PROFILE+"')")
public @ResponseBody UserPublicProfile getUserProfile(
        @PathVariable(EducappAPI.USER_PUBLIC_ID) String username,
        HttpServletResponse reply){
    try{
        User user = (User)persistenceManager.getUser(username);
        reply.setStatus(HttpServletResponse.SC_OK);
        return user.getUserProfile();
    }catch(UsernameNotFoundException ex){
        try {
            reply.sendError(HttpServletResponse.SC_NOT_FOUND,
                    "The requested user was not found");
        } catch (IOException e) {}
        return null;
    }
}
```

Bases de datos

En el proyecto, la estructura de datos se encuentra por completo en el servidor. Éste gestiona una serie de tablas, creadas mediante Hibernate a través del framework Java Persistence API (JPA) de Java Enterprise, con la finalidad de simplificar y ayudar a la gestión de las tablas de objetos subyacentes, además de otorgarles una integridad referencial automatizada por este sistema, y permitir el acceso simple a los datos mediante interfaces Java. Este sistema ha sido elegido por proporcionar varias ventajas respecto a una implementación directa en base de datos:

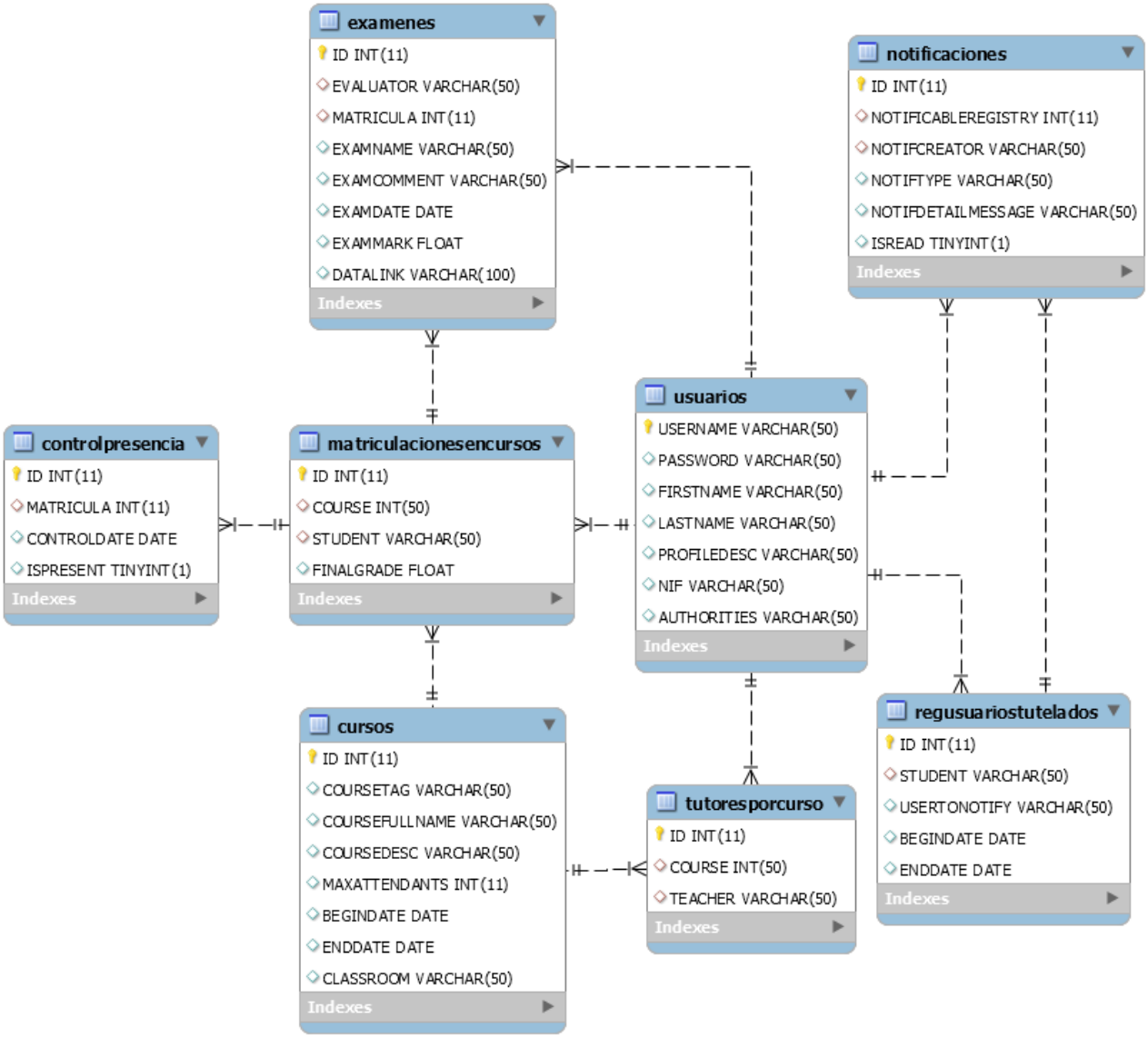
- La creación de tablas se automatiza mediante anotaciones en las clases de datos. Esto evita errores en la creación de las mismas e inconsistencias.
- Al quedar vinculados los objetos con el cómo son almacenados, se evita el tener que realizar conversiones o manipulaciones directas sobre ellos o sus referencias externas.
- JPA automatiza la creación de tablas auxiliares que contengan uniones de otros elementos sin necesidad de declaraciones explícitas, lo que agiliza la ampliación del número de tablas en el caso de desear añadir funcionalidades, y previene errores de integridad referencial o duplicidad de información.
- Permite realizar consultas simples tanto usando una serie de palabras clave en los nombres de métodos como consultas complejas usando consultas explícitas en un subconjunto del lenguaje HQL.

La solicitud de datos al sistema encapsulador JPA se realiza mediante repositorios CRUD de Spring, lo cual permite el uso de interfaces para crear una fachada que nos permita, de ser necesario, alterar el sistema de gestión de bases de datos empleado por nuestro servidor, sin necesidad de alterar la implementación lógica del programa en ningún otro aspecto que inyectándole la nueva clase gestora de persistencia que implemente dicha interfaz.

A continuación incluyo dos esquemas de la base de datos, uno conceptual simplificado y otro que incluye todas las adaptaciones realizadas por JPA/Hibernate en función de las relaciones existentes entre diferentes objetos.

Esquema de tablas simplificado:

Este esquema es el diseño original de la base de datos, previo a la adaptación realizada por JPA/Hibernate en implementación.



Esquema de tablas completo:

Este esquema es el resultado de la implantación sobre *MySQL* a través de JPA de las partes del servidor implementadas. Se puede observar que en algunos casos (*classroom*, *phonenumber*…) se han empleado *placeholders* temporales en espera de una implementación completa de su funcionalidad.