4 Restricciones

Tras haber realizado el análisis de requisitos y antes de continuar con la definición y planificación de las tareas del proyecto, se presenta esta sección para tener en cuenta las restricciones a las que se enfrenta el alumno a la hora de llevar a cabo el proyecto y que condicionarán positiva o negativamente a la planificación.

4.1. Tiempo

En este aspecto, existe una limitación inherente a la estancia en prácticas del alumno. El periodo definido en el plan de estudios es de 300 horas que, de acuerdo al horario acordado entre el alumno, el tutor y el supervisor, se traduce en, aproximadamente 3 meses de estancia. Por tanto, será importante ajustarse tanto como sea posible a este periodo y, gestionar y planificar las tareas de forma que puedan completarse el máximo número posible antes de que finalice la estancia. En el mejor caso, todas ellas podrían planificarse y terminarse dentro del plazo disponible.

4.2. Tecnología

A lo largo de su trayectoria, la empresa y sus trabajadores han adquirido experiencia y confianza en un conjunto específico de tecnologías que emplean para desarrollar e implementar sus productos. Para evitar conflictos desconocidos a la hora de integrar los nuevos servicios que se impementaran para *FlatCrew Server* con otros ya existentes, el alumno aprenderá y adoptará para este proyecto dichas tecnologías, se han listado anteriormente en la sección *Objetivos de Aprendizaje* 1.2.3, pag. 7. No es tanto una limitación sino una integración del alumno en el equipo de trabajo, así como una oportunidad de aprender de profesionales que trabajan día a día con estas tecnologías. Además, aporta seguridad a la totalidad del proyecto en el sentido de que garantiza una supervisión apropiada y guía hacia buenas prácticas de desarrollo al uso de éstas.

4.3. Experiencia

La experiencia con la que cuenta el alumno a la llegada a la empresa, es la que ha adquirido a través de los estudios de Grado en Ingeniería Informática y la obtenida gracias a su participación en una beca de colaboración a cargo del INIT (Institute of New Imaging Tecnologies). Gracias a

esta última, el alumno cuenta con cierta experiencia en el desarrollo de servicios RESTful y su integración con aplicaciones Android, aspecto que puede resultar muy interesante en el desarrollo de este proyecto y que puede favorecer los recursos temporales invertidos. Sin embargo, son escasos los conocimientos y práctica que posee en el uso de PHP, agravándose si consideramos que se ha de emplear el framework CakePHP. Esto afectará sin duda a la planificación puesto que el alumno deberá invertir parte del tiempo del que dispone en la estancia a la instalación, configuración y aprendizaje de este framework.

5 Planificación del proyecto

Una vez descrito el proyecto y las diferenets unidades funcionales con las que contará la aplicación, es necesario llevar a cabo un desglose en tareas que determinará de qué forma se lleva a cabo y cómo avanza el desarrollo de *FlatCrew Server*. Esto también quedará condicionado por la estimación de recursos y la planificación temporal de cada una de las tareas. En las secciones que se presentan a continuación, puede verse con más detalle toda la planificación.

5.1. Metodología y definición de tareas

Se empleará una metodología tradicional en cascada, desglosando y planificando temporalmente las tareas antes de comenzar con el desarrollo del proyecto. Sin embargo, debido al escaso tiempo que el alumno posee para llevar a cabo todo el diseño y desarrollo, así como a la velocidad con la que deberá ir cambiando el producto y adaptarse según las necesidades de la empresa y los cambios propuestos en *FlatCrew Client*, se llevarán a cabo revisiones cortas de los objetivos que se han alcanzado en cada momento y los recursos invertidos para replantear los objetivos futuros y conseguir de esta forma un desarrollo un tanto más flexible.

A continuación se lleva a cabo del desglose de tareas, acompañadas de una breve descripción:

Definir requisitos
El objetivo es conseguir una definición de requisitos generales de la aplicación, sin tener en cuenta la división entre *FlatCrew Server* y *FlatCrew Client*. Como punto de partida para llevar a cabo esta tarea se emplearán unos mockups de *FlatCrew Client* que la empresa proporcionará al alumno.

Estudiar tecnologías
Tal como se ha explicado anteriormente, hay tecnologías que el alumno no conoce o a las que necesita dedicarle cierto tiempo para desenvolverse con comodidad y seguridad suficiente para llevar a cabo la implementación de los servicios posteriormente. Además, también se requiere cierto tiempo para instalar y configurar correctamente la plataforma y entorno de desarrollo. Por último, el objetivo final de esta tarea es implementar un servicio básico de ejemplo empleando las tecnologías estudiadas y desplegarlo en el entorno que se ha instalado, comprobando que todo funciona correctamente, dejándolo así listo para comenzar la implementación.

Establecer pautas de trabajo y cooperación
Como algunas herramientas que se emplearán, por ejemplo Gerrit, requieren de cierta colaboración o interacción entre los desarrolladores, se pretende establecer una serie de

pautas y acuerdos de buen hacer para evitar detenciones o ralentizaciones. Para ello, se reunirán los desarrolladores implicados en el proyecto y discutirán cuál es la mejor forma de proceder.

Analizar casos de uso
Esta tarea consiste en estudiar y definir todas las circunstancias e interacciones que se producirán entre *FlatCrew Server* y *FlatCrew Client*. Quizás sea necesario discutir algunos aspectos relacionados con esta tarea con los desarrolladores Android de la empresa.

Analizar requisitos de datos
El trabajo que se debe realizar en esta tarea es identificar todos los datos que son relevantes para el funcionamiento de la aplicación y que será necesario almacenar en la base de datos. Más adelante, gracias a los resultados obtenidos al finalizar esta tarea, será más sencillo llevar a cabo el diseño de la base de datos.

Diseñar modelo de datos
Antes de comenzar con el diseño de la arquitectura del servicio, es imprescindible haber definido cuál será la estructura de la base de datos, qué tablas contendrá, qué campos y cómo se relacionan los datos entre ellos.

Diseñar arquitectura del servicio
Para conseguir una relación coherente entre las clases del servicio, se realiza un diseño inicial que organice el código para obtener un reparto y asignación correcto de las funcionalidades a cada entidad.

Implementar servicio de gastos comunes
Tras la finalización de esta tarea se dispondrá de las funciones necesarias en el servidor para crear gastos comunes y gestionarlos.

Implementar servicio de listas
Esta tarea consiste en implementar los métodos necesarios para dar soporte a la creación de listas y su actualización y seguimiento.

Implementar reparto de tareas
Supone implementar las funciones necesarias para gestionar las tareas vinculadas a una Crew. El procedimiento será muy similar a las dos tareas anteriore y se realizarán las mínimas comprobaciones, hay tareas dedicadas a pruebas más exhaustivas en etapas posteriores del desarrollo.

Integrar SMS
Para enviar los SMS de confirmación de registro, es necesario emplear una pasarela de envío de SMS. El servicio contratado por PaynoPain es BulkSMS [1], que proporciona un API que es necesario integrar en *FlatCrew Server*. Conseguir esta integración y un correcto envío de SMS a través del servicio es el objetivo de esta tarea.

Integrar OAuth 2.0
Para integrar OAuth 2.0 en el servicio, se optará por una solución económica en términos temporales y se empleará un plug-in desarrollado específicamente para CakePHP. Para acceder al repositorio donde se encuentra el código, ver [4]. Tras la finalización de esta

tarea, el sevicio debería ser capaz de llevar a cabo la gestión adecuanda de tokens propia de OAuth 2.0 y que serán necesarios para llevar a cabo las funciones relacionadas con el registro y el login.

Implementar servicio de registro y login

Esta tarea consistirá en implementar la lógica de registro y login haciendo uso de las herramientas integradas anteriormente, SMS y OAuth 2.0. Al finalizar esta tarea, el servicio será capaz de registrar a un nuevo usuario o entregar un token (una cadena que contiene los credenciales de seguridad y es opaca para el usuario) válido de autenticación en caso de que ya se haya registrado.

Integrar Android push notifications

Como la aplicación se lanzará en primer lugar para la plataforma Android, se dedica esta tarea a integrar el uso del API que proporciona Google para hacer uso de las notificaciones push a aquellos dispositivos que se hayan registrado para recibirlas. Los dispositivos se registráran en el servidor para recibir dichas notificaciones en el momento en que el usuario accede a la aplicación desde un terminal Android. Esto afectará también al resto de funcionalidades implementadas anteriormente ya que será necesario lanzar una notificación cada vez que se produzca un evento que se considere relevante para el usuario. Una vez completada esta tarea, el usuario debería recibir un aviso en la bandeja del sistema de su dispositivo cada vez que, por ejemplo, un usario cree un grupo del que él forma parte.

Completar gestión de contraseñas

Tras implementar los servicios de registro y login, quedarán pendientes algunas funcionalidades de gestión de las cuentas de usuario relacionadas en gran parte con las contraseñas. En esta tarea el alumno se dedicará a implementar las funciones que permitan, por ejemplo, reestablecer su contraseña en caso de que la hayan olvidado.

Integración monedero PangoPay

Con el objetivo de conseguir completar los pagos de forma automática a través de la aplicación, se integrará el servicio PangoPay que la empresa ya tiene en funcionamiento. Para ello, será necesario consultar a los desarrolladores de dicho servicio, las funcionalidades y capacidades de éste.

Probar integración con el cliente

Esta tarea es una de las más importantes ya que supone integrar los servicios de *FlatCrew Server* con *FlatCrew Client* y comprobar que interaccionan correctamente y que se han cubierto todas las funcionalidades que se habían propuesto para la aplicación. Sin duda, esta tarea se llevará a cabo conjuntamente con el equipo de desarrollo Android de la empresa.

5.2. Planificación temporal de las tareas

Para ilustrar de forma clara la planificación de las tareas, y también para poder llevar a cabo un seguimiento del progreso de forma más cómoda a lo largo del proyecto, se ha empleado un diagrama de Gantt, ver Figura 5.1 y Figura 5.2. Como puede observarse, al dedicarse el alumno

Nombre	Fecha de inicio	Fecha de fin
Planificación	3/3/14	17/3/14
Definir requisitos	3/3/14	5/3/14
Estudiar tecnologías	6/3/14	14/3/14
Leer documentación CakePHP	6/3/14	7/3/14
Leer documentación Gerrit	10/3/14	10/3/14
Instalar y configurar entorno de desarrollo	11/3/14	11/3/14
Implementar servicio ejemplo	12/3/14	14/3/14
Establecer pautas de trabajo y cooperación	17/3/14	17/3/14
Análisis	18/3/14	21/3/14
Analizar casos de uso	18/3/14	19/3/14
Analizar requisitos de datos	20/3/14	21/3/14
Diseño	24/3/14	28/3/14
Diseñar modelo de datos	24/3/14	26/3/14
Diseñar arquitectura del servicio	27/3/14	28/3/14
Implementación	31/3/14	21/5/14
Implementar servicio de gastos comunes	31/3/14	8/4/14
Implementar servicio de listas	9/4/14	17/4/14
Implementar reparto de tareas	18/4/14	28/4/14
Integrar OAuth	29/4/14	30/4/14
Integrar SMS	2/5/14	2/5/14
Implementar servicio de registro y login	5/5/14	9/5/14
Integrar Android push notifications	12/5/14	13/5/14
Completar operaciones de contraseñas	14/5/14	15/5/14
Integración monedero PangoPay	16/5/14	21/5/14
Pruebas	22/5/14	28/5/14
Probar integración con el cliente	22/5/14	26/5/14
Probar recuperación de contraseñas	27/5/14	27/5/14
Probar integración con PangoPay	28/5/14	28/5/14

Figura 5.1: Gantt: Desglose de tareas y fechas

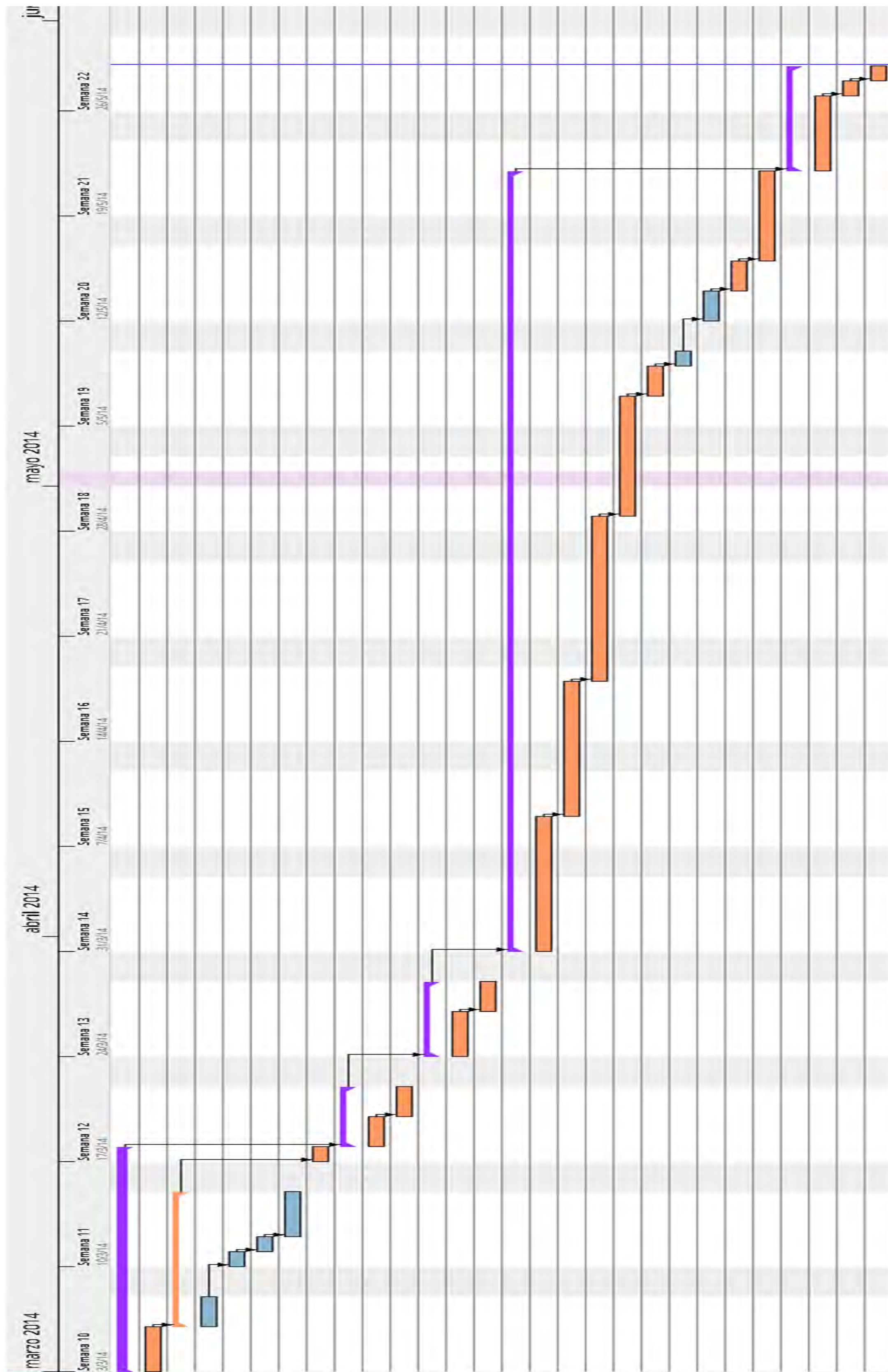

Figura 5.2: Gantt: Diagrama de barras

de forma exclusiva a *FlatCrew Server* cada una de las tareas que se presentan, dependen de la tarea anterior, por tanto, el desarrollo de todas las tareas avanza de un modo estrictamente secuencial.

5.3. Estimación de recursos del proyecto

Para estimar el coste del proyecto, se tienen en cuenta dos tipos de recursos. En primer lugar, se calculará el coste del equipo empleado para llevar a cabo el desarrollo y, en segundo lugar, el coste de personal para *FlatCrew Server*, dejando a parte los gastos derivados del desarrollo de *FlatCrew Client* ya que no supone la parte principal del desarrollo asignado al estudiante.

En la equación 5.1 puede observarse la expresión aplicada para considerar el gasto derivado del ordenador empleado por el alumno; recurso de infraestructura necesario para realizar el desarrollo. Se estima el valor del ordenador en 990 € y el valor residual en 4 años, 0 €. Las horas dedicadas al proyecto, como ya se ha mencionado anteriormente, son 300 en total. Tomando un plazo de amortización de 4 años, el gasto se calcula según la ecuación 5.2.

$$\text{Gasto ordenador} = \frac{\text{Valor de compra} - \text{Valor residual}}{\text{Plazo de amortización}} \times \text{Número de horas} \tag{5.1}$$

$$\text{Gasto ordenador} = \frac{990\,€ - 0\,€}{35063{,}3h} \times 300h = 8{,}4703\,€ \tag{5.2}$$

Por otro lado, queda el gasto del personal dedicado a la implementación del servicio. En este caso, sólo el alumno se dedica a la implementación de *FlatCrew Server* pero tendremos también en cuenta a los trabajadores de la empresa que dedicaron algunas horas a poner a punto los sistemas de control de versiones y revisión de código, ayudando también al alumno a seguir unas buenas pautas de trabajo. Para calcular el coste del personal emplearemos la expresión matemática que refleja la ecuación 5.3.

$$\text{Gasto personal} = \text{Horas de trabajo} \times \text{precio hora} \times 1{,}3 \tag{5.3}$$

Se estima el precio hora del alumno a 10 €, teniendo en cuenta que se considera programador novel, en la ecuación 5.4 puede observarse el gasto asociado al alumno, en caso de que este hubiera prestado los servicios en calidad de trabajador y no de estudiante en prácticas. Para los empleados que colaboraron con el alumno se han estimado un total de 50h de tiempo dedicado al proyecto y un precio por hora de 15 €, puede observarse el resultado del cálculo para esta parte en la ecuación 5.5.

$$\text{Gasto personal (alumno)} = 300 \times 10\,€ \times 1{,}3 = 3900\,€ \tag{5.4}$$

$$\text{Gasto personal (empleados)} = 50 \times 15\,€ \times 1{,}3 = 975\,€ \tag{5.5}$$

Así pues, la ecuación 5.6 muestra la suma total de los gastos derivados de los recursos previstos para este proyecto.

$$\text{Gasto total} = 8{,}4703\,€ + 3900\,€ + 975\,€ = 4883{,}4703\,€ \tag{5.6}$$

Para terminar esta sección, sólo queda mencionar que se ha considerado que, de media, el gasto del personal aumenta un 30 % respecto a la cantidad bruta que percibe el empleado, debido a la seguridad social y otros costes del empresario. Puede observarse este efecto en las ecuaciones de gasto personal en el último factor del producto.

6 Diseño del sistema

En este capítulo se presentan los aspectos que se han tenido en cuenta a la hora de realizar el diseño, así como la organización, interrelación y descripción de algunas de sus partes. Se comenzará presentando el diseño del modelo de datos sobre el que se apoyará el diseño del diagrama de clases que dará lugar a una primera visión de la arquitectura del servicio.

6.1. Diseño del modelo de datos

Para facilitar la comprensión de las relaciones entre los datos de la aplicación, se presentan 3 diagramas ER(Entidad-Relacion) separando las tablas y sus relaciones en bloques determinados por su ámbito de acción en la aplicación.

En primer lugar, en la Figura 6.1 se muestra el diseño del modelo de datos reduciendo el ámbito únicamente a aquellos datos que se generan como resultado del uso de las funcionalidades de *FlatCrew Client*, por ejemplo, crear un grupo, añadir un elemento a una lista, saldar una deuda, etc. Como puede observarse, no se ha pulido el modelo hasta el punto de eliminar los posibles bucles causados por las dependencias entre datos. Se ha hecho de esta forma puesto que CakePHP se encarga de resolver este aspecto, el *framework* gestiona por si solo la carga de dependencias evitando bucles.

En segundo lugar, se muestra el diagrama que representa las tablas necesarias para almacenar los datos correspondientes al *plug-in* de OAuth 2.0. Esta estructura viene determinada por cómo está implementado el complemento en sí y es, por tanto, un diseño poco flexible. Como puede verse en la Figura 6.2, la tabla *Users* que ya se había mostrado en el diagrama anterior, se relaciona con algunas tablas propias del complemento OAuth 2.0.

Por último, queda observar la Figura 6.3 donde se muestran los datos relacionados con la gestión de cuentas y notificaciones. De nuevo puede verse la tabla de usuarios relacionada con algunas de las entidades presentes en este diagrama. Sin embargo, existe una tabla que no necesita relacionarse con ninguna otra, almacena datos referentes a las peticiones de registro y, por tanto, no pueden asociarse a ningún usuario a priori. Al igual que en el caso de los datos temporales necesarios para recuperar contraseñas, los datos de registro son caducos y se borrarán de la base de datos pasado un periodo establecido. Más adelante, en los detalles de implementación pag. 61, se comentará cómo se llevará a cabo esta labor.

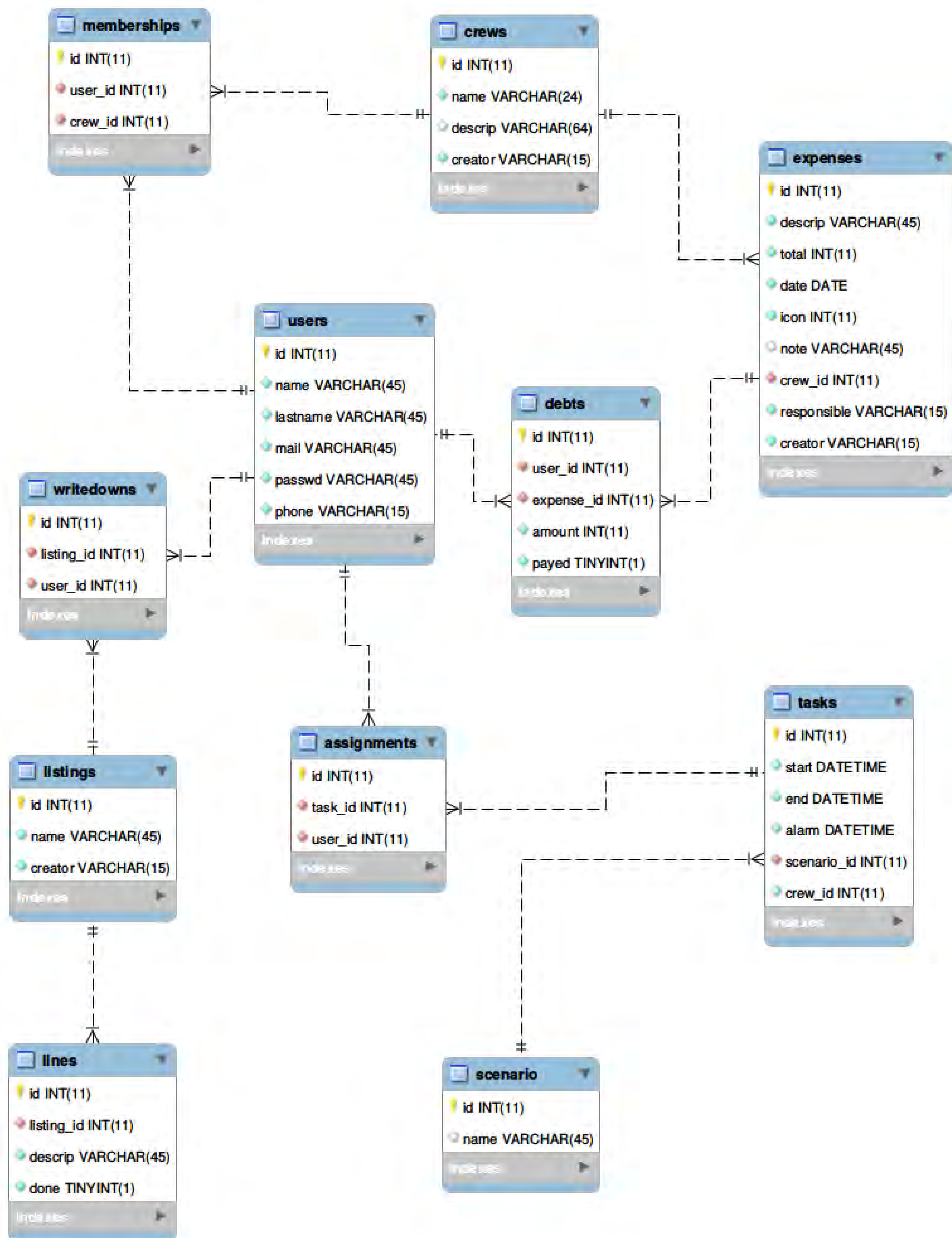

Figura 6.1: Diagrama Entidad-Relación: datos de usuario

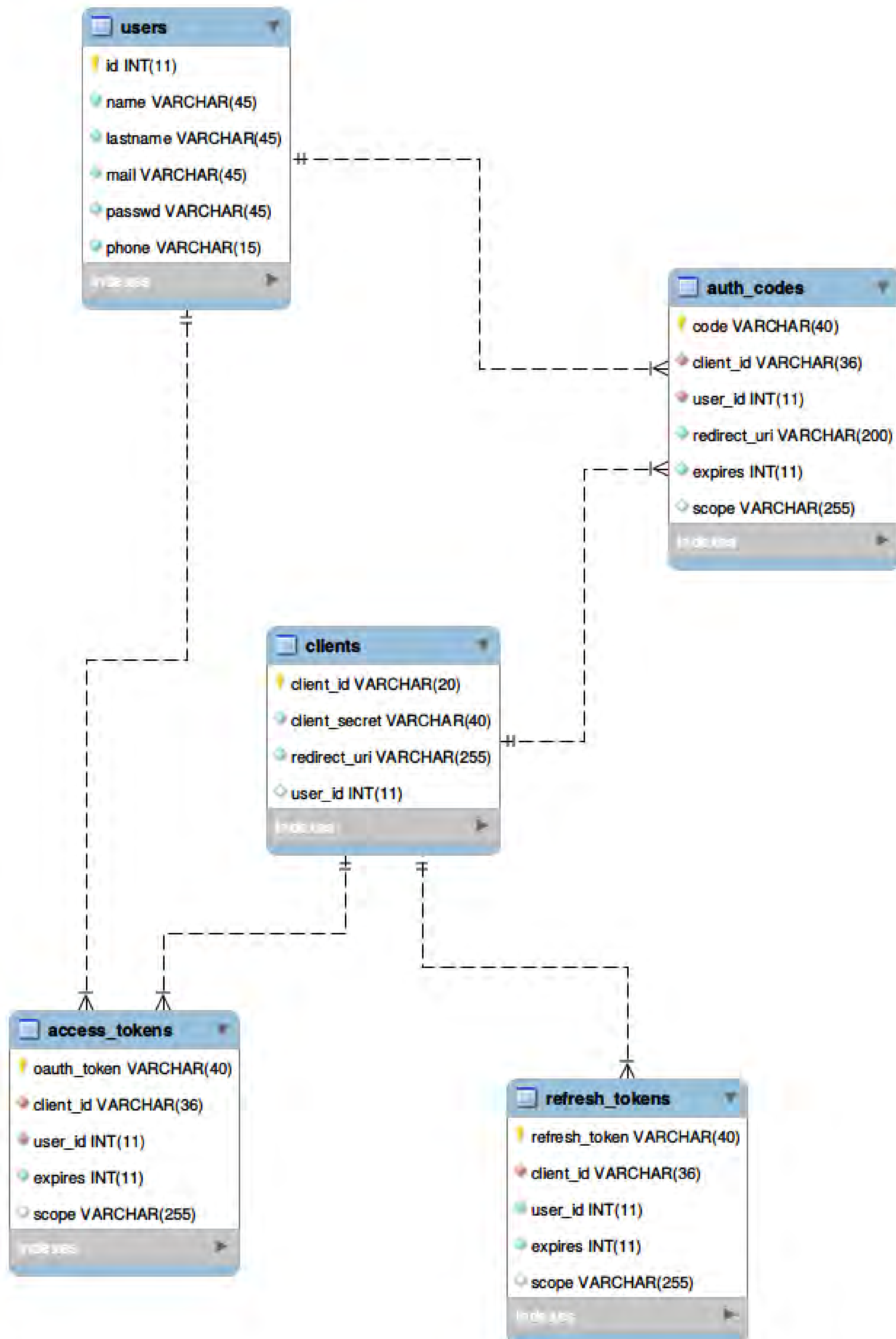

Figura 6.2: Diagrama Entidad-Relación: datos de OAuth 2.0

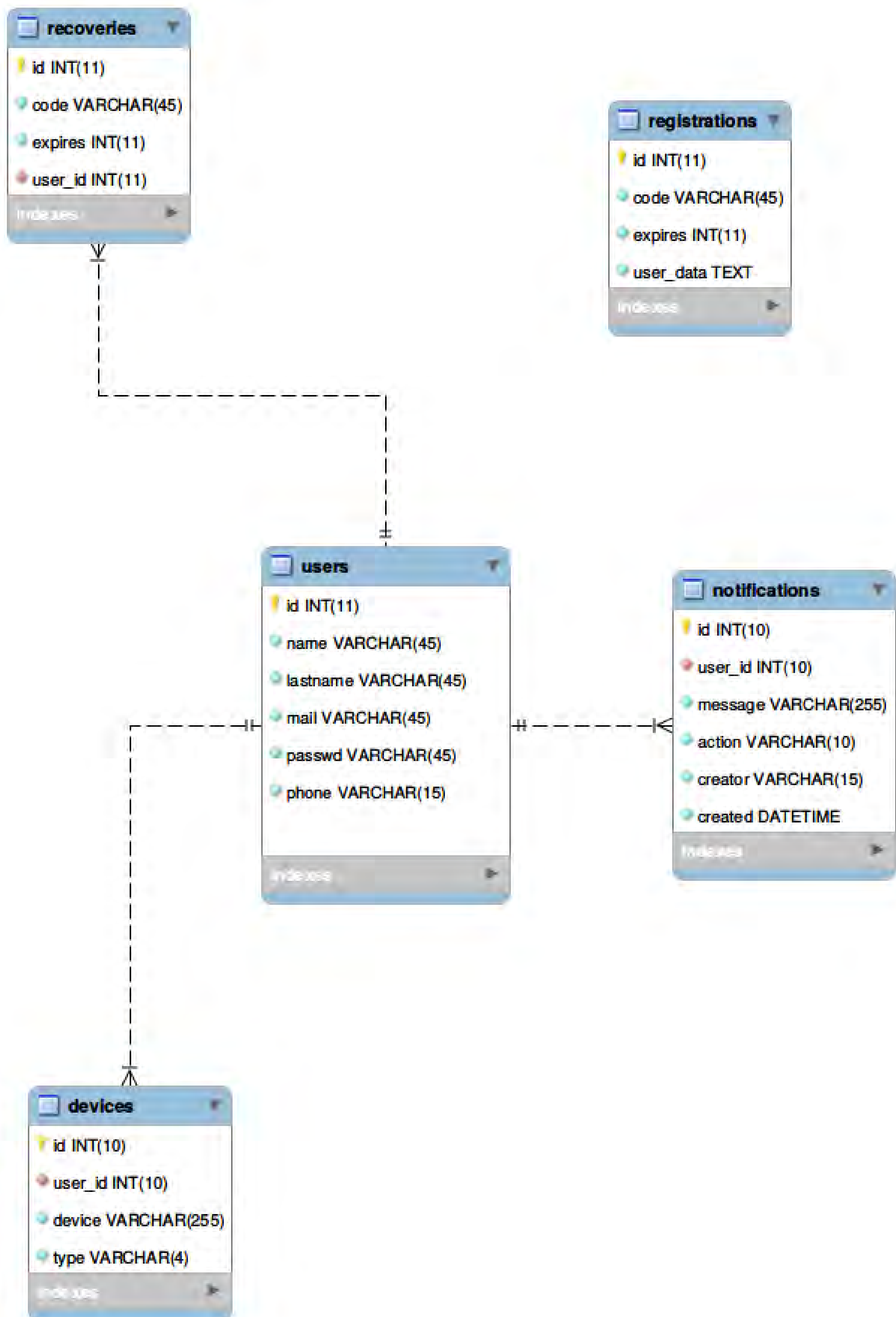

Figura 6.3: Diagrama Entidad-Relación: datos de gestión de cuentas y notificaciones

6.2. Arquitectura del servicio

Cuando se trata de definir la arquitectura de un servicio Web existen varias estrategias que se pueden seguir. Una de las más populares es acogerse al estilo de arquitectura REST (Representational State Transfer). Esta estilo de arquitectura hacer uso de los estándares Web (HTTP, URI , MIME *types*, XML, HTML, etc) para obtener un beneficio en aspectos del servicio Web como pueden ser: rendimiento, escalabilidad, simplicidad, fiabilidad y portabilidad.

A continuación se listan algunas de las características y principios de diseño más destacables de una arquitectura REST:

- *Client-Server*: La separación entre el servidor y el cliente es parte del modelo REST. Esto permite a ambas partes evolucionar de forma independiente.

- *Stateless*: No se retiene ningún tipo de información de sesión de cliente en el servidor. Cada petición contiene toda la información que se necesita para satisfacerla.

- *Identified Resources*: Todos los recursos que el servidor puede ofrecer son accessibles a través de una URI (Uniform Resource Identifier), que es asignada a cada uno de ellos como identificador.

- *Uniform Interface*: Se emplean los metodos HTTP (GET, POST, PUT, DELETE) como verbos para determinar las acciones que se llevan a cabo sobre los recursos.

- *Resource Based*: La abstracción del concepto referenciado por cualquier hiperenlace es el recurso.

- *Representational Oriented*: La representación de los recursos puede hacerse en diferentes formatos, por ejemplo en XML o JSON.

- *Layered System*: El sistema puede dividirse en varias capas de forma que la arquitectura puede estar compuesta por niveles jerárquicos de componentes, permitiendo un desacople que puede favorecer aspectos de la seguridad.

- *Hypermedia*: Gracias a que todos los recursos son identificados por una URI, pueden enlazarse entre ellos. A través de esta característica se puede conseguir que resulte más sencillo para el cliente descubrir el servicio, por ejemplo. Esto ocurre cuando el cliente obtiene nuevas URI en la respuesta a una petición y puede acceder a nuevos recursos a través de éstas, se denomina *Hypermedia As The Engine Of Application State* (HAETOAS).

Estos principios pueden seguirse de forma más o menos estricta según las necesidades del proyecto. No obstante, resulta conveniente no alejarse demasiado de estas premisas a la hora de desarrollar el servicio una vez se han identificado las ventajas que aporta al caso de uso en concreto.

Así pues, tomando como guía el estilo REST, se procede al diseño de la arquitectura del servicio. Para ilustrar cómo se engrana el conjunto de todas las funcionalidades necesarias, se

diseña un diagrama de clases que se irá mostrando a lo largo de esta sección.

En primer lugar, la Figura 6.4 presenta la clase AppController que es el eje central y la raíz de toda la lógica de la aplicación. Esta clase extiende a Controller que es una clase proporcionada por la propia biblioteca de CakePHP. Como puede observarse en dicha figura, también existen dos clases que tienen como padre a la clase Component que también es una clase propia del framework de desarrollo. Este tipo de clases, proporcionan un punto de partida y una funcionalidad básica que puede extenderse como se ha hecho en este caso. Además, en esta primera figura también puede observarse cómo la clase AppController emplea el *plug-in* OAuth 2.0 que se integrará en la aplicación, se importa como un *component* más aunque éste lo proporcione el *plug-in*.

A continuación, en la Figura 6.5 se muestran otras tres clases que extienden a Component y que serán empleadas concretamente por la clase UsersController como se muestra en la Figura 6.7. Más adelante, en este documento, en la sección 7.1.2 se explicará con más detalle el sentido y la implementación de estas clases que extienden a Component.

Por otro lado, AppController es el padre de varias clases en el diseño de este sistema. Como puede verse en la Figura 6.6, se han creado varios *controller*, uno para las operaciones relacionadas con cada entidad de la aplicación. De esta forma, la lógica queda mucho más organizada y se puede seguir la arquitectura REST con mucha más facilidad.

Finalmente, para proporcionar una visión global de cuál es la jerarquía de clases y cómo se relacionan éstas entre ellas, se emplea la Figura 6.8 que simplemente muestra cada una de ellas, sin el detalle de las operaciones o atributos para que sea más sencillo prestar atención únicamente a las relaciones.

Cuando se lleve a cabo la implementación del servicio, se intentará seguir esta arquitectura de clases para conseguir una separación entre los distintos propósitos a los que sirven cada una de ellas de forma que la posterior extensión, mantenimiento e inclusión de nuevas funcionalidades pueda llevarse a cabo sin grandes dificultades y sin distorsinar el funcionamiento de otras partes del software no relacionadas.

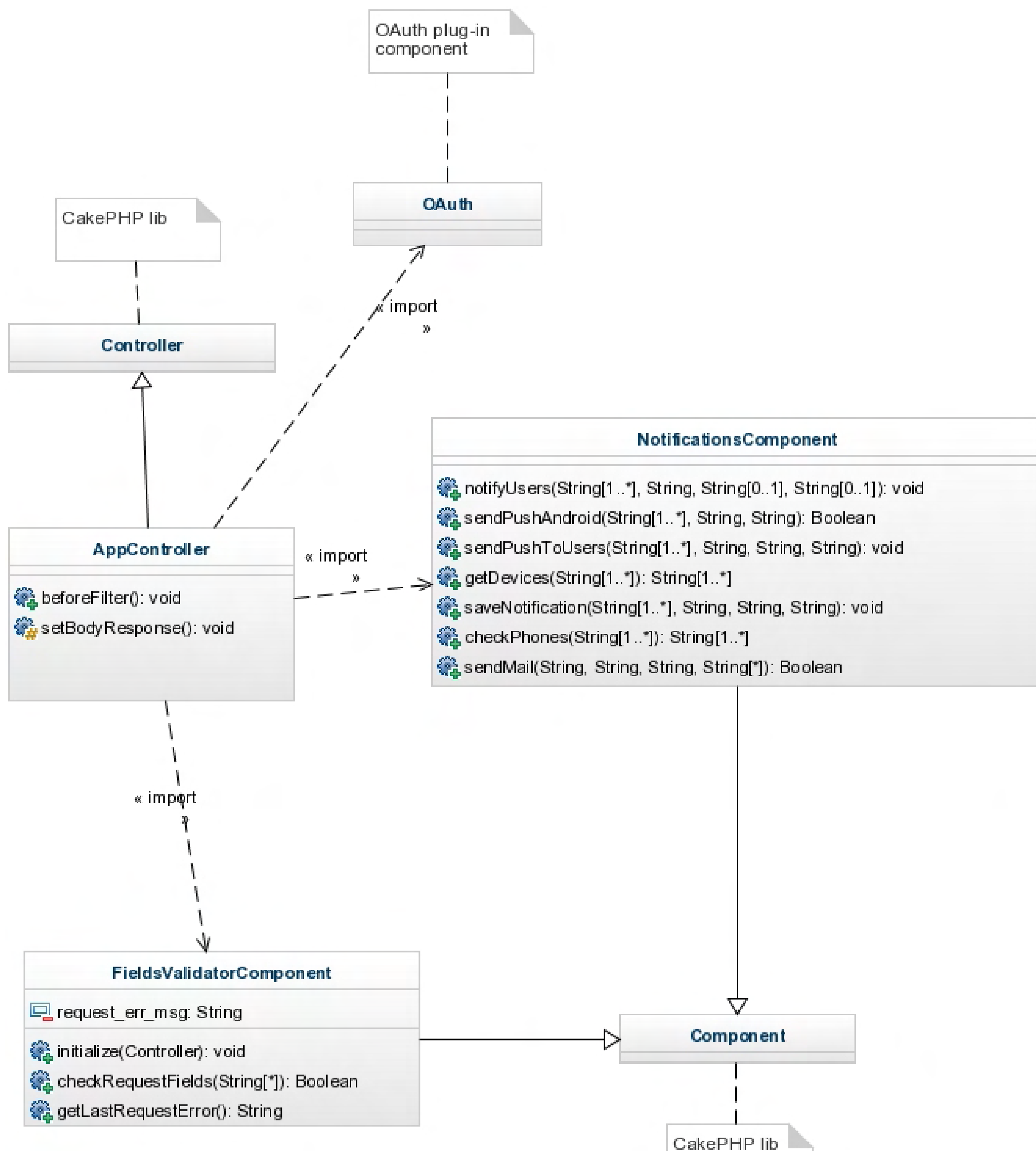

Figura 6.4: Diagrama de clases de AppController y los *components* que éste utiliza

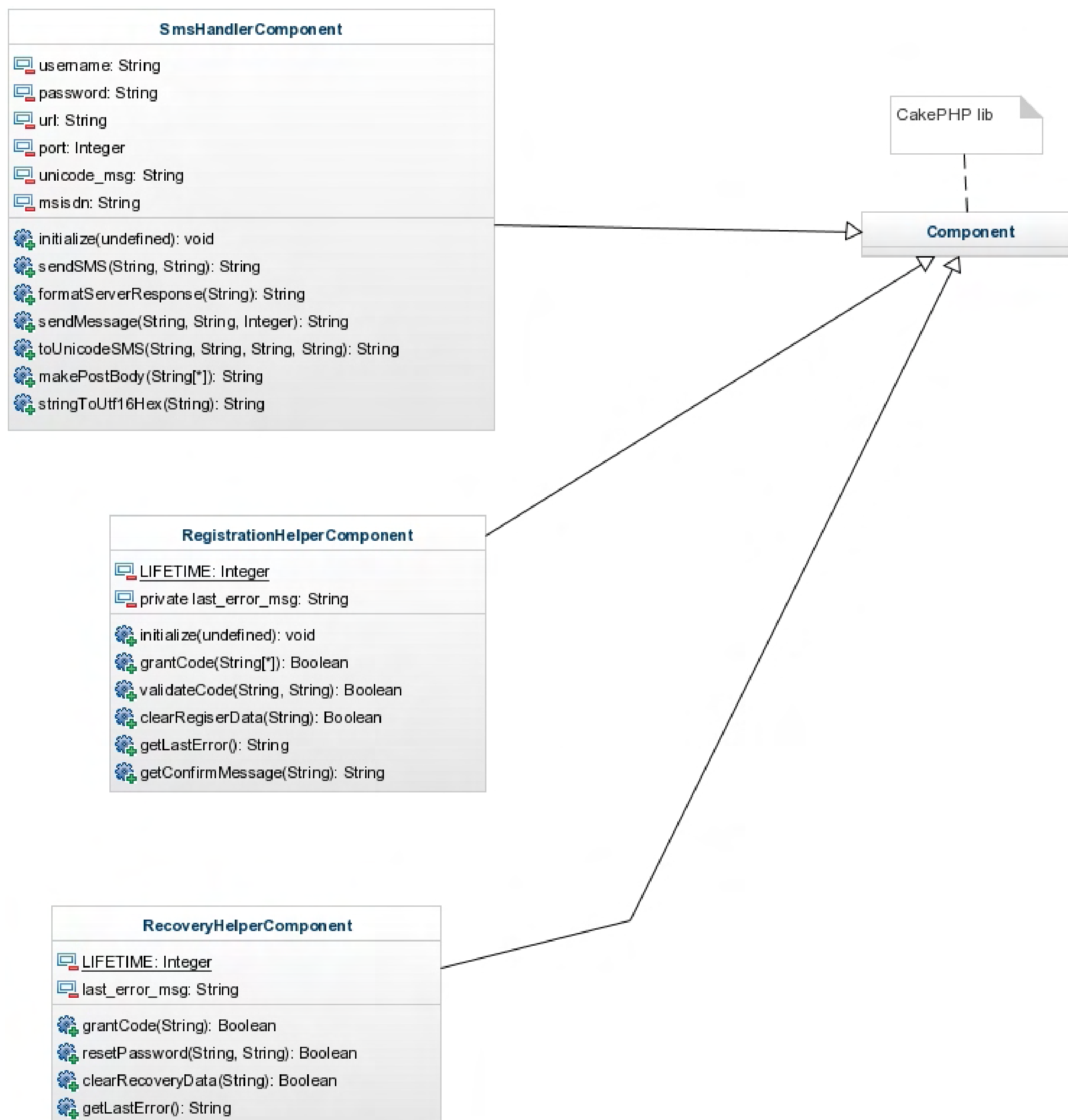

Figura 6.5: Diagrama de clases para extra *components*

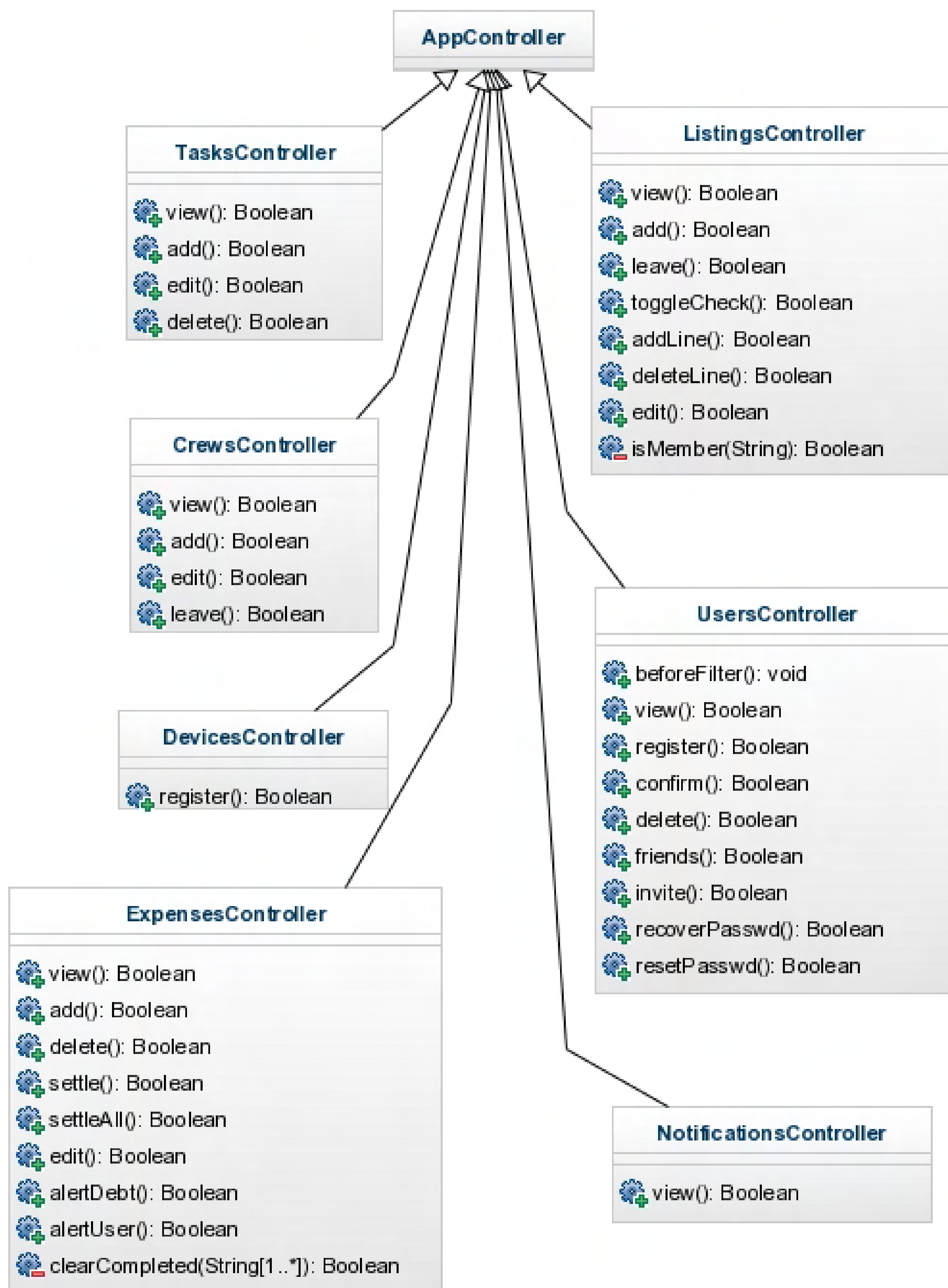

Figura 6.6: Diagrama de clases para *controllers*

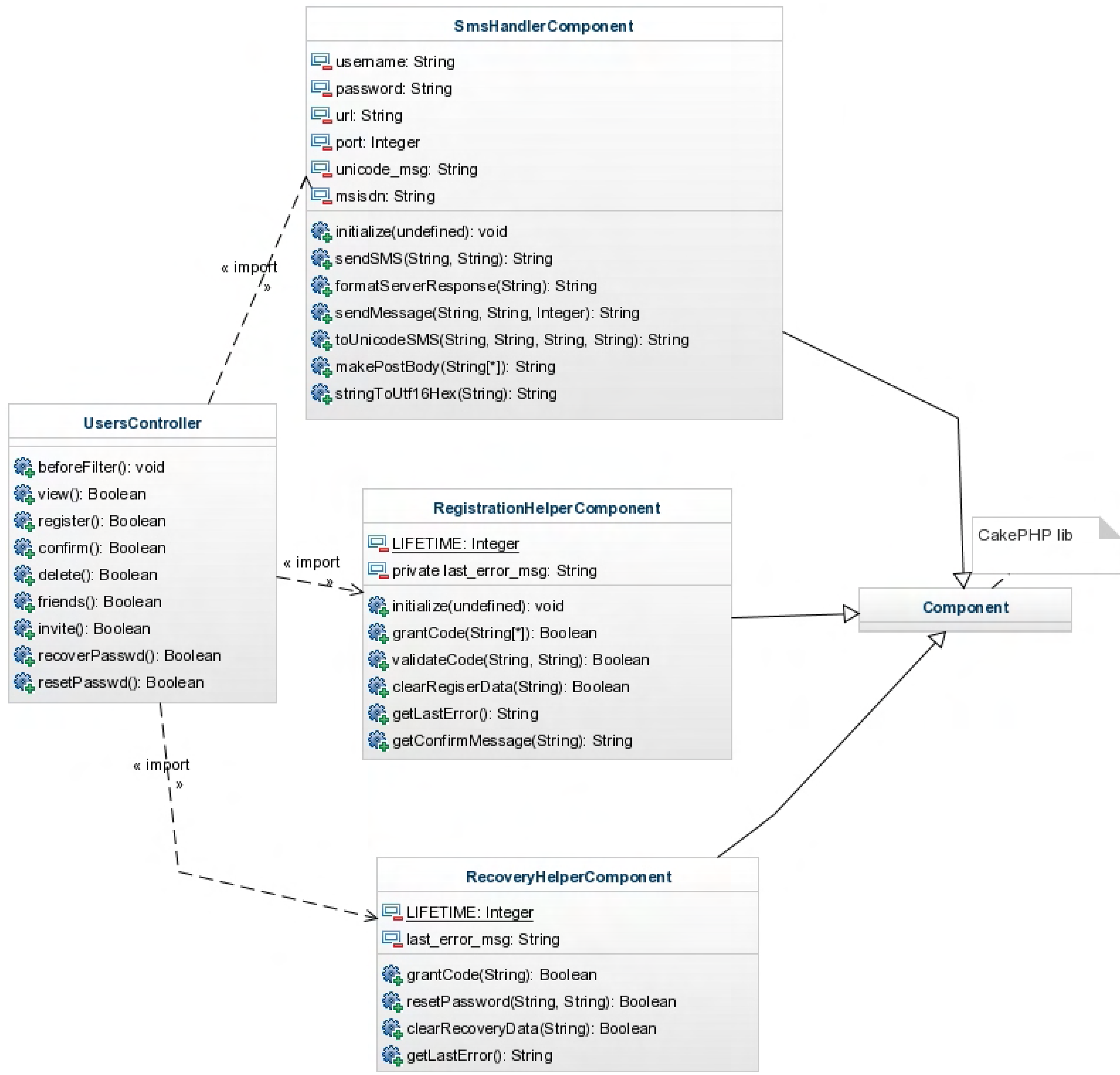

Figura 6.7: Diagrama de clases para UsersController y sus *components*