Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

Universidad de Jaén
Escuela Politécnica Superior

Trabajo Fin de Grado

ESTUDIO Y DESARROLLO DE UN PROTOTIPO DE APLICACIÓN WEB PARA UNA ASOCIACIÓN JUVENIL

Alumno: José Javier Moral Gallego

Tutor: José Ramón Balsas Almagro
Dpto: Informática

Septiembre, 2017

Índice de contenidos:

Índice de ilustraciones:

1. Introducción

1.1.- Motivación

En este trabajo de fin de grado se va a desarrollar una aplicación web para la gestión de una asociación juvenil en la localidad de Valdepeñas de Jaén. El desarrollo de una aplicación web es más adecuado que, por ejemplo, una aplicación de escritorio porque de esta manera no se necesitan adquirir y mantener equipos informáticos además de poder añadir un pequeño sitio web.

El motivo de elegir una aplicación web en lugar de cualquier otra opción es lo que personalmente me resulta más interesante. El desarrollo de aplicaciones web es, actualmente, una de las actividades más demandadas por empresas, como se puede observar en [1], en el ámbito de las aplicaciones informáticas; una gran fuente de empleo.

Las aplicaciones web constituyen una parte muy importante del desarrollo de aplicaciones informáticas. Además, es un campo con una constante y rápida evolución, donde hay numerosas tecnologías y alternativas disponibles para la solución de un mismo problema. Más adelante se estudiarán y comentarán algunas de ellas, sirviendo para refinar la solución propuesta.

Hay que tener en cuenta que, como en muchos otros proyectos, la junta directiva de la asociación juvenil Aincrad (a partir de ahora ‘el cliente’) tiene conocimiento de la problemática actual que se presenta y la necesidad de corregirla, pero no de qué es lo que necesita o cual puede ser una solución. Esto va a condicionar el enfoque que se dé al desarrollo de la aplicación, así como a las técnicas necesarias para obtener los diferentes requisitos de esta.

1.2.- Objetivos

Realizar un estudio de necesidades y soluciones existentes en el contexto seleccionado.

Diseñar un sistema informático especialmente orientado a la utilización de arquitecturas, principios de diseño y metodologías de desarrollo web.

Seleccionar un entorno de desarrollo web e implementar un prototipo de aplicación web que satisfaga las necesidades que se hayan determinado.

1.3.- Metodología

Para el desarrollo de este TFG vamos a adoptar un enfoque ágil empleando una metodología ágil en concreto: SCRUM[1] [2].

El desarrollo ágil nos va a permitir ajustarnos a los distintos cambios que vayan surgiendo conforme la aplicación avance. Un desarrollo tradicional o en cascada no es apropiado para este proyecto, hay demasiada incertidumbre en los requisitos y se prevén cambios constantes. El desarrollo ágil está especialmente indicado en estos casos y será, por tanto, el que elegiremos.

No obstante, necesitamos adaptar esta metodología de trabajo debido a que el equipo de trabajo está formado únicamente por una persona. Como bien se indica en [2], en Scrum se dedica mucho esfuerzo a las numerosas reuniones y a la división de trabajo, esto es innecesario en el primer caso (salvo las reuniones con el cliente) e inexistente en el segundo.

Ya se ha comentado en el punto 1.1 que el cliente no conoce exactamente cuál es la solución a su problemática, por lo que será muy complicado obtener los detalles de la solución al comienzo y avanzar en el desarrollo sin tener cambios en esta. Sí que necesitamos, sin embargo, conocer las necesidades reales del cliente para poder realizar una aplicación que se adapte a las necesidades.

En una metodología ágil esto se traduce en crear historias de usuario y requisitos que abarquen estas necesidades. Para ello realizaremos una serie de entrevistas, una ya realizada antes de comenzar con desarrollo del TFG, otra programada al comienzo de este y otra más por cada entrega, con posibilidad de realizar alguna más si se previera que sería necesario. Se ha preferido usar entrevistas en lugar de cualquier otra técnica porque el cliente manifestó que no era ningún inconveniente para él asistir a estas y que era bastante flexible en cuanto a la planificación de las mismas.

1 https://www.scrumalliance.org/

1.4.- Estructura del documento

El presente documento se ha estructurado de la siguiente manera: en el apartado 2 (Análisis) se han presentado todos los aspectos relacionados con el análisis: obtener requisitos, revisar soluciones existentes, estudiar propuestas tecnológicas y realizar, finalmente, un estudio de viabilidad. En el apartado 3 (Diseño) se han mostrado los aspectos clave del diseño de la solución: diferentes diagramas (dominio, clases, etc), diseño de la interfaz y el plan de pruebas. En el apartado 4 (Implementación) se muestra la arquitectura del sistema desarrollado así como los detalles más interesantes de la implementación. En el apartado 5 (Pruebas) se muestran las pruebas realizadas a la aplicación desarrollada, así como los resultados y conclusiones obtenidos. En el apartado 6 (Conclusiones) se detallan las conclusiones alcanzadas al finalizar el proyecto, así como posibles futuras mejoras a la aplicación obtenida. Por último, en el apartado 7 (Bibliografía) se ha indicado la bibliografía y fuentes utilizadas a lo largo del documento.

Se ha de mencionar que, además de los puntos anteriores, se han creado 3 apéndices no muy extensos. El manual de instalación, que nos describe cómo hemos de instalar la aplicación; el manual de usuario, que nos indica cómo utilizar la aplicación desarrollada, y la descripción de los contenidos suministrado, donde exponemos todos los documentos y archivos desarrollados y entregados en este trabajo de fin de grado.

2. Análisis

2.1.- Análisis preliminar

Partimos de una asociación juvenil relativamente reciente, que tiene una sede física concedida por la junta de Andalucía, y que tiene una serie de actividades que incluyen, y esta es precisamente la parte a la que tenemos que proporcionar una solución, la gestión y control de una serie de actividades relacionadas con su haber diario de esta, como por ejemplo, gestionar las diferentes cuotas de los socios o los movimientos de dinero eventuales (ya sea proporcionar un refresco a un socio o comprar un nuevo activo como un televisor). Además, se ha mostrado en varias ocasiones interés por utilizar un sitio web donde mostrar novedades y eventos para los socios de una manera cómoda y atractiva.

Actualmente estas actividades se materializan en una serie de documentos físicos redactados manualmente junto a otros como facturas o pagarés emitidos por terceras partes. En este punto es donde se propone el desarrollo de una aplicación web, permitiendo que el cliente evite rellenar y gestionar manualmente lo anteriormente citado, además de la gestión, de manera sencilla, de los distintos eventos que se pretenden difundir a través de la web.

En resumen, hay 2 problemas principales a resolver:

-Necesidad de una herramienta que facilite la gestión de cuotas, movimientos de dinero y usuarios.

-Creación de un sitio web en la que la asociación pueda mostrar noticias, programar eventos, etc. relacionados con sus actividades ordinarias.

Estudio de soluciones existentes

Antes de desarrollar una aplicación el primer paso es buscar una ya existente. De haberla ésta se debería recomendar y/o adaptar, si fuera posible, para que resuelva la problemática encontrada. Para ello se buscarán diferentes alternativas válidas antes de decantarse por desarrollarla.

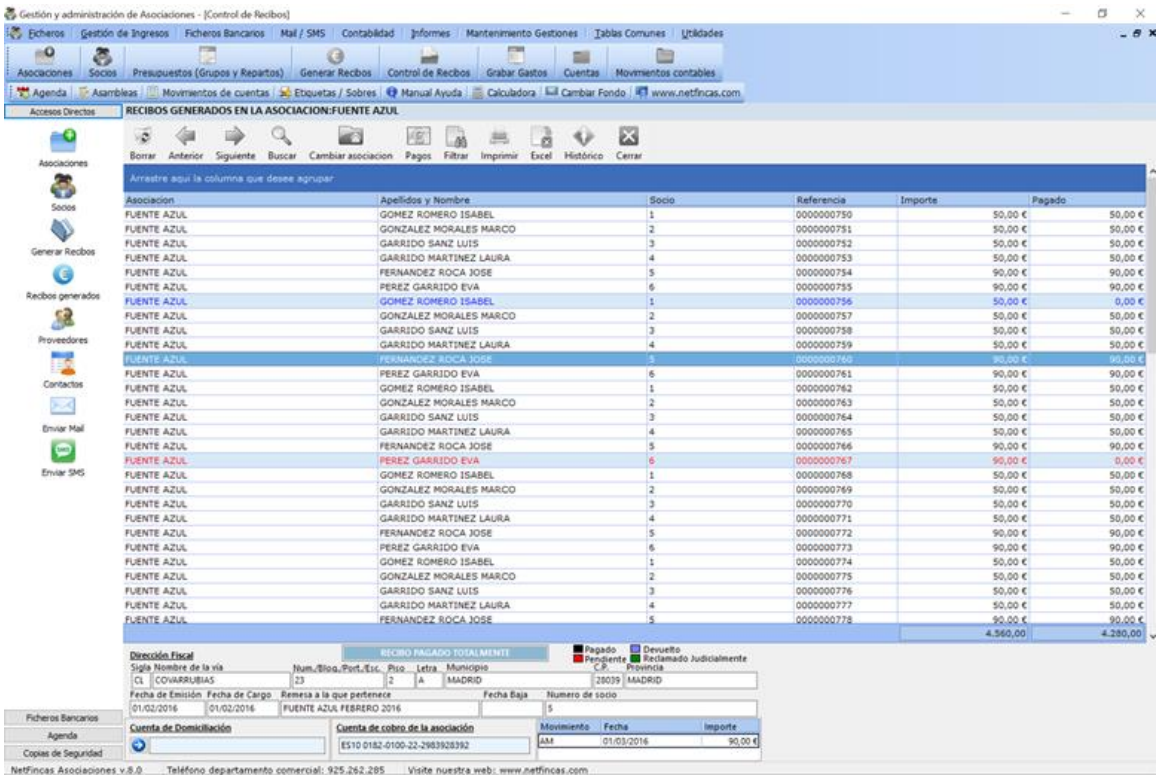
Gestasoc[2] es un servicio, mayormente gratuito, para la gestión de una asociación de cualquier tipo. Entre sus virtudes destacan los correos globales, la gestión de los datos de todos sus socios y la gestión de los recibos. Gestasoc presenta varios inconvenientes: el primero es la imposibilidad de crear y actualizar un interfaz (en forma de página web) específica para Aincrad. El segundo es que el servicio no es capaz de proporcionar una solución para los diferentes movimientos de dinero de la asociación (nos referimos a líquido de la asociación, no a las diferentes cuotas, lo que es un punto aparte) así como las sanciones de los socios. La gestión de eventos, además, está alojada en la parte pública de la web, cuando esto no debería de ser así (lo gestiona como noticias y es abierto). El último inconveniente es que no permite ningún control sobre las actas.

Berrly[3] ofrece otro servicio para la gestión de estas. Es bastante similar a Gestasoc con la salvedad de que ahora se exige una cuota mensual. Cuanta con una interfaz adaptativa y una gran potencia con el control y gestión de socios y eventos. Sin embargo adolece de una problemática similar: no controla el movimiento de dinero ni las actas creadas.

2 https://www.gestasoc.com/
3 https://www.berrly.com/es/

Netfincas[4] proporciona un software para la gestión de asociaciones. Es bastante potente pero presenta una deficiencia fundamental por lo que hemos de rechazarlo: es únicamente una herramienta para el gestor.

4 https://www.netfincas.com/programas+para+asociaciones.html



Por tanto, con todo lo expuesto, necesitamos desarrollar un software a medida para el cliente; una aplicación web que, por un lado almacene y permita gestionar la información de los socios y movimientos, y por otro una pequeña página web donde mostrar distintos eventos y novedades.

Estudio de propuestas tecnológicas existentes

En este punto se deben estudiar las diferentes plataformas para el desarrollo de un servidor. Hay muchas alternativas como php, .net, python, java, node.js o Ruby, pero java es el más interesante. ¿Por qué? porque cuenta con una gran comunidad, una gran madurez, una curva de aprendizaje baja, un gran soporte para la creación de aplicaciones empresariales y el equipo de desarrollo tiene mucha experiencia trabajando con él.

JavaEE (Java Enterprise Edition)

Java ha evolucionado mucho desde sus inicios y ahora es un lenguaje de programación muy potente donde desarrollar aplicaciones web. Es el lenguaje más extendido actualmente como se puede apreciar en [3], por lo que cuenta con una gran comunidad tras él; esto se traduce en multitud de componentes y opciones presentes para el desarrollo de cualquier aplicación.

Realizar una aplicación web, sin embargo, puede ser un proceso complejo, especialmente si se es muy purista y se trabaja únicamente con java, por ello los *frameworks* de desarrollo de aplicaciones web son una excelente solución a esta problemática. Para Java tenemos varias opciones, la más notable es JavaEE (Java Enterprise Edition, donde se puede profundizar mucho más en [4]), que forma parte de la plataforma java. Tiene varias versiones a sus espaldas en las cuales ha evolucionado para proporcionar más componentes y herramientas para los desarrolladores, así como facilitar el uso de los mismos.

Spring framework[5]

Realizar una aplicación web con un *framework* es, en casi todos los escenarios, una buena elección. Spring es un *framework* veterano muy demandado por empresas y con una gran cantidad de herramientas así como una gran capacidad, además ha sido estudiado con una relativa profundidad en diferentes asignaturas estudiadas en la carrera. Sin embargo, adolece de una complejidad inducida por sus numerosas opciones, lo que requiere tener un conocimiento previo de la herramienta, afortunadamente hay mucha bibliografía sobre ello, como por ejemplo [5].

También se optará por desarrollar una aplicación orientada a proporcionar un servicio REST (del inglés *Representational state transfer*). Un servicio REST no es más

5 https://spring.io/

que una manera de proveer de interoperabilidad entre sistemas informáticos a traveś de la red mediante la manipulación de los diferentes recursos web disponibles empleando un sistema de peticiones. Adaptarse a esta filosofía de trabajo no es muy complejo con la ayuda de un *framework* de desarrollo de aplicaciones web, aunque sí es bien cierto que usaremos emplear una biblioteca externa, Jackson[6], que nos permita estandarizar la información con la que trabajan las peticiones.

Angularjs

HTML5 es genial, pero no permite crear una página web dinámica, es necesario emplear javascipt para este cometido, lo que aumenta la complejidad tanto en el desarrollo como mantenimiento. Los Frameworks MVC de programación en el cliente son, debido a esto, una gran opción. La aplicación web que pretendemos desarrollar necesita de una vista dinámica que, con la ayuda de Angular, podemos desarrollar fácilmente de una manera cómoda y profesional.

Angular está, cada vez, más integrado en la comunidad, por lo que al elegirlo es una opción sensata.

JPA

JPA (Java Persistence API) es un API de persistencia de datos desarrollado para Java EE. Se basa en la técnica de programación ORM (Object-Relational Mapping) para transformar las entidades de una base de datos relacional en objetos de un lenguaje de programación orientado a objetos, en este caso Java.

Necesitamos una base de datos que almacene la información de la asociación. JPA nos va a facilitar mucho esta tarea, pues puede realizar todo el esquema de la base de datos y ocuparse de la traducción de entidades relacionales de la base de datos a objetos del modelo de la aplicación web. JPA también implementa un estándar7, por lo que tiene una gran comunidad detrás de él.

Hibernate[8]

Hibernate es otra herramienta basada en ORM para java y .net, y una clara alternativa a usar JPA. La principal diferencia es que está distribuida bajo una licencia de software libre GNU LGPL.

---

6 https://github.com/FasterXML/jackson

7 JSR 338: Java™ Persistence 2.1

8 http://hibernate.org/

Bootstrap[9]

Puesto que hay que diseñar e implementar una interfaz atractiva e interesante y además esta tiene que adaptarse a diferentes dispositivos y tamaños de pantalla, este trabajo es complicado si no se utiliza alguna tecnología como Bootstrap. Boostrap es un framework web de código abierto que contiene un gran número de plantillas HTML y CSS especialmente diseñadas para los elementos de la interfaz, como pueden ser formas, botones, formularios o tipografía. Al contrario que la mayoría de los frameworks existentes, bootsrap sólo permite un desarrollo *front-end*.

9 https://goo.gl/NuVnmS

2.2.- Propuesta de solución

Con el análisis de las diferentes propuestas del punto anterior se puede realizar una propuesta de solución seleccionando todas las tecnologías necesarias de entre las propuestas.

Crearemos una aplicación web que almacene información de los socios, cuotas de estos, eventos y movimientos de dinero. La aplicación permitirá que, mediante dos roles de usuario (junta directiva o administradores y socios) se hagan todas las operaciones necesarias con ellos (crear, actualizar, borrar y consultar en su mayoría) de una forma cómoda, sencilla y atractiva.

La solución final el desarrollo de dos aplicaciones que trabajen de manera colaborativa entre ellas; una aplicación ‘servicio’ que se ocupe de la gestión y almacenamiento de datos y una aplicación ‘cliente’ que se relaciones directamente con el usuario proporcionando la interfaz necesaria para que el usuario trabaje con el servidor.

La parte servidor usará SpringMVC basado en un servicio REST mientras que la parte cliente usará Angular y será la responsable de utilizar dicho servicio. El servidor usará también Hibernate para la persistencia de la información mientras que el cliente empleará Bootstrap para simplificar la creación de su interfaz.

2.3.- Requisitos del sistema

Los requisitos, como se puede ver en [6], son una descripción completa del comportamiento del sistema que se va a desarrollar. Los requisitos se han obtenido de las necesidades del cliente, estas son muchas y es posible que no se puedan implementar dentro de la fecha límite. En este apartado nos centraremos en todas ellas y, más adelante, se priorizarán y ordenarán según el criterio del cliente y el equipo de desarrollo

El proceso de obtener los requisitos puede realizarse de múltiples formas. En este caso se han obtenido mediante la realización de entrevistas reales con el cliente.

Los requisitos se dividen en funcionales y no funcionales según si se refieren al comportamiento o el carácter de la aplicación, respectivamente. También se pueden dividir en requisitos de usuario o de sistema si es el cliente quien los determina o son necesarios para que el sistema funcione.

Dividiremos los requisitos en funcionales y no funcionales de la siguiente forma:

Funcionales:

[1] Calendario.

El sistema debe gestionar un calendario o agenda en la página web donde los usuarios puedan consultar los diferentes eventos de la asociación.

[2] Gestión de eventos.

El administrador quiere gestionar en la aplicación los diferentes eventos (crear, borrar, actualizar y consultar).

[3] Gestión de afiliados.

El administrador quiere poder gestionar toda la información relativa a los socios.

[4] Correos de novedades.

El sistema debe enviar correos automáticamente a los miembros informando de las novedades.

[5] Usuarios conectados.

El sistema debe mostrar, en su página web, los usuarios conectados.

[6] Chat entre usuarios conectados.

El sistema debe mostrar, en su página web, un chat en la web para los usuarios conectados.

[7] Tesorería.

El administrador quiere gestionar los pagos de los miembros y consultar el estado actual de la tesorería.

[8] Control de actas.

Los usuarios quieren poder ver y descargar las actas de la asociación.

[9] Gestión de cuotas.

El administrador debe poder crear cuotas para los socios, así como gestionar cuando las pagan.

[10] Emails de pago.

El sistema debe enviar un correo electrónico a los miembros automáticamente cuando adeuden un pago o lo hayan realizado.

[11] Pasarela de pago.

Los socios deben poder realizar sus pagos a través de una pasarela de pago.

[12] Gestión de impagos.

El administrador debe poder gestionar los impagos y, si fuese necesario, sancionar al socio correspondiente.

[13] Enlaces con redes sociales.

El sistema debe publicar las novedades automáticamente en las redes sociales pertinentes.

No funcionales:

[NF1] Interfaz adaptativa

La aplicación web debe ser compatible con smartphones.

[NF2] Sencillez de uso.

La aplicación no debe necesitar un periodo de aprendizaje para su correcto uso por parte del cliente (tanto socios como administradores).

A continuación vamos a realizar una priorización a efectos de planificación que nos va a permitir seleccionarlos a lo largo de cada una de las iteraciones. De esta forma podremos asegurar que, de haber variaciones en los tiempos previstos de desarrollo, se sabrá qué requisitos se pueden posponer y cuales no a efectos de obtener un producto operativo.

Dividiremos las prioridades en baja, media y alta. De entre los requisitos que tengan la misma prioridad los ordenaremos según el criterio del cliente y, en última instancia, nosotros mismo.

Requisito	**Prioridad**
[1]Calendario	Alta
[2]Gestión de eventos	Alta
[3]Gestión de afiliados	Alta
[4]Correos de novedades	Baja
[5]Usuarios conectados	Baja

[6]Chat entre usuarios conectados	Baja
[7]Tesorería	Alta
[8]Control de actas	Media
[9]Gestión de cuotas	Alta
[10]Emails de pago	Baja
[11]Pasarela de pago	Baja
[12]Gestión de impagos	Media
[13]Enlaces con redes sociales	Baja
[NF1]Interfaz adaptativa	Alta
[NF2]Sencillez de uso	Alta

2.4.- Historias de Usuario

Una historia de usuario es una representación de un requisito escrito en una o dos frases utilizando el lenguaje común que detallan aún más al requisito. Las historias de usuario son ampliamente utilizadas en las metodologías ágiles como SCRUM.

En este apartado se detallarán todas las historias relacionadas con los requisitos del apartado 2.3, donde se dará una breve explicación del porqué de las prioridades antes mostradas así como de las posibles dependencias entre historias de usuario. Las dependencias son útiles porque no siempre es posible definir requisitos que no estén relacionados con otros y no necesiten de colaboración con estos. Las dependencias van a marcar un orden relativo entre requisitos.

ID: 1.
Título: Calendario.
Descripción: El sistema debe gestionar un calendario o agenda en la página web donde los usuarios puedan consultar los diferentes eventos de la asociación. En este debe aparecer el organizador, fecha y una descripción.
Prioridad: Alta. La creación de un calendario o agenda es esencial porque es la manera en que la aplicación mostrará a los usuarios los eventos existentes.
Dependencias: 2.

ID: 2.
Título: Gestión de eventos.
Descripción: Como administrador de la aplicación quiero gestionar (CRUD) los diferentes eventos que puedan ser creados por la asociación así como la información que contengan.
Prioridad: Alta. Esto es debido a que los eventos son una parte fundamental de la propia asociación.
Dependencias: Ninguna.

ID: 3.
Título: Gestión de afiliados.
Descripción: Como administrador de la aplicación gestionar información de los diferentes tipos de miembros (administradores y socios) así como su información personal.

Prioridad: Alta. La gestión de afiliados es una parte fundamental de la propia asociación.
Dependencias: Ninguna.

ID: 4.
Título: Correos de novedades.
Descripción: El sistema debe enviar correos automáticamente a los miembros informando de las novedades. Estos serán debido a eventos y tienen que que contener toda la información relevante.
Prioridad: Baja. El cliente no considera que los correos sean parte esencial de la aplicación.
Dependencias: 3.

ID: 5.
Título: Usuarios conectados.
Descripción: El sistema debe mostrar, en su página web, los usuarios conectados.
Prioridad: Baja. El cliente considera mostrar los usuarios como un pequeño detalle estético.
Dependencias: 3.

ID: 6.
Título: Chat entre usuarios conectados.
Descripción: El sistema debe mostrar, en su página web, un chat en la web para los usuarios conectados. Todo lo que se escriba en un chat debe ser registrado.
Prioridad: Baja. No se prevé que haya un gran uso del chat, por lo que es un añadido secundario.
Dependencias: 3.

ID: 7.
Título: Tesorería.
Descripción: Como tesorero quiero gestionar los movimientos de dinero cotidianos en la asociación. Cada movimiento consta de una cantidad y un concepto.
Prioridad: Alta. El conteo a mano del movimiento de dinero es complicado y uno de los aspectos fundamentales de la aplicación.
Dependencias: Ninguna.

ID: 8.
Título: Control de actas.
Descripción: Como usuario quiero poder ver las actas de la asociación. Las actas se deben almacenar en la aplicación, el administrador debe ser capaz de subirlas y el socio de descargarlas.
Prioridad: Media. El control de actas es interesante, aunque la manera en que se realiza manualmente es bastante eficiente y, además, necesita que se siga haciendo así incluso si se incorpora este requisito.
Dependencias: Ninguna.

ID: 9.

Título: Gestión de cuotas.
Descripción: Como Secretario quiero crear cuotas para los socios, así como gestionar cuando las pagan. Las cuotas tendrán una cantidad y una fecha de emisión y liquidación, así como una vigencia específica para cada una.
Prioridad: Alta. La gestión de cuotas es uno de los apartados más importantes de toda la aplicación, puesto que actualmente se hace a mano y es extremadamente tedioso.
Dependencias: 3.

ID: 10.
Título: Emails de pago.
Descripción: El sistema debe enviar un correo electrónico a los miembros automáticamente cuando adeuden un pago o lo hayan realizado. En estos correos no se debe mostrar información personal.
Prioridad: Baja. Debido al trato tan personal que se da a los socios los emails son sólo un bonito añadido; tanto los socios como la junta directiva prefieren comunicarse cara a cara en este aspecto.
Dependencias: 9.

ID: 11.
Título: Pasarela de pago.
Descripción: Como socio quiero poder realizar mis pagos a través de una pasarela de pago. La aplicación, además, procesará todas las cuotas pertinentes.
Prioridad: Baja. No todos los socios son mayores de edad, por lo que puede haber problemas en este punto (PayPal, por ejemplo, requiere que los usuarios sean mayores de edad).
Dependencias: 9.

ID: 12
Título: Gestión de impagos.
Descripción: Como secretario quiero administrar los impagos y, si fuese necesario, sancionar al socio correspondiente. La aplicación debe permitir marcar un usuario como sancionado, así como desmarcarlo. Las sanciones son específicas para cada usuario por lo que la aplicación sólo recogerá información sobre si un usuario está o no sancionado.
Prioridad: Media. Las sanciones son necesarias aunque no son automáticas, que la aplicación que se desarrolle pueda gestionarlas es muy útil, aunque necesitará de su contraparte manual por parte de la junta directiva.

ID: 13.
Título: Enlaces con redes sociales.
Descripción: El sistema debe publicar las novedades automáticamente en las redes sociales pertinentes.
Prioridad: Baja. El cliente manifestó que sería interesante añadirlo en caso de que sobrara tiempo antes de la entrega final.
Dependencias: 2.

2.5.- Planificación inicial de tareas

Llegamos a un punto muy interesante. Primero se ordenarán las historias de usuario según su prioridad para, más tarde, realizar su división en tareas. Una vez obtenidas las tareas pasaremos a estimar su duración y así poder obtener la pila del producto[10]. Con todos estos datos se podrá ver cuántos de los requisitos obtenidos se podrán implementar al final y cuáles se quedarán fuera de la solución inicial debido a falta de tiempo.

Por último, se mostrará la variación entre los resultados estimados y los reales, así como las causas de ello.

La ordenación de las tareas se realizará según la prioridad asignada pero teniendo en cuenta dos factores: en primer lugar, una tarea dependiente de otra necesita de la otra para poder desarrollarse. En segundo lugar, cuando dos tareas tengan igual prioridad el cliente debería priorizar una de ellas.

3	Gestión de afiliados
9	Gestión de cuotas
2	Gestión de eventos
1	Calendario
7	Tesorería
12	Gestión de impagos
8	Control de actas
4	Correos de novedades
10	Emails de pago
13	Enlaces con redes sociales
11	Pasarela de pago
5	Usuarios conectados
6	Chat entre usuarios conectados

Ahora pasaremos a dividir cada una de las historias de usuario en tareas y asignarles diferentes puntos de historia para poder medir así su duración. En este punto hay que tener en cuenta los dos requisitos no funcionales puesto que en algunas historias de usuario habrá tareas adicionales para asegurarnos de que se cumplen. Para asignar los puntos de historia a cada tarea se usará *Planning Poker* [7], empleando, para ello, la serie de Fibonacci: 0, 0’5, 1, 2, 3, 5, 8, 13, 21.

Hay que tener en cuenta que, además de las tareas relacionadas con las propias historias de usuario, necesitamos añadir otras tareas relacionadas con la arquitectura y tecnologías adoptadas en el desarrollo de la aplicación, como, por ejemplo, configurar Spring o crear la base de datos.

[10] https://goo.gl/FGu4NT

Al margen de estas, las tareas de cada una de las historias son bastante similares entre sí; diseñar el algoritmo, crear o complementar las interfaces (aquí entra el primer requisito no funcional), crear o complementar los controladores, implementar el algoritmo diseñado, acciones CRUD en la base de datos y, finalmente, pruebas (teniendo en cuenta, también, el segundo requisito no funcional).

Por último se añadirán también los perfiles de cada uno de los miembros del equipo de desarrollo que trabajen en la tarea. Como ya hemos comentado anteriormente, al haber un único miembro este debe tomar los roles de analista y programador dependiendo de cada una de las tareas. Esto va a permitir, en el punto siguiente, calcular el coste del proyecto.

3 Gestión de afiliados (76 puntos de historia en total)

Tarea	Puntos de historia	Rol
Creación del prototipo de la aplicación	13	Analista
Configuración de Spring	13	Programador
Configuración de AngularJS	13	Programador
Creación de la base de datos	5	Programador
Análisis y Diseño	5	Analista
Interfaz para el servidor (NF1)	2	Programador
Interfaz para el cliente (NF1)	3	Programador
Creación de DAOs	5	Programador
Creación de controladores	5	Programador
Implementación del algoritmo	5	Programador
Persistencia en BBDD	1	Programador
Pruebas (NF2)	5	Programador
Pruebas del prototipo	1	Programador

9 Gestión de cuotas (27 puntos de historia en total, 103 puntos de historia acumulados)

Tarea	Puntos de historia	Rol
Análisis y Diseño	5	Analista
Interfaz para el servidor	2	Programador
Interfaz para el cliente	3	Programador
Creación de DAOs	5	Programador
Creación de controladores	5	Programador
Implementación del algoritmo	3	Programador
Persistencia en BBDD	1	Programador
Pruebas (NF2)	3	Programador

2 Gestión de eventos (27 puntos de historia en total, 130 puntos de historia acumulados)

Tarea	Puntos de historia	Rol
Análisis y Diseño	5	Analista
Interfaz para el servidor (NF1)	2	Programador
Interfaz para el cliente (NF1)	3	Programador
Creación de DAOs	5	Programador
Creación de controladores	5	Programador
Implementación del algoritmo	3	Programador
Persistencia en BBDD	1	Programador
Pruebas (NF2)	3	Programador

1 Calendario (37 puntos de historia en total, 137 puntos de historia acumulados)

Tarea	Puntos de historia	Rol
Configuración de Spring Security	13	Programador
Análisis y Diseño	8	Analista
Interfaz para el servidor (NF1)	2	Programador
Interfaz para el cliente (NF1)	3	Programador
Modificar controladores	3	Programador
Implementación del algoritmo	3	Programador
Pruebas (NF2)	3	Programador

7 Tesorería (27 puntos de historia en total, 164 puntos de historia acumulados)

Tarea	Puntos de historia	Rol
Análisis y Diseño	5	Analista
Interfaz para el servidor (NF1)	2	Programador
Interfaz para el cliente (NF1)	3	Programador
Creación de DAOs	5	Programador
Creación de controladores	5	Programador
Implementación del algoritmo	3	Programador
Persistencia en BBDD	1	Programador
Pruebas (NF2)	3	Programador

12 Gestión de impagos (21 puntos de historia en total, 185 puntos de historia acumulados)

Tarea	Puntos de historia	Rol
Análisis y Diseño	5	Analista
Interfaz para el servidor (NF1)	2	Programador
Interfaz para el cliente (NF1)	3	Programador
Modificar controladores	3	Programador
Implementación del algoritmo	3	Programador
Persistencia en BBDD	2	Programador
Pruebas (NF2)	3	Programador

8 Control de actas (27 puntos de historia en total, 212 puntos de historia acumulados)

Tarea	Puntos de historia	Rol
Análisis y Diseño	5	Analista
Interfaz para el servidor (NF1)	2	Programador
Interfaz para el cliente (NF1)	3	Programador
Creación de DAOs	5	Programador
Creación de controladores	5	Programador
Implementación del algoritmo	3	Programador
Persistencia en BBDD	1	Programador
Pruebas (NF2)	3	Programador

4 Correos de novedades (19 puntos de historia en total, 231 puntos de historia acumulados)

Tarea	Puntos de historia	Rol
Análisis y Diseño	2	Analista
Interfaz para el servidor (NF1)	1	Programador
Interfaz para el cliente (NF1)	3	Programador
Modificar DAOs	3	Programador
Modificar controladores	3	Programador
Implementación del algoritmo	3	Programador
Persistencia en BBDD	1	Programador
Pruebas (NF2)	3	Programador

10 Emails de pago (19 puntos de historia en total, 250 puntos de historia acumulados)

Tarea	Puntos de historia	Rol
Análisis y Diseño	2	Analista
Interfaz para el servidor (NF1)	1	Programador
Interfaz para el cliente (NF1)	3	Programador
Modificar DAOs	3	Programador
Modificar controladores	3	Programador
Implementación del algoritmo	3	Programador
Persistencia en BBDD	1	Programador
Pruebas (NF2)	3	Programador

13 Enlaces con redes sociales (18 puntos de historia en total, 268 puntos de historia acumulados)

Tarea	Puntos de historia	Rol
Análisis y Diseño	2	Analista
Interfaz para el servidor (NF1)	1	Programador
Interfaz para el cliente (NF1)	3	Programador
Modificar DAOs	3	Programador
Modificar controladores	3	Programador
Implementación del algoritmo	3	Programador
Pruebas (NF2)	3	Programador

11 Pasarela de pago (39 puntos de historia en total, 307 puntos de historia acumulados)

Tarea	Puntos de historia	Rol
Análisis y Diseño	3	Analista
Estudio de tecnologías existentes	5	Programador
Interfaz para el servidor (NF1)	2	Programador
Interfaz para el cliente (NF1)	3	Programador
Modificar DAOs	3	Programador
Modificar controladores	3	Programador
Adaptación de la tecnología seleccionada	8	Programador
Implementación del algoritmo	5	Programador
Sincronización con BBDD	2	Programador
Pruebas (NF2)	5	Programador

5 Usuarios conectados (12 puntos de historia en total, 319 puntos de historia acumulados)

Tarea	Puntos de historia	Rol
Análisis y Diseño	2	Analista
Interfaz para el servidor (NF1)	1	Programador
Interfaz para el cliente (NF1)	3	Programador
Implementación del algoritmo	3	Programador
Pruebas (NF2)	3	Programador

6 Chat entre usuarios conectados (26 puntos de historia en total, 345 puntos de historia acumulados)

Tarea	Puntos de historia	Rol
Análisis y Diseño	3	Analista
Interfaz para el servidor (NF1)	1	Programador
Interfaz para el cliente (NF1)	3	Programador
Modificar DAOs	3	Programador
Modificar controladores	3	Programador
Implementación del algoritmo	8	Programador
Persistencia en BBDD	2	Programador
Pruebas (NF2)	3	Programador

Con todo lo obtenido podemos, finalmente, crear la pila del producto y seleccionar qué requisitos se estima que no se podrán incorporar a la solución. Para ello, primeramente convertiremos los puntos de historia de las historias de usuario a horas de trabajo. Una vez obtenidas estas simplemente tenemos que hacer la división pertinente.

Un trabajo de fin de grado se compone de, aproximadamente, 300 horas de trabajo. De esas 300 horas hay que dedicar una parte importante a la creación y actualización de la memoria del trabajo de fin de grado que, aunque no está presente en la planificación (porque es una memoria, no un manual de la aplicación) consume una buena parte de ese tiempo. Además hay que tener en cuenta las reuniones con el cliente.

La memoria del tfg es bastante extensa por su naturaleza. Para medir el tiempo necesario para crearla la compararemos con las historias de usuario. Se prevé que sea un poco más grande que dos historias de usuario, por lo que se alargará durante unos 60 puntos de historia.

Además de todo lo mencionado también hay que añadir 10 puntos de historia en la creación de un prototipo. El prototipo (del que se hablará más detenidamente en el punto 2.8) nos va a permitir obtener un mayor conocimiento de los requisitos del proyecto, especialmente los no funcionales (que son los que están íntimamente relacionados con la interfaz del cliente). El prototipo se ha añadido como la primera tarea

de la primera historia de usuario.

Los puntos de historia son una medida bastante abstracta, por lo que traducirlos a horas es algo totalmente subjetivo y dependiente del equipo de desarrollo. Si hacemos un sencillo cálculo de 1 punto de historia = 1 hora de trabajo, podemos hacer una planificación inicial. Eliminando la memoria del tfg tenemos, aproximadamente, 240 horas de trabajo. Esto significa que tenemos 60 horas de trabajo por mes, desde Marzo hasta Junio, que serán 240 puntos de historia.

Los *sprints* que definiremos serán, ahora, de 15 días, dos semanas (dos *sprints* por mes). Debido al tamaño del equipo de desarrollo es excesivo realizar una entrega al final de cada sprint, pues no habrá un incremento significativo en el desarrollo de la aplicación. Este es el motivo por el cual se ha decidido realizar las entregas cada dos sprints (Al final del mes, aunque probablemente se retrase hasta principio del mes siguiente debido a que se debe consensuar un día entre la asociación y el equipo de desarrollo, lo que en principio no supone ningún inconveniente en el desarrollo)

Si no hay cambios (que los habrá), se cumplirá con todos los requisitos de prioridad alta. No podemos asegurar que todos los de prioridad media se cumplan ya que el equipo de desarrollo no tiene demasiada experiencia estimando y hay muchas tecnologías en las que se tiene, también, poca experiencia, por lo que el tiempo real puede ser distinto, aunque es bastante probable que se incorporen. Es poco probable que se incorpore algún requisito de prioridad baja, aunque estos (exceptuando la pasarela de pago) son relativamente pequeños y fáciles de implementar (siempre y cuando no se altere la prioridad de los requisitos).

Con todo esto tenemos la siguiente planificación inicial:

Marzo:
- 1-15 primer sprint.
  - Gestión de afiliados.
- 16-30 segundo Spring.
  - Gestión de afiliados (continuación).
- 31 Primer hito (primera entrega).

Abril:
- 1-15 tercer sprint.
  - Gestión de afiliados (continuación).
  - Gestión de cuotas.
- 16-29 cuarto Spring.
  - Gestión de cuotas (continuación).
  - Gestión de eventos.
- 30 segundo hito (segunda entrega).

Mayo:
- 1-15 quinto sprint.
  - Gestión de eventos (continuación).

Calendario.

16-30 sexto sprint.

Calendario (continuación).

Tesorería.

31 tercer hito (tercera entrega).

Junio:

1-15 séptimo sprint.

Gestión de impagos.

Control de actas.

16-29 octavo sprint.

Control de actas (continuación).

30 cuarto hito (entrega final).

Por último es necesario comentar la temporización real final que se ha obtenido durante el desarrollo del proyecto, que es algo distinta de la inicial:

Marzo:

1-15 primer sprint.

Gestión de afiliados.

16-30 segundo Spring.

Gestión de afiliados (continuación).

31 Primer hito (primera entrega).

Abril:

1-15 tercer sprint.

Gestión de afiliados (continuación).

Gestión de cuotas.

16-29 cuarto Spring.

Gestión de cuotas (continuación).

30 segundo hito (segunda entrega).

Mayo:

1-15 quinto sprint.

Gestión de cuotas (Cambio en el requisito).

Gestión de eventos.

Calendario.

16-30 sexto sprint.

Calendario (continuación).

31 tercer hito (tercera entrega).

Junio:

1-15 séptimo sprint.

Gestión de cuotas (Cambio en el requisito).

Calendario (continuación)

Gestión de eventos.

16-29 octavo sprint.
Gestión de eventos (continuación).
Gestión de impagos.
30 cuarto hito (entrega final).

Las estimaciones no fueron totalmente correctas y además hubo cambios. Como se puede apreciar, la gestión de afiliados duró más de los esperado, en gran parte por subestimar la duración de la configuración de Spring. La gestión de cuotas se ha visto arrastrada por ello, aunque en realidad ha durado más o menos lo planificado. La gestión de eventos también se retrasó, y esto fue en parte por la complejidad de la interfaz de la aplicación cliente; creando el controlador Angular para los eventos.

Además, en la segunda y tercera entrega hubo cambios en la gestión de cuotas. Se pasó de un sistema de cuotas trimestrales para los mayores de 28 años y mensual para el resto a otro anual para todos a finales de la segunda entrega, lo que hizo que tuviésemos que incorporar las modificaciones al inicio del quinto sprint. En el sexto sprint volvió a cambiar por un sistema en que cada una de las cuotas tendría una duración y una cuantía específicas para ellas, lo que volvió a necesitar de una modificación del requisito, que se materializó en una nueva reordenación de la pila del producto en el séptimo sprint.

El calendario y la gestión de impagos, por su parte, fueron desarrollados dentro de la planificación estimada, haciendo que no hubiese más cambios llegados a ese punto.

2.6.- Estudio de viabilidad

En este punto se analizará en primer lugar los costes de desarrollo y explotación (si los hubiese). Además se darán varias alternativas de modelo de negocio y se justificará la elección de una de ellas.

En primer lugar debemos comentar que el coste de desarrollo es únicamente el sueldo de cada uno de los empleados ya que no es necesario adquirir licencias y hacer un desembolso en activos, aunque en este proyecto hay un único trabajador (el autor). Para calcular la nómina es necesario conocer las horas de trabajo total así como cuáles de ellas han sido en su papel de analista o programador, debido a que los sueldos base de estos no son iguales.

En este punto se podría, simplemente, adoptar un rol de analista-programador, bastante usual en la actualidad, pero teniendo una descomposición en tareas de cada uno de los requisitos es bastante sencillo e interesante hacer las divisiones mencionadas.

Según el convenio TIC [8], en 2017 el sueldo base de un analista de sistemas es de 1570€ mensuales, mientras que un programador es de 1100€ mensuales, aunque el salario medio está entorno a unos 2200€ en el caso del analista y unos 1820 en el caso del programador. No todos los meses tienen la misma cantidad de días pero podemos aproximarnos con un ligero margen de error a 176 horas de trabajo mensuales, lo que

hace un total de 12’5€ por hora para un analista de sistemas y 10’34 para un programador.

Como se puede calcular con la descomposición en tareas de las historias de usuario y todo lo expuesto en el punto 2.5, el número de horas que el analista debe dedicar es de 46 en el desarrollo de la solución y 60 en el desarrollo de la memoria, además de las 10 restantes que debe estar reunido con el cliente, lo que hace un total de 116 horas de trabajo; 1450€.

El programador dedicará, por su parte, el resto de horas, que son 181 (de las trescientas originales eliminamos las realizadas por el analista). Esto hace un total de 1871€.

No habrá ningún gasto por licencias o adquisición de nuevas herramientas, puesto que no necesitamos adquirir ninguna licencia ni comprar algún material para el desarrollo de la aplicación. No obstante, se numerarán de manera informativa.

Tecnología	Coste
Angularjs	0
Bootstrap	0
Spring	0
Spring Security	0
Hibernate	0
Json	0

Eso sí, el equipo de desarrollo ha necesitado usar una conexión a la red eléctrica y a internet durante las 300 horas, lo cual se traduce en un coste adicional.

Según la red eléctrica de España[11], el precio medio en las horas laborales de la conexión eléctrica es de 0.12€/KWh, lo que multiplicado por 300h da un total de 36€ en consumo eléctrico.

Si tenemos una conexión estándar a internet de unos 20€ mensuales tenemos un consumo de 0’027 por hora, lo que hace un total de 8’22€ por la conexión a internet.

Resumiendo:

Concepto	Coste
Salario del analista	1450€
Salario del programador	1871€
Licencias	0€
Consumo eléctrico	36€
Conexión a internet	8’22€
Total	**3365’22€**

[11] http://www.ree.es/es/

Ahora bien, la aplicación debe estar almacenada en algún lugar donde se ejecute y se lance. Una opción muy interesante es utilizar el EC2 de Amazon[12] , donde eligiendo una instancia reservada tipo t2.nano, Aincrad sólo necesitaría 3.29€ mensuales por el mantenimiento, una cantidad muy reducida. Eso sí, es necesario que se haga, en primer lugar, un despliegue en el servidor de toda la arquitectura y software necesario (como por ejemplo el servidor de aplicaciones web).

Otra opción es adquirir un dominio y elegir un equipo de la propia asociación como servidor. Un dominio .es puede resultar en 5€ anuales. Como podemos ver es una cantidad mucho menor, además de contar con algún que otro servicio añadido como correo y demás (que no tienen ningún valor desde el punto de vista del proyecto) pero presenta un inconveniente: necesita del servidor que proporcione Aincrad. Aincrad tiene equipos que hagan este papel, pero simplemente el coste energético del funcionamiento de estos es mayor (por mucho) a los casi 40€ anuales de Amazon EC2. También habría, además, que configurar de igual manera la plataforma que eligiendo la opción de Amazon.

También habría más opciones, claro está, aunque este es un punto que el cliente debe elegir, aunque el equipo de desarrollo tiene el deber profesional de aconsejar lo mejor posible a este para elegir la opción más acertada y conveniente para él.

En este punto necesitamos pensar en el modelo de negocio que vamos a adoptar. Cubrir gastos no es suficiente, como resulta lógico, por lo que deberíamos tener un beneficio. Si quisiéramos un beneficio del 30% necesitaríamos conseguir 4350€ por la aplicación. Puede no parecer gran cantidad para una asociación cualquiera, pero el cliente no deja de ser una asociación juvenil. Es posible que la Junta de Andalucía dispusiese de un fondo especial para subvencionar este tipo de gastos, por lo que la venta de la aplicación podría ser una opción.

Hay otra alternativa, se puede vender la aplicación cubriendo su coste, 3362’22€, y obtener beneficios con el mantenimiento y actualización de la aplicación. O reduciendo su valor por debajo del costo para que, conforme avance el tiempo en el que se mantiene y actualiza la aplicación, empiece a arrojar beneficios (o incluso haciéndola gratuita y cobrando el resto de tiempo invertido).

Una última alternativa sería llegar a más mercados e intentar abarcar varias asociaciones y tener más de un cliente. Esta opción sería la más lucrativa, ya que nos permitiría obtener beneficios de más de una sola entidad, haciendo posible, por consecuencia, que se pueda reducir el precio de venta de la aplicación. Aunque parezca ideal, la aplicación que se va a desarrollar está íntimamente ligada al carácter de esta, por lo que habrá que hacer un estudio de requisitos de más asociaciones y crear un software

12 https://goo.gl/vMoN6E

de carácter general que cubra no sólo los requisitos de esta, sinó de todas las posibles asociaciones que puedan ser, en algún momento, clientes nuestros. Este objetivo es, tal vez, demasiado ambicioso para un proyecto con 300 horas de trabajo de un único miembro con poca experiencia.

La opción más interesante es, tal vez, la segunda. Si se pone a la venta el software por la mitad del costo de desarrollo aproximadamente (unos 2000€, una cantidad asumible) y definimos con 10€ cada una de las horas que dedicaremos al mantenimiento y actualización de la aplicación sólo necesitaríamos 230 horas de trabajo para que la asociación hubiese pagado la aplicación, menos tiempo del dedicado al desarrollo. En este punto también se tendrían gastos (como las nóminas), pero la posibilidad de que el mantenimiento y actualización (que será sencillo de realizar por las tecnologías seleccionadas) se alargue resulta en un modelo de negocio bastante atractivo.

2.7.- Modelo de dominio

El modelo del dominio es un modelo gráfico conceptual en el que se representan todas las entidades relacionadas en un contexto específico, en el caso del presente documento, todos los temas relacionados con la representación de la solución.

En el modelo del dominio se representan los aspectos claves del dominio del problema, así como las relaciones entre las entidades en él representadas.

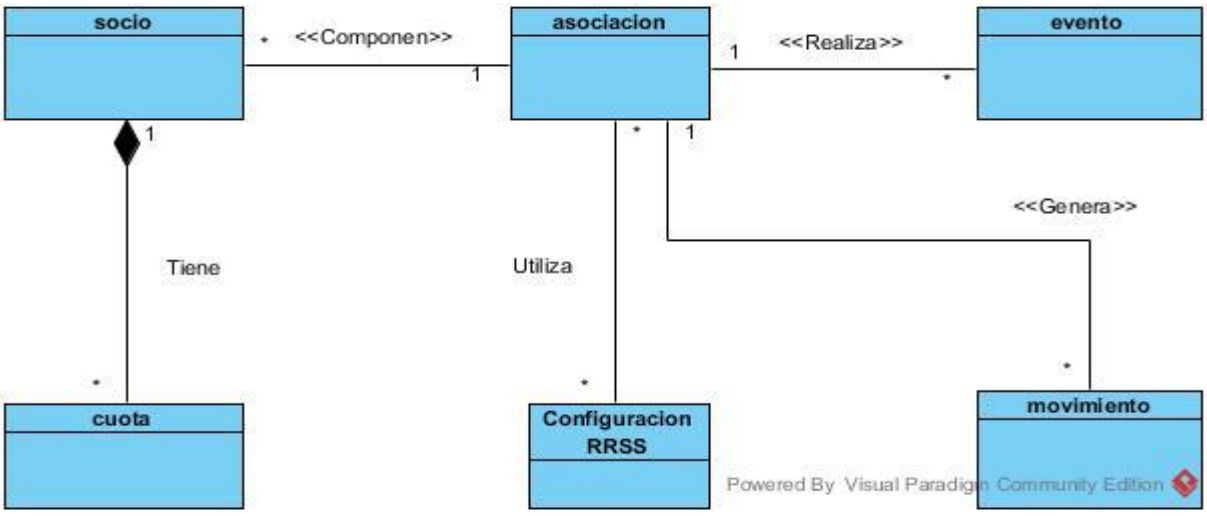


Ilustración 1: Modelo del dominio

En UML[13] un modelo del dominio se representa con un diagrama de clases.

13 http://www.uml.org/

Como se puede apreciar, se han obtenido diferentes entidades y relaciones entre ellas, como la clase cuota, que va a modelar el comportamiento de las cuotas dentro de la aplicación que se tiene intención de desarrollar.

Se debe mencionar que en el modelo del dominio se representa todo el alcance del problema desde un punto de vista conceptual, por lo que probablemente aparezcan entidades o relaciones que sean modificadas o sustituidas en diferentes diagramas posteriores en la etapa de diseño según sea necesario. En nuestro caso podemos ver la entidad "configuración de redes sociales" que modelaría todo lo necesario para los requisitos de enlaces con redes sociales. Esto es necesario aunque hayamos previsto que algunos requisitos (como este ejemplo) queden fuera de la solución, ya que probablemente haya cambios y no podemos asegurar que, efectivamente, la planificación se cumpla al pie de la letra, por lo que necesitamos crear un modelo del dominio que cubra absolutamente todos los requisitos obtenidos anteriormente.

2.8.- Análisis de la interfaz

La interfaz es un apartado complicado en cualquier aplicación de cualquier tipo, y en la que se ha propuesto en particular también lo es.

Nuestra aplicación tendrá dos perfiles de usuario, por lo que se debe hacer que la interfaz muestre a cada perfil de usuario lo que realmente debe (o necesita) ver.

En este punto se tienen varios retos, por una parte uno de los perfiles de usuario tiene, notablemente más, mucha más interacción con la aplicación web que el otro, pero tiene que ser eficiente a la hora de realizar su trabajo, mientras que el que tiene menos interacción con la aplicación no debe percibir una interfaz simplista. Crear una aplicación SPA (del inglés, Single Page Application) solucionaría este primer problema.

Otro reto será el de crear una interfaz atractiva. La propuesta de aplicación va centrada al público más joven de la sociedad (es una asociación juvenil), por lo que una interfaz más elegante y sobria, que podría ser una opción en caso de contar con un público más adulto, no parece ser una opción demasiado acertada.

Por último se debe mencionar que, tal vez por suerte, no hay ningún usuario que tenga alguna discapacidad o necesidad especial que precise de atención específica por parte de la interfaz (que sería el caso de personas con problemas de visión y similares). Aunque podría librar, en principio, de invertir un esfuerzo extra en la interfaz, no evita que prestemos cierta atención en el uso de colores, formas, audio y demás. Se debe encontrar un balance entre crear una interfaz atractiva para un grupo juvenil con una buena elección de colores y formas que evite que la vista se canse o que sea difícil percibir partes de la interfaz, requiriendo esfuerzo extra de nuestra vista para ello.

Para estos dos últimos puntos la solución radica en usar estándares. Bootstrap proporciona diferentes plantillas que nos darán diferentes juegos de colores y formas. No

es descabellado pensar que los desarrolladores de Bootstrap tengan más experiencia y conocimiento sobre el uso de colores y formas, por lo que se proporciona algo más de calidad a la solución.

3. Diseño

En este apartado nos centraremos en el diseño de la solución, tanto del modelo como de la interfaz. Empezaremos primero usando diferentes diagramas (como el diagrama entidad-relación) para mostrar la estructura de la información mediante entidades y relaciones a la vez que se comentarán los aspectos más importantes del diseño de estos así como su funcionamiento. Más tarde se pasará a explicar la propuesta de interfaz. Por último haremos mención al plan de pruebas, donde mostraremos definiremos cuales son los aspectos claves que queremos cubrir.

3.1.- Diagrama Entidad-Relación

El diagrama entidad-relación es una herramienta totalmente necesaria para esbozar el diseño de una base de datos. A partir del modelo de dominio de la información se observa la necesidad de un sistema de persistencia de datos, y la solución elegida ha sido una base de datos relacional.

Comentamos con antelación que se emplearía una tecnología ORM que nos va a permitir generar este apartado de una manera automática si se parte de las entidades del modelo del dominio. Sin embargo es interesante comentar cómo es la estructura generada por este sistema y analizar su forma normal para detectar posibles redundancias innecesarias en los datos.

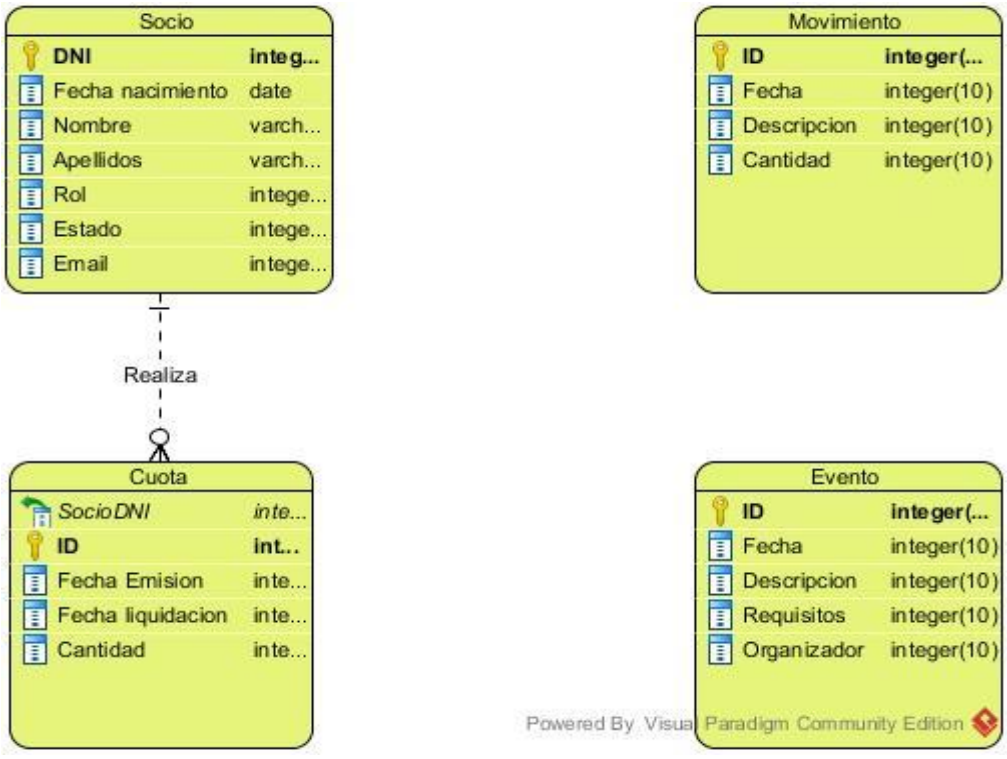


Ilustración 2: Diagrama Entidad-relación

Como podemos ver, se nos ha generado (o se generará, más bien) 4 entidades, la superior izquierda corresponde al objeto ‘Socio’, que es el encargado de modelar el comportamiento deseado de estos. Podemos ver una relación con otra entidad, ‘Cuota’, que modela el comportamiento del objeto homónimo, encargado de modelar las diferentes cuotas de cada uno de los socios.

Las otras dos entidades, aisladas, no tienen relación alguna con las primeras, esto no quiere decir que en nuestra aplicación no se relacionen de ninguna manera con el entorno; simplemente no hay necesidad de almacenar una relación entre otras entidades persistentes.

En este punto es imposible no preguntarse si JPA ha sido capaz de crear una base de datos normalizada hasta, al menos, tercera forma normal. Sabemos que, como cada entidad tiene su clave, está en primera forma normal, además, como no hay ninguna compuesta, tenemos asegurada la segunda forma normal.

Para alcanzar la tercera forma normal es necesario que esté en segunda forma normal, como se expone en el apartado 4.3 de [9], lo que es cierto en este caso, y que además no haya dependencias transitivas de los atributos no claves a la clave de cada una de las relaciones. Pues bien, un vistazo a los atributos de las entidades nos muestra que sólo hay un posible caso que habría que corregir, en la entidad ‘cuota’. Con los atributos ‘fecha de liquidación’ y ‘fecha de emisión’ podríamos tener este problema, ya que en una factura una fecha de liquidación depende de una de emisión. Sin embargo, tenemos suerte en este aspecto ya que, uno de los requisitos de las cuotas de los socios era, después de los cambios, que las cuotas pudiesen ser liquidadas en cualquier momento, independientemente de su fecha de emisión. Esto es así porque hay socios que están ausentes durante bastante tiempo y, cuando vuelven, liquidan varias facturas a la vez, si es necesario.

3.2.- Diagrama de Clases

El diagrama de clases es una herramienta fundamental para comprender cómo se estructura y organiza el código. En ella plasmamos las clases creadas y las relaciones entre ellas, además de los atributos y métodos necesarios para realizar la comunicación y colaboración necesaria.

El diagrama de clases puede, y de hecho ocurre, extenderse y resultar complejo. En este caso, la complejidad viene dada por el número de clases presentes y no por la dificultad para comprender el funcionamiento de estas. Pasaremos a comentar cada una de ellas (o mejor dicho, cada uno de los módulos, ya que en algunos casos, conocido un

representante de un módulo, conocidos todos).

Este apartado se organizará en varios sub-apartados ya que hay que diferenciar entre la aplicación cliente y la aplicación servidor, y comentar también la evolución en cada una de las iteraciones.

3.2.1.- Diagrama de Clases del servidor

En la siguiente figura se puede observar el diagrama de clases que se ha diseñado para la aplicación servidor.

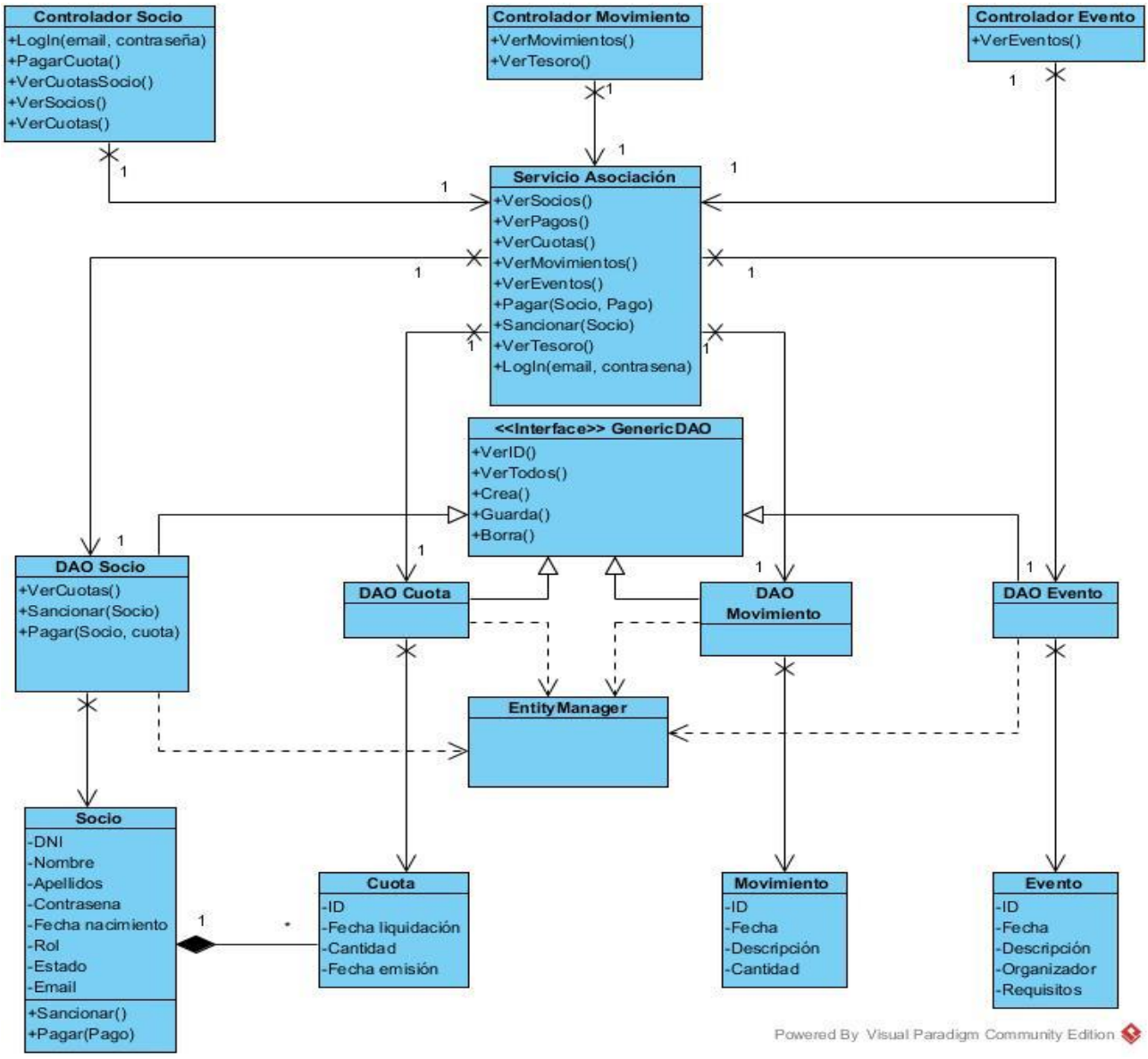


Ilustración 3: Diagrama de clases de la aplicación servidor

Pasemos a desmenuzar el diagrama y comentemos cada una de las clases, poniendo especial énfasis en los motivos que nos han llevado a incorporarlas.

Las clases ubicadas en la parte más baja del diagrama son las más obvias y las que se derivan lógicamente de las historias de usuario y requisitos no funcionales. Estas son 'Socio', 'Cuota', 'Movimiento', y 'Evento'. Podemos ver que la clase 'Socio' se encarga de gestionar las características de cada uno de los socios de la asociación, así como las cuotas de cada uno de los socios. 'Movimiento' es la clase que se encarga de la gestión del movimiento de dinero diario de la asociación (por ejemplo, una venta de un café o la compra de un proyector) mientras que 'Evento' hace lo propio con los eventos que la asociación realiza a lo largo de su haber diario.

Cada una de ellas está relacionada con un DAO[14] (del inglés Data Access Object) específico no bidireccional, ya que la utilidad del DAO es para la clase 'Servicio Asociación'. El patrón de diseño DAO proporciona una capa nueva que suministra una interfaz para evitar que sea necesario que la clase que los use conozca los detalles de persistencia de las clases que el DAO maneja. El patrón DAO es considerado una buena práctica, porque reduce el acoplamiento del controlador con los sistemas de persistencia además de las ventajas ya mencionadas.

La interfaz *GenericDAO* está ahí para asegurarnos que los DAO que hemos creado tiene una interfaz común y se usan de la misma manera; la coherencia en una aplicación hace que trabajar con ella sea mucho más sencillo, y eso sin mencionar la facilidad para comprenderla.

En el apartado 2.1 mencionamos que emplearíamos *Spring Framework* dentro del apartado de tecnologías. Una de las clases que incorpora, y que nosotros utilizaremos, es la clase *EntityManager*. Esta clase se ocupa de almacenar y recuperar información de una base de datos (o, en nuestro caso, directamente objetos). Los DAO generados la emplean para almacenar y recuperar objetos directamente de la base de datos mediante sus servicios. Cabe mencionar que, aunque esté presente en el diagrama porque es una clase esencial, no ha sido diseñada por nosotros, solamente utilizada.

La clase 'Servicio Asociación' es el corazón de la aplicación. Es la encargada de proporcionar el servicio que deseamos, de realizar la mayor parte del cómputo y de los cálculos de cada uno de los servicios proporcionados por la aplicación servidor. Entre ellos podemos mencionar crear un nuevo usuario o consultar los eventos existentes.

Sobre la clase Servicio actúan tres controladores. Cada uno de ellos se ocupa de la

14 https://en.wikipedia.org/wiki/Data_access_object

gestión de una de las entidades (salvo el de socio puesto que las cuotas están ligadas a los socios por lo que si queremos trabajar con ellas necesariamente tenemos que hacerlo con los socios pertinentes) y de traducir peticiones http a invocaciones entre objetos.

Hay que comentar que el diagrama ha ido evolucionando conforme avanzaba el proyecto partiendo del modelo del dominio. Esto ha hecho que, en la segunda entrega, se eliminase una clase existente (Información) que controlaba, junto a su DAO, las características de la siguiente cuota. Esto se tuvo que eliminar puesto que hubo un cambio en el requisito que hacía que las cuotas fuesen independientes las unas de las otras, por lo que la información de las cuotas se trasladó a la clase cuota, que era la responsable, entonces, de la información de estas.

3.2.2.- Diagrama de Clases del cliente

El diagrama de clases del cliente, figura 4, es algo diferente al del servidor., aunque ambos mantienen la misma filosofía. Es importante mencionar que en este caso no tenemos diferentes versiones según las entregas, tenemos un único diagrama que ha ido evolucionando según avanzaba el proyecto debido a, en gran medida, que precisaba de que la aplicación servidor estuviese terminada en su mayoría.

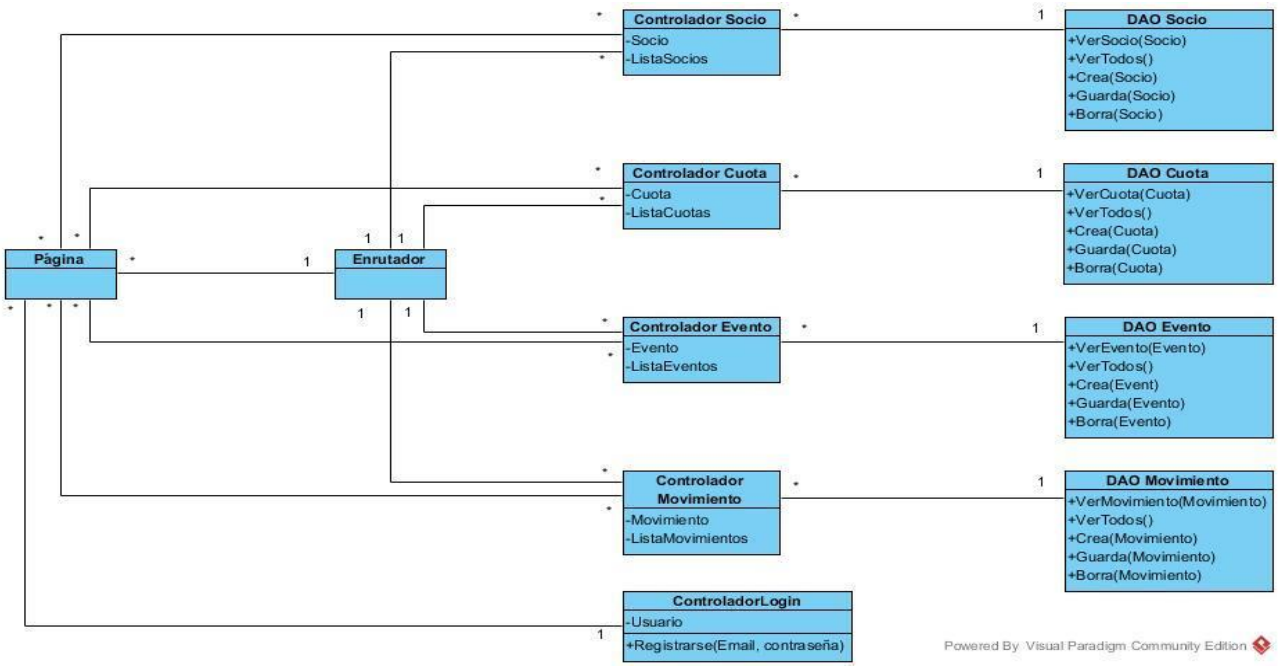


Ilustración 4: Diagrama de clases de la aplicación cliente

Pasemos a comentar cada uno de los aspectos del diagrama.

Los DAO del cliente son similares a los DAO del servidor con dos excepciones. La primera excepción es que su objetivo no es la persistencia de la información, su objetivo es la transferencia de información con el servidor; estas clases no son las que crean o mantienen entidades en la base de datos, son las que crean peticiones al servidor para que este las cree o mantenga. La segunda excepción es que estos permiten desacoplar la lógica de negocio, implementada en el servidor, del cliente.

Los controladores son mucho más sencillos que los del servidor. Ahora hay 4 en lugar de 3 como antes. Esto es así porque el servidor es el que se encarga de mantener la relación entre cuotas y socios, liberando al cliente de este requerimiento. Esto nos permite crear controladores que únicamente necesitan trabajar con un DAO para que les suministre información del servidor, y una vista en la que mostrar dicha información.

El lector ya se habrá percatado del controlador especial “Login”. Este controlador es necesario porque el servicio que proporcionamos es un servicio REST. Los servicios de este tipo son “stateless”, lo que quiere decir que no va a guardar información sobre sesiones Esto es necesario que lo haga el cliente, que debe almacenar información del usuario que está trabajando en este momento. Además, el servidor necesita que el usuario que va a acceder a un determinado recurso utilice unas credenciales en forma de un correo electrónico y una contraseña. Esta clase también se encargará de ello.

A continuación hablaremos del enrutador, una clase especial de Angular que nos va a permitir crear una aplicación SPA (Single Page Application). Para explicar esta clase necesitamos comprender cómo funciona el flujo en una aplicación Angular.

La vista (reflejada como clase, aunque en realidad es cualquier página HTML que se cargue) es la primera que se procesa, esta necesita de uno o varios controladores. Los controladores entonces entran en acción. Estos intercambian información, normalmente objetos, con los DAO que, a su vez, trabajan con el servidor. Una vez tienen en su poder la información necesaria modifican la vista que los invocó si fuese necesario. En una aplicación SPA sólo hay una página donde un fragmento de ella es sustituido dinámicamente por otro. Para que esto sea posible el enrutador debe elegir qué fragmento se cambia por cual otro (normalmente por medio de la url presente) y que controlador asignamos al nuevo fragmento cargado.

Microsoft utiliza un gráfico muy interesante para explicar el funcionamiento de una aplicación SPA[15].

15 https://msdn.microsoft.com/en-us/magazine/dn463786.aspx

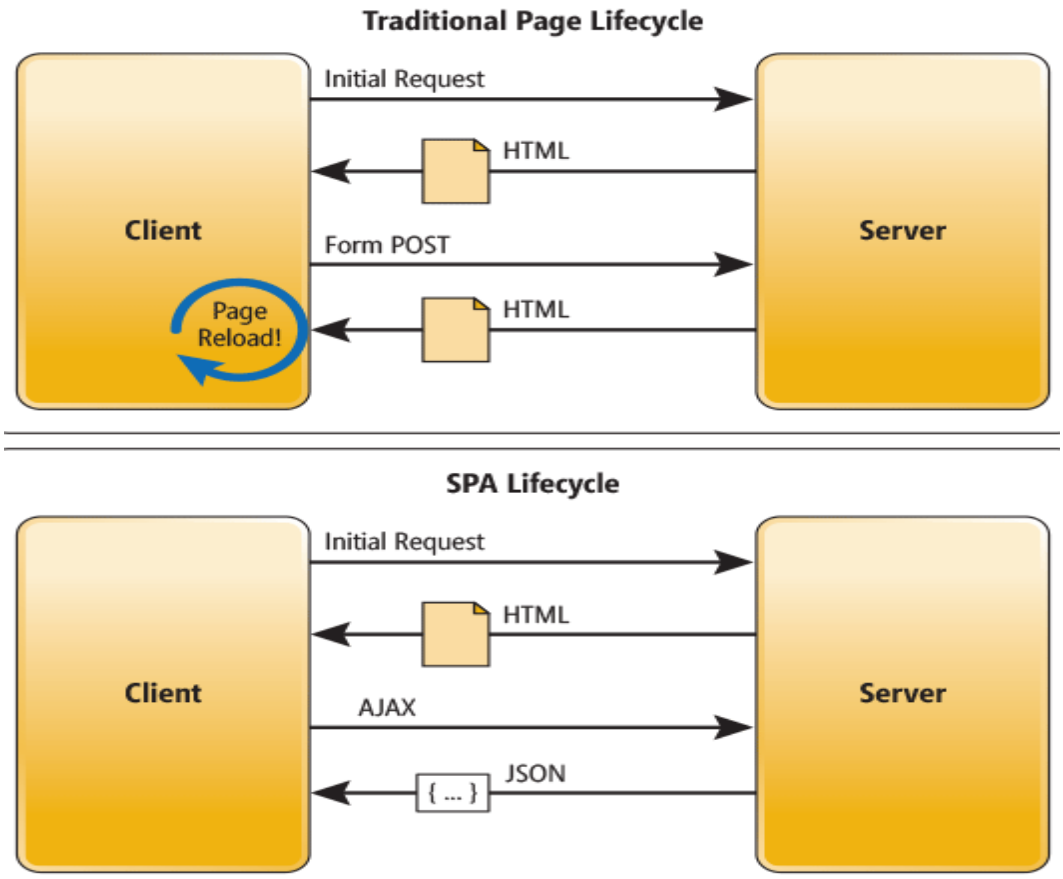


Ilustración 5: Ciclo de vida tradicional vs ciclo de vida SPA de una página web

En la imagen se puede apreciar el ciclo de una página web tradicional frente a una SPA. En la página SPA no es necesario recargar la página actual o cargar otra página, simplemente se añade la información por la petición asíncrona (AJAX) que se ha obtenido en el JSON recibido en el momento en el que se recibe; constantemente se están intercambiando fragmentos de la página en lugar de cargar una nueva.

3.3.- Diagramas de secuencia

Los diagramas de secuencia son una excelente herramienta que nos va a permitir mostrar con claridad la interacción entre los diferentes componentes de la aplicación. Aunque no se mostrarán todos los existentes (porque en realidad muchos son similares entre sí) sí que se van a explicar los más representativos.

Hay que tener en cuenta que, como en el punto anterior, se va a realizar una división entre cliente y servidor para finalizar con uno que muestra la interacción entre ambos.

3.3.1.- Diagramas de Secuencia del servidor.

En este punto mostraremos y comentaremos 3 diagramas de secuencia. El objetivo de este punto es mostrar al lector la comunicación interna entre las diferentes clases de la aplicación servidor a la hora de realizar su trabajo.

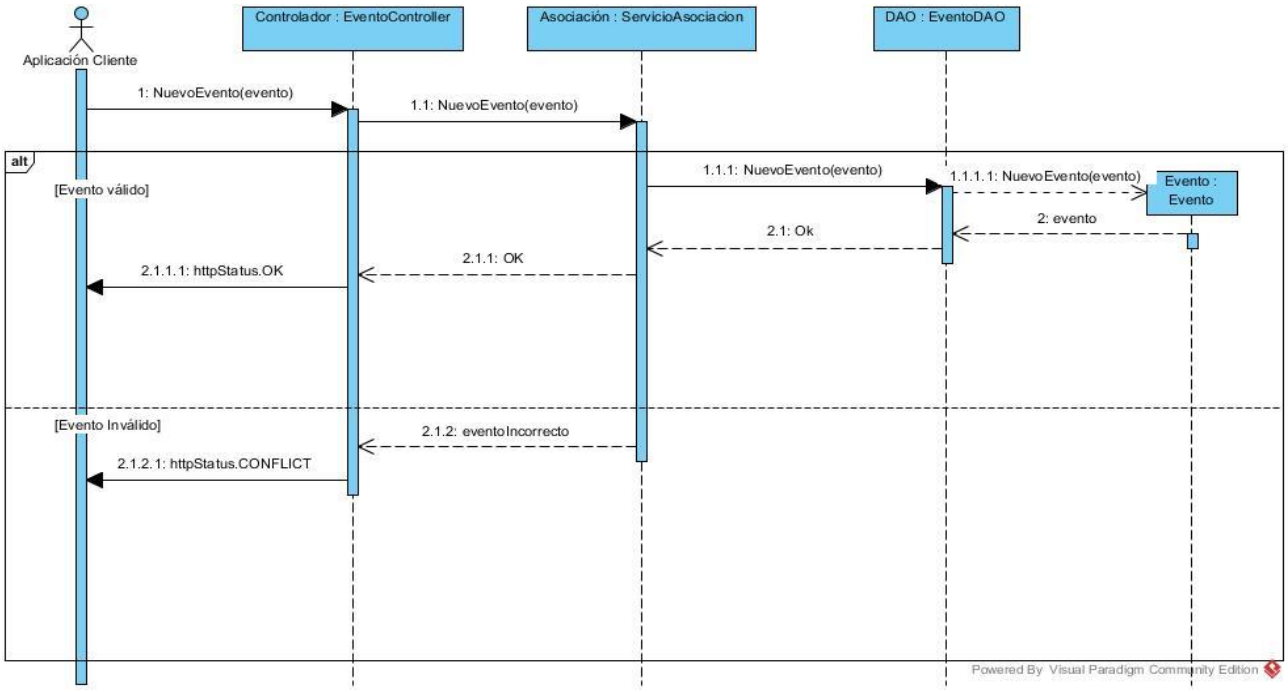


Ilustración 6: Diagrama de secuencia 1: crear un nuevo evento

En este diagrama podemos observar la jerarquía de invocaciones y flujo de datos entre las clases presentes en la creación de un evento. Crear un nuevo evento puede no ser posible (como, por ejemplo, en el caso de que intentemos crear un evento sin especificar una fecha de comienzo), por lo que se ha diseñado un escenario alternativo para esta casuística.

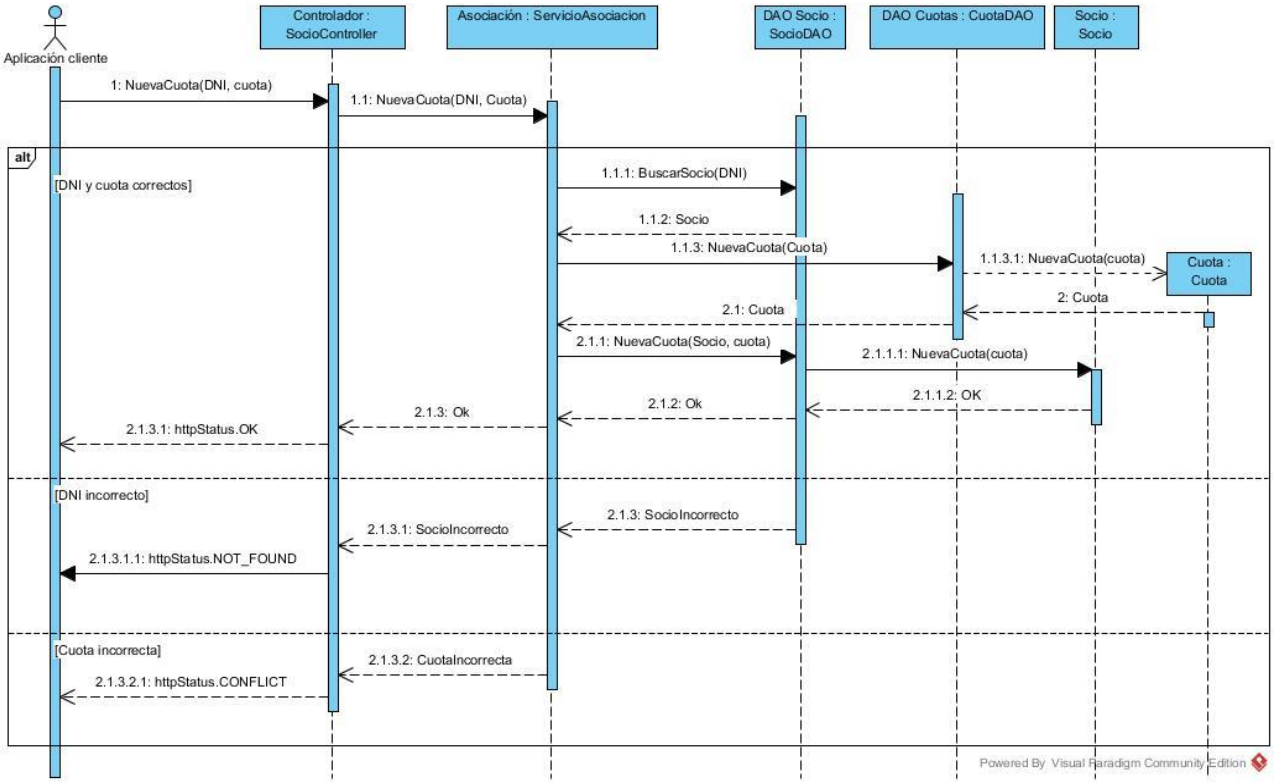


Ilustración 7: Diagrama de secuencia 2: crear una nueva cuota

Este es uno de los diagramas más complejos de la aplicación servidor. Como en el anterior, se puede observar la jerarquía y flujo de datos entre clases a la hora de crear una nueva cuota. De igual manera podemos tener problemas a la hora de crearla, en este caso contamos con dos escenarios alternativos que muestran las acciones que se seguirán cuando la cuota creada sea incorrecta o se cree para un socio incorrecto.

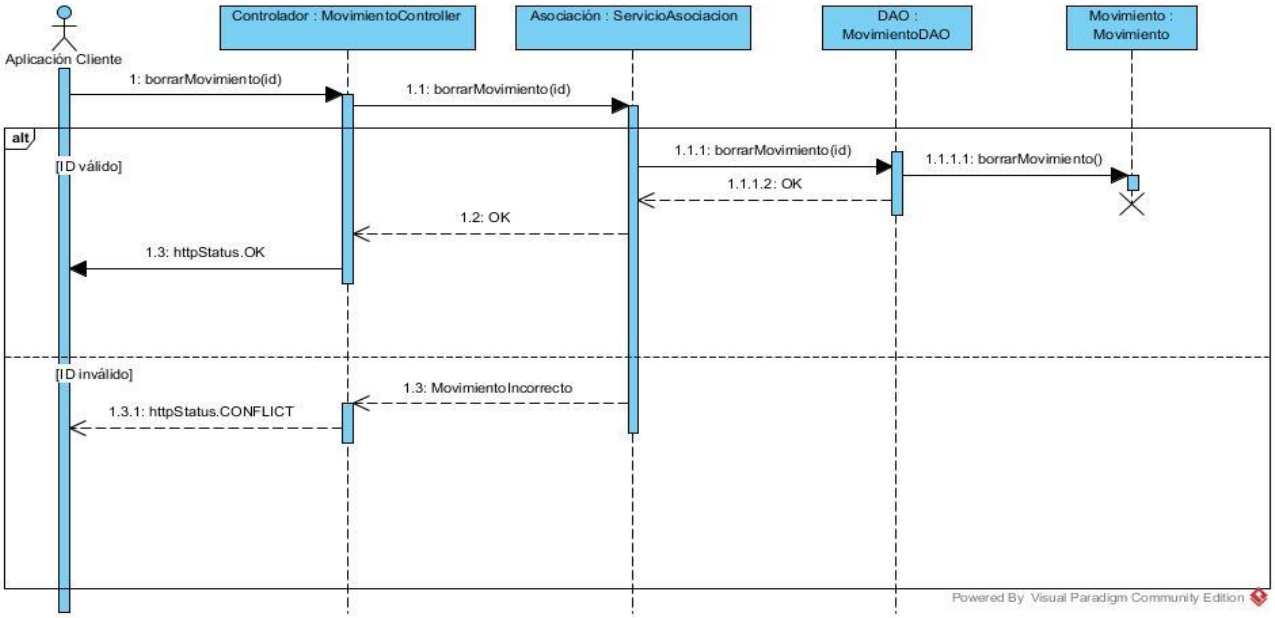


Ilustración 8: Diagrama de secuencia 3: borrar un movimiento

Por último un diagrama de secuencia que muestre cómo se elimina un movimiento. Es interesante mostrar un diagrama donde se aprecie, también, la manera en que se elimina una entidad. No es necesario comentar mucho más ya que es bastante similar a los anteriores debido al diseño MVC propuesto, facilitando que las operaciones se realicen de una manera muy similar.

3.3.2.- Diagramas de Secuencia del cliente.

Como ya se ha comentado con anterioridad, tenemos dos aplicaciones colaborando en nuestro diseño. Para la parte del cliente hemos creado dos diagramas que se muestran a continuación.

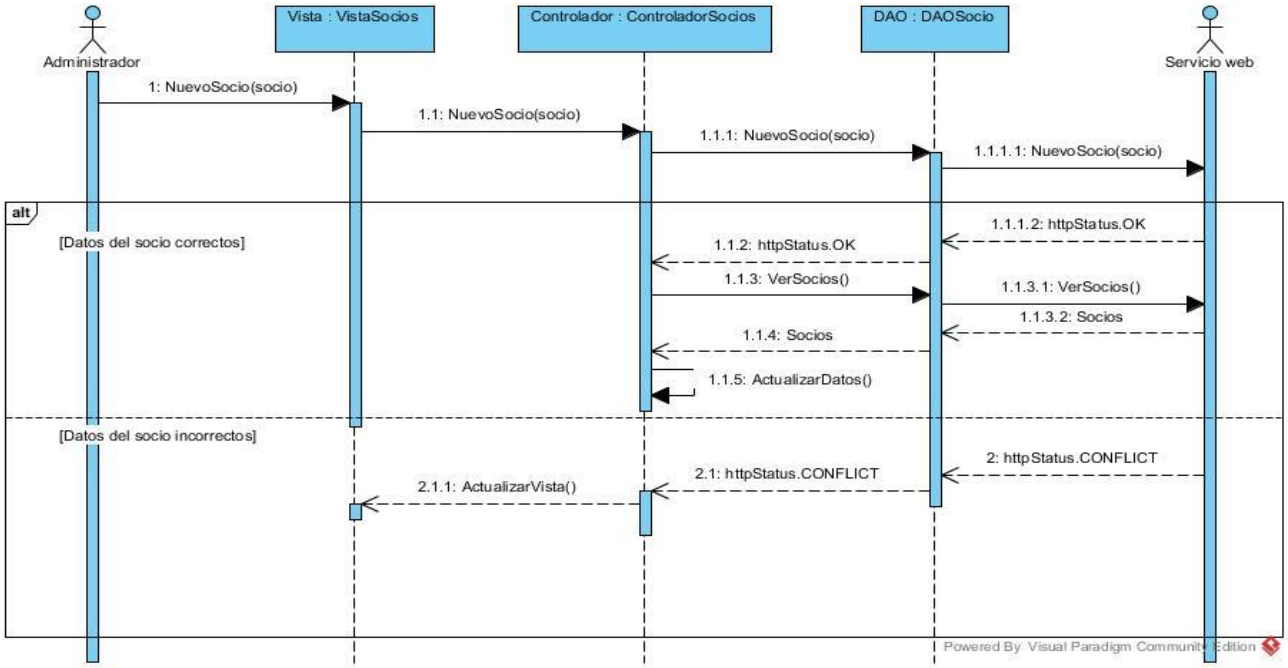


Ilustración 9: Diagrama de secuencia 4: crear un nuevo socio

La comunicación y flujo de datos es muy similar a la existente en la parte del servidor. Una diferencia notable es la comunicación que la aplicación cliente tiene entre el usuario y el servidor. Esto muestra la necesidad de crear una interfaz correcta.

En este diagrama en concreto mostramos cual es el proceso de crear un socio en la parte del cliente.

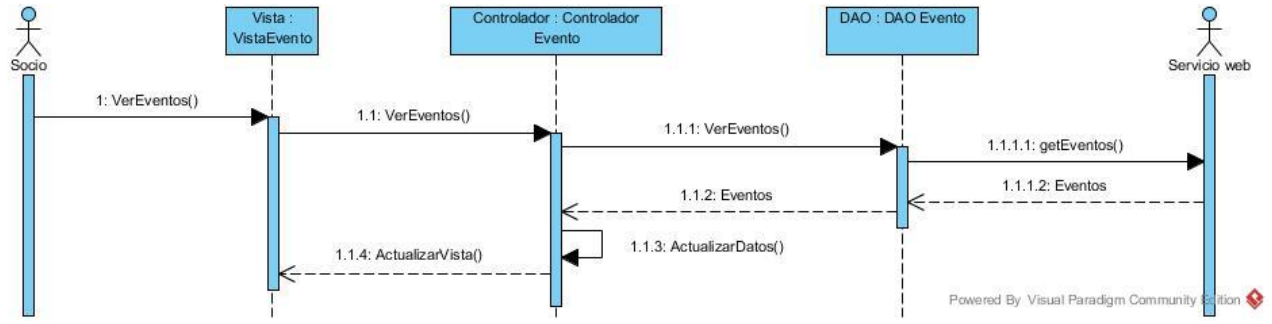


Ilustración 10: Diagrama de secuencia 5: ver los eventos

En este diagrama de secuencia se puede observar la visualización de eventos en la parte del cliente. Ver los eventos de la asociación es algo muy sencillo que no requiere de escenarios alternativos.

3.3.3.- Diagrama de secuencia conjunto.

Para finalizar se va a mostrar un diagrama de secuencia que ilustre la comunicación entre cliente y servidor a la hora de visualizar un evento.

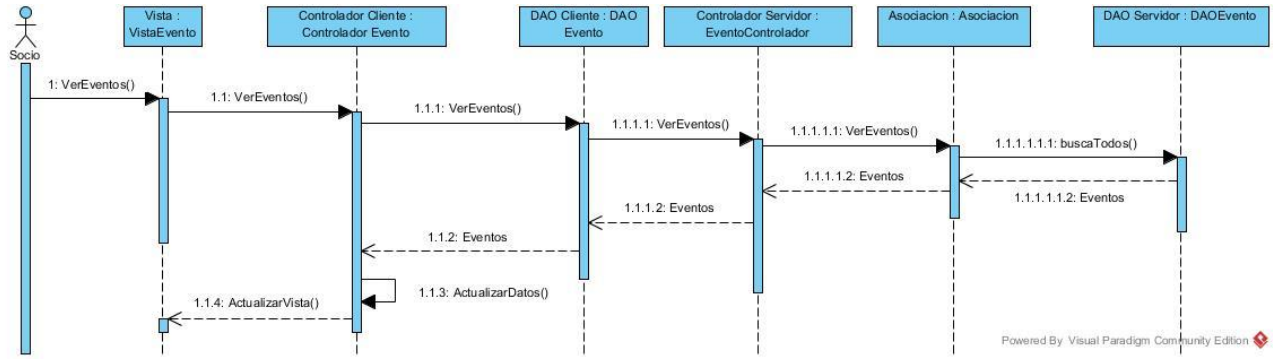


Ilustración 11: Diagrama de secuencia 6: ver los eventos (cliente-servidor)

Se ha elegido también un diagrama de una de las acciones más sencillas. Como podemos ver hay numerosas entidades trabajando entre sí, por lo que se crearán muchas llamadas. Lo más interesante de este diagrama es mostrar cómo colaboran entre sí todas las entidades de la solución a la hora de realizar una acción en concreto.

3.4.- Diseño de la Interfaz

La interfaz es, posiblemente, la parte más delicada del desarrollo de cualquier aplicación, pues cualquier usuario da por hecho que se le va a proporcionar una interfaz sencilla, elegante, intuitiva y robusta. El equipo de desarrollo debe garantizar una interfaz que cumpla con lo aquí especificado así como con los dos requisitos no funcionales (NF1 y NF2) de interfaz adaptativa y facilidad de uso.

Además de la complejidad intrínseca de la creación de una buena interfaz, en el desarrollo de una aplicación web encontramos el componente de “subjetividad” en esta. Como ya hemos mencionado en varias ocasiones, la aplicación web estará compuesta de dos aplicaciones colaborando entre sí para realizar su labor.

Aunque también hay otra interfaz, una interfaz REST orientada a cualquier cliente que trabaje con el servicio proporcionado por el servidor. Esta interfaz no está orientada al usuario final, aunque no por ello no debe de cumplir los dos requisitos no funcionales, la diferencia está en que para lograrlo se empleará la arquitectura REST junto al estándar http, lo que lo hará más simple y convencional que otras alternativas. Esta interfaz, sin embargo, será con la que interactúe la aplicación cliente, pues aunque será robusta y sencilla carece de todas las demás características necesarias para ser una interfaz para el usuario. La interfaz de la aplicación cliente será la que nos propicie más de un quebradero de cabeza.

Esta interfaz REST va a permitir que se realicen peticiones GET, POST, DELETE y PUT para obtener, enviar, eliminar y modificar información respectivamente, de los cuales el lector puede indagar más en [10]. Como método de transferencia, se va a utilizar el estándar de Javascript, el envío de información mediante JSON.

El procesamiento en la aplicación cliente es menor que en el servicio, pero el esfuerzo en la interfaz es inmensamente mayor que es este último. Para el desarrollo de la interfaz de la aplicación cliente hemos optado por la creación de un prototipo. El prototipo nos va a permitir libertad en el desarrollo de la interfaz y, a la vez, flexibles en cuanto a cualquier cambio. El prototipo será desechable (no todos lo son, aunque nosotros hemos decidido que el nuestro lo sea) ya que las tecnologías seleccionadas permiten desarrollarlo con relativa facilidad.

Aunque el prototipo esconde más valor que el mencionado anteriormente. Un buen prototipo es una buena herramienta de obtención de requisitos. Aunque una buena parte de los requisitos se deberían haber obtenido antes de la creación del prototipo (el prototipo es una "idea" básica de la aplicación, y para esta idea es necesario conocimiento, en nuestro caso conocimiento de los requisitos) podremos descubrir algunos nuevos además de refinar los ya existentes o modificarlos según la respuesta del cliente a nuestro prototipo desarrollado.

Pero antes de crear el prototipo se debe realizar una propuesta inicial de la interfaz como guía para la implementación de este. Para ello realizaremos un *storyboard* que permitirá hacernos una idea mental de la interfaz que se quiere crear, así como localizar posibles errores y oportunidades de mejora.

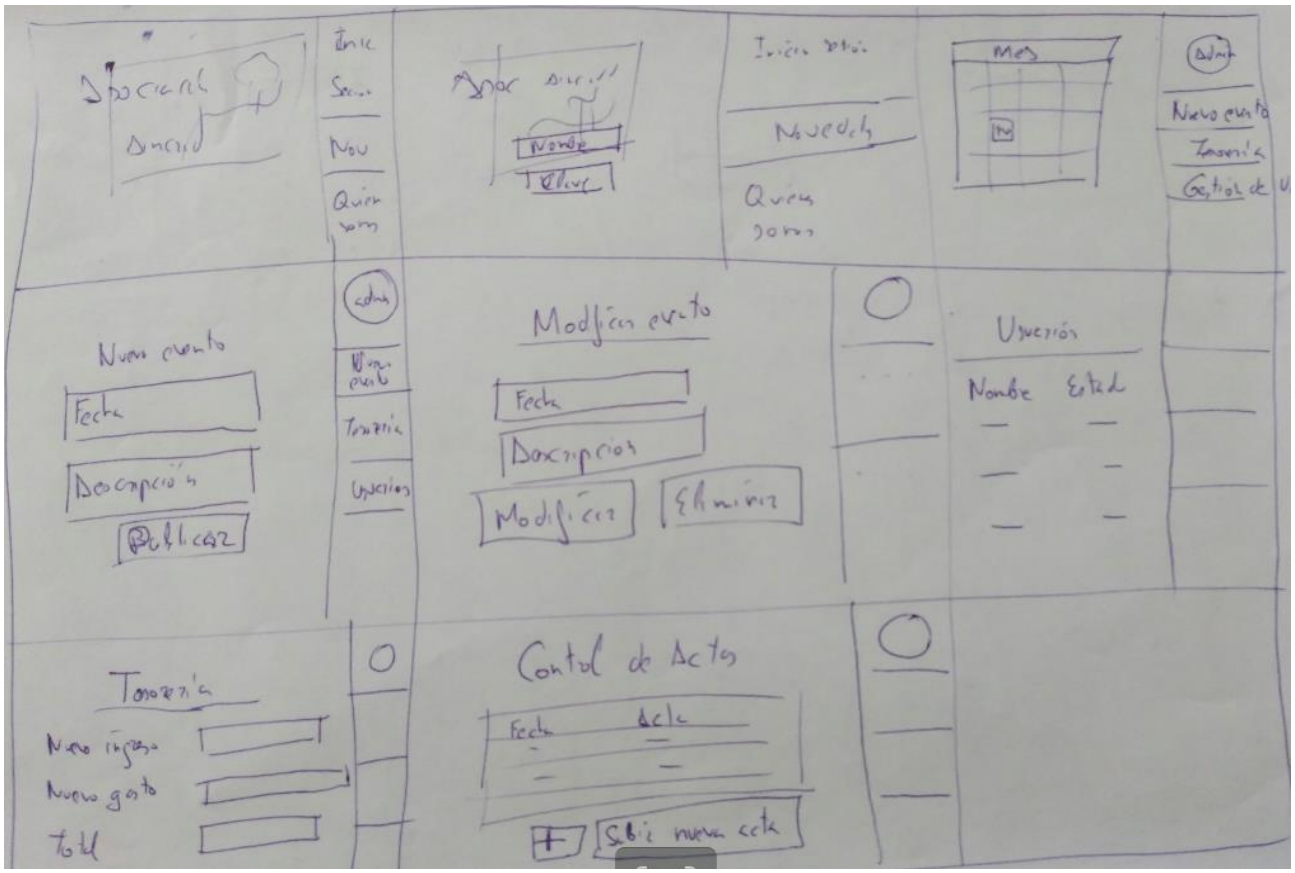


Ilustración 12: Storyboard

Como se puede observar, cada una de las viñetas del *storyboard* corresponde con cada una de las opciones de la barra de navegación que hay a la derecha de cada una. Al analizar el trabajo realizado pudimos observar que la barra de navegación iría mejor en la parte superior y que los textos de cada una de las opciones debería ser más pequeño.

Pasemos ahora a analizar unas pocas imágenes del prototipo. El prototipo cubría la totalidad (salvo modificaciones) de la interfaz del cliente, por lo que era extenso. En este punto no pretendemos extendernos demasiado, por lo que simplemente se mostrarán y comentarán ligeramente las imágenes seleccionadas.

Ilustración 13: Calendario de eventos del prototipo

En la ilustración 13 se puede ver la pantalla de inicio de un socio, en la que había un calendario a modo de prueba diseñado, el calendario se modificó mucho en la versión final, aunque la idea se mantuvo.

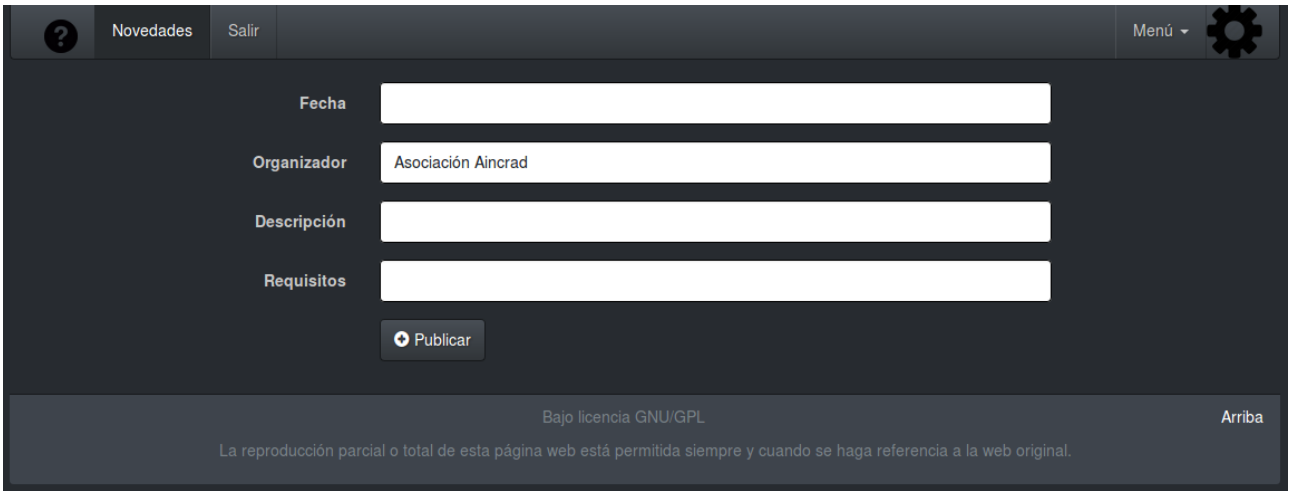


Ilustración 14: Pantalla de inicio de un socio del prototipo

Aquí, en la ilustración 14, se puede ver como el administrador crea un evento nuevo, la versión final es similar.

Ilustración 15: Listado de socios del prototipo

Por último podemos ver la gestión de los socios en la ilustración 15, aunque se mantiene bastante similar, se han añadido más campos que son necesarios. En el manual de usuario pueden observarse estas modificaciones con las diferentes imágenes aportadas.

3.5.- Plan de pruebas

Las pruebas son una parte esencial de la aplicación y, como no podría ser de otra manera, necesitaremos herramientas y estudiar con detalle cuáles serán las más adecuadas.

Habrá dos tipos de pruebas: por un lado usaremos pruebas de unidad para encontrar el mayor número posible de errores en el software; por otro lado probaremos que se cumplen los requisitos especificados en el diseño de nuestra aplicación. Este es un punto importante porque se podrá medir el nivel al cual nuestra aplicación se ajusta a las necesidades del usuario. Las pruebas de unidad medirán la corrección de la aplicación. Las pruebas de aceptación el nivel en que hemos desarrollado la aplicación correcta.

Para las pruebas de unidad se usará, además, distintas herramientas (detalladas más adelante en el punto 5) que nos ayudarán en esta tarea. Para las pruebas de aceptación es imprescindible contar con el cliente, pues él es quien debe valorar en qué medida la aplicación es correcta. No obstante, para estas últimas el equipo de desarrollo cuenta con los diferentes diagramas desarrollados, así como las historias de usuario, que van a permitir adelantarnos al juicio del cliente estimando nosotros mismos un grado de corrección a priori, posibilitando la detección más o menos temprana de posibles errores.

4. Implementación

4.1.- Arquitectura

Como ya se ha indicado con anterioridad, la solución está formada por diferentes capas y componentes colaborando entre sí. Para poder comprender mejor las relaciones entre estos diseñaremos un diagrama arquitectónico que muestre las relaciones entre los componentes.

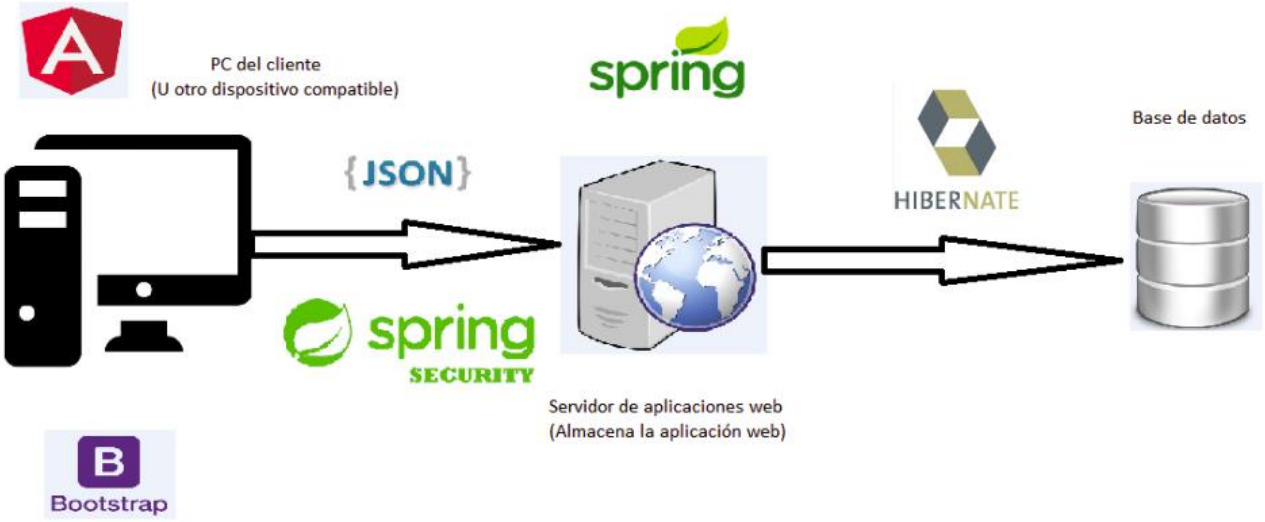


Ilustración 16: Diagrama arquitectónico

Como podemos ver en la ilustración 16, la aplicación web estará almacenada en un servidor de aplicaciones web, este utilizará Spring framework.

La conexión con la base de datos se hará por medio de Hibernate, que actuará como intermediario.

En el equipo del cliente (un ordenador, tablet o cualquier otro dispositivo compatible) se ejecutará la aplicación cliente (la página web). Bootstrap y Angular se ejecutarán también en este.

Para finalizar, la conexión entre servidor y cliente se hará mediante peticiones y objetos JSON. Spring Security será el encargado de proporcionar seguridad a todo el proceso.

4.2.- Detalles sobre implementación

Llegamos al que sea, posiblemente, el punto más técnico de todo el documento, el objetivo de este punto es comentar los aspectos de la metodología y tecnologías empleadas en el desarrollo de la solución.

En primer lugar se comenzará comentando las dependencias necesarias así como la versión de cada una, luego se irán nombrando y explicando brevemente las tecnologías empleadas como las configuraciones necesarias. Más tarde explicaremos el uso de diferentes patrones y decisiones aplicadas en el desarrollo, como el uso de controladores o el patrón de diseño MVC. Luego se mostrará el uso de cada una mostrando, además, ejemplos extraídos de la misma aplicación para comprender mejor su funcionamiento y su utilidad. Para finalizar se comentarán los problemas que hemos encontrado en la integración de todos estos componentes y cómo han funcionado en conjunto.

Las dependencias necesarias para la aplicación son las siguientes:

-Bootstrap versión 2.5.0.
-Hibernate jpa 2.0 versión 1.0.1.
-Jackson versión 2.6.5.
-Spring versión 4.2.5 (de entre todos sus módulos, que son muchos, utilizamos: core, aop, aspects, beans, context-support, jdbc, jms, orm, config y web)
-Spring security versión 3.0.5.

Además se utilizan diferentes bibliotecas, algunas se descargan dinámicamente en tiempo de ejecución, mientras que otras ya están presentes cuando la aplicación se despliega.

Bibliotecas descargadas previamente:

-aopalliance-1.0.jar
-jboss-transaction-api-1.0.0.jar
-json-simple-1.1.1.jarç

Y las que se descargan cuando son necesarias:

-angular versión 1.6.4.
-angular-route versión 1.6.4.
-angular-locale-es-es versión 1.6.4.

Para acabar debemos hacer una mención especial a dos hojas de estilo personalizadas descargadas previamente.

-bootstrap.min.css, modificada con colores más oscuros y del agrado del cliente.
-calendar.css, modificada para mostrar los eventos del calendario de una forma más agradable y destacada.

En este punto se ha de comentar el porqué de los controladores y cómo se han

implementado usando las diferentes tecnologías elegidas.

Hemos aplicado el patrón de diseño MVC (Modelo-Vista-Controlador, del que hay bastante información en el recurso online [11]) tanto a la aplicación cliente como la aplicación servidor. Esto es así porque este patrón de diseño permite reducir el acoplamiento y aumentar la cohesión entre las diferentes partes de la aplicación, lo que nos va a permitir desarrollar una aplicación final de más calidad. Es un patrón de diseño con multitud de *frameworks* que lo incorporan directamente y está muy extendido y recomendado en el desarrollo de aplicaciones web. Aunque ya se habló de las clases controlador con anterioridad, es debido a este patrón por el cual se crearon.

Spring permite crear un controlador de una forma muy sencilla:

```
@Controller
@RequestMapping("/evento")
@SessionAttributes()
public class eventoController {
```

Y es muy sencillo enlazar un método del controlador con una petición HTTP

```
@RequestMapping(value = "", method = RequestMethod.POST, produces = "application/json; charset=utf8")
public ResponseEntity<Void> nuevoEvento(@RequestBody Evento ev) {
```

Angular también proporciona herramientas para los controladores, aunque son algo más complicados de manejar que en Spring, su declaración es muy sencilla; simplemente hay que añadirlos a un módulo contenedor de la siguiente forma:

```
.controller('SocioCtrl', ['$scope', '$routePa
        var self = this;
        this.socio = {};
        this.socios = [];
```

Y tratarlos como una clase más.

Ahora pasaremos a comentar brevemente dónde y para qué se usan cada una de las tecnologías mencionadas anteriormente. Además de mostrar ejemplos de cada una.

Spring. Spring es un framework de programación que cambia totalmente la manera en la que desarrollamos una aplicación. Está pensado para utilizarse en aplicaciones empresariales aunque se puede utilizar en cualquier otra. Proporciona una innumerable cantidad de herramientas que nos van a facilitar enormemente nuestro trabajo como desarrolladores. Sin embargo todo esto pasa por la necesidad de conocer el framework en cuestión.

Intentar explicar en unos pocos párrafos el funcionamiento de Spring es extremadamente ambicioso, así como la configuración necesaria para empezar a trabajar con él. Sin embargo mostraremos los aspectos más importantes de esta última.

Spring puede configurarse de varias maneras, nosotros hemos elegido crear el archivo de contexto de Spring; *SpringDispatcher-servlet.xml* además de utilizar el archivo *web.xml.*
El fichero *application.Context.html* se usa para declarar beans. Tanto propios, los DAO o beans creados por nosotros,

```
<bean id="SocioDAO" class="com.morgal.tfg.model.SocioDAO" />
```

como predefinidos, como el gestor de transacciones.

```
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="emf"></property>
</bean>
```

En el archivo *web.xml* se configuran otras propiedades de Spring, como la inyección de dependencias. Además necesitamos especificar cual es el archivo de configuración que utilizamos (el archivo applicationContext.html)

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>
```

Una vez configurado, Spring se usa en infinidad de partes en nuestra aplicación. Para ello utilizamos anotaciones, una manera sencilla, elegante y compatible con la programación orientada a objetos. Por ejemplo, podemos definir transacciones (siempre que el gestor de transacciones haya sido debidamente configurado, por ejemplo en el archivo applicationContext.html),

```
@Transactional(propagation = Propagation.SUPPORTS, readOnly = true, rollbackFor = Throwable.class)
public class Asociacion {
```

utilizar la inyección de dependencias (con la cual los objetos especifican las relaciones con otros objetos, pero es Spring el que se encarga de “enlazar” las relaciones)

```
@Autowired
private EventoDAO DAOEvento;
```

o simplificarnos muchísimo la creación de controladores reduciendo la cantidad de código necesario al usar anotaciones.

```
@RequestMapping(value = "", method = RequestMethod.GET, produces = "application/json; charset=utf8")
public ResponseEntity< List<Evento>> verEventos() {
```

Spring Security. Dentro de Spring, Spring Security merece un apartado propio. La seguridad es cada vez más importante y Spring Security es una gran elección para ello. Spring Security nos va a permitir definir qué roles pueden acceder a qué recursos, rechazar conexiones, autentificar, etcétera.

Spring Security necesita una configuración similar aunque menos extensa que Spring. En la solución se ha optado por usar un archivo, Spring-Security.xml y configurar el archivo web.xml.
En el archivo web.xml necesitamos declarar el archivo Spring-Security.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/applicationContext.xml
        /WEB-INF/Spring-Security.xml
    </param-value>
</context-param>
```

y en el archivo Spring-Security.xml configuraremos todas las características deseadas, como los permisos para cada url

```
<intercept-url pattern="/aincrad/cuota/*" access="hasAnyRole('user','administrator')" method="GET" />
```

Bootstrap. Bootstrap se usa principalmente para el desarrollo de la interfaz de la aplicación cliente. En lugar de diseñar la apariencia estética de la interfaz utilizando la complejidad de CSS .css, se pueden utilizar clases de estilos ya creadas por expertos y configurarlas para que se adapten a necesidades del cliente. Utilizar Bootstrap es muy sencillo, simplemente tenemos que utilizar o diseñar una hoja de estilo y cargarla en nuestro archivo .html.

```
<link href="/TFG/css/bootstrap.min.css" rel="stylesheet">
```

Luego añadiremos las clases necesarias en el atributo "*class*" a nuestros elementos html y habremos terminado.

```
<ul class="nav navbar-nav navbar-right">
```

Hibernate. Se utiliza Hibernate como API para trabajar con bases de datos relacionales sin necesidad de utilizar JDBC. Hibernate utiliza directamente con JDBC permitiéndonos trabajar directamente con objetos en lugar de entidades, además muestra

mucho énfasis en el rendimiento, minimizando el consumo de recursos siempre que sea posible.

Trabajar con Hibernate es algo más complicado, puesto que tendremos que indicar a Spring que deseamos utilizarlo. Además se utilizará el bean de Spring "entityManager" para hacerlo de la manera más transparente posible y acercarlo al máximo a la programación orientada a objetos. Para ello debemos crear un bean "*dataSource*"

```
<bean id ="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
```

y otro "*jpaAdapter*"

```
<bean id="jpaAdapter" class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
```

para luego utilizarlo en un tercero, el mencionado anteriormente como "entityManager" dentro del archivo de contexto de Spring (en nuestro caso, applicationContext.html).

```
<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="packagesToScan" value="com.morgal.tfg.model"></property>
    <property name="dataSource" ref="dataSource"></property>
    <property name="jpaVendorAdapter" ref="jpaAdapter"></property>
</bean>
```

Usarlo ahora es muy sencillo, sólo tenemos que utilizar la inyección de dependencias de Spring al declararlo

```
public class SocioDAO implements Serializable{
    @PersistenceContext
    EntityManager em;
```

Y usarlo como un atributo de tipo objeto más de la clase, como por ejemplo en la siguiente función que elimina un socio dado de la base de datos.

```
public boolean borra(Socio sc) {
    try {
        em.remove(sc);
    } catch (Exception e) {
        throw new errorUsuarioInvalido();
    }
    return true;
}
```

Jackson. Antiguamente conocido como la biblioteca estándar de JSON es

extremadamente sencillo de usar. Una vez añadida la dependencia mencionada anteriormente los controladores la usarán para transformar los objetos de la aplicación a objetos JSON a la hora de transferir información con el exterior (en nuestro caso, la aplicación cliente) simplemente indicando en la parte del cuerpo de la petición y respuesta que el tipo de dato que manejamos es JSON.

```
@RequestMapping(value = "", method = RequestMethod.GET, produces = "application/json; charset=utf8")
```

Además podemos modificar ligeramente su comportamiento mediante anotaciones, con las que podemos, por ejemplo, indicar que no debe exportar un atributo "contraseña" al exterior.

```
@JsonProperty(access = Access.WRITE_ONLY)
@NotNull
private String contra;
```

5.- Pruebas

En este punto vamos a detallar las pruebas más importantes y los resultados obtenidos.

En el punto 3.1 se habló de cuál sería el plan, en este punto ahora hablaremos de cómo se ha desarrollado. Teníamos dos tipos de pruebas, que eran aceptación y unidad. Para las pruebas de unidad se han elegido pruebas de caja negra [12] mientras que para las de aceptación hemos optado por el juicio del cliente.

¿Por qué el juicio del cliente? porque hemos de comprobar el grado en el que hemos realizado la aplicación correcta. El equipo de desarrollo siempre trabaja intentando crear la mejor aplicación, por lo que siempre va a tener la sensación de que la aplicación es correcta, lo cual no tiene porqué ser cierto, ya que puede haber errores que el equipo de desarrollo no sea capaz de observar. Además no hay otro equipo externo para las pruebas (que sería lo ideal) y se ha de recordar que este tfg está desarrollado por un único alumno, lo cual se traduce en un único punto de vista. La única opción posible es que el cliente lo juzgue.

Comenzaremos el desarrollo de este apartado con las pruebas de caja negra para luego centrarnos en las pruebas de aceptación. Ambas son importantes debido a los aspectos que cubren cada una. Las pruebas de caja negra nos van a permitir medir la corrección de los métodos creados, o lo que es lo mismo, el grado de buen funcionamiento de las operaciones de la aplicación. Finalmente enumeraremos las herramientas empleadas para las pruebas de unidad ya que el juicio del cliente es suficiente para las pruebas de aceptación.

Caja negra:

Se realizaron pruebas para cada uno de los métodos de todos los DAO (23).

Se esperaba como resultado que se actualizase y/o recuperase información de la base de datos correctamente. Se cumplieron en todos los casos salvo en los métodos “Actualizar” de todos los DAO (4) que impedían que actualizaran objetos correctamente en la base de datos debido a las transacciones, así como en los métodos “buscaTodos” (4) por haber escrito una sentencia SQL incorrecta.

En la clase “Asociación” se realizaron pruebas para sus 24 métodos. Excepto en los métodos “get/set” (8) se encontraron problemas, esto era previsible ya que contiene la mayoría del cálculo de la aplicación, aunque la mayoría de los errores fueron condicionales equivocados. En los métodos “crea” se esperaba como resultado los objetos creados, estos se creaban salvo algún atributo mal inicializado que fue sencillo de solucionar. Los métodos “actualiza” se solucionaron cuando se corrigieron las clases DAO pues su error era el error de los propios DAO. Los métodos borra funcionaban correctamente pero no lanzaban las excepciones oportunas.

El método “sancionarSocio” sancionaba al socio correctamente pero gracias a las pruebas se descubrió que no se había creado el atributo correspondiente en la clase “Socio”.

Cabe destacar el método “pagar”. Se esperaba que crease una nueva cuota para los socios. El método en sí produjo bastantes problemas, tanto es así que se tuvieron que aplicar pruebas de caja blanca en él. Además este método se rediseñó varias veces, ya que el cambio de requisitos así lo especificaba.

En los controladores se produjeron problemas en los métodos “Crea” (3), pues creaban objetos con atributos nulos cuando no podían ser nulos, esto se arregló añadiendo restricciones a la base de datos (mediante anotaciones en las clases)

Los métodos “ver” de los controladores funcionaban correctamente pero mostraban información no deseada (como claves) que hubo que corregir.

En la parte del cliente extrañamente no se produjo ningún problema para las 20 pruebas realizadas a los métodos de los DAO, se esperaban los mismos valores que en los DAO del servidor y esos fueron los que se obtuvieron. Esto se debe, probablemente, a que las pruebas a los DAO del servidor se realizaron antes que la implementación de los DAO del cliente, y aunque no son comparables, la filosofía de estos es muy similar a la de los primeros.

Más complicado fueron los métodos de los controladores del cliente (33) salvo alguno puntual, todos tuvieron algún tipo de problema. Enumerar los valores esperados y obtenidos de cada uno sería largo y tedioso, así que simplemente diremos que se tuvo que realizar pruebas de caja blanca para solucionar la mayoría (y no fue para nada fácil ni rápido).

Para las pruebas de aceptación necesitamos del juicio del cliente, esto se ha conseguido en las diferentes entregas. En cada una de ellas se le proporcionaba a este el estado actual de la aplicación y se le observaba mientras trabajaba con esta, para comprobar que, efectivamente, consideraba que la aplicación (o las partes que se le entregaban) estaba correctamente desarrollada.

Las pruebas no detectaron más errores que los cambios en el requisito 9, Gestión de cuotas.

Herramientas utilizadas

Para las pruebas de caja negra son necesarias algunas herramientas. Para el cliente hemos contado con la ayuda de las pestañas consola, navegador y monitor de red del navegador. Para la interfaz del cliente, además, se ha contado con el plugin de Chrome NetBeans Connector. Para el servidor se ha utilizado el depurador de Netbeans así como el plugin de Firefox RESTClient para la interfaz REST del servidor.

6. Conclusiones

Para finalizar la presente memoria realizaremos una breve reflexión sobre el alcance de los objetivos inicialmente propuestos.

Se han analizado las necesidades de la asociación y estudiado diferentes soluciones existentes. Debido a que las soluciones no proporcionaban una solución total a las necesidades del cliente, se optó por proponer una solución que, finalmente, se ha acabado desarrollando.

Se ha realizado un estudio de tecnologías donde se ha visto que una arquitectura REST utilizando una tecnología JEE en el servidor con una lógica de control era una propuesta adecuada. Este estudio de tecnologías ha servido para desarrollar la solución propuesta.

Se ha realizado el diseño e implementación de una aplicación web. La aplicación se ha desarrollado empleando metodologías y tecnologías propias de la ingeniería del software que se han estudiado a lo largo de la carrera. Estas metodologías y tecnologías aplicadas han servido para realizar una aplicación de mayor calidad.

Se ha verificado que se ha desarrollado una solución adaptada a las necesidades reales del cliente y cumplía con sus expectativas. Para ello, como ya se ha indicado en el punto 5, se ha contado con el juicio del cliente. El éxito de una aplicación depende, en gran medida, de la aceptación del cliente, que ha sido muy buena.

No obstante, se ha visto que el alcance inicial del proyecto era inviable, pero gracias al uso de una metodología ágil se ha podido ajustar el desarrollo al centrarse en la parte más importante del sistema, dejando para un desarrollo futuro el resto de los requisitos que no se han cumplido.

Como una versión inicial operativa se ha desarrollado una aplicación web que permite al cliente la gestión de afiliados, cuotas, eventos e impagos, dejando para un futuro el desarrollo del resto de requisitos, como se explicará en el apartado 6.1.

El análisis de las necesidades de la asociación ha sido un aspecto clave en el desarrollo del presente tfg. Sin el esfuerzo dedicado a este análisis la solución podría no haberse ajustado a las expectativas del cliente.

En cuanto a las tecnologías seleccionadas, se puede afirmar que han sido acertadas. Spring siempre es una herramienta muy interesante, potente y versátil, sin embargo no es nada sencilla de usar y, aunque se contaba con experiencia previa (en dos asignaturas se ha estudiado en mayor o menor profundidad), ha requerido una cantidad de tiempo mayor de lo esperado, que ya de por sí era bastante importante. En Angular no se tenía tanta experiencia, por lo que también se ha dilatado el tiempo de aprendizaje que, también, era generoso, sin embargo el resultado es muy bueno y personalmente me ha

encantado trabajar con él una vez se supera la curva inicial de aprendizaje.

Emplear un desarrollo ágil ha sido también una decisión acertada. Una vez más se puede comprobar que es apropiado para proyectos pequeños y/o con una gran incertidumbre.

La planificación podría haber sido más acertada, aunque no ha estado mal teniendo en cuenta la escasa experiencia en algunas tareas. Se debe mencionar aquí, además, que muchas tareas han requerido mucho tiempo debido a esto mismo, la falta de experiencia.

En cuanto al proyecto en sí, ha sido muy interesante la toma de decisiones en cada punto, sopesando los pros y contras de cada opción. Creo que ha sido una gran experiencia, y muy útil, además. Poder contar con una asociación dispuesta a hacer el papel de cliente ha sido extraordinario también, sobre todo en etapas tempranas como diseño y planificación. Obviamente no ha sido tan útil como contar con un cliente, pero ha sido mucho más realista y desafiante que la falta de este.

Personalmente, el aspecto más relevante del proyecto ha sido el desarrollo de la aplicación servidor. Es muy versátil y capaz, además de ser independiente del cliente, lo que permite que, si la aplicación cliente necesita ser modificada o reemplazada, la información almacenada no se pierda, además de seguir permitiendo trabajar sobre la aplicación servidor, aunque con una interfaz más limitada y primitiva. Tal vez por ello ha sido el punto fuerte de la solución; el más robusto. Además mostraré mi agrado a la nomenclatura empleada en la bibliografía de la memoria diseñada, el formato IEEE para las citas bibliográficas que ha permitido ser más “profesional” a la hora de incorporar y referenciar la bibliografía utilizada en un documento formal.

Por último comentar que este proyecto de fin de grado me ha servido para confirmar la sospecha que tenía durante muchísimo tiempo: siempre hay nuevas tecnologías y conocimiento en el ámbito de la informática. Uno de los retos de un ingeniero informático es el evolucionar junto a esta, siendo capaz de aprender nuevas metodologías y tecnologías durante su carrera profesional, así como actualizar y adaptar las ya conocidas para mantenerse, de esta manera, competitivo.

6.1.- Mejoras y trabajos futuros

La idea de este punto es comentar las posibles mejoras que se podrían emprender en un futuro teniendo en cuenta las herramientas seleccionadas y la solución desarrollada..

La mejora más obvia de la aplicación es continuar desarrollando los diferentes requisitos e historias de usuario que no han sido implementados y han quedado fuera de

la solución inicial, que son el control de actas, novedades, pasarela de pagos, usuarios conectados y enlaces con redes sociales. El principal motivo por el cual no se han implementado era el tamaño real de los requisitos funcionales del proyecto, mucho mayor de lo que se podía asumir. Sin un tiempo de desarrollo de 300 horas y un único desarrollador se podría haber alcanzado incorporar un mayor número de requisitos. Sin tener esta limitación, incorporar estos requisitos a la solución no debería ser muy complicado con la de la arquitectura propuesta, las tecnologías y herramientas utilizadas en la solución.

Pero hay una mejora futura aún más importante; seguridad. La aplicación no ha contado con un alto nivel de seguridad debido a que se trataba de un trabajo de fin de grado; un prototipo de una aplicación web. Si se quiere continuar con el trabajo se deben implementar un mayor número de mecanismos de seguridad.

Estamos hablando de una aplicación web que puede almacenar algún dato sensible de algún posible menor. Según la LOPD[16] (Ley Orgánica de Protección de Datos) no nos encontramos ante una problemática menor, sino que hay que elegir una solución eficaz.

Para ello habría que implementar una mayor seguridad en las peticiones HTTP realizando una encriptación sobre ellas, para lo que se debería generar y firmar los correspondientes certificados digitales una vez decidido el nombre definitivo para el dominio de la aplicación. Para esto contamos con Spring Security que proporciona las herramientas necesarias para ello.

No nos podemos contentar sólo con ello, necesitamos proveer de una mayor seguridad. Para ello se propone la encriptación de los campos sensibles de la base de datos. De nuevo Spring Security tiene herramientas para ello.

16 https://goo.gl/iftSEy

7. Bibliografía

[1] Infojobs. *Puestos emergentes*. Mayo de 2016. [Online] Disponible en: https://goo.gl/hQ6tmG. Acceso: Agosto 2017.

[2] Troy Dimes. *Conceptos Básicos De Scrum: Desarrollo De Software Agile Y Manejo De Proyectos Agile*. Babelcube Inc. 17 de Marzo de 2015.

[3] Tiobe. *The top programming languages.* Agosto de 2017. [Online] Disponible en: https://www.tiobe.com/tiobe-index/. Acceso: Agosto de 2017.

[4] E. Jendrok et al. Junio de 2013 *The Java EE 7 Tutorial*. Oracle. [Online] Disponible en: https://docs.oracle.com/javaee/7/JEETT.pdf. Acceso: Agosto de 2017.

[5] Amuthan G. *Spring MVC: Beginner's Guide.* Packt Publishing. 25 de Junio de 2014.

[6] Universidad de Cantabria. *Especificaciones de los requisitos software*. [Online] Disponible en: https://goo.gl/BEPksq. Acceso: Agosto de 2017.

[7] PlanningPoker.com. *Planning Poker: The Best Agile Planning Tool.* [Online] Disponible en: https://www.planningpoker.com/about/. Acceso: Agosto de 2017

[8] Bankintercomite. *Nóminas del convenio TIC del año 2017 en España.* [Online] Disponible en: https://goo.gl/QvxvYZ. Acceso: Agosto de 2017.

9] Wikipedia. *Normalización de base de datos*. [Online] Disponible en: https://goo.gl/hPDR85. Acceso: Agosto de 2017.

[10] Mozilla. *Métodos de petición HTTP.* 30 de Mayo de 2017. [Online] Disponible en: https://developer.mozilla.org/es/docs/Web/HTTP/Methods. Acceso: Agosto de 2017.

[11] Juán Pavon Maestras. Universidad Complutense de Madrid. *El patrón MVC*. 8 de Septiembre de 2008. [Online] Disponible en: https://goo.gl/jSKDxV. Acceso: Agosto de 2017.

[12] Wikipedia. Pruebas de caja negra (sistemas). [Online] Disponible en: https://goo.gl/THvc9T. Acceso: Agosto de 2017.

Apéndice I: Manual de instalación del sistema

Para poder arrancar la aplicación suministrada necesitamos disponer de diferentes herramientas y plataformas para ello. Como ya hemos dicho con anterioridad, se ha proporcionado los archivos fuente del proyecto para lanzarlo en un IDE o, si somos más creativos y tenemos medios para ello, cargarlo en un servidor de aplicaciones web en la nube y arrancarlo desde ahí.

El proyecto se ha creado como una aplicación Java usando Maven. maven se encargará de descargar algunas de las dependencias mientras que otras ya estarán presentes. Enlazando con el tema de dependencias, es necesario que dispongamos de conexión a internet puesto que algunas de las bibliotecas empleadas se descargarán automáticamente cuando la aplicación se lance.

Para cargar la aplicación debemos disponer de un servidor de aplicaciones web. Si lo lanzamos desde un IDE como Netbeans, podremos elegir entre algunos como Glassfish o Tomcat. La aplicación ha sido probada en ambos, debería funcionar en cualquier otro aunque no se puede garantizar.

Una vez aclarados estos puntos lo único que tenemos que crear es una base de datos relacional. Una vez creada necesitamos crear una conexión con ella. Es importante recordar el usuario y clave que hemos utilizado en este último punto pues se debe instanciar un bean especial en el archivo applicationContext.xml para que Spring acceda a la conexión con las credenciales apropiadas:

```
<!--JDBC-->
 <bean id ="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
     <property name="driverClassName" value="org.apache.derby.jdbc.ClientDriver"></property>
     <property name="url" value="jdbc:derby://localhost:1527/aincradDB"></property>
     <property name="username" value="administrador"></property>
     <property name="password" value="administrador"></property>
 </bean>
```

Como podemos ver, necesitamos proporcionar un valor para los campos "*username*" y "*password*".

Además, en el mismo archivo debemos configurar también, en el bean "*JPAadapter*", dos campos en los cuales nos pide que base de datos usaremos. Si, por ejemplo, creamos una base de datos con Derby el bean debería verse así:

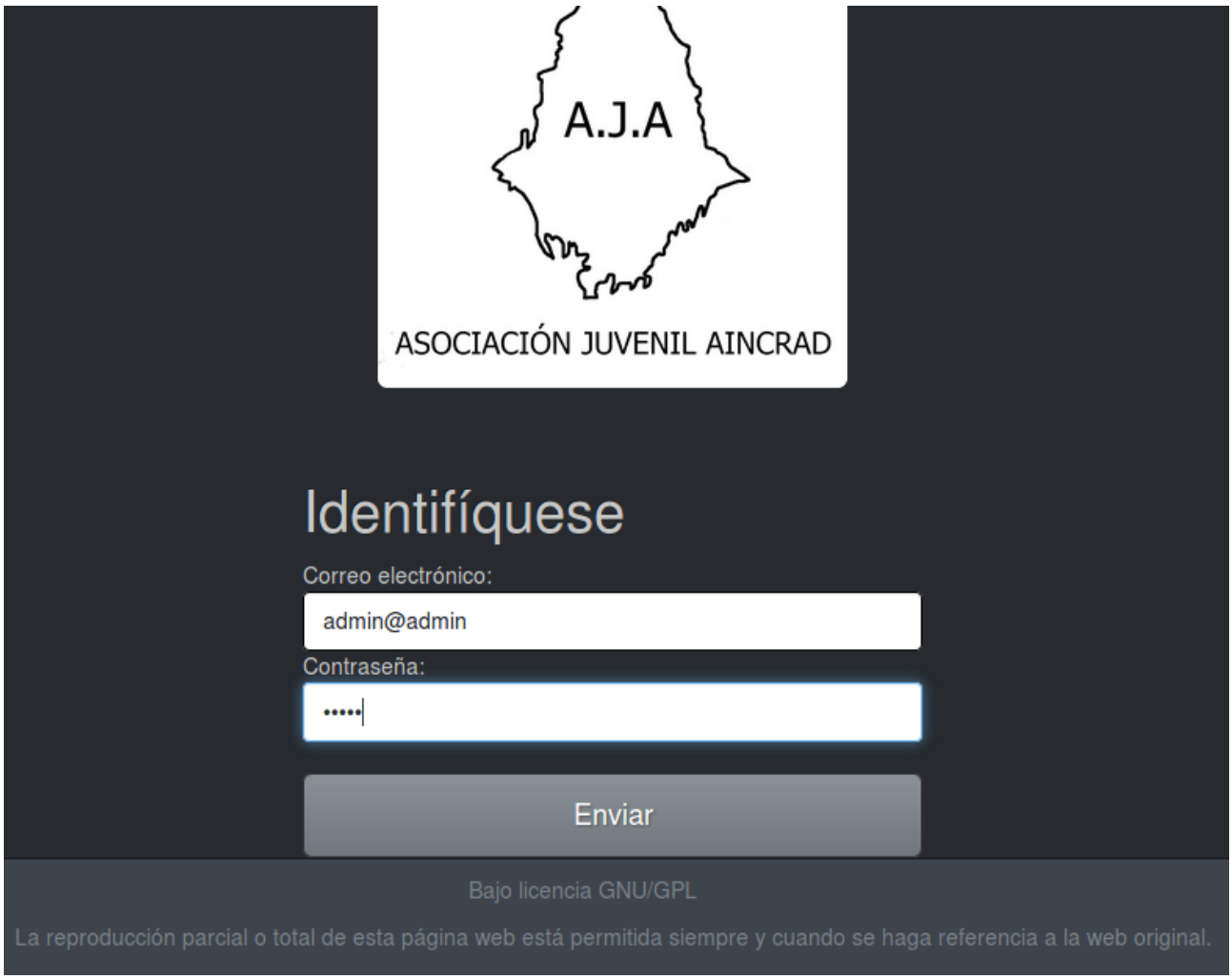
```
<bean id="jpaAdapter" class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
    <property name="database" value="DERBY" ></property>
    <property name="databasePlatform" value="org.hibernate.dialect.DerbyTenSevenDialect"></property>
```

Donde debemos dar un valor para el campo "*database*" y otro para "*databasePlatform*".

Apéndice II: Manual de Usuario

Gracias al segundo requisito no funcional, NF2 sencillez de uso, se ha creado una interfaz sencilla que no necesite de un entrenamiento previo. Aun así se comentará algunos de los puntos que pueden resultar más complejos, así como algunas características de la aplicación.

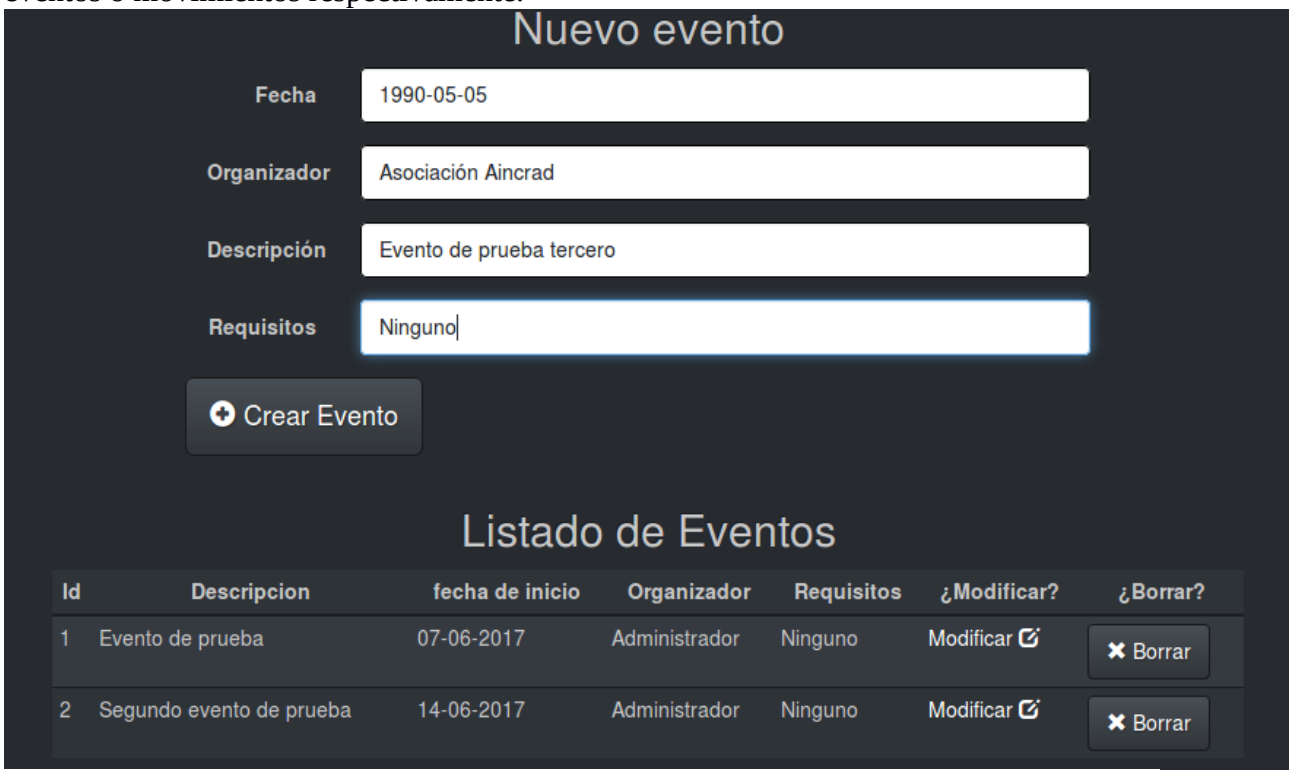
Al acceder a la aplicación debemos identificarnos:



Dependiendo de si somos un socio o un administrador tendremos unas opciones u otras, si somo un administrador la barra de navegación debería verse así:

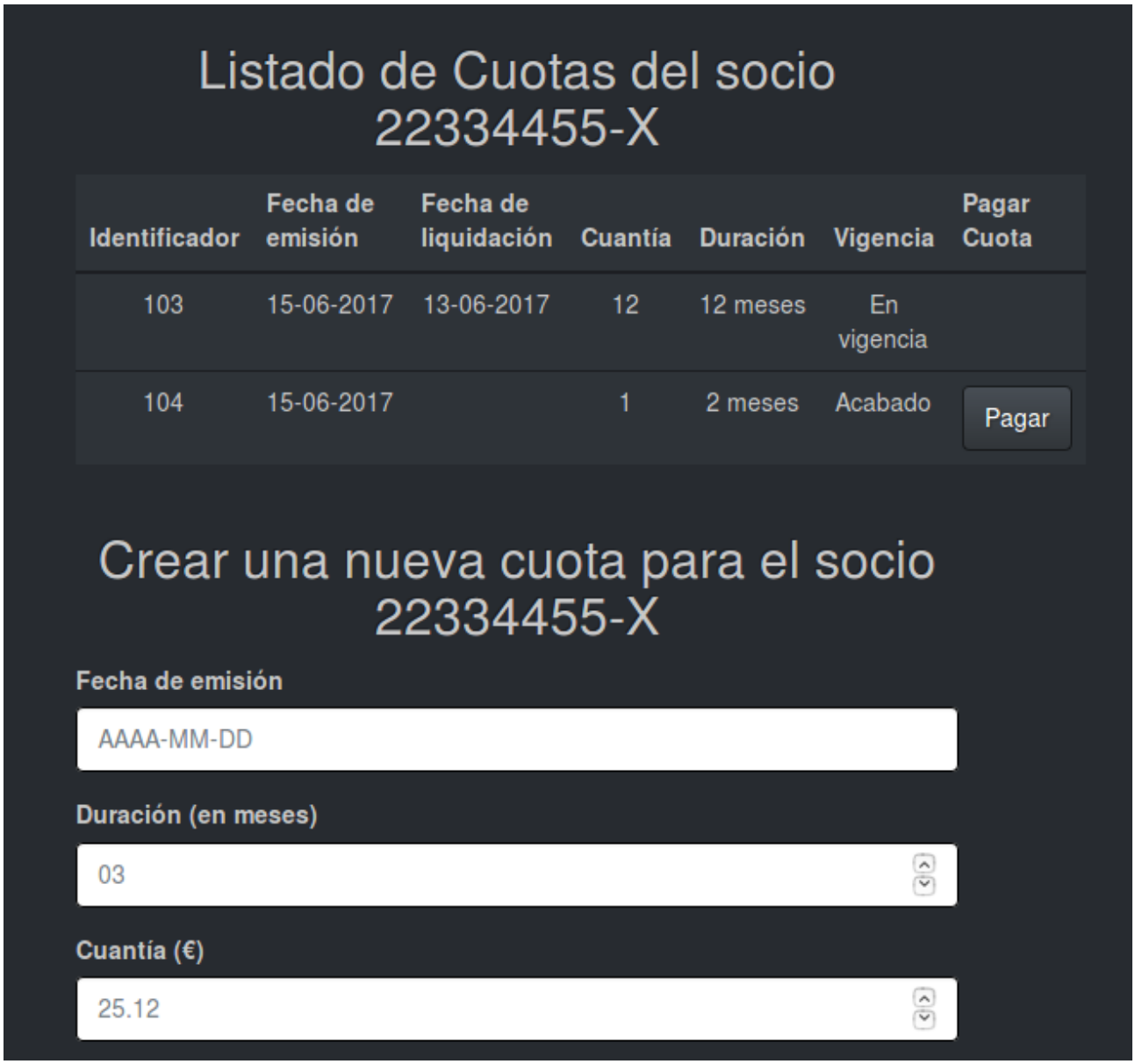
Las opciones son bastante claras, pues se refieren a cada una de las acciones que podemos realizar (que a su vez están relacionadas con los requisitos especificados)

En las pestañas eventos y movimientos podremos ver, modificar, borrar y crear eventos o movimientos respectivamente.



Gestión de socios funciona de una manera similar, con la excepción de que ahora también podremos sancionar o eliminar la sanción de uno de los socios.

Los socios, por seguridad, no pueden eliminarse directamente desde la aplicación, debe hacerse sobre la base de datos. Esto se ha hecho así porque nunca se eliminarán los socios para tener los datos históricos de ellos. Si hacemos clic en gestionar podremos crear una nueva cuota para el socio y gestionar alguna información extra suya:



Perfil nos permite visualizar los datos personales mientras que listado de cuotas nos permite ver todas las cuotas existentes.

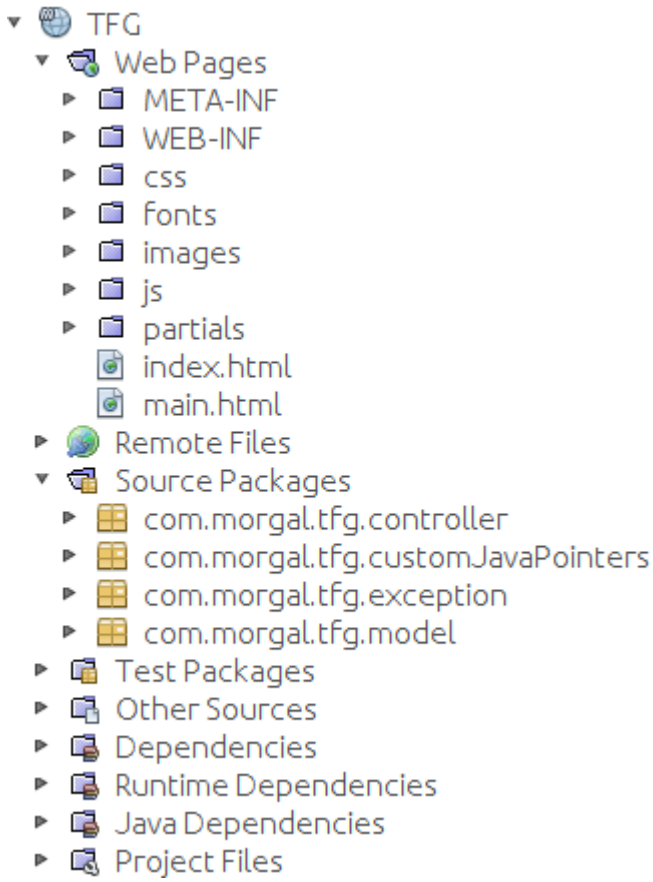
Apéndice III: Descripción de contenidos suministrados

En este trabajo de fin de grado se han suministrado los siguientes elementos:

- Memoria (en un documento pdf)
- Código fuente (En un archivo zip, donde hay una jerarquía de carpetas) y software complementario
- Documentación de la aplicación (En formato javadoc)

La memoria es el presente documento, el cual no necesita más explicación.

El código fuente (dentro del archivo zip) está organizado de la siguiente manera:



Dentro de Web Pages:

En META-INF estará el archivo de contexto de la base de datos y del driver necesario.

En WEB-INF los diferentes archivos de configuración de Spring y Spring Security.

En css los diferentes archivos css que necesita Bootstrap.

En fonts algunos *Glyphicons* necesarios para Bootstrap.

En images hay imágenes necesarias para la página web.

En js los diferentes archivos javascript creados, así como algunos necesarios ya descargados.

En partials los diferentes fragmentos html.

En Remote Files las bibliotecas que se decargarán de internet (software complementario).

En Source Packages:

En com.morgal.tfg.controller los controladores del servidor.

En com.morgal.tfg.customJavaPointers un manejador de éxito en la autentificación de Spring Security personalizado.

En com.morgal.tfg.exception las excepciones personalizadas creadas.

En com.morgal.tfg.model los archivos del modelo (las clases java)

Las tres carpetas de dependencias siguientes contienen diferentes tipos de dependencias ordenadas según el momento en el que son necesarias.

En Project Files el archivo de configuración del proyecto y el de configuración de Maven.

La documentación se ha realizado usando el formato javadoc y está formada por una serie de documentos html.