Proyecto fin de master Administración web y comercio electrónico

Desarrollo de librería Django para desarrollo de RIAs e implementación de zona privada de portal web sminn.com

Sergio Blanco Diez

Francisco Javier Noguera

Bilbao, 6 de Junio del 2011

Reconocimiento – Compartir bajo la misma licencia

Usted es libre de:

Compartir – copiar, distribuir y comunicar públicamente la obra

Remezclar – transformar la obra

Uso comercial – hacer un uso comercial de esta obra

Bajo las condiciones siguientes:

Reconocimiento – Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).

Compartir bajo la misma licencia – Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

Entendiendo que:

Renuncia – Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor.

Dominio público – Cuando la obra o alguno de sus elementos se halle en el dominio público según la ley vigente aplicable, esta situación no quedará afectada por la licencia.

Otros derechos – Los derechos siguientes no quedan afectados por la licencia de ninguna manera:

- Los derechos derivados de usos legítimos u otras limitaciones reconocidas por ley no se ven afectados por lo anterior.
- Los derechos morales del autor.
- Derechos que pueden ostentar otras personas sobre la propia obra o su uso, como por ejemplo derechos de imagen o de privacidad.

Aviso – Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.

Resumen

El presente proyecto consiste en dos partes bien diferencias: terminar la funcionalidad prevista para el portal web sminn.com en forma de su sección privada para profesionales y experimentar la evolución de dicho portal a una aplicación web compleja tipo “single page RIA” implementando una librería a liberar que ayude al desarrollo de dicho tipo de aplicaciones. De esta forma la primera parte aportará conocimiento de diseño y desarrollo para diferentes aspectos de un portal web estándar sobre Django y la segunda parte aportará un recurso que evolucionará junto a la comunidad de software libre. Dentro del experimento de uso de la librería para una siguiente versión del portal web sminn.com se propone la realización de pruebas de interfaz adaptable.

Descriptores

RIA, Web, Django, Python, Middleware, Integración, Javascript

Índice general

1. INTRODUCCIÓN

El presente documento ha sido redactado por un alumno del Máster en Software Libre de la *UOC* para poner a disposición del director de proyecto, del tribunal de defensas de las asignaturas correspondientes al *Proyecto de Fin de Máster* y de cualquier lector que pudiera tener este documento en sus manos la información más relevante concerniente al desarrollo del proyecto que se presenta a continuación

Los capítulos de los que consta este documento son los siguientes:

1. **Introducción.** El capítulo de introducción contiene una descripción general del contenido del presente documento y la introducción al concepto de “Single page RIA”, poniendo así al lector al corriente de la motivación de fondo del proyecto.
2. **Objetivos del proyecto.** El capítulo de objetivos contiene básicamente los diferentes aspectos de la planificación del proyecto como son sus objetivos, alcance, tareas a desarrollar y planificación temporal.
3. **Análisis.** El capítulo de análisis contiene la descripción de los requisitos para el desarrollo del sistema obtenidos durante la fase de análisis y una serie de consideraciones en cuanto a interfaces de usuario.
4. **Tecnologías.** El capítulo de tecnologías contiene una descripción de las tecnologías investigadas y utilizadas en la realización del proyecto como son Django y varios de sus módulos y JQuery, comentando en cada caso lo más relevante al respecto de las tecnologías en cuestión y su uso.
5. **Especificación del diseño.** El capítulo de especificación del diseño contiene la descripción del diseño del proyecto detallando la arquitectura del sistema.
6. **Desarrollo del proyecto.** El capítulo de desarrollo del proyecto contiene la descripción de una serie de puntos claves del proceso de implementación del proyecto, haciendo hincapié en la propia implementación. La información de este capítulo se ve complementada por el manual del programador anexo que proporciona otro punto de vista a la implementación de la librería desarrollada.
7. **Conclusiones.** El capítulo de conclusiones contiene las conclusiones a nivel personal del único miembro del equipo de trabajo y redactor de la presente memoria, comentando los aspectos más relevantes en lo que se refiere al proyecto.
8. **Manual del desarrollador.** El manual del desarrollador es un anexo que representa el manual que va a acompañar a la librería a liberar y supone una introducción rápida y sencilla al uso de dicha librería.

Se ha tratado que la presente memoria cubra los aspectos más relevantes y en general, menos evidentes, tratando de mantener un estilo conciso y claro. En general su contenido ha sido concebido para la lectura por parte de lectores con conocimientos técnicos avanzados en materia de desarrollo web. Aún con todo, el contenido está muy esquematizado y resulta muy accesible tanto para leer de principio a fin como para consultar datos concretos. Las figuras han sido colocadas buscando la mínima ruptura del flujo de texto, lo cual ha sido facilitado por la tecnología de TEX.

En cuanto al propio proyecto, éste es fruto del duro trabajo de varios meses y de un claro objetivo: facilitar el desarrollo de *Rich Internet Application*[39] sobre el framework web *Django*[9]. Para ello el presente proyecto busca el desarrollo de una librería a liberar a la comunidad que cumpla este objetivo y la demostración de la misma sobre el portal web de la empresa que financia y participa en

el proyecto. Como punto adicional al proyecto y para asegurar un mínimo de resultados sin riesgo, se desarrollará también la zona privada de profesionales del portal, incluyendo el puente entre el ERP interno de la empresa y la parte de comercio electrónico de su portal.

Tras esta breve introducción al presente documento y al proyecto, se ofrece una introducción al concepto central del proyecto: *Single page RIA*.

1.1 Single page RIA

La tecnología web ha cambiado mucho desde su concepción inicial; en gran parte la idea base del acceso a contenido estructurado se mantiene pero la forma de estructurar dicho contenido, de presentarlo, e incluso de ofrecer capas de interacción complejas sobre el mismo ha ido acercando lo que se diseñó como una enorme enciclopedia distribuida a un entorno de aplicaciones accesibles desde un gran número de equipos informáticos siempre y cuando éstos soporten algunos estándares. Esta transición de contenido puro a aplicación ha ido pasando por varias fases hasta llegar a la actualidad, existiendo ahora dos hechos; el estado del arte consiste en complejas aplicaciones web de página única denominadas *single page RIAs* que guardan el máximo parecido posible con las correspondientes aplicaciones de PC (como se verá a continuación) y el portal web medio, que utiliza algunos medios de dinamismo para visualizar su contenido y adornar la presentación. Esto se ve propiciado porque hay muchos portales que por objetivos son ricos en contenido pero no tienen muchas posibilidades de interacción y ocurre también muchas veces por la complejidad de la creación de aplicaciones web complejas.

¿Qué caracteriza entonces a una single page RIA? Su aspecto más visible, desde el punto de vista del paradigma web, es que en todo el flujo de navegación el usuario descarga un único fichero HTML[19] que la mayor parte de las veces no incluye ningún tipo de contenido. Dicho fichero incluye los suficientes programas Javascript[22] como para poner en macha la capa de presentación y hacer peticiones asíncronas al servidor para recuperar datos. Este aspecto, aún pareciendo muy elemental en su presentación, supone un gran cambio de paradigma en la web; supone una transición desde un cliente ligero, que recibe contenido estructurado con unas reglas de estilo bien definidas y tal vez algunos puntos de dinamismo, a un cliente pesado que se autogestiona y recibe tan solo los datos que necesita, sin que estén necesariamente estructurados o preparados para la presentación. Dicho de otra forma, el servidor se convierte en un motor de arranque y en un repositorio de datos y se pasa gran parte del peso al navegador.

Entrando en las similitudes con las aplicaciones de escritorio, las single page RIA suelen soportar varios mecanismos de interacción como la gestión avanzada de la entrada de usuario; atajos de teclas, uso de la rueda del ratón y de combinaciones de ratón y teclado, . . . Se produce un cambio de un modelo de interacción simple y reglado a un modelo más libre en el que el desarrollador de la aplicación puede definir como tratar la entrada de usuario. La salida también gana en complejidad, especialmente con las últimas versiones de HTML[19], CSS[7]y Javascript[22]; generación dinámica de fotogramas mediante dibujado en dos y tres dimensiones en un objeto tipo lienzo, reproducción de multimedia de forma estándar, transiciones controlables de estilo. . .

Aunque existen muchas nuevas posibilidades asociadas a estas tecnologías, no siempre son necesarias o útiles todas ellas; sigue habiendo portales web dedicados totalmente a la presentación de contenidos estructurados y es posible que solo sean útiles para estos algunos puntos de estas tecnologías. Un aspecto poco explotado pero muy demandado es la adaptación a la pantalla; cada usuario tiene una pantalla diferente en ratio de aspecto, resolución,. . . y no solo eso, cada usuario utiliza el navegador de diferente forma; aunque lo más usual es utilizarlo maximizado o a pantalla completa, a veces es habitual dimensionar la ventana al gusto propio. La web tradicional generalmente impone unas medidas mínimas y adapta el contenido mediante uso de barras de desplazamiento horizontales y verticales. Una opción propia de las nuevas tecnologías es la adaptación dinámica del contenido a la pantalla para aprovecharla al máximo, como puede verse por ejemplo en la aplicación web de gran prestigio *Google Mail* (la cual es una single page RIA a todos los efectos):

Figura 1.1.: Adaptación horizontal en Google Mail

Como puede verse, Google ha decidido adaptar su contenido horizontalmente pero dejar el desplazamiento vertical ya que tiene sentido para su aplicación. En cualquier caso, es posible adaptar los contenidos en ambos ejes.

2. PLANIFICACIÓN

En el presente capítulo se establece el punto de partida del proyecto considerando los aspectos de planificación relevantes para la consecución del mismo, cubriendo para esto los siguientes epígrafes:

- **Definición del proyecto:** Descripción de los objetivos y alcance del proyecto.
- **Producto final:** Descripción de los productos intermedios y finales a desarrollar. Incluye un análisis de alternativas y describe el software a utilizar y el modelo de licenciamiento.
- **Metodología de desarrollo:** Descripción del método de desarrollo.
- **Organización y equipo:** Descripción del esquema organizativo del proyecto, del método de seguimiento y de los perfiles implicados en el proyecto.
- **Planificación:** Resumen de tareas y plan de trabajo.
- **Análisis de riesgos:** Identificación de los riesgos en la consecución del proyecto y consideraciones con respecto a su disminución y control.
- **Presupuesto:** Presupuesto económico asociado al proyecto.

2.1 Definición del proyecto

El proyecto a presentar al final del segundo semestre del año 2011 como *Proyecto de Fin de Máster* para el *Máster en Software Libre* de la *UOC* consiste en la segunda versión de un portal web desarrollado con Django[9] publico, *sminn.com*, de una empresa dedicada a la venta de producto de electrónica propio, *Elson Sistemas S.L*. Esta segunda versión incluye principalmente dos puntos de novedad sobre la versión anterior:

- **Desarrollo y puesta en marcha de la zona profesional de portal:** Los clientes podrán crearse un perfil de usuario para acceder a una zona privada desde la que tendrán acceso a información más específica de los productos y podrán gestionar sus transacciones con la empresa (peticiones de oferta, ofertas y pedidos).
- **Redefinición de toda la capa de presentación en forma de *Single Page RIA[39]*:** El principal target de la página son usuarios poco adeptos a las tecnologías web y que esperan una experiencia más similar a la de una aplicación de escritorio. El jefe de proyecto ha propuesto hacer la lógica de presentación más compleja para que la interfaz se adapte automáticamente a la pantalla y las transiciones sean más suaves.

2.1.1 Objetivos

Este proyecto tiene varios objetivos primarios:

- Desarrollo e implantación de la zona profesional del portal sminn.com.
- Realizar la transición a un paradigma web superior transformando el portal en una Single Page RIA con una capa de presentación más adecuada para su usuario medio.
- Liberación a la comunidad de unos resultados de valor para otros proyectos web.

Para ello, se consideran los siguientes objetivos generales:

- Desarrollo y liberación de una librería para el framework web Django[9] que facilite el desarrollo de RIAs complejas. Esta librería proporcionará medios para la definición de comportamientos desde el lado servidor que serán activados dinámicamente desde el cliente web, siempre utilizando tecnologías estándar (HTML[19], CSS[7], Javascript[22]) a poder ser en sus últimas versiones. Entre las facilidades que incluirá está librería, cabe destacar la carga y descarga dinámica y optimizada de módulos javascript, la definición desde la aplicación servidor del procedimiento de carga de cada *página*, sistema de componentes extensible, . . .
- Desarrollo de un puente entre el sistema de gestión de Elson Sistemas S.L. y el servidor de la aplicación web para sincronización de las transacciones de los clientes.
- Desarrollo de la sección de profesionales del portal.
- Conversión del portal al nuevo sistema que utilizará la librería antes mencionada. Al menos se realizará una prueba de concepto ya que el diseño recae sobre otra empresa, Dr. Minsky.
- Implantación y mantenimiento de esta nueva versión del portal sobre un VPS contratado por la empresa.
- Optimización del procesamiento en el lado servidor por medio de mecanismos de cache.

2.1.2 Alcance del proyecto

El proyecto debe incluir como mínimo:

- Desarrollo y liberación de la librería en una forja para la creación de RIAs en Django.
- Desarrollo de la sección de profesionales:
  - Integración con el sistema de gestión de la empresa Elson Sistemas S.L.
  - Desarrollo del backend para todos los contenidos propios de esta sección.
- Al menos, una prueba de concepto de interfaz de usuario sobre el nuevo sistema para demostrar a la empresa de diseño e imagen Dr. Minsky la línea de estilo a seguir en la conversión de todas las secciones ya existentes.

Y por ende, el proyecto *no* incluirá:

- Desarrollo de estilos CSS finales para toda la aplicación web, ya que se encargará Dr. Minsky de realizar estos y su planificación, que aunque paralela, se extiende más allá de este proyecto.
- Población de la base de datos con toda la información necesaria: Es posible que gran parte de esta tarea esté hecha ya que es paralela al desarrollo del portal, sin embargo, la totalidad de su realización no es responsabilidad del alumno.

Dicho de otra manera, el alumno se responsabiliza en el proyecto de realizar todo el desarrollo del backend servidor, de la librería conjunta en la parte servidor y cliente y de tener la aplicación web funcionando al completo sobre esta premisa. El estilado final y los datos de base de datos, aunque seguramente estarán en gran parte y serán demostrados y comentados, no entran dentro del alcance per sé del proyecto.

2.2 Producto final

Existirán varios productos a lo largo de la vida del proyecto. A continuación se detallan los mismos:

- **Informe de Requisitos:** Documento que contendrá de forma ordenada los requisitos obtenidos durante la fase de definición de requisitos en base al estudio de las necesidades de la empresa Elson Sistemas S.L. y del perfil de sus clientes dentro del alcance definido.

- **Informe de Diseño:** Documento que especificará en su propio lenguaje la transición de los requisitos a un modelo de software que permita cumplir los objetivos del proyecto, haciendo especial hincapié en la librería a liberar.
- **Programas:** Existirán por así decirlo dos sistemas a desarrollar; el lado servidor con su parte de la librería y el lado cliente con su parte de la librería correspondiente. Esta librería será publicada bajo una licencia tipo BSD para permitir el máximo uso de la misma.

2.2.1 Elección de soluciones software

A la hora de desarrollar el proyecto es importante fijarse en su división en dos partes bien diferenciadas; por un lado está el lado servidor y por otro el lado cliente. El lado servidor es el más rico en cuanto a elecciones; hay que elegir uno o varios servidores web especializados (una configuración usual es utilizar un servidor ligero para el contenido estático y otro más pesado para el contenido dinámico), un motor de base de datos, un lenguaje para el procesado de las peticiones dinámicas, una vez escogido el lenguaje muy posiblemente un framework que facilite el desarrollo, librerías asociadas al framework . . . El servidor cuenta, por tanto, con el equivalente a una *stack LAMP*[25]. El cliente, en cambio, está limitado a un conjunto de soluciones estándar (HTML, CSS y Javascript) quedando la elección en las versiones de algunas de estas soluciones y en librerías asociadas a ellas.

■ Soluciones software en el servidor

Lenguaje de programación Es importante considerar, en primer lugar, el lenguaje sobre el que se va a trabajar ya que esta decisión puede limitar al resto de consideraciones de desarrollo; soporte para motores de bases de datos, disponibilidad de módulos para el servidor web que permitan trabajar con dicho lenguaje. . .

A día de hoy se trabaja habitualmente con cuatro lenguajes en el lado servidor:

- Java[21]: Mayor rendimiento pero en general mayor complejidad; las librerías disponibles tienen un enfoque claramente *enterprise* y añaden capas de complejidad que salvo para aplicaciones muy concretas hacen el desarrollo más complicado.
- PHP[32]: El lenguaje servidor por excelencia; es el más utilizado, ahora bien, está demasiado pensado para ser embebido en el servidor web y ser utilizado como un procesador de plantillas. Su rendimiento es medio y tiene el que más librerías y soporte.
- Ruby[41]: Junto a Python, un lenguaje cada vez más usado gracias a frameworks que simplifican el desarrollo. Su rendimiento es en general menor que el de PHP, la comunidad es menor y existen menos librerías y utilidades (aunque más que suficientes).
- Python[36]: Lenguaje de script con mejor rendimiento general que PHP. Tiene una comunidad menor (igualmente grande) pero experta. Aunque la disponibilidad de librerías y módulos es ligeramente menor los módulos Python tienen un diseño excelente y una facilidad de uso sin precedentes.

Hay que aclarar que todo es posible con cualquiera de los cuatro lenguajes, y salvo las ventajas que pueda aportar Java para el desarrollo de aplicaciones *enterprise*, en general todos tienen los medios para el desarrollo de portales web. El rendimiento, salvo en casos particulares, no es una clave demasiado importante ya que el cuello de botella suele estar en la base de datos y en la conexión al servidor per sé. Por lo tanto, al menos desde el punto de vista del alumno que suscribe esto, el parámetro de elección más importante es la experiencia con el lenguaje y el segundo parámetro, la facilidad del mismo. Por esta razón, cuando comenzó el proyecto del portal web *sminn.com* la elección recayó sobre Python; no solo su rendimiento es excelente sino que tiene prácticamente todo lo necesario ya incluido en sus librerías base, se puede usar en conjunto con un importante número de librerías de calidad ya existentes, está respaldado por una gran comunidad de desarrolladores y es el lenguaje indicado para el desarrollo ágil de todo tipo de aplicaciones. La licencia de Python es una licencia propia open-source no viral.

Framework web Python cuenta con varios frameworks para el desarrollo de aplicaciones y portales web. Cabe destacar entre ellos los siguientes tres:

- Django[9]: Django es un framework web que está tomando un gran renombre en la comunidad; su diseño exquisito y minimalista, gran versatilidad (aunque su diseño original está orientado a sitios web basados en la presentación de contenido) y excelente documentación hacen que trabajar con este framework sea un verdadero placer.
- Grok[17]: Grok es un enorme framework web heredero de Zope[50]. Se considera un framework de mayor potencia que el resto, pero también más complejo y con un diseño más intrincado.
- Pylons[35]: Pylons es un framework web que se considera intermedio en cuanto a complejidad entre Grok y Django; se basa en una mayor personalización del workflow, pensando más en todas las particularidades de los proyectos que en las partes comunes.

Una vez más es importante la experiencia y la decisión termina siendo subjetiva. Los tres frameworks son muy válidos y aunque pueden ofrecer ventajas para proyectos específicos, son lo suficientemente completos y ágiles como para adaptarse a las necesidades del proyecto. Desde el punto de vista del alumno que suscribe esto, la experiencia indica que es mejor trabajar con mecanismos lo más simples y básicos posibles, que faciliten el trabajo pero no oculten mecanismos demasiado complejos tras de sí. Dicho esto, la elección recayó sobre Django. La licencia de Django es BSD, open-source no viral.

Utilidades para el framework Django cuenta con varios módulos de terceros interesantes para el desarrollo de portales web. A continuación se presentan algunos utilizados para el desarrollo del proyecto:

- South[42]: South es un sistema de migración de esquema de datos; monitoriza los modelos de datos y facilita el cambio del esquema de la base de datos sin pérdida de datos. Licencia Apache, open-source no viral.
- Rosetta[40]: Rosetta es una extensión para permitir la internacionalización del portal web desde una página interna del propio portal. Licencia MIT, open-source no viral
- Debug toolbar[8]: La Django Debug Toolbar es una librería que introduce en todas las páginas una barra con información de debug para facilitar la traza y el análisis de tiempos. Licencia BSD, open-sorce no viral.

Servidores web Python cuenta con un tipo de interfaz para permitir que un servidor web interactúe con él, llamado WSGI[47], de forma que cualquier servidor web que soporte WSGI puede ser utilizado. Existen en general dos opciones en cuanto a servidores web: servidor web único y servidor web con servidor web proxy. La opción de usar un servidor web en conjunto con otro servidor web proxy surge especialmente para ahorrar memoria. Cuando utilizamos un servidor web que consume mucha memoria, podemos quitarle parte de las peticiones usando otro servidor web ligero a modo de filtro. El servidor que actúa como filtro recibe todas las peticiones y le pasa al principal sólo aquellas que mejor puede atender. A continuación se comentan algunas opciones:

- Apache[4]: Apache es el servidor web más utilizado sin lugar a dudas, teniendo su momento de mayor acogida durante el boom de PHP hace algunos años. Cuenta con todos los módulos que puedan necesitarse y soporta todas las tecnologías web en uso, pero desde hace algún tiempo la comunidad lo considera pesado en cuanto a consumo de memoria, sobre todo en comparación con otros servidores web más ligeros (que también cuentan con menos posibilidades). Licencia Apache, open-source no viral.
- Nginx[31]: Nginx es un servidor web que está tomando una gran inercia en los últimos años. Se trata de un servidor web muy completo y con un rendimiento excepcional, en general superior al rendimiento de Apache. Nginx acaba de llegar a su versión 1.0, y aunque se considera muy estable, no tiene tanto uso detrás como Apache. Licencia tipo BSD, open-source no viral.

- Lighttpd[26]: Este servidor web tiene muchas menos capacidades que el resto, pero es también el más rápido y el que menos recursos consume. Se suele utilizar en sitios con un gran número de peticiones para exprimir al máximo el hardware. Licencia BSD, open-source no viral.
- Nginx + Apache: Esta configuración se basa en utilizar Nginx para servir medios estáticos y Apache para la generación de contenido dinámico. De esta forma, el consumo de memoria para servir recursos (en la mayoría de portales, cada petición dinámica puede verse acompañada de decenas de peticiones estáticas) se minimiza.
- Nginx + Gunicorn[18] o uWSGI[45]: Esta configuración es similar a la anterior, pero utiliza un servidor ligero específico para servir aplicaciones web por WSGI en lugar de Apache, reduciendo mucho el consumo de recursos. Se considera a uWSGI como la opción más rápida[5] aunque Gunicorn ofrece la instalación más sencilla y rápida al estar escrito en Python. Gunicorn tiene licencia tipo BSD, open-source no viral y uWSGI tiene licencia GPL2, open-source viral.

Para el presente proyecto se ha escogido utilizar Apache ya que se cuenta con un servidor dedicado con 1 Gigabyte de memoria RAM, por lo que el consumo de memoria no es prioritario y salvo ese aspecto ofrece un muy buen rendimiento, consta de una gran cantidad de módulos muy probados y tiene, en general, mucho tiempo de utilización, con lo que está garantizada su estabilidad.

Motor de base de datos Python (y por ende Django) soporta por defecto los motores de bases de datos más utilizados:

- MySQL[30]: Junto a PostgreSQL, uno de los motores de bases de datos más utilizados. Tuvo su momento de mayor acogida durante el boom de PHP ya que formaba parte de la *stack LAMP*. Históricamente se consideraba a MySQL como una opción más rápida pero con menos características que PostgreSQL, lo cual hace tiempo que dejó de ser notable ya que ambos han ido mejorando en sus aspectos débiles. Licencia GPL, open-source viral.
- PostgreSQL[33]: PostgreSQL es un motor de base de datos más complejo que MySQL, considerado como la alternativa libre a sistemas de bases de datos propietarios complejos como Oracle. Está teniendo ahora un momento de gran acogida dada la situación de compra de Sun por parte de Oracle y por tanto la incertidumbre que rodea a MySQL. Licencia tipo BSD, open-source no viral.
- SQLite[43]: SQLite no es per sé un motor de base de datos, sino una librería que permite crear bases de datos relacionales sencillas sobre ficheros, contando con la mayor parte de características que se utilizan en bases de datos y con un rendimiento muy alto. Es muy útil para sistemas embebidos, pruebas o simplemente, cuando no merece la pena instalar y mantener un servidor de base de datos pero se desea trabajar con los datos en forma de base de datos. Licencia de dominio público, no viral.

Para el presente proyecto se han escogido MySQL para la versión de despliegue y SQLite para la versión de desarrollo como motores de base de datos básicamente por tener más experiencia con los mismos. Se valoró abandonar MySQL y migrar a PostgreSQL por la compra de Sun por parte de Oracle y el futuro incierto de MySQL dentro del marco del software libre, pero finalmente se optó por MySQL igualmente dada la existencia de forks del mismo como MariaDB[27], que es compatible al 100 % con MySQL.

Sistema de caché El sistema de caché no es un componente obligatorio ni mucho menos, pero tal y como se ha demostrado en portales de gran actividad y rendimiento como Facebook[14], una forma barata de obtener escalabilidad y rendimiento es utilizar caché en memoria RAM para evitar repetir procedimientos costosos constantemente. La memoria RAM es más barata que la optimización pormenorizada del código servidor y eligiendo bien qué introducir en la caché y durante cuanto tiempo, se puede aumentar el rendimiento sensiblemente. Actualmente en uso existen dos sistemas de caché bastante diferentes:

- Memcached[29]: Memcached es un servidor que se ha hecho muy famoso por ser utilizado en portales como Flickr, Twitter, Youtube, . . . Se trata de un servidor muy sencillo al que se le pueden

volcar datos asociados a una clave de texto y un tiempo de vida determinado. Durante dicho tiempo de vida el servidor guardará en memoria RAM esos datos y pueden ser recuperados a velocidades muy altas. Licencia BSD, open-source no viral.

- Redis[37]: Redis se considera como un Memcached con soporte para reflejar los datos en disco, más tipos de datos, replicación, . . . Redis tiene como ventajas todas esas capacidades añadidas, pero el rendimiento de redis es inferior al de memcached[28]. Licencia BSD, open-source no viral.

Para el presente proyecto se ha elegido Memcached por su simplicidad de configuración y de uso y su perfecta integración con Django.

■ **Soluciones software en el cliente**

HTML No hay mucho que elegir en esta cuestión salvo utilizar en la medida de lo posible la última versión del estándar HTML, HTML5. La especificación de esta versión aún no ha finalizado y no todos los navegadores soportan todas las características publicadas hasta ahora, por lo que se utilizarán tantos aspectos del mismo como tenga sentido para garantizar que el portal sea funcional. La librería a desarrollar no será dependiente de HTML5 pero si lo soportará y dará facilidades para adaptarse al navegador que realiza la petición.

CSS Al igual que ocurre con HTML, se utilizará en la medida de lo posible CSS3, aún no publicado, para aprovechar sus capacidades en cuanto a transiciones, efectos, gradientes y bordes redondeados.

Javascript Javascript queda atado a los elementos utilizados en los dos estándares anteriores.

Librerías Javascript Este aspecto es uno de los más importantes en el cliente; desde hace ya varios años existen varios frameworks Javascript que facilitan el desarrollo de aplicaciones web como JQuery[23], Prototype[34], YUI[49], . . .

La librería Javascript a desarrollar será independiente de estos frameworks y soportará que se use cualquiera de ellos. La aplicación de dicha librería al portal de Sminn se realizará con JQuery; su mayor comunidad, cantidad de recursos disponibles, diseño y la experiencia previa del alumno en dicha tecnología lo hacen ideal para el trabajo. JQuery tiene licencia MIT o GPL2, por lo tanto open-source, elección de viral o no viral.

2.3 Metodología de desarrollo

2.3.1 Control de versiones

Se utilizará un sistema de control de versiones para la gestión de cambios y soportar el trabajo en equipo una vez publicada la librería. Se ha optado por la utilización de git[16], un sistema de control de versiones distribuido por su flexibilidad y su modelo de ramificación, que permite una paralelización del trabajo excelente.

2.3.2 Plan de pruebas

Para las pruebas se utilizarán dos enfoques; por un lado se dividirá la web en dos versiones, una de desarrollo y otra de despliegue. Se trabajará sobre la de desarrollo y se irán integrando características en la versión de despliegue poco a poco según son probadas. Por otro lado se utilizarán suites de tests automáticos para asegurarse de que los cambios realizados en la librería a publicar mantienen la consistencia en sus resultados.

2.4 Organización y equipo

2.4.1 Esquema organizativo

La estructura organizativa del equipo es la siguiente:

- **Director de proyecto:** es el responsable del seguimiento del proyecto en base a reuniones semanales con el equipo del proyecto, encargándose también de la validación de los resultados de las diferentes fases del proyecto.
- **Equipo de desarrollo:** es el responsable del desarrollo del proyecto. En este caso estará compuesto por una única persona que firma el presente documento.
- **Equipo de diseño de interfaz:** es el responsable del diseño de la interfaz y de la creación de los estilos CSS, siendo validado tanto por el equipo de desarrollo como por el jefe de proyecto.

2.4.2 Equipo de trabajo

■ **Descripciones de los puestos de trabajo**

- **Director de proyecto:** Su función es realizar las actividades de organización, coordinación y seguimiento del proyecto, incluyendo la validación de los diferentes productos intermedios. El director de proyecto requiere conocer a fondo las diferentes actividades propias de la planificación y supervisión del proyecto, contando además con una serie de conocimientos básicos de la naturaleza del proyecto y sobre todo de lo que el cliente espera de la aplicación para ser capaz de validar las diferentes partes del proyecto.
- **Analista:** Su función es realizar el documento de requisitos a partir del estudio de la problemática que enfrenta el proyecto. Debe contar con experiencia en la recolección y extracción de requisitos de diferentes medios como pueden ser las entrevistas y el funcionamiento y/o documentación de otras aplicaciones. Deben conocer las necesidades del equipo de diseño para lograr un documento de requisitos completo que permita trabajar sobre el mismo para obtener los documentos de diseño. Deben conocer técnicas de entrevista y poseer un conocimiento amplio pero generalizado del mundo del software.
- **Ingeniero de Software:** Su función es realizar las labores de diseño para el desarrollo y despliegue de la aplicación. Debe conocer a fondo las herramientas de diseño estándar y debe tener experiencia amplia tanto en diseño como en desarrollo para comprender las repercusiones de sus documentos de diseño.
- **Programador:** Su función es desarrollar y poner a punto los programas y subprogramas descritos en el diseño. Los programadores deben estar capacitados para entender completamente los documentos de diseño y desarrollar las aplicaciones y subaplicaciones de acuerdo a buenas practicas. Además, requieren conocimiento directo de sistemas de despliegue para la correcta preparación del mismo.
- **Diseñador web:** Su función es el diseño gráfico y conceptual de los portales web con los que cuenta el proyecto. Debe conocer las diferentes tecnologías que conforman el panorama web actual y buenas practicas de diseño de páginas web.
- **Técnico de sistemas:** Su función es la puesta a punto de la infraestructura hardware necesaria para el proyecto. Requiere cierto conocimiento de diseño de redes, ensamblado de hardware e instalación de sistemas operativos.

En el caso de este proyecto, estos roles se reparten entre 4 actores; la dirección del proyecto recae conjuntamente sobre *Irantzu Vergara* y sobre *Francisco Javier Noguera* de la UOC. El análisis, ingeniería, programación y mantenimiento recae sobre *Sergio Blanco*, el alumno. El diseño recae sobre una empresa subcontratada, *Dr. Minsky*.

2.5 Planificación

Se consideran las siguientes tareas a realizar:

- Análisis de requisitos.
- Diseño de la solución.
- Desarrollo de la librería libre.
- Desarrollo del puente entre la aplicación web y el sistema de gestión de Elson Sistemas.
- Desarrollo de la sección de profesionales de la aplicación web.
- Implantación de la librería sobre el portal.
- Implantación de la aplicación web sobre el VPS y optimización.
- Confección de la memoria de proyecto.
- Liberación de la librería.

De esta forma, se consideran los siguientes hitos:

- Análisis de requisitos. Debe finalizarse para el 5 de Abril del año 2011 como tarde.
- Diseño de la solución. Debe finalizarse para el 15 de Abril del año 2011 como tarde.
- Desarrollo de la librería libre. Debe finalizarse para el 28 de Abril del año 2011 como tarde.
- Desarrollo del puente entre la aplicación web y el sistema de gestión de Elson Sistemas. Debe finalizarse para el 28 de Abril del año 2011 como tarde. Se considera un periodo de optimización y prueba en el mes siguiente, pero para este momento la implementación debe estar hecha.
- Desarrollo de la sección de profesionales de la aplicación web. Debe finalizarse para el 28 de Abril del año 2011 como tarde. Se considera un periodo de optimización y prueba en el mes siguiente, pero para este momento la implementación debe estar hecha.
- Implantación de la librería sobre el portal. Debe finalizarse para el 18 de Mayo como tarde.
- Implantación de la aplicación web sobre el VPS y optimización. Debe finalizarse para el 28 de Mayo del año 2011 como tarde.
- Confección de la memoria de proyecto. Debe finalizarse antes del 1 de Junio del año 2011.
- Liberación de la librería. Debe finalizarse antes del 6 de Junio del año 2011.

En cuanto a número de horas, se considera:

- Especificación de requisitos: 25 horas
- Diseño de la solución: 40 horas
- Desarrollo: 70 horas
- Documentación: 25 horas
- Total: 160 horas

2.6 Análisis de riesgos

El desarrollo y utilización de la librería para hacer una capa de interacción más rica es el gran riesgo de este proyecto ya que este depende del resultado de la implementación de la librería y por lo tanto, repercute en un riesgo muy difícil de asumir. Por ello, se ha decidido dividir el proyecto en partes diferenciadas; por un lado se completa el portal en su estado actual, lo cual repercute en un riesgo

mínimo y permite tener un resultado estable y adecuado. Por el otro lado, se implementa la librería y se hace una ramificación del portal utilizando ésta nueva tecnología.

De esta forma, las consecuencias se reducen a un experimento fallido, repercutiendo en coste de personal y tiempo pero dejando al proyecto en buen lugar. Aun con todo, y para minimizar el riesgo de que la implementación y uso de la librería quede en un experimento fallido, se plantea el desarrollo de la misma de forma ágil, comprobando la implementación progresivamente contra el caso de uso real (el portal web de Sminn) de forma que pueda hacerse frente a los problemas lo antes posible.

2.7 **Presupuesto**

Dado que este proyecto se realiza conjuntamente con una empresa, una parte de las horas dedicadas al proyecto serán imputadas a la propia empresa. Considerando que la especificación y diseño se harán en el marco de la empresa, se imputarán 65 horas del alumno a la empresa. Se considera también que parte del desarrollo recaerá en el mismo marco, por ejemplo, un 25 % de las horas de desarrollo, otras 18 horas. En total se estiman 85 horas a imputar.

Se calculan por otro lado unas 10 horas de trabajo de dirección sobre *Irantzu Vergara*, también a imputar a la empresa. El coste anual del servidor dedicado y el mantenimiento anual de la solución a realizar también han de ser imputados a la empresa, considerando 50 horas anuales dedicadas a mantenimiento. De esta forma:

- Coste de horas de trabajo de Sergio Blanco: 1000€
- Coste de horas de trabajo de Irantzu Vergara: 250€
- Primera anualidad de servidor dedicado: 300€
- Total: 1550€

El coste anual a partir de entonces se estima en 890€.

3. ANÁLISIS

En el presente capítulo se encuentran los diferentes requisitos que el sistema final deberá cumplir y algunas consideraciones en cuanto a la interfaz de usuario. Los requisitos detallan todas las restricciones a tener en cuenta durante las fases de diseño y desarrollo para cada una de las partes del proyecto.

3.1 Requisitos

3.1.1 Requisitos de la zona privada de profesionales

- El acceso a la zona privada de profesionales se realizará mediante un usuario y contraseña creado por el cliente durante el proceso de registro.
- Un cliente registrado no podrá acceder a la sección privada de profesionales hasta que Elson Sistemas valide manualmente a dicho cliente.
- Elson Sistemas debe poder delimitar a que opciones de comercio electrónico tiene acceso cada cliente.
- El cliente no podrá comprar directamente, sino que podrá realizar peticiones de oferta. Estas peticiones de oferta se realizarán igual que una compra típica en cualquier tienda web, salvo que podrán añadirse varias veces el mismo elemento con diferentes comentarios y/o cantidades.
- Si Elson Sistemas lo permite, el cliente tendrá acceso a todo su historial de peticiones de oferta, ofertas y pedidos finales a través de la zona privada de profesionales.
- Los documentos administrativos a los que tendrá acceso el cliente serán actualizados automáticamente cuando Elson Sistemas realice cambios en ellos en su ERP interno.
- Dichos documentos harán referencia a cualquier documento relacionado de forma que el cliente pueda seguir la línea de transacciones.
- El cliente podrá modificar sus datos personales. Los cambios serán notificados a Elson Sistemas (ERP y correo electrónico) para que los nuevos datos puedan ser validados.
- El cliente podrá suscribirse a la newsletter del portal.
- La zona privada incluirá un apartado para gestionar el carro de petición de oferta actual.
- El carro de petición de oferta será guardado entre sesiones.
- Un cliente registrado tendrá acceso a más recursos de los productos.
- La zona privada de profesionales contendrá información de ayuda en forma de un apartado de preguntas frecuentes y un apartado de información de cultura de producto.
- La comunicación entre el ERP y el portal será bidireccional y se basará en la notificación de eventos de adiciones o cambios de información.

3.1.2 Requisitos de la librería para desarrollo de RIAs

- La librería deberá mantener separadas sus partes mediante interfaces definidas.

- La librería deberá permitir la definición de estructuras de contenido del servidor, pero no forzar el hacerlo.
- La librería deberá gestionar la carga y descarga de elementos automáticamente y deberá dar facilidades al desarrollador para actuar ante dichos eventos.
- La librería deberá mantenerse simple y desacoplada de otras soluciones, de forma que puedan construirse diferentes soluciones sobre ella.
- La librería no debe asociarse a ningún framework Javascript y debe permitir el uso de cualquiera de ellos.
- La librería debe soportar la ampliación de la misma fácilmente.
- La librería utilizará en la medida de lo posible aspectos ya implementados de las tecnologías que utiliza y de la forma más usual y estándar posible para minimizar su curva de aprendizaje.

3.2 Caso de uso: Zona privada de profesionales

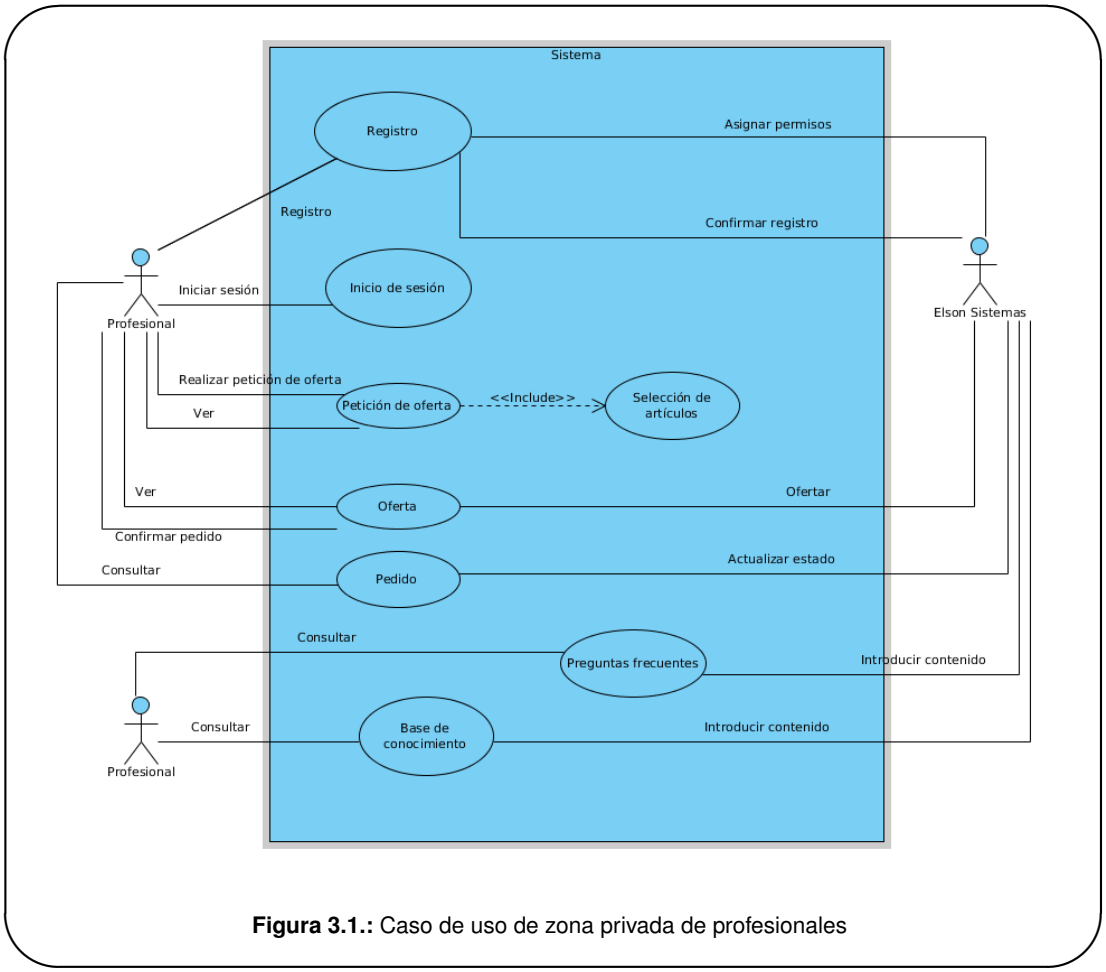

Figura 3.1.: Caso de uso de zona privada de profesionales

La zona privada de profesionales pone a disposición de los mismos contenidos que estos pueden consultar (dicho contenidos son introducidos y modificados por Elson Sistemas), permite a los profesionales registrarse, iniciar sesión y gestionar su usuario y lo más importante, permiten realizar pedidos de producto. El sistema utilizado es algo inusual con respecto al caso típico de comercio web; en lugar de realizar directamente un pedido, el usuario profesional puede realizar peticiones de oferta.

Una petición de oferta no es necesariamente el pedido que va a realizar, sino una petición de precios y plazos para diferentes elementos en diferentes cantidades.

Elson Sistemas responde al usuario con una oferta en la que el usuario puede ver dichos precios y plazos. Es posible que a su vez la empresa se ponga en contacto con el profesional para asesorarle. El profesional puede a partir de una oferta lanzar un pedido con los elementos que finalmente quiere, teniendo que corresponderse los números de elementos ya que son la base para el precio. Elson Sistemas considera que si un cliente desea poder realizar pedidos directamente ya que suele pedir lo mismo o ya se han establecido las condiciones típicas para los pedidos este puede solicitar que sus peticiones de oferta sean tratadas como pedidos directamente. Este dato queda registrado en el software de gestión interno de la empresa que es desde el que se realiza la gestión de peticiones de oferta, ofertas y pedidos.

3.3 Interfaz de usuario

3.3.1 Zona privada de profesionales del portal web

El acceso a la zona privada de profesionales se realizará a través de la sección “Profesionales” (ver figura 3.2) del portal web, iniciando sesión con un usuario previamente registrado y validado.

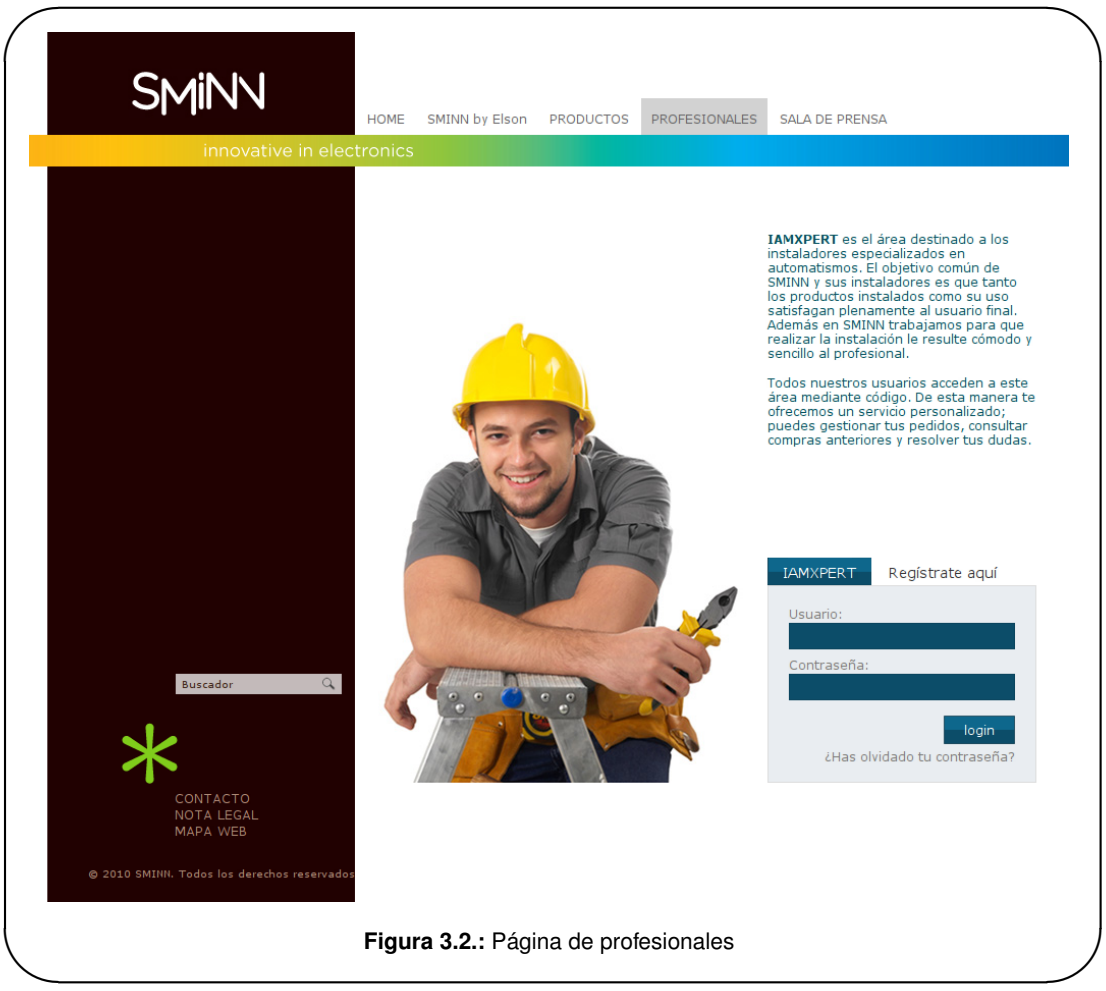

Figura 3.2.: Página de profesionales

La creación de usuarios se realizará mediante un formulario como el que puede verse en la figura 3.3.

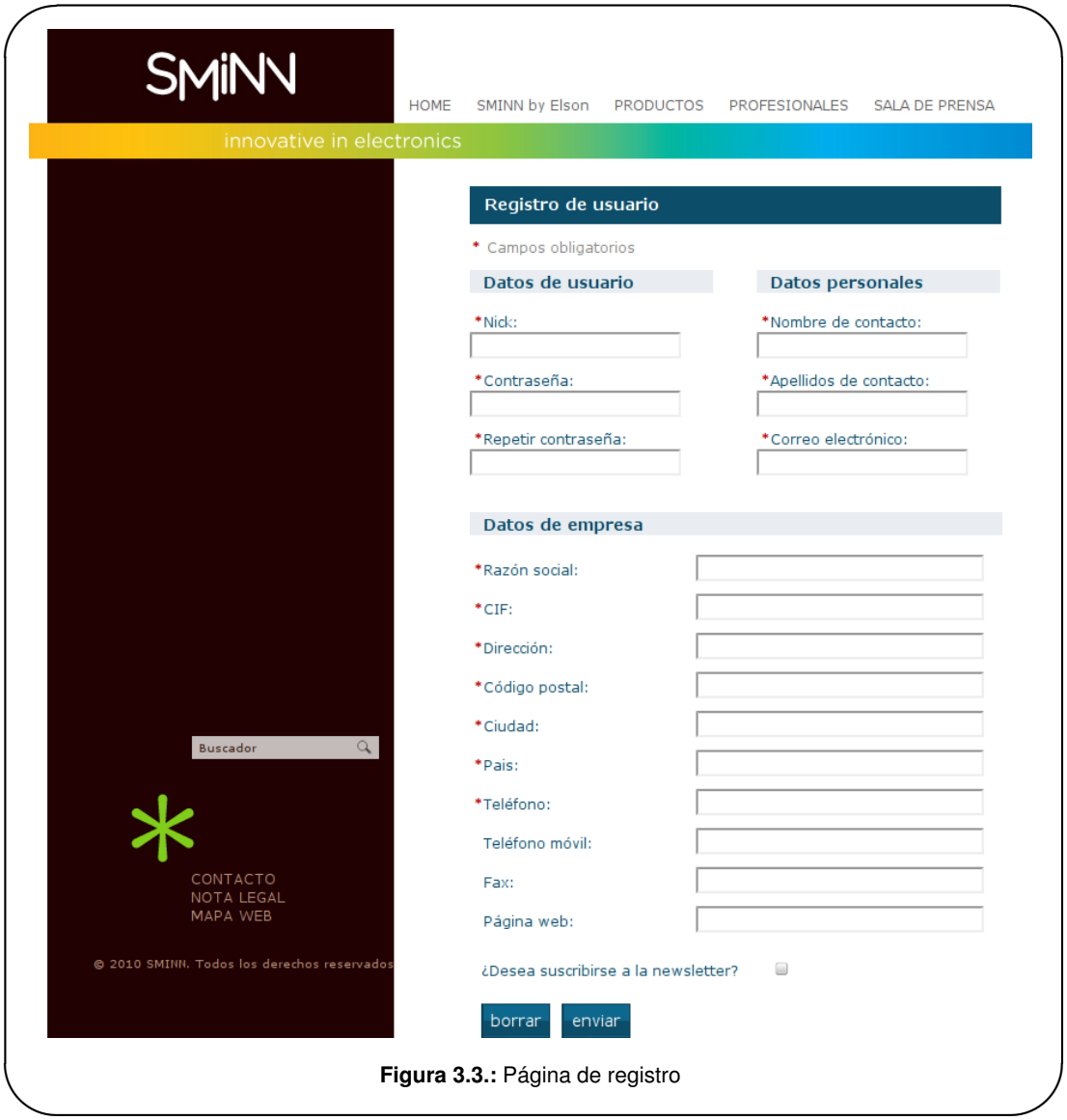

Figura 3.3.: Página de registro

Una vez se inicia sesión se accede a la zona privada. Esta zona privada (ver figura 3.4) tendrá un formato de página y cabecera diferente.

Figura 3.4.: Zona privada de profesionales

La zona de profesionales debe tener diferentes secciones:

- **Carro de oferta**

Esta sección (ver figura 3.5) permitirá al usuario ver su carro de “compra” (petición de oferta realmente) y realizar diferentes operaciones como duplicar y borrar lineas, editar comentarios, vaciar el carro y realizar la petición de oferta per sé.

- **Mis transacciones**

Esta sección permitirá al cliente consultar su histórico de peticiones de oferta, ofertas y pedidos. Las peticiones de oferta es lo único que el cliente realiza a través de la web. Las ofertas son respuestas a dichas peticiones. Si el cliente está contento con la oferta, puede proceder a realizar el pedido ya sea de la oferta completa o de partes de la misma. Dado que este proceso conlleva en la mayor parte de los casos comunicación verbal con el cliente, Elson Sistemas ha decidido que no se implementará la consecución de pedidos desde la web.

Al entrar en la zona de “Mis transacciones” (ver figura 3.6) el usuario podrá ver un resumen de sus transacciones y podrá acceder a cada una de ellas por su tipo o directamente si está en la lista de las últimas transacciones.

Cada tipo de transacción tiene su propia página de listado. Salvo por las columnas correspondientes a cada transacción la interfaz de todas ellas será la misma. Cabe destacar que se pretende implementar un sistema de filtrado temporal cuyo diseño inicial considera un listado de los meses disponibles para el año actual en horizontal y un dropdown de años vertical. La interfaz de estas páginas será similar a lo que se puede ver en la figura 3.7.

Las transacciones en si tienen también una estructura similar entre ellas, con la salvedad de que las ofertas y pedidos tienen un total de precio además de las columnas adicionales. y que hay columnas que pueden referenciar a otros documentos, ya que cada linea de una oferta o pedido

Figura 3.5.: Carro de oferta

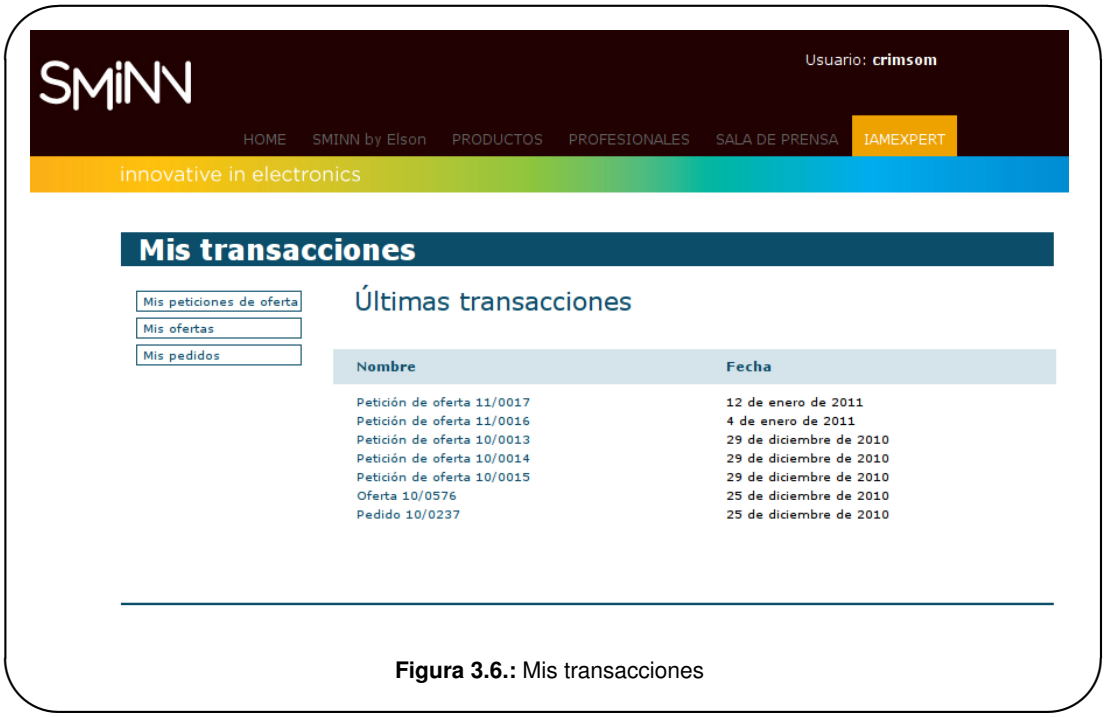

Figura 3.6.: Mis transacciones

Figura 3.7.: Peticiones de oferta

puede referenciar a un documento anterior. La interfaz será similar a lo que se puede ver en la figura 3.8.

- **Preguntas frecuentes**

Las preguntas frecuentes estarán divididas en grupos y cada pregunta-respuesta constará de un texto para cada uno. La interfaz será similar a lo que se puede ver en la figura 3.9.

- **Información sobre productos**

Al entrar en la sección de información sobre productos se presentará un índice alfabético para organizar la información. La interfaz de esta sección aún no ha terminado su fase de diseño y por lo tanto lo que se puede ver en la figura 3.10 es solo un diseño previo.

Cada entrada es un artículo cuya interfaz será similar a lo que se puede ver en la figura 3.11.

- **Edición de datos de usuario**

La interfaz de edición de datos usuario será igual que la de registro de nuevo usuario.

3.3.2 Consideraciones de interfaz de usuario en cuanto a la implantación de la librería

La implantación de la librería a desarrollar implicará cuanto menos que se cargará y descargará únicamente lo que tenga que cambiar en cada momento y que el contenido se adaptará mejor a diferentes pantallas. Asumir que solo cambiará en ese aspecto sería desaprovechar el potencial de la tecnología y una labor a realizar a posteriori del proyecto consiste en repensar el flujo de interfaz y los mecanismos de presentación e interacción para mejorar el portal. Se hará un trabajo de introducción a este aspecto en el portal para que sirva a modo de prueba de concepto y punto de partida para el trabajo posterior que se realizará junto a una empresa especializada en diseño e imagen.

Figura 3.8.: Oferta

Figura 3.9.: Preguntas frecuentes

Figura 3.10.: Índice de información sobre productos

Figura 3.11.: Artículo de información sobre productos

4. TECNOLOGÍAS

En el presente capítulo se detallan las tecnologías investigadas y utilizadas para el desarrollo del proyecto por medio de una introducción al uso y funcionamiento de cada una de ellas. Las tecnologías que se encuentran detalladas a continuación son:

- Django
- South
- Rosetta
- JQuery

4.1 Django

Django[9] es un framework web escrito en *Python*[36] que se basa completamente en el principio *DRY*, automatizando en lo posible todas las tareas repetitivas en la construcción de una aplicación web pero sin ocultar los aspectos de bajo nivel de las tecnologías que subyacen y en la arquitectura MVC.

Las características más interesantes de Django son:

- **Mapeador Objeto-Relacional:** Basta con escribir las clases que conforman el modelo de la aplicación y el sistema se encarga de crear la estructura de base de datos y una API de acceso a datos completa.
- **Interfaz administrativa:** Django viene por defecto con una aplicación de portal administrativo que se monta sobre cualquier aplicación web con solo unos pocos parámetros.
- **Asignación de URLs a vistas por expresiones regulares:** A cada expresión regular se le asigna una clase o una función que se encarga de preparar la vista.
- **Sistema de plantillas:** Aunque puede utilizarse cualquier sistema de aplantillado escrito en Python, Django implementa un sistema de aplantillado que expone ciertos elementos del lenguaje y las variables que el programador decida.
- **Soporte para middleware:** Funcionalidad que trabaja sobre las peticiones y respuestas antes del código propio, lo que permite por ejemplo integración con motores de caché como memcached.
- **División entre proyectos y aplicaciones:** Las aplicaciones son funcionalidad reutilizable que se puede introducir fácilmente en cualquier proyecto.
- **Servidor web ligero de desarrollo:** evita al equipo de desarrollo tener que configurar un servidor web completo durante el desarrollo de la aplicación.
- **Vistas genéricas:** Vistas que realizan funcionalidad típica como listar objetos o mostrar los detalles de un recurso ya implementadas.
- **Sistema de formularios:** Es posible preparar las formularios utilizando clases de Django, de forma que gran parte de la validación y presentación de errores puede reutilizarse.

En la versión 1.3 de Django (la última al momento de publicar esta memoria) se ha realizado la transición de vistas genéricas basadas en funciones a vistas genéricas basadas en clases. Estas vistas genéricas basadas en clases dan un paso más allá en cuanto a reutilización de lógica de preparación de vistas; Django provee varias vistas genéricas muy parametrizables para mostrar vistas

de listados de entidades, detalles de entidades, etc. . . y es razonablemente sencillo implementar vistas genéricas propias.

Las nuevas vistas genéricas se basan en dos componentes: la vista genérica base (de la cual se hereda, siendo el punto más alto de la jerarquía la clase View, la cual implementa los mecanismos mínimos para una vista genérica) y los Mixins, pequeñas clases que realizan una funcionalidad determinada dentro del marco de una vista (renderizado a plantilla, renderizado a formatos de texto, filtrado de entidades, búsqueda, . . .)

Como se verá más adelante la librería principal del presente proyecto se basa sobre esta tecnología; se detallará por tanto su uso en el capítulo de *Desarrollo*.

Para más información, los creadores de Django escriben varios libros[10][1] con documentación sobre Django que también merece la pena consultar.

4.2 South

South es una aplicación para Django que añade una de las funcionalidades más solicitadas en el mundo Django: actualización del esquema de la base de datos a partir de cambios en el modelo. En el ciclo de desarrollo de una aplicación web con el framework Django es habitual implementar el modelo de datos en primer lugar y aunque es posible conseguir un modelo de datos completo a la primera, es más habitual ir adaptándolo según se van implementando funcionalidades nuevas, en la mayoría de los casos manteniendo los datos ya introducidos en la base de datos.

Sin South, las modificaciones se complican; o se modifica el esquema de la base de datos manualmente o bien se genera desde 0 una nueva base de datos. En la segunda opción si se quieren mantener los datos es habitual utilizar el comando *dumpdata* de Django para obtener una copia de seguridad de los datos en formato JSON, modificar manualmente el fichero JSON para incluir los cambios, realizar los cambios en el modelo de datos, regenerar la base de datos y en último lugar, utilizar *loaddata* para restaurar los datos. Este proceso es engorroso, especialmente si hay que realizarlo con frecuencia.

Para solucionar esto, South se basa en un sistema de análisis y comparación; hay que registrar un estado inicial del modelo de datos y a partir de ahí, se pueden generar *migraciones* (scripts Python generados por el propio South y lanzados desde Django) que a partir del estado anterior saben realizar el cambio en la base de datos, preguntando al usuario en caso de duda de cómo realizarlos. Los cambios se mantienen en un orden secuencial y el sistema, determinando el punto en el que se encuentra la base de datos actual, se encarga de ir realizando las modificaciones. Cada base de datos lleva dentro unas tablas dedicadas a South en donde se registra el estado actual, de forma que es posible utilizar estos scripts para diferentes bases de datos en diferentes estados.

Para más información conviene consultar la documentación oficial del proyecto South[42].

4.3 Rosetta

Rosetta[40] es una aplicación Django que facilita la traducción de aplicaciones web Django. Django basa su sistema de localización en *gettext*; utilizando un comando propio de Django se buscan todas las cadenas a traducir tanto en las plantillas como en las vistas y se genera un fichero de traducción para el idioma solicitado. Este fichero de traducción debe ser compilado a posteriori, también con un comando de Django que no deja de ser un frontend a gettext. Este mecanismo puede resultar pesado, especialmente teniendo en cuenta que generalmente la traducción la realiza un tercero con conocimiento muy básico de informática y conocimiento nulo de GNU/Linux.

La solución de Rosetta es ofrecer una interfaz web administrativa destinada a la traducción dentro de la propia aplicación web, de forma que cualquier usuario al que se le de permiso de traducción puede traducir fácilmente las diferentes cadenas de texto y ver el resultado prácticamente al momento.

4.4 JQuery

JQuery es una librería Javascript que simplifica de manera significativa la programación web en el lado del cliente[23]. Además de proporcionar una manera más sencilla de realizar tareas comunes en el desarrollo web sirve como capa de abstracción del navegador. Su autor original es John Resing y su funcionalidad incluye:

- Manipulación del DOM[12]
- Manejo de eventos
- Manipulación del CSS
- Comunicación AJAX

Esta librería más que una colección de funciones, aunque también aporta funciones independientes, aporta el objeto JQuery. El objeto JQuery puede ser creado a partir de un selector CSS o crearse a partir de un String de HTML. Estos objetos permiten alterar elementos del DOM, borrarlos o crearlos a través de los métodos del objeto JQuery que los contenga. Además estos objetos pueden ampliar su funcionalidad de manera sencilla. Así que los plug-ins y librerías para JQuery simplemente añaden nuevos métodos a estos objetos.

```
//Rellenar un elemento de ID bienvenida con un string
$("#bienvenida").html("hola <B>Lucia<//B>")

//Cambiar el color de todos los titulos de tipo 1 a rojo
$("H1").css("color","red")

//Agregar al DOM un nuevo elemento
mensaje= $("<div>hola mundo<//div>")
$("body").append(mensage)
```

Listado 4.1: Ejemplos de JQuery

5. ESPECIFICACIÓN DEL DISEÑO

5.1 Introducción

En el presente capítulo se mostraran los resultados del proceso de diseño del proyecto, tratando así de exponer las decisiones tomadas en el mismo. A lo largo de este capítulo se desarrollarán los siguientes epígrafes:

- **Diseño de la zona de profesionales:** Especificación de diseño correspondiente a la zona de profesionales.
- **Diseño de la librería para desarrollo de RIAs:** Especificación de diseño correspondiente a la librería implementada.

5.2 Diseño de la zona de profesionales

5.2.1 Introducción

La zona de profesionales del portal web de Sminn debe incluir los siguientes elementos:

- Vistas y plantillas correspondientes a cada sección de la zona privada
- Modificación de vistas y plantillas de secciones públicas con contenidos solo visibles por profesionales
- Mecanismo de sesión
- Sistema de permisos para acceso a diferentes secciones
- Descargas privadas
- Sincronización con ERP de Elson Sistemas
- Aspectos de seguridad para la zona privada

5.2.2 Vistas y plantillas

La zona de profesionales debe contar con varias subsecciones de contenido. El diseño de estas subsecciones no difiere con respecto al resto del portal web ya desarrollado y en general con respecto al flujo de trabajo del framework de Django salvo en los aspectos que siguen en la presente sección. Al igual que en el resto de la aplicación, cada vista consultará un mecanismo de caché configurable para sus contenidos antes de generarlos dinámicamente (solo si es necesario) y se utilizarán plantillas modulares para mayor reutilización.

5.2.3 Sesiones y permisos

Se debe proveer un mecanismo para iniciar y cerrar sesión. Las sesiones no deben ser persistentes (una vez se cierra el navegador la sesión queda cerrada) por seguridad pero si deben mantenerse ciertos datos, como el carrito de oferta del cliente, entre sesión y sesión. Para facilitar el desarrollo de las vistas y plantillas, una solución efectiva para alterar el comportamiento de los diferentes

componentes según los datos propios de la sesión sería exponer los datos del usuario a través de algún tipo de mecanismo de contexto, de forma que todas las plantillas tengan acceso a los datos del usuario al momento de ser renderizadas. Estos datos llevarán consigo los permisos efectivos del usuario, de forma que también será posible escoger qué renderizar según los mismos.

Dado que hay secciones completas que requieren unos permisos concretos, también es importante implementar algún sistema para comprobar permisos para una vista completa. Una idea a seguir para esto es crear un decorador de función que intercepte la llamada a la función y filtre según los permisos.

5.2.4 **Descargas privadas**

Otro problema es el de las descargas privadas. Los recursos privados solo deberían poder ser descargados por usuarios lícitos que hayan iniciado sesión. El problema es que, por regla general, los ficheros los sirve el servidor web y la autenticación del servidor web y de Django son separadas (y por lo tanto no es posible iniciar sesión solo en Django). Hay que implementar algún método de compartir autenticación entre la aplicación web y el propio servidor web o por lo menos encontrar una forma de pasar por la aplicación web antes de descargar el fichero con su ruta para poder determinar los permisos sin que sea la propia aplicación web la que se encargue de la descarga, ya que eso multiplicaría el consumo de memoria y mantendría vivo el hilo de una petición por tanto tiempo como dure la descarga.

Véase el capítulo de *Desarrollo* para conocer la solución final.

5.2.5 **Aspectos de seguridad**

No basta con configurar el servidor web con un certificado SSL[44] firmado por alguna autoridad confiable, sino que hace falta asegurarse de que el usuario utilice el protocolo HTTPS[20] en las vistas pertinentes. Dado que los certificados SSL generalmente certifican un dominio, hay que asegurarse además que los dominios equivalentes redirijan al dominio certificado para que el certificado no de ningún problema. Este aspecto probablemente tendrá una parte de configuración del servidor web y una parte de filtrado en la aplicación web que se encargue de la redirección a HTTPS en donde sea necesario.

Nótese que sería factible, si se sigue un esquema jerárquico para las URLs de las vistas, solventar este problema mediante la configuración del servidor web con un módulo de redirección siempre y cuando todas las URLs por debajo de un nivel exijan el mismo nivel de seguridad. Si existe la posibilidad de que el nivel de seguridad sea arbitrario por vista (y es interesante mantener esa flexibilidad) es necesaria alguna solución que no dependa de la configuración del servidor web o dependa lo mínimo posible.

5.2.6 **Comunicación entre sistemas**

Elson Sistemas cuenta con una aplicación desarrollada internamente que sirve como su ERP[13]. Se trata de una aplicación de escritorio sobre una base de datos centralizada pero que por motivos de seguridad no debe poder accederse desde fuera de la red interna de la empresa. Al ser una aplicación de escritorio (y poder haber en un mismo momento varias instancias simultaneas) no existe una plataforma de servicio única con la que comunicarse. De esta forma, y ante el hecho de que pueden existir un número variable de instancias, la comunicación desde el ERP hacía la aplicación web debe hacerse de alguna forma iniciada por el propio ERP y que permita varias conexiones simultáneas. Cada vez que cualquier instancia del ERP realice alguna transacción deberá enviar la misma de forma segura a a la aplicación web.

En cuanto a las transacciones originadas desde la web, y dado cómo está hecho el ERP, la mejor opción consiste en implementar algún tipo de servicio en la red privada de Elson Sistemas que sea

accesible desde redes externas. Este servicio recibirá las peticiones y trabajará contra la base de datos. Dicho servicio será implementado por el personal de Elson Sistemas.

Dada la situación se utilizará un sistema basado en mensajes. Los eventos desde el ERP pueden enviarse como una petición POST por HTTPS a una vista específica del portal web y los eventos desde el portal web pueden enviarse de la misma forma al adaptador del ERP, de forma que sea muy sencillo implementar dicho adaptador. El formato a usar debería ser uno sencillo, como JSON.

Quedan en este diseño dos problemas por resolver: ¿qué pasa si el servicio de unos de los lados está caído? y ¿Queda asegurado el orden de las transacciones? El segundo problema es fácil de resolver; cada petición tendrá una respuesta de transacción correcta o incorrecta. Hasta recibir una respuesta de transacción correcta no se procederá a la siguiente transacción. La primera pregunta depende ligeramente de la segunda; basta con reintentar a intervalos regulares la transacción, pero obviamente no debe pasarse a la siguiente transacción hasta no haber terminado la primera con éxito. Obviamente es importante, si se quiere mantener cada lado separado de los problemas que pueda tener el otro, que ambos almacenen cada uno en su correspondiente base de datos la información que necesitan, de forma que no tengan que solicitarla cada vez.

Véase el capítulo de *Desarrollo* para más detalles.

5.3 Diseño de la librería para desarrollo de RIAs

5.3.1 Introducción

El diseño de la librería para desarrollo de RIAs sobre Django resulta más complejo que el de la zona de profesionales. La zona de profesionales está basada en una solución ya comenzada y por tanto sigue sus formas, estando su complejidad en la resolución de algunos problemas derivados de la necesidad de seguridad o protección mientras que la librería debe estar diseñada para soportar tantos usos como sea posible, de forma limpia y con un nivel de integración elevado con Django y otras soluciones que los desarrolladores puedan utilizar. Siguiendo los principios de Python y Django, la librería no debería ocultar el funcionamiento tecnológico de lo que hace, sino facilitar su uso y evitar que los desarrolladores tengan que escribir demasiado, o peor aún, repetir cosas.

Para ello, el diseño de la librería se hace a varios niveles:

- Diseño de arquitectura: Primera aproximación basada en la distribución arquitectónica del sistema, en este caso, la arquitectura cliente servidor y en particular la arquitectura web con sus diferentes componentes.
- Diseño de flujo: Segunda aproximación en la que se detalla las labores que debe ejecutar cada componente y en qué orden.

Es recomendable la lectura del anexo *Manual del programador* antes de leer esta sección ya que así el lector estará más familiarizado con los elementos de la librería y su funcionamiento, lo que le permitirá asimilar más fácilmente las decisiones de diseño aquí expuestas. No se realiza un diseño de clases por su sencillez y por estar mejor expuesto en el capítulo de *Desarrollo*.

5.3.2 Arquitectura

A nivel de arquitectura es importante para el diseño de la librería fijarse en el funcionamiento general de la tecnología web. En esta arquitectura se tiene típicamente un servidor y varios clientes. Desde el comienzo de la web los papeles de ambos se han ido transformando y existe cierta flexibilidad en cuanto a los roles de cada parte. A un nivel muy básico con respecto a la arquitectura se puede considerar quién lleva el estado de la aplicación: el servidor, el cliente (navegador) o ambos. Desde un punto de diseño y desarrollo de aplicaciones es importante mantener la flexibilidad, lo que supone que cualquiera de los lados de la arquitectura pueda gestionar parte del estado de la aplicación o incluso el estado completo.

Quién gestiona el estado tiene sus repercusiones en el flujo de información entre los dos lados. En una aplicación estática no es necesario conservar estado y por lo tanto se puede considerar cada petición como un entorno aislado. Pero en cualquier aplicación web moderna destaca el dinamismo; según la interacción, datos introducidos tanto al momento como previamente, situación en el tiempo, incluso datos ad-hoc como sensores externos cada petición puede resultar en una respuesta diferente. Si el servidor no contempla ningún tipo de estado es necesario enviar al menos parte del estado con las peticiones para que éste pueda usar dichos datos para elaborar la respuesta. Es más habitual que sea el servidor el que gestiona el estado; el cliente realiza peticiones isoladas y el servidor mantiene un marco de sesión por cada cliente según el cuál las respuestas varían.

En cualquier caso, y según la situación, puede generalizarse que un desarrollador de aplicaciones va a poder necesitar el envío de datos arbitrarios en cualquier dirección. Aunque existen nuevas tecnologías para permitir una comunicación bidireccional asíncrona entre servidor y cliente como la API de WebSockets[46], tradicionalmente la comunicación se origina en el cliente y por tanto la comunicación desde el servidor es el resultado de una petición. Aunque se prevee una transición a un modelo de comunicación bidireccional, este modelo es solo necesario para aplicaciones que necesiten sincronización entre varios clientes (por ejemplo aplicaciones de dibujo colectivas, juegos multijugador, etc.). A la mayoría de aplicaciones les basta con realizar peticiones a intervalos regulares y refrescar la capa de presentación. Dado que Django además se fundamenta sobre el modelo clásico, se trabajará en principio con este.

La comunicación entre el cliente y el servidor se realiza mediante el protocolo HTTP o HTTPS, usualmente mediante los métodos GET y POST (se usan adicionalmente PUT y DELETE en las interfaces REST[38]). Aunque cada método tiene un propósito determinado (GET para solicitar un recurso, POST para enviar información al servidor, PUT para modificar o crear un recurso y DELETE para borrar un recurso) la utilización de los mismos es bastante flexible, de forma que muchas veces se utilizan peticiones GET para enviar datos y peticiones POST para pedir recursos. Un caso típico de la utilización de POST para la solicitud de recursos es en el caso de lo que se conoce como peticiones AJAX (peticiones asíncronas con respuesta típicamente en XML). Aunque no se envíen datos, estas peticiones se suelen realizar mediante peticiones POST. Una razón habitual es para ocultar ligeramente los datos, si los hay, ya que en una petición GET los datos suelen viajar como parámetros de URL.

Sobre este marco cabe analizar el funcionamiento habitual de una RIA; el servidor web realiza una petición inicial en la que obtiene, como en una página web normal, un documento HTML que hace referencia a diferentes recursos, incluyendo scripts Javascript y hojas de estilo CSS. Haciendo un símil con las aplicaciones de escritorio, los script Javascript descargados forman la aplicación web en sí; se encargan de controlar el flujo de aplicación y solicitar al servidor los datos necesarios, encargándose de adaptarlos para la capa de presentación. Aunque gran parte del proceso de aplicación y lógica de negocio residen en estos scripts, el servidor retiene por su parte algunos puntos de la lógica de negocio, sobre todo los que se quieren ocultar o tengan que ver con seguridad. La separación entre el cliente y el servidor es, para el desarrollador, completa; el servidor acaba convirtiéndose en un servidor de ficheros y datos con algunos procesos de negocio y la relación entre ambos componentes se basa estrechamente en una relación arbitraria de petición-respuesta.

Este mecanismo tiene su parte buena y su parte mala y ambas partes caen en el mismo hecho; el acoplamiento es mínimo, tan mínimo que una aplicación web termina siendo dos aplicaciones separadas en dos lenguajes separados, haciendo muchas veces la trazabilidad y visión de conjunto muy complicados. Es habitual preguntarse al desarrollar este tipo de aplicaciones: ¿Qué scripts se usan para esta página? ¿Cómo? El servidor no tiene prácticamente nada que decir al respecto; éste solo genera XML y HTML para que el cliente lo consuma. Es necesario trabajar estrechamente en el lado cliente incluso en los aspectos en los que es el servidor el que manda. No existe un interfaz común a cada aspecto de una aplicación más allá de la propia conexión entre ambos lados.

La intención de la librería es añadir una capa que aunque acople ligeramente las dos capas de la aplicación, permita obtener una visión general de los componentes de la aplicación, su composición y uso en el lado servidor pero de una forma que la aplicación en sí siga haciéndose en Javascript de forma tradicional. Dicho de otra forma, la idea es tener a cada lado de la arquitectura una capa

muy pequeña de software que establezca un vínculo simbólico y declarativo entre ambos lados, de forma que se siga trabajando a un nivel natural de páginas (un patrón de diseño de interfaces muy demostrado) pero aprovechando el dinamismo y flexibilidad de situar la lógica de control en el cliente. En este diseño el servidor define declarativamente lo que compone cada vista (la granularidad se define en cada aplicación, se puede seguir creando una única vista para hacer una RIA típica o puede subdividirse para permitir mayor reutilización y facilitar el seguimiento) y se lo da al cliente en un formato conocido al realizar una petición, de forma que este puede ponerse en funcionamiento con lo establecido por el servidor.

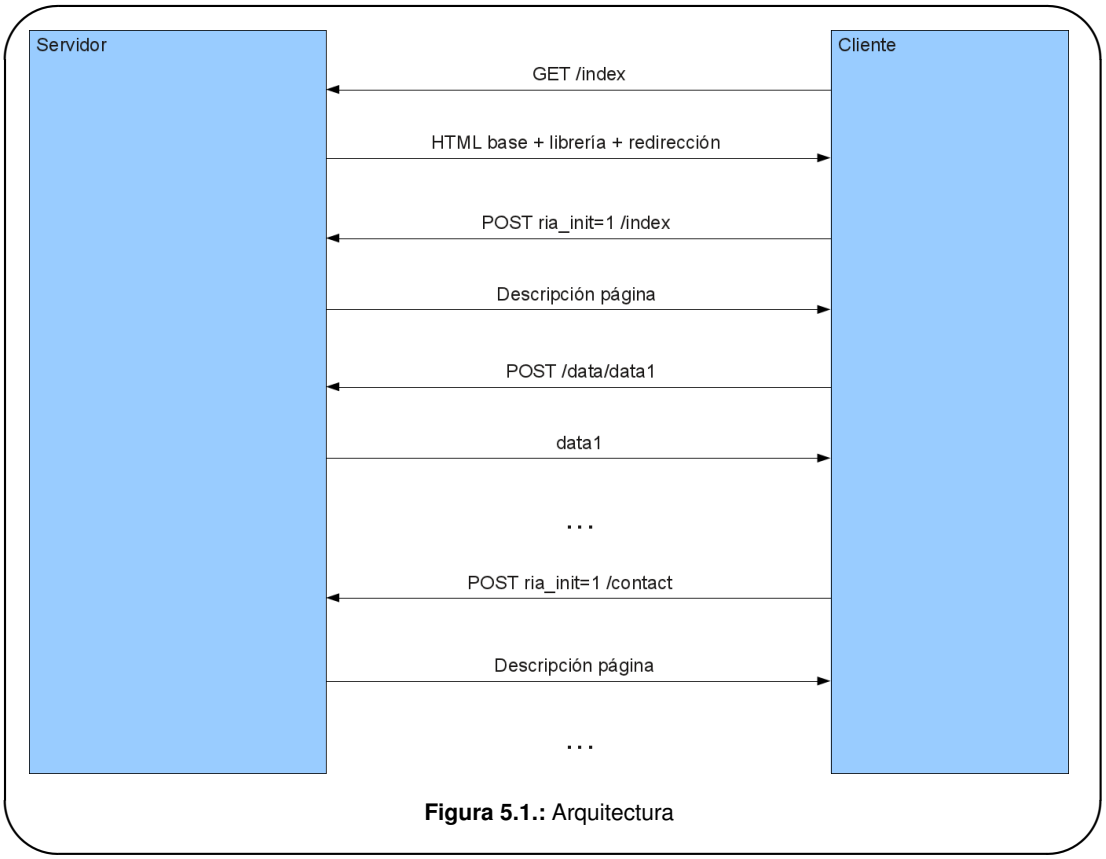

Figura 5.1.: Arquitectura

Siempre debe existir una primera petición que descargue un documento HTML, y en este caso esa petición obtendrá un documento básico con la librería incluida para ponerse en funcionamiento y cargar lo que sea necesario. Para permitir que el usuario pueda acceder a cualquier página y no tenga que pasar por un punto inicial de navegación (muy habitual en las aplicaciones de escritorio y RIAs) se detectará si ya se ha cargado la maquinaría de la librería o no y si hay que cargarla, tras hacerlo se procederá a obtener lo solicitado por el usuario. Esto ofrece una ventaja con respecto al diseño típico de RIAs ya que permite mezclar las ventajas de la navegación tradicional junto a la navegación tipo RIA de forma sencilla y natural.

Nótese que esta arquitectura resulta en un ligero cambio de paradigma con respecto al usual de las RIAs; mientras que estas abandonan el desarrollo dividido en vistas o páginas a favor de un sistema más procedural, el diseño propuesto permite obtener el resultado de una RIA con una metodología similar a la usada para portales web clásicos. Esto tiene su ventaja en que la complejidad se divide y se mantiene más baja y se utiliza el patrón de vistas basadas en páginas, un patrón muy probado y considerado como buen hacer.

5.3.3 Flujo de ejecución

En una RIA podemos considerar principalmente tres tipos de recursos:

- Scripts
- Estilos CSS
- Contenidos

Generalmente los contenidos son cargados asíncronamente cuando se necesitan por medio de los scripts. Los estilos también pueden ser cargados desde los script, aunque es más habitual precargarlos como parte del documento o página. Los scripts son el pilar de una RIA; son el elemento a gestionar y por tanto se deben ofrecer facilidades para su carga, descarga y ejecución. Los ficheros CSS, aunque exista la opción de su gestión manual, pueden ser gestionados de la misma forma y además son similares a los scripts en cuanto a que es posible realizar un preprocesado (en el lado servidor) para ellos para minimizarlos en tamaño o incluso unirlos a otros para minimizar el número de conexiones y por tanto mejorar el tiempo de respuesta.

Los contenidos pueden ser cargados dinámicamente o ser incluidos en la propia respuesta a la petición de página para minimizar el número de peticiones, aunque es más usual hacerlo dinámicamente. En cualquier caso los contenidos deben ser adaptados a la interfaz y situados en algún punto; es obvio que cada página debe constar de una serie de elementos contenedores. Estos elementos contenedores deberían también tener su propio ciclo de vida, de forma que cada uno se encargue de las peticiones y datos que le atañen y tengan capacidad de autoconfiguración y autofuncionamiento, buscando siempre la creación de módulos reutilizables y que mantengan la complejidad baja[2].

De esta forma se considera un flujo de ejecución, primero en el servidor y después en el cliente, que establece como se interrelacionan estos elementos, como se cargan y descargan, cómo funcionan y cómo se gestionan. El primer actor es el servidor; el servidor recibe una petición, la cual puede ser una petición “normal” o una petición del sistema que implementa la librería. Ésta petición hace referencia con su URL a una “página”, es decir, una vista de la aplicación. Si la petición que ha realizado el cliente deja en evidencia que no dispone todavía de su lado de la librería el servidor debe responder con un documento HTML con la librería que automáticamente realice la petición a la página solicitada, esta vez sí, con la librería como vehículo. Una vez llegado a este punto el flujo prosigue normalmente. El servidor contiene una descripción declarativa de la página que incluye:

- Scripts que requiere la página y configuración asociada: Cada página lleva consigo una lista ordenada de scripts que necesita para funcionar; cada uno de estos scripts puede ser minimizado en tamaño, anexado a otros script (siempre manteniendo el orden) y puede tener un tiempo de vida (estático o volátil), es decir, puede ser un script que debe estar presente durante toda la sesión o un script específico de la página en cuestión o de unas pocas páginas, con lo que el sistema puede evitar cargar scripts que ya ha cargado. Es posible que sea necesaria la ejecución de funciones al cargar (inicialización) o al descargar (destrucción), por lo que también se deben poder declarar funciones a llamar para cada caso, lo que puede ser muy útil al utilizar frameworks javascript como JQuery. También es posible que sea deseable que un script vuelva a ser descargado o al menos reinicializado aunque ya estuviera cargado al llegar a una página, por lo que se consideran ambas opciones.
- Estilos CSS que requiere la página y configuración asociada: Cada página lleva asociada una lista ordenada de hojas de estilo CSS que necesita para ser estilada correctamente. Cada hoja de estilo, al igual que en el caso de los scripts, puede ser minimizada y anexada a otras y puede tener un tiempo de vida estático o volátil.
- Grupos de elementos (Layouts): Cada layout es un grupo de elementos de una página; un layout puede tener funciones de inicialización y destrucción y debe tener una lista de elementos, los cuales son los contenedores de contenido per sé y deben ser capaces de autogestionarse.
- Elementos: Los elementos son el elemento mínimo del sistema; un elemento puede ser desde un simple elemento HTML hasta un completo grupo de elementos dinámicos con un script trabajando detrás. Cada elemento pertenece a un tipo de elementos y tiene asociados una

serie de scripts y hojas de estilo necesarios para su funcionamiento, una serie de subelementos (elementos también) y una función opcional a la que llamar al terminar su carga. Cada elemento puede contener elementos que a su vez pueden contener elementos, lo que permite establecer jerarquías de contenidos modificables. Como se verá después este punto permite dos formas de parametrización de un elemento y permite incluir el contenido ya en el propio elemento si se desea. Como ya se ha establecido, un elemento debe autogestionarse; esto se logra a través de su tipo como se verá posteriormente.

Nótese que tanto los elementos como las páginas exigen scripts y hojas de estilo. Es posible declarar los requisitos solo en las páginas, pero es más útil declarar cada requisito en el punto más bajo de la jerarquía, de forma que cada elemento sea portable y reutilizable.

Esta descripción es transformada por el servidor en una respuesta en un formato ligero, en este caso JSON[24]. Se ha elegido JSON ya que se puede considerar prácticamente como un subconjunto declarativo entre Python y Javascript; el paso de estructuras de datos basadas en diccionarios y listas de Python a JSON es trivial y JSON es la notación de objetos de Javascript, por lo que se lee más rápido que XML y a la hora de localizar errores se ve la correspondencia de elementos mucho más fácilmente. Cada script, hoja de estilos, layout y elemento debe responsabilizarse de su serialización a JSON de forma que sea posible añadir nuevos elementos y todos puedan ser traducidos al mismo formato. Cada elemento, al generar su representación JSON, también puede realizar procesos propios, como es la minificación y unión de scripts u hojas de estilo; lo importante es que lo que quede reflejado en la representación JSON vaya acorde al proceso realizado.

En resumen, el servidor contiene una descripción declarativa de los elementos que conforman una página o vista que tiene los datos suficientes para que la página pueda ponerse en funcionamiento de forma autónoma, siempre buscando que cada elemento tenga un ciclo de vida autogestionado. Es posible para cada elemento que el servidor realice cierto grado de preprocesado configurable siempre y cuando se cumpla con lo que el cliente espera de él: una descripción en JSON de los elementos a cargar con una serie de elementos prefijados (algunos opcionales) y posiblemente datos personalizados que como se verá después el cliente podrá procesar gracias a un sistema de módulos. Cabe recalcar que el único límite lo impone la conversión al mensaje que el cliente procesará, y gran parte de este mensaje es personalizable para permitir que los desarrolladores puedan adaptar el sistema a sus necesidades.

Un último aspecto a tener en cuenta en el servidor es que existen diferentes dispositivos con diferentes capacidades por lo que es posible que se quieran servir diferentes descripciones de páginas (o incluso páginas clásicas) dependiendo del dispositivo que realiza la petición. Aunque esto puede ser implementado por el desarrollador para cada aplicación se provee un mecanismo de interceptación modificable que permite definir qué manejador de vista se encargará de servir a la petición.

De esta forma el cliente recibe como respuesta a su petición una cadena JSON con una jerarquía de elementos. Hay elementos fijos o conocidos (scripts, hojas de estilo y layouts) y elementos variables (elementos per sé) que cumplen una interfaz común por la cual el sistema debe poder gestionarlos. Para los elementos fijos la parte cliente de la librería ya llevará consigo todos los mecanismos necesarios para la gestión de dichos elementos mientras que para los elementos variables se implementa un sistema de módulos cargables y descargables para la construcción de elementos; entre los script exigidos por un elemento o página habrá algún script que contendrá la función o funciones que saben como utilizar el objeto extraído del mensaje JSON para el elemento. Estas funciones se registrarán mediante el tipo de elemento de forma que cuando el sistema reciba un elemento de dicho tipo delegará en ellas para su construcción e inicialización. De esta forma el sistema es totalmente extensible y permite la definición de componentes parametrizables que pueden ser declarados desde el servidor y serán gestionados de forma uniforme en el cliente.

Existe un orden lógico en la gestión de todos estos elementos; el orden lo establecen de forma natural las dependencias en la arquitectura. En la transición de una página a otra (la primera página es una transición desde una página vacía) hay que descargar los script y hojas de estilo que no se vayan a usar, cargar los script y hojas de estilo que no se tengan, descargar los elementos que ya no hacen falta y cargar en su lugar jerárquico los elementos nuevos. Entre medias de éstas operaciones también hay que llamar a las diferentes funciones designadas a modo de manejadores de evento.

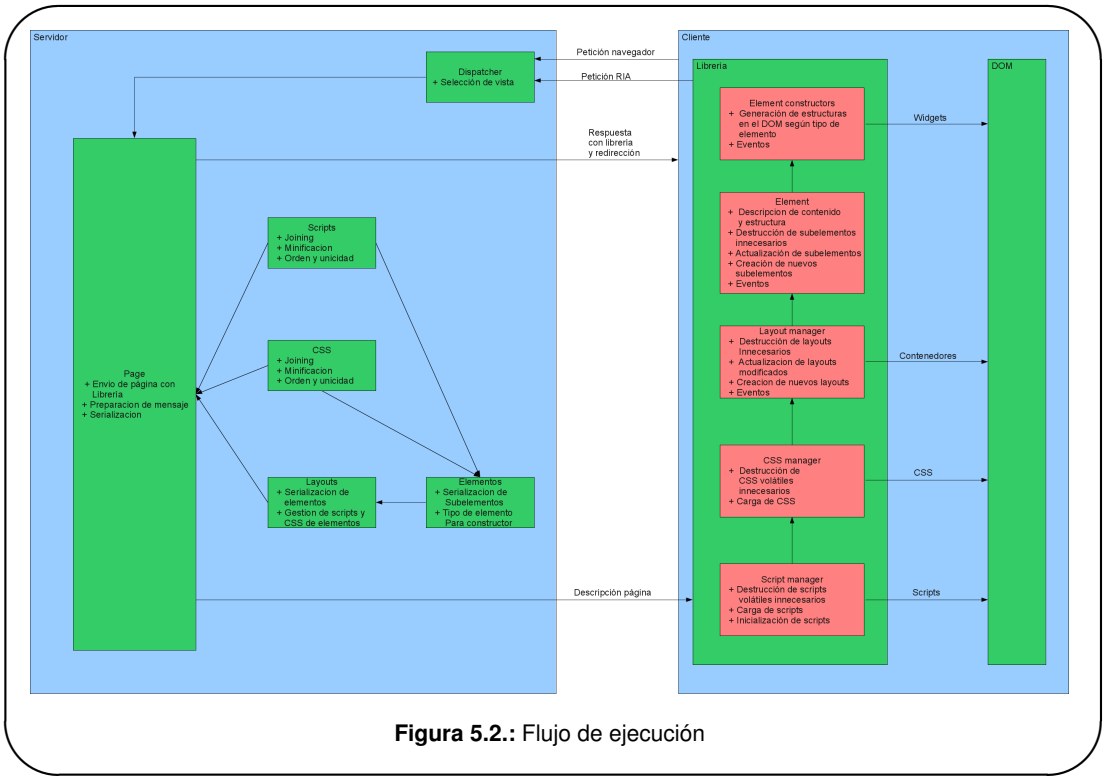

Figura 5.2.: Flujo de ejecución

En este procedimiento de carga y descargar hay que tener en cuenta el orden de los elementos y hay que asegurar que el funcionamiento sea lo más similar posible si no igual independientemente de que se anexen o minimicen los elementos. Si tenemos varios script y cada uno tiene sus procedimientos de carga y descarga es necesario crear un procedimiento de carga y otro de descarga unificado al anexarlos asegurando que dichos procedimientos invoquen a los subyacentes en el orden adecuado (orden de lista en la carga y orden inverso en la destrucción, ya que el orden suele indicar precedencia). Para que el sistema funcione de forma determinista primero se cargan todos los scripts y después se ejecutan sus funciones de inicialización en orden (de forma que si los scripts tienen código fuera de las funciones de inicialización el efecto es similar tanto para scripts anexados como para scripts independientes).

Una vez se han cargado y descargado los scripts se realiza un procedimiento similar para las hojas de estilo CSS (aunque mucho más sencillo ya que solo hay que cargar y descargar las hojas per sé). Tras esto, se procede a la creación y destrucción de elementos. La clave en este punto son los identificadores únicos de cada elemento y layout; ambos tienen un nombre único que se puede utilizar para establecer si se tiene ya un elemento en la página o no. El comportamiento a implementar difiere ligeramente de lo que puede parecer lógico por una razón: dentro de una jerarquía de elementos es posible que cambie cualquier elemento entre página y página y lo óptimo es solo cambiar los elementos que sea necesario. Para ello, es necesario determinar que elemento hay que sustituir por su identificador; en el mensaje descriptivo de la página no viene un elemento con dicho identificador y seguramente vienen uno o más elementos con identificadores que ya no existían. En dicho caso, se elimina el elemento que ya no forma parte de la página y se introducen en el nivel jerárquico los nuevos elementos.

Por ejemplo, se considera una aplicación web con una cabecera que en general tiene los mismos elementos. La cabecera en sí podría ser un layout con un identificador como “cabecera”. En algunas páginas el contenido de la cabecera cambia ligeramente. La primera aproximación que puede pensarse consiste en cambiar el identificador de la cabecera cuando esta cambia ya que se puede considerar como una cabecera distinta pero esto supondría que se eliminaría toda la cabecera y se pondría en su lugar otra, cuando es posible que aún compartan muchos elementos y estos así deben

ser regenerados. Por ello, como se ha expuesto anteriormente, lo lógico es utilizado identificadores únicos para cada elemento en la jerarquía identificando a los elementos únicamente como contenedor y no como contenido: la cabecera siempre será la cabecera y el enlace a la página de contacto siempre será el enlace a la página de contacto.

Como se verá en el capítulo de *Desarrollo*, asegurar el orden en un lenguaje con un diseño asíncrono como Javascript no es trivial, pero puede resolverse mediante el uso intensivo de callbacks.

6. DESARROLLO DEL PROYECTO

En el presente capítulo se complementará la información del capítulo anterior con detalles de desarrollo e implementación interesantes de diferentes partes del proyecto, describiendo así particularidades que quedan ocultas en el diseño o que se alejan de la propia disciplina del desarrollo de aplicaciones. Para más información relativa al desarrollo del proyecto y sus aspectos a nivel de desarrollo, ver el anexo de manual del programador.

Los epígrafes que se cubrirán en este capítulo son:

- Aspectos de desarrollo de la zona privada del portal web
  - Integración con ERP de Elson Sistemas
  - Decoradores como gestores de permisos y seguridad
  - Descargas privadas
- Aspectos de desarrollo de la librería para desarrollo de RIAs y su aplicación
  - Llamar funciones por nombre en Javascript
  - Mecanismo de transición entre páginas
  - Registro de constructores de elementos
  - Conversión a JSON de objetos Django y extensibilidad
  - Elección de vista según petición y vistas RIA
  - Adaptación de los contenidos al tamaño de la ventana

6.1 Aspectos de desarrollo de la zona privada del portal web

6.1.1 Integración con ERP de Elson Sistemas

Para la integración con el ERP de Elson Sistemas es necesaria una solución que cumpla los siguientes requisitos:

- Las transacciones deben enviarse y recibirse en el orden en que ocurren
- No se debe enviar una transacción hasta no confirmar que la anterior ha sido procesada con éxito
- El sistema debe ser tolerante a que el otro lado de la comunicación no esté presente al momento

La aplicación web funciona ante las peticiones que llegan; el servidor web crea un número variable de procesos preparados para la recepción y procesado de dichas peticiones y lo lógico es que el mecanismo de envío sea independiente para asegurar su funcionamiento y no obligar al servidor a mantener procesos vivos para el envío de la información. Por ello la solución implementada es un proceso externo escrito en Python que comprueba un directorio de mensajes y se encarga de enviarlos.

```
import time
import os
import urllib
import urllib2
```

```
import httplib
import sys

SLEEP_SECONDS = 2
QUEUE_DIR = os.path.join(os.path.dirname(os.path.realpath( __file__ )),"queue")
USERNAME = 'Prueba'
PASSWORD = 'Prueba'
URL = "https://urlElsonSistemas:8090/transacciones"

def checkAndSend():
    messages=os.listdir(QUEUE_DIR)
    messages.sort(key=lambda s: os.path.getmtime(os.path.join(QUEUE_DIR, s)))
    messages.reverse()
    for message in messages:
        m = open(os.path.join(QUEUE_DIR,message), 'r').read()

        password_mgr = urllib2.HTTPPasswordMgrWithDefaultRealm()
        password_mgr.add_password(None, URL, USERNAME, PASSWORD)
        handler = urllib2.HTTPBasicAuthHandler(password_mgr)
        opener = urllib2.build_opener(handler)
        data = urllib.urlencode({'message' : m})
        req = urllib2.Request(URL,data)
        f = opener.open(req)

        if f.read() == "OK":
            print "Send OK!"
            os.remove(os.path.join(QUEUE_DIR,message))
        else:
            print "Send ERROR!"


if __name__ == '__main__':
    while(True):
        time.sleep(SLEEP_SECONDS)
        checkAndSend()
```

Listado 6.1: Demonio de envío de transacciones

Este demonio obtiene listados de ficheros de un directorio llamado queue, los ordena por tiempo de modificación y se encarga de enviarlos uno por uno utilizando la librería urllib2 de Python utilizando autenticación HTTP básica. Cuando logra enviar un mensaje lo borra de disco y accede al siguiente. Lo bueno de este sistema es que cumple todos los requisitos con una implementación sumamente sencilla. A la hora de utilizar este sistema basta con crear un fichero con el mensaje a enviar al otro lado.

```
import random, shutil, codecs, os

def generate_random_token():
    alphabet = 'abcdefghijklmnopqrstuvwxyz'
    min = 5
    max = 15
    string=''
    for x in random.sample(alphabet,random.randint(min,max)):
        string+=x
    return string

def message_to_file(message,username):
    tmp = gettempdir()
    queue = os.path.join(settings.SITE_ROOT,"communicator/queue")
    name = username + generate_random_token() + ".msg"
    while os.path.isfile(os.path.join(queue,name)):
```

```
        name = username + generate_random_token() + ".msg"
    f = codecs.open(os.path.join(tmp,name), 'w',"utf-8")
    f.write(message)
    f.close()
    shutil.move(os.path.join(tmp,name),os.path.join(queue,name))
```

Listado 6.2: Preparación de mensajes desde Django

La función *message_to_file* se encarga de la creación de un fichero con el mensaje como contenido, asegurándose de que no existe ya un fichero con el mismo nombre (generado al azar y utilizando el nombre del usuario que origina la transacción, lo que garantiza que no es posible que existan coincidencias). Para evitar que el proceso que se encarga del envío intente abrir el fichero mientras éste se escribe se crea dicho fichero en el directorio de ficheros temporales del sistema y luego se mueve, garantizando su bloqueo.

Por último, hace falta una vista dedicada a la recepción y procesado de mensajes.

```
from django.http import HttpResponse
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt
def jsonservices(request):
    if request.method == 'POST':
        if request.POST.has_key('message'):
            try:
                process_json_message(request.POST['message'])
                return HttpResponse("OK")
            except Exception as ex:
                return HttpResponse("FAIL")
    return HttpResponse("FAIL")
```

Listado 6.3: Recepción de mensajes en Django

Esta función necesita una función llamada *process_json_message* que se encargue del procesado de los diferentes mensajes definidos por Elson Sistemas.

6.1.2 Decoradores como gestores de permisos y seguridad

Hay vistas que requieren haber iniciado sesión, algún permiso especial o bien que se acceda a ellas mediante HTTPS en lugar de HTTP. Django provee un decorador Python llamado *@login_required* que se encarga de comprobar si la petición está asociada a un usuario que ha iniciado sesión y si no redirige la petición a una página. La idea es implementar decoradores similares que sirvan como filtro para los permisos dedicados a peticiones de oferta, ofertas y pedidos y para el acceso a secciones seguros con HTTPS.

```
from django.conf import settings
from django.http import HttpResponseRedirect

def secure_required(view_func):
    def _wrapped_view_func(request, *args, **kwargs):
        if settings.DEBUG:
            return view_func(request, *args, **kwargs)
        if not request.is_secure():
            if getattr(settings, 'HTTPS_SUPPORT', True):
                request_url = request.build_absolute_uri(request.get_full_path())
                secure_url = request_url.replace('http://', 'https://')
                return HttpResponseRedirect(secure_url)
        return view_func(request, *args, **kwargs)
    return _wrapped_view_func
```

```
def permissions_compra_required(view):
    def _wrapped_view_func(request,*args,**kwargs):
        if request.user.get_profile().compra_activada:
            return view(request,*args,**kwargs)
        else:
            return HttpResponseRedirect("/index")
    return _wrapped_view_func

def permissions_oferta_required(view):
    def _wrapped_view_func(request,*args,**kwargs):
        if request.user.get_profile().oferta_activada:
            return view(request,*args,**kwargs)
        else:
            return HttpResponseRedirect("/index")
    return _wrapped_view_func

def permissions_pedido_required(view):
    def _wrapped_view_func(request,*args,**kwargs):
        if request.user.get_profile().pedido_activado:
            return view(request,*args,**kwargs)
        else:
            return HttpResponseRedirect("/index")
    return _wrapped_view_func
```

Listado 6.4: Decoradores como filtros de vistas

6.1.3 Descargas privadas

Existen recursos que solo pueden ser descargados por un usuario que ha iniciado sesión. El problema es que los ficheros los sirve el servidor web sin hablar con la aplicación web en ningún momento y la autenticación no es compartida, por lo que el servidor web no sabe nada de la autenticación. Obviamente es necesario que se encargue de servir el fichero la propia aplicación web, pero el consumo de memoria y de procesador es mucho más elevado que si lo hace el servidor web, por lo que sería una solución inapropiada.

Como solución se ha aprovechado un módulo del servidor web para que al situar una respuesta especial en la cabecera de la respuesta generada se encargue de enviar el fichero en dicha respuesta. Este módulo[48] debe ser compilado para la versión de Apache instalada y se encarga de servir ficheros en un punto según su configuración con el parámetro XSendFilePath. Para completar, desde Django se provee una vista para todas las URLs que pasen por un directorio de descargas privadas.

```
import os
from django.contrib.auth.decorators import login_required
from django.http import HttpResponse
from django.http import Http404

PRIVATE_DOWNLOADS_PATH="/path/"

@login_required
def private_download(request,file_name):
    f = PRIVATE_DOWNLOADS_PATH + file_name
    if os.path.isfile(f):
        response = HttpResponse()
        response['Content-Disposition']='attachment;filename="%s"' % file_name
        response["X-Sendfile"] = f
        response['Content-type'] = "application/force-download"
        return response
    raise Http404
```

Listado 6.5: Vista para descargas privadas

6.2 Aspectos de desarrollo de la librería para desarrollo de RIAs y su aplicación

6.2.1 Llamar funciones por nombre en Javascript

La librería permite establecer de forma declarativa funciones javascript que se encargan de diferentes acciones en el ciclo de vida de los elementos gestionados por la librería. Esta declaración utiliza el nombre de la función (incluyendo jerarquía de objetos si dicha función forma parte de un objeto) para que luego la parte javascript pueda llamar a dicha función. Hace falta por tanto un mecanismo que permita asegurarse de que existe una función dado su nombre, un mecanismo para llamarla y en algunos casos, un mecanismo para crear una función dinámicamente de forma que el desarrollador vea el objeto de la misma forma en ambos lados de la arquitectura.

```
function function_exists (function_name) {
    if (typeof function_name != "string") {
        return false;
    }

    var namespaces = function_name.split(".");
    var func = namespaces.pop();
    var context = window;

    for(var i = 0; i < namespaces.length; i++) {
        context = context[namespaces[i]];
    }

    if (context == undefined) {
        return false;
    } else {
        return (typeof context[func] == "function");
    }
}
```

Listado 6.6: Comprobación de si una función existe por su nombre jerárquico

En Javascript todos los objetos, incluyendo funciones, parten del objeto *window*. De esta forma, si el nombre de función es una función directamente ésta estará situada directamente en el objeto *window*. Si en cambio el nombre es una jerarquía de objetos o espacios de nombre (por ejemplo *ria_core.layout_manager.load_layout*) el primer punto de la jerarquía estará bajo el objeto *window* y la función deberá avanzar a través de la jerarquía.

```
function execute_function_by_name (function_name) {
    var args = Array.prototype.slice.call(arguments).splice(1);
    var namespaces = function_name.split(".");
    var func = namespaces.pop();
    var context = window;

    for(var i = 0; i < namespaces.length; i++) {
        context = context[namespaces[i]];
    }

    return context[func].apply(this, args);
}
```

Listado 6.7: Función para invocar una función por su nombre

La forma de llegar a la función es exactamente la misma que en la función anterior. La particularidad reside en que es posible que se quiera llamar a dicha función con parámetros. Sería posible pasar los parámetros a utilizar como una lista o algo similar, pero eso implica preparar un objeto adicional. La

solución implementada se aprovecha de que se pueden enviar tantos parámetros como se desee a cualquier función javascript; la función mapea dichos parámetros a su lista de parámetros a recibir y el resto quedan ocultos, pero es posible acceder a todos los parámetros como una lista con el objeto *arguments*. De esta forma se puede aislar el nombre de función que llega como primer parámetro y aplicar el resto de parámetros de forma natural.

```
create_function = function (object_function_name,function_name,function_container) {
    if (function_exists(function_name)) {
        function_container[object_function_name] = function () {
            var args = Array.prototype.slice.call(arguments);
            execute_function_by_name.apply(this,[function_name,].concat(args));
        }
    }
}
```

Listado 6.8: Función para invocar una función por su nombre

Por último este sistema utiliza las dos funciones anteriores para creación de funciones en objetos. Un ejemplo claro son los elementos; considérese que el servidor declara un elemento con un atributo llamado *show* cuyo valor es el nombre de una función javascript. Este sistema permitiría que dicho elemento, al ser construido en Javascript, pueda tener una función show que llame implícitamente a la función designada, de forma que se pueden crear objetos complejos con interfaces bien definidas dinámicamente.

6.2.2 Mecanismo de transición entre páginas

La transición entre páginas sucede como resultado de una petición lanzada desde la librería (una petición POST con al menos un dato que designa que la librería está funcionando en el lado cliente). Cuando llega la respuesta que representa la nueva página la librería pone en marcha su mecanismo de transición.

```
this.process_json_message = function (message) {
    self.script_manager.load_and_init(message.javascript, function () {
        self.css_manager.load(message.css);
        self.layout_manager.load_layouts(message.layouts);
    });
};
```

Listado 6.9: Función para invocar una función por su nombre

Este método, situado en el objeto *ria_core*, utiliza diferentes gestores para la carga y descarga de los recursos y elementos. Nótese que primero lanza la carga, descarga e inicialización de los scripts y le pasa como callback una función para cargar el resto de elementos. Esto se hace así para garantizar que primero se realice todo el proceso de carga y descarga de scripts ya que estos son vitales para el funcionamiento del sistema. Las hojas de estilo CSS y la carga de elementos se realiza de forma simultanea para aprovechar el tiempo de proceso de elementos para la descarga de las hojas de estilo. Nótese también que cada cargador recibe la lista de los elementos y recursos que debe cargar; en su interior estos cargadores gestionan una lista de los elementos y recursos activos y mediante el cruce de ambos son capaces de determinar qué deben descargar y qué deben cargar.

■ Gestión de scripts

El primer paso en la transición es la gestión de scripts.

```
this.load_and_init = function (scripts_json, callback) {
    //Me quedo con el callback para llamarlo posteriormente
    if (callback) {
        self.callback = callback;
```

```
    } else {
        self.callback = null;
    }

    //Por cada script de la nueva pagina...
    for (var j = 0; j < scripts_json.length; j += 1) {
        var script = scripts_json[j], found = false;
        //Miro a ver si ya esta cargado y en dicho caso si lo
        //tengo que volver a descargar o inicializar
        for (var i = self.loaded.length - 1; i >= 0; i -= 1) {
            var l_script = self.loaded[i];

            if (script.id === l_script.id) {
                found = true;
                if (l_script.lifetime !== "static") {
                    if (l_script.should_redownload || l_script.should_reinitialize) {
                        //Si lo tengo que descargar o inicializar, seguro que tengo
                        //que inicializarlo, asi que lo meto a la lista para
                        //inicializar. Lo hago en el indice que le corresponde para que
                        //la inicializacion siga el orden.
                        self.to_init[j] = l_script;
                        if (l_script.should_redownload) {
                            //Si ademas lo tengo que bajar lo destruyo y lo meto a la lista
                            //de scripts a cargar en el indice que le corresponde
                            l_script.destroy();
                            self.to_load.push(l_script);
                        }
                        //Si hay que reinicializar o descargar lo quito de la lista de
                        //cargados ya que lo metere posteriormente
                        self.loaded.splice(i, 1);
                    }
                }
                //Si ya hemos encontrado el elemento en la lista de cargados
                //dejamos de recorrerla
                break;
            }
        }

        if (!found) {
            //Si no hemos encontrado el script lo creamos y lo metemos a las listas
            //de carga e inicializacion en el sitio que corresponde
            var sc = new Script(scripts_json[j]);
            self.to_load.push(sc);
            self.to_init[j] = sc;
        }
    }

    for (var i = self.loaded.length - 1; i >= 0; i -= 1) {
        //Los que queden en la lista de cargados son scripts antiguos
        //que hay que destruir si son volatiles
        if (self.loaded[i].lifetime === "volatile") {
            self.loaded[i].destroy();
            self.loaded.splice(i, 1);
        }
    }

    //Procedemos a la descarga e inicializacion de los scripts que hemos
    //preparado
    self.load_scripts();
};
```

Listado 6.10: Carga y descarga de scripts - Preparación

La carga de scripts se divide en tres fases: descarga, destrucción e inicialización. La inicialización es el punto más complejo ya que debe seguir el orden establecido por la lista de scripts necesarios para el funcionamiento para la página; de esta forma, si un script debe volverse a descargar o reinicializar (ya está cargado e inicializado) hay que asegurarse de mantener el orden en la inicialización.

Hay que asegurarse por otro lado de mantener la secuencia de acciones. Como se puede ver la función anterior se encarga de preparar listas de descarga e inicialización de scripts y de destruir los scripts que ya no son válidos. La descarga en si y la inicialización se realizan en el método *load_scripts*.

```
this.init_loaded = function () {
       while (self.to_init.length > 0) {
           var script = self.to_init.shift();
           if (script) {
               if (script.init) {
                   script.init();
               }
               self.loaded.push(script);
           }
       }
       if (self.callback) {
           self.callback();
       }
   };

   this.load_scripts = function () {
       if (self.to_load.length === 0) {
           self.init_loaded();
       } else {
           var script = self.to_load.shift();
           script.load(self.load_scripts);
       }
   };
```

Listado 6.11: Carga y descarga de scripts - Carga e inicialización de scripts

Primero es necesaria la descarga de todos los scripts y después la inicialización. En el array *to_load* hay un número variable de objetos de tipo Script (ver más adelante) que, siguiendo el diseño basado en autogestión comentado en el capítulo anterior, son capaces de realizar su propia carga. Dado que la carga es asíncrona y es posible que se ejecute código del script durante su carga hay que garantizar que no se va a cargar un script hasta que se termine de cargar el que le precede. Para ello se proporciona un callback al proceso de carga para que se vuelva a la misma función y se proceda con el siguiente.

El método *load_scripts* es en cierto modo recursivo; cuando termina la carga de un script vuelve a llamarse a este método que va quitando elementos de la lista de carga. Cuando esta lista queda vacía se procede a llamar al método de inicialización y se sale de la función recursiva.

El método de inicialización delega una vez más en los objetos de tipo Script para la inicialización, siempre siguiendo el orden, y al terminar llama al callback designado si lo había (en el uso normal de la librería siempre ya que tras terminar la gestión de los Script se realiza la gestión de hojas de estilo y layouts).

```
function Script(script_json) {
   this.id = script_json.id;
   this.path = script_json.path;
   this.lifetime = script_json.lifetime;
   this.status = "missing";
   var self = this;
```

```
this.init = function () {
    if (script_json.hasOwnProperty("init_func")) {
        execute_function_by_name(script_json.init_func);
    }
    self.status = "initialized";
};


this.unload = function () {
    if (script_json.hasOwnProperty("unload_func")) {
        execute_function_by_name(script_json.unload_func);
    }
    self.status = "unloaded";
};

this.should_reinitialize = false;
if (script_json.hasOwnProperty("should_reinitialize")) {
    this.should_reinitialize = script_json.should_reinitialize;
}

this.should_redownload = false;
if (script_json.hasOwnProperty("should_redownload")) {
    this.should_redownload = script_json.should_redownload;
}

this.load = function (callback) {
    if (!document.getElementById(self.id)) {
        var head = document.getElementsByTagName("head")[0];
        var script = document.createElement("script");
        script.type = "text/javascript";
        script.id = self.id;

        if (script.readyState) {
            script.onreadystatechange = function () {
                if (script.readyState === "loaded" || script.readyState === "complete") {
                    script.onreadystatechange = null;
                    self.status = "loaded";
                    callback();
                }
            };
        } else {
            script.onload = function () {
                self.status = "loaded";
                callback();
            };
        }
        script.src = self.path;
        head.appendChild(script);
    }
};

this.destroy = function () {
    if (self.hasOwnProperty("unload")) {
        self.unload();
    }
    var scr = document.getElementsByTagName("script");
    for (var i = 0; i < scr.length; i+=1) {
        if (typeof scr[i].id != "undefined" && scr[i].id === self.id) {
            scr[i].parentNode.removeChild(scr[i]);
            i-=1;
```

```
            }
        }
        self.status = "missing";
    };

    this.reload = function () {
        if (document.getElementById(self.id)) {
            self.destroy();
            self.load();
        }
    };
}
```

Listado 6.12: La clase Script

Como ya se ha mencionado, la clase Script es capaz de autogestionarse y por tanto lleva consigo los métodos de carga, descarga,... Nótese que no se ha utilizado en este caso el generador dinámico de funciones antes explicado ya que en este caso son siempre las mismas funciones y no se desean añadir más a priori. Nótese también que se considera un script descargado cuando se elimina el elemento que lo referencia de la cabecera del documento; a día de hoy esto no garantiza que se destruya el script de verdad, por lo que es recomendable enmascarar toda la funcionalidad de un script en un objeto y desasignar dicho objeto en la función de destrucción, de forma que se garantice que ya no se tendrá acceso a sus funcionalidades. El sentido común indica en cualquier caso que si en algún momento se considera la necesidad de descargar scripts del navegador para ahorrar memoria su extracción de la cabecera será la forma de hacerlo.

■ Gestión de hojas de estilo CSS

La gestión de hojas de estilo CSS es similar a la de scripts quitando bastante parte de su complejidad (no existe inicialización ni destrucción particular ni tienen diferentes modos de carga) por lo que lo visto en el epígrafe anterior es válido para el caso de los CSS. Para más información conviene consultar el manual de programador o el propio código fuente de la librería.

■ Gestión de layouts y elementos

Al igual que en los dos casos anteriores, se comienza con la gestión de layouts y después es cada layout el que gestiona sus elementos.

```
this.load_layouts = function(layouts_json) {
    to_delete = []
    //Por cada layout cargado miro si viene alguno
    //con su mismo id. Si no es el caso, lo
    //descargo y lo meto en la lista a borrar
    for (index in self.current_layouts) {
        var found = false;

        for (i in layouts_json) {
            if (layouts_json[i].id === index) {
                found = true;
            }
        }

        if (!found) {
            self.current_layouts[index].unload();
            to_delete.push(index);
        }
    }
```

```
    //Borro los layouts que ya no se usan
    for (i in to_delete) {
        delete self.current_layouts[to_delete[i]];
    }

    //Por cada layout de la nueva pagina creo un objeto
    //para representarlo y lo cargo tomando como referencia
    //el que ya exista con el mismo id, si lo hay. Despues
    //borro el antiguo objeto de layout y meto el nuevo
    for (index in layouts_json) {
        var new_layout = new Layout(layouts_json[index]);
        var old_layout = null;

        if (self.current_layouts.hasOwnProperty(new_layout.id)) {
            old_layout = self.current_layouts[new_layout.id];
        }

        new_layout.load(old_layout);
        delete self.current_layouts[new_layout.id];
        self.current_layouts[new_layout.id] = new_layout;
    }
};
```

Listado 6.13: Carga y descarga de layouts

La preparación en la carga de layouts es razonablemente similar a lo visto anteriormente, salvo que en este caso que un layout ya exista no implica que sus contenidos vayan a ser los mismos. De esta forma, es necesario procesar todos los layouts de entrada siguiendo la filosofía de autogestión ya comentada anteriormente.

```
function Layout(layout_json) {
    this.id = layout_json.id;
    this.elements = {};
    this.document_element = undefined;
    var self = this;

    this.preload_callback = function () {
        if (layout_json.hasOwnProperty("preload_callback")) {
            execute_function_by_name(layout_json.preload_callback);
        }
    };

    this.postload_callback = function () {
        if (layout_json.hasOwnProperty("postload_callback")) {
            execute_function_by_name(layout_json.postload_callback);
        }
    };

    this.load = function (reference_layout) {
        self.preload_callback();
        //Si no existe un layout con el mismo id,
        //creamos un div oculto, cargamos los elementos
        //del layout y los insertamos dentro del div.
        //Despues mostramos el div
        if (reference_layout == null) {
            var layout = document.createElement("div");

            self.document_element = layout;
            layout.id = self.id;
            self.hide();
            layout.style.position = "absolute";
```

```
            document.getElementsByTagName("body")[0].appendChild(layout);

            for (index in layout_json.elements) {
                var element = new Element(layout_json.elements[index]);
                element.load();
                layout.appendChild(element.document_element);
                self.elements[element.id] = element;
            }
            self.show();
        }
        //Si existe, tomamos el elemento del documento original y
        //creamos estructuras para los elementos del layout nuevo.
        //Por cada elemento del layout antiguo miramos si esta presente
        //en la lista del nuevo. Si no lo esta lo destruimos, y si lo esta
        //lo mandamos procesar ya que pueden tener subelementos diferentes.
        //Por ultimo, los elementos nuevos que se han quedado sin cargar
        //se mandan procesar sin elemento de referencia.
        else
        {
            self.document_element = reference_layout.document_element;

            for (index in layout_json.elements) {
                var element = new Element(layout_json.elements[index]);
                self.elements[element.id] = element;
            }

            for (key in reference_layout.elements) {
                if (!self.elements.hasOwnProperty(key)) {
                    reference_layout.elements[key].unload();
                } else {
                    self.elements[key].load(reference_layout.elements[key]);
                }
            }

            for (key in self.elements) {
                if (self.elements[key].status !== "loaded") {
                    self.elements[key].load(null);
                }
            }
        }
        self.postload_callback();
    };

    this.show = function () {
        self.document_element.style.opacity = 1.0;
    };

    this.hide = function () {
        self.document_element.style.opacity = 0;
    };

    this.unload = function () {
        self.document_element.parentNode.removeChild(self.document_element);
    };
}
```

Listado 6.14: La clase Layout

La gestión de elementos se realiza de forma similar a la gestión de layouts.

```
function Element(element_json) {
    this.id = element_json.id;
```

```
    this.type = element_json.type;
    this.innerElements = {};
    this.status = "not loaded";
    var self = this;

    //Mapeamos propiedades del objeto JSON al propio objeto.
    //Si una propiedad es una funcion se genera dinamicamente
    for (property in element_json) {
        if (function_exists(element_json[property])) {
            create_function(property,element_json[property],this);
        } else {
            this[property] = element_json[property];
        }
    }

    this.document_element = undefined;

    this.load = function (reference_element) {
        //Si no hay elemento de referencia hay que crear el elemento
        if (reference_element == null) {
            //Obtenemos el constructor para el elemento por su tipo
            //lo utilizamos para crear el elemento y despues cargamos
            //los subelementos de la misma forma, introduciendolos
            //en el elemento del DOM y en una lista
            var loader = ria_core.layout_manager.element_loaders[self.type];

            if (loader != undefined) {
                loader.create_element(self);
                if (typeof self.load_callback == "function") {
                    self.load_callback(element);
                }
                self.status = "loaded";
                for (index in element_json.elements) {
                    var innerElement = new Element(element_json.elements[index]);
                    innerElement.load(null);
                    self.document_element.appendChild(innerElement.document_element);
                    self.innerElements[innerElement.id] = innerElement;
                }
            }
            //Nos aseguramos de que el elemento sea visible
            self.document_element.style.opacity = 1;
        }
        //Si existe, tomamos el elemento del documento original y
        //creamos estructuras para los subelementos del elemento nuevo.
        //Por cada subelemento del elemento antiguo miramos si esta presente
        //en la lista del nuevo. Si no lo esta lo destruimos, y si lo esta
        //lo mandamos procesar ya que puede tener subelementos diferentes.
        //Por ultimo, los subelementos nuevos que se han quedado sin cargar
        //se mandan procesar sin elemento de referencia.
        else
        {
            self.document_element = reference_element.document_element;
            self.status = "loaded";

            for (index in element_json.elements) {
                var innerElement = new Element(element_json.elements[index]);
                self.innerElements[innerElement.id] = innerElement;
            }

            for (key in reference_element.innerElements) {
                if (!self.innerElements.hasOwnProperty(key)) {
```

```
                reference_element.innerElements[key].unload();
            } else {
                self.innerElements[key].load(reference_element.innerElements[key]);
            }
        }

        for (key in self.innerElements) {
            if (self.innerElements[key].status !== "loaded") {
                self.innerElements[key].load(null);
                self.document_element.appendChild(self.innerElements[key].document_element);
            }
        }
    }
}

this.unload = function () {
    self.document_element.parentNode.removeChild(self.document_element);
    self.status = "not loaded";
}
}
```

Listado 6.15: La clase Element

Nótese que aunque podría deducirse del código que no tiene sentido el uso de layouts (podrían utilizarse elementos directamente ya que la gestión de los unos y los otros es igual) se ha decidido la separación para permitir la interceptación previa y posterior a la creación de un grupo de objetos y ya que en el futuro se plantea la posibilidad de que el usuario pueda dejar contenedores para layouts ya preparados en la plantilla inicial del sistema, con lo que estos se situarían en estos contenedores en lugar de crear contenedores nuevos.

6.2.3 Registro de constructores de elementos

Una de las características más importantes de la librería es el soporte para que los desarrolladores creen elementos personalizados, maximizando así la reutilización de componentes y facilitando su trabajo. Dado que la información de la página que reside en el servidor y viaja al cliente es declarativa hace falta alguna forma de asociar información procedural para la creación de elementos.

La solución implementada se basa en las dependencias de scripts declaradas; un elemento declara su tipo y declara los scripts que necesita para funcionar, estando entre estos habitualmente un script con la funcionalidad necesaria para la construcción del elemento. Cuando dicho script se carga e inicializa dicho script debe responsabilizarse de registrar el constructor del elemento en el gestor de layouts.

```
function assign_classes(element) {
    for (index in element.classes) {
        element.document_element.className = element.document_element.className +
        " " + element.classes[index];
    }
}

function ImageLoader() {
    this.type = "image";
    this.create_element = function (element) {
        element.document_element = document.createElement("img");
        element.document_element.id = element.id;
        element.document_element.src = element.src;
        assign_classes(element);
    }
}
ria_core.layout_manager.register_element_loader(new ImageLoader());
```

```
function LinkLoader() {
    this.type = "link";
    this.create_element = function (element) {
        element.document_element = document.createElement("a");
        element.document_element.id = element.id;
        element.document_element.href = element.href;
        element.document_element.innerHTML = element.text;
        if (element.hasOwnProperty("on_click")) {
            element.document_element.onclick = function() {
                if (typeof element.on_click == "function")
                    element.on_click();
            }
        }
        assign_classes(element);
    }
}
ria_core.layout_manager.register_element_loader(new LinkLoader());

function DivLoader() {
    this.type = "div";
    this.create_element = function (element) {
        element.document_element = document.createElement("div");
        element.document_element.id = element.id;
        assign_classes(element);
    }
}
ria_core.layout_manager.register_element_loader(new DivLoader());
```

Listado 6.16: Constructores de elementos

Como puede verse un constructor es una pequeña clase que debe declarar su tipo y proveer un método llamado *create_element* que dado un objeto de tipo Element se encargue de la creación del elemento DOM correspondiente. Esto permite dos formas de crear elementos complejos: es posible que el elemento incluya como atributos todo lo que necesita y el constructor cree la jerarquía completa a partir de dicha información o bien realizar la jerarquía mediante subelementos que se construirán por si mismos. El método a utilizar dependerá de las preferencias del desarrollador y la situación.

6.2.4 Conversión a JSON de objetos Django y extensibilidad

En el lado servidor cada objeto que vaya a ser transferido al cliente debe acabar siendo serializado en formato JSON. En lugar de serializar directamente cada objeto se ha optado por un enfoque diferente; los objetos debe serializarse a un elemento de un diccionario compuesto por datos de tipos simples (básicamente lo que soporta JSON de forma nativa).

De esta forma, cada objeto proporciona o hereda un método llamado *to_context_map* que tiene la responsabilidad de devolver una representación en forma de diccionario del objeto en cuestión. Esto es especialmente útil en el caso de los script y las hojas de estilo CSS ya que en el propio proceso de serialización se pueden minificar o unir, con lo que finalmente cambia solo algún dato del diccionario y esto es totalmente transparente para el cliente.

Por otro lado, las clases *Layout* y *Element* están pensadas para servir como base para todo tipo de componentes personalizados. Basta con heredar de ellas, añadir lo que se desee y asegurarse de implementar la serialización a diccionario para reflejar los elementos añadidos. Los añadidos pueden incluir por ejemplo una serie de scripts y hojas de estilo ya preincluidos, con lo que es posible crear componentes reutilizables fácilmente.

```
class Layout:
    def __init__(self,id,preload_callback=None,postload_callback=None,elements = []):
```

```
        self.id = id
        self.preload_callback = preload_callback
        self.postload_callback = postload_callback
        self.elements = elements

    def __hash__(self):
        return self.id.__hash__()

    def __eq__(self, obj):
        return obj.id == self.id

    def get_required_js(self):
        required_js = []
        for element in self.elements:
            r = element.get_required_js()
            for script in r:
                if not script in required_js:
                    required_js.append(script)
        return required_js

    def get_required_css(self):
        required_css = []
        for element in self.elements:
            r = element.get_required_css()
            for css in r:
                if not css in required_css:
                    required_css.append(css)
        return required_css

    def to_context_map(self):
        context = {"id":self.id,}
        if self.preload_callback:
            context["preload_callback"] = self.preload_callback
        if self.postload_callback:
            context["postload_callback"] = self.postload_callback
        if self.elements:
            context["elements"] = []
            for element in self.elements:
                context["elements"].append(element.to_context_map())
        return context

class Element:
    def __init__(self,id,type,required_js = [], required_css = [], elements = [],load_callback=None):
        self.id = id
        self.type = type
        self.required_js = required_js
        self.required_css = required_css
        self.elements = elements
        self.load_callback = load_callback

    def __hash__(self):
        return self.id.__hash__()

    def __eq__(self, obj):
        return obj.id == self.id and obj.type == self.type

    def get_required_js(self):
        required_js = []
        for element in self.elements:
            r = element.get_required_js()
            for script in r:
```

```
                if not script in required_js:
                    required_js.append(script)
        for script in self.required_js:
            if not script in required_js:
                required_js.append(script)
        return required_js

    def get_required_css(self):
        r_css = self.required_css
        for element in self.elements:
            r = element.get_required_css()
            for css in r:
                if not css in r_css:
                    r_css.append(css)
        for css in self.required_css:
            if not css in r_css:
                r_css.append(css)
        return r_css

    def to_context_map(self):
        context = {"id":self.id,"type":self.type,}
        if self.elements:
            context["elements"] = []
            for element in self.elements:
                context["elements"].append(element.to_context_map())
        if self.load_callback:
            context["load_callback"] = self.load_callback
        return context
```

Listado 6.17: Layout y Element en Django

Dado que ambas clases pueden contener subelementos incluyen métodos para recoger todas las dependencias de scripts y CSS subyacentes, de forma que todas terminan formando parte de la lista de scripts necesarios para la página que contiene dichos elementos (filtrando repeticiones obviamente).

```
class StyledElement(Element):
    def __init__(self,id,type,classes=[],required_js=[],required_css=[],elements=[],
                 load_callback=None):
        Element.__init__(self, id, type, required_js, required_css, elements, load_callback)
        self.classes = classes

    def to_context_map(self):
        context = Element.to_context_map(self)
        if self.classes:
            context["classes"] = self.classes
        return context

class Div(StyledElement):
    def __init__(self,id,classes=[],elements=[],load_callback=None):
        StyledElement.__init__(self,id,"div",classes,[],[],elements, load_callback)

class Image(StyledElement):
    def __init__(self,id,src,classes=[],load_callback=None):
        StyledElement.__init__(self, id, "image",classes, [], [], [], load_callback)
        self.src = src

    def to_context_map(self):
        context = StyledElement.to_context_map(self)
        context["src"] = self.src
        return context
```

```
class TextLink(StyledElement):
    def __init__(self,id,text,href,on_click_callback="",classes=[],load_callback=None):
        StyledElement.__init__(self, id, "link",classes, [], [], [], load_callback)
        self.text = text
        self.href = href
        self.on_click_callback = on_click_callback

    def to_context_map(self):
        context = StyledElement.to_context_map(self)
        context["text"] = self.text
        context["href"] = self.href
        context["on_click"] = self.on_click_callback
        return context

class Columna(Layout):
    def __init__(self,id,preload_callback=None,postload_callback=None):
        Layout.__init__(self,id,preload_callback,postload_callback,[])
        self.elements = [
            Div(id="contenido_columna",elements=[
                Image(id="asterisco",src="/media/images/columna/asterisco.gif"),
                TextLink(id="a_contacto",href="#",text="CONTACTO",
                        on_click_callback="contacto_clicked"),
                TextLink(id="a_nota_legal",href="#",text="NOTA LEGAL",
                        on_click_callback="nota_legal_clicked"),
                TextLink(id="a_mapa_web",href="#",text="MAPA WEB",
                        on_click_callback="mapa_web_clicked"),
            ]),
        ]

class Image_Cycle(StyledElement):
    def __init__(self,id,images,width="100%",height="100%",load_callback=None):
        self.images = images
        self.width = width
        self.height = height
        StyledElement.__init__(self,id,type="image_cycle",classes=["slideshow",],
            required_js=[
                Script(id="jquery",path="media/js/jquery.js",lifetime="static"),
                Script(id="jquery_easing",path="media/js/easing.js",lifetime="volatile"),
                Script(id="jquery_cycle",path="media/js/cycle.js",lifetime="volatile"),
                Script(id="js_cycle_ria",path="media/js/cycle_ria.js",lifetime="static"),
            ],
            required_css=[],
            elements=[],
            load_callback=load_callback
        )

    def to_context_map(self):
        context = StyledElement.to_context_map(self)
        context["images"] = self.images
        context["width"] = self.width
        context["height"] = self.height
        return context
```

Listado 6.18: Heredando de Layout y Element

Como puede verse en el ejemplo anterior es posible crear todo tipo de componentes y facilidades simplemente estableciendo una pequeña jerarquía de clases. Para más detalle sobre el desarrollo de componentes personalizados ver el anexo *Manual del desarrollador*.

6.2.5 **Elección de vista según petición y vistas RIA**

Al llegar una petición a Django hay ciertos aspectos a tener en cuenta: Según la procedencia de la petición es posible que se desee servir contenido diferente (versión de escritorio y versión móvil, por ejemplo) y si la petición no viene marcada como que procede de la librería Javascript es necesario responder primero con un documento HTML que provea la librería.

```
class PageDispatcher(View):
    def get(self, request, *args, **kwargs):
        return self.device_dispatch(request)

    def post(self, request, *args, **kwargs):
        return self.device_dispatch(request)

    def device_dispatch(self,request):
        import re

        if request.META.has_key('HTTP_USER_AGENT'):
            user_agent = request.META['HTTP_USER_AGENT']

            # Test common mobile values.
            pattern = """(up.browser|up.link|mmp|symbian|smartphone|midp|wap|phone|
                       windows ce|pda|mobile|mini|palm|netfront|android|iphone)"""
            prog = re.compile(pattern, re.IGNORECASE)
            match = prog.search(user_agent)

            if match:
                return self.modern_mobile(request)
            else:
                # Now we test the user_agent from a big list.
                user_agents_test = ("w3c ", "acs-", "alav", "alca", "amoi", "audi",
                                    "avan", "benq", "bird", "blac", "blaz", "brew",
                                    "cell", "cldc", "cmd-", "dang", "doco", "eric",
                                    "hipt", "inno", "ipaq", "java", "jigs", "kddi",
                                    "keji", "leno", "lg-c", "lg-d", "lg-g", "lge-",
                                    "maui", "maxo", "midp", "mits", "mmef", "mobi",
                                    "mot-", "moto", "mwbp", "nec-", "newt", "noki",
                                    "xda", "palm", "pana", "pant", "phil", "play",
                                    "port", "prox", "qwap", "sage", "sams", "sany",
                                    "sch-", "sec-", "send", "seri", "sgh-", "shar",
                                    "sie-", "siem", "smal", "smar", "sony", "sph-",
                                    "symb", "t-mo", "teli", "tim-", "tosh", "tsm-",
                                    "upg1", "upsi", "vk-v", "voda", "wap-", "wapa",
                                    "wapi", "wapp", "wapr", "webc", "winw", "winw",
                                    "xda-",)

                test = user_agent[0:4].lower()
                if test in user_agents_test:
                    return self.antique_mobile(request)
            return self.desktop(request)

    def desktop(self,request):
        pass

    def modern_mobile(self,request):
        pass

    def antique_mobile(self,request):
        pass

class Page(View,TemplateResponseMixin,JSONResponseMixin):
```

```
    name = "generic_page"
    template_name = ria_settings.DEFAULT_RIA_TEMPLATE

    javascript = [
        Script(id="core",path=settings.STATIC_URL + "ria_core.js",lifetime="static"),
    ]

    css = [

    ]

    layouts = [
        Layout(id="main_layout"),
    ]

    def dispatch(self,request, *args, **kwargs):
        get_token(request)
        return super(Page, self).dispatch(request,*args, **kwargs)

    def render_to_response(self, context):
        if self.request.method == 'POST' and self.request.POST.has_key('ria_init'):
            return JSONResponseMixin.render_to_response(self,context)
        else:
            return TemplateResponseMixin.render_to_response(self, context)

    def get(self, request, *args, **kwargs):
        return self.render_to_response(self.get_initial_context(request))

    def post(self, request, *args, **kwargs):
        return self.render_to_response(self.get_ria_context(request))

    def get_initial_context(self,request):
        return {'redirect':request.path}

    def get_ria_context(self,request):
        javascript = []
        jss = []
        for script in self.javascript:
            jss.append(script)
        for layout in self.layouts:
            for script in layout.get_required_js():
                if not script in jss:
                    jss.append(script)
        for script in jss:
            javascript.append(script.to_context_map())

        css_context = []
        csss = self.css
        for layout in self.layouts:
            for css in layout.get_required_css():
                if not css in csss:
                    csss.append(css)
        for css in csss:
            css_context.append(css.to_context_map())

        layouts = []
        for layout in self.layouts:
            layouts.append(layout.to_context_map())
        return {'javascript':javascript,'css':css_context,'layouts':layouts}
```

Listado 6.19: PageDispatcher y Page

PageDispatcher es una implementación básica de un sistema de elección de vista basado en las vistas genéricas de Django. Para utilizar PageDispatcher solo hace falta heredar de el mismo e implementar los métodos *desktop*, *modern_mobile* y *antique_mobile* de forma que cada uno devuelva la vista que proceda. Este sistema obviamente es bastante limitado pero sirve como funcionalidad básica y es muy fácil de mejorar y cambiar por el desarrollador.

Page es la piedra angular de la librería; define todos los elementos que contendrá una página y el funcionamiento básico de las mismas, incluyendo las diferentes respuestas dependiendo de qué tipo de petición llega. Nótese que se utiliza la función *get_token* en todas las peticiones para obligar a Django a generar un código de protección para CSRF[6] y que lo mande como cookie al cliente web.

6.2.6 **Adaptación de los contenidos al tamaño de la ventana**

En el experimento de utilización de la librería se ha intentado adaptar el contenido al tamaño de la ventana. A la hora de implementar esto se consideran dos enfoques:

- Elementos que gestionan su tamaño: Este enfoque consiste en hacer que cada elemento de la página se registre como manejador del evento de cambio de tamaño de ventana.
- Escalamiento promedio de elementos: Este enfoque consiste en cambiar un valor de escala según el tamaño de la ventana y que los elementos se ajusten por CSS de acuerdo a dicho valor de escala automáticamente

El primer enfoque es más complicado de implementar y tiene la desventaja de tener un tiempo de procesamiento más largo en la carga de la página y en los cambios de tamaño de ventana pero es el que ofrece mejores resultados visuales ya que se tiene más control sobre la representación del contenido e incluso es posible cambiar la forma de presentar el contenido según el tamaño. Para hacer funcionar este sistema es recomendable utilizar elementos contenedores con anchura y altura porcentuales y utilizando el alto y ancho adaptar su contenido.

El segundo enfoque ofrece menos control sobre la visualización, es más limitado pero también es mucho más sencillo de implementar y adapta los contenidos mucho más rápido. Para hacer funcionar este sistema se establecen las alturas y anchuras de los elementos porcentualmente y mediante medidas relativas con respecto al tamaño de fuente (unidades em). Al cambiar el tamaño de ventana se cambia el tamaño de fuente base y con eso cambia el tamaño del contenido. Dado el escaso tiempo del que se ha dispuesto para la realización de la prueba se ha optado por esta opción.

```
function is_in_between(value,lower_limit,higher_limit) {
  return value >= lower_limit && value <= higher_limit;
}

var width_margin = 30;
var height_margin = 30;

function check_resolution(width,height) {
  return (is_in_between(window.outerWidth,width-width_margin,width+width_margin) &&
    is_in_between(window.outerHeight,height-height_margin,height+height_margin));
}

$.fn.fitKeepRatio = function(parent){
  $this=$(this);
  if (parent == undefined) {
    parent = $this.parent();
  }

  var h = $this.height();
  var w = $this.width();
  var fh = parent.height();
  var fw = parent.width();
```

```
  if ((w/ h)*fh > fw) {
    h = "auto";
    w = "100%";
  }
  else {
    w = "auto";
    h = "100%";
  }
  var measures={"height":h,"width":w};
  $this.css(measures);
}

var widthWeight = 110;
var heightWeight = 62;

function resizedw() {
  $window= $(window);
  $body= $("body");
  if (check_resolution(1024,600) {
    $body.css("font-size",9);
  } else if (check_resolution(1024,768) {
    $body.css("font-size",9);
  } else if (check_resolution(1280,800) {
    $body.css("font-size",10);
  } else if (check_resolution(1280,960) {
    $body.css("font-size",11);
  } else if (check_resolution(1280,1024) {
    $body.css("font-size",11);
  } else if (check_resolution(1366,768) {
    $body.css("font-size",11);
  } else if (check_resolution(1440,900) {
    $body.css("font-size",12);
  } else if (check_resolution(1600,1200) {
    $body.css("font-size",12);
  } else if (check_resolution(1920,1080) {
    $body.css("font-size",13);
  } else if (check_resolution(1920,1200) {
    $body.css("font-size",13);
  } else {
    var t = $window.width()/ widthWight;
    var t2 = $window.height()/heightWeight;
    if (t<t2)
    {
      $body.css("font-size",t);
    }
    else
    {
      $body.css("font-size",t2);
    }
  }
  $(".fitImage").fitKeepRatio();
}

var doit;
$(window).resize(function(){
  clearTimeout(doit);
  doit = setTimeout(function(){resizedw();}, 100);
});
```

Listado 6.20: Adaptación de contenidos

El sistema de ajuste de tamaño tiene dos partes: cálculo del tamaño de fuente y ajuste de imágenes manteniendo el ratio de aspecto. En el caso del cálculo del tamaño de fuente se implementan a mano tamaños para las resoluciones típicas y se deja un sistema aproximado por pesos ajustados a mano para el resto de tamaños.

No todos los navegadores se comportan igual ante el cambio de tamaño de la ventana; mientras que algunos lanzan un evento al terminar de cambiar el tamaño, otros lo lanzan constantemente mientras se cambia. Para evitar cambiar el tamaño demasiadas veces se implementa un sistema de espera por el cual al menos tienen que pasar 100 milisegundos antes de realizar un proceso de cambio de tamaño.

7. CONCLUSIONES

Este proyecto empezó solo como la idea de crear una librería pequeña que facilitase la creación de RIAs dada la experiencia anterior en el desarrollo de una junto a un antiguo compañero de trabajo. Al pensar que tal vez se quedaría pequeño el proyecto y entender que el riesgo de no obtener un resultado aceptable era moderadamente alto decidí completar el proyecto con la implementación de algunos aspectos pendientes del portal web Sminn sobre el que se implantará la librería implementada. La motivación era clara: aportar una librería de calidad a la comunidad del software libre y a la vez proporcionar resultados a corto plazo a la empresa que financia mis estudios.

Dado que no podría liberar el código en sí de los aspectos implementados para dicha empresa, si que podía en cualquier caso registrar el trabajo de diseño y el razonamiento seguido para las soluciones en la presente memoria, por lo que en todo momento he percibido beneficio en todos los sentidos del proyecto: beneficio para la empresa financiadora, beneficio para mi, beneficio para la comunidad y beneficio para la propia UOC.

Tras varios meses de trabajo duro al fin llego al final de este trozo de camino y mirando hacia atrás hecho de menos más tiempo para el trabajo sobre lo que considero la parte más importante del proyecto, es decir, la librería a liberar. Estoy en general satisfecho con la forma que ha tomado pero veo al mismo tiempo puntos de diseño por mejorar, cosas que he hecho mal y he sabido que he hecho mal desde el primer momento y cosas que habré hecho mal y descubriré cuando prosiga mi trabajo tras un breve descanso. Soy consciente en cualquier caso de que para la librería probablemente esto solo sea el principio, y con suerte, lo que está por venir nos permita a todos los que trabajamos con Django un trabajo aún más satisfactorio, ya sea mediante la propia librería o mediante otras que se muevan en su ámbito.

He tenido adicionalmente el placer de encontrarme en una situación ligeramente complicada en lo que se refiere al software libre. Trabajo para una empresa con experiencia escasa en el tema del software libre y que por diversas razones no se encuentra en posición de liberar la mayoría de su software. En este punto acordé con la dirección técnica que aunque estaba claro que no se podía liberar todo aquello que tuviera información de sus clientes o información implícita del funcionamiento de sus sistemas internos, si que era interesante la liberación de middleware tanto por la contribución a la comunidad como por la propia contribución de la comunidad a dicho software. Espero que poco a poco pueda liberar otras librerías o middleware desarrollados en dicha empresa.

Espero que si has llegado hasta aquí sea porque has encontrado interesante esta memoria y espero que te haya servido para algo. Si tienes alguna pregunta relacionada con el proyecto o con esta memoria puedes ponerte en contacto conmigo en la dirección sblanco@ts2.es.

A. HOW TO: DJANGO-JABBERWOCKY

A.1 Introduction

This brief how-to provides an overview of the use of the Django web framework and Django-Jabberwocky to create RIAs and reusable components.

A.2 Why Jabberwocky?

The name Jabberwocky is an interesting name given it's different sources. First off, it was a nonsensical poem written by *Lewis Carrol* with a lot of invented words which were actually added to the English dictionary later on. It also is a medieval film by *Terry Gilliam* (hard to miss the pythonic reference) in which the kingdom is threatened by a beast called Jabberwocky (based somehow on the original poem). Lastly, it references the fantastic *Better of Ted* series which had a very funny episode in which Ted and Veronica had to make a keynote about a non-existing project called Jabberwocky which was "supposed to change the way we do business".

A.3 Where do I start?

Is this your first time developing with Django? Then go read it's fantastic documentation, starting with the 4-part tutorial to get you started.

First off you need to install Jabberwocky. You need Django version 1.3 or greater and optionally you can install cssmin and jsmin to allow resource minification. Pip is probably your best friend to install those packages and many more. As of today, Jabberwocky isn't available via pip or easy_install, so you'll have to go with the git repository for now. Jabberwocky is just a normal Django app, so you can either install it in your python environment or just install it inside your project; it will work either way.

A.4 Ok then... How can I get going fast?

Such a hurry! I guess you're familiar with class based generic views introduced in Django 1.3. If not, go read about it first, you won't regret it. If you are, let's start with the configuration. Open up your project's *settings.py* and let's see what we can do.

```
DEFAULT_RIA_TEMPLATE= 'ria/default.html'
RIA_MINIFY_JS = True
RIA_MINIFY_CSS = True
RIA_DEFAULT_RESOURCE_LIFETIME = 3600
RIA_RESOURCE_ORIGIN_PATH = settings.STATIC_ROOT
RIA_RESOURCE_DESTINATION_PATH = settings.MEDIA_ROOT
RIA_RESOURCE_ORIGIN_URL = settings.STATIC_URL
RIA_RESOURCE_DESTINATION_URL = settings.MEDIA_URL
```

Listado A.1: Jabberwocky configuration options

All of the options above have default values so you don't have to specify any of them if you don't wish to. Let's see what each of them does:

- DEFAULT_RIA_TEMPLATE: Django template that will be used for the first request in a session. You can create your own template to include anything you see fit, but you probably won't need it as the system does it all for you. You can safely ignore this one for most cases as it implements a basic template by default.
- RIA_MINIFY_JS and RIA_MINIFY_CSS: Those are default values for minification. If you ignore the *should_minify* parameter when creating *Script, JoinedScript, CSS and JoinedCSS* instances those values will be used. Both are True by default.
- RIA_DEFAULT_RESOURCE_LIFETIME: This settings configure the time interval between regenerating each joined or minified resource in seconds. This setting is only valid when *DEBUG is False*. When you are debugging every resource will be regenerated each time to ensure the latest changes are included. The default value is 1 hour (3600 seconds).
- RIA_RESOURCE_ORIGIN_PATH, RIA_RESOURCE_DESTINATION_PATH, RIA_RESOURCE_ORIGIN_URL and RIA_RESOURCE_DESTINATION_URL: Those four instruct the system to look for original resources in RIA_RESOURCE_ORIGIN_PATH, deploy minified and joined resources in RIA_RESOURCE_DESTINATION_PATH and use RIA_RESOURCE_ORIGIN_URL and RIA_RESOURCE_DESTINATION_URL as the base URLs to get the processed resources. The default values of these settings are STATIC_ROOT,MEDIA_ROOT, STATIC_URL and MEDIA_URL as they work well with the Django development server.

Now that we have our settings ready to go, we can start hacking our RIA away. Like with any Django project, you'll need at least one app for your logic. To keep this how-to centered in how Jabberwocky works we won't create any Django model. Open the *views.py* file or another Python module if you want to keep your views separated.

```
from ria.views import Page, PageDispatcher
from ria.models import Script, CSS, Layout, StyledElement, Image, Link, Div

class IndexAutogeneratedContent(StyledElement):
    def __init__(self,id,load_callback=None):
        classes = ["index_content"]
        required_js = [
            Script(id="index_constructor",path="js/index.js",lifetime="volatile"),
        ]
        required_css = [
            CSS(id="css_index",path="css/index_tutorial.css"),
        ]
        StyledElement.__init__(self, id, type="index_content", classes=classes,
                               required_js= required_js, required_css = required_css,
                               elements = [],load_callback=load_callback)

class IndexLayout(Layout):
    def __init__(self, id, preload_callback=None, postload_callback=None):
        Layout.__init__(self, id, preload_callback, postload_callback, [])
        self.elements = [
            Div(id="autogenerated_container",
                elements = [
                    IndexAutogeneratedContent(id="autogenerated_content",
                                              load_callback="autogenerate_callback")
                ],
            ),
            Div(id="pregenerated_content",
                elements=[
                    Image(id="index_image",src="/media/images/index.png"),
                    TextLink(id="contact_link", href="#", text="CONTACT",
                            on_click_callback="contact_clicked"),
                ],
                required_js=[
```

```
                Script(id="index_constructor",path="js/index.js",lifetime="volatile"),
            ]
        )
    ]


class Index(Page):
    name="Index"
    javascript=[
        Script(id="jquery",path="js/jquery.js",lifetime="static"),
        Script(id="index_constructor",path="js/index.js",
              lifetime="volatile",should_minify=False),
    ]
    css=[
        CSS(id="css_base",path="css/base.css"),
        CSS(id="css_index",path="css/index_tutorial.css"),
    ]
    layouts=[
        IndexLayout(id="index_content", preload_callback="index_preload",
                  postload_callback="index_postload")
    ]
```

Listado A.2: Our first view

Whoa! That's seems like a lot to take in, but if you examine the excerpt closely you'll see that most of it is just for demo purposes. Let's check what that batch of code does.

1. The Index class is a child of the Page class based view. It basically contains a description of the page that will be rendered. As seen above, you can declare which scripts and CSS style sheets this page will work with and you should declare at least one layout (a container of sorts).
2. The IndexLayout class is instantiated to describe the content or structure of content for the index page of our new RIA. This class is a child of the Layout class and must declare a hierarchy of elements (instances of the Element class or it's children). A layout can also point to two callbacks so you can implement whichever you need just before or after the layout creation in the client side. This layout has two content elements that show two different ways of creating content:
    - Autogenerated content: The element is technically empty and it will be it's Javascript constructor (normally declared as a dependency) the one that will generate the content. This is a classic RIA approach and puts all the responsibility and power in the client side.
    - Pregenerated content: The element has a hierarchy of subelements of it's own, some of them of autogenerated content and some of them of pregenerated content. You can create any component you can think of and you can design your components to be parameterisable.
3. The IndexAutogeneratedContent is instantiated as part of an IndexLayout to generate content in the layout. It is a child of StyledElement (which is actually a child of Element that adds a *classes* attribute) and is a reusable component, as it can declare its Javascript and CSS dependencies (which by the way are repeated in the Index class just to show that Jabberwocky merges all dependencies automatically).

Before going down the Javascript path, let's give a quick reference for the classes we have introduced so far:

- Page: A Page is a class based Django view with which you can describe a RIA view declaratively. It contains the following elements:
    - name: A name for the page, just for reference.
    - javascript: Just a list of Script objects declaring the scripts the view needs to work. Each element can also declare its dependencies so you don't have to put them all here, just the ones you know you need.

  - css: Just a list of CSS objects declaring the CSS style sheets the view needs to work. Just like with Javascript, each element can declare its own dependencies.
  - layout: Just a list of Layout objects that declare the structure and possibly the content of the page. You should have at least a layout per view.
- Script: A Script object represents a Javascript file with added configuration values. A Script contains:
  - id: Unique identifier. Each different element should have a different ID, obviously.
  - path: File path relative to RIA_RESOURCE_ORIGIN_PATH for the script.
  - lifetime: Indicates whether the script should be unloaded when not needed (volatile) or should reside in memory as long as the RIA is running (static). It's default value is "volatile".
  - init_func: Optional javascript function that will be called to initialize the script.
  - unload_func: Optional javascript function that will be called to uninitialize the script.
  - should_reinitialize: Indicates whether the script must be reinitilized each time a page that uses it is loaded even if it is already loaded or whether it will be leaved as is. It's default value is "False".
  - should_redownload: Indicates whether the script must be reloaded each time a page that uses it is loaded even if it has already been loaded or whether it will be leaved as is. It's default value is "False".
  - should_minify: Indicates if the script will be minified before sending it to the client. If true, and if the previous minified copy is older than "processed_lifetime" seconds, Jabberwocky will minify the script and leave the minified copy in RIA_RESOURCE_DESTINATION_PATH for the client to download. It's default value is "False".
  - processed_lifetime: Age in seconds that a minified copy should at least have before being discarded and regenerated. This should be low if the script changes very frequently and high if the script rarely changes. When the Django DEBUG setting is activated each script is processed no matter how old the previous file is so this value should be set for release deployments.
- CSS: A CSS object represents a CSS file with some added configuration values. A CSS contains:
  - id: Unique identifier. Each different element should have a different ID, obviously.
  - path: File path relative to RIA_RESOURCE_ORIGIN_PATH for the CSS.
  - lifetime: Indicates whether the CSS should be unloaded when not needed (volatile) or should reside in memory as long as the RIA is running (static). It's default value is "volatile".
  - should_minify: Indicates if the CSS will be minified before sending it to the client. If true, and if the previous minified copy is older than "processed_lifetime" seconds, Jabberwocky will minify the CSS and leave the minified copy in RIA_RESOURCE_DESTINATION_PATH for the client to download. It's default value is "False".
  - processed_lifetime: Age in seconds that a minified copy should at least have before being discarded and regenerated. This should be low if the CSS changes very frequently and high if the script rarely changes. When the Django DEBUG setting is activated each script is processed no matter how old is the previous file so this value should be set for release deployments.
- Layout: A layout is a piece of the page structure. Each layout has three elements:
  - id: Unique identifier. Each different element should have a different ID, obviously.
  - elements: A list of Element objects that represent content blocks.

- preload_callback: The name of a Javascript function or method that will be automatically called just before constructing and placing the layout. You can use this callback to manually construct a layout without elements if you want to manage everything by yourself.
- postload_callback: The name of a Javascript function or method that will be automatically called just after the layout construction and placement. You can use this callback to manually construct a layout without elements if you want to manage everything by yourself.

- StyledElement: StyledElement is a child class of Element and the usual starting point to create your own components. It contains:
  - id: Unique identifier. Each different element should have a different ID, obviosuly.
  - type: String that indicates the type of element. This is used to associate constructor functions to types in the client side and thus allow extensibility.
  - classes: Optional list of CSS classes assigned to this element.
  - required_js: Optional list of Script objects that are required by this element. Each Element's required_js list will be joined with the container Page's javascript list.
  - required_css: Optional list of CSS objects that are required by this element. Each Element's required_css list will be joined with the container Page's css list.
  - elements: Optional list of elements that will be created inside this element.
  - load_callback: Optional javascript function that will be called when the element has been created and loaded.
- Div, Image and TextLink: Those are all subclasses of StyledElement. Each one receives several parameters describing their content and reactions to different events. They are just pretty basic elements to show how custom elements should be developed.

Now that we are familiarized with the server side, let's see how the client side looks like. The view references many times the "index.js" script. Obviously, this script should provide the following:

- A constructor function for "index_content" elements.
- All callbacks and functions referenced by the view or views that use the script.
- Any other needed functionality.

```
function IndexContentLoader() {
    this.type = "index_content";
    this.create_element = function (element) {
        element.document_element = document.createElement("div");
        element.document_element.id = element.id;
        var content = document.createElement("h1");
        content.innerHTML = "Hello Index!";
        element.document_element.appendChild(content);
        assign_classes(element);
    }
}
ria_core.layout_manager.register_element_loader(new IndexContentLoader());

function autogenerate_callback () {
    alert("Autogenerate Callback!!!");
}

function contact_clicked (element) {
    alert("Contact clicked! " + element.id);
    ria_core.do_ria_request("/contact",null);
}

function index_preload () {
```

```
    alert("Index preload!");
}

function index_postload () {
    alert("Index postload!");
}
```

Listado A.3: The client side of our first Page

It wasn't that bad, was it? First off, this script adds a new element loader that knows how to create elements with "index_content" type. The IndexContentLoader class provides the function and the type so the layout manager can register the loader. As we saw earlier, an element described in the server could have any number of attributes which are serialized to JSON and then end up in an Element object in the Javascript world. This means that an element loader or constructor can use whatever you see fit to create you element and tie it up to any events imaginable. You can do it as manual or automatic as you wish. It's just what you want to do with your data and your great skills. You want to create a supercomplex menu component? Give it a try and you'll see.

For example purposes all callbacks are simple one-liners. Note that the click event handler for the link (an example of event binding in the included TextLink constructor) receives the element with all the data it needs to do something meaningful. Note that the aforementioned event handler does something more; it requests another Page. Once the client gets the new page description it will automatically destroy the scripts, CSS and elements that are no longer needed, it will create the new ones and keep the common ones. How? Thanks to all the descriptive information you put on the server side.

Let's create a little Contact page to finish this example.

```
from ria.views import Page, PageDispatcher
from ria.models import Script, CSS, Layout, StyledElement, Image, Link, Div

class IndexDifferentLayout(Layout):
    def __init__(self,id,preload_callback=None,postload_callback=None):
        Layout.__init__(self,id,preload_callback,postload_callback,[])
        self.elements = [
            Div(id="autogenerated_container",
                elements = [
                    IndexAutogeneratedContent(id="autogenerated_content",
                                              load_callback="autogenerate_callback")
                ],
            ),
            Div(id="pregenerated_content",
                elements=[
                    Image(id="index_image",src="/media/images/index.png"),
                    TextLink(id="index_link",href="#",text="INDEX",
                            on_click_callback="index_clicked"),
                ],
                required_js=[
                    Script(id="index_constructor",path="js/index.js",
                          lifetime="volatile"),
                    Script(id="contact_constructor",path="js/contact.js",
                          lifetime="volatile"),
                ]
            )
        ]


class Contact(Page):
    name="Contact"
    javascript=[
        Script(id="contact_constructor",path="js/contact.js",
              lifetime="volatile",should_minify=True),
```

```
    ]
    css=[
        CSS(id="css_base",path="css/base.css"),
        CSS(id="css_contact",path="css/contact.css"),
    ]
    layouts=[
        IndexDifferentLayout(id="index_content", preload_callback="contact_preload",
                             postload_callback="contact_postload"),
    ]
```

Listado A.4: Another page

Note that this example has been written to show what Jabberwocky can do for you as fast as possible; there are some times like in the previous code snippet that things are done in a very nasty way and can be done a lot better. Just read the code and understand how it works, the rest will work out. Please don't write as badly as these examples are!

Let's analyse what this new page does... First off, there is a very similar layout to the previous one. There are only two differences with IndexLayout: different TextLink element and two script dependencies. My intention with this example is to show how the system works with a very similar group of elements. The IndexDifferentLayout is used by the Contact Page and surprisingly enough, using the same id that we used for the Index page layout.

If it were a different id, the javascript machinery would think is a different layout, so it would just destroy the IndexLayout and create this new layout. But with the same id, it will look for differences and work them out. In this example the page transition will destroy the previous link (as it has a different id) and create the new one. The rest of the content will be left untouched.

Note also that the javascript and CSS prerequisites have changed. JQuery was a static script, so it will remain loaded and ready even if it is not used in this page. index.js will also remain untouched as it is required by IndexDifferentLayout. As for contact.js, it will be loaded when loading the Contact page and unloaded when navigating to a Page that does not require it.

The javascript side of this page should be very familiar by now.

```
function index_clicked (element) {
    alert("Index clicked! " + element.id);
    ria_core.do_ria_request("/index",null);
}

function contact_preload () {
    alert("Contact preload!");
}

function index_postload () {
    alert("Contact postload!");
}
```

Listado A.5: The client side of the Contact page

The most part of the page will use index.js, so there is little left to do in contact.js. Two callbacks and the new link's click event do exactly the same thing their alter egos did, but according to their new identity.

To end this brief example, add the pertinent URLS in you URLconf just like with any Django class based generic view. If we did everything right we should have a very simple single page RIA that apparently just changes a link. The interesting thing about it is that we developed it a lot more like our usual content web project (view/page based) but we got as result a single page RIA. I could go on and on about the advantages of sharing the complexity of the presentation layer between the server and the client, but I think you'll enjoy discovering it by yourself.

A.5 Resource optimization

Up to now you have worked with the basic features of Jabberwocky. If you take a look at the source code, you'll see there are some things that I haven't mentioned. One of those things is the use of the JoinedScript and JoinedCSS classes (server side). Those classes are child classes of Script and CSS respectively and they allow for granular resource optimization. And by that I mean that you can join any Scripts and CSS together (with or without minification) automatically in each page, generating a equivalent resource.

The advantage to joining resources is that the client needs less connections to get all it needs to work, thus pages will load faster. There is also a disadvantage, unfortunately. Even though as of today we are not worried about script memory consumption, as RIAs get more complex we might want to optimize memory consumption, specially for mobile devices. Script joining means you will load and unload large chunks that have a lot in common and you won't be able to get the minimum memory consumption. The good thing, you can choose how to it. Let me show you the previous IndexLayout but this time resource optimized.

```
from ria.views import Page, PageDispatcher
from ria.models import Script, JoinedScript, CSS, JoinedCSS, Layout,
          StyledElement, Image, Link, Div

class Index(Page):
    name="Index"
    javascript=[
           JoinedScript(id="js_index",path="index_min.js",lifetime="volatile",
                     should_minify=True,
               scripts = [
                   Script(id="jquery",path="media/js/jquery.js"),
                   Script(id="index_constructor",path="js/index.js"),
               ])
    ]
    css=[
        JoinedCSS(id="css_index",path="index_min.css",should_minify=True,
           css_list=[
               CSS(id="css_base",path="css/base.css"),
               CSS(id="css_index",path="css/index.css"),
           ])
    ]
    layouts=[
        IndexLayout(id="index_content",
               preload_callback="index_preload",
               postload_callback="index_postload")
    ]
```

Listado A.6: Resource optimization

A.6 Adapt to the client

Another useful feature you may need is a way to easily adapt your RIA to each device. Instead of creating Page subclasses that adapt to all possible devices, Jabberwocky let's you create PageDispatchers. A PageDispatcher is a generic view itself that sits between your URL and your Page. When creating a PageDispatcher child you only need to implement three methods, one for each type of device (PageDispatcher is a simple way of adapting to devices. You should modify it or create your own system if you need anything more complex).

Your URLConf should point to your PageDispatchers and your PageDispatchers should return the correct view (Page) for each device.

```
from ria.views import Page, PageDispatcher

class IndexDispatcher(PageDispatcher):
    def desktop(self,request):
        return IndexDesktop.as_view()(request)

    def modern_mobile(self,request):
        return IndexMobile.as_view()(request)

    def antique_mobile(self,request):
        return IndexTextBased.as_view()(request)
```

Listado A.7: Device adaptation

A good thing about this method is that you can use any view for any browser type, so you can combine other Django projects with Jabberwocky easily.

A.7 Create your own elements

The most important feature of Jabberwocky is it's extensibility; you can easily create your own reusable elements. If you want to create a custom element, you only have to worry about three things:

- Give your element a unique type name
- Implement the to_context_map method adding to the context dictionary each attribute you want to be able to use in the javascript constructor.
- Implement the javascript constructor and make sure your Element references the script file that contains it.

Let's check a simple implementation of a custom JQuery Cycle component.

```
from ria.models import Script, StyledElement

class Image_Cycle(StyledElement):
    def __init__(self,id,images,width="100%",height="100%",load_callback=None):
        self.images = images
        self.width = width
        self.height = height
        StyledElement.__init__(self,id,type="image_cycle",classes=["slideshow",],
            required_js=[
                Script(id="jquery",path="media/js/jquery.js",lifetime="static"),
                Script(id="jquery_easing",path="media/js/easing.js",lifetime="volatile"),
                Script(id="jquery_cycle",path="media/js/cycle.js",lifetime="volatile"),
            ],
            required_css=[
            ],
            elements=[
            ],
            load_callback=load_callback
        )

    def to_context_map(self):
        context = StyledElement.to_context_map(self)
        context["images"] = self.images
        context["width"] = self.width
        context["height"] = self.height
        return context
```

Listado A.8: Cycle Element in the server side

```
function CycleLoader() {
    this.type = "image_cycle";
    this.create_element = function (element) {
        element.document_element = document.createElement("div");
        element.document_element.id = element.id;
        for (index in element.images) {
            var image = document.createElement("img");
            image.src = element.images[index].path;
            image.onclick = function() {
                window.location = element.images[index].link;
            }
            element.document_element.appendChild(image);
        }

        assign_classes(element);

        $(element.document_element).cycle({
            fx: 'fade',
            fit: 1,
            width: element.width,
            height: element.height
        });
    }
}
ria_core.layout_manager.register_element_loader(new CycleLoader());
```

Listado A.9: Cycle constructor in client side

A.8 All up to you

Django Jabberwocky is WIP right now. It will probably remain fairly stable but there is still a lot of work to do. Contact me if you want to contribute somehow.

B. AGRADECIMIENTOS

- A la gente de Elson Sistemas por su colaboración y apoyo en la ejecución del proyecto.
- A Dr. Minsky por su ayuda en todo lo concerniente a diseño y estructura del portal web de Sminn y por su ayuda en la elaboración del vídeo de presentación. Ha sido un largo camino el del portal sminn.com y aun queda lo mejor por venir.
- A Oihane Kamara por sus consejos y ayuda en la revisión de la documentación. Espero que el trabajo realizado en Django-Jabberwocky puede serle tan útil como su ayuda lo ha sido para mi.
- A Josu Bermudez por su inspiración, por el tiempo que hemos pasado pensando en el desarrollo de RIAs y en los posibles enfoques y su ayuda a la revisión del documento.
- A Ignacio Blanco por su ayuda y apoyo en los momentos de incertidumbre y presión.
- A Donald E. Knuth por haber creado TEX, a Leslie Lamport por LATEXy a toda la gente que ha hecho que a día de hoy sea, sin lugar a dudas, el mejor sistema de creación de documentos.
- A Borja Sotomayor, Alvaro Uría y todas las personas que han colaborado para hacer posible la plantilla de LATEXde la Universidad de Deusto, la cual he modificado para esta memoria.
- A Guido Van Rossum y toda la gente que ha convertido Python en la maravilla que es.
- A toda la gente que ha trabajado en hacer Django un sistema a la altura del lenguaje que hay detrás.

BIBLIOGRAFÍA (LIBROS Y ARTÍCULOS)

[1] Adrian Holovaty Jacob Kaplan-Moss. *The Definitve Guide to Django: Web Development Done Right*. 2007.

[2] Steve McConnell. *Code Complete Second Edition*. 2004.

[3] Roger S. Pressman. *Ingeniería del software. Un enfoque práctico*. 2005.

BIBLIOGRAFÍA (OTRAS FUENTES)

[4] Página de proyecto de Apache HTTP Server [online]. URL: http://projects.apache.org/projects/http_server.html.

[5] fcgi vs gunicorn vs uwsgi [online]. URL: http://www.peterbe.com/plog/fcgi-vs-gunicorn-vs-uwsgi.

[6] CSRF en Wikipedia [online]. URL: http://en.wikipedia.org/wiki/Cross-site_request_forgery.

[7] Introduction to CSS3 [online]. URL: http://www.w3.org/TR/css3-roadmap/.

[8] Página principal de Django Debug Toolbar [online]. URL: https://github.com/robhudson/django-debug-toolbar.

[9] Django Main Page [online]. URL: http://www.djangoproject.com/.

[10] Django Book [online]. URL: http://www.djangobook.com/.

[11] Django Tutorial [online]. URL: http://docs.djangoproject.com/en/1.0/intro/tutorial01/.

[12] Página del Document Object Model de la W3C [online]. URL: http://www.w3.org/DOM/.

[13] ERP en Wikipedia [online]. URL: http://es.wikipedia.org/wiki/Planificaci%C3%B3n_de_recursos_empresariales.

[14] Uso de memcached en Facebook [online]. URL: http://www.facebook.com/note.php?note_id=39391378919.

[15] Página de proyecto de gettext [online]. URL: http://www.gnu.org/software/gettext/.

[16] Página principal de Git [online]. URL: http://git-scm.com/.

[17] Grok - A Smashing web framework [online]. URL: http://grok.zope.org/.

[18] Página principal de Gunicorn [online]. URL: http://gunicorn.org/.

[19] HTML 5 Specifications Draft [online]. URL: http://dev.w3.org/html5/spec/Overview.html.

[20] HTTPS en la wikipedia [online]. URL: http://es.wikipedia.org/wiki/Hypertext_Transfer_Protocol_Secure.

[21] Página oficial de Java [online]. URL: http://www.java.com/es/.

[22] ECMAScript Language Specification [online]. URL: http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[23] Página principal de JQuery [online]. URL: http://jquery.com/.

[24] Página oficial de JSON [online]. URL: http://json.org/.

[25] Building a LAMP server [online]. URL: http://www.lamphowto.com/.

[26] Página principal de lighttpd [online]. URL: http://www.lighttpd.net/.

[27] Página principal de MariaDB [online]. URL: http://mariadb.org/.

[28] Redis vs Memcached [online]. URL: http://systoilet.wordpress.com/2010/08/09/redis-vs-memcached/.

[29] Página principal de Memcached [online]. URL: http://memcached.org/.

[30] Página principal de MySQL [online]. URL: http://www.mysql.com/.

[31] Página principal de Nginx [online]. URL: http://nginx.org/.

[32] Página oficial de PHP [online]. URL: http://www.php.net/.

[33] Página principal de PostgreSQL [online]. URL: http://www.postgresql.org/.

[34] Página principal de Prototype [online]. URL: http://www.prototypejs.org/.

[35] Pylons main page [online]. URL: http://pylonshq.com/.

[36] Python Programming Language - Official Website [online]. URL: http://www.python.org/.

[37] Página principal de redis [online]. URL: http://redis.io/.

[38] REST en Wikipedia [online]. URL: http://es.wikipedia.org/wiki/Representational_State_Transfer.

[39] Wikipedia - Rich Internet Applications [online]. URL: http://en.wikipedia.org/wiki/Rich_Internet_application.

[40] Página principal de Django Rosetta [online]. URL: http://code.google.com/p/django-rosetta/.

[41] Página oficial de Ruby [online]. URL: http://www.ruby-lang.org/es/.

[42] Página principal de Django South [online]. URL: http://south.aeracode.org/.

[43] Página principal de SQLite [online]. URL: http://www.sqlite.org/.

[44] SSL y TLS en Wikipedia [online]. URL: http://es.wikipedia.org/wiki/Transport_Layer_Security.

[45] uWSGI Wiki [online]. URL: http://projects.unbit.it/uwsgi/.

[46] API de Websocket [online]. URL: http://dev.w3.org/html5/websockets/.

[47] WSGI Wiki [online]. URL: http://wsgi.org/wsgi/.

[48] Página de proyecto del módulo xsendfile [online]. URL: https://tn123.org/mod_xsendfile/.

[49] Página principal de YUI [online]. URL: http://developer.yahoo.com/yui/.

[50] Página oficial de Zope [online]. URL: http://www.zope.org/.