Integración de Single-Page Application en Liferay

Trabajo Final de Grado

Autor

José de la Fuente Macía

Poniente: Marc Alier Forment

Director: Oriol Alemany

Grado en Ingeniería Informática

Ingeniería del Software

Facultad de Informática de Barcelona

Universidad Politécnica de Cataluña

Resumen

Las aplicaciones de una sola página (Single-Page Application [1] o SPA) son aquellas aplicaciones en las que todas las funcionalidades se desarrollan en una única página, cargando toda la lógica necesaria una vez al cargar la página o cargándola de forma dinámica en el momento en que sea necesario. Esta forma de desarrollar aplicaciones nace de la necesidad de mejorar la experiencia de usuario y el flujo de trabajo de las aplicaciones web.

Gracias al SPA ya no es necesario recargar la página web entera cada vez que el usuario ejecuta una operación, evitando así tiempos de espera que dependen tanto de la calidad de la web y sus servidores como de la conexión a Internet que tenga disponible el usuario en el momento de su uso o las características del equipo desde el que realiza la operación.

Al no tener que recargar la web entera para refrescarse, las interacciones y cambios de estado de la aplicación se manejan en el contexto de un único documento web, logrando una experiencia de usuario cercana a una aplicación de escritorio.

En este proyecto se creará una web Single Page Application para poder ver de primera mano las ventajas que este tipo de webs ofrecen con respecto a las páginas web convencionales.

Resum

Les aplicacions d'una sola pàgina (Single-Page Application[1] o SPA) són aquelles aplicacions en les quals totes les funcionalitats es desenvolupen a una única pàgina, carregant tota la lògica necessària una vegada al carregar la pàgina o carregant-la de forma dinàmica en el moment en què sigui necessari. Aquesta forma de desenvolupar aplicacions neix de la necessitat de millorar l'experiència d'usuari i el flux de treball de les aplicacions web.

Gràcies al SPA, ja no és necessari recarregar la pàgina web sencera cada vegada que l'usuari executa una operació, evitant així temps d'espera que depenen tant de la qualitat de la web i els seus servidors, com de la connexió a Internet que tingui disponible l'usuari en el moment del seu ús o les característiques de l'equip des de el que realitza l'operació.

Com que no s'ha de recarregar la web sencera per a refrescar-se, les interaccions i els canvis d'estat de l'aplicació es manegen en el context d'un únic document web, aconseguint una experiència d'usuari propera a una aplicació d'escriptori.

A aquest projecte es crearà una pàgina web Single Page Application per a poder veure de primera mà els avantatges que aquest tipus de web ofereixen respecte a les pàgines web convencionals.

Abstract

Single Page Applications[1] or SPA are those applications where all functionalities are developed on a single page, carrying all the necessary logic once on page load or loading it dynamically when necessary. This way of developing applications stems from the need to improve the user experience and workflow of web applications.

Thanks to SPA there is no longer necessary to reload the entire web page each time the user performs an operation, thus avoiding timeouts that depend on both the quality of the site and its servers and Internet connection the user has available at the time of use or characteristics of the system from which the operation is performed.

By not having to reload the entire web, interactions and changes of state of the application are handled in the context of a single web document, achieving an experience close to a desktop application user.

In this project a single page application web is created to see firsthand the advantages that such websites offer regarding conventional web pages.

Índice

1. Introducción

Los usuarios de webs son cada vez más impacientes y quieren ver los resultados de sus acciones de forma inmediata. Mirar a la pantalla mientras un círculo da vueltas para indicar que se está cargando la web no es agradable.

Además, cuantas más veces suceda esto o más se tarde en mostrar la vista de la página web más posibilidades hay de que el usuario abandone la web o no vuelva a utilizarla.

Para que las web sean cada vez más cercanas a aplicaciones de escritorio se está aprovechando de tecnologías JavaScript, Ajax y HTML para que estas sean cada más rápidas y necesiten menos comunicaciones con servidores para realizar sus funciones. Por este motivo las webs SPA cada vez tienen más mercado e incluso se implementan junto a web tradicionales dando lugar a webs híbridas.

1.1. Contexto

En un SPA el usuario no navega por un sistema de enlaces tradicionales si no que, mediante el uso de JavaScript [2], Ajax [3], HTML5 [4] o una combinación de estos, se actualiza lo que el usuario ve desde la misma página, de esta manera no hay recarga de páginas ni el control se transfiere a otras. Esta forma de desarrollar aplicaciones está tomando un fuerte impulso debido a la importancia de conseguir fluidez en la experiencia de usuario.

El funcionamiento es el siguiente, cuando el usuario interactúa con la página realizando una acción se envía una petición asíncrona al servidor, cuya respuesta, normalmente XML [5], JSON [6] o similares, es interpretada por la aplicación que actualizará el DOM (Document Object Model [7]) sin recargar el sitio, de esta forma los cambios se visualizarán a una gran velocidad y sin desperdicio de ancho de banda. Las SPA delegan muchas operaciones que anteriormente se realizaban en el servidor al lado del cliente, reduciendo así la complejidad del sistema.

AngularJS es uno de los *frameworks* más populares para el desarrollo de aplicaciones del lado de cliente utilizando JavaScript, esto junto al hecho de que está enfocado a la creación y mantenimiento de SPA y que haya sido creado y mantenido por Google han sido los puntos esenciales en la elección de este *framework* para el desarrollo del proyecto.

Liferay es, según el grupo Gartner [8] y por sexto año consecutivo, el gestor de contenidos de código abierto líder en el cuadrante mágico de Gartner [9] para portales horizontales, además es ampliamente utilizado en el mercado y cuenta con una buena visión de futuro. La arquitectura de *Liferay* ofrece escalabilidad, fiabilidad y alto rendimiento, al ser ligero y ágil los proyectos se llevan a cabo en

menos tiempo y con menor coste. Además cuenta con herramientas avanzadas de gestión de contenido, colaboración y portal.

Liferay hace uso de *portlets* para generar contenido del portal, estos *portlets* pueden extraer información de distintas fuentes, ya sea de bases de datos, gestores de contenido… Los *portlets* son aplicaciones contenidas dentro de un portal y se encargan de generar contenidos dinámicos para él. A nivel técnico, un *portlet* empieza por una clase java que implementa la clase interfaz *javax.portlet.Portlet*. Los *portlet* solo tienen sentido cuando están contenidos dentro de un contenedor de *portlet* que suele ser el portal, e interactúan con los usuarios web con un paradigma de peticiones y respuestas (*request/response*).

A diferencia de los *servlet*, los *portlet* no pueden ser llamados de forma directa, es decir, de un *servlet* se puede escribir su dirección web pero de un *portlet* solo se podrá tener la dirección web de la página que lo contiene, pero no una petición directa sobre él. Otra diferencia es que los *servlets* pueden generar páginas o documentos completos, mientras los *portlets* solo generan una parte del contenido.

Los portlets deberán hacer uso de la tecnología SPA.

A continuación, se describen qué partes interesadas están implicadas en el proyecto que se va a desarrollar.

1.1.1. Desarrollador

Se encarga de implementar las diferentes partes del proyecto y de validar el funcionamiento del mismo. Además es quien estudiará el comportamiento del sistema una vez integrado en *Liferay*.

1.1.2. Beneficiarios

Al finalizar el proyecto, se tendrá una página web SPA programada con *AngularJS* e integrada en *Liferay*. Cualquier persona o empresa interesada tendrá acceso al código y podrá ver cómo se ha realizado esta integración, para así poder implementar él mismo un SPA en *Liferay* o para realizar mejoras a dicho código. Los usuarios de estas empresas serán quienes más se beneficiarán.

También se beneficiará en gran medida el desarrollador, ya que aprenderá, en mayor o menor medida, a utilizar las tecnologías *AngularJS, Node* [10] y *Liferay*.

Al ser un tema de actualidad y al cual se pretende dar más importancia en un futuro cercano, diferentes empresas pueden tener interés en el proyecto, ya sea simplemente en la parte de la creación de aplicaciones SPA o en su integración en *Liferay*.

1.2. Motivación personal

Este proyecto nace como propuesta de la empresa en la cual trabajo a través de un convenio de colaboración con la FIB.

La principal motivación para el desarrollo de este proyecto era poderla realizar en una empresa, innovando, estudiando y aprendiendo a utilizar nuevas tecnologías. Así pues, la oportunidad de realizar este proyecto dentro de una empresa, ofrecía la oportunidad tanto de aprender nuevas tecnologías como de acercarse al mercado laboral y conocer cómo se trabaja en un entorno profesional real.

Además, el interés principal del desarrollador dentro de la informática es el desarrollo web. Así, esta ha sido una oportunidad para realizar un proyecto web utilizando una tecnología interesante y que cada vez cobra más importancia.

1.3. Estado del arte

Al tratarse el proyecto de una forma diferente a crear una página web a la convencional no hay productos en sí que se puedan estudiar. Sí que existen páginas web de este tipo, por ejemplo Gmail, Google Maps, Google Drive o Twitter. En las que poder basarse, pero no se tendrán muy en cuenta ya que para la creación del SPA que nos ocupa lo necesario es estudiar su código e implementación en *Liferay* y de este hay más bien escasos ejemplos.

1.4. Formulación del problema y objetivos

Habiendo presentado los conceptos pertinentes, en esta sección se explica detalladamente en qué consiste el problema a resolver y se exponen los objetivos principales del proyecto.

En relación al contexto del trabajo, este proyecto se centra en buscar una solución al hecho de que los usuarios son cada vez más impacientes y quieren ver el resultado de sus operaciones cuanto antes. En el caso que nos ocupa, la navegación de una página web. De este hecho nace la necesidad del uso de SPA, ya que nos permitirá que, tras una carga inicial más lenta de lo habitual, el usuario pueda navegar por los diferentes estados de una página web apenas notando los tiempos de carga.

Esto es de agradecer, ya que con esta forma de crear webs el usuario no sentirá que pierde el tiempo esperando, por ejemplo, a la recarga de una vista que ya ha visitado o a la carga de una nueva vista.

Además como ya se ha comentado, de un *portlet* solo se podrá tener la dirección web de la página que lo contiene, pero no una petición directa sobre él. Con lo cual cada vez que se realiza una interacción con un *portlet* no solamente este deberá recargarse sino también el resto de *portlets* que ocupen la página.

Así pues, el uso de la tecnología SPA repercutiría de forma muy positiva en este entorno. Por tal de solucionar el problema planteado se han establecido los siguientes objetivos:

- Creación de una web SPA

Crear una web SPA con *AngularJS*, aprovechando que el *framework* está destinado a la creación y mantención de webs SPA.

- Integración de la web en el CMS Liferay

Integrar la web resultante en el CMS Liferay y comprobar que funciona de la forma esperada.

No menos importante que estos objetivos es conseguir llevar adelante un proyecto real, acotado temporalmente y de acuerdo a unas expectativas, utilizando metodologías actuales y teniendo en cuenta técnicas y conocimientos adquiridos a lo largo de la carrera.

1.5. Alcance

Para conseguir la integración de una página web SPA en el portal de gestión de contenidos *Liferay* [11], primero se han investigado qué tecnologías se utilizan para la creación de dichas páginas, cual es la que ofrece una mejor proyección y se ajusta mejor a las características ofrecidas por *Liferay*. Se ha escogido como tecnología a utilizar *AngularJS* [12], así pues el próximo paso será estudiar esta tecnología y comprender cómo funciona para poder crear la página web deseada

Una vez obtenidos los conocimientos necesarios para crear SPAs con *AngularJS*, se procederá a investigar cómo implementar estas aplicaciones en *Liferay*. Esta comprensión se logra adquiriendo los conocimientos necesarios sobre técnicas de SPA y estudiando las bases teóricas del funcionamiento de ambas tecnologías.

El resultado final del proyecto será la integración de una web SPA que habrá sido creada con *AngularJS*, un *framework* ampliamente utilizado [13] para la creación de SPA. La web resultante mostrará los resultados de la temporada 2016 de Fórmula 1 junto a otros datos como resultados de los pilotos, puntos de los equipos… organizados en diferentes vistas.

1.6. Limitaciones del proyecto y riesgos

En este proyecto se deben tener en cuenta los riesgos involucrados en el proceso de desarrollo para poder detectarlos cuanto antes. Así para el proyecto se han detectado los siguientes riesgos:

- Desconocimiento de las tecnologías a utilizar: Al tratarse tanto AngularJS como Liferay de tecnologías desconocidas para el autor de este proyecto, se considera este hecho un riesgo importante para el devenir del proyecto. Para evitar

este riesgo existen varios tutoriales y documentación, tanto oficial como extraoficial, que puede ser consultada para comprender y ampliar el conocimiento de estas tecnologías.

- Control del tiempo. Un riesgo muy importante es el excederse en el plazo de tiempo establecido para el proyecto. Para evitar este riesgo se hará un control exhaustivo para que se cumpla la planificación establecida.

- Producto poco eficiente. Al tratarse de un producto final destinado a cualquier tipo de usuario, el producto resultante debe tener un buen rendimiento en cuanto a velocidad de uso y más aún cuando este está pensado para una navegación web cercana a una aplicación de escritorio. Para evitar este riesgo, durante el desarrollo del proyecto se tendrá en cuenta la velocidad de ejecución y la correcta programación de las partes que influyen críticamente en este aspecto.

- Producto poco atractivo. Al ser una de las finalidades del proyecto la investigación e implementación de SPA en Liferay no se persigue la creación de una web donde el usuario pueda introducir información, sino que la web simplemente muestra contenido relacionado con la temporada 2016 de Fórmula 1 que el usuario puede consultar. Esto será suficiente para comprobar los beneficios del uso de webs SPA. Es por esto que la web puede resultar poco atractiva al contener únicamente información acerca de la temporada de Fórmula 1. Para evitar este riesgo la web muestra contenido interesante en cuanto a la temporada de Fórmula 1 aunque este puede no ser de gran interés para ciertos usuarios.

1.7. Métodos de validación

Al seguir la metodología *Scrum* (ver punto 2.1.) no es necesario definir criterios de aceptación para la totalidad del proyecto, ya que la propia dinámica ofrece un fuerte control del producto por parte del negocio.

Uno de los mejores aspectos de esta metodología es la gran cohesión que crea entre todos los miembros involucrados en el proyecto. En general siempre hay una misma visión del producto que se está creando y el negocio puede ir perfilando el producto durante el transcurso de los *sprints*.

No se definen criterios de aceptación para todo el producto, pero sí que se definen criterios de aceptación para las diferentes tareas. Gracias a que las tareas tienen una granularidad muy pequeña los criterios de aceptación pueden ser muy precisos y concisos.

Los encargados de validar los criterios de aceptación de las tareas son los miembros de negocio, durante las reuniones de demostración.

2. Gestión del proyecto

2.1. Metodología y rigor

La mayor dificultad de implementación viene por parte de la integración del SPA creado con *AngularJS* a *Liferay*. Por ello, la idea para es que su integración se dividida en diferentes etapas, añadiendo funcionalidades y alcanzando pequeñas metas en cada etapa. Al finalizar cada una de las etapas se comprobará que la implementación funcione correctamente dentro de sus capacidades.

Es por esto que, para el desarrollo de este proyecto, se necesita una metodología ágil que permita introducir cambios en el proyecto a medida que este vaya avanzando. Esto se debe a que a medida que el proyecto vaya avanzando se pueden ir encontrando problemas de integración que requieran redefinir tareas ya terminadas o que estén aún por hacer.

2.1.1. Metodología SCRUM

Dentro del paradigma de metodologías ágiles se elige utilizar la metodología SCRUM, ya que esta metodología es flexible y ágil, características clave para el éxito de un proyecto. Así esta metodología es la que mejor se amolda a las necesidades del proyecto, puesto que en entornos como el desarrollo web los proyectos se ven envueltos en cambios constantes.

2.1.1.1. Roles

En SCRUM el equipo se centra en construir un software de calidad. La gestión del proyecto se focaliza en definir las características que debe tener el producto a construir. Los roles del equipo SCRUM son los siguientes:

- SCRUM master: Persona encargada de liderar el grupo para que este cumpla las reglas y procesos de la metodología. Gestiona los posibles problemas con los que se tope el proyecto y trabaja junto al *Product Owner* con el objetivo de maximizar el ROI (*Return On Investment*, entregar un valor superior al dinero invertido).

- Product Owner: Es el representante de los accionistas y clientes que usarán el software. Está focalizado en la parte de negocio y es el responsable del ROI así como de trasladar la visión del proyecto al equipo, añadiendo historias al *Product Backlog* y priorizándolas según crea conveniente.

- Team: Grupo de profesionales que se encargan de desarrollar el proyecto de forma conjunta y que se encargan de completar las historias a las que se comprometen al inicio de cada sprint.

2.1.1.2. Proceso

El proceso SCRUM se desarrolla de forma iterativa e incremental. SCRUM está basado en iteraciones que reciben el nombre de *sprint* y tiene una duración determinada. Un sprint está formado por un conjunto de tareas que se han de completar para cumplir sus objetivos. En cada sprint se desarrollan completamente una o más funcionalidades, tras cada *sprint* se obtiene una nueva versión del software con prestaciones listas para ser utilizadas. Los requisitos durante los *sprints* son intocables, al inicio de cada nuevo sprint se ajustan las funcionalidades ya finalizadas y/o se añaden nuevas, priorizando siempre aquellas que aporten mayor valor de negocio.

Las iteraciones corta permiten obtener *feedback* continuo de los usuarios y el cliente. Esto permite que el producto cambie continuamente en función de la reacción de los usuarios y del cliente. Esta metodología acepta y propicia la aparición de nuevos requisitos, así como la modificación de requisitos ya existentes. Estos enriquecen el proyecto y garantizan el éxito del proyecto.

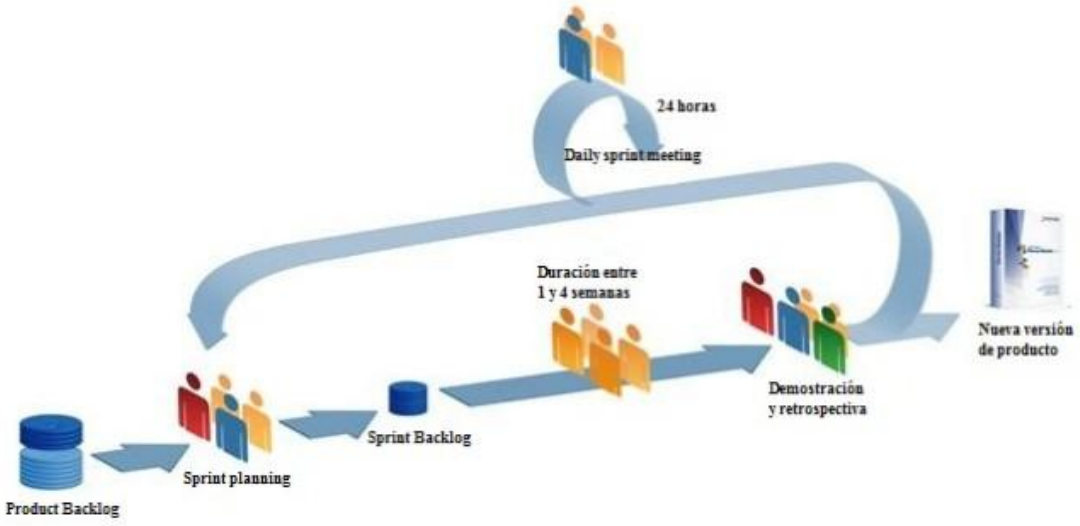


Ilustración 1- Ciclo de vida SCRUM

El ciclo de vida de la metodología SCRUM es el siguiente:

- Product Backlog: Conjunto de requisitos iniciales denominados historias, descritos en lenguaje no técnico y ordenados según su valor de negocio. Estos requisitos y prioridades se van revisando y modificando a intervalos regulares a medida que avanza el proyecto.

- Sprint Planning: Reunión en la que el *Product Owner* presenta las historias del *Product Backlog* a los miembros involucrados del proyecto (negocio y equipo desarrollador) según su orden de prioridad. En este momento el equipo se compromete a realizar una cierta cantidad de historias durante ese sprint. Estas historias se ponen en común, para verificar que todos los miembros entienden lo mismo para cada una de las tareas y el equipo desarrollador valora las tareas.

A partir de esta valoración y teniendo en cuenta el orden de prioridad establecido por el negocio se define una línea de corte para establecer que tareas entran en el *sprint*. Estas tareas se colocan en el *sprint backlog.*

- Sprint: El objetivo principal de esta división en iteraciones es la de entregar el producto al cliente o usuario final lo más pronto posible. De esta manera se pueden detectar los problemas rápidamente y se pueden refinar los requisitos para maximizar las garantías de éxito. Iteración de duración prefijada, normalmente entre 2 y 4 semanas, durante la cual el equipo trabaja en implementar las diferentes historias que tiene asignadas del *Product Backlog* en software operativo. Durante el sprint las historias permanecen invariables para así tener unos objetivos claros.

- Sprint Backlog: Lista de tareas a realizar para llevar a cabo las historias del *sprint*.

- Daily sprint meeting: Reunión diaria, que normalmente tiene una duración de 15 minutos, en la que le equipo comenta qué hizo el día anterior, qué hará hoy y que posibles problemas se pueden encontrar, para así poder trabajar de forma coordinada. Así, al comentar cualquier elemento bloqueante que pueda interferir con alguna tarea del *sprint* se puede evitada o reducir al mínimo las consecuencias de este.

- Demo y retrospectiva: Reunión que se realiza al final de cada sprint en la cual el equipo muestra el estado actual del producto mediante una demostración. Una vez mostrada la demo, en la retrospectiva, el equipo analiza qué se hizo bien, qué se podría mejorar y como conseguirlo. Llegados a este punto se vuelve al *sprint planning* hasta que el producto esté en condiciones de ser lanzado como un producto final.

2.1.2. Control de versiones

El control de versiones de este proyecto se ha realizado con *Git*. *Git* es un sistema distribuido de control de versiones. Cada miembro del equipo trabaja con su repositorio privado en su máquina. Cuando un miembro quiere sincronizar el trabajo realizado, envía los cambios al repositorio compartido. Esta acción se denomina *push*.

El sistema *Git* se encarga de realizar la mayor parte de la mezcla de los cambios. Como la mayoría de las herramientas de control de versiones, *Git* no puede mezclar dos versiones de dos repositorios que hayan realizado cambios en la misma parte del código. En estos casos, *Git* pide al usuario que realice los cambios a mano.

El repositorio compartido el privado y tiene alojado el cliente en sus servidores. Todos los miembros del equipo pueden acceder desde cualquier lugar.

2.2. Planificación temporal inicial

2.2.1. Planificación general

El proyecto tiene una duración estimada de 5 meses, siendo la fecha de inicio el día 25 de Enero y con fecha límite el 20 de Junio, contando con una margen de 13 días anteriores a la presentación para posibles desviaciones que puedan surgir en la planificación temporal, para así poder solventar los problemas antes de realizar la defensa del proyecto sobre el día 30 de Junio.

Tiene una carga de trabajo estimada de 115 días (460 horas), lo que engloba la realización de la totalidad de las tareas a realizar.

2.2.1.1. Recursos

Los recursos previstos para la realización de este proyecto son:

- Recursos personales: una persona, con una dedicación de 20 horas semanales durante todo el periodo de desarrollo del proyecto.

- Recursos materiales:

Recursos	Tipo	Finalidad
Ordenador portátil Dell Inter Core i5-2520M, 2.5 Ghz, 4GB RAM, Windows 10 64 bits	Herramienta de desarrollo	Desarrollar la aplicación web y la documentación
Liferay 6.2	Portal de gestión de contenidos	Gestionar los contenidos de la aplicación
AngularJS	Framework JavaScript	Creación de la SPA
Node	Entorno en tiempo de ejecución	Realizar pruebas en local de AngularJS
JetBrains WebStorm 11.0.3	Herramienta de desarrollo	JavaScript IDE para desarrollo de cliente y servidor con Node.js
Sublime Text 2 v2.02	Herramienta de desarrollo	Editor de texto para desarrollar la aplicación web
Microsoft Office Word 2013 Microsoft Office PowerPoint 2013 Microsoft Office Excel 2013	Herramienta de desarrollo	Realizar la documentación del proyecto
Git v1.9.5	Herramienta de control	Control de versiones del repositorio del código fuente

Tabla 1- Recursos materiales

2.2.1.2. Plan de acción y valoración de alternativas

Al utilizar metodologías ágiles, en caso de que aparezcan problemas durante el transcurso del proyecto se espera que sea sencillo modificar la planificación para adaptarse a los problemas encontrados. Si una fase termina con antelación a la fecha escogida se procederá a comenzar la siguiente.

Si se producen desviaciones temporales que afecten a las iteraciones del proyecto o a su fecha de entrega, se cuenta con un margen de 13 días entre la fecha prevista de entrega del proyecto y la exposición de este para poder conseguir mejoras sustanciales en la calidad del proyecto o para completar tareas que se hayan visto atrasadas.

Se pueden producir desviaciones temporales en las iteraciones de desarrollo, así como en el tiempo dedicado a aprender el funcionamiento de las diferentes herramientas a utilizar. Estas desviaciones pueden ser, o bien por exceso, o bien por defecto; se pueden dedicar más horas de las estimadas a hacer una tarea o bien, por el contrario, se puede dedicar menos. Si en uno de estos momentos existe una desviación de más de 5 días en la realización de una tarea se aplicarán medidas correctivas y se hará una replanificación.

Las medidas correctivas que se podrían aplicar son la cancelación de la funcionalidad que se esté desarrollando, su simplificación de tal manera que sea más fácil de tratar o, si se trata de algo crítico, se podrían cancelar otras funcionalidades que no sean críticas ni perjudiquen a la calidad del producto para poder emplear este tiempo en la tarea crítica e importante.

Aunque no se espera que se alteren los requisitos del proyecto a lo largo del proceso, se planifica una iteración opcional, de mejoras, que se llevara a cabo si se cumplen los términos de tiempo establecidos en esta planificación inicial para las fases anteriores. De esta manera, se asegura que en caso de desviación respecto a esta planificación, se pueda entregar un producto acabado en la fecha de entrega prevista.

2.2.1.3. Consideraciones globales

Cabe destacar que este proyecto lo realizará una sola persona, así pues no se podrán hacer tareas en paralelo. En principio no se realizará ninguna tarea hasta terminar con la anterior, aunque es posible que se realicen cambios si una tarea se vuelve complicada o consume mucho más tiempo del esperado para llevarla a cabo.

2.2.2. Descripción de las tareas

2.2.2.1. Búsqueda y puesta en marcha

En la primera fase se buscará información sobre el dominio del proyecto, necesario para el desarrollo, y se preparará el entorno de trabajo. En esta búsqueda

se elegirán las tecnologías que se utilizarán para más tarde dedicar un tiempo al autoaprendizaje de estas tecnologías.

Esta fase no dispone de dependencia de precedencia.

2.2.2.2. Gestión del proyecto

En esta fase se definen los aspectos de cómo se realizará el proyecto, su alcance, objetivos finales, planificación temporal, económica, etc.

Una vez esta fase termine, los aspectos no relacionados directamente con el proyecto quedarán definidos y se comenzará con la parte técnica de este.

La dependencia de precedencia de esta fase es la de búsqueda y puesta en marcha.

2.2.2.3. Desarrollo de aplicación con AngularJS

En esta fase se desarrollará una primera versión con *AngularJS* de lo que se pretende integrar en *Liferay*. En esta fase se aprenderá a utilizar *AngularJS* y se realizará una primera toma de contacto con lo que más tarde será la aplicación final.

En esta fase se desarrollarán las vistas y lógicas necesarias para *Drivers*, *Driver*, *Teams*, *Team*, *Races* y *Grand Prix*. Se implementará una pestaña de navegación entre las vistas Drivers, *Teams* y *Races* y la navegación de las diferentes vistas. Por último se implementará un *breadcrumb* y un buscador para las vistas *Drivers*, *Teams* y *Races*.

La dependencia de precedencia de esta fase es la de gestión de proyecto.

2.2.2.4. Integración aplicación AngularJS en Liferay

En esta fase primeramente se estudiará cómo integrar la aplicación realizada en la anterior iteración y una vez se tengan los conocimientos necesarios se procederá a realizar pruebas de integración y la propia integración. En caso de encontrarse con problemas de integración con alguna funcionalidad y que no parezca que tenga una solución pronta o sencilla se podría dejar esto para la fase de mejoras.

La dependencia de precedencia de esta fase es la de desarrollo de aplicación con *AngularJS*.

2.2.2.5. Mejoras

Última fase del desarrollo del proyecto que contempla mejoras en el funcionamiento, en el diseño o la adición de nuevas o problemáticas funcionalidades.

Las mejoras iniciales que se plantean son el cacheo de las llamadas HTTP y el cambio del ruteo base de *AngularJS* por otro más completo.

La dependencia de precedencia de esta fase es la de integración de la aplicación *AngularJS* en *Liferay*.

2.2.2.6. Documentación y presentación

En esta fase se terminará la documentación y se revisará que sea correcta. Dicha documentación se irá realizando a lo largo de las anteriores fases. Además, se preparará la defensa del proyecto, que tendrá lugar sobre el día 30 de Junio.

La dependencia de precedencia de esta fase es la de mejoras.

2.2.3. Calendario

2.2.3.1. Estimación de horas

Tarea	Duración	Horas
Búsqueda y puesta en marcha	11/01/16 – 18/02/16	116 horas
Gestión del proyecto	22/02/16 – 17/03/16	76 horas
Desarrollo en AngularJS	18/06/16 – 13/04/16	76 horas
Integración en Liferay	14/04/16 – 10/05/16	76 horas
Mejoras	11/05/16 – 24/05/16	40 horas
Documentación y presentación	25/05/16 – 20/06/16	76 horas
		Total: 460 horas

Tabla 2- Estimación de horas

2.2.3.2. Diagrama de Gantt

	Nombre	Duración	Inicio	Fin	Predecesoras	Recursos
1	**1. Búsqueda y puesta en marcha**	**29d**	**11/01/2016**	**18/02/2016**		
2	1.1. Búsqueda y puesta en marcha	10d	11/01/2016	22/01/2016		
3	1.2. Apredizaje autónomo	20d	22/01/2016	18/02/2016		
4	**2. Gestión del proyecto**	**19d**	**22/02/2016**	**17/03/2016**	**1**	
5	2.1. Definición del alcance y contextualización	6d	22/02/2016	29/02/2016		
6	2.2. Planificación temporal	2d	01/03/2016	02/03/2016		
7	2.3. Gestión económica y sostenibilidad	2d	02/03/2016	03/03/2016		
8	2.4. Presentación preliminar	2d	04/03/2016	07/03/2016		
9	2.5. Pliego de condiciones	3d	08/03/2016	10/03/2016		
10	2.6. Presentación y documento final	5d	11/03/2016	17/03/2016		
11	**3. Desarrollo en AngularJS**	**19d**	**18/03/2016**	**13/04/2016**	**4**	
12	3.1. Análisis de requisitos	2d	18/03/2016	21/03/2016		
13	3.2. Diseño y especificación	3d	22/03/2016	24/03/2016		
14	3.3. Implementación y pruebas	14d	25/03/2016	13/04/2016		
15	**4. Integración en Liferay**	**19d**	**14/04/2016**	**10/05/2016**	**11**	
16	4.1. Análisis de requisitos	2d	14/04/2016	15/04/2016		
17	4.2. Diseño y especificación	3d	18/04/2016	20/04/2016		
18	4.3. Implementación y pruebas	14d	21/04/2016	10/05/2016		
19	**5. Mejoras**	**10d**	**11/05/2016**	**24/05/2016**	**15**	
20	5.1. Análisis y diseño	2d	11/05/2016	12/05/2016		
21	5.2. Implementación y pruebas	8d	13/05/2016	24/05/2016		
22	**6. Documentación y presentación**	**19d**	**25/05/2016**	**20/06/2016**	**19**	
23	6.1. Redacción de la memória	14d	25/05/2016	13/06/2016		
24	6.2. Preparación presentación oral	5d	14/06/2016	20/06/2016		

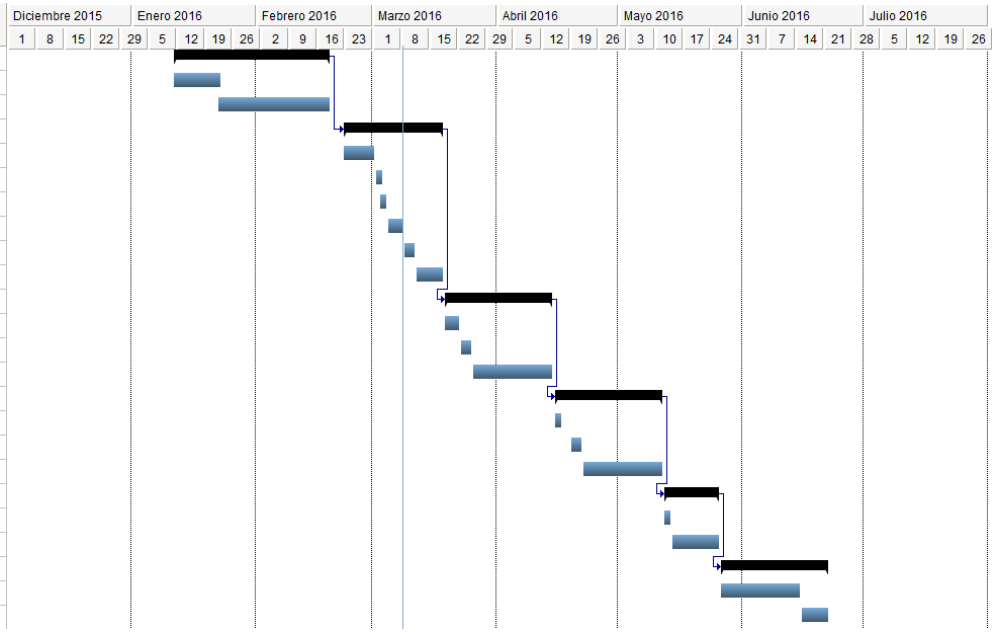


Tabla 3- Diagrama de Gantt

2.3. Planificación temporal final

La planificación final de este proyecto se corresponde con la planificación inicial establecida en la sección anterior. No ha habido desviaciones de la planificación inicial gracias al uso de la metodología ágil.

Se han realizado todos los *sprints* establecidos y se ha cumplido con el plazo de documentación y preparación,

Se han realizado las tareas establecidas al inicio del proyecto y también los considerados como mejoras. Además este último *sprint* ha servido para resolver algunos problemas detectados.

3. Gestión económica y sostenibilidad

3.1. Identificación y estimación de los costes

La identificación y estimación de los costes del proyecto es el paso previo a la elaboración del presupuesto, estos costes están directamente relacionados con los recursos descritos en el apartado de la planificación temporal. Seguidamente se identifican los recursos humanos, recursos hardware y software, gastos generales, imprevistos y contingencias y se hace una estimación de los costes asociados tomando como referencia el la duración del diagrama de Gantt y que cada día de trabajo en el proyecto tiene una duración de 4 horas.

3.1.1. Recursos humanos

La remuneración asociada a cada rol se corresponde a la media del precio por hora y la menor experiencia en años establecida por el informe de *Page Personnel* [14] del año 2016.

Rol	Remuneración (€/h)
Jefe de proyecto	19.5
Analista funcional	17.7
Diseñador	18.2
Programador	10.1
Responsable de pruebas	12.5

Tabla 4- Roles

Actividad	Horas	Rol	Precio de mercado (€/h)	Coste de mercado (€)
Búsqueda y puesta en marcha	40	Programador	10.1	404
Aprendizaje autónomo	80	Programador	10.1	808
GEP	76	Jefe de proyecto	19.5	1482
Sprint1 – Análisis de requisitos	8	Analista funcional	17.7	141.6
Sprint1 – Diseño y especificación	12	Diseñador	18.2	218.4
Sprint1 – Implementación y pruebas	56	Programador 80% / Responsable	10.1 / 12.5	452.5 / 140

		de pruebas 20%		
Sprint2 - Análisis de requisitos	8	Analista funcional	17.7	141.6
Sprint2 – Diseño y especificación	12	Diseñador	18.2	218.4
Sprint 2 – Implementación y pruebas	56	Programador 80% / Responsable de pruebas 20%	10.1 / 12.5	452.5 / 140
Mejoras- Análisis y diseño	8	Analista funcional 40% / Diseñador 60%	17.7 / 18.2	56.6 / 87.4
Mejoras – Implementación y pruebas	30	Programador 80% / Responsable de pruebas 20%	10.1 / 12.5	242.4 / 75
Documentación y presentación	74	Jefe de proyecto	19.5	1443
Total	**460**			**6503.4**

Tabla 5- Coste actividades

3.1.2. Recursos hardware

La estimación de vida útil de los recursos hardware está estimada en 4 años. Para establecer el coste de amortización por hora se toma como referencia que un año de vida útil son 249 días hábiles a un ritmo de trabajo de 4 horas diarias.

Coste de amortización = precio / (4 años * (249 días * 4 horas/día))

Producto	Precio (€)	Horas estimadas	Coste de amortización (€/h)	Coste estimado (€)
Portátil	750	460	0.19	141.19
Total estimado				**141.19**

Tabla 6- Recursos hardware

3.1.3. Recursos software

Los recursos software se han conseguido mayormente mediante licencias que la FIB tiene con empresas o utilizando software libre, la única excepción es el CMS *Liferay* (30.000 €/año), aunque al ser este utilizado por toda la empresa solamente se tendrá en cuenta un 1% de su precio anual.

Los recursos software se renovarán cada 3 años.

Coste de amortización = precio / (3 años * (249 días * 4 horas/día))

Producto	Precio (€)	Horas estimadas	Coste de amortización (€/h)	Coste estimado (€)
Liferay	300	460	0.10	30.12
AngularJS	0	No trasciende	0	0
Node	0	No trasciende	0	0
JetBrains WebStorm v.1.0.3	0	No trasciende	0	0
Sublime Text	0	No trasciende	0	0
Windows 7	0	No trasciende	0	0
Git	0	No trasciende	0	0
Microsoft Office Word 2013 Microsoft Office PowerPoint 2013 Microsoft Office Excel 2013	0	No trasciende	0	0
Total estimado	**0**		**0**	**30.12**

Tabla 7- Recursos software

3.1.4. Gastos generales

· Conexión a internet de 2.45 MB de bajada y 20 MB de subida con un precio de 36.18 €/mes, aunque al ser esta utilizada por toda la empresa solamente se tendrá en cuenta un 1% de su precio.

· Desplazamiento en transporte público utilizando una tarjeta T-50/30 de una zona.

· Al final del proyecto se deberá hacer una entrega en papel de la memoria a cada uno de los tres miembros del tribunal y una al director. Supondremos que dicha memoria tendrá una extensión de 200 páginas aproximadamente, con un coste de 0,05 €/hoja.

Descripción	Unidades	Precio	Dedicación	Coste estimado (€)
Conexión a internet	6 meses	0.36 €/mes	100%	2.16
Desplazamiento	6 meses	42.50 €/mes	100%	255
Impresiones	800 páginas	0.05 €/pág.	100%	40.00
Total				**297.16**

Tabla 8- Gastos generales

3.1.5. Imprevistos

Aunque en la planificación del proyecto el día 20 de Junio está marcado como fecha de finalización del proyecto se podría dar el caso de que debido a reajustes en la planificación se alargase esta fecha. Para contemplar la posibilidad de que se dé este caso se asume un riesgo de que suceda del 40% y se asigna que el coste destinado a cada día de trabajo fuera de la planificación sea igual al coste medio de la planificación del proyecto entre el número de días del proyecto.

Coste por día imprevisto de planificación = 0.4 riesgo * ((coste recursos humanos + coste hardware + coste gastos generales) / (duración en horas del proyecto * 4 horas de trabajo diarias)) * 13 días = 19.7 = * 13 días = 256.2

3.1.6. Contingencias

Como medida de contingencia se establece un margen del 7% sobre el coste total del proyecto.

3.1.7. Coste total

Recurso	Coste estimado (€)
Recursos humanos	6503.4
Recursos hardware	141.9
Recursos software	30.12
Gastos generales	297.16
Imprevistos	256.2
Coste parcial	**7228.8**
Contingencias	7%
Coste total	**7734.8**

Tabla 9- Coste total

3.2. Viabilidad económica

Teniendo en cuenta el presupuesto necesario para realizar el proyecto y que en este caso particular la parte hardware ya está amortizada y todo el software se ha obtenido gratuitamente mediante convenios de la UPC con diversas empresas o a través de software libre, los gastos finales se reducen a los recursos humanos, y al

ser un proyecto de TFG de 15 créditos ECTS las horas están dentro de los límites calculados en el apartado de planificación temporal.

Por tanto, el proyecto es viable tanto temporal como económicamente.

3.3. Sostenibilidad y compromiso social

A continuación presentamos el estudio sobre sostenibilidad que se ha hecho referente al proyecto. En la siguiente tabla se presentan los resultados, que se explican más adelante, en forma de matriz propuesta en la guía de la asignatura GEP (solo la parte de planificación).

Sostenibilidad	Económica	Social	Ambiental	Total
Planificación	Viabilidad económica	Mejora de la calidad de vida	Análisis de recursos	
Valoración	8	7	7	22

Tabla 10- Sostenibilidad

El resultado total, de 22, se considera un buen resultado dado que el proyecto no está pensado para mejorar de una manera drástica la calidad de vida del usuario, no afecta directamente a la huella medioambiental y en cuestiones económicas obtiene una buena puntuación.

A continuación se explica de forma más detallada el porqué de las diferentes puntuaciones.

3.3.1. Económica

Para considerar al proyecto viable económicamente es básico realizar una evaluación de costes, tanto materiales como de recursos humanos (el cual se encuentra en apartados anteriores de este documento). También se han establecido protocolos para controlar y ajustar el presupuesto durante la realización del proyecto.

Se debe tener en cuenta que en este documento se calcula un precio teórico, el precio real resultaría más barato debido a que el material que se utiliza para desarrollar el mismo ya estaba disponible antes de comenzar el proyecto y ha sido amortizado. También hay que tener en cuenta que debido a la naturaleza del proyecto no se tiene intención de lanzarlo al mercado, aun así este sería un proyecto con un precio competitivo ya que la mayoría de los gastos se encuentran en los sueldos de los distintos roles que participan en la realización de este.

Así pues, en cuestión de sostenibilidad económica, este proyecto merece una valoración de 8. No obtiene una nota mayor debido a que el proyecto se podría finalizar en menos tiempo, pero eso solo sería posible o bien aumentando el personal del proyecto o realizándolo a tiempo completo, es decir, 8 horas diarias, hecho que resulta imposible debido a los otros compromisos del desarrollador.

3.3.2. Social

Puesto que cada vez los usuarios son más impacientes y quieren ver la respuesta a sus acciones cuanto antes, la creación y utilización de webs SPA es algo que mejora ligeramente su calidad de vida. Esto es debido a que esta mejora en el tiempo de carga de las vistas de la web no es algo que vaya a influir de una forma drástica en la vida del usuario o algo sin lo que el usuario no pudiera vivir, es simplemente una comodidad más, una pequeña mejora en la forma de interactuar con páginas web.

Debido a que la tecnología que se utiliza en este proyecto ya existe en el mercado y la innovación del mismo es su integración en el CMS *Liferay* no existe un colectivo que se pueda ver perjudicado por el mismo, más bien al contrario. Por esto, en cuanto a sostenibilidad social, se valora al proyecto con un 7, ya que los beneficios obtenidos del uso de esta tecnología, aunque importantes e interesantes, no son esenciales para la navegación web, sino que son un paso más allá para conseguir una navegación más fluida.

3.3.3. Ambiental

Como todos los recursos necesarios para la realización del proyecto ya estaban disponibles y se utilizaban antes del mismo, no será necesario comprar nuevos recursos. La antigüedad del portátil hace que su coste esté amortizado. Una vez terminado el proyecto, este podrá ser actualizado y mejorado con modificaciones o ser utilizado como plantilla a seguir para la implantación de sistemas con objetivos similares.

Al ser este proyecto una herramienta software, no requiere de ningún coste de fabricación ni genera contaminación. El hecho de que tampoco haya que preocuparse por el reciclaje, que el hardware está amortizado y que se trata de un producto software hacen que la huella medioambiental que este proyecto sea prácticamente nula, no afectará de una forma positiva ni negativa significativamente.

En cuestiones ambientales, este proyecto se puntúa con un 7.

3.5. Leyes y regulaciones

En la aplicación web resultante de este proyecto no se guarda información personal de los usuarios que interactúan con ella. Toda la información necesaria, si esta no se encuentra cacheada, se obtiene a través de una API cada vez que el usuario accede a la web. Por este motivo el sistema desarrollado en este trabajo queda fuera del alcance de la LOPD.

4. Especificación

En este apartado se definirán los requisitos funcionales y no funcionales del sistema.

4.1. Requisitos funcionales

Los requisitos funcionales definen una función del sistema de software o sus componentes. Una función es descrita como un conjunto de entradas, comportamientos y salidas. Los requisitos funcionales son los encargados de establecer los comportamientos del sistema.

En este proyecto se han definido los requisitos funcionales siguiendo el modelo de historias de usuario, ya que se ha seguido la metodología *scrum*. Cada una de las historias de usuarios define una funcionalidad del sistema.

4.1.1. Historias de usuario

La metodología *scrum* utiliza las historias de usuario para definir las diferentes funcionalidades del sistema. Estas historias de usuario están definidas por el cliente y puestas en común con todo el equipo durante los *sprint plannings*.

Cada historia de usuario se compone de un título que la identifica, el texto de la historia, un número de historia de usuario (que no implica orden ni prioridad), una cifra de prioridad y los criterios de aceptación.

Los criterios de aceptación son los criterios que ha de cumplir la funcionalidad a desarrollar para que sea aceptada como buena.

1. Ver clasificación por pilotos
Como usuario **quiero** ver la clasificación de la F1 según la puntuación obtenida por los pilotos **para poder** ver información de los pilotos.
Prioridad: 100
Criterios de aceptación:
· Quiero ver la clasificación de la F1 según la puntuación obtenida por los pilotos. · Por cada piloto quiero que se muestre su posición en la clasificación general, su nombre y apellido, el equipo para el que corre y los puntos que ha obtenido hasta el momento. · Quiero poder clicar sobre el nombre del piloto para poder ir a la página que muestra los resultados del piloto. · Quiero poder clicar sobre el nombre del equipo para poder ir a la página que muestra los resultados del equipo.

2. Ver resultados de un piloto
Como usuario **quiero** ver los resultados de un piloto **para poder** ver información de los resultados que ha obtenido en las diferentes carreras disputadas.
Prioridad: 80
Criterios de aceptación:
· Quiero ver los resultados de un piloto para las distintas carreras disputadas. · Por cada carrera disputada quiero que se muestre a que ronda pertenece, el nombre del Gran Premio, el equipo al que pertenece, la posición de salida en la parrilla, la posición final y la puntuación obtenida. · Quiero poder clicar sobre el nombre del Gran Premio para poder ir a la página que muestra los resultados de la carrera. · Quiero poder clicar sobre el nombre del equipo para poder ir a la página que muestra los resultados del equipo. · Quiero poder clicar sobre la biografía del piloto para poder ir a la página que muestra la biografía del piloto.

3. Ver clasificación por equipos
Como usuario **quiero** ver la clasificación de la F1 según la puntuación obtenida por los pilotos **para poder** ver información de los pilotos.
Prioridad: 100
Criterios de aceptación:
· Quiero ver la clasificación de la F1 según la puntuación obtenida por los equipos. · Por cada equipo quiero que se muestre su posición en la clasificación, su nombre, los detalles de equipo y los puntos que ha obtenido hasta el momento. · Quiero poder clicar sobre el nombre del equipo para poder ir a la página que muestra los resultados del equipo. · Quiero poder clicar sobre los detalles del equipo para poder ir a la página que muestra los detalles del equipo.

4. Ver resultados de un equipo
Como usuario **quiero** ver los resultados de un equipo **para poder** ver información de los resultados que ha obtenido en las diferentes carreras disputadas.
Prioridad: 80
Criterios de aceptación:
· Quiero ver los resultados de un equipo para las distintas carreras disputadas. · Por cada carrera disputada quiero que se muestre a que ronda pertenece, el nombre del Gran Premio, la posición en la clasificación general de sus distintos pilotos y la puntuación total del equipo. · Quiero poder clicar sobre el nombre del Gran Premio para poder ir a la página que muestra los resultados de la carrera. · Quiero poder clicar sobre la información del equipo para poder ir a la página que muestra la información del equipo.

5. Ver carreras
Como usuario **quiero** ver las distintas carreras **para poder** ver información de ellas.
Prioridad: 100
Criterios de aceptación:
· Quiero ver información de las distintas carreras. · Por cada carrera quiero se muestre a que ronda pertenece, el nombre del Gran Premio, el nombre del circuito y la fecha en que se disputa. · Quiero poder clicar sobre el nombre de Grandes Premios disputados para poder ir a la página que muestra los resultados de la carrera. · Quiero que no se pueda clicar sobre el nombre de Grandes Premios que no han sido disputados. · Quiero poder clicar sobre el nombre del circuito para poder ir a la página que muestra la información del circuito.

6. Ver resultados de una carrera
Como usuario **quiero** ver los resultados de una carrera **para poder** ver información de los resultados resultantes al finalizar la carrera.
Prioridad: 80
Criterios de aceptación:
· Quiero ver información de los resultados de una carrera finalizada. · Por cada piloto que haya corrido en la clasificación previa a la carrera quiero que se muestre su posición, su nombre y apellido, el equipo para el que corre, su Q1, Q2 y Q3. · Por cada piloto que haya disputado la carrera quiero que se muestre su posición, su nombre y apellido, el equipo para el que corre, las vueltas que ha realizado, la posición en la parrilla de salida, el tiempo en el que ha completado la carrera y los puntos que ha obtenido. · Quiero poder clicar sobre el nombre del piloto para poder ir a la página que muestra los resultados del piloto. · Quiero poder clicar sobre el nombre del equipo para poder ir a la página que muestra los resultados del equipo. · Quiero poder clicar sobre el nombre del circuito para poder ir a la página que muestra la información del circuito.

7. Ver biografía de un piloto
Como usuario **quiero** ver la biografía de un piloto **para poder** ver detalles del piloto.
Prioridad: 60
Criterios de aceptación:
· Quiero ver los detalles de la biografía de un piloto.

8. Ver información de un equipo
Como usuario **quiero** ver información de un equipo **para poder** ver detalles históricos de un equipo.
Prioridad: 60
Criterios de aceptación:
· Quiero ver los detalles históricos de un equipo.

9. Ver información de un circuito
Como usuario **quiero** ver información de un circuito **para poder** ver detalles del circuito.
Prioridad: 60
Criterios de aceptación:
· Quiero ver información de la carrera disputada en el circuito.

10. Realizar búsqueda en pilotos/equipos/carreras
Como usuario **quiero** realizar una búsqueda **para poder** ver solamente los resultados que me interesan.
Prioridad: 50
Criterios de aceptación:
· Quiero ver filtrados los resultados que me interesan según los datos introducidos en el campo "*Search*".

4.2. Requisitos no funcionales

Además de los requisitos funcionales del apartado anterior el sistema debe cumplir una serie de requisitos no funcionales. Un requisito no funcional es un requisito que especifica criterios que puedan usarse para juzgar la operación de un sistema en lugar de sus comportamientos específicos, ya que éstos corresponden a los requisitos funcionales. Por tanto, se refieren a todos los requisitos que no describen información a guardar, ni funciones a realizar, sino características de funcionamiento.

Estos requisitos están definidos siguiendo una plantilla simplificada que incluye la siguiente información: identificador, nombre del requisito y descripción del requisito.

A continuación se muestran los requisitos no funcionales del sistema:

Requisito no funcional #1
Nombre:
Usable
Descripción:
El sistema debe ser sencillo de utilizar e intuitivo para los usuarios a los que va dirigido.

Requisito no funcional #2
Nombre:
Cambiable
Descripción:
El sistema debe permitir la realización de cambios sin que estos afecten a las funcionalidades ya existentes.

Requisito no funcional #3
Nombre:
Extensible
Descripción:
El sistema debe ser fácilmente extensible. Debe ser posible tanto añadir fácilmente nuevas funcionalidades como modificar las ya existentes.

Requisito no funcional #4
Nombre:
Compatible con la última versión de navegadores.
Descripción:
El sistema debe ser compatible mínimo con la última versión de los navegadores Chrome, Firefox, Safari, Opera e Internet Explorer.

Requisito no funcional #5
Nombre:
Buena apariencia.
Descripción:
El sistema debe tener una buena apariencia mediante la aplicación de plantillas, unificación de colores y la aplicación de CSS, obteniendo así una interfaz con una buena apariencia.

Requisito no funcional #6
Nombre:
Accesibilidad y disponibilidad
Descripción:
El sistema debe ser programado siguiendo un diseño *responsive design* y obtener los datos necesarios de una API que ofrezca una buena estabilidad. Así pues el sistema podrá ser accedido en cualquier momento desde ordenador, móvil, tablet o un dispositivo compatible.

Requisito no funcional #7
Nombre:
Soporte
Descripción:
En el caso de que el CMS utilizado en el proyecto (Liferay 6.2 EE) presentase problemas, este dispone de un servicio propio de asistencia bajo licencia. Por otro lado si aparecieran problemas con el código generado en Angular el creador del mismo se encargará de su solución.

Requisito no funcional #8
Nombre:
Iconos convencionales
Descripción:
El sistema debe utilizar iconos convencionales e intuitivos que el usuario pueda identificar fácilmente.

Requisito no funcional #9
Nombre:
Rápido
Descripción:
El sistema debe ser implementado de forma correcta, obteniendo un sistema que cargue y muestre su contenido de forma fluida.

Requisito no funcional #10
Nombre:
Seguridad
Descripción:
El sistema debe ofrecer datos seguros, obteniendo estos de una API externa en la que se confía.

5. Diseño arquitectónico

La aplicación web consta de una web SPA generada con *AngularJS* que se comunica con un API externa mediante llamadas HTTP GET, y recibe datos en formato JSON de esta. Esta aplicación está integrada dentro de *Liferay*.

5.1. SPA

Como se puede observar en la ilustración 2, en las aplicaciones web tradicionales, cada vez que es necesario cargar una página, la aplicación llama al servidor, éste obtiene el HTML necesario, lo envía al navegador y éste se refresca para mostrar los nuevos datos.

En un SPA todos los códigos de HTML, JavaScript, y CSS se cargan de una vez o los recursos necesarios se cargan dinámicamente como lo requiera la página y se irán agregando, normalmente en respuesta a las acciones del usuario.

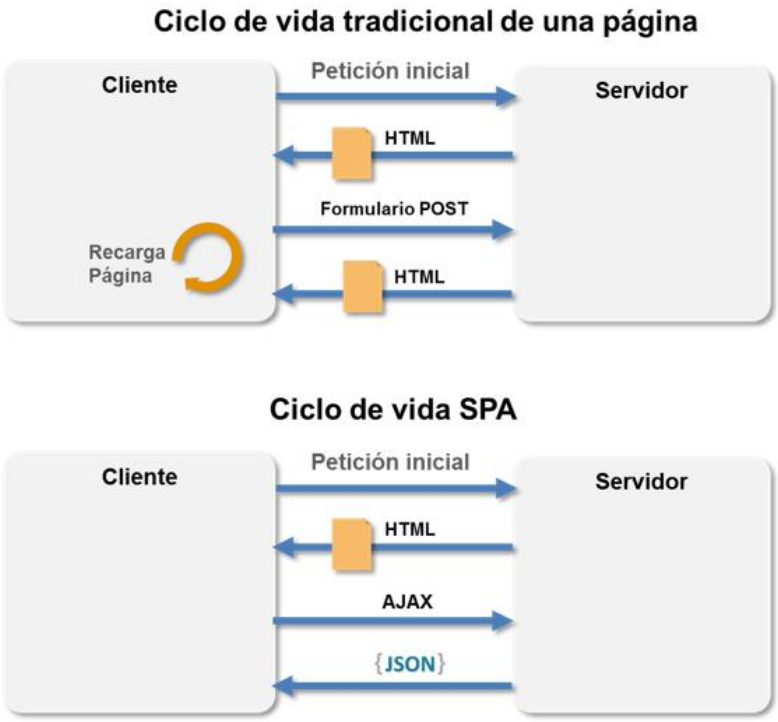


Ilustración 2- Ciclo vida web tradicional vs SPA

En contraposición, las SPAs, después de la primera llamada se cargan todos los códigos de HTML, JavaScript, y CSS o se cargan los recursos necesarios dinámicamente como lo requiera la página y estos se irán agregando, normalmente en respuesta a las acciones del usuario.

Así pues, ya no vuelve a interactuar directamente con el servidor, a partir de ese momento todas las interacciones con el servidor se producen mediante llamadas AJAX, las cuales devuelven los datos necesarios y el navegador no necesita refrescar la página, sino que esta se refresca dinámicamente con los datos devueltos

de la llamada. Mover el proceso de crear y gestionar vistas a la interfaz de usuario, lo desacopla del servidor.

Gracias a esto el navegador no necesita recargarse, en SPA las vistas no son paginas HTML completas, son porciones del DOM que forman la parte visible de la pantalla. Tras la primera carga todas las herramientas necesarias para crear y mostrar vistas están descargadas y listas para ser usadas. Si una vista nueva es necesaria, se genera localmente en el navegador y se adjunta dinámicamente al DOM a través de JavaScript. Como la lógica de presentación se encuentra mayormente en la parte del cliente, la tarea de combinar el HTML y los datos se mueve del servidor al navegador. El ruteo de las vistas, combinar datos con el *template* HTML y gestionar el ciclo de vida de las vista se delega al *framework*.

Existen ciertos inconvenientes provocados por el uso de SPA, por ejemplo al hacer uso de la tecnología JavaScript el usuario debe tenerlo activado para poder verse beneficiado y utilizar esta tecnología y hay que tener en cuenta que el navegador recibe una gran carga de trabajo. Además, bastante lógica de negocio se implementa en el lado del cliente, permitiendo que cualquier pueda leer el código JavaScript, por lo que hay que reforzar la seguridad y trabajar con minificadores js. En términos de SEO el crawler de Google funcioan correctamente, pero no es así para Bin, Yahoo o Yandex que no indexarán la web correctamente.

Por otra parte las grandes ventajas obtenidas del uso de SPA son que el tráfico de servidor se reduce al no tener que generar HTML en el este. Por esto mismo, servidor y *front-end* están separados, SPA se comunica con el servidor enviando recibiendo JSON utilizando AJAX, esto permite que ambas partes puedan ser desarrolladas y testeadas independientemente. Con todo esto obtenemos una web rápida que no necesita recargar cada vez que se interactúe con ella y que además puede funcionar *offline*.

5.2. Patrones de diseño

En esta sección se explican los patrones de diseño utilizados en el proyecto.

5.2.1. Modelo Vista VistaModelo

Angular ha sido definido como un *framework* Model-View-Whatever, es decir Modelo-Vista-*Lo-que-sea*, aunque tiene un marcado estilo Modelo-Vista-VstaModelo (MVVM).

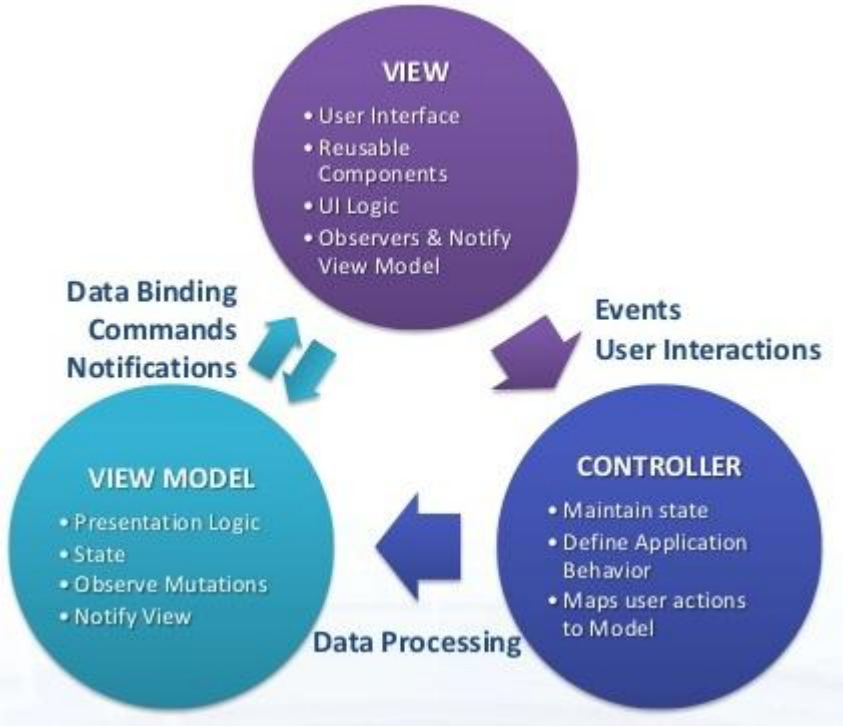


Ilustración 3- Angular MVVM

Aun teniendo un estilo MVVM, la aplicación se organiza entorno a controladores, que son los responsables de construir los *VistaModelos* que se enlazarán a las vistas. El papel que juegan los controladores es limitado en comparación al de los controladores típicos de la arquitectura MVC. En Angular el controlador puede construir un *VistaModelo* que luego toma el control completo de las operaciones realizadas en la vista, actuando como factorías de *VistaModelos*.

El enlace entre vistas y *VistaModelo* se realiza mediante un sistema de *databinding* declarativo bidireccional. Para ello a los elementos HTML se les añade *directivas* y se usa un sistema de plantillas parecido a *handlebars*, por ejemplo:

```
<div ng-controller="PhoneCtrl">
        <ul>
          <li ng-repeat="phone in phones">{{phone.name}}</li>
        </ul>
</div>
```

Hay que tener en cuenta que no existe una definición propia del *VistaModelo* sino que cada controlador tiene asociado un objeto $*scope* sobre el que construirlo. En el ejemplo anterior, el controlador *PhoneCtrl* añadiría a su objeto $*scope* una lista de teléfonos, cada uno de ellos con una propiedad *name* y eso sería lo que se enlazaría a la vista.

Además los $scopes (*VistaModelos*) están jerarquizados. Es decir, en una misma página se puede tener varior $scopes anidados unos con otros y los $*scope* hijos pueden acceder a las propiedades y las funciones de los $*scope* padre.

La raíz del árbol de $*scopes* es el $*rootScope*, este puede ser considerado como el *VistaModelo* global en el que colocar datos como, por ejemplo, el usuario *loggeado* actualmente o funciones de utilidad a las que sea interesante que pueda acceder cualquier otro $*scope*.

La jerarquía de controlador/$*scope* se corresponde con la jerarquía de elementos HTML de la página a la que están asociados. Ya que los controladores están asociados a elementos HTML, si el elemento asociado a un controlador contiene un elemento asociado a otro controlador, el $*scope* del segundo tendrá acceso a las propiedades del primero.

Para la navegación entre las páginas de la aplicación se utilizan rutas, que pueden ser asociadas a controladores y fragmentos de HTML. Al acceder a una ruta, se carga ese fragmento de HTML mediante Ajax y se inserta en la página actual en el elemento que esté marcado con la directiva ng-view (ui-view en caso de utilizar ui-router), por ejemplo:

```
<body ng-app="F1App">
  <div ng-view>
    <!-- Aquí se cargarán las vistas asociadas a las rutas de la
aplicación definida en el elemento body con la directiva ng-app-->
  </div>
</body>
```

5.2.2. Singleton

El patrón de diseño *singleton* se utiliza para tener una única instancia de una clase y esta debe ser fácilmente accesible por otros objetos de la aplicación. Normalmente se implementa un método que crea una nueva instancia de la clase en el caso de que esta no exista. En el caso de que sí exista retorna una referencia al objeto.

Este patrón de diseño se aplica a los servicios de *AngularJS*, ya que todos los servicios de Angular son *singletons*, así pues se crearán como mucho una vez y se cacheará la referencia para futuros usos.

5.2.3. Inyección de dependencias

La inyección de dependencias es un patrón de diseño orientado a objetos en el que se suministran objetos a una clase en lugar de ser la propia clase quien crea el

objeto. Este patrón de diseño se encarga de tratar como los componentes se relacionan con sus dependencias.

En el caso de *AngularJS*, el subsistema de inyección se encarga de la creación de componentes, la resolución de sus dependencias y de proveerlos a otros componentes bajo demanda. Así pues, a la hora de definir cualquier componente, ya sea un controlador, un servicio u otro componente cualquiera, se debe indicar de qué otros componentes dependen y angular se encargará de proporcionárselos a través de la función constructora.

Al ser todos los componentes registrados en angular *singleton* si varios componentes dependen de un mismo objeto todos recibirán la misma instancia del objeto. Gracias a esto se consigue fácilmente almacenar estado, ya que dos controladores que dependan del mismo servicio estarán utilizando el mismo objeto y, por tanto, podrán compartir información a través de él.

5.3. Protocolos

En este proyecto no se dispone de capa de datos ya que toda la información se obtiene de una API externa. Por este motivo los protocolos, estándares y especificaciones utilizados tienen una especial importancia. A continuación se definen los más importantes que se han utilizado.

5.3.1. REST

Técnica de arquitectura del software para sistemas hipermedia distribuidos como la web. Aunque este término también se emplea para describir cualquier interfaz web simple que utilice XML, JSON y HTTP sin las abstracciones adicionales de protocolos basados en patrones de intercambio de mensajes como el protocolo SOAP.

El mundo web ha escalado correctamente gracias a seguir un conjunto de diseños fundamentales, que según REST son:

· Protocolo cliente-servidor sin estado: cada mensaje HTTP contiene toda la información necesaria para resolver la petición, no depende de peticiones anteriores.

· Conjunto de operaciones que se aplican a todos los recursos de información: HTTP define las operaciones GET, POST, PUT y DELETE que se utilizan para obtener, modificar, añadir y borrar.

· Sintaxis universal para identificar los recursos: en un sistema REST se puede acceder a cada recurso únicamente a partir de su URI.

5.3.2. RESTful API

La RESTful API proporciona al sistema la información necesaria, para este proyecto los servicios obtienen la información demandada mediante el protocolo HTTP.

Ejemplo de petición GET:

Método GET para obtener detalles de un piloto con identificador driverId	
Dirección URL del recurso	http://ergast.com/api/f1/current/drivers/'+ driverId +'/driverStandings.json?callback=JSON_CALLBACK
Información sobre el recurso	Formato de respuesta JSONP
Respuesta ejemplo para Alonso	{"MRData":{"xmlns":"http:\/\/ergast.com\/mrd \/1.4","series":"f1","url":"http://ergast.com /api/f1/current/drivers/alonso/driverstandings. json","limit":"30","offset":"0","total":"1", "StandingsTable":{"season":"2016","driverId": "alonso","StandingsLists":[{"season":"2016", "round":"8","DriverStandings":[{"position":"13", "positionText":"13","points":"18","wins":"0", "Driver":{"driverId":"alonso","permanentNumber": "14","code":"ALO","url":"http:\/\/en.wikipedia. org\/wiki\/Fernando_Alonso","givenName": "Fernando","familyName":"Alonso","dateOfBirth": "1981-07-29","nationality":"Spanish"}, "Constructors":[{"constructorId":"mclaren", "url":"http:\/\/en.wikipedia.org\/wiki\/McLaren" ,"name":"McLaren","nationality":"British"}]}]}]}}}

Tabla 11- Ejemplo petición GET

5.3.3. Servicio $http

El servicio $http es un servicio propio de Angular que facilita la comunicación con servidores HTTP remotos a través de objetos XMLHttpRequest del navegador o JSONP.

El servicio $http es una función que toma un solo argumento, un objeto de configuración, que es utilizado para generar una *request* HTTP y devuelve una response en forma de promesa.

```
// Ejemplo de resquest GET:
$http({
  Method: 'GET',
  url: '/unaUrl'
}).then(function successCallback(response) {
// este callback se llamará de forma asíncrona cuando el response esté
disponible
  }, function errorCallback(response) {
  // Llamado de forma asíncrona si un error ocurre o el servidor devuelve
  response con un estado de error
  });
```

5.4. Capa de presentación

5.4.1. Diseño externo

El diseño externo de este proyecto viene fijado por el programador. Como se muestra en la ilustración 12, la aplicación dispone de un navegador para visitar las distintas páginas (marcado en color azul), un buscador en las tres páginas principales (en color marrón) a las que se puede acceder mediante el navegador (clasificación por pilotos, clasificación por equipos y carreras) y un *breadcrumb* (en color naranja) que muestra en que ruta de directorios se encuentra el usuario, además del contenido propio de cada vista (en color lila).

Tabla 12- Diseño web

Todas las vistas tienen un diseño similar, en la parte izquierda se encuentra el navegador, en la cabecera se muestra un *breadcrumb* junto a un campo de búsqueda para las tres vistas principales, mientras en el resto de la web se muestra contenido. El elemento principal para la navegación entre vistas es el navegador, aunque se puede navegar por las distintas vistas clicando sobre los hipervínculos que se muestran en las tablas.

5.4.2. Mapa de navegación

El sistema está dividido en vistas, en este apartado se muestran los modelos de navegación en cada una de las vistas.

La vista que se mostrará inicialmente es la de la clasificación por pilotos, a partir de ésta se puede navegar a las diferentes vistas según los intereses del usuario.

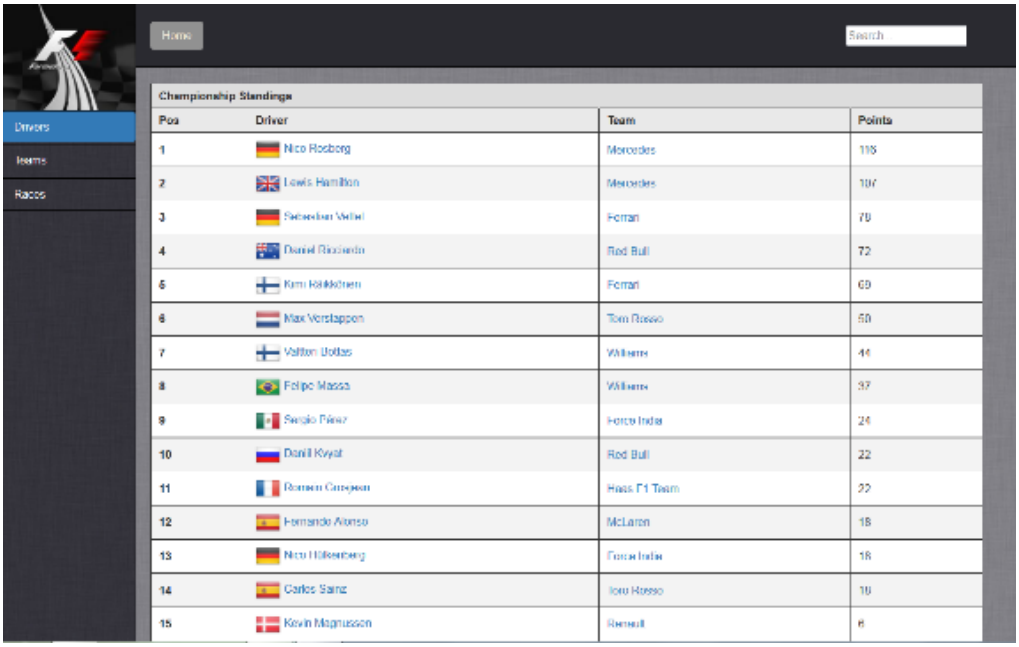


Tabla 13- Vista Driver

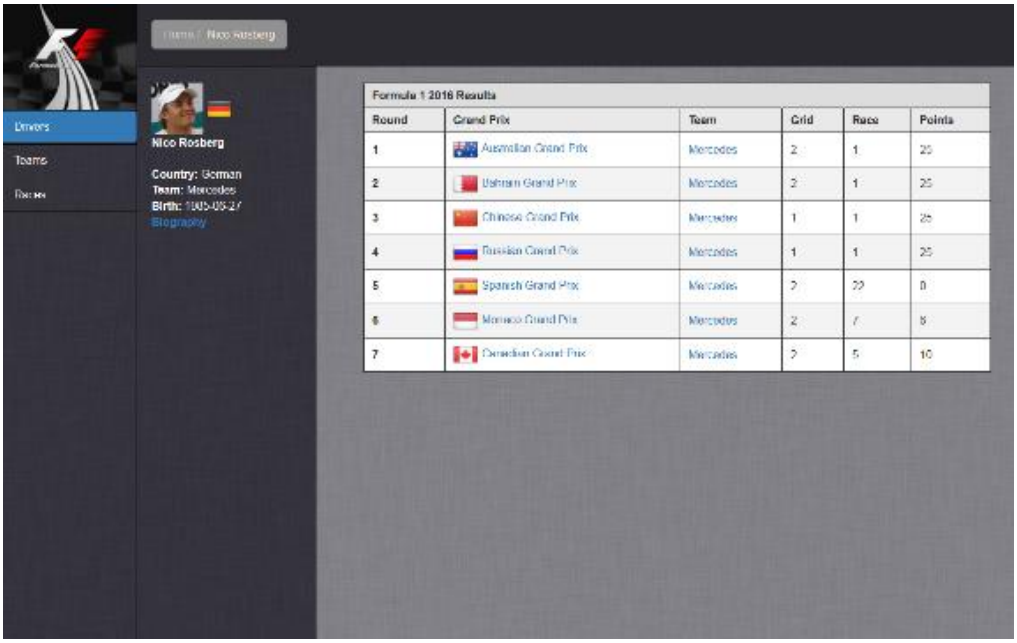


Tabla 14- Vista Drivers

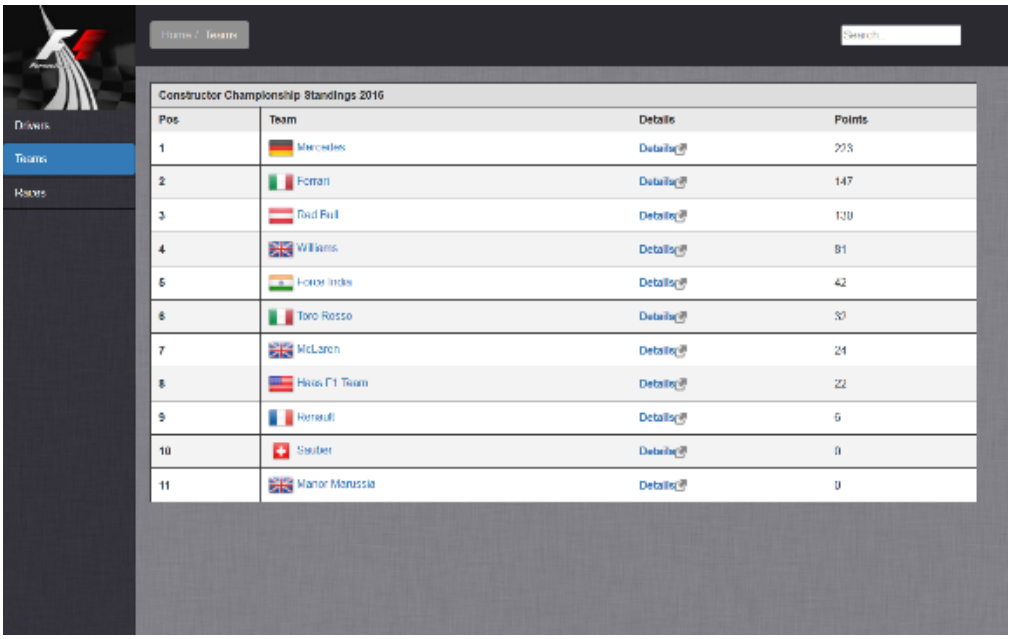


Tabla 15- Vista Team

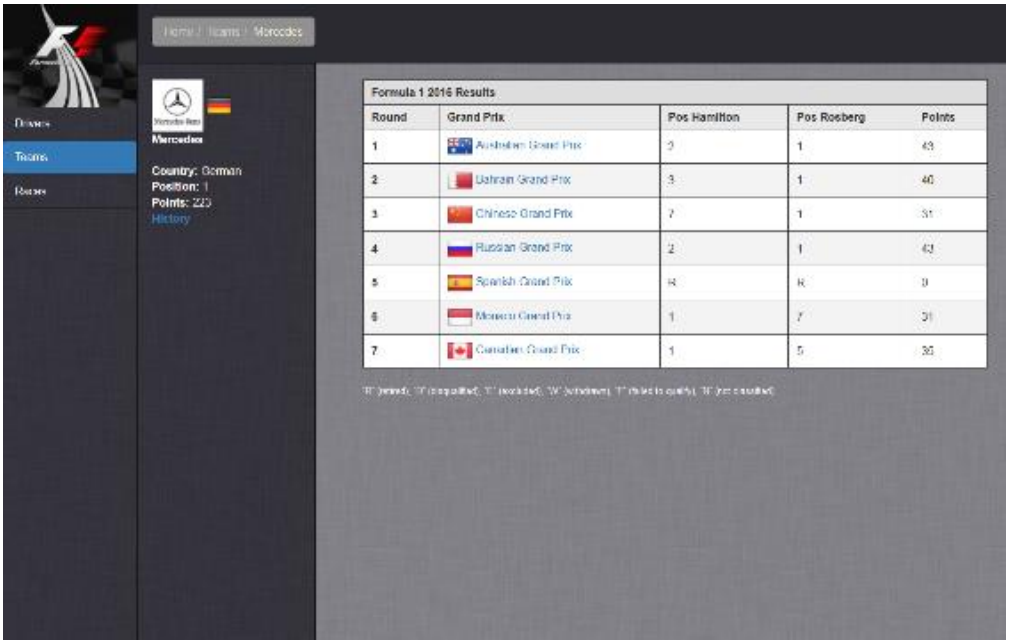


Tabla 16- Vista Teams

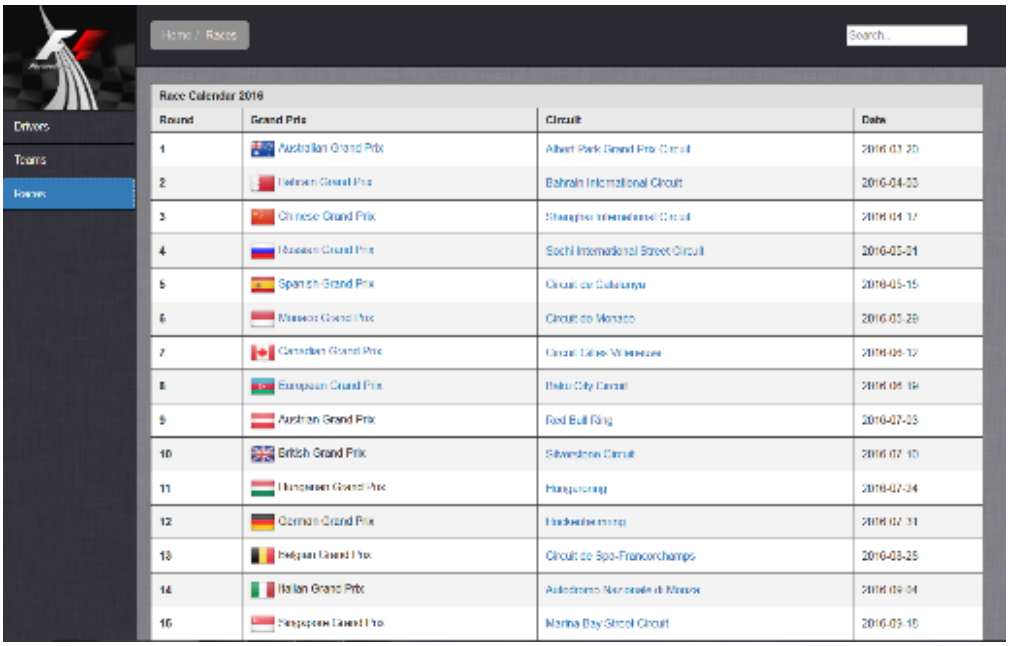


Tabla 17- Vista Races

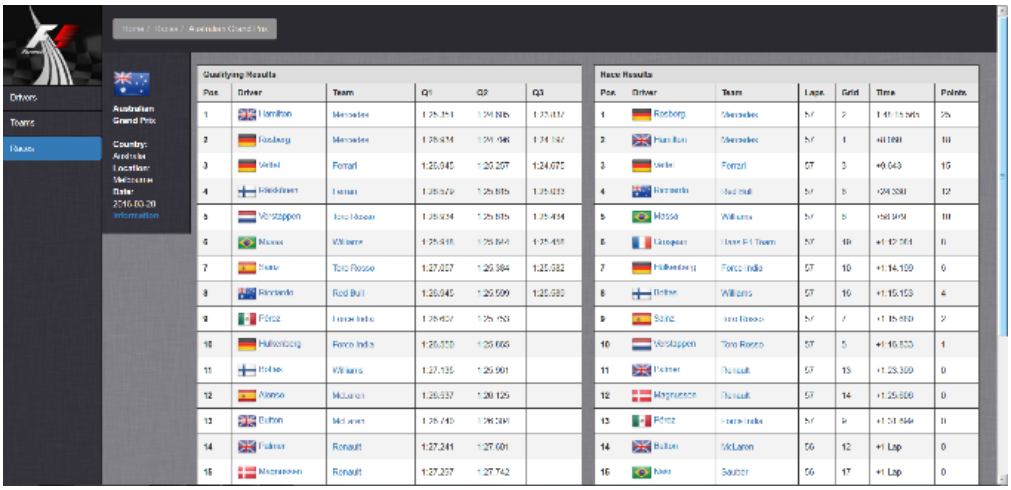


Tabla 18- Vista Grand Prix

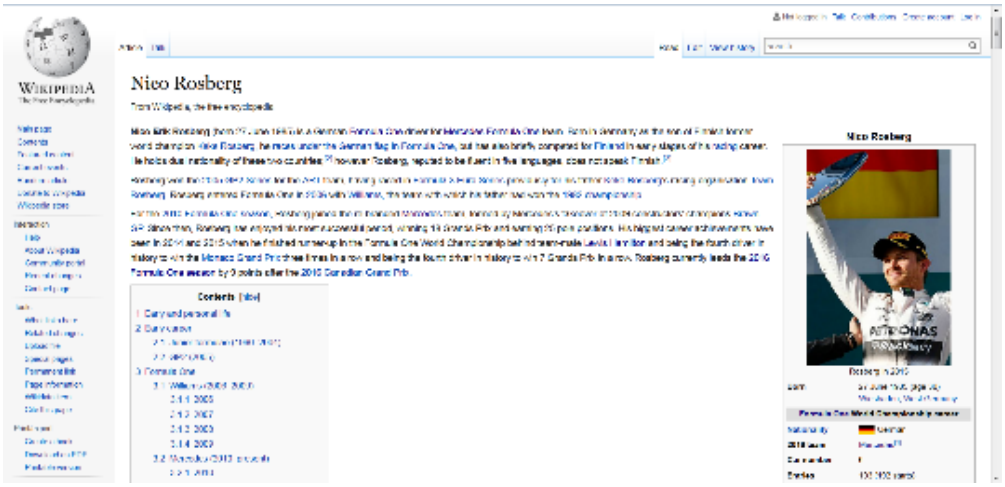


Tabla 19- Vista biografía piloto

Tabla 20- Vista información equipo

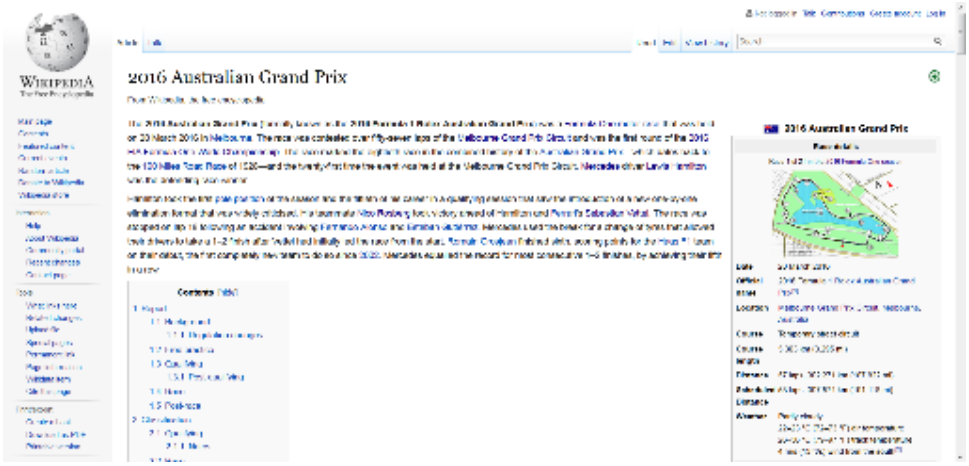


Tabla 21- Vista información carrera

Desde cualquier vista al hacer clic sobre el menú de navegación se va hacia la vista clicada.

Desde la vista *Drivers*, haciendo clic sobre un piloto se accede a la vista *Driver*, al hacer clic sobre un equipo se muestra la vista *Team*.

Desde la vista *Driver*, haciendo clic sobre sobre un *Grand Prix* se navega a la vista *Race*, al hacer clic sobre biografía se navega a la biografía del piloto.

Desde la vista *Teams*, haciendo clic sobre un equipo se accede a la vista *Team*, al hacer clic sobre detalles se muestra la vista información del equipo.

Desde la vista *Team*, haciendo clic sobre un *Grand Prix* se navega a la vista *Race*, al hacer clic sobre historia se muestra la vista información del equipo.

Desde la vista *Races*, haciendo clic sobre una carrera se accede a la vista *Race*, al hacer clic sobre el nombre del circuito se muestra la vista información de la carrera.

Desde la vista *Race*, haciendo clic sobre un piloto se accede a la vista *Driver*, al hacer clic sobre un equipo se muestra la vista *Team*, al hacer clic sobre información se muestra vista información de la carrera.

5.5. Capa de dominio

5.5.1. Modelo conceptual

Como resultado de aplicar la metodología *scrum* la especificación del modelo conceptual ha ido incrementando a medida que el proyecto avanzaba. Con cada *sprint* se ha ido ampliando y perfeccionando el modelo conceptual.

A continuación se muestra el modelo conceptual resultante al finalizar el proyecto.

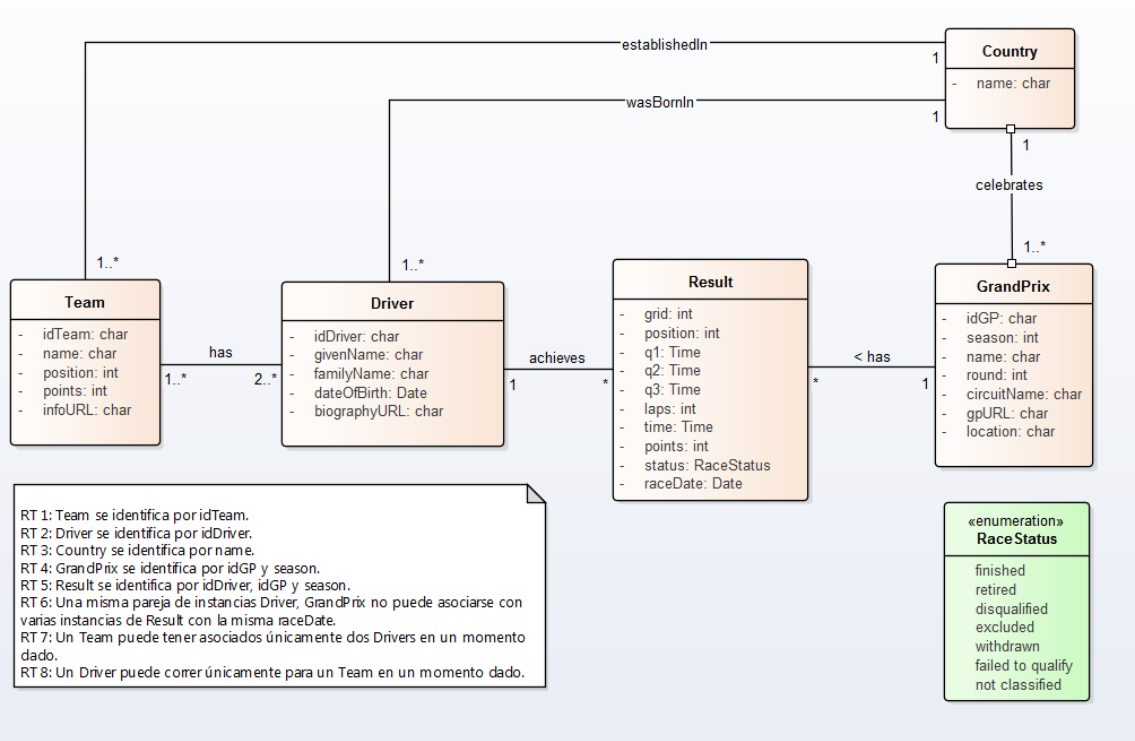


Ilustración 4- Modelo conceptual

5.5.2. Modelo de comportamiento

Todas las operaciones del sistema se pueden agrupar en operaciones de obtención de información o de búsqueda.

Como todas las operaciones son parecidas solamente se ha especificado un modelo de comportamiento por cada grupo.

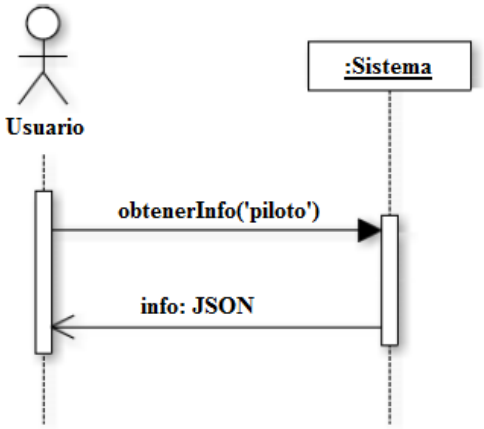


Ilustración 5- Diagrama secuencia obtenerInfo

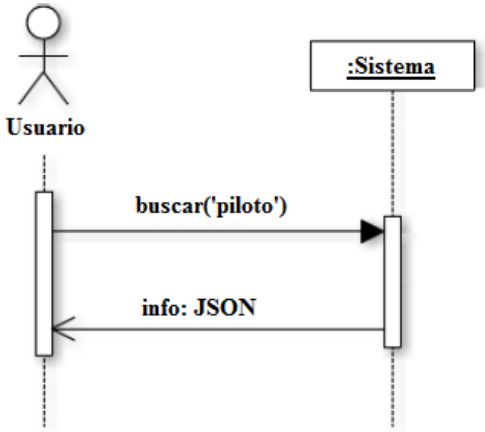


Ilustración 6- Diagrama secuencia buscar

5.6. Capa de datos

Este proyecto no dispone de capa de datos ya que todos los datos los obtiene de la API *ergast.* Estos datos no se almacenan en la aplicación, por lo tanto cada vez que se cierra y se vuelve a abrir la aplicación estos se vuelven a cargar desde la API.

6. Liferay

Liferay Portal es un portal de gestión de contenidos (*CMS*) de código abierto escrito en *Java*. *Liferay* ofrece un entorno de colaboración y una plataforma para redes sociales fáciles de utilizar, gracias a su interfaz de usuario. A continuación se describen brevemente sus principales puntos fuertes.

- Facilita el diseño de interfaces de usuario: *Liferay* simplifica el desarrollo de sitios web públicos y privados, que nos será de gran ayuda para configurar el sitio web tipo Usuario.

Este punto nos será muy útil para la reproducción de las aulas dentro de la Red de Aulas ya que desde el panel de control se pueden crear una plantillas web, con su menú y páginas, así una vez creada una plantilla se puede reproducir tantas veces como se quiera.

- Personalización de usuarios: Los usuarios podrán personalizar sus propias páginas; añadiendo, quitando, ordenando o configurando acorde a sus necesidades las aplicaciones disponibles, parecido a lo que nos ofrecen aplicaciones web conocidas como *Google* o *Gmail*. A diferencia de un desarrollo manual *Liferay* ofrece esta personalización de serie y además permite añadir nuevos elementos, páginas web.

- Soporte de Single Sign On (SSO): Nos permite integrar sus contenidos y aplicaciones en una única plataforma sin necesidad de autenticarse múltiples veces. *Liferay* es capaz de integrar todos sus sistemas, permitiendo acceder a ellos desde una sola sesión, gracias a la integración con múltiples mecanismos de *Single Sign On* (SSO). SSO es un procedimiento de autenticación que habilita al usuario para acceder a varios sistemas con una sola instancia de identificación siéndonos útil ya que nuestra plataforma de comunicación integra una *MediaWiki* y un *Moodle* de CIMNE.

- Liferay Sync: *Liferay* Sync es una aplicación que sincroniza los archivos de su biblioteca de documentos de *Liferay* con su entorno de escritorio local, permitiendo que cualquier cambio realizado en sus archivos locales se actualice automáticamente en el repositorio del portal convirtiendo el portal de comunicación en una nube. Esta funcionalidad se ha empleado para los sitios web tipo Usuario, dando a cada usuario la opción de colgar archivos en el portal y poder visualizarlos mediante *Liferay Sync* en cualquier dispositivo incluido móviles tipo Ipad o Iphone.

- Social Officel: *Social Office* es un conjunto de herramientas que permiten una visibilidad ininterrumpida de la actividad de la organización o del grupo de trabajo en un entorno web compartido, fomentando de este modo que la comunicación y las relaciones de cada miembro del equipo tengan lugar en el contexto de un esfuerzo de colaboración común. Estas herramientas serán el eje de nuestro canal de

comunicación, siendo el motor de la red social. Este paquete también nos ofrece herramientas de gestión de trabajo.

6.1. Introducción a *Liferay*

Liferay es un gestor de portales *Java*, presenta más de 60 *portlets* integrados en el núcleo, que facilitan la puesta en marcha de un portal web. Este gestor en su versión 6 va acompañado de un *SDK* (*softwaredevelopment kit*) y un *plugin* de *Eclipse* que ayuda a implementar un portal totalmente a medida.

Liferay separa sus herramientas en 5 bloques: *portlets*, *Hooks*, *layout*s, *templat*es, y *Theme*. También existe la posibilidad de extender del núcleo de *Liferay* con el *plugin* Ext.

Para la elaboración de este proyecto se ha hecho uso de *portlets* y *themes*.

6.1.1. Portlets

Los *portlets* son aplicaciones contenidas dentro de un portal y se encargan de generar contenidos dinámicos para él.

Existen dos especificaciones JSR 168 y JSR 286 (Anexo III). Estas especificaciones definen una serie de funciones y clases que deben implementar las aplicaciones para poder seguir el estándar y de esa forma, ser portables de un portal a otro.

A nivel técnico, un *portlet* empieza por una clase *Java* que implementa la clase interfaz *Java*x.*portlet*.*Portlet*. Los *portlet* sólo tienen sentido cuando están contenidos dentro de un contenedor de *portlet* que suele ser el portal, e interactúan con los usuarios web con un paradigma de peticiones y respuestas (*request/response*).

A diferencia de los *servlet* (pequeños programas que se ejecutan en el contexto de un navegador web), los *portlet* no pueden ser llamados de forma directa, es decir, de un *servlet* se puede escribir su dirección web pero de un *portlet* sólo se podrá obtener la dirección web de la página que lo contiene, pero no una petición directa sobre él. Otra diferencia es que los *servlets* pueden generar páginas o documentos completos, mientras los *portlets* solo generan una parte del contenido.

En *Liferay*, todas las aplicaciones como *wiki*, *blogs*, contenidos, etc. son *portlets*, ya que *Liferay* es un contenedor de *portlets* y si una aplicación quiere integrarse en este portal tiene que cumplir con este requisito. Además de las funcionalidades existentes, podremos nuevos *portlets* a medida para realizar otras tareas adicionales que no implementa *Liferay* por defecto. Esto resulta interesante para muchas empresas y organizaciones donde es común disponer de aplicaciones en funcionamiento para realizar tareas, tales como imputación de horas, peticiones de viajes, o registro de pagos.

6.1.2. Hooks

Este término es propio de *Liferay*. Los *Hooks* viene a ser una de las maneras que tiene *Liferay* de sobrescribir funcionalidades del portal, se agradece por parte de desarrolladores y administradores de sistema puesto que evita tener que recompilar el código del portal cada vez que se desea realizar una modificación en determinados parámetros internos del portal.

Un claro ejemplo de esto son las traducciones o los parámetros de las páginas. En el caso de que se deseé añadir un eslogan al portal web, la vía tradicional implica modificar el fichero de los idiomas del portal y recompilarlo. Con los *Hooks* de *Liferay* sólo es necesario generar un *Hook*, modificar el fichero que se desea y cargar el *Hook* al portal. Otro caso similar sucede si se quiere añadir un parámetro nuevo a las páginas, por ejemplo para que no muestren la cabecera.

Los *Hooks* utilizan una gran variedad de tecnologías puesto que dependen de la parte del portal que se vaya a modificar, por esa razón abarca *Java*, *JSP*, *HTML*, *XML*.

6.1.3. Layout

El *layout* se encarga de la disposición de los *portlet* dentro de la página, organizándolos en filas y columnas según la combinación que se diseñe. Cada página puede tener un único *layout*, pero varias páginas pueden utilizar el mismo. Una vez escogido el *layout* se podrán introducir los *portlets* que van a aparecer, en las filas y columnas escogidas de la página.

Es importante planificar los *layout* y el *Theme* de forma conjunta puesto que la estructura diseñada en *layout* será maquetada por el *Theme* es decir, los colores, separaciones y anchos serán fijados por el *Theme* y no por *layout*. De esta forma, un *layout* puede ser reutilizado por varios *Themes* y un mismo *Theme* puede emplear varios *layout*.

Los *layout* están desarrollados con *HTML* y *Velocity.*

6.1.4. Theme

El *Theme* es el módulo de *Liferay* que se encarga de gestionar la apariencia del portal web. Se desarrolla mayoritariamente en 3 tecnologías: *CSS*, *Velocity*, *Javascript* con *Jquery*.

Es recomendable que este módulo se inicie partiendo de uno ya existente o de las bases que ofrece la *SDK*, al generar un módulo de tipo *Theme*.

Dentro del módulo *Theme*, los ficheros se encuentran organizados en carpetas, agrupados según la tecnología que contienen. Entre estas carpetas se encuentra la denominada _diffs. En esta, se debe replicar la estructura anterior y modificar el fichero que se quiere adaptar, puesto que *Liferay* al compilar el módulo de tema,

consulta la carpeta _diffs y añade los cambios que esta contenga al resto de carpetas. En caso de realizar cambios en otras carpetas y no en _diffs, es posible que al compilar los cambios introducidos sean eliminados.

6.1.5. Ext.

Ext. es un *plugin* que proporcionan los métodos más avanzados para extender de *Liferay*. Ext. fue diseñado para utilizarse en casos específicos en que los otros *plugin*s , como los *portlets* o *Hooks*, no satisfagan las necesidades del proyecto.

Antes de decidirse a usar un *plugin* Ext. es importante entender los costes. El principal coste es el mantenimiento del portal, dado que Ext. permite el uso de API interna o sobrescribir los archivos, incluso en el núcleo *Liferay*. Es por ello que es necesario revisar todos los cambios realizados al actualizar una nueva versión de *Liferay*. Además, a diferencia de los otros tipos decomplementos, Ext. exige que el servidor se reinicie después de la implementación, así como requiere pasos adicionales a implementar o al reprogramar los sistemas de producción.

6.2. Arquitectura de Liferay

Liferay es un gestor de portales que sigue el diseño Modelo Vista Controlador (*MVC*) y permite la creación y gestión de usuarios, páginas, grupos y roles.

En este apartado, se hablará con más detalle sobre la arquitectura de *Liferay* y su estructura básica.

6.2.1. Instancias

Las instancias básicas de un portal de *Liferay* son los *sites* (sitios web), las páginas, los usuarios, los roles, las organizaciones y los grupos, donde todos ellos interactúan de la manera que se describe a continuación.

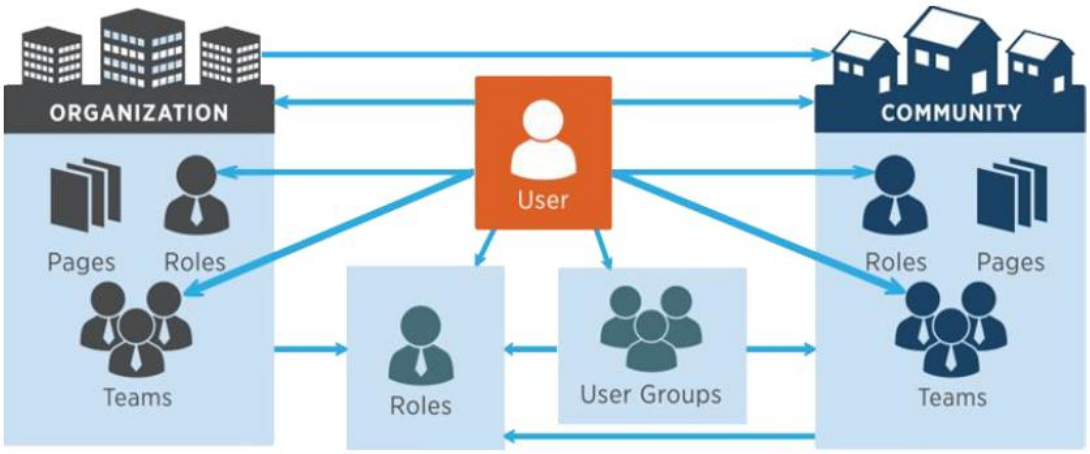


Ilustración 7- Diagrama de instancias de Liferay

Los *sites* están compuestos por un conjunto de páginas, y estos pueden ser de dos tipos:

- ***Sites* de organización:** Permiten la aplicación de jerarquías a los usuarios y a los *sites*, permitiendo así la delegación de permisos de administración.

- ***Sites* de comunidad:** Son como los *sites* de organización pero no permiten jerarquía, evitando así la delegación de permisos.

Los usuarios pueden pertenecer a varios *sites*, a la vez que pueden estar asignados a diferentes grupos de usuarios y tener asignados diferentes roles.

6.2.2. Roles

Liferay gestiona los permisos a través de los roles, donde los roles son un conjunto de permisos que determinan las acciones sobre los diferentes recursos de *Liferay* como los *portles*, las entidades y los archivos.

Existen tres tipos de roles:

- ***Regular roles*:** Determinan los permisos genéricos sobre el portal.

- ***Organization roles*:** Determinan los permisos sobre las organizaciones.

- ***Sites roles*:** Determinan los permisos sobre los *sites*.

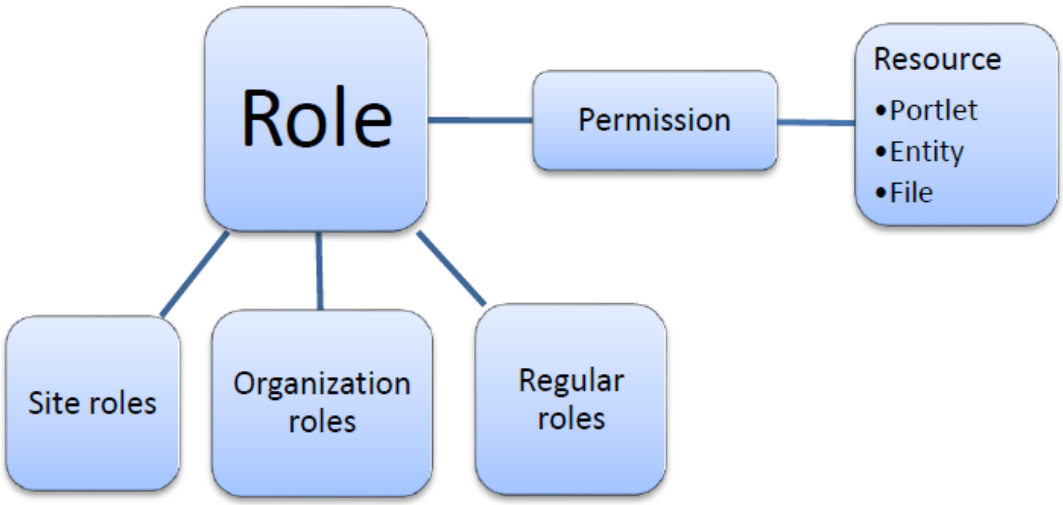


Ilustración 8- Diagrama de roles

6.2.3. Site pages

Los *sites* son un conjunto de páginas, los *sites* pueden ser públicos a cualquier usuario del portal o privados, donde solo podrán acceder miembros de un mismo *site*.

En el caso de los *sites* privados, pueden ser de dos tipos:

- **Públicos:** Todos los usuarios del *site* pueden acceder libremente desde el panel de control.

- **Privados:** A diferencia de los públicos, los usuarios no tienen acceso a él a través del panel de control, por lo que necesitan saber la dirección específica del *site* para poder acceder.

6.2.4. Jerarquía

Las organizaciones son una colección jerárquica de usuarios, a las cuales se les pueden asignar roles y *sites*.

Ilustración 9- Diagrama de jerarquía

Para este proyecto se ha diseñado una organización con un *site* público que contiene la web SPA.

6.3. Arquitectura de Liferay

Liferay utiliza una arquitectura de tres capas siguiendo el patrón de diseño Modelo Vista Controlador (*MVC*). El patrón *MVC* consiste en separar la lógica de negocio de la presentación de la manera siguiente:

- **Modelo:** Es la parte donde se definen las clases de la información del mundo real. El modelo es la capa que se encuentra por encima de la base de datos y realiza la gestión de estos.

- **Vista:** Es la parte que se encarga de la representación visual de la información del sistema.

- **Controlador:** Es la parte que se encarga de gestionar las peticiones generadas por los usuarios, siendo intermediario de la comunicación entre el modelo y la vista. El controlador recibe la petición del usuario, la delega al modelo, y una vez gestionada por el modelo se envía en forma de respuesta a la vista a través del controlador.

Tras esta breve introducción al patrón *MVC*, *Liferay* se estructura de la manera siguiente:

- ***Frontend*:** Correspondería a la Vista, dónde se muestra toda la información del sistema a través de código *HTML* y *CSS*. El *frontend* comunica directamente con el usuario mostrándose a través de navegadores web o aplicaciones nativas.

- **Servicios:** Correspondería al Controlador, ya que esta parte es la encargada de comunicar el *frontend* con la base de datos mediante las llamadas a los servicios a través de la *API* de *Liferay*.

- **Persistencia:** Correspondería al Modelo, ya que es la parte encargada de comunicarse con la base de datos a través de sentencias *MySQL*, permitiendo la persistencia de los datos del sistema.

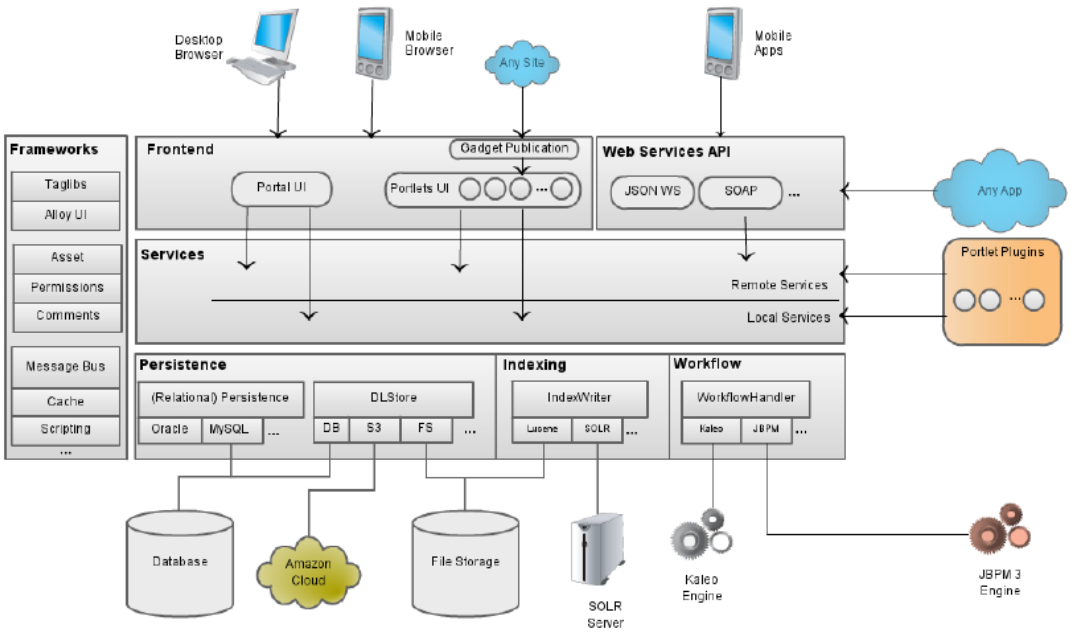


Ilustración 10- Arquitectura de Liferay

6.4. Tecnologías utilizadas por *Liferay*

Uno de los puntos fuertes de *Liferay* es el uso de estándares para su portal. Como norma general tiende a utilizar una tecnología estándar, libre y generalizada para cualquier funcionalidad que realiza. Las tecnologías más utilizadas son: *Java*, *JSP*, *XML*, *CSS*, *HTML*, *Javascript* con *Jquery* y *Velocity*.

En este apartado se comentan estas tecnologías.

6.4.1. Velocity

Velocity es un motor generador de estructuras de página basado en *Java*. Proporciona un modelo sencillo de referenciar objetos definidos en el código *Java*. Su propósito es realizar de forma clara y simple la capa de presentación dentro del patrón de diseño "Model View Controler" (MVC), donde se separan claramente las capas de presentación, estructura y control, siendo *Velocity* la capa de presentación para un modelo MVC desarrollado en *Java*.

La gran ventaja que aporta el uso de *Velocity* es la posibilidad de trabajar en paralelo desarrolladores de software y diseñadores web, siempre y cuando ambos mantengan el modelo MVC.

6.4.2. CSS

Las Hojas de Estilo en Cascada (*Cascading Style Sheets* o *CSS*) son un lenguaje para describir la presentación de los documentos. Aparecieron con el estándar de *HTML* 4 como complemento para definir el aspecto y la presentación que debían mantener los documentos estructurados de tipo *HTML*.

CSS permite adaptar un mismo contenido a diversos medios de visualización como son: una pantalla de ordenador, una versión impresa o un intérprete de braille.

6.4.3. HTML

El formato *HTML* (*HyperText Markup Language*) es el estándar para la navegación en la web.

Liferay, como cualquier herramienta web, acaba generando un código en *HTML* que es enviado al navegador del cliente para ser interpretado. Este lenguaje se apoya en los demás para modificar su comportamiento y aspecto visual, por que por sí solo entregaría contenido estático.

6.4.4. JSP

JSP es el acrónimo de *Java Server Pages* y consiste en una tecnología *Java* que permite generar contenido *HTML* o *XML* de forma dinámica, haciendo uso de la máquina virtual de *Java*, pudiendo utilizar las clases y todo el potencial que este lenguaje ofrece.

Al tratarse de *Java*, Sun Microsystems fue el responsable de los dos estándares que se han generado *JSP* 1.2 y *JSP* 2.1, y están estrechamente ligados a los estándares de *servlet* que han aparecido.

6.4.5. XML

XML es un lenguaje de marcado que predomina en la parte de configuración del servidor. Gran parte de los parámetros y valores iniciales de las variables son guardados en este tipo de documentos.

6.4.6. JavaScript y Jquery

*Java*Script es un lenguaje de programación interpretado por el navegador web. Esta tecnología se ejecuta en el lado del cliente (dentro de su navegador) y es utilizada para reaccionar ante eventos del usuario o para dar dinamismo a las páginas web, como por ejemplo, efectos de deslizarse o de sustitución de imágenes.

Jquery es uno de los *framework* de *Java*script más extendidos actualmente. Principalmente, por facilitar el acceso a los elementos *HTML* del documento y poder interactuar con facilidad con el aspecto visual que establece el *CSS*, creando efectos visuales muy vistosos que sin *Jquery* conllevarían un desarrollo elevado.

7. Tecnologías utilizadas

En este apartado se hablará sobre las tecnologías utilizadas en el proyecto.

7.1. Angular

AngularJS, es un *framework* de *JavaScript* de *código abierto*, mantenido por *Google*, que se utiliza para crear y mantener webs SPA.

AngularJS está construido en torno a la creencia de que la programación declarativa es la que debe utilizarse para generar interfaces de usuario y enlazar componentes de software, mientras que la programación imperativa es excelente para expresar la lógica de negocio. Este *framework* adapta y amplía el HTML tradicional para servir mejor contenido dinámico a través de un *data binding* bidireccional que permite la sincronización automática de modelos y vistas. Como resultado, *AngularJS* pone menos énfasis en la manipulación del DOM y mejora la testeabilidad y el rendimiento.

Angular es *client-side*, la interfaz de usuario y la lógica de se encuentran en el lado del cliente en lugar de servidores web. En angular el estado de los controladores y el modelo se mantiene en el navegador del cliente, así las páginas se pueden generar sin interacción con el servidor.

Para que toda la lógica no se encuentre entre los controladores y los *ViewModels* ($s*copes*), se hace uso de los servicios. En estos servicios se agruparán funcionalidades que luego estarán disponibles para ser utilizadas en los controladores.

En angular los servicios son la herramienta básica para mantener el estado de la aplicación, desde la capa cliente, y compartir información entre los controladores. Un controlador modifica el estado de un servicio, y el siguiente controlador utiliza ese servicio para acceder al nuevo estado.

Mediante el uso de inyección de dependencias en el momento de definir cualquier componente, se debe indicar de qué otros componentes depende y angular se encargará de proporcionárselos a través de la función constructora.

Al ser todos los componentes registrados en *AngularJS singletons* se facilita que los servicios almacenen el estado, ya que dos *controladores* que dependan del mismo servicio estarán utilizando el mismo objeto y así podrán compartir información a través de él.

El hecho de utilizar un contenedor de inyección de dependencias hace que sea necesario definir explícitamente las dependencias ya que no existe un sistema de tipos que permita inferirlas automáticamente.

Para poder organizar mejor el código, angular implementa un sistema de módulos así, cada vez que se define un componente se hace dentro de un módulo.

Se ha tenido que utilizar un *routeo* diferente al tradicional, y se ha optado por utilizar el ui-router, este funciona como una máquina de estados. Al tratarse cada vista como un estado se ha podido añadir un breadcrumb y se ha podido hacer uso de vistas anidadas, lo que de otra forma sería difícil de implementar, se convierte en un trabajo sencillo.

7.1.1. Arquitectura AngularJS

En AngularJS el sistema de *templates* funciona en objetos DOM, no en *strings*. Aun así el *template* está escrito en un *string* HTML pero es HTML. El navegador *parsea* el HTML en el DOM y el DOM se vuelve la entrada al motor de plantillas conocido como compilador. El compilador busca directivas que a su vez añade *watches* en el modelo. El resultado de esto es una vista continuamente actualizada que no necesita hacer *remerge* del modelo y el *template*. El modelo se vuelve *single-source-of-truth* para la vista.

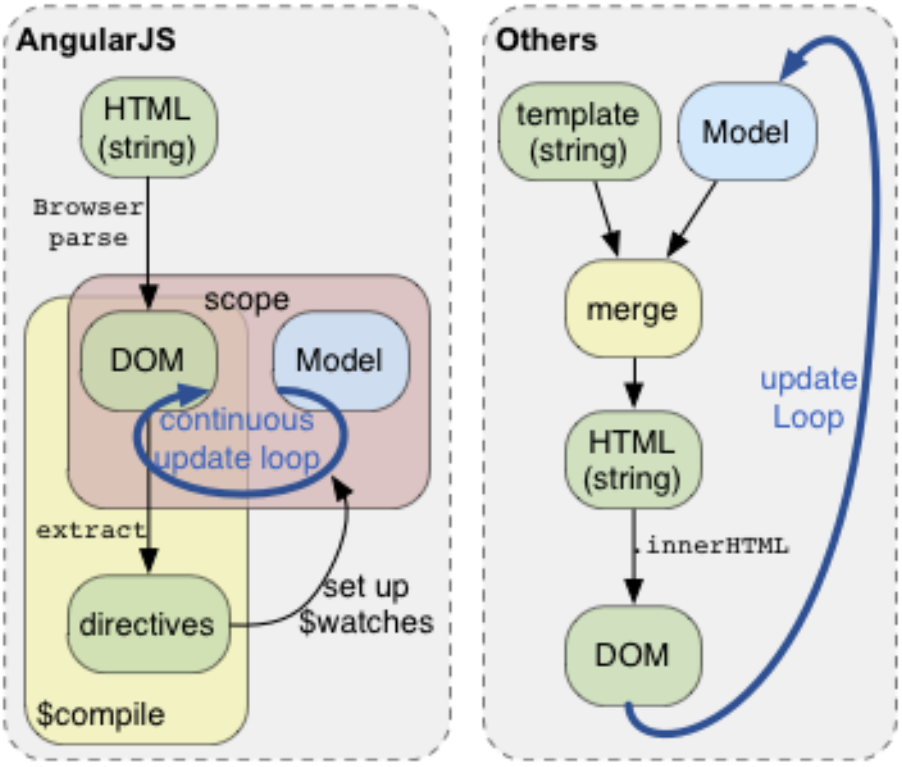


Ilustración 11- Arquitectura AngularJS vs otros

7.2. Liferay

Liferay es un gestor de contenido con Java de código abierto. Inicialmente apareció para organizaciones sin ánimo de lucro en el año 2000.

Este gestor de contenido, o CMS, se utiliza tanto para la creación de portales web como para intranets privadas. Por ejemplo, la web de TMB (Transports metropolitans de Barcelona) fue renovada utilizando Liferay. Se creó un nuevo portal con soporte medio para 2000 usuarios concurrentes y 8000 páginas visitadas por hora, que en hora punta podía llegar a 3000 usuarios y 10000 páginas por hora. Otro beneficio que aportó fue el posicionamiento SEO (*Search Engine Optimization*) de la web, introducción de contenido multidioma y la escalabilidad del portal.

Otro ejemplo de uso de *Liferay*, este vez también para pagines privadas, es el de la consejería de Sanidad Valenciana, que creó sus pagines privadas y públicas para promocionar la colaboración y comunicación entre los más de 6000 empleados.

Para la creación de la capa de datos *Liferay* integra el *service builder*, que es una herramienta de construcción que permite la creación de los interfaces y clases para la capa de persistencia y los servicios. Al utilizar este sistema, se crean las tablas en la BD y se implementan los métodos básicos de *create*, *update* y *delete*. Estos métodos se invocan a través del *localServiceUtil* del objeto y permiten realizar el acción sobre el objeto o con el identificador, pero también se pueden definir otras funciones en los *LocalServiceUtilImpl* o sino, utilizar *DynamicQuery* para obtener elementos de la BD que cumplan algunas condiciones.

Liferay se basa en un sistema de *portlets* independientes entre ellos y cada uno con una funcionalidad propia, que se encuentran en las páginas del portal.

7.2.1. Ventajas

Al ser un gestor de contenidos nos aporta muchas ventajas respecto a tener que hacer todo el sistema de cero, por ejemplo incluye, la gestión de usuarios, gestión documental, funcionalidades ya creadas en más de 60 *portlets* y la creación de funcionalidades como páginas de wiki.

Entre todas estas funcionalidades ya creadas está, por ejemplo, los foros con varias funcionalidades típicas (como marcar contenido inadecuado), chat interno, servicio de mensajes, clasificar contenidos y mostrar según estos. También cuenta con integración con Microsoft Office, Marketplace de donde descargar librerías, temas y otras funcionalidades, calendario compartido, creación de eventos, sistema de permisos, Mobile Device Rules para adaptar las vistas según los dispositivos, multidioma, etc.

Otra ventaja es que funciona correctamente sobre la mayoría de servidores de aplicaciones y contenedores de *servlets*, y también es multiplataforma.

Además, la creación de la página se hace con *Drag&Drop* de los *portlets*. Al ser un sistema que existe desde hace más de 13 años tiene robustez, es fácil encontrar información acerca de él y cuenta con un equipo de desarrollo que se encarga de corregir bugs que encuentran los usuarios en las nuevas versiones, y ayudar al desarrollo a través de foros.

Integra varias tecnologías y estándares que pueden ser utilizados como: *velocity, AJAX, Lucene, jQuery, Hibernate, Spring, openSearch, servicios web, struts, jBPM...*

Y por último, está *Benchmarked* cómo uno de los portales más seguros en el uso de *LogicLibrary's Logiscan suite*.

7.2.2. Inconvenientes

Las funcionalidades que requieran crear o modificar un *portlet* pueden ser complejas y requerir conocimientos específicos de cómo funciona. Si se desea modificar el estilo de las vistas de un *portlet* ya existente sin utilizar ningún tema ya creado, puede ser necesario modificar la estructura y para ello se debe modificar el código del núcleo, cosa que requiere conocimientos avanzados.

El *servicebuilder* provoca algunas limitaciones como son no poder crear claves foráneas entre las diferentes tablas, tener que crear métodos y limita las consultas en comparación a las que se pueden hacer con *MySQL*.

El uso de Java, que últimamente no es uno de los lenguajes de programación que más entusiasma a los programadores.

7.3. Java

Java es el lenguaje sobre el cual está construido *Liferay*. Java fue creado por James Gosling de Microsistems y fue publicado al 1995. Es un lenguaje de uso general, orientado a objetos, concurrente y basado en clase. Deriva de C y C++, pero no permite realizar acciones abajo nivel como este, así como tampoco permite la sobrecarga de operadores.

Se ejecuta sobre una máquina virtual (JVM) con lo cual se puede ejecutar sobre cualquier sistema y lo hace multiplataforma y portable. Pero el inconveniente es que esto provoca que su ejecución sea más lenta. Además, para liberar memoria, utiliza el recolector de basura, que elimina los objetos cuando no tienen ninguna referencia que los apunte, y la ejecución constando de este junto a la ejecución sobre la máquina virtual ralentiza la ejecución.

7.4. Apache Tomcat

Apache Tomcat está escrito en Java, y por lo tanto, funciona sobre cualquier sistema operativo que tenga la máquina virtual de Java.

Es un servidor web con apoyo para *servlets* y JSPs. Incluye el compilador Jasper que compila los JSP y los convierte en *servlets*. Además, puede funcionar como servidor web por sí mismo, y es utilizado de manera autónoma en entornos con un alto nivel de tránsito y alta disponibilidad.

7.5. JSP

JavaServer Page es similar al lenguaje PHP pero en Java. Permite la creación de páginas web dinámicas en el servidor utilizando Java.

Son una forma alternativa de crear *servlets*, puesto que la primera vez que los invocan se traducen a código de *servlet* Java, y desde entonces, es este código el que se ejecuta, generando una salida de código HTML.

7.5.1. Ventajas

Los *servlets* y JSPs se ejecutan en una máquina virtual de Java, por lo tanto son portables y multiplataforma.

Una de las principales ventajas de los JSPS es que es apto para crear clases que controlen la lógica de negocio y acceso a datos, de forma que se puede separar en niveles la aplicación y dejar al JSP el código encargado de generar el HTML.

Además, cada JSP se ejecuta en su propio hilo de ejecución y contexto, y este perdura de una petición a la siguiente, por lo tanto permite crear conexiones a la BD o la gestión de sesiones de manera eficiente.

7.5.2. Inconvenientes

Solamente se puede utilizar sobre Java, por lo tanto solo puede funcionar sobre servidores que lo soporten. Además, al ejecutarse sobre una máquina virtual es posible que afecte a su rendimiento.

7.6. HTML

HTML (o *HyperText Markup Language*) es un lenguaje para crear páginas web que fue inventado en 1990 por Tim Berners-Lee.

Permite crear la estructura básica de las páginas web utilizando un sistema de etiquetas (o *tags*) que representan los elementos. Cada elemento consta de dos partes: atributos y contenido.

7.6.1. Ventajas

Las principales ventajas de la HTML son que es muy fácil de entender y de utilizar y que es compatible con los navegadores actuales.

7.6.2. Inconvenientes

Los inconvenientes de este lenguaje son que al ser un lenguaje estático necesita utilizar otros lenguajes cono *JavaScript* para poder generar HTML dinámico o realizar acciones. Para dar estilo al esqueleto creado y que resulte agradable se tiene que utilizar CSS.

7.7. CSS

CSS o (*cascade style sheet*) es un lenguaje utilizado para dar estilo a lenguajes de marcas como HTML o XML. Se crean reglas para definir el diseño y estilo de los elementos.

7.7.1. Ventajas

La ventaja del CSS es que permite una gran personalización de los elementos de las páginas web.

7.7.2. Inconvenientes

El principal inconveniente de este lenguaje se la dificultad para obtener buenos resultados en todos los navegadores, puesto que no funciona igual en todos y es necesario definir reglas específicas para solventar este problema.

7.8. JavaScript

JavaScript es un lenguaje de programación interpretado, orientado a objetos, dinámico, imperativo, basado en prototipos y de tipado débil. Creado por Brendam-Eich en 1995. Su principal uso está enfocado a la parte del cliente (*client-side*), que se ejecuta en el navegador cliente y permite crear páginas web dinámicas y mejorar las interfaces. También se puede utilizar en la parte del servidor.

7.8.1. Ventajas

Sus ventajas son el envío de información mediante la técnica AJAX, la mejora de la experiencia de los usuarios y que permite crear elementos dinámicos. Además, existen muchos *plugins* creados con *JavaScripts*, y librerías como *jQuery*.

7.8.2. Inconvenientes

La desventaja de este lenguaje es que para ejecutarlo en el navegador se descarga y es visible por el cliente.

7.9. AJAX

AJAX (o *Asynchronous Javascript And XML*) es una tecnología para el intercambio de información asíncrona entre el cliente y el servidor. Se ejecuta en el navegador y permite realizar intercambios de información en segundo plan sin

interferir en la visualización y el comportamiento de la página, mejorando la usabilidad, la interactividad y la velocidad de las aplicaciones.

7.9.1. Ventajas

Las ventajas de esta técnica son derivados del hecho que las peticiones y el intercambio de información se ejecuten en segundo plano.

Permite el envío y validación de formularios sin tener que recargar las páginas y se pueden hacer cambios en páginas sin que se tengan que volver a cargar, mejorando así la experiencia del usuario y la usabilidad.

7.9.2. Inconvenientes

Cómo utiliza JavaScript, y se puede desactivar, se necesario tener otras opciones para que funcione siempre.

7.10. AUI

Alloy UI se un *framework* de interfaz de usuario que proporciona un entorno de desarrollo sencillo por la creación de aplicaciones web. Tiene acceso a los tres niveles del navegador: estructura, estilo y comportamiento, y por eso utiliza: HTML, CSS y *JavaScript*.

7.10.1. Ventajas

Como todos los *frameworks,* tiene una serie de funciones ya creadas que facilitan el uso, además, al utilizar *JavaScript*, CSS y HTML tiene utilidades que son el resultado de la combinación de los tres lenguajes pero que resultan más sencillas de utilizar. Por ejemplo, la validación de formulario, el alineación de los elementos y el *label* con el nombre...

7.10.2. Inconvenientes

El inconveniente más importante es que al no estar muy extendido su uso, no hay tanta documentación como por ejemplo *JQuery*. Otro inconveniente que se ha encontrado durante la realización del proyecto es que interfiere con el CSS creado teniendo únicamente en cuenta Angular.

8. Implementación

- **Integración Liferay**

Para crear una aplicación web con *AngularJS* es necesario definir un atributo *ng-app* como se muestra a continuación para que este haga *bootstrap* de sí mismo:

```
<div ng-app="main">
   <div ng-view></div>
</div>
```

Cuando la página cargue, *AngularJS* busca el atributo *ng-app*. Si encuentra el atributo inicializará la aplicación, de otro modo no lo hará. Pero cuando se utiliza en entorno de portales no siempre es posible tener un único atributo *ng-app*. Puede haber más *portlets* desarrollados con *AngularJS* en la misma página o puede haber *portlets* instanciables que se utilizan más de una vez en la misma página. En estos casos en los que habría más de un *ng-app* en la misma página, la estructura mostrada anteriormente no funciona correctamente con *portlets*, ya que solamente se permite un *ng-app* por página.

Para solucionar esto, es necesario evitar que *AngularJS* haga *bootstrap* por sí mismo y habrá que hacer *bootstrap* manualmente, como se muestra a continuación.

Para esto primero se le da un id único a la aplicación, utilizando para ello el *tag* <porlet:namespace/> que provee el portal para JSP y es único por diseño.

```
<div id="<portlet:namespace />main">
   <div ng-view></div>
</div>
```

Al llamar al método *bootstrapRouter*, se le pasa el *namespace* para que *AngularJS* sea correctamente *namespaced*. Además, es necesario usar el *tag* AlloyUI combinado con el atributo 'use' para asegurar que ciertos elementos JavaScript se cargan de forma correcta y son accesibles.

```
<aui:script use="liferay-portlet-url,aui-base" position="inline">

   bootstrapRouter('<portlet:namespace />main', '<portlet:namespace
/>');

</aui:script>
```

Dentro del método *bootstrapRouter* se harán o llamarán a todas las funciones relacionadas con Angular.

Id se enlaza con *ng-app* manualmente. Como se muestra a continuación, *ng-app* se inicializa manualmente con el método *angular.bootstrap()*, que toma el id único

del elemento HTML y el id del *portlet*. Con este método se puede definir más de un ng-app en la misma página.

```
angular.bootstrap(document.getElementById(id),[id]);
```

Además, para el correcto visualizado de la aplicación ha sido necesario crear un tema personalizado para hacer que el estilo *aui* que impone por defecto *Liferay* no se tenga en cuenta en ciertos aspectos y así poder utilizar el *CSS* generado teniendo en cuenta solamente la aplicación *AngularJS*.

- **Inicialización de la aplicación**

El trozo de código que se muestra abajo inicializa la aplicación y registra los módulos de los que depende.

```
angular.module('F1FeederApp', [
  'F1FeederApp.services',
  'F1FeederApp.controllers',
  'ui.router',
  'ncy-angular-breadcrumb'
]).
```

F1FeederApp es el nombre del módulo que se cargará al iniciar la aplicación, F1FeederApp.services contiene el código de los servicios utilizados por la API, F1FeederApp.controllers contiene el código de los controladores.

Ui.router es un *framework* de ruteo Single Page Application del lado de cliente para *AngularJS* que sustituye al que tiene por defecto. Este *framework* actualiza la URL del navegador a medida que el usuario navega a través de la aplicación. Esto permite realizar cambios a la URL del navegador para navegar a través de la aplicación, lo que permite crear marcadores en lugares más profundos del SPA. Las aplicaciones *ui-router* están modeladas como un árbol de estados jerárquico. *Ui-router* provee una máquina de estados para controlar las transiciones entre los estados de la aplicación.

Ncy-Agular-breadcrumb es un módulo para *AngularJS* que genera un *breadcrumb* para cualquier página dentro de la aplicación. Está basado en el *framework ui-router* y su árbol de estados jerárquico.

- **Ruteo**

Para definir una ruta es necesario el uso del método .config, $*stateProvider* provee interfaces para declarar estados en la aplicación. Añadiendo con inyección la dependencia *ui.router* a la aplicación se utilizará este módulo en lugar del original.

```
angular.module('F1FeederApp', [
  'F1FeederApp.services',
  'F1FeederApp.controllers',
  'ui.router',
```

```
  'ncy-angular-breadcrumb'
]).
config(['$urlRouterProvider','$stateProvider',
function($urlRouterProvider, $stateProvider) {
  $urlRouterProvider.otherwise('/drivers');
  $stateProvider
      .state('código…'
      .state('races', {url:'/races', templateUrl:
'partials/races.html',
        controller: 'racesController', ncyBreadcrumb: {label: 'Races',
parent:'drivers'}})
}]);
```

En este caso, *.state* registra una configuración de estado para un nombre dado. Para el estado de la aplicación *races*, la URL que se mostrará en el navegador es */races*, esta es la URL en la que se encuentra la aplicación en el estado especificado. La plantilla de la vista *races.html* está enlazada al controlador *racesController*. Además, para poder crear un *breadcrumb*, tiene una etiqueta con el nombre que mostrará y el nombre del estado padre del cual depende.

Las vistas se muestran dentro del *tag* <div ui-view> , que se encuentra en el *body* del HTML principal.

- **Controladores**

La variable $*scope* que se pasa como parámetro al controlador une el controlador con la vista de la que se encarga. Esta variable contiene todos los datos que serán utilizados dentro de la plantilla. Cualquier elemento añadido al $*scope* será directamente accesible desde las vistas.

```
angular.module('F1FeederApp.controllers', []).
controller('racesController', function($scope, ergastAPIservice) {
    $scope.gpFilter = null;
    $scope.searchFilter = function (race) {
        var re = new RegExp($scope.gpFilter, 'i');
        return !$scope.gpFilter || re.test(race.raceName);
    };
    $scope.racesList = [];

    ergastAPIservice.getRaces().success(function (response) {
        $scope.racesList = response.MRData.RaceTable.Races;
    });

    $scope.compareDates = function(date) {
        var today = new Date();
        var raceDate = new Date(date);
        return (today >= raceDate);
    }
}).
```

- **Servicios**

El servicio *factory* genera un único objeto que muestra el servicio al resto de la aplicación. El objeto retornado por el servicio se inyecta en el componente especificado por el servicio del cual depende, en este caso F1FeederApp.services.

```
angular.module('F1FeederApp.services', [])
  .factory('ergastAPIservice', function($http) {

    var ergastAPI = {};

    ergastAPI.getQualifications = function(id) {
    return $http({
        cache: true,
        method: 'JSONP',
        url: 'http://ergast.com/api/f1/current/circuits/'+ id
+'/qualifying.json?callback=JSON_CALLBACK'
      });
    }
  return ergastAPI;
});
```

AngularJS ofrece el servicio $http para facilitar la comunicación con servidores HTTP a través del objeto del navegador XMLHttpRequest o vía JSONP. El servicio $http realiza una petición HTTP GET al servidor y este responde devolviendo los datos en formato JSONP.

- **Icono de load mientras cargan las tablas**

Para no mostrar las tablas de las vistas vacías de contenido, mientras no se disponga del *response* de la API y se pueblen las tablas aparecerá un icono de carga en su lugar.

El controlador de cada vista tiene definida una variable *elemento*Loaded que estará inicializada como *false*, una vez se dispongan de los datos esta variable pasará a ser *true* dejando así de verse el icono de carga y pudiendo así mostrar el contenido.

Variable definida a false en el controlador.

```
controller('grandPrixController', function($scope, $stateParams,
ergastAPIservice) {
  $scope.qualificationsLoaded = false;
```

Variable pasa a ser true una vez se dispone de los datos.

```
ergastAPIservice.getQualifications($scope.id).success(function
(response) {
    $scope.qualifications = response.MRData.RaceTable.Races[0];
    $scope.qualificationsLoaded = true;
});
```

En caso de que la variable qualificationsLoaded sea *false* se ejecutará la directiva ng-hide, mostrando así el icono de carga. Si por el contrario su valor es *true* se ejecutará la directiva ng-show, mostrando así el contenido HTML correspondiente a la tabla.

```
<div> <!-- <div ng-show="qualificationsLoaded" class="ng-hide"> -->
    <p ng-hide="qualificationsLoaded" class="text-center">
        <i class="fa fa-spinner fa-pulse fa-2x fa-fw margin-
bottom"></i>
        <span class="sr-only">Loading...</span>
    </p>
    <table ng-show="qualificationsLoaded" id="qualRes" class="table-
hover" border="1">
```

La directiva *ng-show* muestra o esconde el contenido del elemento HTML basándose en la expresión proporcionada en el atributo *ng-show*. El elemento se muestra o se esconde eliminando o añadiendo la clase CSS *.ng-hide* al elemento. La clase CSS *.ng-hide* está predefinida en *AngularJS* y establece estilo de visualización a ninguno, usando el *flag ¡important.* En caso de evaluarse negativamente, entonces la clase CSS *.ng-hide* se añade a la clase atributo del elemento en cuestión haciendo que no se muestre. Cuando se evalúa a positivo, la clase CSS *.ng-hide* se elimina del elemento, causando que este no se muestre escondido.

En el caso de la directiva *ng.hide* ocurre lo contrario, al evaluarse a verdadero se añade la clase CSS *.ng-hide* y cuando se evalúa a falso la clase CSS *.ng-hide* se elimina.

- **Cacheo de los datos**

Realizar el cacheo con Angular resulta sencillo, para habilitar el cacheo de una *request* HTTP dándole valor *true* a la propiedad cache de configuración del objeto $http.

```
ergastAPI.getQualifications = function(id) {
    return $http({
        cache: true,
        method: 'JSONP',
        url: 'http://ergast.com/api/f1/current/circuits/'+ id
+'/qualifying.json?callback=JSON_CALLBACK'
    });
}
```

Ahora Angular cacheará la respuesta objeto HTTP en la caché objeto $http por defecto. Este objeto puede obtenerse usando el método *get* en la $*cachefactory* y pasándole su $http como primer parámetro.

```
var httpCache = $cachefactory.get('http');
```

Los elementos del servicio $*cachefactory* se guardan como pares *key*-valor. Cuando se cache una petición $http la URL especificada en la configuración del objeto $https se utiliza como *key* para el valor cacheado. Para obtener el objeto, simplemente hay que llamar al método GET en la $*httpCache* y pasar la URL como primer parámetro. El valor que es cacheado es el objeto *response* http, y ese será el valor retornado.

```
var cachedResponse = httpCache.get('api/path);
```

- **Barra de búsqueda**

Para crear un campo de búsqueda que filtre los resultados mostrados por las tablas, primeramente se ha añadido dentro del código HTML correspondiente a la visa deseada, un texto simple de búsqueda que filtrará los resultados.

```
<div id="cabecera">
    <div id="breadcrumb" ncy-breadcrumb></div>
    <input id="search" type="text" ng-model="gpFilter"
placeholder="Search..."/>
</div>
```

Gracias al uso de la directiva *ng-model*, se enlaza el texto que haya en el campo de búsqueda con la variable $*scope.gpFilter*, así su valor siempre estará actualizado con el valor del input.

La línea de código HTML que se encarga de poblar las filas de la tabla contiene un filtro para el *array racesList* que filtra los valores que se mostrarán según el valor guardado en *searchFilter*, esto ocurre antes del *output* de los datos.

```
<tr ng-repeat="race in racesList | filter: searchFilter">
```

En este punto gracias al *two-way data binding*, cada vez que un valor se introduce en el campo de búsqueda, Angular inmediatamente se asegura de que el $scope.gpFilter asociado sea actualizado con el nuevo valor. Ya que el enlace funciona en ambos sentidos, en el momento en que el valor de *gpFilter* se actualice, la segunda directiva asociada a él, en este caso *ng-repeat*, también recibirá este valor y la vista de actualizará de forma inmediata.

```
controller('racesController', function($scope, ergastAPIservice) {
    $scope.gpFilter = null;
    $scope.searchFilter = function (race) {
        var re = new RegExp($scope.gpFilter, 'i');
        return !$scope.gpFilter || re.test(race.raceName);
    };
```

9. Líneas Futuras

Tras la realización de este proyecto, se dispone de un prototipo básico de *portlet* que utiliza la tecnología SPA. Pero que solamente es capaz de mostrar datos obtenido de una API externa. Para poder integrar completamente la tecnología SPA en el entorno *portlet* de *Liferay* es necesario ahondar más en el proyecto para hacer compatibles más funcionalidades de *Liferay* con esta forma de crear webs y así no hacer uso solamente de las funcionalidades propias de *AngularJS* sino de ambos.

10. Conclusiones

Tras la realización de este proyecto se han conseguido alcanzar los objetivos definidos al inicio del mismo. Ha sido interesante ver cómo funciona y cómo crear una aplicación web como son las SPA, obteniendo como resultado una web sobre la cual navegar de vista a vista se realiza en un instante.

Además integrar esta aplicación en *Liferay* ha sido un reto ya que son tecnologías contrapuestas, al ser necesario desde *Liferay* actualizar todos los *portlets* y recargar la web cada vez que se realiza una interacción sobre un único *portlet* mientras que la tecnología SPA precisamente busca el no recargar constantemente la web.

Además, la forma de realizar el proyecto ha sido muy similar a la realidad, ya que el proyecto ha seguido la metodología *SCRUM*, que es la utilizada en la empresa para la que se ha realizado. Dicha metodología solo la conocía de forma teórica y tras haber trabajado con ello ha resultado ser una metodología interesante y útil.

11. Bibliografía

[1] Tejero, Javier - PA, un paradigma de Arquitectura de Aplicaciones Web en auge

https://cink.es/blog/2013/10/07/spa-un-paradigma-de-arquitectura-de-aplicaciones-web-en-auge/

[2] JavaScript

https://www.javascript.com/

[3] Ajax

http://api.jquery.com/jquery.ajax/

[4] HTML 5

http://www.w3schools.com/html/html5_intro.asp

[5] XML

http://www.w3schools.com/xml/

[6] JSON

http://www.json.org/

[7] Document Object Domain
https://es.wikipedia.org/wiki/Document_Object_Model

[8] Gartner technology

http://www.gartner.com/technology/home.jsp

[9] Gartner Magic Cuadrant
http://www.gartner.com/technology/research/methodologies/research_mq.jsp

[10] NodeJS

https://nodejs.org/en/

[11] Liferay portal

https://www.liferay.com/es/

[12] AngularJS

https://angularjs.org/

[13] Garcia, Oscar - AngularJs, o el arte de extender HTML para desarrollar Single-Page Applications

http://www.elclubdelprogramador.com/2015/07/30/angularjs-o-el-arte-de-extender-html-para-desarrollar-single-page-applications/

[14] Estudios de remuneración 2016

http://www.pagepersonnel.es/sites/pagepersonnel.es/files/er_tecnologia16.pdf

12. Glosario

API: Application Programming Interface, inteficie software que especifica como diferentes componentes informáticos pueden interactuar entre ellos.

Crawler: programa que inspecciona las páginas del *World Wide Web* de forma metódica y automatizada.

DOM: Document Object Model, interface de programación de aplicaciones para documentos HTML y XML, una convención multiplataforma e independiente del lenguaje de programación que sirve para presentar documentos HTML.

Framework: Infraestructura de programación que, en programación orientada a objetos, facilita la concepción de las aplicaciones mediante el uso de las bibliotecas de clases o generadores de programas.

Gestor de contenido: programa que permite crear una estructura de soporte para la creación y administración de contenidos de una web.

GIT: software de control de versiones pensando para la eficiencia y el mantenimiento de código fuente.

JSON: estándar abierto basado en texto diseñad para el intercambio de datos leíble para humanos para representar estructuras de datos simples y listas asociativas.

LOPD: ley orgánica de protección de datos de carácter personal que tiene como objetico garantizar y proteger los datos personales, las libertades públicas y los derechos fundamentales de las personas físicas, y especialmente su intimidad y privacidad personal y familiar.

Portal: plataforma software para construir aplicaciones y páginas web. Ha de incluir multitud de características, por eso son adecuados para desarrollar una amplia variedad de aplicaciones.

Portlet: componente modular de las interfaces de un portal. Producen fragmentos de código que se agregan a la página.

Product backlog: lista de requisitos funcionales ordenada por prioridades y definida por el equipo de desarrollo en una motología SCRUM.

Product owner: representa los *stakeholders* y es la voz del cliente.

SCRUM: metodología de desarrollo de software ágil que se basa en procesos iterativos e incrementales.

Sprint: iteración de la metodología SCRUM y unidad básica de desarrollo. La duración de un sprint está fijada antes de su inicio y suele durar entre una semana y un mes.

Sprint review: al finalizar un sprint, el equipo se reúne para evaluar el estado del sprint superado y defnir el siguiente.

Stakeholders: grupo de personas que tienen un interés concreto en el soporte de una organización.

String: tipo de estructura de datos que contiene una secuencia de caracteres con un orden y un tamaño determinados.

Anexo I - Instalar Liferay Portal 6.2

Instalar *Java* Developer Kit (JDK)

Un requisito previo para instalar *Liferay* es instalar primero el JDK. Iremos a http://*Java*.sun.com y descargar versión actual, en este caso **1.8.0_91**.

Hay que tener en cuenta que este documento está pensado para el sistema operativo Windows y por ello nos es necesario indicarle al S.O desde donde tiene que ejecutar el JDK, por ello habrá que indicárselo añadiendo una variable de entorno:

Ir al panel de control, ejecutamos el icono sistema. Ir a avanzado, y al botón de las variables de entorno. Agregar una variable de sistema llamada *JAVA*_HOME y como valor, el directorio donde se instaló el JDK. (Ilustración 12).

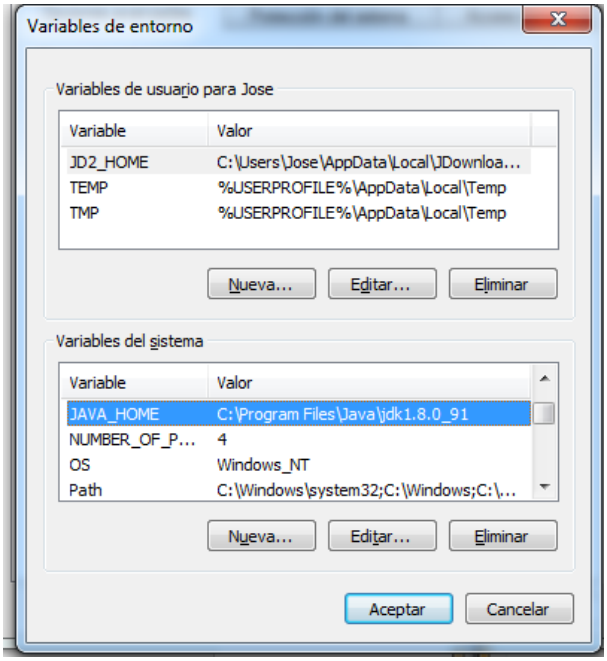
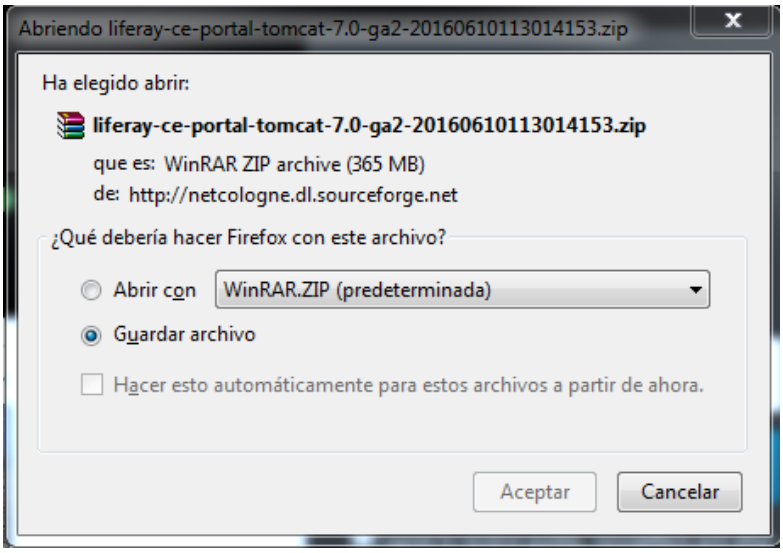


Ilustración 12- JAVA_HOME

Si ya hemos desarrollado con *Java* probablemente este variable ya la tenemos añadida, en ese caso nos aseguraremos de que la versión sea la misma.

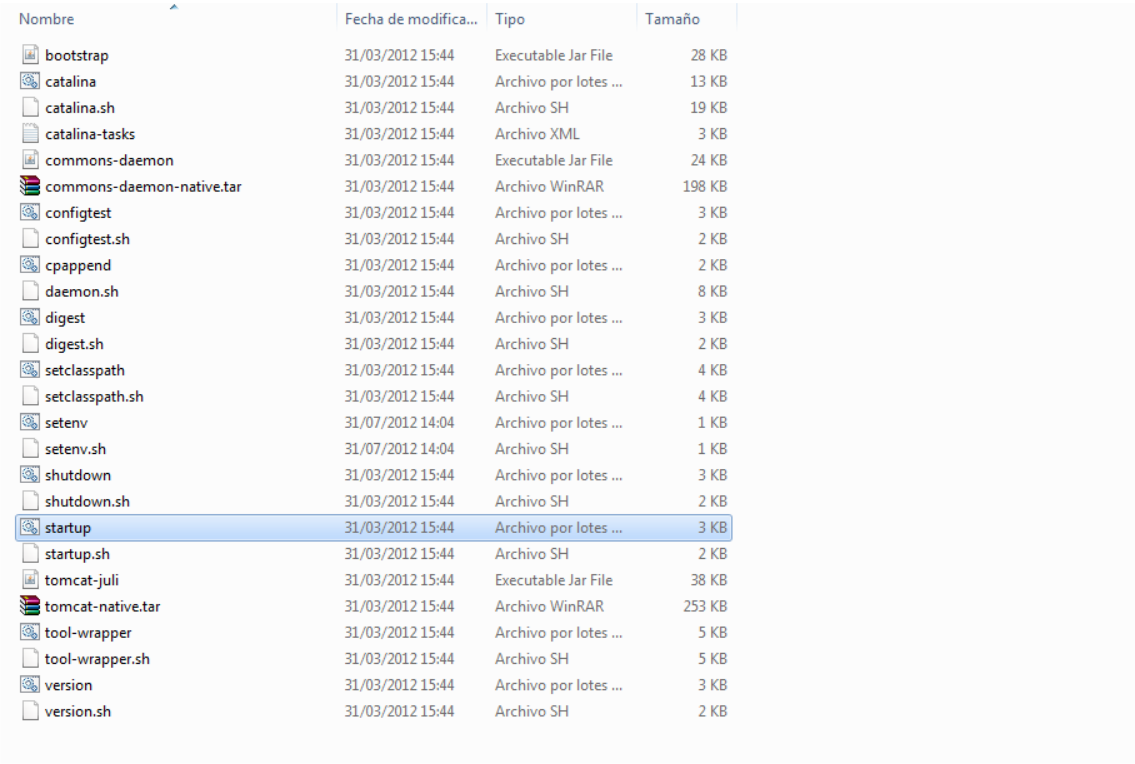
Instalar Liferay

Para instalar *Liferay* en nuestro pc tendremos que ir a la URL: http://sourceforge.net/projects/lportal/files/latest/download



Esta dirección nos permitirá bajar la última versión de *Liferay*. Este archivo es de tipo bundle, esto nos permite instalarlo en todos los SO más comunes, Windows, Linux etc... , el bundle permite que *Liferay* Portal sea portable, una vez descomprimido tendremos dentro de esa carpeta todo lo necesario para que *Liferay* Portal funcione.

Una vez bajado lo descomprimimos en la raíz C:\.Ahora lo arrancaremos, pare poder hacerlo utilizaremos el explorador de Windows e iremos a la carpeta **bin** que está en el path:" C:*Liferay*-portal-6.1.1-ce-ga2\tomcat-7.0.27\bin", es en esta carpeta donde tenemos todos los ejecutables, en Windows serán los .bar y en Linux .sh.(Fig 35).



Nombre	Fecha de modifica...	Tipo	Tamaño
bootstrap	31/03/2012 15:44	Executable Jar File	28 KB
catalina	31/03/2012 15:44	Archivo por lotes ...	13 KB
catalina.sh	31/03/2012 15:44	Archivo SH	19 KB
catalina-tasks	31/03/2012 15:44	Archivo XML	3 KB
commons-daemon	31/03/2012 15:44	Executable Jar File	24 KB
commons-daemon-native.tar	31/03/2012 15:44	Archivo WinRAR	198 KB
configtest	31/03/2012 15:44	Archivo por lotes ...	3 KB
configtest.sh	31/03/2012 15:44	Archivo SH	2 KB
cpappend	31/03/2012 15:44	Archivo por lotes ...	2 KB
daemon.sh	31/03/2012 15:44	Archivo SH	8 KB
digest	31/03/2012 15:44	Archivo por lotes ...	3 KB
digest.sh	31/03/2012 15:44	Archivo SH	2 KB
setclasspath	31/03/2012 15:44	Archivo por lotes ...	4 KB
setclasspath.sh	31/03/2012 15:44	Archivo SH	4 KB
setenv	31/07/2012 14:04	Archivo por lotes ...	1 KB
setenv.sh	31/07/2012 14:04	Archivo SH	1 KB
shutdown	31/03/2012 15:44	Archivo por lotes ...	3 KB
shutdown.sh	31/03/2012 15:44	Archivo SH	2 KB
startup	31/03/2012 15:44	Archivo por lotes ...	3 KB
startup.sh	31/03/2012 15:44	Archivo SH	2 KB
tomcat-juli	31/03/2012 15:44	Executable Jar File	38 KB
tomcat-native.tar	31/03/2012 15:44	Archivo WinRAR	253 KB
tool-wrapper	31/03/2012 15:44	Archivo por lotes ...	5 KB
tool-wrapper.sh	31/03/2012 15:44	Archivo SH	5 KB
version	31/03/2012 15:44	Archivo por lotes ...	3 KB
version.sh	31/03/2012 15:44	Archivo SH	2 KB

Para arrancar Tomcat 7 y con ello *Liferay* 6.2 habrá que ejecutar el archivo **startup.bat.**

Una vez llegado aquí ya tenemos el portal *Liferay* 6.2 arrancado y preparado para empezar a trabajar.

Para acceder al portal: **http://localhost:8080**

User/Pass**: test/test@*Liferay*.com**

Instalar Eclipse

Para instalar *Eclipse* necesitamos descargarlo de esta dirección:

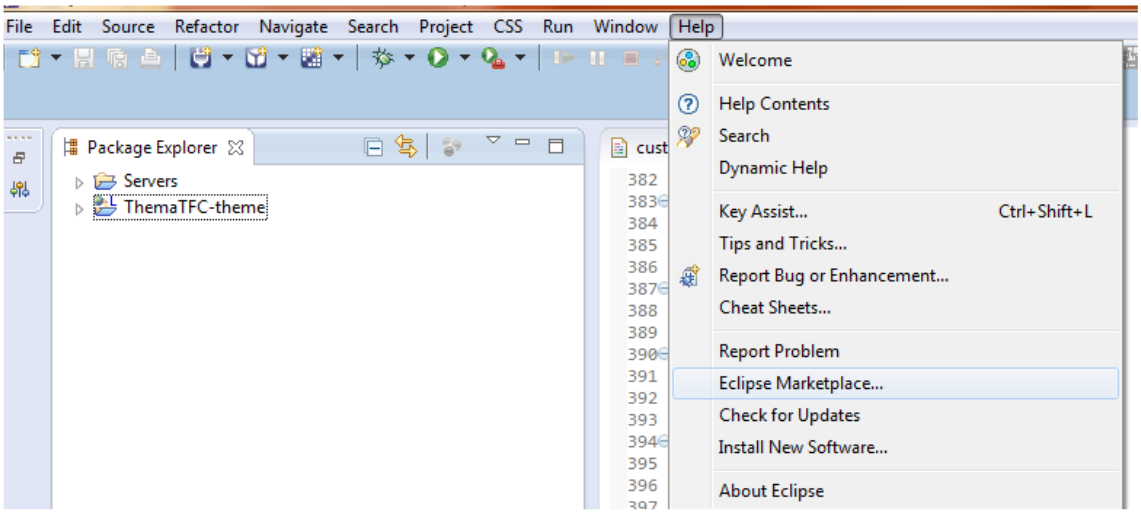
http://www.*Eclipse*.org/downloads/packages/*Eclipse*-ide-*Java*-ee-developers/indigosr2

Como se muestra en la Fig 36 tendremos varias opciones de descarga según el S.O (Opciones de la derecha "Download Links"). Elegiremos la que coincida con el nuestro.

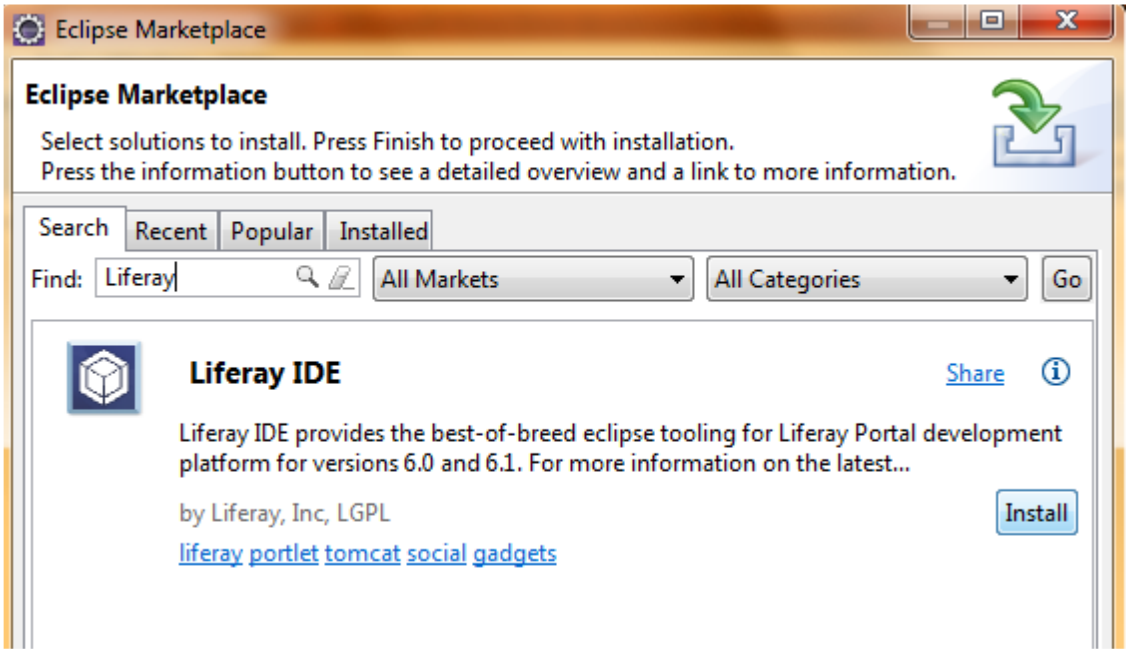
El tipo de archivo que descargamos es del tipo bundle así que únicamente tendremos que descomprimirlo y ya lo tendremos listo para trabajar.

Plugin Liferay

Para poder desarrollar componentes para *Liferay* es necesario adaptar *Eclipse*, para ello no es necesario instalar este *plugin*.



Como muestrala Fig 27, para instalar este *plugin* utilizaremos el Marketplace de *Eclipse*.” Help->Eclispe Marketplace…”



Una vez dentro añadimos al buscador el *plugin Liferay* y nos mostrara un resultado. *Liferay* IDE. A la derecha nos aparecerá la opción de instalar. Clicamos encima de esta y se nos instalara. Para poder empezar a utilizarlo habrá que reiniciar *Eclipse*.

Anexo II - Creación de un portlet

En muchas ocasiones los *portlets* predefinidos que encontramos en *Liferay* o en el *MarketPlace* de *Liferay* no se adaptan en absoluto a nuestras necesidades reales, por ello, necesitaríamos crear nuestro propio *portlet*.

Para este proyecto se ha creado el *portlet* F1 angular. Para crear un nuevo *portlet*, primero debemos crear un *Liferay plugin Project* de tipo *portlet* donde en su interior podremos crear un *Liferay portlet*. Tras nombrar y crear el nuevo *portlet*, se nos crearán los directorios siguientes:

- **Docroot/WEB-INF/src**: Para definir los paquetes *java*.

- **Css**: Para definir todo el *css* utilizado en las vistas *jsp*.

- **Html**: Para definir todas las vistas con archivos *jsp*.

- **Js**: Para definir los J*avaScripts* utilizados en las vistas *jsp*.

- **WEB-INF**

·**Lib**: Aquí se deben incluir todas las librerías utilizadas.

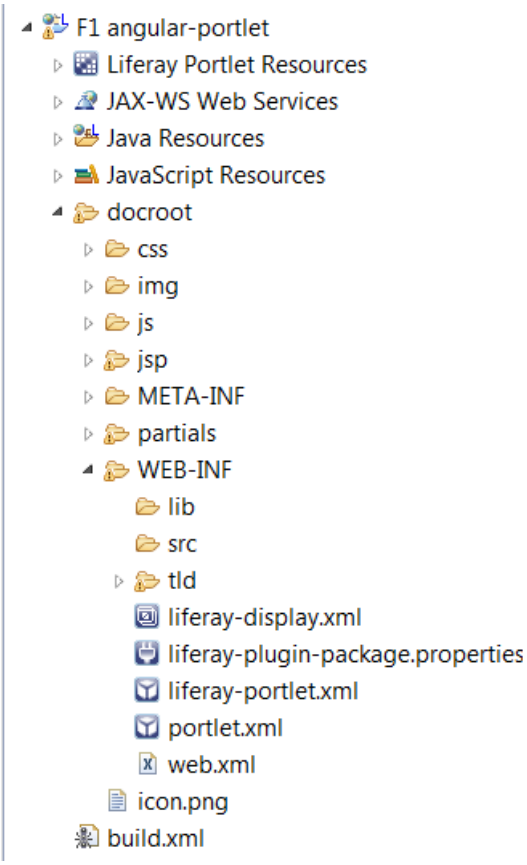
·**Tld** (*Tag Library Documentation Generator*): Aquí se deben incluir todas las librerías de etiquetas.

·**Liferay-display.xml**: Aquí se define la categoría y el identificador del portlet.

·**Liferay-portlet.xml**: Aquí se definen las características del *portlet* como son su nombre, la instanciabilidad y las referencias a los archivos *javascript* y *css*.

·**Portlet.xml**: Aquí se definen las referencias a los archivos java y a las vistas *jsp* a mostrar.

A continuación se muestra una captura de ejemplo del *F1 angular-portlet*.



Como podemos comprobar, *Liferay* facilita el trabajo generando automáticamente las carpetas necesarias para poder programar de forma estructurada.

Tras haber completado nuestro *portlet*, únicamente deberemos arrastrarlo al servidor para que este se ejecute.

Anexo III – Estándar *portlet*s JSR 168 y JSR 286

Los estándares de *portlets* realizados por Sun, definen unas funciones que deben implementar los *portlets*.

Un *portlet* es una aplicación que se puede ejecutar en una página del portal. Cada página utiliza esa aplicación contiene una instancia del *portlet.* Las diferentes instancias permiten que se pueda configurar de forma diferente en cada página.

Las instancias de *portlet* guardan preferencias, estas preferencias son la configuración propia de la instancia de *portlet*.

Hay dos funcionalidades de los *portlet*s que es interesante conocer: "Window state" y "*Portlet* Mode".

- *Portlet* Mode: Selecciona el tipo de visualización que se va a lanzar, sirve para separar diversas funciones como la visualización, edición o la ayuda.

- Window state: Esta funcionalidad le indica al *portlet* de cuanto espacio dispone para mostrar el contenido, normal, maximizado o minimizado.

Las funciones básicas de un *portlet* y las tareas que realizan son las siguientes:

- **Ini():** Inicializa el *portlet.*
- **processAction():** Es llamado cuando el usuario realiza alguna acción sobre el *portlet*, por ejemplo rellenar un formulario.
- **render():** Se ejecuta cada vez que se tiene que refrescar el *portlet* (La implementación *Java*x.*portlet*.Generic*Portlet* recupera las cabeceras de la petición con el metodo "doHeaders" y el titulo con el metodo "getTitle" e invoca el método "doDispatch").
    - **doHeaders**
    - **getTitle**
    - **doDispatch:** esta función se encarga de redirigir la petición hacia el modo correspondiente según el estado en que se encuentra el *portlet*. Por defecto son los siguientes, aunque se pueden crear nuevos.

      - doView

      - doEdit

      −doHelp

- **destroy():** elimina el objeto *Java* para que pueda ser eliminado de memoria

Todas estas funciones deben ser implementadas para realizar un *portlet*, ya que *Javax.portlet.GenericPortlet* solo actúa de interface en muchas de ellas.