CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Facultad de Informática

TRABAJO FIN DE GRADO

Gestión de ideas innovadoras

Autor: Javier González Benito 080119
Director: Oscar Corcho

MADRID, JUNIO DE 2013

"He aprendido que todo el mundo quiere vivir en la cima de la montaña, sin saber que la verdadera felicidad está en la forma de subir la escarpada"

Gracias a todos.

Tabla de contenido

1.- Resumen en castellano 6
2.- Resumen en inglés 7
3.- Preámbulo y objetivos de la aplicación 8
4.- Modelo de datos 10
4.1- MongoDB 10
4.2- MySQL (Liferay) 16
5.- API RESTful 19
5.1- Categories 19
5.2-Category 23
5.3- Ideas 26
5.4- Idea 30
5.5- Comments 33
5.6- Comment 37
5.7- Social 39
6.- Diseños técnicos 47
6.1- Estructura del proyecto 47
6.2- Visión general del sistema 47
6.3- Arquitectura 49
6.4- Spring y perfiles de la aplicación 50
6.5- Sistema de trazas “logs” 54
6.6- Capas de la aplicación 55
6.6.1- Capa de repositorios 55
6.6.2- Capa de Servicios 59
6.6.3- Capa de controladores 60
6.6.3- Capa de Vistas 66
6.6.4.- Portlets 72
7.- Descripción funcional de la aplicación 75
7.1- Toolbar 77
7.2- Social Top 79
7.2.1- Ideas más valoradas 79
7.2.2- Ideas más comentadas 80
7.3- Last Ideas 81
7.3.1- Crear idea 82

7.4- Category Ideas 84
7.5- Idea 86
7.5.1- Añadir comentario 88
7.6- User Panel 89
7.7- Internacionalización (i18n) 91
8.- Sistema de pruebas 91
8.1- Creado y borrado de datos para las pruebas 91
8.2- Pruebas para Repositorios y Servicios 95
8.3- Pruebas para Controladores 96
8.4- Pruebas SmartGWT 99
9.- Liferay 100
9.1- Tema 100
9.2- Layout 102
9.3- Configuración 103
10.- Seguridad 105
10.1- Filtro 106
10.2- Gestión de contenido 108
11.- Conclusión y líneas futuras 109
12.- Bibliografía 112

Resumen (castellano-inglés)

1.- Resumen en castellano

El trabajo de fin de grado que se va a definir detalladamente en esta memoria, trata de poner de manifiesto muchos de los conocimientos que he adquirido a lo largo de la carrera, aplicándolos en un proyecto real.

Se ha desarrollado una plataforma capaz de albergar ideas, escritas por personas de todo el mundo que buscan compartirlas con los demás, para que estas sean comentadas, valoradas y entre todos poder mejorarlas. Estas ideas pueden ser de cualquier ámbito, por tanto, se da la posibilidad de clasificarlas en las categorías que mejor encajen con la idea.

La aplicación ofrece una API RESTful muy descriptiva, en la que se ha identificado y estructurado cada recurso, para que a través de los "verbos http" se puedan gestionar todos los elementos de una forma fácil y sencilla, independientemente del cliente que la utilice.

La arquitectura está montada siguiendo el patrón de diseño modelo vista-controlador, utilizando las últimas tecnologías del mercado como Spring, Liferay, SmartGWT y MongoDB (entre muchas otras) con el objetivo de crear una aplicación segura, escalable y modulada, por lo que se ha tenido que integrar todos estos frameworks.

Los datos de la aplicación se hacen persistentes en dos tipos de bases de datos, una relacional (MySQL) y otra no relacional (MongoDB), aprovechando al máximo las características que ofrecen cada una de ellas.

El cliente propuesto es accesible mediante un navegador web, se basa en el portal de Liferay. Se han desarrollado varios "Portlets o Widgets", que componen la estructura de contenido que ve el usuario final. A través de ellos se puede acceder al contenido de la aplicación, ideas, comentarios y demás contenidos sociales, de una forma agradable para el usuario, ya que estos "Portlets" se comunican entre sí y hacen peticiones asíncronas a la API RESTful sin necesidad de recargar toda la estructura de la página. Además, los usuarios pueden registrarse en el sistema para aportar más contenidos u obtener roles que les dan permisos para realizar acciones de administración.

Se ha seguido una metodología "Scrum" para la realización del proyecto, con el objetivo de dividir el proyecto en tareas pequeñas y desarrollarlas de una forma ágil. Herramientas como "Jenkins" me han ayudado a una integración continua y asegurando mediante la ejecución de los test de prueba, que todos los componentes funcionan.

La calidad ha sido un aspecto principal en el proyecto, se han seguido metodologías software y patrones de diseño para garantizar un diseño de calidad, reutilizable, óptimo y modulado. El uso de la herramienta "Sonar" ha ayudado a este cometido. Además, se ha implementado un sistema de pruebas muy completo de todos los componentes de la aplicación.

En definitiva, se ha diseñado una aplicación innovadora de código abierto, que establece unas bases muy definidas para que si algún día se pone en producción, sirva a las personas para compartir pensamientos o ideas ayudando a mejorar el mundo en el que vivimos.

2.- Resumen en inglés

The Final Degree Project, described in detail in this report, attempts to cover a lot of the knowledge I have acquired during my studies, applying it to a real project.

The objective of the project has been to develop a platform capable of hosting ideas from people all over the world, where users can share their ideas, comment on and rate the ideas of others and together help improving them. Since these ideas can be of any kind, it is possible to classify them into suitable categories.

The application offers a very descriptive API RESTful, where each resource has been identified and organized in a way that makes it possible to easily manage all the elements using the HTTP verbs, regardless of the client using it.

The architecture has been built following the design pattern model-view-controller, using the latest market technologies such as Spring, Liferay, Smart GWT and MongoDB (among others) with the purpose of creating a safe, scalable and adjustable application.

The data of the application are persistent in two different kinds of databases, one relational (MySQL) and the other non-relational (MongoDB), taking advantage of all the different features each one of them provides.

The suggested client is accessible through a web browser and it is based in Liferay. Various "Portlets" or "Widgets" make up the final content of the page. Thanks to these Portlets, the user can access the application content (ideas, comments and categories) in a pleasant way as the Portlets communicate with each other making asynchronous requests to the API RESTful without the necessity to refresh the whole page. Furthermore, users can log on to the system to contribute with more contents or to obtain administrator privileges.

The Project has been developed following a "Scrum" methodology, with the main objective being that of dividing the Project into smaller tasks making it possible to develop each task in a more agile and ultimately faster way.

Tools like “Jenkins” have been used to guarantee a continuous integration and to ensure that all the components work correctly thanks to the execution of test runs.

Quality has been one of the main aspects in this project, why design patterns and software methodologies have been used to guarantee a high quality, reusable, modular and optimized design. The “Sonar” technology has helped in the achievement of this goal. Furthermore, a comprehensive proofing system of all the application's components has been implemented.

In conclusion, this Project has consisted in developing an innovative, free source application that establishes a clearly defined basis so that, if it someday will be put in production, it will allow people to share thoughts and ideas, and by doing so, help them to improve the World we live in.

Introducción y objetivos

3.- Preámbulo y objetivos de la aplicación

En este proyecto trato de demostrar muchos de los conocimientos que he adquirido gracias a los estudios realizados en la carrera Grado en Ingeniería Informática en la universidad Politécnica de Madrid, dónde sin duda he encontrado lo que me gusta.

En los últimos años, en el mundo de las tecnologías de información han aparecido millones de nuevas ideas que han cambiado el mundo, aportando continuas mejoras para la sociedad.

Cada vez se hace más difícil encontrar nuevas ideas innovadoras que aporten de verdad algo nuevo, y no sean una mera copia de las ya existentes. Es por esto, que se utilizan métodos como "Brainstorming", en la que varios individuos se juntan para proponer muchas ideas básicas, a las que posteriormente se le van aplicando diversos métodos para ir creando ideas más completas y con mayor sentido.

Pero, y ¿por qué no crear una plataforma dónde usuarios de todo el mundo puedan acceder y compartir ideas con otras personas, y así eliminar la limitación de estar físicamente reunidos?, esta es la idea base de la aplicación diseñada.

Actualmente existen aplicaciones similares, como "Ideas4All", "UserVoice", "Laboratorio de ideas de Bankinter" etc., son plataformas muy grandes que contienen demasiada información y se hace muy difícil encontrar lo que estás buscando.

El objetivo del proyecto es diseñar una aplicación accesible mediante un navegador web a través de internet, en el que cualquier usuario pueda acceder, registrarse y proponer todas las ideas que considere oportunas, con el objetivo de que otros usuarios las lean, las valoren y las comenten (aspectos sociales).

Para que estas ideas estén bien clasificadas y puedan encontrarse con facilidad, se ha diseñado una estructura de categorías y subcategorías a las que se asocian todas las ideas creadas.

La aplicación ofrece una API RESTful muy descriptiva, en la que se han identificado cada recurso para que a través de los verbos del protocolo HTTP, puedan ser accesibles de una forma muy intuitiva (estilo arquitectónico RESTful). Esta API puede ser utilizada desde cualquier cliente que quiera obtener los datos existentes en el sistema.

También se ha diseñado un cliente, que se basa en portal Liferay, compuesto por una serie de "Portlets o Widgets" independientes, que interactúan con la API para obtener los datos de las ideas y demás contenidos, con el objetivo de mostrarlos a los usuarios de una forma amigable, sencilla e intuitiva.

En todo el diseño de la aplicación se ha tenido muy en cuenta aspectos de calidad, seguridad, escalabilidad, aplicando diversas metodologías del software. Además, se ha utilizado las últimas tecnologías del mercado en cuanto a "frameworks JAVA", imprescindibles a la hora de diseñar cualquier aplicación.

Los objetivos generales conseguidos han sido:

- Base para un sistema de gestión de ideas, comentarios y aspectos sociales.
- Diseño de una API RESTful para acceder a los datos.
- Aplicación web, que da acceso a los usuarios para interactuar con las ideas y demás contenidos.
- Aspectos sociales, entre los que se incluyen puntuaciones, ideas más comentadas, últimas ideas etc.
- Sistema de pruebas para toda la aplicación.
- Integración de las últimas tecnologías JAVA del mercado.
- Calidad, seguridad, modularidad, escalabilidad.

Todos estos contenidos, se van a describir detalladamente en el contenido de esta memoria.

Desarrollo

4.- Modelo de datos

El modelo de datos está dividido en dos partes. La parte de gestión de usuarios, roles y grupos de usuario se va a gestionar a través de una base de datos relacional, en concreto MySQL ya que el cliente propuesto se basa en el portal Liferay, que ya cuenta con estos datos en su modelo. Más adelante explicaré toda la integración realizada.

En cuanto a la lógica de negocio, se ha propuesto realizarla sobre una base de datos no relacional (MongoDB).

4.1- MongoDB

En los últimos años, ha aparecido un nuevo paradigma en cuanto al almacenamiento de datos, en concreto son las bases de datos no relacionales (NoSQL) que difieren en muchos aspectos en cuanto a las bases de datos tradicionales, no tienen "schemas", no permiten "JOINs" y no intentan garantizar ACID (atomicidad, consistencia, aislamiento y durabilidad) entre otras cosas.

MongoDB es una base de datos no relacional, la estructura de datos que podemos guardar en ella está en formato JSON, con un esquema dinámico llamado BSON, lo que implica que no existe un esquema predefinido.

A diferencia de las bases de datos relacionales, MongoDB guarda los datos en "Documentos", que a su vez se almacenan en "colecciones". Estos documentos están compuestos por pares "clave-valor".

La aplicación que he desarrollado, tiene tanto lectura como escritura de datos, siendo la lectura lo más importante, ya que queremos mostrar los datos lo más rápido posible, para que la experiencia de usuario sea lo más satisfactoria posible.

MongoDB mantiene el máximo número de datos en memoria RAM, esto hace que el tiempo de acceso a los datos sea muy rápido, así las consultas son muy veloces. Además, está orientado al desarrollo web, ofreciendo de un modelado de datos natural, que ayuda al programador ofreciéndole un desarrollo más fácil, rápido y sencillo.

Por todas estas razones MongoDB ha sido una base de datos adecuada para el desarrollo del proyecto.

Como se mencionó en la introducción, la aplicación se centra en tres conceptos fundamentales, "**Ideas**", "**Comentarios**" y "**Categorías**" y en menor medida, se encuentran los "**aspectos sociales**".

Estos datos son los que componen la lógica de negocio, y será de vital importancia hacerlos persistentes en nuestro sistema. Para ello, y como hemos descrito, utilizaremos la base de datos MongoDB.

Como estamos tratando con una base de datos no relacional, no se puede diseñar el típico modelo "**entidad-relación**" presente en cualquier diseño de modelo de datos típico, ya que no son "tablas" con lo que vamos a trabajar, sino, colecciones de documentos. En cambio se ha establecido unas premisas base, para posteriormente poder modelar los datos.

Estas premisas son:

- Las ideas son la colección principal de la aplicación.
- Existirán "n" categorías principales, que estarán predefinidas.
- Existirán "n" subcategorías, cada una de ellas, estará asociada a una categoría principal.
- Cada una de las ideas, estará asociada a una subcategoría.
- Cada idea, tendrá asociados contenidos sociales.
- Cada comentario, estará asociado a una idea.

Una vez definidos estas premisas clave, procedemos a dividirlos en colecciones y documentos, según especifica la estructura de datos de MongoDB.

Se han dividido en tres colecciones:

- **Ideas (CfgIdeas):** Esta colección representa las ideas, sus campos son:
  - **Identificador de usuario en Liferay:** Puesto que la gestión de usuarios está desarrollada con el portal Liferay (lo veremos a continuación) es necesario almacenar el identificador de usuario que crea las ideas.
  - **Nombre de usuario:** Nombre del usuario creador de la idea.
  - **Fecha*:** Fecha de creación de la idea.
  - **Título de la idea.**
  - **Contenido de la idea.**
  - **Imagen de perfil del usuario:** Ruta relativa dónde se almacena la imagen del perfil de usuario en Liferay.
  - **Subcategoría:** Referencia a otro documento, que se encuentra en otra colección (CfgToolbarCategory).
  - **Social:** Documento embebido que contiene los siguientes aspectos sociales:

- **Valoraciones:** Número de valoraciones que tiene la idea.
- **Comentarios:** Número de comentarios que tiene la idea.

***Fecha:** En un futuro la idea de la aplicación es que puedan acceder usuarios de todo el mundo para aportar sus ideas, lo que implica que cada usuario tenga una zona horaria distinta. Es por esto, que la hora que almacenamos se transforma a "GMT+0", para tener todas las ideas en la misma zona horaria, después, cuando los usuarios solicitan las ideas esta fecha se convierte al "GMT" que tenga el usuario.

- **Categorías(CfgToolbarCategory):** Esta colección representa las categorías y subcategorías a las que se asocian las ideas, contienen los siguientes campos:
  - **Categoría padre:** Es una referencia a otra categoría. Es un elemento opcional, si un elemento lo posee, se trata de una subcategoría, y esta es la referencia a la categoría principal. Por tanto, las categorías principales no tendrán valor en este campo.
  - **Recurso:** Este es un campo para asociarle una imagen representativa, pero que no se ha utilizado en la implementación. Se utilizará en futuras mejoras.

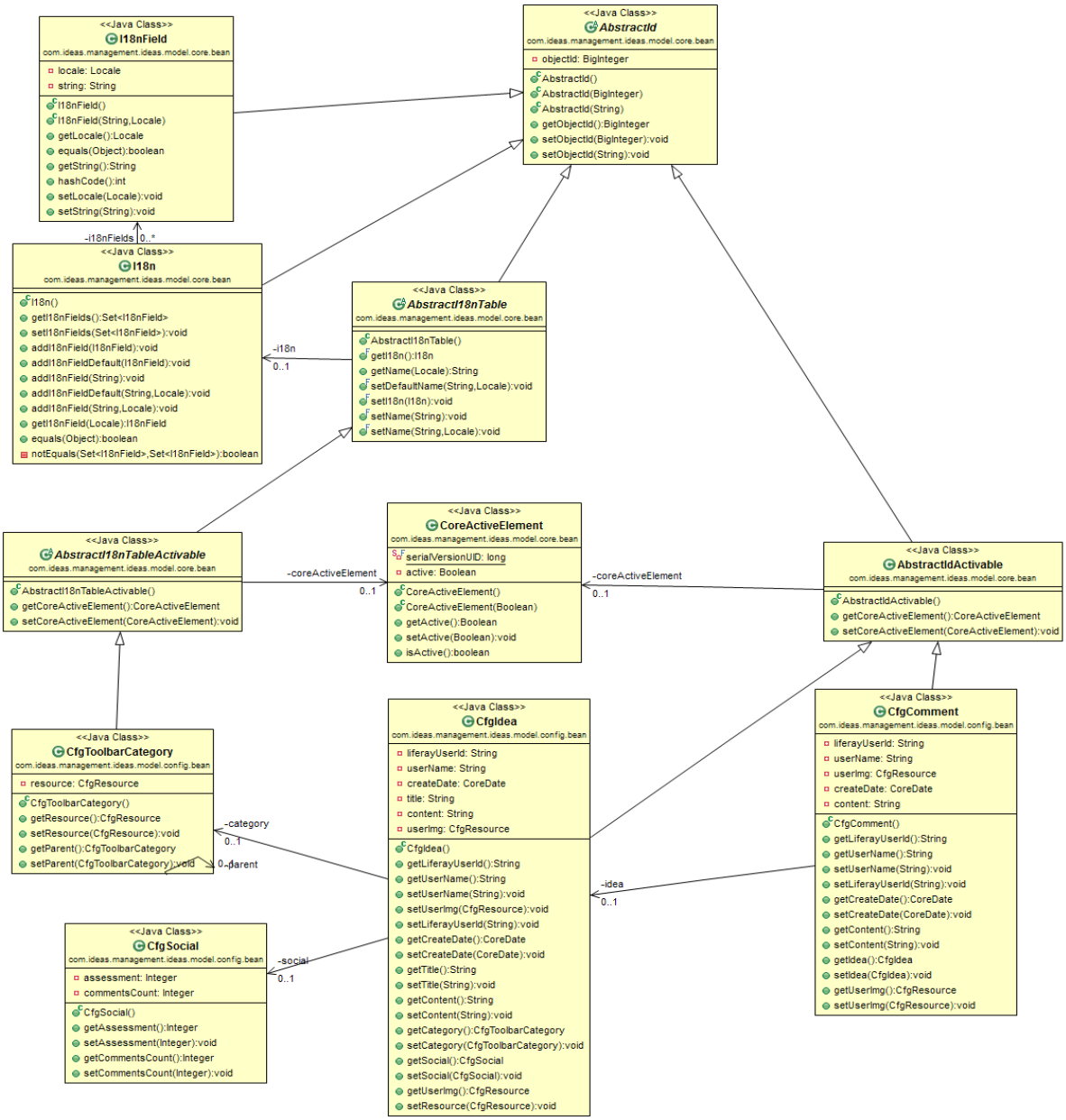
- **Comentarios (CfgComment):** Esta colección representa los comentarios que los usuarios realizan en cuanto al contenido de una idea. Los campos que posee son:
  - **Identificador de usuario en Liferay:** Puesto que la gestión de usuarios está desarrollada con el portal Liferay (lo veremos a continuación) es necesario almacenar el identificador de usuario que crea las ideas.
  - **Nombre de usuario:** Nombre del usuario creador del comentario.
  - **Imagen de perfil del usuario:** Ruta relativa dónde se almacena la imagen del perfil de usuario en Liferay.
  - **Fecha:** Fecha de creación del comentario (en "GMT+0" como se comentó anteriormente).
  - **Contenido del comentario**
  - **Idea:** Referencia a un documento Idea, de la colección de ideas, a la que está asociado este comentario.

Hay que destacar, que los campos indicados anteriormente para cada colección no son los únicos que poseen. Se ha creado una serie de premisas

a partir de las cuales se ha generado el modelo correspondiente, estas premisas son:

- Todo elemento con el que tratemos tiene que poder activarse/desactivarse, para poder realizar un borrado lógico.
- Es necesario que cada elemento tenga un identificador único, al igual que cualquier base de datos.
- Los elementos que puedan tener internacionalización (18n) tendrán la posibilidad de guardar el nombre en "n" idiomas.

A partir de estas premisas, que son comunes a las colecciones mencionadas, es el momento de mostrar el modelo de datos, realizado directamente en clases de java con anotaciones de "Spring-data-mongo" que explicaré a continuación.



Como podemos observar en la estructura propuesta, se ha creado en la parte inferior las tres colecciones principales del modelo (categorías, ideas y comentarios). Por encima de ellas, se ha creado una serie de decoradores, uno para cada premisa establecida anteriormente:

- Para que todo elemento se pueda activar, todos los elementos heredan "CoreActiveElement" a través de "AbstractI18nTableActivable" y "AbstractIdActivable".
- Las categorías son las únicas que pueden tener internacionalización, por tanto son las únicas que heredan de "AbstractI18nTable", dónde se han definido una estructura para que pueda tener una lista de nombres asociados a "Locales".
- Todo elemento hereda de AbstractId, para tener un identificador único.

Además de la estructura mostrada en el dibujo, existen una serie de interfaces que no se muestran. Hay que destacar, que todas las colecciones heredan de "IDocument" y los documentos embebidos en otro documento, heredan de "IDocumentField". Aunque esta interfaz, no obliga a implementar nada, tiene una razón de existencia. La idea es poder tener polimorfismo en todos los elementos de la base de datos, porque en el sistema de pruebas que he desarrollado, se utiliza para insertar o borrar datos sin necesidad de saber con qué colección estamos tratando. Lo explicaré con más detalle en el sistema de pruebas. Además, todos implementan "Serializable" ya que estos objetos van a ser transmitidos por la red hasta el servidor de MongoDB.

Como framework de desarrollo he utilizado "Spring-data-mongo", con él, añadimos una capa de abstracción que se encarga de utilizar el driver de mongo para java. Utilizar este driver directamente habría sido muy pesado porque es bastante elemental.

Con este framework conseguimos manejar cómodamente la base de datos, sin preocuparnos por la integridad referencial. Además nos aporta los métodos típicos de "CRUD", junto con muchas otras funcionalidades, sin olvidar, que al ser de Spring, se integra con todas las funcionalidades del "core", que he utilizado en todo el proyecto.

El diagrama de clases que he descrito, cuenta con una serie de anotaciones java en sus atributos, definidas por "Spring-data-mongo" para mapear los objetos java a JSON (BSON concretamente), que recordemos, son las estructuras de datos con las que trabaja MongoDB, y por lo tanto, todo objeto que queramos almacenar en una colección, debe estar en este formato.

Ejemplo:

```
@Document
public class CfgIdea extends AbstractIdActivable implements IDocument{

	/** The liferay user id. */
	@NotNull
	@Field
	private String liferayUserId;

	/** The user name. */
	private String userName;

	/** The create date. */
	@NotNull
	@Field
	private CoreDate createDate;

	/** The title. */
	@NotNull
	@Field
	private String title;

	/** The content. */
	@NotNull
	@Field
	private String content;

	/** The category. */
	@NotNull
	@DBRef
	private CfgToolbarCategory category;

	/** The social. */
	@NotNull
	@Field
	private CfgSocial social;

	/** The user img. */
	@NotNull
	@Field
	private CfgResource userImg;

	…….
```

Este ejemplo corresponde al "bean" de Ideas. Como podemos observar, están presentes todos los atributos que hemos definido anteriormente. Todo ellos están anotados con "@Field", esta anotación indica que a la hora de ser mapeados estos objetos Java a JSON, este atributo va a ser un campo del JSON, siguiendo el modelo clave-valor, por ejemplo:

```
@Field
private String liferayUserId;
```

{ "liferayUserId" : "12346" }

Los atributos que van anotados con "@Id", que además de seguir el mismo concepto de mapeo que "@Field", sirve para indicar que es el identificador del documento, esto quiere decir que en una colección, no pueden existir dos documentos que tengan como identificador el mismo valor:

```
@Id
private BigInteger objectId;
```

Todos estos "bean" están anotados como "@Document", con esta anotación conseguimos que el objeto y todos los atributos que contenga (y estén correctamente anotados) sean convertidos a un documento mongo, en formato BSON.

```
@Document
public class CfgComment extends AbstractIdActivable implements IDocument{
```

4.2- MySQL (Liferay)

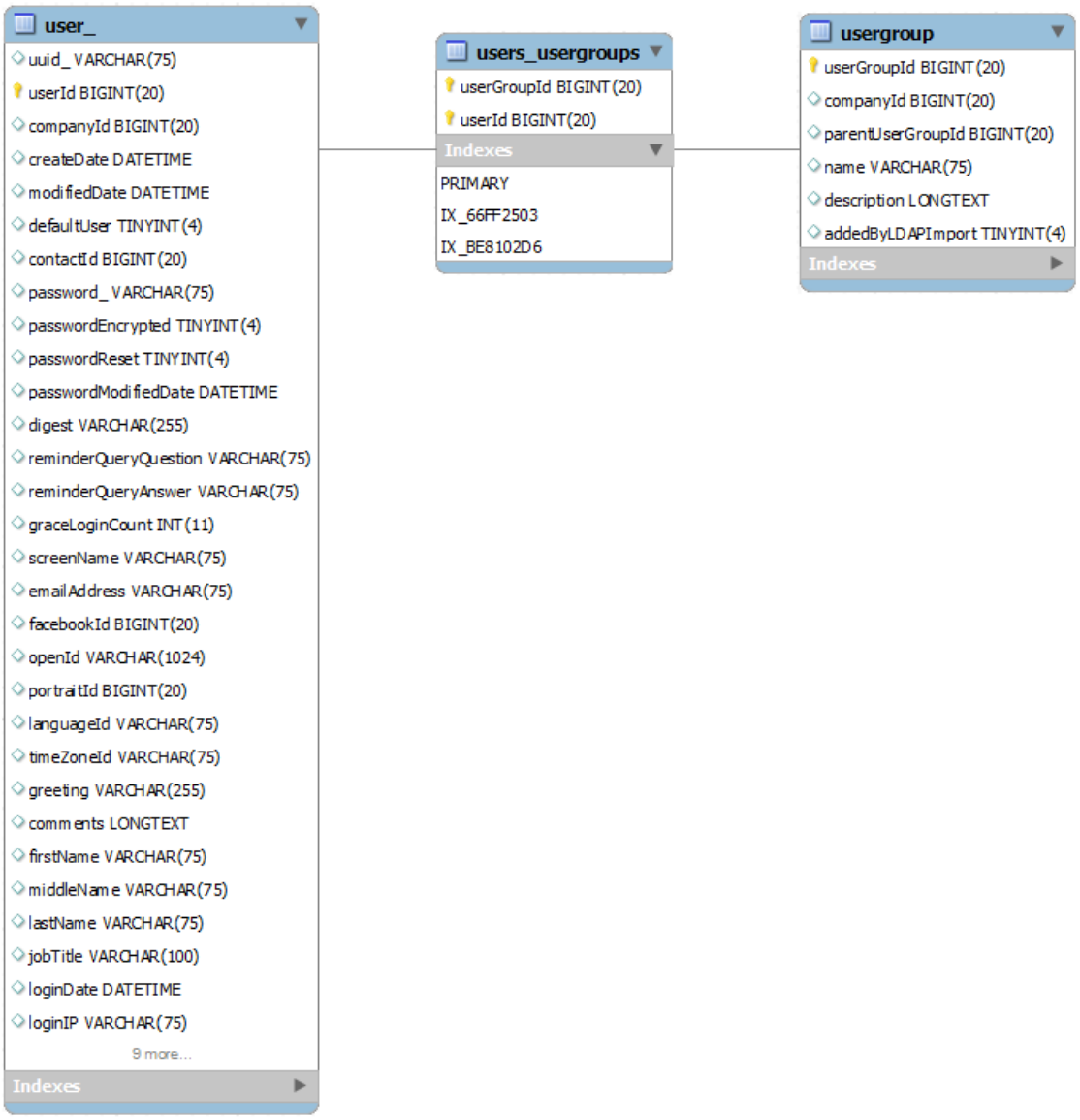
A través de MongoDB hemos preparado la aplicación para albergar toda la lógica de negocio, pero la aplicación también necesita gestión de usuarios, ya que necesitamos que se registren y creen ideas, comentarios etc. aparte de controlar los roles que tiene cada usuario para permitirles el acceso a unos datos u otros.

El término portal en el campo de las tecnologías de la información está en pleno auge, es por esto, por lo que me he decidido basar la implementación del cliente propuesto (lo definiré más adelante) en Liferay, un portal desarrollado en JAVA que ofrece una plataforma independiente del sistema operativo y de la base de datos.
Liferay nos proporciona gran cantidad de características como gestión de usuarios (roles, grupos…), paneles de administración, Portlets ya desarrollados, como "Blog", "Foro o "Gestión de contenido multimedia" que se pueden añadir en la aplicación. Además, permite modificar su contenido "core" por una implementación propia.

Proporciona un entorno seguro, de alta escalabilidad, desplegable en la nube y disponible como SaaS. Soporta una gran cantidad de usuarios y páginas visitadas diariamente.

Liferay utiliza Hibernate como capa de abstracción de la base de datos, por lo que puede utilizar cualquiera. Yo he basado la implementación en MySQL.

El modelo de datos que posee es el siguiente (es mucho más grande pero la implementación del proyecto se basa solo en estas tablas):



- **User_:** En esta tabla se almacenan los usuarios que se registren en la aplicación. Los usuarios que accedan no tienen que registrarse obligatoriamente pues si lo que quieren es leer ideas y comentarios pueden hacerlo. En cambio, si lo que desean es crear ideas o comentarios, necesitarán registrarse. Tanto para el "login de usuario" como para el registro de nuevos usuarios utilizaremos el portlet "Login" de Liferay.
- **Usergroup**: Esta tabla representa los grupos de usuarios. Los utilizaremos en la aplicación para dividir en grupos a los usuarios, y en función de en qué grupo se encuentren, tendrán unos privilegios u otros.
    - **No registrados**: Estos son los usuarios que acceden a la aplicación pero no se registran. Solamente podrán realizar acciones de lectura.

- **Registrados**: Estos usuarios pertenecerán al grupo "user" y tendrán acceso a lectura de datos y escritura de ideas y comentarios.
- **Administradores**: Estos usuarios además de los permisos del anterior grupo, podrán realizar acciones de administración, cómo borrar ideas. Pertenecerán al grupo "admin".

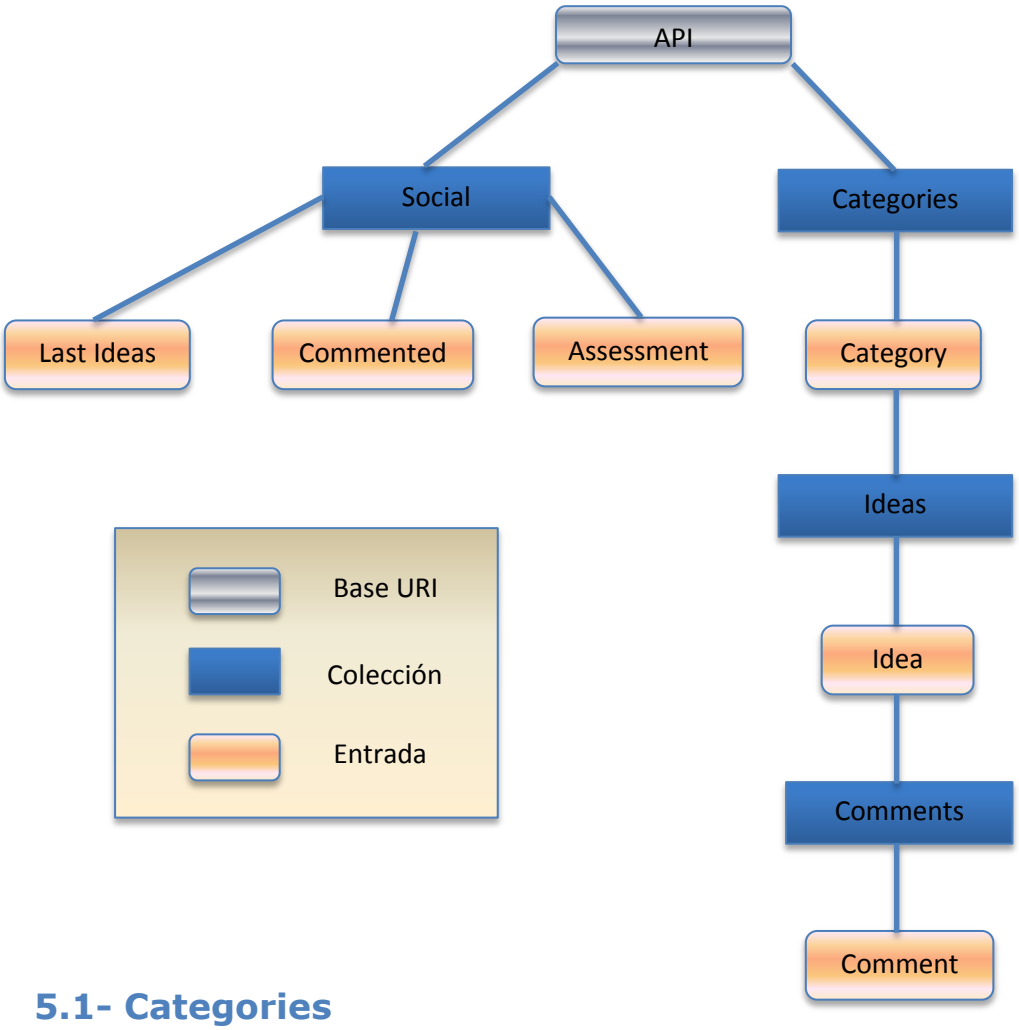
Cuando se especifique individualmente las vistas del cliente desarrollado se indicará las posibilidades que tiene cada tipo de usuario.

- **Users_usergroups**: La relación entre las dos tablas anteriores da lugar a esta tercera tabla, ya que Liferay da la posibilidad de que un usuario pueda estar en varios grupos a la vez. Sin embargo, en la aplicación, un usuario sólo estará en un grupo a la vez.

La creación de estos grupos se hará directamente desde el panel de administración del portal.

5.- API RESTful

A partir del modelo de datos definido, se ha diseñado la siguiente API RESTful, para que cualquier tipo de cliente pueda acceder a la información del sistema. Esta es la especificación:



5.1- Categories

Este recurso corresponde a una colección de categorías/subcategorías, las peticiones que podemos realizar son:

URI	/api/categories
Método	GET
Parámetros	parent={ID} Filtro por categoría padre*
Devuelve	200: Ok + POX (application/JSON)
	401: Unauthorized
	404: Not Found

Este recurso devuelve una lista de categorías, cada una con su correspondiente información. Posee un parámetro de consulta ("parent"), con él, podemos pedir las subcategorías que pertenecen a una categoría maestra.

El contenido del JSON de respuesta contiene los siguientes campos:

- **toolbarElementList** : Lista de categorías
- **categoryDataId**: Identificador de la categoría
- **elementName**: Nombre de la categoría

*Para solicitar las categorías maestras, el parámetro "parent" tiene que tomar el valor "0".

Ejemplo:

Petición:

Todas las subcategorías que pertenecen a la categoría con identificador "1".

GET /api/categories?parent=1

Resultado JSON:


```
{
        "toolbarElementList": [
        {
                "categoryDataTo": {
                        "categoryDataId": "252519059046512373093546955 89"
                },
                "elementName": "Cultura"
        }
        {
                "categoryDataTo": {
                        "categoryDataId": "252519059046512373093546951 53"
                },
                "elementName": "Cines"
        }]
}
```


URI	/api/categories
Método	POST
Cuerpo de la petición	POX (application/JSON)
Devuelve	201 Created y POX (application/JSON)
	401 Unauthorized
	415 Unsupported Media Type

Una petición Http POST a este recurso, crea una categoría o subcategoría.

El contenido del JSON de petición contiene los siguientes campos:

- **name**: Nombre de la subcategoría
- **fatherId**: Identificador de la categoría maestra a la que la asociamos*.

*Para crear una categoría maestra, hay que asociarle como "fatherId" el valor "0".

Ejemplo:

Petición:

Creamos una subcategoría llamada "Fútbol" y la asociamos a una categoría maestra, que tiene como identificador el "6". Nos devuelve el identificador de la subcategoría creada.

POST /api/categories

Cuerpo JSON:

```
{
        "name":"Fútbol",
        "fatherId":"6"
}
```

Resultado JSON:

```
{
         "id":"252617315602343210747997006 27"
}
```

URI	/api/categories
Método	PUT
Devuelve	405: Not Supported

El método Http PUT no está soportado en la API.

URI	/api/categories
Método	DELETE
Devuelve	405: Not Supported

El método Http DELETE no está soportado en la API.

5.2-Category

Este recurso corresponde a una categoría o subcategoría, las peticiones que podemos realizar son:

URI	/api/categories/{categoryId*}
Método	GET
Devuelve	200: Ok + POX (application/JSON)
	401: Unauthorized
	404: Not Found

*categoryId: Identificador de la categoría o subcategoría

Una petición Http GET a este recurso nos devuelve un JSON con la información asociada a la categoría. Contiene los siguientes campos:

- **toolbarElementList** : Lista de categorías
- **categoryDataId**: Identificador de la categoría
- **elementName**: Nombre de la categoría

Ejemplo:

Petición:

*Categoría o subcategoría con identificador "**252617315602343210747997006 27**".*

GET /api/categories/252617315602343210747997006 27

Resultado JSON:


```
{
        "categoryDataTo": {
                "categoryDataId": "252617315602343210747997006 27"
        },
        "elementName": "Cultura"
}
```


URI	/api/categories/{categoryId*}
Método	PUT
Cuerpo de la petición	POX (application/JSON)
Devuelve	201 Created y POX (application/JSON)
	401 Unauthorized
	415 Unsupported Media Type

*categoryId: Identificador de la categoría o subcategoría

Una petición Http PUT a este recurso, sirve para modificar una categoría o subcategoría que se ha creado con anterioridad. Es necesario que el cuerpo de la petición contenga el siguiente campo:

- **elementName**: Nombre de la categoría

Ejemplo:

Petición:

*Categoría o subcategoría con identificador "**252617315602343210747997006 27**".*

PUT /api/categories/25261731560234321074799700627

Cuerpo JSON:

```
{
        "elementName": "Cultura Nuevo"
}
```

URI	/api/categories/{categoryId*}
Método	DELETE
Devuelve	204: No content
	401: Unauthorized
	404: Not Found

*categoryId: Identificador de la categoría o subcategoría

Una petición Http DELETE borra este recurso.

URI	/api/categories/{categoryId*}
Método	POST
Devuelve	405: Not Supported

*categoryId: Identificador de la categoría o subcategoría

El método Http POST no está soportado en la API.

5.3- Ideas

Este recurso corresponde a una colección de ideas, las cuales pertenecen a una categoría o subcategoría. Las acciones que se pueden realizar son:

URI	/api/categories/{categoryId*}/ideas	
Método	GET	
Parámetros	Page={Integer}	Paginación
Devuelve	200: Ok + POX (application/JSON)	
	401: Unauthorized	
	404: Not Found	

*categoryId: Identificador de la categoría o subcategoría

Una petición Http GET a este recurso nos devuelve las ideas que están asociadas a la categoría indicada en la petición. Una categoría puede contener muchas ideas, por eso se proporciona un parámetro de consulta ("page") que sirve para dividir en "páginas o trozos" las ideas que pertenecen a una categoría. Cada página está compuesta por 10 ideas.

El contenido del JSON de respuesta contiene los siguientes campos:

- **count** : Número de ideas que existen en esta categoría.
- **ideas**: Lista de ideas, cada una con los siguientes campos:
  - **id**: Identificador.
  - **tittle:** Título.
  - **user:** Usuario creador de la idea.
  - **category:** Categoría y subcategoría a la que pertenece
  - **ago:** Fecha en la que se creó la idea
  - **userImg:** Ruta a la imagen de perfil del creador de la idea.
  - **categoryId:** Identificador de la subcategoría a la que pertenece.
  - **Assessment:** Número de valoraciones que posee.
  - **commentsCount:** Número de comentarios que posee.

Además contiene dos campos de información en cuanto a los roles del usuario que realiza la petición:

- **admin**: Campo que indica si el usuario es un administrador.
- **registered**: Campo que indica si es un usuario registrado.

Ejemplo:

Petición:

> *Ideas que pertenecen a la categoría con identificador "***25251907177476578395313757095***".*
>
> *Pedimos la primera página.*

GET /api/categories/25251907177476578395313757095/ideas?page=0

Resultado JSON:


```
{
        "count": 1,
        "ideas": [{
                "id": "252519071774765783953137570 96",
                "title": "Protección geolocalizada para las mujeres maltratadas. ",
                "user": "Javier Gonzalez",
                "category": "Humanidad > Maltratos",
                "ago": "18/05/2013",
                "userImg": "/image/user_male_portrait?img_id=10706",
                "categoryId": "25251907177476578395313757095",
                "assessment": 1,
                "commentsCount": 3
        }],
        "admin": false,
        "registered": false
}
```


URI	/api/categories/{categoryId*}/ideas
Método	POST
Cuerpo de la petición	POX (application/JSON)
Devuelve	201 Created y POX (application/JSON)
	401 Unauthorized
	415 Unsupported Media Type

*categoryId: Identificador de la categoría o subcategoría

Una petición Http POST a este recurso, sirve para crear una idea nueva, asociándola a una subcategoría. El contenido de la petición debe ser un JSON con los siguientes campos:

- **tittle:** Título de la idea.
- **content:** Contenido de la idea.

El contenido de la respuesta, es un campo ("**created**") que nos indica si la idea se creó correctamente.

Ejemplo:

Petición:

> *Creamos una idea asociándola a la categoría con identificador"***2523156023410747997006***".*

POST /api/categories/2523156023410747997006/ideas

Cuerpo JSON:

```
{
        "title": "aquí va el título",
        "content": "<div align=\"center\"><b>Esto es el contenido...<br></b></div>"
}
```

Respuesta:

```
{
        "created": true,
}
```

URI	/api/categories/{categoryId*}/ideas
Método	PUT
Devuelve	405: Not Supported

El método Http PUT no está soportado en la API.

URI	/api/categories/{categoryId*}/ideas
Método	DELETE
Devuelve	405: Not Supported

El método Http DELETE no está soportado en la API.

5.4- Idea

Este recurso corresponde a una idea, las peticiones que podemos realizar son:

URI	/api/categories/{categoryId*}/ideas/{ideaId*}
Método	GET
Devuelve	200: Ok + POX (application/JSON)
	401: Unauthorized
	404: Not Found

*categoryId: Identificador de la categoría o subcategoría

*ideaId: Identificador de la idea

Una petición Http GET a este recurso nos devuelve un JSON con la información asociada a la idea. Contiene los siguientes campos:

- **tittle**: Título de la idea.
- **content**: Contenido de la idea.
- **user**: Usuario que creó la idea.
- **category:** Category y subcategoría a la que pertenece la idea.
- **Date:** Fecha de creación de la idea.
- **userImg:** Ruta a la imagen de perfil del usuario que creó la idea.
- **assessmentCount:** Número de valoraciones que posee la idea.

Además contiene un campo de información en cuanto al usuario que realiza la petición:

- **registered**: Campo que indica si es un usuario registrado.

Ejemplo:

Petición:

> *Solicitamos la idea con identificador "**7096**" que pertenece a la subcategoría con identificador "**2525**".*

GET /api/categories/2525/ideas/7096

Resultado JSON:


```
{

    "title": "Protección geolocalizada para las mujeres maltratadas. ",

    "content": "<p>Antes de empezar siento si mi idea puede resultar un poco polémica....</p>",

    "user": "Javier Gonzalez",

    "category": "Humanidad > Maltratos",

    "date": "18/05/2013",

    "userImg": "/image/user_male_portrait?img_id=10706",

    "assessmentCount": 2,

    "registered": false

}
```


URI	/api/categories/{categoryId*}/ideas/{ideaId*}
Método	PUT
Cuerpo de la petición	POX (application/JSON)
Devuelve	201 Created y POX (application/JSON)
	401 Unauthorized
	415 Unsupported Media Type

*categoryId: Identificador de la categoría o subcategoría

*ideaId: Identificador de la idea

Una petición Http PUT a este recurso, sirve para modificar una idea que se ha creado con anterioridad. Es necesario que el cuerpo de la petición contenga algunos de los siguientes campos:

- **tittle**: Título de la idea.
- **content**: Contenido de la idea.

Ejemplo:

Petición:

*Modificamos la idea con identificador "**7096**".*

PUT /api/categories/2525/ideas/7096

Cuerpo JSON:

```
{
        "tittle": "Nuevo título",
        "content": "Este es el nuevo contenido..."
}
```

URI	/api/categories/{categoryId*}/ideas/{ideaId*}
Método	DELETE
Devuelve	204: No content
	401: Unauthorized
	404: Not Found

*categoryId: Identificador de la categoría o subcategoría

*ideaId: Identificador de la idea

Una petición Http DELETE borra este recurso.

URI	/api/categories/{categoryId*}/ideas/{ideaId*}
Método	POST
Devuelve	405: Not Supported

*categoryId: Identificador de la categoría o subcategoría

*ideaId: Identificador de la idea

El método Http POST no está soportado en la API.

5.5- Comments

Este recurso corresponde a una colección de comentarios, los cuales pertenecen a una idea. Las acciones que se pueden realizar son:

URI	/api/categories/{categoryId*}/ideas/{ideaId*}/comments	
Método	GET	
Parámetros	Page={Integer}	Paginación
Devuelve	200: Ok + POX (application/JSON)	
	401: Unauthorized	
	404: Not Found	

*categoryId: Identificador de la categoría o subcategoría

*ideaId: Identificador de la idea

Una petición Http GET a este recurso nos devuelve los comentarios que están asociadas a la idea indicada en la petición. Una idea puede contener muchos comentarios, por eso, se proporciona un parámetro de consulta ("page") que sirve para dividir en "páginas o trozos" los comentarios que pertenecen a una idea. Cada página está compuesta por 10 comentarios.

El contenido del JSON de respuesta contiene una lista comentarios, cada uno con su descripción:

- **user**: Nombre del usuario que creó la idea.
- **userImg**: Ruta de la imagen de perfil del usuario que creó la idea.
- **content**: Contenido del comentario.
- **date:** Fecha en la que se creó el comentario.

Ejemplo:

Petición:

*Comentarios que pertenecen a la idea con identificador "***1564***".*

Pedimos la primera página.

GET /api/categories/9578/ideas/1564/comments?page=0

Resultado JSON:


```
[{
        "user": "Javier Gonzalez",
        "userImg": "/image/user_male_portrait?img_id=10706",
        "content": "Me encanta la idea!",
        "date": "18/05/2013"
},
{
        "user": "Catalina Iluminada",
        "userImg": "/image/user_male_portrait?img_id=0",
        "content": "A mí también",
        "date": "18/05/2013"
}]
```

URI	/api/categories/{categoryId*}/ideas/{ideaId*}/comments
Método	POST
Cuerpo de la petición	POX (application/JSON)
Devuelve	201 Created y POX (application/JSON)
	401 Unauthorized
	415 Unsupported Media Type

*categoryId: Identificador de la categoría o subcategoría

*ideaId: Identificador de la idea

Una petición Http POST a este recurso, sirve para crear un nuevo comentario, asociándolo a una idea. El contenido de la petición debe ser un JSON con el siguiente campo:

- **content:** Contenido del comentario.

El contenido de la respuesta, es un campo ("**created**") que nos indica si la idea se creó correctamente.

Ejemplo:

Petición:

*Creamos un comentario asociándolo a la idea con identificador"***1564***".*

POST /api/categories/9578/ideas/1564/comments

Cuerpo JSON:

```
{
        "content": "Aquí va el comentario…"
}
```

Respuesta:

```
{
        "created": true,
}
```

URI	/api/categories/{categoryId*}/ideas/{ideaId*}/comments
Método	PUT
Devuelve	405: Not Supported

El método Http PUT no está soportado en la API.

URI	/api/categories/{categoryId*}/ideas/{ideaId*}/comments
Método	DELETE
Devuelve	405: Not Supported

El método Http DELETE no está soportado en la API.

5.6- Comment

Este recurso corresponde a un comentario, las peticiones que podemos realizar son:

URI	/api/categories/{categoryId*}/ideas/{ideaId*}/comments/{commentId*}
Método	GET
Devuelve	200: Ok + POX (application/JSON)
	401: Unauthorized
	404: Not Found

*categoryId: Identificador de la categoría o subcategoría

*ideaId: Identificador de la idea

*commentId: Identificador de comentario

Una petición Http GET a este recurso nos devuelve un JSON con la información asociada al comentario. Contiene los siguientes campos:

- **content**: Contenido del comentario.
- **user**: Usuario que creó el comentario.
- **date:** Fecha de creación del comentario.
- **userImg:** Ruta a la imagen de perfil del usuario que creó el comentario.

Además contiene un campo de información en cuanto al usuario que realiza la petición:

- **registered**: Campo que indica si es un usuario registrado.

Ejemplo:

Petición:

> *Solicitamos el comentario con identificador "***7456***" que pertenece a la idea con identificador "***7096***".*

GET /api/categories/2525/ideas/7096/comments/7456

Resultado JSON:


```
{
        "content": “Me gusta la idea”,
        "user": "Javier Gonzalez",
        "date": "18/05/2013",
        "userImg": "/image/user_male_portrait?img_id=10706",
        "registered": false
}
```


URI	/api/categories/{categoryId*}/ideas/{ideaId*}/comments/{commentId*}
Método	DELETE
Devuelve	204: No content
	401: Unauthorized
	404: Not Found

*categoryId: Identificador de la categoría o subcategoría

*ideaId: Identificador de la idea

*commentId: Identificador de comentario

Una petición Http DELETE borra este recurso.

URI	/api/categories/{categoryId*}/ideas/{ideaId*}/comments/{commentId*}
Método	PUT
Devuelve	405: Not Supported

El método Http PUT no está soportado en la API.

URI	/api/categories/{categoryId*}/ideas/{ideaId*}/comments/{commentId*}
Método	POST
Devuelve	405: Not Supported

El método Http POST no está soportado en la API.

5.7- Social

URI	/api/social/
Método	GET
Devuelve	200: Ok + POX (application/JSON)
	401: Unauthorized
	404: Not Found

Este recurso corresponde a una lista de contenidos sociales de la aplicación.

Posee tres:

- Últimas ideas añadidas
- Ideas más valoradas
- Ideas más comentadas

Sólo son accesibles mediante el método Http GET, ya que son características que se calculan a partir de los recursos que anteriormente se han descrito (categorías, ideas y comentarios).

Ejemplo:

Petición:

GET /api/social

Resultado JSON:


```
[{
        "id": "assessment",
        "description": "Ideas  más  valoradas”
},
{
        "id": "commented",
        " description ": "Ideas  más  comentadas"
}
{
        "id": "lastideas",
        " description ": "Últimas ideas añadidas"
}]
```

URI	/api/social/assessment
Método	GET
Devuelve	200: Ok + POX (application/JSON)
	401: Unauthorized
	404: Not Found

Este recurso corresponde a la descripción de las tres ideas que son las más valoradas entre todas las ideas. La lista está ordenada, comenzando con la más valorada.

El JSON de respuesta contiene los siguientes campos:

- **id**: Identificador de la idea.
- **tittle**: Título de la idea.
- **user**: Usuario que creó la idea.
- **category**: Categoría y subcategoría a la que pertenece.
- **ago**: Fecha en la que se creó la idea.
- **userImg**: Ruta de la imagen de perfil del usuario creador de la idea.
- **categoryId**: Identificador de la subcategoría a la que pertenece.
- **assessment**: Número de valoraciones que posee la idea.
- **commentsCount**: Número de comentarios que posee la idea.

Ejemplo:

Petición:

GET /api/social/assessment

Resultado JSON:


```
[{
        "id": "252519598613776526909793172397",
        "title": "Reciclar es impresindible para la conservación de la Naturaleza.",
        "user": "octavio gonzález",
        "category": "Reciclado > Salud",
        "ago": "18/05/2013",
        "userImg": "/image/user_male_portrait?img_id=0",
        "categoryId": "252519598613776526909793172396",
        "assessment": 9,
        "commentsCount": 1
},
{
        "id": "252519059046512373093546955 90",
        "title": "Anillo Urbano Cultural",
        "user": "Javier Gonzalez",
        "category": "Cultura > Ocio",
        "ago": "18/05/2013",
        "userImg": "/image/user_male_portrait?img_id=10706",
        "categoryId": "252519059046512373093546955 89",
        "assessment": 8,
        "commentsCount": 2
},
{
        "id": "252519577215553403594851849 39",
        "title": "El papa francisco",
        "user": "caty iluminada",
        "category": "Papa > Humanidad",
        "ago": "18/05/2013",
        "userImg": "/image/user_male_portrait?img_id=12358",
        "categoryId": "252519444952398395097366762 65",
        "assessment": 5,
        "commentsCount": 1
}]
```


URI	/api/social/commented
Método	GET
Devuelve	200: Ok + POX (application/JSON)
	401: Unauthorized
	404: Not Found

Este recurso corresponde a la descripción de las tres ideas que son las más comentadas entre todas las ideas que existen. La lista está ordenada, comenzando con la más comentada.

El JSON de respuesta contiene los siguientes campos:

- **id**: Identificador de la idea.
- **tittle**: Título de la idea.
- **user**: Usuario que creó la idea.
- **category**: Categoría y subcategoría a la que pertenece.
- **ago**: Fecha en la que se creó la idea.
- **userImg**: Ruta de la imagen de perfil del usuario creador de la idea.
- **categoryId**: Identificador de la subcategoría a la que pertenece.
- **assessment**: Número de valoraciones que posee la idea.
- **commentsCount**: Número de comentarios que posee la idea.

Ejemplo:

Petición:

GET /api/social/commented

Resultado JSON:


```
[{
        "id": "252519598613776526909793172397",
        "title": "Reciclar es impresindible para la conservación de la Naturaleza.",
        "user": "octavio gonzález",
        "category": "Reciclado > Salud",
        "ago": "18/05/2013",
        "userImg": "/image/user_male_portrait?img_id=0",
        "categoryId": "252519598613776526909793172396",
        "assessment": 9,
        "commentsCount": 6
},
{
        "id": "252519059046512373093546955 90",
        "title": "Anillo Urbano Cultural",
        "user": "Javier Gonzalez",
        "category": "Cultura > Ocio",
        "ago": "18/05/2013",
        "userImg": "/image/user_male_portrait?img_id=10706",
        "categoryId": "252519059046512373093546955 89",
        "assessment": 8,
        "commentsCount": 2
},
{
        "id": "252519577215553403594851849 39",
        "title": "El papa francisco",
        "user": "caty iluminada",
        "category": "Papa > Humanidad",
        "ago": "18/05/2013",
        "userImg": "/image/user_male_portrait?img_id=12358",
        "categoryId": "252519444952398395097366762 65",
        "assessment": 5,
        "commentsCount": 1
}]
```


URI	/api/social/lastideas
Método	GET
Devuelve	200: Ok + POX (application/JSON)
	401: Unauthorized
	404: Not Found

Este recurso devuelve una descripción de las últimas diez ideas añadidas en la aplicación, ordenadas según la fecha de creación, la más reciente primero.

El JSON de respuesta contiene los siguientes campos:

- **count**: Número de ideas
- **ideas**: Lista con la descripción de las ideas, contiene los siguientes campos:
  - **id**: Identificador de la idea.
  - **tittle**: Título de la idea.
  - **user**: Usuario que creó la idea.
  - **category**: Categoría y subcategoría a la que pertenece.
  - **ago**: Tiempo transcurrido desde que se creó la idea.
  - **userImg**: Ruta de la imagen de perfil del usuario creador de la idea.
  - **categoryId**: Identificador de la subcategoría a la que pertenece.
  - **assessment**: Número de valoraciones que posee la idea.
  - **commentsCount**: Número de comentarios que posee la idea.

Además contiene un campo de información en cuanto al usuario que realiza la petición:

- **registered**: Campo que indica si es un usuario registrado.

Ejemplo:

Petición:

GET /api/social/lastideas

Resultado JSON:


```
{
	"registered": false,
	"count": 2,
	"ideas": [
	{
		"id": "252519071774765783953137570 96",
		"title": "Protección geolocalizada para las mujeres maltratadas. ",
		"user": "Javier Gonzalez",
		"category": "Humanidad > Maltratos",
		"ago": "3h 40min ",
		"userImg": "/image/user_male_portrait?img_id=10706",
		"categoryId": "25251907177476578395313757095",
		"assessment": 3,
		"commentsCount": 2
	},
	{
		"id": "25251905904651237309354695590",
		"title": "Anillo Urbano Cultural",
		"user": "Javier Gonzalez",
		"category": "Ocio > Cultura",
		"ago": "3h 41min ",
		"userImg": "/image/user_male_portrait?img_id=10706",
		"categoryId": "25251905904651237309354695589",
		"assessment": 1,
		"commentsCount": 3
	}]
}
```

6.- Diseños técnicos

6.1- Estructura del proyecto

A la hora de empezar a estructurar el proyecto buscaba una herramienta que me ayudará a la hora de gestionarlo, ya que en proyectos de esta magnitud es necesario dividirlos correctamente en capas/proyectos para asegurarnos de diseñar una aplicación mantenible y de fácil gestión.

La elección fue "Maven", cuya filosofía general es la estandarización de las construcciones generadas para seguir el principio de "Convención sobre Configuración", a fin de utilizar modelos existentes en la producción de software. Asume un comportamiento por defecto que permite empezar a trabajar sin necesidad de configuración.

Con "Maven" conseguimos:

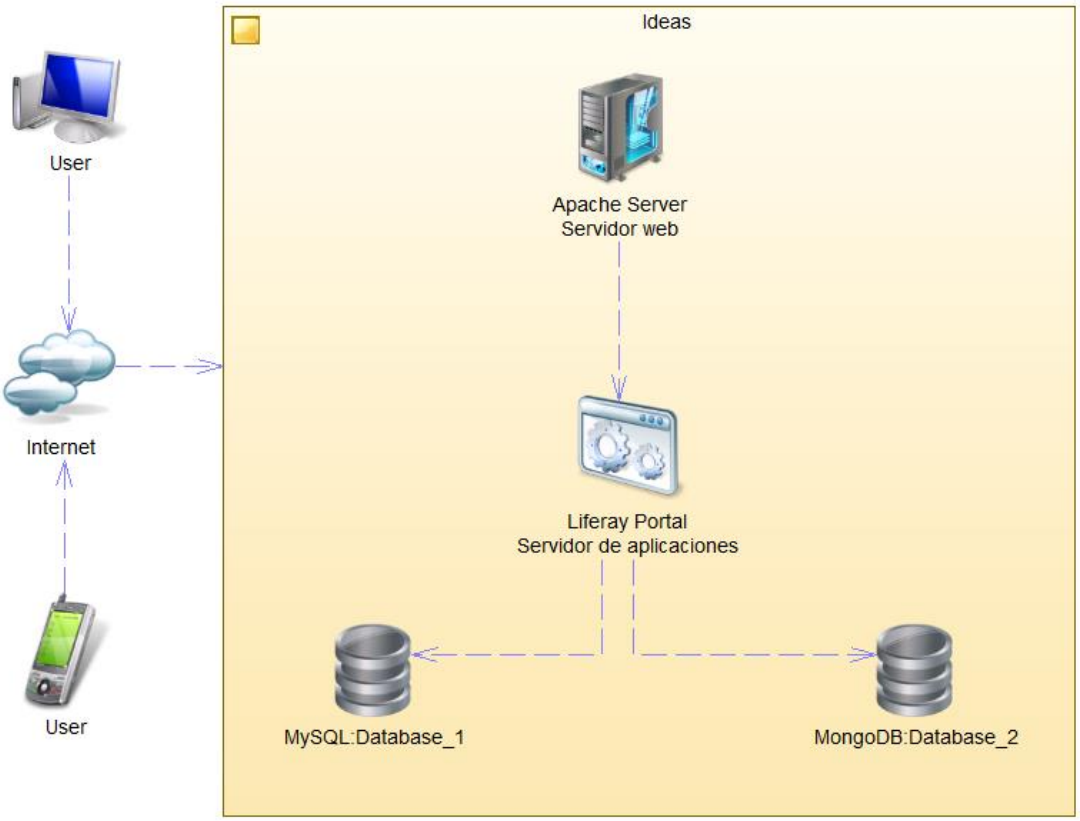
- Un modelo de objeto del proyecto.
- Un sistema de gestión de dependencias.
- El ciclo de vida del proyecto.
- La lógica para ejecutar nuevas tareas en determinadas fases del ciclo de vida.

La división del proyecto se basa en las siguientes capas, que explicaré posteriormente:

- Configuración del modelo
- Repositorios
- Servicios
- Vistas (SmartGWT)
- Controladores
- Portlets
- Componentes de Liferay
- Utilidades

6.2- Visión general del sistema

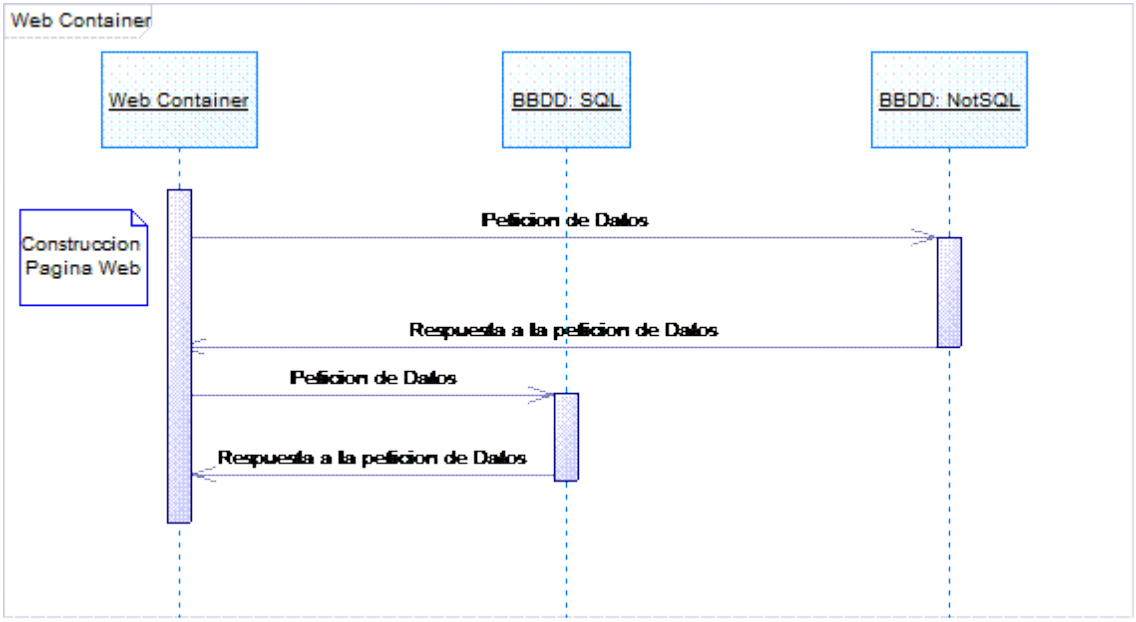
El siguiente diagrama muestra a alto nivel (nivel hardware) los componentes que se utilizan para dar servicio a la aplicación:



- Apache server: Un servidor web o servidor HTTP es un programa que procesa cualquier aplicación del lado del servidor realizando conexiones bidireccionales y/o unidireccionales y síncronas o asíncronas con el cliente generando o cediendo una respuesta en cualquier lenguaje o Aplicación del lado del cliente. No he llegado a configurar el servidor web, ya que este componente tiene sentido si hubiera puesto en producción la aplicación (no se descarta en un futuro). Además serviría como balanceador de carga entre los "n" portales Liferay que se montaran en al infraestructura.
- Servidor de aplicaciones: En Java Platform Enterprise Edition, un contenedor web, también conocido como un contenedor de Servlets "implementa el contrato de componente web de la arquitectura Java EE". Este contrato especifica un entorno de ejecución para componentes web que incluye seguridad, concurrencia, gestión de ciclo de vida, transacción, el despliegue, y otros servicios. En concreto, la aplicación está basada en Apache Tomcat con Liferay desplegado.
- Base de datos MongoDB (no relacional) para la lógica de negocio.
- Base de datos MySQL (relacional) para gestión de usuarios.

El contenedor web es el componente encargado de la creación de la página Web, para la construcción de cada una es necesario recoger datos de

las distintas bases de datos que compone la aplicación. A continuación se muestra el diagrama de actividad del componente.

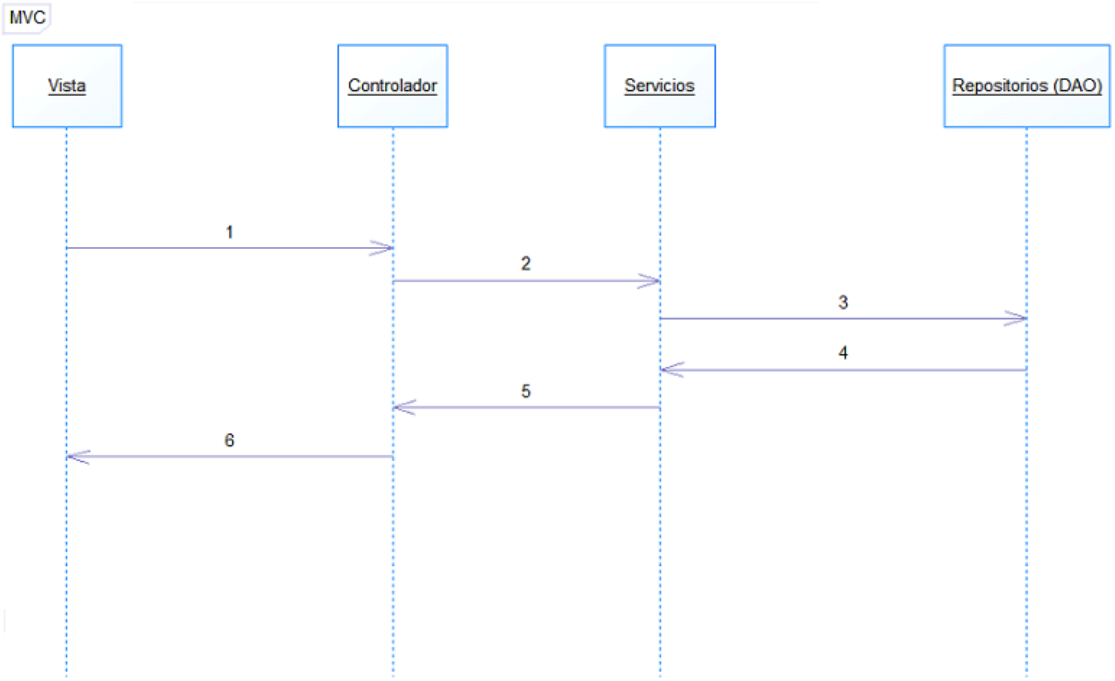


6.3- Arquitectura

La arquitectura propuesta para la aplicación se basa en el patrón de arquitectura del software "Modelo Vista-Controlador (MVC)", que define una separación de los datos de la aplicación en las siguientes capas:

- Modelo: En esta capa se encuentra los objetos de negocio y el acceso a datos "DAO".
- Negocio: Se encarga de obtener los datos del modelo y enviarlos a las vistas. Aquí se encuentran los controladores de la API RESTful y las interfaces de servicios.
- Vista: Se encarga de mostrar los datos de una forma visual a los usuarios.

En el siguiente diagrama de actividad se muestra la actividad general de las peticiones de la aplicación, más adelante se detallará individualmente las acciones posibles:



1. La vista solicita información a los controladores a través de una petición a la API RESTful.
2. El controlador correspondiente recibe la petición, y llama a los servicios.
3. Los servicios, aplican lógica de negocio y llama a los repositorios.
4. Los repositorios obtienen los datos de base de datos, y los mandan a los servicios en los objetos de negocio.
5. Los servicios reciben los datos, aplican lógica de negocio y los envían al controlador en los objetos de negocio.
6. El controlador recibe estos objetos y los transforma en "objetos de transferencia (Transfer Object)", que corresponden a los JSON de respuesta de la API RESTful. La vista obtiene los datos y "renderiza" la vista.

6.4- Spring y perfiles de la aplicación

Todos los componentes de la aplicación están gestionados a través de Spring, un framework para aplicaciones JAVA. Con él, conseguimos un contenedor de inversión de control, ya que permite la configuración de los componentes de la aplicación y la administración del ciclo de vida de los objetos java. Se lleva a cabo mediante la inyección de dependencias.

Cada proyecto cuenta con su propia configuración, en la que se han definido los componentes necesarios. Todos los ficheros de configuración siguen una nomenclatura común, tanto en el paquete JAVA (***.config**) como en la clase JAVA (***Config.java**). Con ello conseguimos tener separado la lógica de negocio de su configuración.

Toda la configuración e inyección de dependencias se hace a través de anotaciones como:

- @Configuration
- @ComponentScan
- @Import
- @ImportResource
- @Bean
- @Inject
- Etc..

Se han definido una serie de perfiles en el proyecto, con los que se puede configurar la aplicación, ya que existen diferencias en la configuración, dependiendo en que ciclo de vida se encuentra el proyecto, no tiene sentido que el proyecto tenga la misma configuración en desarrollo que en producción. Por tanto, se han definido los siguientes perfiles principales:

- CLIENT: Los componentes que van en el lado del cliente ("portlets") llevan asociado este perfil.
- DEV: Este perfil es el que se utiliza durante el desarrollo del proyecto.
- PRODUCTION: Cuando el proyecto se ponga en producción, se activará este perfil.
- TEST: Este perfil se asocia a todos los test de la aplicación.

Además existen tres perfiles secundarios relacionados con los test:

- TEST_JENKINS: Perfil que asociamos a Jenkins para que ejecute los test.
- TEST_NO_CREATE_DATA: Con este perfil no se crean datos de prueba.
- CREATE_TEST_DATA: Con este perfil se crean datos de prueba para los test.
  Estos últimos perfiles, se asocian al sistema de pruebas creado, que explicaré posteriormente.

Todos estos perfiles se utilizan a través de los ficheros de configuración de Spring, a través de anotaciones. Todos los ficheros que siguen la nomenclatura mencionada antes ("*Config.java") tienen asociado uno o varios de estos perfiles.

Ejemplo:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = MyServiceConfig.class, loader =
AnnotationConfigContextLoader.class)
@ActiveProfiles(ProfileConstant.TEST)
public abstract class AbstractMvcControllerTest {

.....

}
```

En este ejemplo, a la clase "AbstractMvcControllerTest" se le asocia perfil de "Test", todas las implementaciones que hereden de esta clase abstracta también tendrán esta configuración.

En los "Portlets", es necesario configurar los perfiles a través del inicializador del contexto de Spring, se ha definido la siguiente porción de código en la clase "AbstractSpringAppCtxtInit":

```
String env = System.getProperty("ideas.env");
if (env != null) {
	environment.setActiveProfiles(env);
} else {
	environment.setActiveProfiles(ProfileConstant.DEV);
}
```

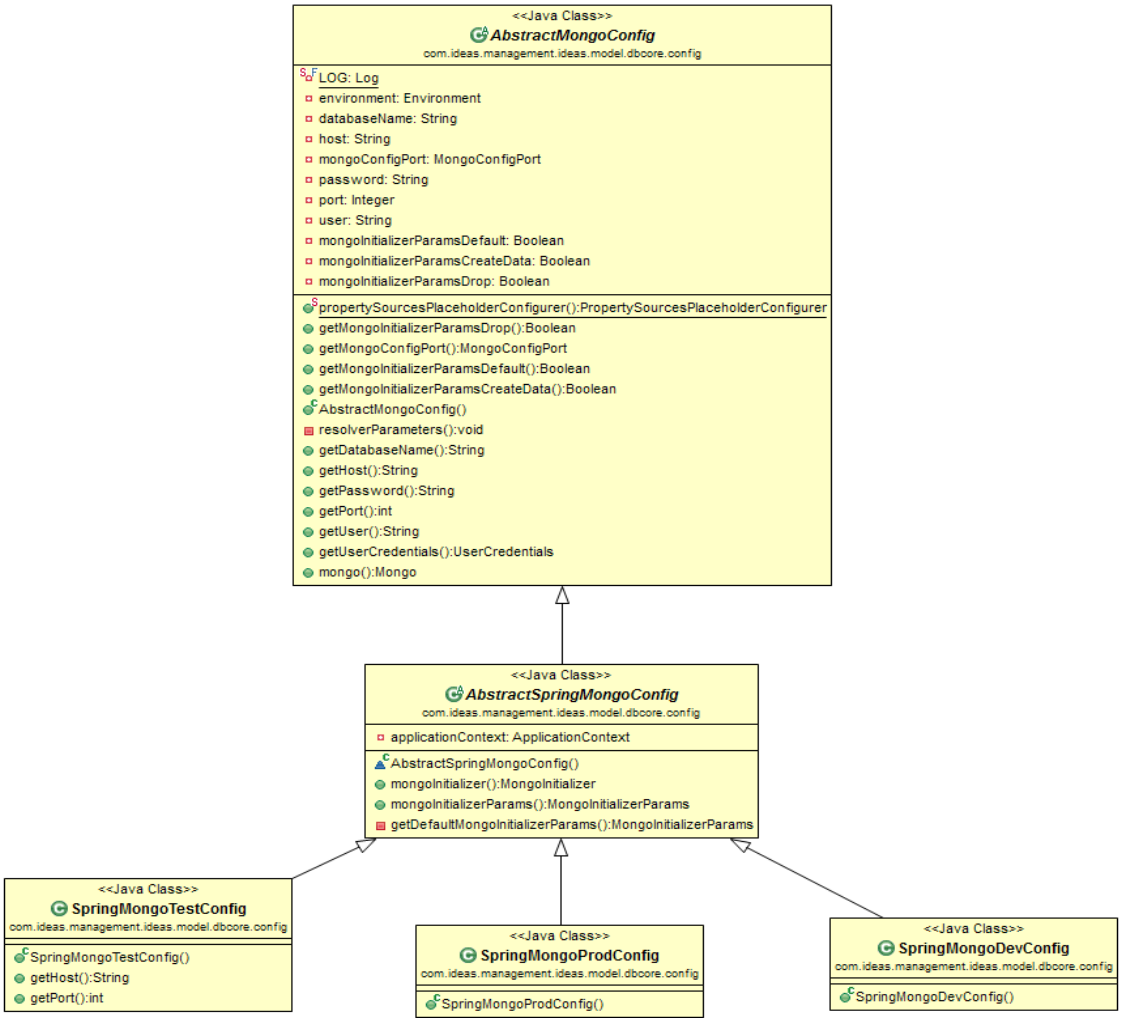
La primera acción es obtener la propiedad "ideas.env" del sistema, si está definida asociamos se la asociamos al entorno, si no, activamos el perfil de desarrollo (por defecto).

La propiedad "ideas.env" hay que definirla en el arranque del servidor cuando queramos que la aplicación se inicie con el perfil de producción. Hay que asociarle el siguiente valor:

```
-Dideas.env=production
```

Con esta definición, se ha descrito cómo utilizar los perfiles, pero no he descrito que configuración se le asocia a cada uno. Ahora vamos a ello.

Se ha definido la siguiente estructura:



Existen tres implementaciones en cuanto a la configuración de la base de datos MongoDB, a cada una se le asocia uno de los perfiles principales (TEST, DEV, PROD) de la siquiente forma:

```
@Configuration
@PropertySource("classpath:/database.test.properties")
@Profile(value = { ProfileConstant.TEST })
public class SpringMongoTestConfig extends AbstractSpringMongoConfig
{

…

}
```

Cada una de estas configuraciones utiliza un fichero ".properties" distinto, dónde se definen la configuración acerca de la base de datos mongoDB y del sistema de pruebas (lo explicaremos en el apartado de sistema de pruebas). Los campos son:

- db.mongo.host: Dirección IP del servidor MongoDB
- db.mongo.port= Puerto del servidor MongoDB

- db.mongo.databaseName= Nombre de la base de datos
- db.mongo.params.default= Activar/desactivar acciones por defecto del sistema de pruebas (test).
- db.mongo.params.create= Crear datos de prueba (test).
- db.mongo.params.drop= Borrar datos de prueba (test).
- db.mongo.user= Usuario de la base de datos MongoDB
- db.mongo.password= Contraseña de la base de datos MongoDB

Por lo tanto, al utilizar un perfil u otro, lo que conseguimos es utilizar diferentes configuraciones de la base de datos MongoDB, configurables a través de los cuatro ficheros ".properties" que existen.

6.5- Sistema de trazas "logs"

Cualquier proyecto debe contar con sistema de "trazas (logs)" con el objetivo de informar acerca de los sucesos que ocurren en el sistema, ya sea mostrándolos por consola o escribiendo en un fichero de "logs". Así se pueden revisar sucesos anómalos que ocurran en el sistema, o detectar las acciones que realizan los usuarios en el mismo.

Se han integrado dos tecnologías de trazas en el sistema, para toda la parte del servidor se utiliza Log4J y en la parte del cliente (SmartGWT) se utiliza GWT-Log.

Para la integración con Log4J se ha desarrollado el subproyecto "ideas.util.logger", dónde se añaden las librerías de Log4J, y se define el fichero de configuración "log4j.properties", dónde se indica el nivel de trazas que se quiere mostrar de cada parte del proyecto y dónde se muestran estas trazas.

Los demás proyectos añaden a este como dependencia para obtener las funcionalidades de este framework.

En concreto, cada proyecto que quiera utilizar trazas, debe definir en la cabecera de la clase el atributo:

```
/** The Constant LOG. */
private static final Log LOG = LogFactory.getLog(<NOMBRE_CLASE>.class);
```

Con esta llamada a la factoría "LogFactory", se obtiene una implementación concreta para esa clase. Si se quiere mostrar una traza, simplemente utilizamos la llamada:

```
LOG.info("contenido de la traza");
```

Además de nivel "info" existen otros niveles de traza, como "debug", "error" etc.

Para la parte del cliente, he integrado "gwt-log" añadiendo en el fichero de configuración de "smartgwt (core.gwt.xml)" el siguiente contenido:

```
<inherits name="com.allen_sauer.gwt.log.gwt-log-DEBUG" />
<extend-property name="log_level" values="DEBUG" />
<!-- Loggers Enabled by default -->
<set-property name="log_ConsoleLogger" value="ENABLED" />
<set-property name="log_DivLogger" value="DISABLED" />
<set-property name="log_WindowLogger" value="DISABLED" />
<set-property name="log_FirebugLogger" value="ENABLED" />
<set-property name="log_GWTLogger" value="DISABLED" />
<set-property name="log_SystemLogger" value="DISABLED" />
```

Con esta configuración, añadimos a cada vista las librerías de trazas de "gwt-log". Estas trazas son visibles en las consolas de los navegadores web, como Firefox o Chrome.

Para añadir las trazas en cada vista "Smartgwt", basta con importar la clase "com.allen_sauer.gwt.log.client.Log" y llamar a los métodos "info", "debug" etc. que queramos.

6.6- Capas de la aplicación

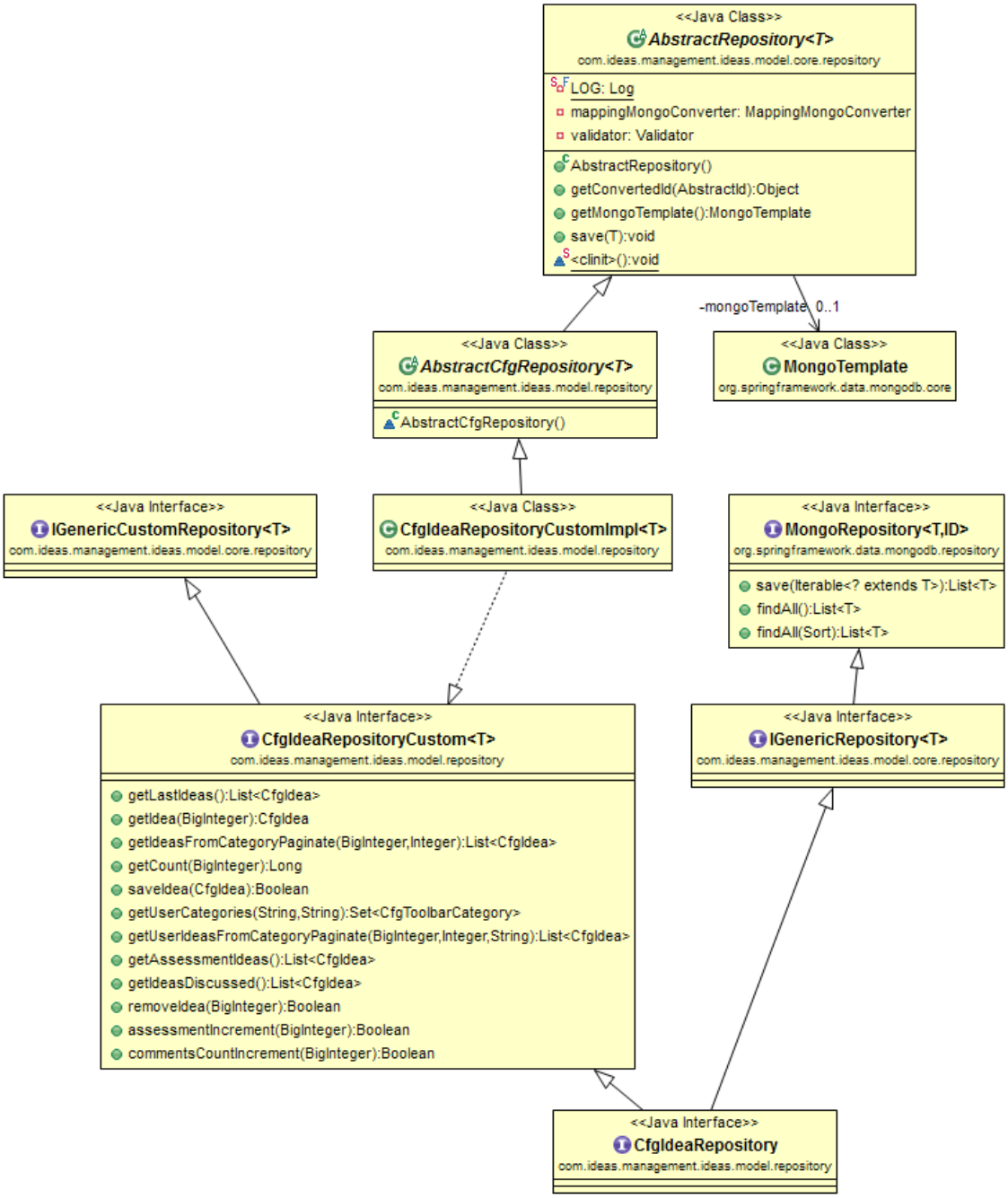
Como he mencionado anteriormente, la aplicación se ha dividido en una serie de capas arquitectónicas, con el objetivo de diseñar una aplicación siguiendo el modelo vista-controlador (MVC). A continuación se va a hacer un análisis técnico de los diseños realizados en cada capa. Son conceptos fundamentales que hay que entender para que posteriormente podamos entender las acciones del cliente propuesto.

6.6.1- Capa de repositorios

Esta capa tiene como objetivo acceder a la base de datos MongoDB y obtener los datos de ideas, comentarios y categorías, que recordemos, son los que almacenamos en esta base de datos y componen los objetos de negocio. Los métodos de esta capa tienen que tener la menos lógica de negocio posible, ya que es la capa de servicios en la que se debe situar este cometido.

Como se describió en el apartado de modelo de datos, he utilizado el framework Spring-data-mongo, por lo que se han tenido que realizar las acciones de configuración e integración oportunas. El diseño técnico se divide en tres repositorios, un para ideas, otro para comentarios y un tercero para las categorías.

El diagrama de clases asociado a la colección ideas es la siguiente:



La estructura general se basa en "tipados de java" (T extends IDocument), recordemos que todos los objetos de negocio implementan la interfaz "IDocument", con esto conseguimos reutilizar la estructura en todos los repositorios que queramos implementar. Si en un futuro se añadieran más colecciones a la base de datos, solo habría que implementar un nuevo repositorio siguiendo la estructura propuesta.

Vamos a explicar el objetivo de las principales clases e interfaces del diseño:

- AbstractRepository: Esta es la clase común principal, en ella "inyectamos" el "MongoTemplate", una clase de Spring que nos aporta una serie de métodos para interactuar con MongoDB a través de las consultas oportunas. Además contiene métodos genéricos para todos los repositorios.
- IGenericRepository: Este interfaz hereda los métodos genéricos de un "CRUD", a través de la interfaz "MongoRepository" de Spring.
- CfgIdeaRepositoryCustom: En esta interfaz se definen los métodos que se van a utilizar para obtener la información de la colección ideas. Su implementación es "CfgIdeaRepositoryCustomImpl".
- CfgIdeaRepository: Esta será la interfaz que inyectaremos cada vez que queramos utilizar los métodos asociados a la colección ideas. En concreto, se inyectará en la capa de servicios. Spring se encargará de buscar la implementación, que en este caso será "CfgIdeaRepositoryCustomImpl".

En la clase "CfgIdeaRepositoryCustomImpl" se centran todos los métodos de acceso a los datos de la colección de ideas. Esto se realiza con "querys", veamos un ejemplo:

```
@Override
public List<CfgIdea> getLastIdeas() {

Criteria criteria = new Criteria("coreActiveElement.active").is(true);
Query query = new Query(criteria);


Order order = new Order(Direction.DESC,"createDate.zeroGmtMatchDate");
Sort sort = new Sort(order);
QueryUtils.applySorting(query, sort);
query.limit(8);

return getMongoTemplate().find(query, CfgIdea.class);
}
```

Este método lo que hace es obtener las últimas 8 ideas (activas) de la colección "CfgIdea (ideas)", las ordena en orden descendente, es decir, de la más nueva a la más vieja. Para ello se utilizan las especificaciones de la API de "Spring-data-mongo", en concreto utilizamos los objectos Criteria, Query, Order y Short. Con ellos, generamos la "query" que entiende el motor de MongoDB para obtener los resultados deseados.

En el objeto "criteria" añadimos la condición de que el campo "active" que se encuentra dentro del campo "coreActiveElement" , tenga el valor "**true**". Estos campos se corresponden con la estructura de una idea (BSON), que es la que se almacena en MongoDB. Con el objeto "Order", indicamos que ordene en orden descendente (*DESC*), a partir del campo "zeroGmtMatchDate", que se encuentra dentro del campo "createDate".

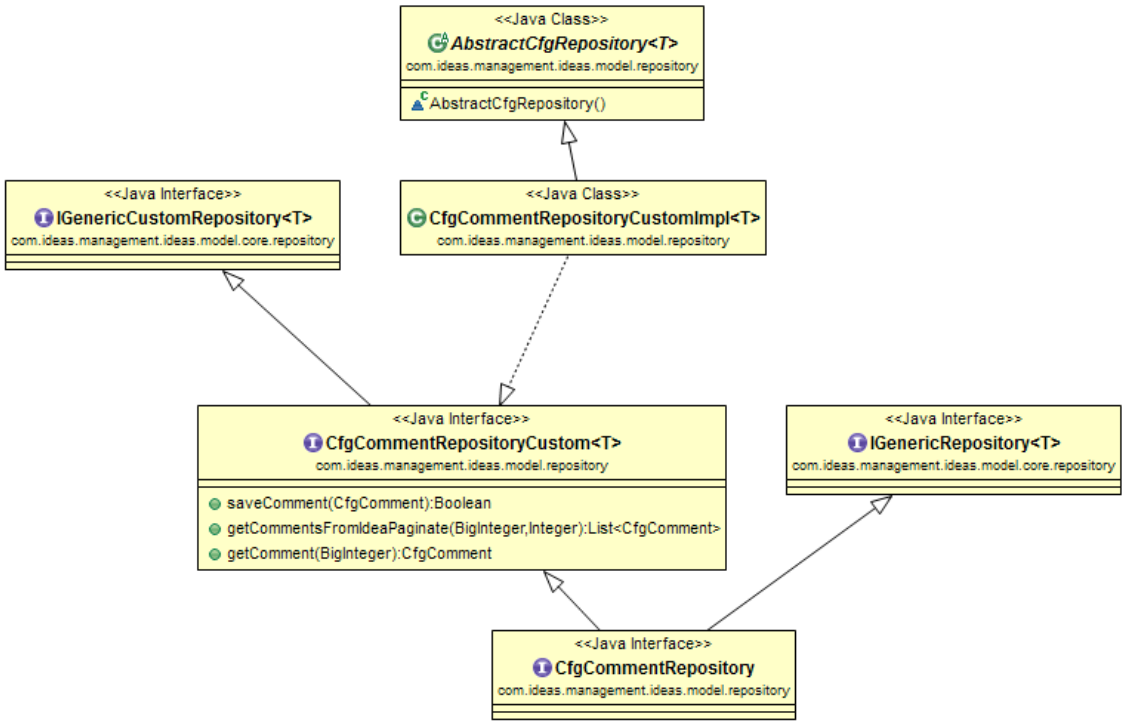
Este valor es una fecha (Date en java), pero MongoDB también trabaja con este tipo de datos y sabe ordenarlos (DateTime). Con esto nos ahorramos hacer la ordenación por JAVA, y sacamos partido a las muchas posibilidades que ofrece el motor de MongoDB.

Por último llamamos al método "find()", que busca en la colección de ideas (CfgIdea), las entradas que cumplen con la consulta creada. Esto nos devuelve una colección de ideas, que devuelve el método.

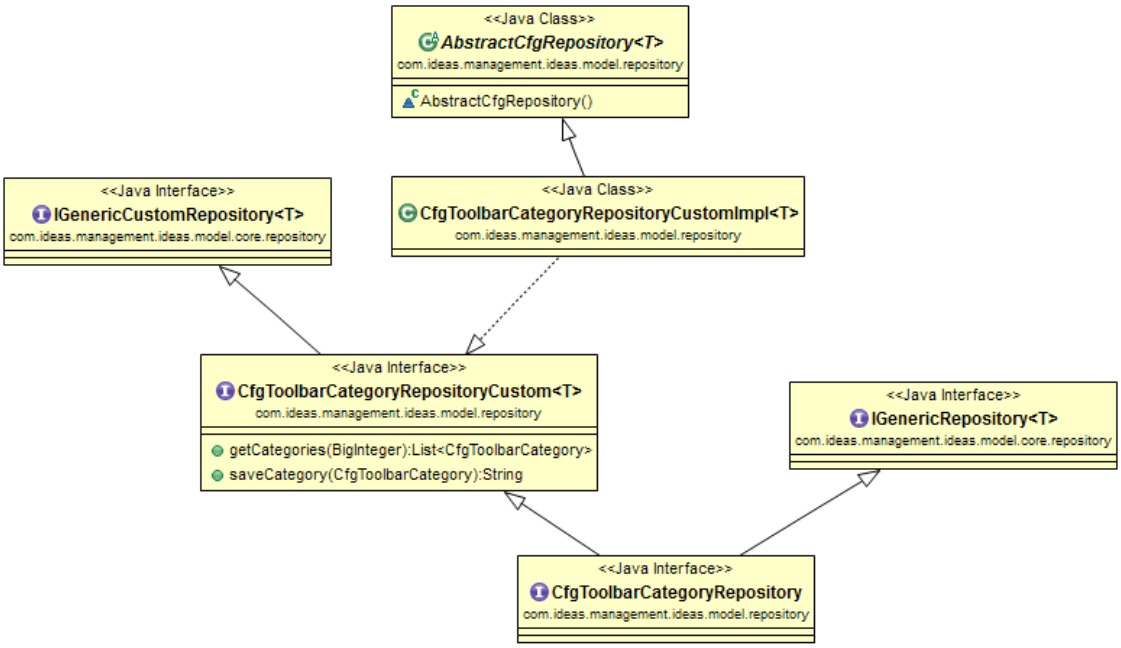
Este es solo un ejemplo de uno de los métodos implementados, hay muchos más tal y como se puede ver en la interfaz del diagrama de clases. No es objetivo de esta memoria explicarlos uno a uno, si se quiere ver las demás implementaciones, recurrir al código fuente de la aplicación.

Las otras dos colecciones existentes en la aplicación, Comentarios y Categorías, siguen el mismo patrón de diseño propuesto.

El diagrama de clases asociado al repositorio de comentarios es:

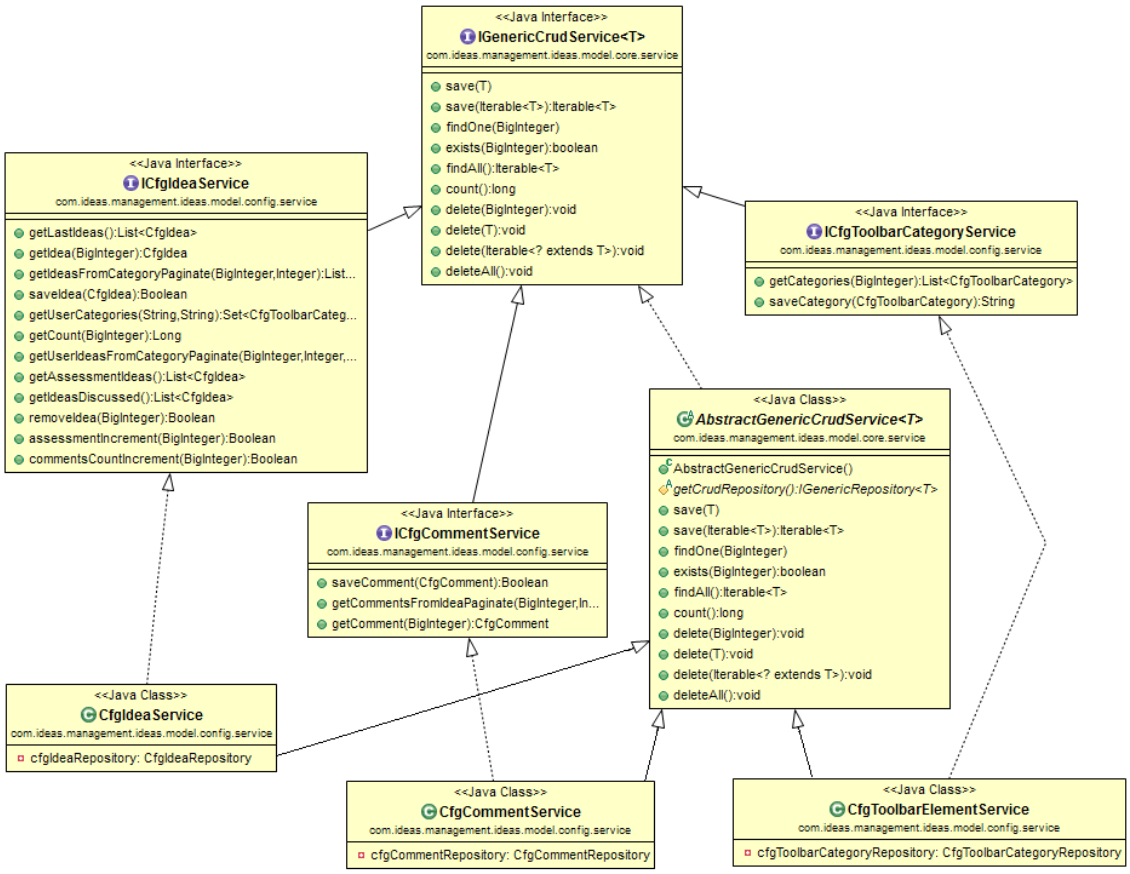


Y el de categorías es:



6.6.2- Capa de Servicios

En esta capa se sitúan las operaciones de negocio, su objetivo es utilizar llamadas a los repositorios que hemos descrito y con los datos obtenidos realizar las acciones oportunas si es que es necesario. El diagrama de clases es el siguiente:



Como se puede observar en el diagrama implementado, existen tres servicios, uno para cada colección. Cada una de estas utiliza su correspondiente repositorio, que definimos en el punto anterior.

Existen exactamente los mismos métodos que en los repositorios, ya que los servicios se basan en ellos para obtener los datos, y después aplican la lógica de negocio correspondiente.

Además está la clase "AbstractGenericCrudService", en ella se implementan los métodos genéricos de un "CRUD", aunque la implementación se limita a llamar a la implementación que se definió en los repositorios, es decir, realiza únicamente un "puenteo".

Estos servicios se inyectan en los controladores cuyo diseño técnico voy a explicar a continuación.

6.6.3- Capa de controladores

En esta parte de la aplicación, se han definido en una serie de controladores todos los métodos que definimos en la API RESTful de la aplicación.

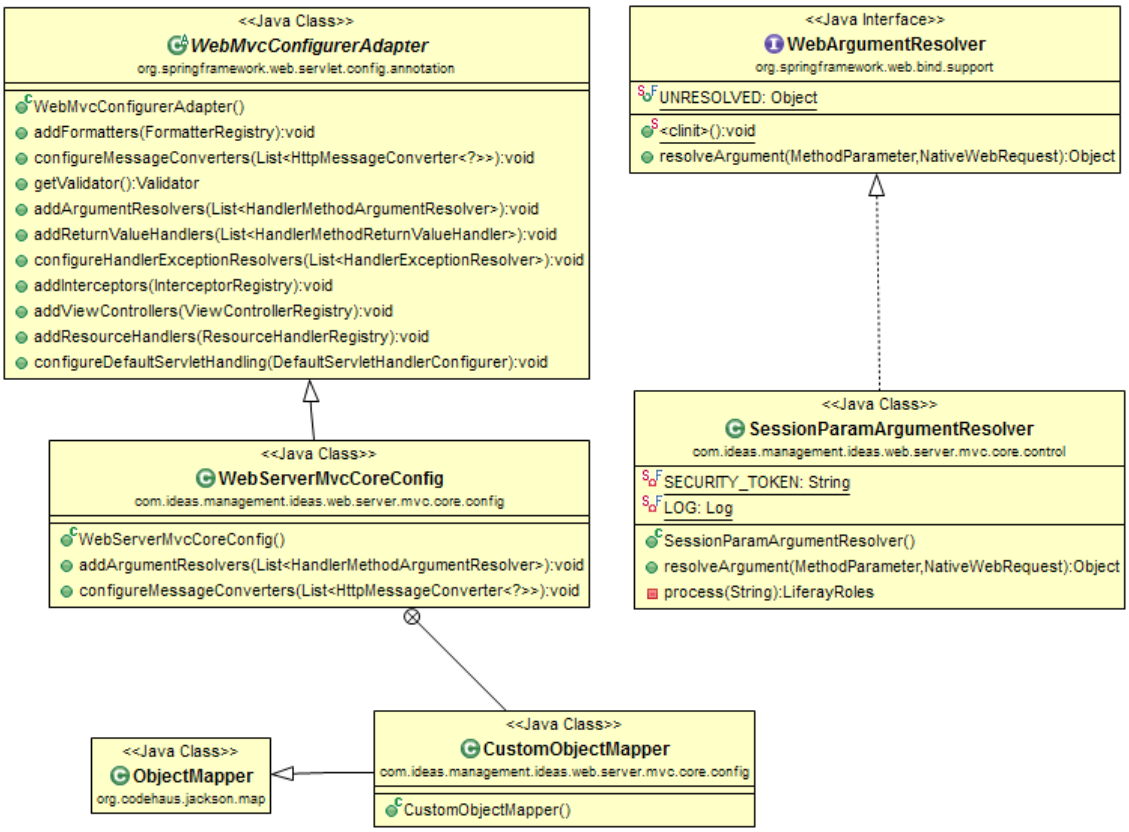
Como las demás partes del proyecto, esta también se gestiona a través de Spring. En concreto, esta parte utiliza "Spring-MVC" y "Spring-REST".

Spring-MVC nos proporciona una capa de abstracción para diseñar los controladores de nuestro modelo, que recordemos sigue el patrón de diseño modelo vista-controlador. Los controladores son el punto de entrada de las peticiones que cualquier cliente realice, ya sea desde el cliente propuesto o desde otro cliente.

Spring-REST nos proporciona una serie de anotaciones muy completa, con la que poder definir la API RESTful y todos los controladores que se encargan de gestionar cada petición a esta API.

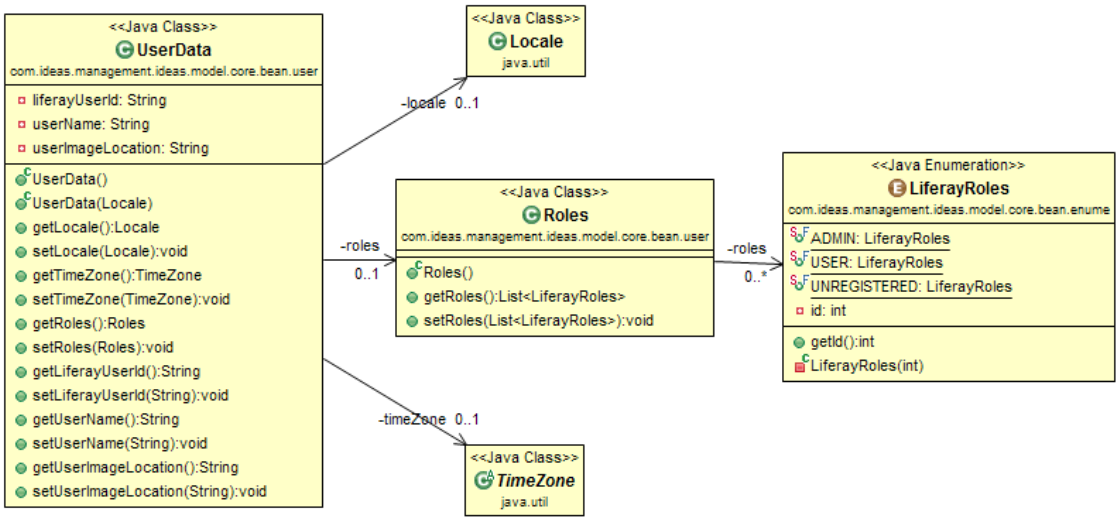
Por lo tanto, el objetivo de los controladores es recibir las peticiones de los clientes, procesarlas según corresponda, llamando a la capa de servicios si es necesario y enviando al cliente el resultado de la petición.

Para configurar los controladores se ha realizado el siguiente diseño técnico:



Estas clases componen el "core" de los controladores, definiendo en ellas todo la configuración necesaria. A continuación voy a explicar el objetivo de cada clase:

SessionParamArgumentResolver: Este componente se utiliza para resolver los parámetros de sesión del usuario que utilice la aplicación. Esto se realiza gracias a un interceptor de Spring, que procesa todas las peticiones que llegan a cualquiera de los controladores. Por lo tanto, cada controlador recibe además del JSON correspondiente (convertido a un objeto JAVA) un objeto llamado "UserData", que contiene los atributos que necesitamos en la aplicación:



Como podemos observar en el diagrama de clases de "UserData", los campos que obtenemos de la sesión del usuario son los siguientes:

- liferayUserId: Identificador en Liferay del usuario que realiza la petición.
- userName: Nombre del usuario que realiza la petición.
- userImageLocation: Localización de la imagen de perfil del usuario.
- Idioma del usuario.
- Zona horaria del usuario.
- Rol que posee el usuario.

Por supuesto, si un usuario no está registrado en el sistema, se dan valores por defecto a estos campos.

Los tres primeros atributos del usuario se resuelven de la siquiente forma:

```
UserData result;
Authentication auth;
…
…
User user = UserLocalServiceUtil.getUser(new Long(auth.getPrincipal().toString()));

result.setUserName(user.getFullName());
result.setUserImageLocation("/image/user_male_portrait?img_id=" + user.getPortraitId());
result.setLiferayUserId(auth.getPrincipal().toString());
```

El objeto "auth (Authentication)", contiene información acerca del usuario que realiza la petición (en el apartado de Portlet-Securty explicaré en detalle cómo se resuelve este objeto), a través de él, se llama al servicio de Liferay "UserLocalServiceUtil", para obtener el usuario. Con este objeto, mapeamos la información deseada a "UserData".

El idioma y la zona horaria del usuario se obtienen a través de las cookies de sesión del usuario, que también llegan en la petición, estas cookies se llaman:

- GUEST_LANGUAGE_ID
- USER_TIMEZONE

Del mismo modo, los valores que posean, se añaden al objeto "UserData".

Los roles se obtiene de sesión a través del objeto "Authentication", se procesan y se convierten a uno de los tres tipos que existen en el sistema. Estos tipos están definidos en un enumerado.

WebServerMvcCoreConfig: En esta clase se activa la configuración de los controladores, para ello se anota la clase con "@EnableWebMvc", una anotación que sirve para importar la configuración de Spring-MVC. Esto hace tener los valores por defecto, pero es necesario realizar varias acciones más:

- Definimos un formato especial para las fechas de la siguiente forma:

```
public CustomObjectMapper() {
	super();
	configure(Feature.WRITE_DATES_AS_TIMESTAMPS, false);
	setDateFormat(new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSSZ"));
}
```

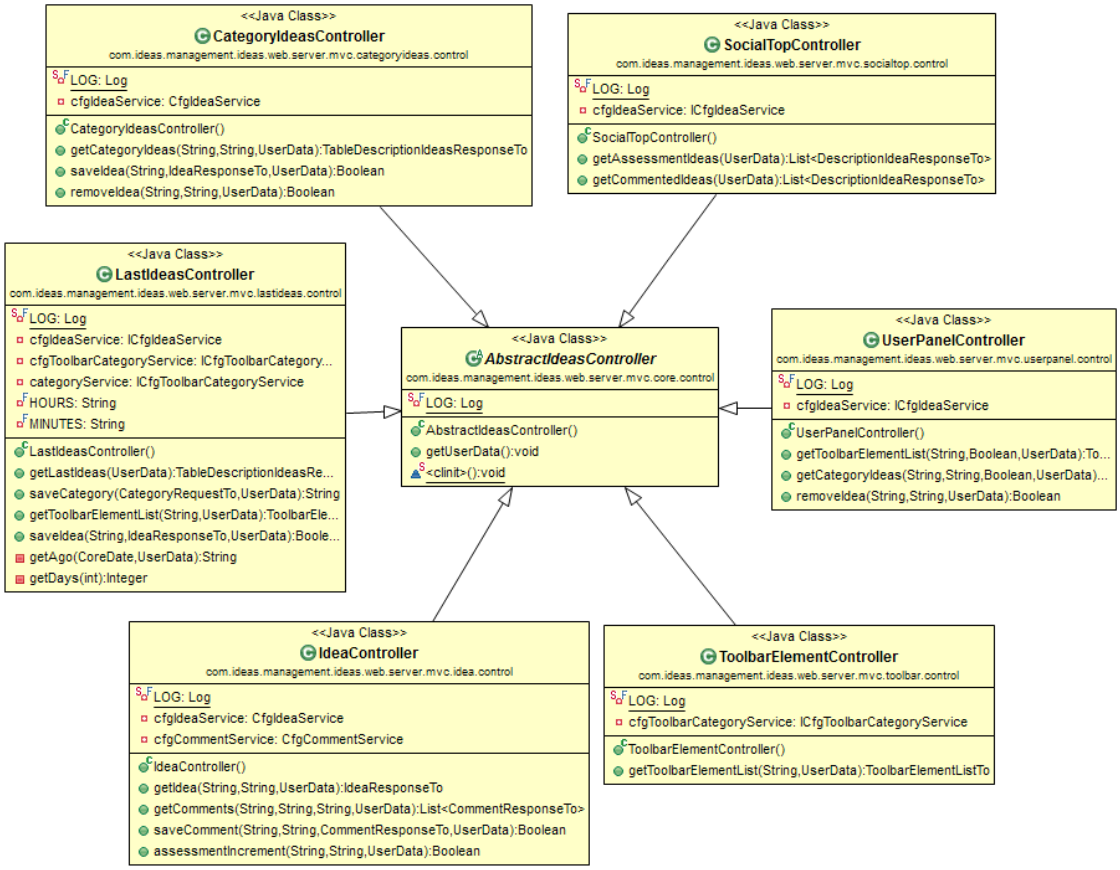
- Indicamos que las peticiones contienen cuerpos de tipo "application/json", la codificación en UTF-8 y el formato de fecha definido en la clase anterior:

```
public void configureMessageConverters(List<HttpMessageConverter<?>> converters)
{

	MappingJacksonHttpMessageConverter mappingJacksonHttpMessageConverter =
new MappingJacksonHttpMessageConverter();
	List<MediaType> supportedMediaTypes = new ArrayList<MediaType>();

	supportedMediaTypes.add(new MediaType("application", "json", Charset
				.forName(EncodingUtil.UTF_8)));
	mappingJacksonHttpMessageConverter
				.setObjectMapper(new CustomObjectMapper());
	mappingJacksonHttpMessageConverter
				.setSupportedMediaTypes(supportedMediaTypes);
	converters.add(mappingJacksonHttpMessageConverter);
}
```

- Añadimos el objeto "SessionParamArgumentResolver" a los argumentos que hay que resolver:

```
public void addArgumentResolvers(List<HandlerMethodArgumentResolver>
argumentResolvers) {
	argumentResolvers.add(new ServletWebArgumentResolverAdapter(
	new SessionParamArgumentResolver()));
}
```

Esta es la configuración que tiene cada controlador de la aplicación, a continuación se muestra el diagrama de controladores desarrollados:



Como se puede observar en el diagrama de clases, se ha dividido en seis controladores los métodos que implementan la definición que hicimos de la API RESTful. Esta división tiene el objetivo de que cada uno de los seis portlets que componen el cliente propuesto posea uno de los controladores, el que necesite utilizar (más adelante lo detallaremos).

Los controladores están definidos de la siguiente forma, vamos a ver un ejemplo para entender cómo se configuran:

```
@Controller
@RequestMapping("/api")
public class IdeaController extends AbstractIdeasController {

…
```

A través de la anotación "@Controller" indicamos a Spring que todos los métodos de esta clase que estén correctamente configurados va a ser los puntos de entrada de la API RESTful.
Con la anotación "@RequestMapping" indicamos la base URI de nuestra API. A partir de esta base URI se concatenarán las demás rutas para acceder a los recursos.

Ahora veamos un ejemplo de definición de un recurso:

```
@RequestMapping(value = "/categories/{categoryId}/ideas/{ideaId}", method =
RequestMethod.GET)
@ResponseBody
public IdeaResponseTo getIdea(@PathVariable String categoryId, @PathVariable
String ideaId,final UserData userData) {

…

}
```

Este ejemplo corresponde a las acciones que se toman cuando se solicita una idea concreta a la API RESTful a través del verbo HTTP "GET" y la URI correspondiente. Vamos a analizar cada uno de los componentes mostrados.

La primera anotación que contiene el método es "@RequestMapping", en el parámetro "value" indicamos la URI que tiene que "mapear". Esta se corresponde con la definición que se hizo de la API RESTful anteriormente. Los parámetros de la URI "{categoryId}/" y "{ideaId}" se encuentran entre corchetes para indicar a Spring que estos valores serán variables que irán en la ruta de la petición. Estas serán transformadas a los dos parámetros que tiene el método del ejemplo ("String categoryId y String ideaId") mediante la anotación "@PathVariable".

El segundo valor es "method", con este campo indicamos ante que verbo HTTP tiene que responder este método.

También anotamos el método con "@ResponseBody", con esto indicamos que esto va a ser el cuerpo de la respuesta que se le va a devolver al cliente, en concreto, el objeto de transferencia que se devuelve es "IdeaResponseTo", transformado en su correspondiente JSON (a través de la configuración que indicamos antes), que definimos en la API RESTful.

La URI completa, que estaríamos "mapeando" de esta forma sería:

```
/api/categories/{categoryId}/ideas/{ideaId}
```

Las peticiones HTTP GET, que hicieran referencia a esta URI, sería tratadas por el método descrito.

Todos los demás controladores está configurados del mismo modo, y cada método que se encuentra en ellos tiene sus respectivas configuraciones de URI y verbo HTTP al que tienen que responder. No es objetivo de esta memoria hacer una descripción de cada uno de estos elementos, para ver las implementaciones concretas utilizar el código fuente.

6.6.3- Capa de Vistas

Para la parte del cliente (visual), buscaba un framework escrito en JAVA que me ayudara a desarrollar todos los componentes que componen la maqueta. SmartGWT provee un kit de herramientas para desarrollar en JAVA (posteriormente será compilado a javascript), con multitud de componentes listos para añadir a la interfaz de usuario y asignarles un estilo determinado (CSS).

Para las llamadas a la API RESTful definida en la parte del servidor, he utilizado la librería RestyGWT, que ofrece a través de anotaciones JAVA una forma sencilla de obtener los recursos.

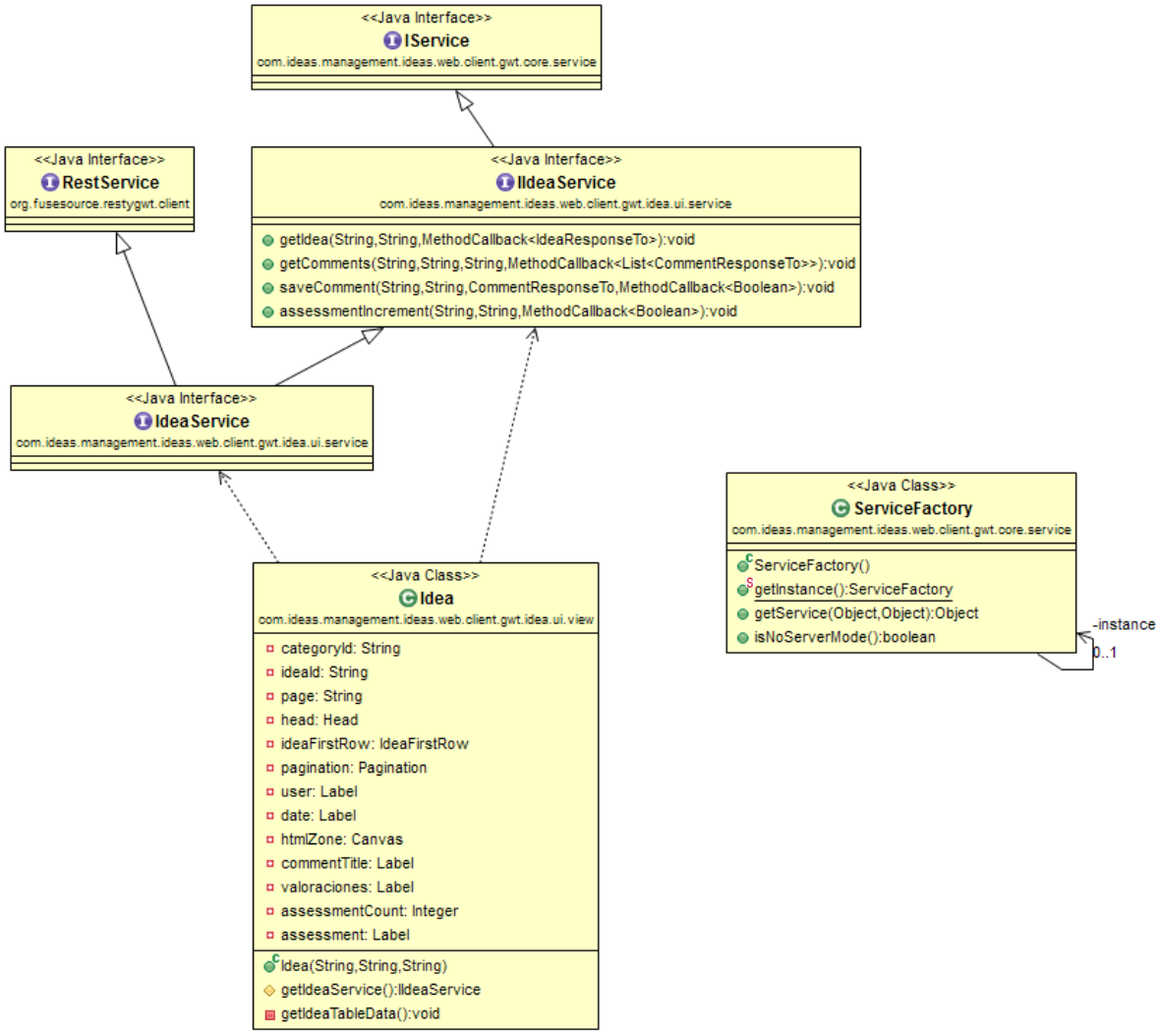
El objetivo de la vista desarrollada es utilizar parte de la API definida, para mostrar a los usuarios de la aplicación, las ideas, comentarios y categorías que la componen, de una forma sencilla y que pueden entenderla fácilmente, aunque no sean usuarios expertos.

En esta parte explicaremos en detalle la arquitectura montada para gestionar todo lo relacionado con el cliente, y en posteriores apartados nos centraremos en cada una de las vistas, analizando en conjunto cliente-servidor, todas las acciones que se realizan.

6.6.3.1- Diseño técnico para las llamadas a la API RESTful

Como he mencionado, para las llamadas a la API he utilizado la librería RestyGWT. Para utilizar en cada una de las seis vista que componen el cliente propuesto se ha diseñado la siguiente estructura, lo vamos a ver en el siguiente ejemplo que corresponde con la vista “Idea (posteriormente explicaré en detalle cada una de las vista desarrolladas)”.

El diagrama de clases es el siguiente:



En el diagrama se muestra la estructura de clases propuesta, vamos a explicar la función de las principales:

IService: Interfaz común a todos los servicios.

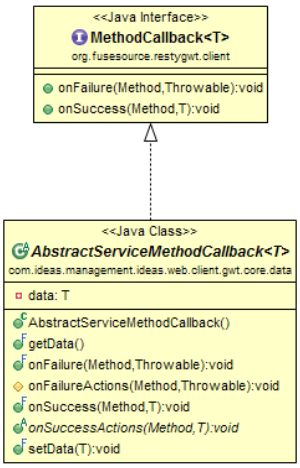
IIdeaService: En esta interfaz se definen los métodos que hará uso la vista "Idea", vamos a ver como ejemplo uno de los métodos que se definen en ella:

```
@GET
@Path("/categories/{categoryId}/ideas/{ideaId}")
void getIdea(@PathParam("categoryId") String categoryId,
             @PathParam("ideaId") String ideaId,
             MethodCallback<IdeaResponseTo> pCallback);
```

Con la anotación "@GET", definimos que este método hará una petición HTTP GET, a la URI definida con la anotación "@Path", en concreto, el recurso que se quiere obtener es

"/categories/{categoryId}/ideas/{ideaId}")", que corresponde a una idea concreta.

Los parámetros {categoryId} y {ideaId} son las variables que se añaden a la URI a través de los parámetros del método "String categoryId" y "String ideaId", que van anotados con "@PathParam".
El último parámetro del método, se corresponde con la respuesta o "callback" que recibe por parte de la parte servidora.



Se ha diseñado esta estructura de clases para que cada vista que desee utilizar un método simplemente tenga que implementar una clase que extienda de "AbstractServiceMethodCallback". Esto le obliga a implementar el método "onSuccessActions()", que es la que se ejecuta cuanto se recibe la petición de la parte servidora.

Como podemos ver, son clases "tipadas", las implementaciones usan este "tipado" para indicar el objeto de transferencia que se utiliza para la respuesta del servidor. De este modo, las librerías de RestyGWT saben a qué objeto se tiene que "mapear" el JSON de respuesta.

IdeaService: Esta interfaz implementa la anterior definida y "RestService", va configurada de la siguiente forma:

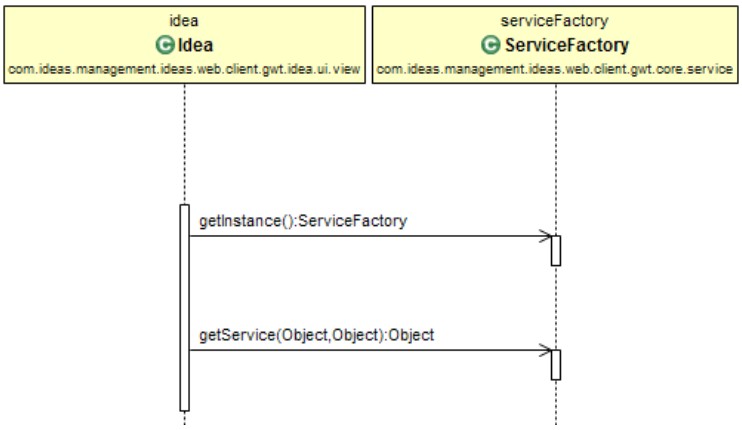
```
@Path("/rest/api")
public interface IdeaService extends RestService, IIdeaService {

}
```

Con la anotación "@Path" indicamos la base URI, que será concatenada antes de cada petición de recurso definido en "IIdeaService".

Como podemos ver, no hay una implementación que poder utilizar, esto lo resolvemos a través de la factoría "ServiceFactory" (es un singleton), que posee el siguiente método:

```
public Object getService(final Object apiService,
                         final Object dummyService) {
.....
```

Recibe dos parámetros, el primero sería la interfaz del servicio del que queramos obtener una implementación, en el ejemplo mostrado sería "IdeaService":



El segundo parámetro es un "Dummy" para realizar prueba en "hosted mode", dependiendo en que entorno nos encontremos ("hosted mode" o "Liferay") nos devuelve la implementación correcta, gracias a las librerías de RestyGWT, que se encargar de procesar las anotaciones que hemos mencionado y en tiempo de compilación las construyen.

Las demás vistas, siguen la misma estructura, crean sus propios interfaces y definen en ellos los métodos que va a utilizar.

6.6.3.2- Comunicación entre vistas (IPC)

Las vistas que se han diseñado van a ser incrustadas en "Portlets" (veremos más adelante como se hace) que posteriormente serán desplegados en Liferay. Esto significa que cada vista es independiente de las demás. Pero para que el usuario pueda interactuar con ellas, es necesario que se comuniquen entre sí, porque cada una realiza una función concreta dentro del conjunto de vistas que ve el usuario final. Es por esto que se ha establecido un modelo de comunicación entre las vistas.

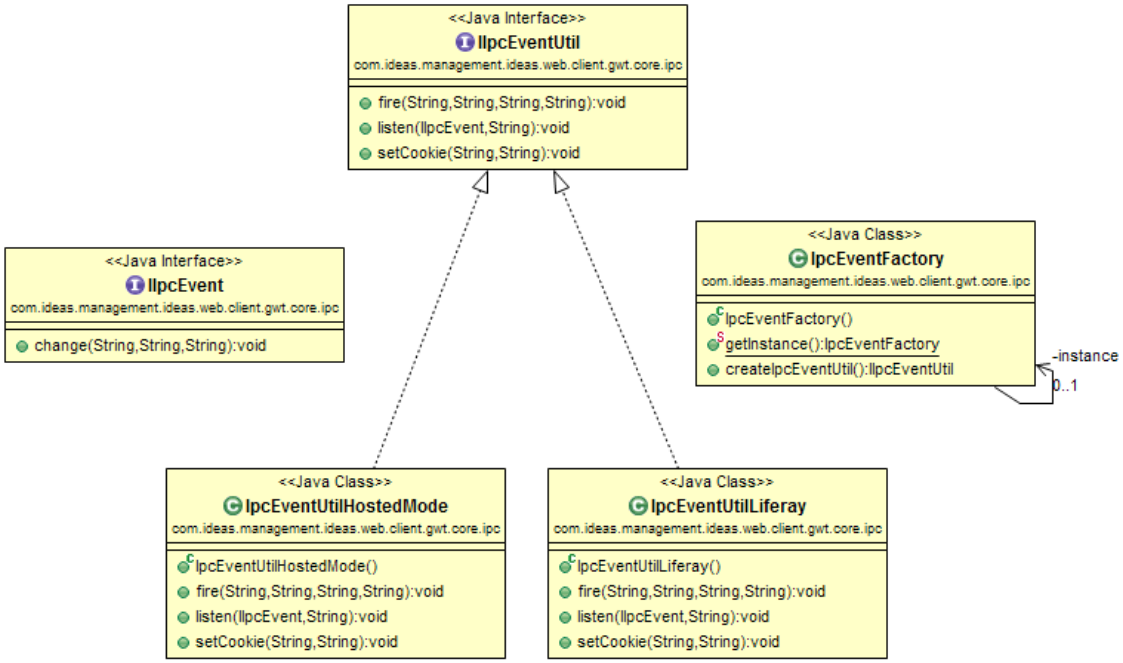
Se ha propuesto un diseño que sigue el modelo "publicador-suscriptor", en el que los publicadores lanzan eventos y estos son recibidos por las vistas que estén suscritas a ese evento concreto.

Antes de definir la arquitectura de comunicación de eventos (IPC) establecida, hay que recordar que las vistas, aunque se hayan codificado en JAVA, al ser compiladas se convierten en JAVASCRIPT, y así es como estarán presentes en cada Portlet.

Liferay posee un modelo de comunicación entre Porlets (IPC) en el lado del cliente, es decir, a través de JAVASCRIPT y sin necesidad de hacer llamadas al servidor. En concreto, posee los métodos "Liferay.fire(…)" y "Liferay.listen(…)", el primero para lanzar eventos a otros portlets y el segundo para recibir eventos, siguiendo el modelo "publicador-suscriptor" descrito. En esto se va a basar la implementación, pero hay que montar una estructura intermedia, ya que no podemos programar directamente en JAVASCRIPT.

Para solventar este problema he utilizado el módulo de JAVA "Java Native Interface (JNI)", que sirve para que un programa escrito en JAVA pueda interactuar con programas escritos en otros lenguajes, en este caso, con JAVASCRIPT.

Este es el diagrama de clases propuesto, que utilizarán las vistas para comunicarse:



IIpcEventUtil: En esta interfaz se definen los métodos que utilizan las vistas, tiene dos implementaciones posibles:

- **IpcEventHostedMode**: Cuando nos encontremos probando la aplicación en "hosted mode" utilizaremos esta implementación, que posee los métodos "dummy".

- **IpcEventUtilLiferay**: Esta es la implementación que utilizan las vistas en el entorno Liferay. En ella se utilizan los métodos de JAVASCRIPT que proporciona Lifray de la siguiente forma:

```
public native void fire(String pObjectId, String pObjectIdSec, String
pMessage, String eventName) /*-{

	$wnd.Liferay.fire(eventName, {
		objectId : pObjectId,
		objectIdSec : pObjectIdSec,
		message : pMessage,
	});
}-*/;
```

Utilizamos métodos nativos de JAVA para utilizar el módulo JNI, en ellos utilizamos el código JAVASCRIPT que define la API de Liferay. En concreto, el ejemplo que se muestra, lanza un evento, con el nombre de evento que se indique en el campo "eventName", y que contiene tres campos de información, "objectId"," objectIdSec" y "message". Con estos tres campos nos vale para poder transmitir la información necesaria entre vistas. En concreto, la información que se transmitirá a través de estos campos, son los identificadores de ideas, categoría y comentarios.

Los demás métodos, "listen" y "setCookie" siguen el mismo diseño.

IIpceEvent: Esta interfaz define el método "change", que es el que se utiliza en el método "listen". Para realizar acciones cuando se reciba un evento en alguna de las vistas, hay que crear una implementación de esta interfaz, realizando las acciones oportunas y pasándosela al método "listen" junto con el nombre del evento al que se quiere suscribir. Cuando este evento se produzca, se ejecutará el método "change" de la implementación. Veremos un ejemplo en el apartado de "Portlets", que es dónde se definen estas implementaciones.

IpcEventFactory: Esta es la factoría (singleton) que utilizan las vistas para obtener la implementación correcta de IIpcEventUtil, dependiendo el entorno en el que nos encontremos. Las vistas pedirán la implementación a través del método "getInstance" y utilizarán los métodos que ofrece según corresponda.

6.6.3.3- Maquetación e imágenes en las vistas

Se han creado dos subproyectos que son comunes a todas las vistas, uno para centralizar todas las imágenes que se necesiten añadir a las vistas y otro para el "tema de SmartGWT". Los dos proyectos se añaden como dependencias y se descomprimen en los proyectos deseados a través de un "plugin" de Maven.

- **Tema de SmartGwt**: En este proyecto contiene la maquetación (imágenes y CSS) que tiene por defecto SmartGwt para todos sus

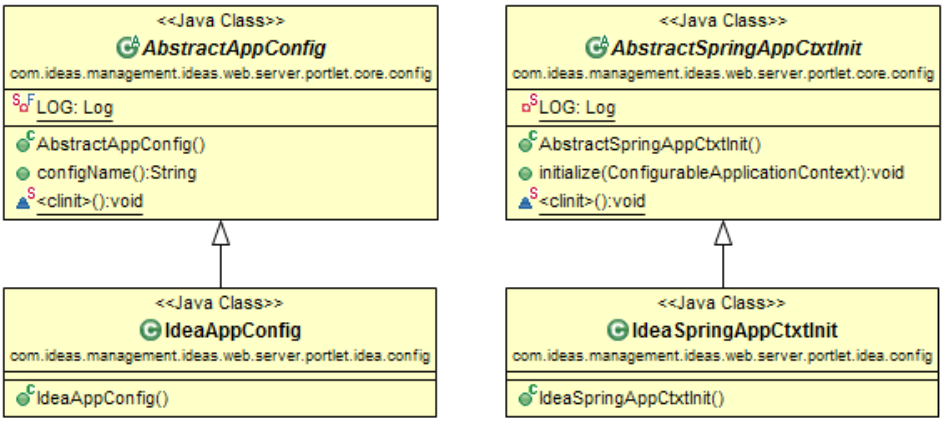
componentes visuales. Además, contiene ficheros en los que he añadido los CSS para las vistas desarrolladas. También ofrece la posibilidad de sobrescribir algunos de los estilos que vienen por defecto.

- **Imágenes**: En este proyecto se han incluido las imágenes que queremos utilizar en las vistas y queremos que el servidor de aplicaciones añada como un recurso disponible y accesible para quien lo solicite a través de su ruta.

6.6.4.- Portlets

6.6.4.1-Configuración

Los Portlets, al ser componentes web, deben ser configurados correctamente para que el servidor de aplicaciones sepa como desplegarlos en el entorno. Para ello, cada uno de los seis Portlets desarrollados, cuenta con una serie de ficheros de configuración, veamos el siguiente diagrama de clases, que corresponde a la configuración del Portlet "Idea":



Las clases abstractas son comunes a todos los Portlets, y contienen los métodos necesarios para arrancar el contexto de Spring, con las opciones necesarias. Además, en ellos se importa la configuración definida en los controladores, para poder desplegarlos en un "servlet".

Estas configuraciones, se indican en el fichero "web.xml" de cada Portlet, que es el que utiliza el contenedor de aplicaciones para desplegarlas correctamente en el contexto y estar accesibles. En este fichero definimos estas acciones:

- Indicamos al contenedor de aplicaciones que clases tiene que utilizar para levantar el contexto y dónde se encuentran las mismas.
- Añadimos un "servlet", en concreto será el definido en el "schema" de Spring (mvc-dispatcher-servlet.xml).

- Añadimos la configuración oportuna al "servlet", este será el encargado de "mappear" los métodos definidos en los controladores en sus respectivas URIs.
- Además, contiene configuración acerca de "Spring-Security", que veremos más adelante.

6.6.4.2.- Integración con SmartGWT

Como definimos anteriormente, SmartGwt produce código JavaScript para generar las vistas que se han definido. Los Portlets de Liferay, poseen con una configuración inicial en la que su contenido se muestra a través de "Java Server Pages (JSP)". En estos JSP se puede añadir código "HTML", "JAVA", "JavaScript" y "CSS". Todo el contenido de estos ficheros, se procesan en la parte servidora, bajo petición por parte del usuario, y se convierte en un HTML, que es lo que ve el usuario final.

En concreto, los Portlets definidos tienen un JSP llamado "view.jsp". Este se procesa cada vez que un usuario accede a la aplicación en Liferay. En él se realiza el punto de unión entre Liferay y SmartGWT.

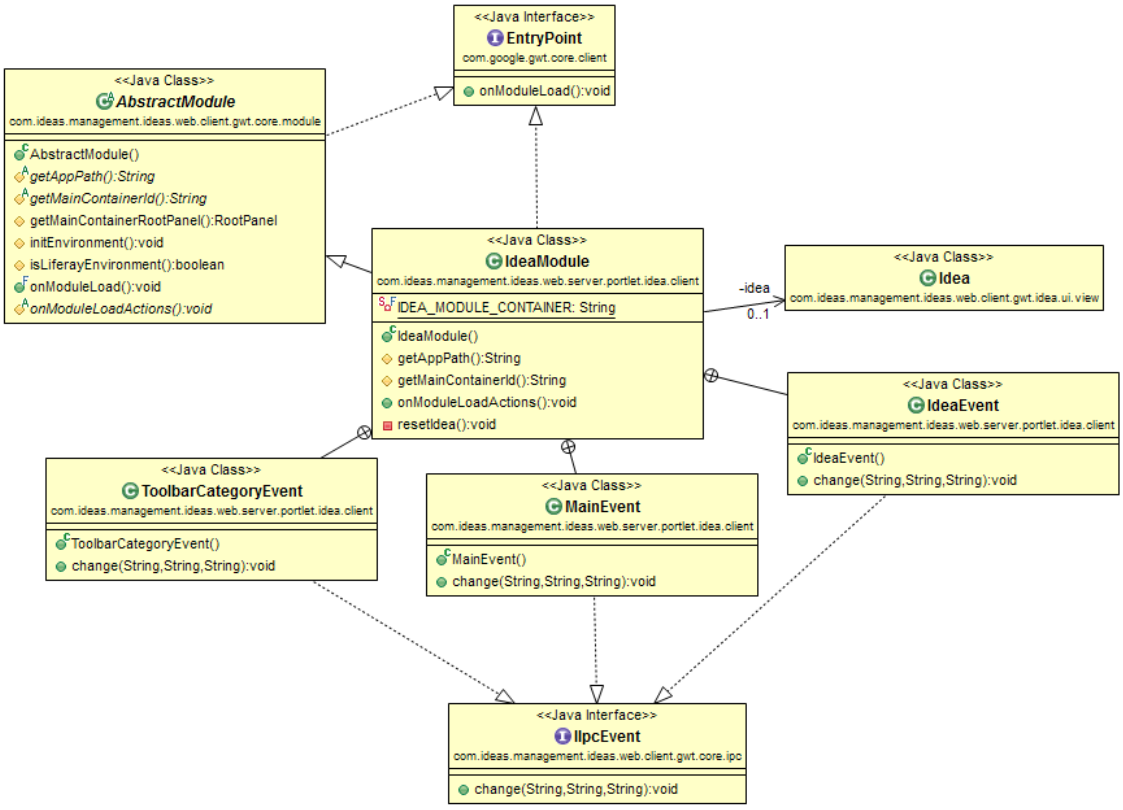
Vamos a ver un ejemplo de este fichero, en concreto el que posee el Portlet "Idea":

```
<%
        String gwtSource=(String) request.getAttribute("urlGWT");
        String urlSmartGWT= request.getContextPath()+ "/IdeaModule/sc/";
%>
<script type="text/javascript">
      var isomorphicDir='<%=urlSmartGWT %>' ;
</script>
<script type="text/javascript" language="javascript"
      src="<%=request.getContextPath()%>/IdeaModule/IdeaModule.nocache.js"><
/script>
<table align="center" width="100%">
      <tr>
            <td id="ideaModuleContainer"></td>
      </tr>
</table>
```

A partir de la "request" del usuario, obtenemos los atributos deseados, que corresponde a las rutas dónde se encuentran los ficheros JavaScript correspondientes a la vista que se quiera "inyectar". En concreto, obtenemos el JavaScript llamado "*IdeaModule.nocache.js*", que corresponde a la vista "Idea" y la inyectamos en el contenedor definido como "*ideaModuleContainer*". Este JavaScript genera una estructura HTML y CSS que es inyectada en este contenedor.

Ahora vamos a definir el punto de entrada de cada vista (EntryPoint), siguiendo con el ejemplo, vamos a explicar cómo está montado el módulo "IdeaModule".

El diagrama de clases asociado es el siguiente:



EntryPoint: Esta es la interfaz que tiene que implementar cualquier componente SmartGwt para ser "renderizado", es el punto de entrada de las vistas.

AbstractModule: Esta clase abstracta es común a todos los módulos de las seis vistas, en él se definen métodos abstractos para obligar a cada módulo a que los implemente. Además, se inicializa el "singleton" AppProperties, mediante el cual, se gestionan los entornos en los que se encuentra la aplicación (Hosted Mode o Liferay).

IdeaModule: Este es el punto de entra de la vista Idea, en él se indican atributos característicos de ese vista, como la especificación del condenedor principal (ideaModuleContainer) dónde hay que inyectar el código HTML generado por el JavaScript de la vista. Además, contiene tres clases internas que se utilizan para comunicarse con las demás vistas, vamos a ver como se utilizan:

```
public void onModuleLoadActions() {
      IIpcEventUtil ipcEventUtil = IpcEventFactory.getInstance()
                        .createIpcEventUtil();

ipcEventUtil.listen(new IdeaEvent(), "ideaEvent");
ipcEventUtil.listen(new ToolbarCategoryEvent(), "toolbarCategoryEvent");
ipcEventUtil.listen(new ToolbarCategoryEvent(), "mainEvent");

}
```

Este el método que inicializa la vista Idea, lo que hacemos es llamar a la factoría de comunicación para obtener una implementación. Con ella, ponemos a la vista Idea a la escucha de tres eventos (ideaEvent, toolbarCategoryEvent, mainEvent), cuando estos se produzcan, se ejecutarán el correspondiente método "change()" de la implementación que se pasa en el primer parámetro.

Por ejemplo, si se produce el evento "ideaEvent", se lanza el método "change()" de la clase "IdeaEvent", que contiene el siguiente código:

```
public void change(String objectId, String pObjectIdSec, String name) {
	resetIdea();
	idea = new Idea(objectId,pObjectIdSec,"0");
	getMainContainerRootPanel().add(idea);

}
```

Lo que hace este método, es inicializar la vista Idea con los parámetros que le llegan de otro Portlet y la inyecta en el contenedor principal (ideaModuleContainer). Es decir, la vista que estaba oculta, aparecería ahora con los datos asociados a una idea concreta.

Este es solo un ejemplo con una de las vista, todas las demás siguen el mismo procedimiento para comunicarse, escuchar eventos de otros Portlets, inicializar la vista u ocultarse.

Con esta comunicación conseguimos una experiencia de usuario muy buena, ya que no es necesario que se recargue toda la página para ver los contenidos. Los Portlets se "renderizan" individualmente y una vez inicializadas, hacen peticiones asíncronas al servidor para obtener los datos y poder mostrárselos a los usuarios.

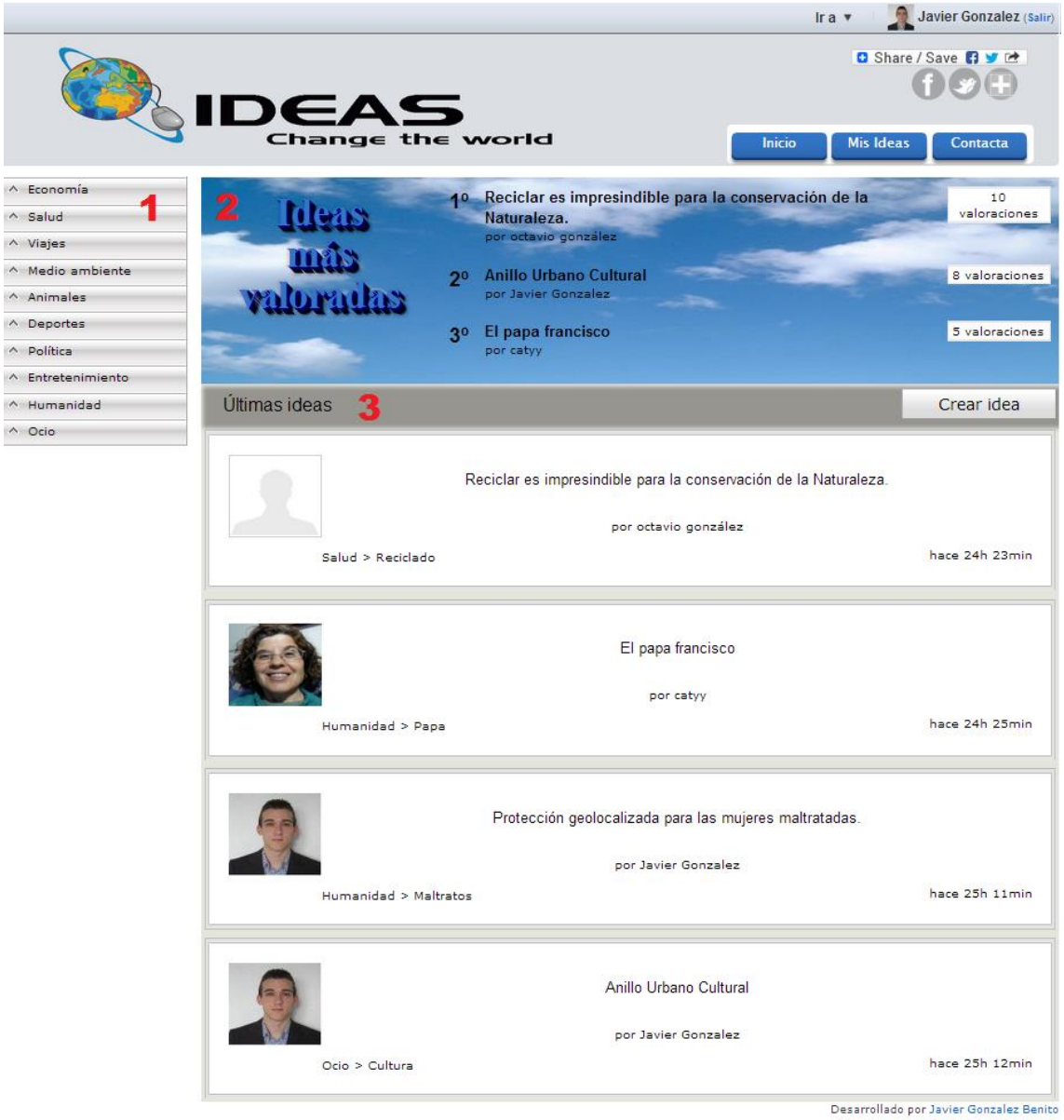
7.- Descripción funcional de la aplicación

Una vez definidos los aspectos técnicos de la arquitectura, ya se puede comprender como funciona cada una de las vistas diseñadas para el cliente. En este apartado se va a explicar cada uno de los seis Portlets diseñados, con el objetivo de comprender que datos muestra, de dónde obtiene esos datos, comunicación con otros Portlets etc.

Los nombres asignados a los Portlets son (están numerados para que en las imágenes se encuentren con facilidad):

1. Toolbar
2. Social Top
3. Last Ideas
4. Category Ideas
5. Idea
6. User Panel

La página inicial de la aplicación en el portal Liferay es la siguiente:



En la parte superior, aparece la barra de usuario, este es un componente que ofrece Liferay y que he utilizado para que cada usuario que acceda a la aplicación, pueda acceder a su perfil y modificar sus datos personales.

En la parte superior, a la derecha, se puede observar varios botones:

- Inicio: Esta es la página de la imagen, en la que están "incrustados" los cinco primero Portlets de la lista. No todos se muestran a la vez, ya que esto no tiene sentido. Unos permanecen visibles y otros ocultos. Cuando un usuario interactúa con ellos, van cambiando y los que estaban visibles se ocultan y al contrario.
- Mis Ideas: Esta es otra página, en ella se encuentra el sexto Portlet. Éste solo se muestra si accede un usuario registrado.

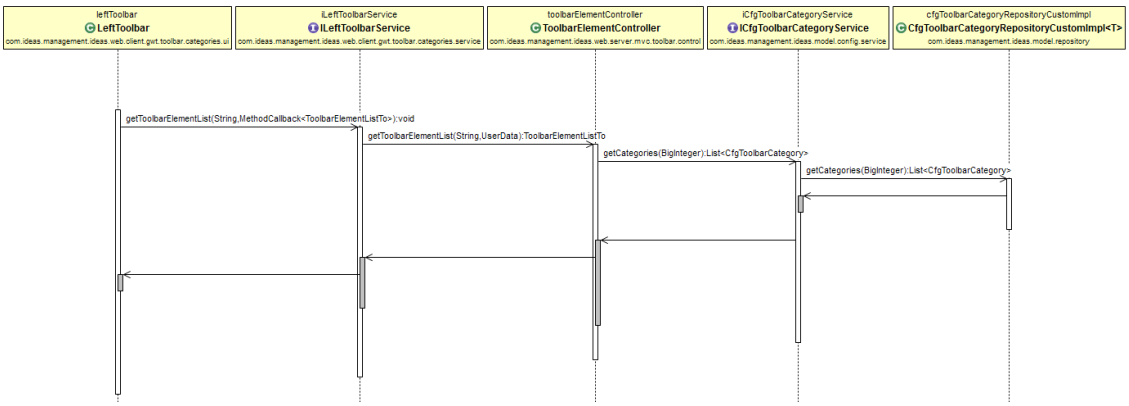
- Contacta: Contiene información acerca del creador de la aplicación.

En la parte izquierda de la imagen, se encuentra el Portlet "Toolbar(1)", y en la parte central, primero "Social Top(2)" y debajo de él "Last Ideas(3)".

A continuación vamos a analizar cada uno de estos tres Portlets para entender su funcionalidad.

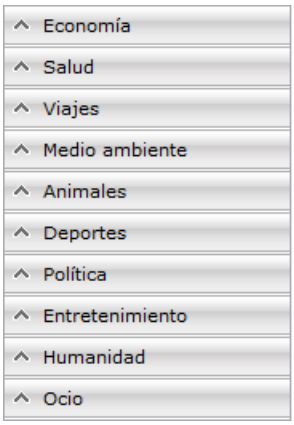
7.1- Toolbar

En esta vista se muestran las categorías y subcategoría que se encuentran en la aplicación. Cuando la vista es "renderizada", se hace una petición HTTP GET asíncrona a la API RESTful definida en la parte servidor, en concreto a la URI: "**/api/categories?parent=0**" (ver especificación de la API). Vamos a ver el diagrama de secuencia asociado a la cadena de llamadas que se realizan al pedir los datos:



- La vista Toolbar realiza la petición asíncrona a través del servicio que obtiene.
- Esta petición se transmite a través del protocolo HTTP, llega al servidor al controlador "ToolbarElementController".
- El controlador, según los parámetros que recibe, hace una consulta a la capa de servicios.
- Los servicios se encargan de obtener los datos de los repositorios y se los mandan al controlador.
- El controlador con los datos obtenidos, rellena el objeto de transferencia y se lo envía a la vista mediante el protocolo HTTP.

El resultado obtenido son las categorías maestras, que se muestran según la imagen:



Cada una de las categorías maestras, se sitúan en un campo de la Toolbar. Al hacer "clic" con el ratón, en alguna de estas categorías, lanzamos un evento que produce una nueva llama asíncrona al servidor, para pedir las subcategorías que pertenecen a la categoría que se ha hecho "clic". Con el resultado obtenido, se añaden "n" sub-módulos a la Toolbar, asociados a la categoría maestra "clicada":

Se forma un árbol de dos niveles, en el cual, el primer nivel se encuentran las categorías maestras y en el segundo nivel las subcategorías. Se pueden abrir y cerrar el primer nivel tantas veces cómo se quiera.

Al hacer "clic" sobre un módulo de segundo nivel, lo que se produce es una publicación de un evento (IPC), en el que se pasa como parámetro el identificador de la categoría que se ha "clicado" (según especificamos anteriormente en el diseño técnico de comunicación entre Portlets). A este evento están suscritos los siguientes Portlets:

- Social Top: Su acción es ocultarse.
- Last Ideas: Su acción es ocultarse.
- Category Ideas: Su acción es mostrarse con la información asociada a la categoría que se ha hecho "clic" en la Toolbar (después se describirá en profundidad este Portlet).

Este evento lanzado, cambia la vista global, en la que aparece otra información para que el usuario pueda seguir interactuando con la aplicación.

7.2- Social Top

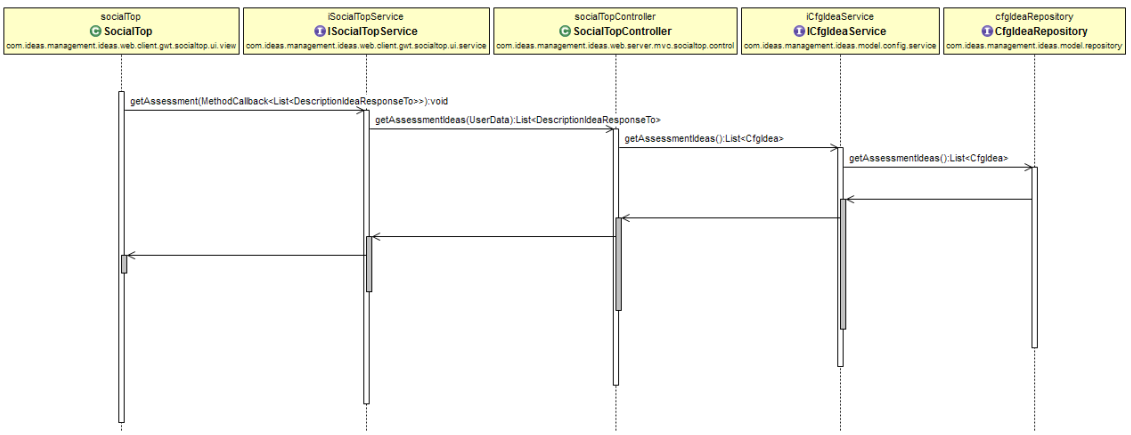
En este Portlet se muestran contenidos sociales de la aplicación, en concreto se trata de una ventana deslizante en la que se muestran dos ventanas, la primera contiene información acerca de las ideas más comentadas y la segunda contiene las ideas más valoradas. Estas vistas están controladas con un "timer", que hace que cada intervalo de diez segundos se muestra una de ellas.

7.2.1- Ideas más valoradas

Esta vista se corresponde a las ideas más valoradas, en ella se muestran las tres ideas más comentadas en ese instante en la aplicación, los campos mostrados son:

- Posición de idea (de más valorada a menos)
- Título de la idea
- Nombre del usuario creador de la idea
- Número de valoraciones que posee.

Para obtener esta información se ha realizado la llamada correspondiente a la API RESTful, en concreto, la URI a la que se realiza la petición es " **/api/social/assessment**" con el método HTTP GET. El diagrama de secuencia asociado a la petición es:



- La vista SocialTop realiza la petición asíncrona a través del servicio que obtiene.
- Esta petición se transmite a través del protocolo HTTP, llega al servidor al controlador "SocialTopController".
- El controlador, según los parámetros que recibe, hace una consulta a la capa de servicios.
- Los servicios se encargan de obtener los datos de los repositorios y se los mandan al controlador.
- El controlador con los datos obtenidos, rellena el objeto de transferencia y se lo envía a la vista mediante el protocolo HTTP.

Con el resultado obtenido, se rellenan los datos mencionados.

7.2.2- Ideas más comentadas

En esta vista se muestran las ideas más comentadas que existen en ese instante en la aplicación, los datos mostrados son:

- Posición de la idea (de más comentada a menos)
- Título de la idea
- Nombre del usuario creador de la idea
- Número de comentarios que posee.

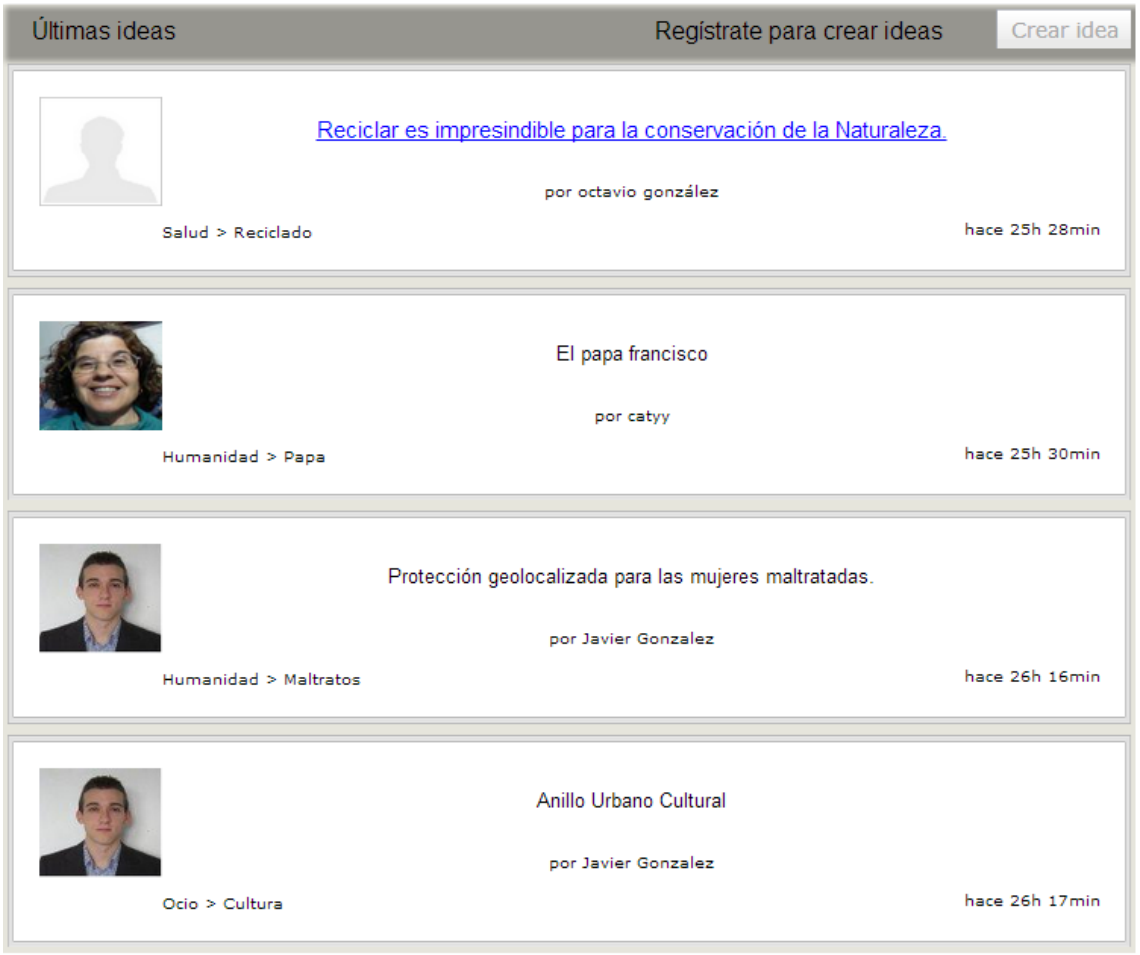
Para obtener la información mostrada, se procede del mismo modo que la vista de ideas más valoradas, se hace una petición asíncrona al servidor a la URI "**/api/social/commented**". El diagrama de secuencia asociado es muy similar al anterior, realiza el mismo camino excepto en la parte servidora, que se procesa en el mismo controlador pero a través de otro método.

Los títulos de ambas vistas, producen un evento (IPC) al hacer "clic" en ellos. Este evento contiene cómo parámetro característico el identificador de la idea "clicada". Los Portlets que están suscritos a este evento son:

- Social Top (él mismo): Su acción es ocultarse.
- Last Ideas: Su acción es ocultarse.
- Idea: Su acción es mostrarse con la información asociada a la idea que se ha hecho "clic" en Social Top (después se describirá en profundidad este Portlet).

7.3- Last Ideas

En este Portlet se muestran un resumen de las diez últimas ideas añadidas en la aplicación, ordenadas de la más reciente a la más antigua. Vamos a analizarla en profundidad:



En la parte superior se indica mediante un título, que el contenido son las "últimas ideas". A la derecha, se encuentra un botón para crear una nueva idea, pero se encuentra deshabilitado (junto con el mensaje correspondiente) ya que se trata de un usuario "no registrado". Como se mencionó anteriormente, los usuarios "no registrados" no pueden crear ideas, por lo que para habilitar este botón, será necesario registrarse.

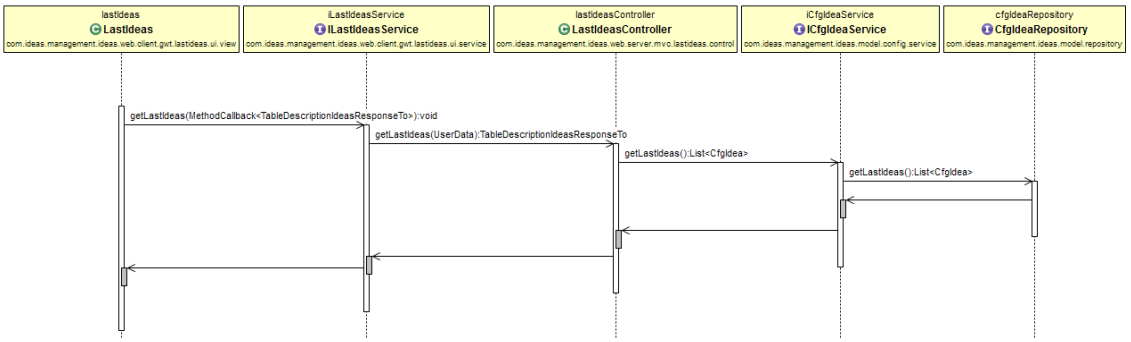
Si accede un usuario registrado, este botón aparecerá habilitado y se le dará la opción de crear una idea (después analizaremos esta vista).

Debajo de la cabecera, se encuentra un listado con las últimas ideas, los contenidos mostrados son:

- Foto del usuario creador de la idea.
- Título de la idea.
- Nombre del creador de la idea.

- Categoría y subcategoría a la que pertenece.
- Tiempo transcurrido desde que se creó la idea.

Para obtener los datos mostrados, como en las demás vistas, se ha realizado una petición asíncrona al servidor de aplicaciones, en concreto a la URI "**/api/social/lastideas**" con el método HTTP GET. El diagrama de secuencia asociado es el siguiente:



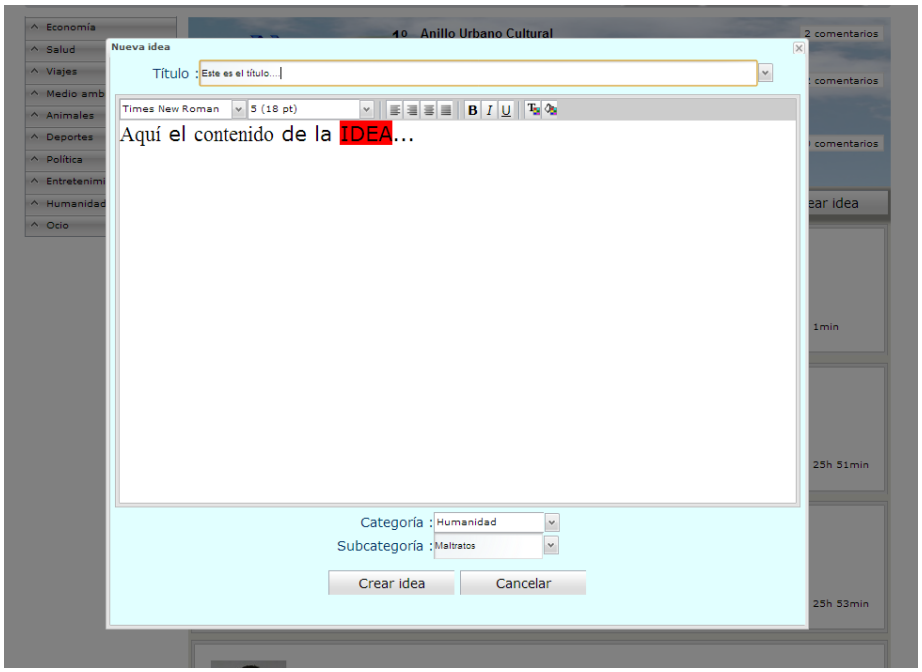
- La vista Last Ideas realiza la petición asíncrona a través del servicio que obtiene.
- Esta petición se transmite a través del protocolo HTTP, llega al servidor al controlador "LastIdeasController".
- El controlador, según los parámetros que recibe, hace una consulta a la capa de servicios.
- Los servicios se encargan de obtener los datos de los repositorios y se los mandan al controlador.
- El controlador con los datos obtenidos, rellena el objeto de transferencia y se lo envía a la vista mediante el protocolo HTTP.

Cada uno de los títulos de las ideas mostradas, producen un evento (IPC) al hacer "clic" en ellos. Este evento contiene cómo parámetro característico el identificador de la idea "clicada". Los Portlets que están suscritos a este evento son:

- Social Top: Su acción es ocultarse.
- Last Ideas (él mismo): Su acción es ocultarse.
- Idea: Su acción es mostrarse con la información asociada a la idea que se ha hecho "clic" en Social Top (después se describirá en profundidad este Portlet).

7.3.1- Crear idea

Como hemos mencionado, la vista Last Ideas contiene un botón en la parte superior para crear ideas:



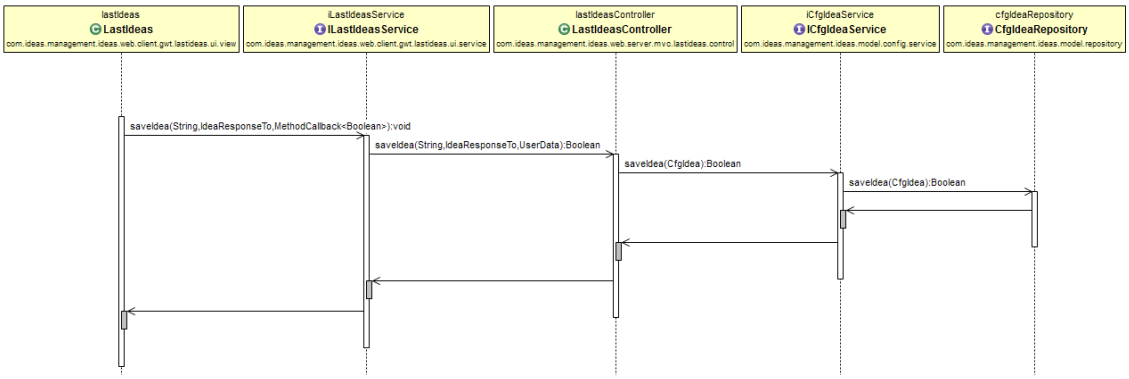
Al hacer "clic" en él, se muestra el "pop up" de la imagen, quedando los demás Portlets debajo y con un color oscurecido, para dar sensación de que el "pop up" es la única venta activa. Esta vista contiene todo lo necesario para crear la idea:

- "Text Box" para añadir el título de la idea.
- Un editor de texto enriquecido* en el que se puede describir la idea, de una forma sencilla, pudiendo dar tamaño a la letra, seleccionar el tipo de letra que queramos, seleccionar distintos colores de letra o subrayado y demás opciones, que ofrecen al usuario una forma muy agradable de explicar su idea.
- En la parte inferior se encuentra dos "combo box", primero hay que seleccionar la categoría principal y después la subcategoría a la que queremos asociar la idea. Si no existe una subcategoría adecuada, se da la posibilidad de añadir una nueva.
- Una vez completados todos los datos, se dispone del botón "Crear idea" para confirmar y crear la misma, o "cancelar si deseamos volver atrás".

*Hay que destacar, que el editor de texto enriquecido genera código HTML, y este es el que almacenamos en base de datos. Esto puede tener un riesgo de seguridad, que consiste en inyección de código malicioso. Para que no se produzca ningún ataque que pueda violar la seguridad del sistema, el contenido de la idea se "filtra", por si contiene algún "script" malicioso este no sea almacenado.

En esta vista se producen varias peticiones a la API RESTful, la más destacada es la que se hace al recurso

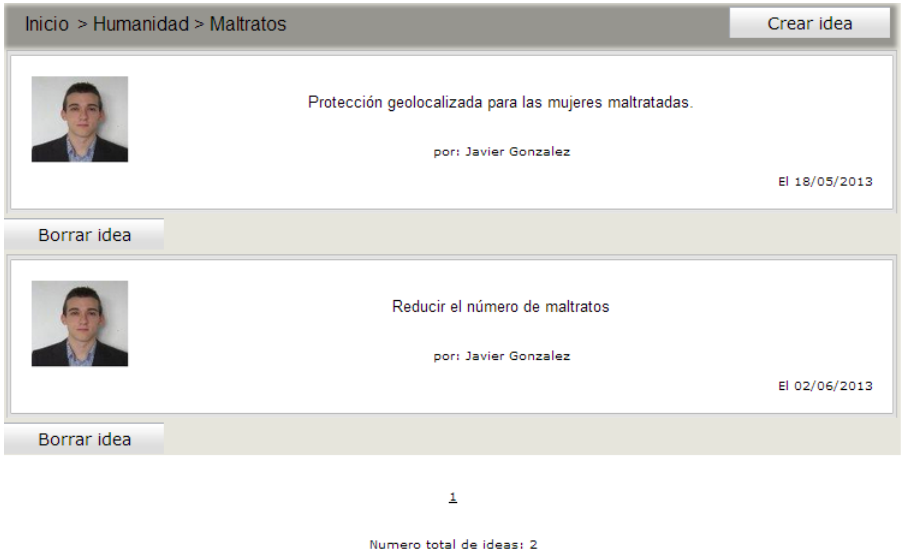
"/api/categories/{categoryId}/ideas", para crear la idea. Este es el diagrama de secuencia asociado:



- La vista Last Ideas realiza la petición asíncrona a través del servicio que obtiene.
- Esta petición se transmite a través del protocolo HTTP, llega al servidor al controlador "LastIdeasController".
- El controlador, según los parámetros que recibe, hace una consulta a la capa de servicios.
- Los servicios se encargan de obtener los datos de los repositorios y se los mandan al controlador.
- El controlador con los datos obtenidos, rellena el objeto de transferencia y se lo envía a la vista mediante el protocolo HTTP.

7.4- Category Ideas

Este Porlet se inicializa oculto, y se muestra cuando se produce un evento en la Toolbar tal y como se definió. Cuando se "renderiza" se muestra el siguiente contenido:



En la parte superior, se encuentra un "camino de migas" que indica la subcategoría en la que nos encontramos. El primer campo, "Inicio" es un

botón, que al hacer "clic" lanza un evento (IPC) al que están suscritos los siguientes Portlets:

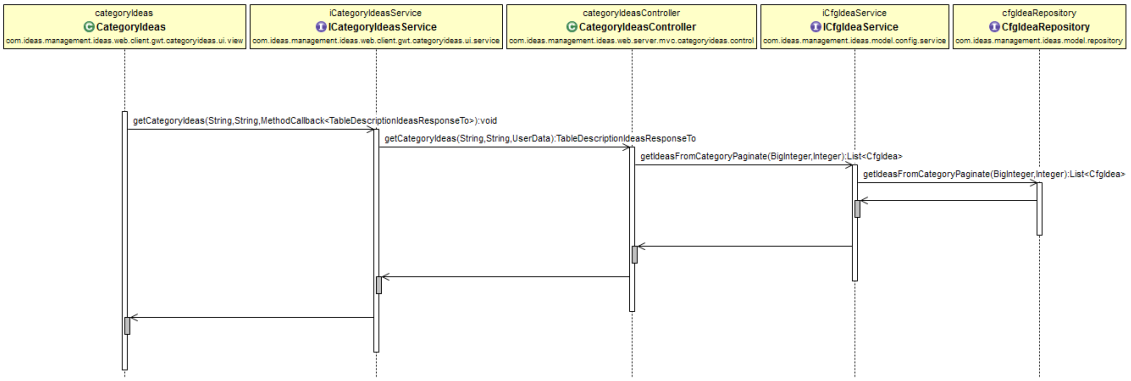
- Social Top: Su acción es hacerse visible ya que estaba oculto.
- Last Ideas: Su acción es hacerse visible, ya que estaba oculto.
- Category Ideas: Su acción es ocultarse.

En la parte superior derecha se encuentra el botón "Crear Idea". Este tiene las mismas características que definimos en el apartado de Last Ideas, con la diferencia, que al añadir a través de este botón no se puede seleccionar una categoría ni una subcategoría, ya que la asociamos a la mostrada en el camino de migas.En el contenido central se muestran las ideas asociadas a la subcategoría que se indica en el camino de migas, paginadas de 10 en 10. Nos podemos mover a través de las páginas con el componente de paginación situado en la parte inferior de la vista.

Los datos que se muestran en la descripción de las ideas son:

- Foto del usuario creador de la idea.
- Título de la idea.
- Nombre del creador de la idea.
- Fecha en la que se creó la idea.

Estos datos se obtienen mediante una petición asíncrona a la API RESTful, en concreto la petición se hace a la URI: "**/api/categories/{categoryId}/ideas?page=0**". El diagrama de secuencia asociado a esta petición es:



- La vista Category Ideas realiza la petición asíncrona a través del servicio que obtiene.
- Esta petición se transmite a través del protocolo HTTP, llega al servidor al controlador "CategoryIdeasController".
- El controlador, según los parámetros que recibe, hace una consulta a la capa de servicios.
- Los servicios se encargan de obtener los datos de los repositorios y se los mandan al controlador.

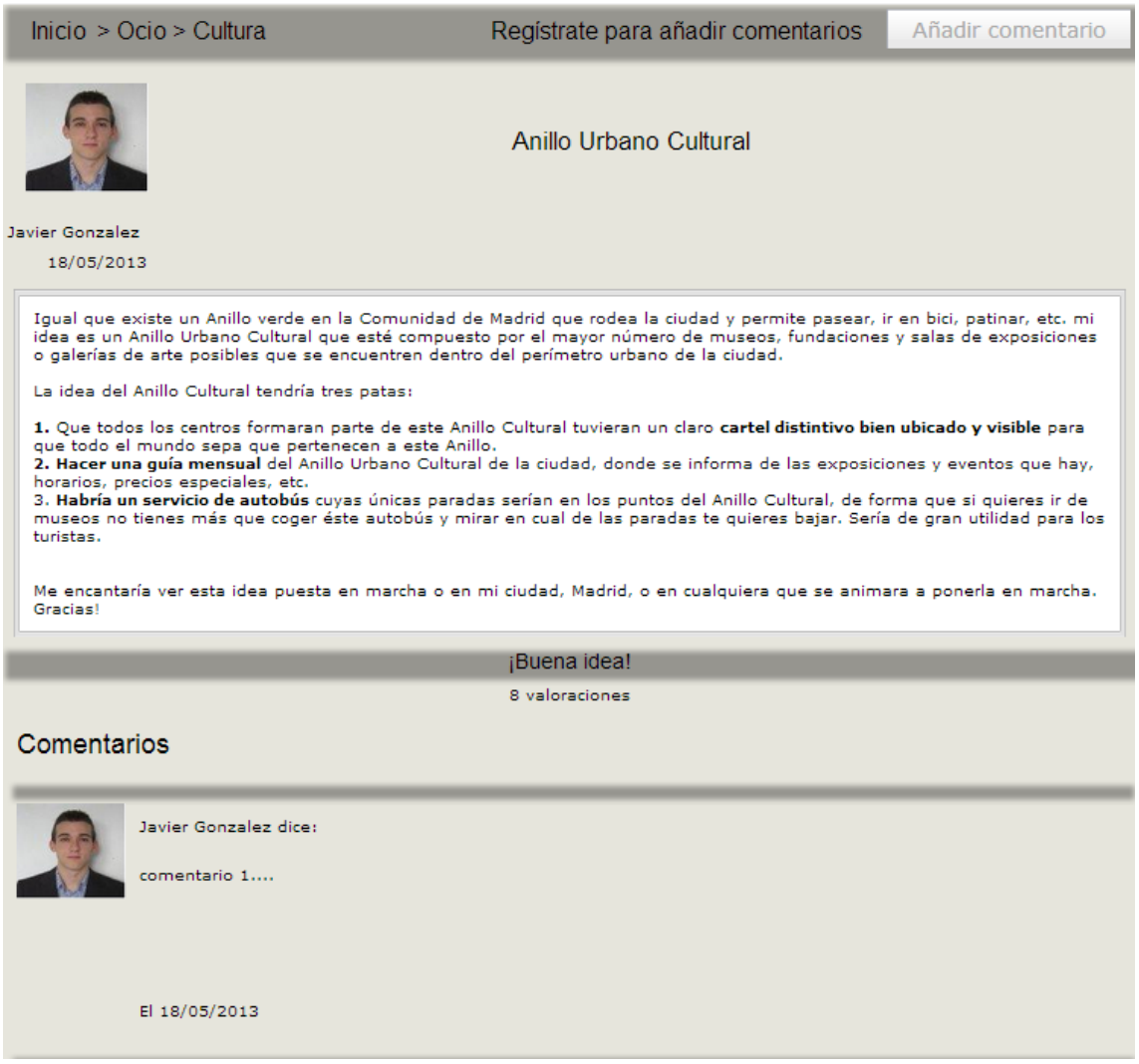
- El controlador con los datos obtenidos, rellena el objeto de transferencia y se lo envía a la vista mediante el protocolo HTTP.

Como podemos observar en la imagen, debajo de cada idea, se encuentra un botón de borrar, esto se debe a que el usuario que estaba en el sistema en ese momento, tenía el rol de "administrador", por lo que puede borrar cualquier idea con este botón. Los usuarios "no registrados" y "usuarios" no tienen privilegios para realizar esta acción, y este botón no les aparece.

Los títulos de las ideas son botones, que al hacer "clic" en ellos, lanza un evento (IPC) de comunicación con otro Portlets, en concreto el Portlet que está suscrito a este evento, es el Portlet "Idea", su acción es inicializarse con el identificador de idea que recibe en este evento.

7.5- Idea

Este Porlet inicializa oculto inicialmente, a la espera de recibir alguno de los eventos a los que se encuentra suscrito. Cuando lo recibe, muestra el siguiente contenido:



En la parte superior, se encuentra un "camino de migas" que indica la categoría y subcategoría que pertenece la idea. El primer campo, "Inicio" es un botón, que al hacer "clic" lanza un evento (IPC) al que están suscritos los siguientes Portlets:

- Social Top: Su acción es hacerse visible ya que estaba oculto.
- Last Ideas: Su acción es hacerse visible, ya que estaba oculto.
- Idea (sí mismo): Su acción es ocultarse.

En la parte superior derecha, se encuentra el botón "Añadir comentario", se encuentra deshabilitado para los usuarios que no están registramos en el sistema, si desean añadir un comentario asociado a esta idea, tendrán que registrarse. Después describiremos este botón.

A continuación, aparece la imagen de perfil del usuario creador de la idea, junto con su nombre y la fecha de creación de la misma. Además, se muestra el título de la idea.

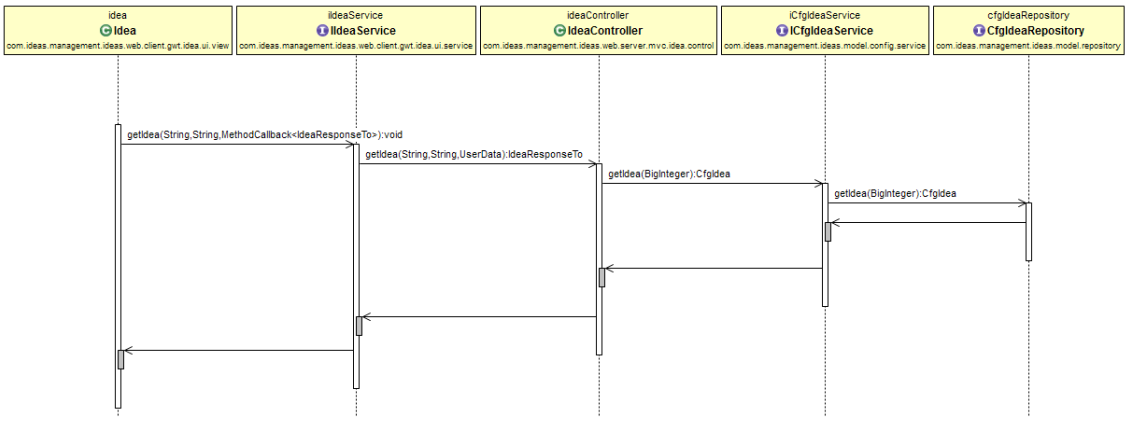
En la parte central, se muestra un cuadro HTML, en el que se muestra el contenido de la idea tal y como definió el creador de la idea.

Después se sitúa uno de los contenidos sociales, en concreto, el botón para valorar una idea. Este se llama "¡Buena idea!", y al hacer "clic" sobre él, se incrementa el número de valoraciones asociado a esa idea. El objetivo de esta acción es que a los usuarios que les gusta la idea que acaban de leer, puedan valorarla positivamente.

Por último se encuentra una lista de los comentarios asociados a esa idea, en la que figuran los siguientes campos:

- Imagen de perfil del creador del comentario.
- Nombre del usuario creador del comentario.
- Contenido del comentario.
- Fecha en la que se creó el comentario.

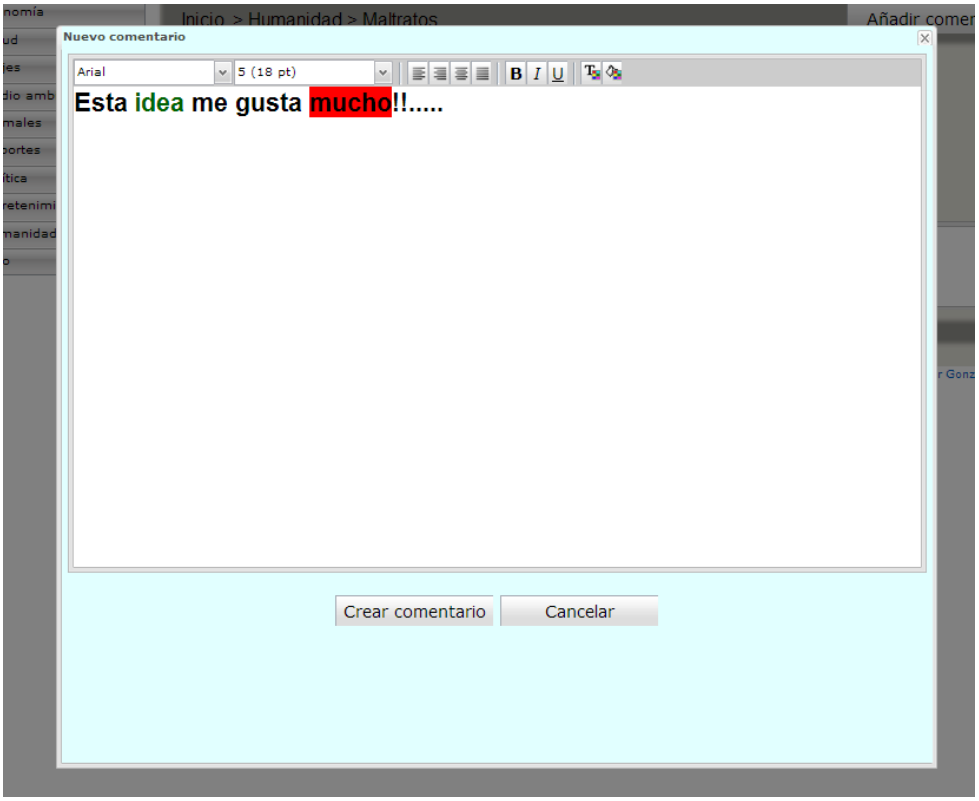
Para la obtención de todos estos datos mostrados, ha sido necesario realizar una petición asíncrona al servidor a la URI "**/api/categories/{categoryId}/ideas/{ideaId}".** Este es el diagrama de secuencia asociado:



- La vista Idea realiza la petición asíncrona a través del servicio que obtiene.
- Esta petición se transmite a través del protocolo HTTP, llega al servidor al controlador "IdeaController".
- El controlador, según los parámetros que recibe, hace una consulta a la capa de servicios.
- Los servicios se encargan de obtener los datos de los repositorios y se los mandan al controlador.
- El controlador con los datos obtenidos, rellena el objeto de transferencia y se lo envía a la vista mediante el protocolo HTTP.

7.5.1- Añadir comentario

En la vista anterior, se mencionó que se podía crear un comentario a través del botón "Añadir comentario", al hacer "clic" se muestra la siguiente vista.



Como podemos observar, aparece un "pop up" centrado en la pantalla, dejando por detrás los demás Portlets "oscurecidos" para dar sensación al usuario de que lo de atrás está desactivado.

En esta ventana se puede añadir el contenido del comentario, mediante un editor de texto enriquecido, que ofrece las mismas posibilidades que el de "crear idea". Al finalizar el texto que se quiera introducir, se da la posibilidad de añadir o volver atrás, mediante dos botones que se muestran.

7.6- User Panel

Este Portlet se encuentra incrustado en la página "Mis ideas" del menú superior, en él se muestran las ideas creadas por el usuario que se encuentra dentro del portal. Por este motivo, los usuarios no registrados, no tienen acceso a este Portlet, se les muestra un mensaje un mensaje informativo para que se registren.

El objetivo es que este Portlet sea un panel de administración para que el usuario pueda gestionar sus ideas.

Esta es la imagen:

En la parte superior se muestran una serie de pestaña, en las que se muestran las categorías principales en las que ese usuario tiene creadas ideas. Al hacer "clic" en una de ellas, se abre un panel secundario con más pestañas, en ellas se muestran las subcategorías que pertenecen a la

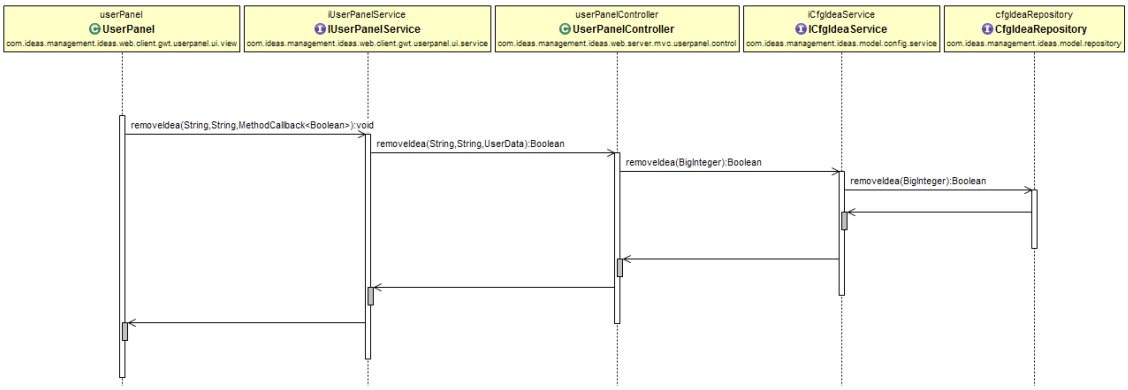
categoría "clicada". A continuación se muestra una lista de las ideas que tiene el usuario en esa subcategoría.

La información mostrada es:

- Imagen de perfil
- Título de la idea
- Fecha de creación de la idea

En la parte inferior se muestran las acciones que se pueden tomar con las ideas mostradas. En esta iteración del proyecto solo ha dado tiempo a incluir una acción, en concreto esta acción es la de borrar la idea. Para ello, se muestra una imagen de una papelera. Para borrar una idea, basta con pincha en la idea y arrastrarla hasta la papelera, al soltarla encima de la papelera se lanza un evento que la borra. Esta forma de interactuar con la aplicación se llama "drag and drop" y ofrece una forma muy intuitiva para el usuario de realizar las tareas.

El evento lanzado lo que hace es hacer una petición asíncrona a la API RESTful para borrar el recurso, este es el diagrama de secuencia asociado:



- La vista User Panel realiza la petición asíncrona a través del servicio que obtiene.
- Esta petición se transmite a través del protocolo HTTP, llega al servidor al controlador "UserPanelController".
- El controlador, según los parámetros que recibe, hace una consulta a la capa de servicios.
- Los servicios se encargan de obtener los datos de los repositorios y se los mandan al controlador.
- El controlador con los datos obtenidos, rellena el objeto de transferencia y se lo envía a la vista mediante el protocolo HTTP.

7.7- Internacionalización (i18n)

Todas las vistas analizadas componen el cliente propuesto, con el que puede interactuar el usuario. Cabe destacar, que la aplicación ha quedado preparada para la internacionalización (i18n), por si en un futuro se quiere traducir a cualquier lenguaje, además del español.

Esta internacionalización está dividida en tres partes:

- Los componentes de Liferay, como el menú superior, se pueden traducir desde el panel de administración de Liferay directamente.
- Las colecciones de base de datos, tienen su estructura preparada para almacenar las traducciones que se deseen (tal y como se describió en el tema de modelo de datos).
- En todas las vistas, existen textos estáticos que pueden ser traducidos a partir de ficheros ".properties" que tiene asociado cada vista.

8.- Sistema de pruebas

Cuando se desarrolla un proyecto, es necesario tener una infraestructura de pruebas, con el objetivo de tener en todo momento "testeados" todos los componentes de la aplicación y detectar rápidamente posibles fallos en alguno de ellos. La idea es que los test tengan la máxima cobertura del código desarrollando, para que todos los caminos posibles sean probados.

Cada módulo (capas) en que se ha dividido el proyecto, cuenta con sus propios "test" unitarios y de integración. La herramienta Jenkins utilizada, se encarga de "lanzar" continuamente los test, e informar de los resultamos obtenidos.

Para la parte del servidor se ha utilizado las tecnologías JUnit y Mockito, que ofrecen una serie de acciones y métodos para realizar estas pruebas.

Además, para todas estas pruebas, se ha creado un sistema de creado y borrado dinámico de datos, para que los test sean totalmente independientes.

8.1- Creado y borrado de datos para las pruebas

Las pruebas que se realizan contra sistemas externos como MongoDB, dependen de que siempre existan los mismos datos, ya que un test hace pruebas contra datos estáticos. Es por esto, que se ha diseñado una estructura para que cuando se inicialice el contexto de Spring, se realice borrado de los datos existentes (de las colecciones indicadas) y posteriormente un llenado de datos a partir de unos ficheros "XML" en que

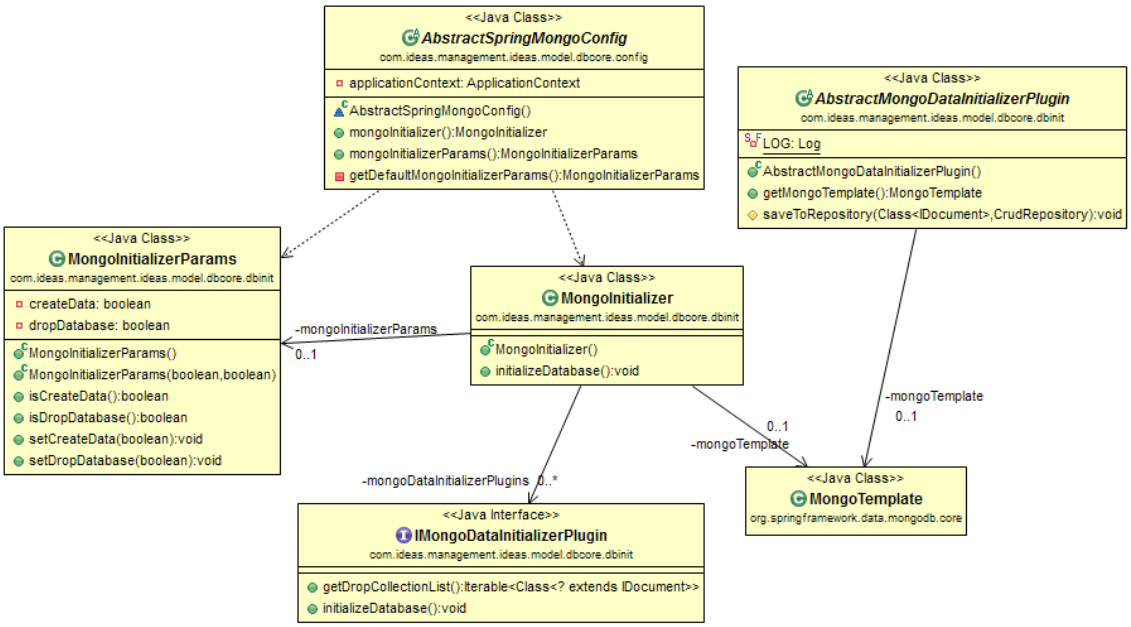
se definen previamente los datos que se van a guardar. Después se pasan los test contra esos datos.

Como definimos en el apartado de perfiles de la aplicación, cada perfil tiene un fichero de configuración. Estos ficheros cuentan con los siguientes campos:

- db.mongo.params.create=false
- db.mongo.params.drop=false

Con el primer campo, indicamos si queremos crear datos de prueba y con el segundo campo definimos si queremos borrar los datos.

A partir de estos datos, vamos a ver como se utilizan, primero veremos el diagrama de clases de la arquitectura diseñada:

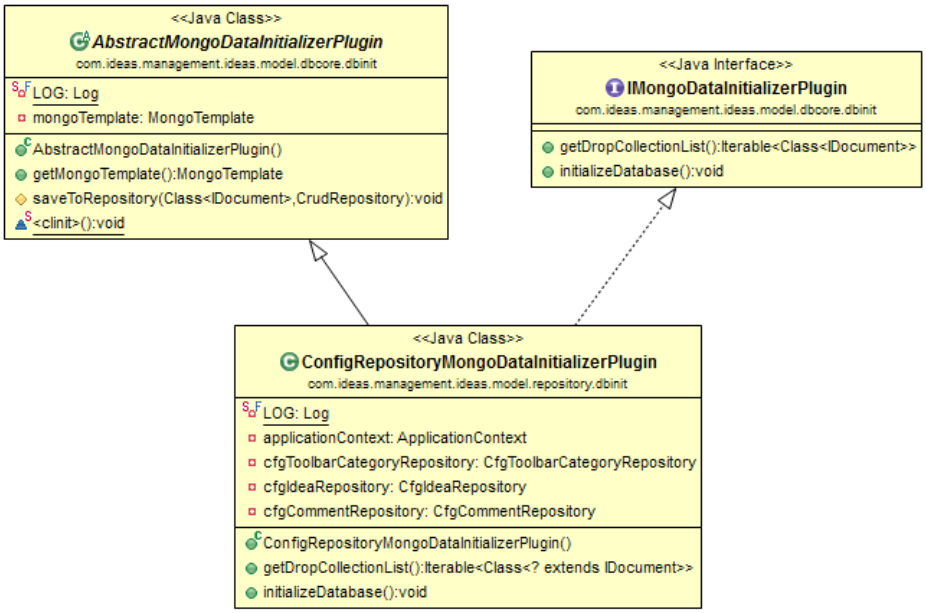


Cuando inicializados el contexto de Spring, lo primero que hacemos es leer el ".properties" asociado al perfíl que estemos utilizando. Los valores que figuren en ese fichero, se guardan en los parámetros preparados para ello, en concreto, nos vamos a fijar en la clase "MongoInizializerParams", que contiene los dos datos referentes a la creación y borrado de datos.

Una vez leidos, se instancian los objetos que hemos anotados a través de las anotaciones de Spring (son singletons), en concreto, nos vamos a centrar en la clase en el método "mongoInitializer()", que contiene lo siguiente:

```
@Bean(initMethod = "initializeDatabase")
public MongoInitializer mongoInitializer()
{
        return new MongoInitializer();
}
```

En la anotación "@Bean" indicamos que al instanciar este "singleton", se lance el método "initializeDatabase()". Este método, lo primero que hace es ver las implementaciones de la interfaz "IMongoDataInitializerPlugin", para saber que colecciones tiene que borrar (sigue el patrón plugin). Esta es el diseño de la implementación definida:



Con el método "getDropCollectionList()" indicamos las colecciones a borrar:

```
public Iterable<Class<? extends IDocument>> getDropCollectionList() {
		final List<Class<? extends IDocument>> classes = new
ArrayList<Class<? extends IDocument>>();
if (!ProfileUtil.containsProfile(ProfileConstant.PRODUCTION,
	applicationContext.getEnvironment().getActiveProfiles())) {
		classes.add(CfgComment.class);
		classes.add(CfgIdea.class);
		classes.add(CfgToolbarCategory.class);
	}
return classes;
}
```

En concreto se trata de las colecciones Ideas, comentarios y categorías.

Como podemos observar se utilizan métodos "tipados" para hacerlo genérico a cualquier colección de base de datos.

Con esta lista de colecciones a borrar, "MongoInitializer" se encarga de llamar a "MongoTemplate" para borrar las colecciones.

Acto seguido, se realiza el mismo procedimiento pero para crear datos, se buscan todos los "plugins" en el contexto y se invoca a los métodos "initializeDatabase". El "plugin" desarrollado, cuenta con el siguiente código:

```
public void initializeDatabase() {
LOG.info("Initialize database in: " + getClass().getSimpleName());
saveToRepository(CfgToolbarCategory.class,
      cfgToolbarCategoryRepository);
saveToRepository(CfgIdea.class, cfgIdeaRepository);
saveToRepository(CfgComment.class, cfgCommentRepository);

}
```

En este método, se asocia la colección que queremos guardar datos de prueba con el repositorio que tiene que utilizar para ello, llamando al método "saveToRepository".

Este método lo que hace es buscar en la carpeta "resources" del proyecto, ficheros XML que se sigan la nomenclatura "*.dbread.xml", dónde "*" es el nombre de la colección.

En el ejemplo, como hemos indicado las colecciones "CfgIdea", "CfgToolbarCategory" y "CfgComment", buscamos tres ficheros que se llamen "CfgIdea.dbread.xml", "CfgToolbarCategory.dbread.xml" y "CfgComment.dbread.xml" respetivamente.

Estos ficheros contiene la siguiente estructura (CfgIdea.dbread.xml):

```
<list>
      <com.ideas.management.ideas.model.config.bean.CfgIdea>
            <objectId>1</objectId>
            <liferayUserId>1111</liferayUserId>
            <createDate>
                  <zeroGmtMatchDate>
                        2414-01-14 19:00:00.0 UTC
                  </zeroGmtMatchDate>
                  <zeroGmtMatchTimeZoneStr>
                        GMT+0
                  </zeroGmtMatchTimeZoneStr>
            </createDate>
            <title>
                  Dinero en el banco de España.
            </title>
            <content>
                  Este es el contenido de la idea…
            </content>
            <category>
                  <objectId>11</objectId>
            </category>
            <userImg>
                  <location>/PATH/A/LA/IMAGEN</location>
            </userImg>
            <coreActiveElement>
                  <active>true</active>
            </coreActiveElement>
      </com.ideas.management.ideas.model.config.bean.CfgIdea>

</list>
```

La estructura de los XML tiene que ser igual a la definida en las colecciones, en el ejemplo se muestra una idea de prueba, con el nombre de los atributos de la colección en las etiquetas y entra ellas se muestra el valor que se quiere guardar. Se pueden añadir "n" elementos ya que es una lista.

Una vez localizados estos ficheros, se utiliza la tecnología "XStream" para leer los XML y transformarlos en objetos JAVA. Acto seguido se llama al repositorio indicado para que lo guarde en base de datos.

Si la estructura está mal se lanzan las excepciones oportunas.

En las capas de "repositorios", "servicios" y "controladores" se usa esta infraestructura de borrado y creado de datos.

8.2- Pruebas para Repositorios y Servicios

Para las pruebas se utiliza el perfil de test, para ello, se ha diseñado una configuración de test especial para ambas capas:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ActiveProfiles(ProfileConstant.TEST)
public abstract class AbstractRepositoryTest<T extends
IGenericRepository> {

…

}

@ContextConfiguration(classes = ServiceConfig.class)
@RunWith(SpringJUnit4ClassRunner.class)
@ActiveProfiles(ProfileConstant.TEST)
public abstract class AbstractServiceTest<T>{

…

}
```

Como podemos observar, con la anotación "@ActiveProfiles", indicamos que se utilice el perfil de "TEST", y con la anotación "@RunWith", indicamos que el punto de arranque de los test será la clase "SpringJUnit4ClassRunner". Esta es una clase de Spring que integra la funcionalidad de JUnit, y hace de "main(punto de arranque)" principal.

A partir de estas dos clases, se lanzan los demás test de las capas repositorios y servicios.

Para definir los test, se ha seguido el siguiente procedimiento.

1. Se crea una clase que extienda a una de las dos mencionadas (dependiendo si queremos hacer test para un repositorio o un servicio).

2. Se inyecta el repositorio o servicio asociado.
3. En el cuerpo de la clase, se definen tantos métodos como test se quieran hacer y estos se anotan con "@Test".
4. Dentro del método, se pueden hacer las pruebas que se quieran, por ejemplo usando "Asserts". Ejemplo:

```
@Test
public void getLastIdeasTest() {

	List<CfgIdea> result = cfgIdeaRepository.getLastIdeas();
	Assert.assertEquals(2,result.size());
}
```

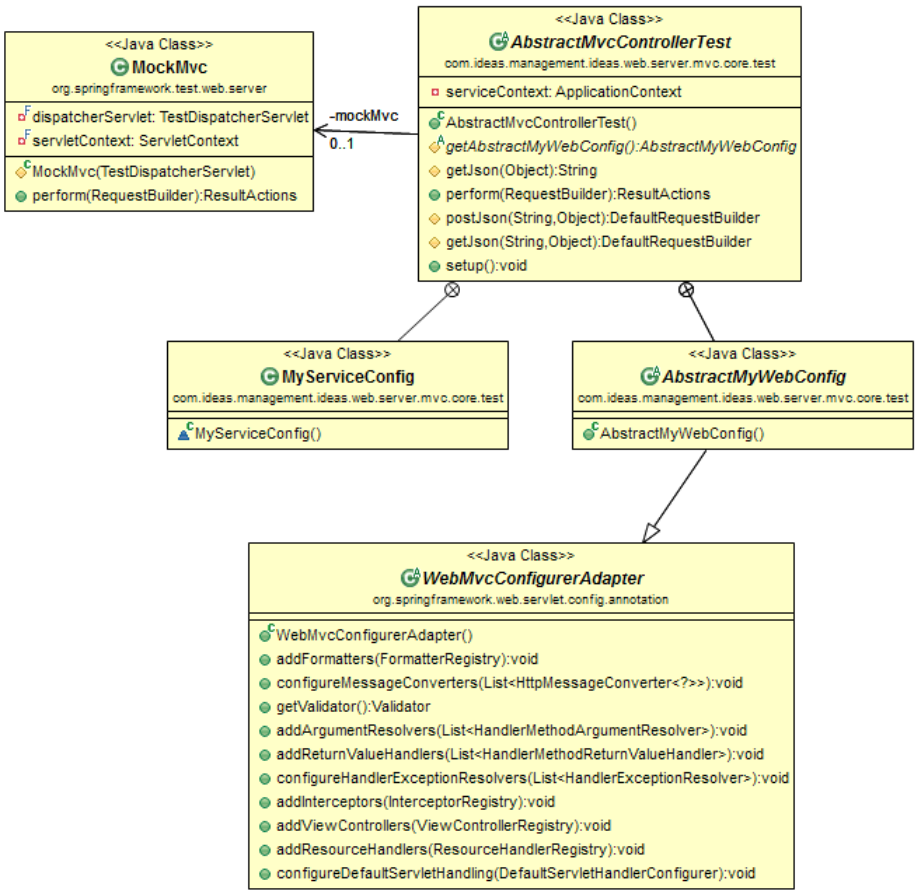
Con este diseño, antes de que se ejecute cada uno de estos métodos anotados con "@Test", se borrarán los datos y se crearán datos de prueba. Acto seguido se ejecutará el test.

8.3- Pruebas para Controladores

Cada uno de los controladores implementados, cuenta con sus test unitarios. En ellos, se utiliza el mismo método de borrado y creado dinámico de datos que se ha explicado en el apartado anterior.

Para probar los métodos desarrollados, se ha montado una infraestructura que simula las peticiones HTTP que los controladores tienen que contestar, ya que no son métodos que podamos probar directamente. Por tanto, se han utilizado librerías de Spring y de Mockito para disponer de un "servlet embebido" en tiempo de ejecución de test, dónde se despliega el controlador que se va a probar. Una vez desplegado, se simula una petición HTTP, incluso se puede añadir al cuerpo de la petición un JSON.

Vamos a ver el diagrama de clases:

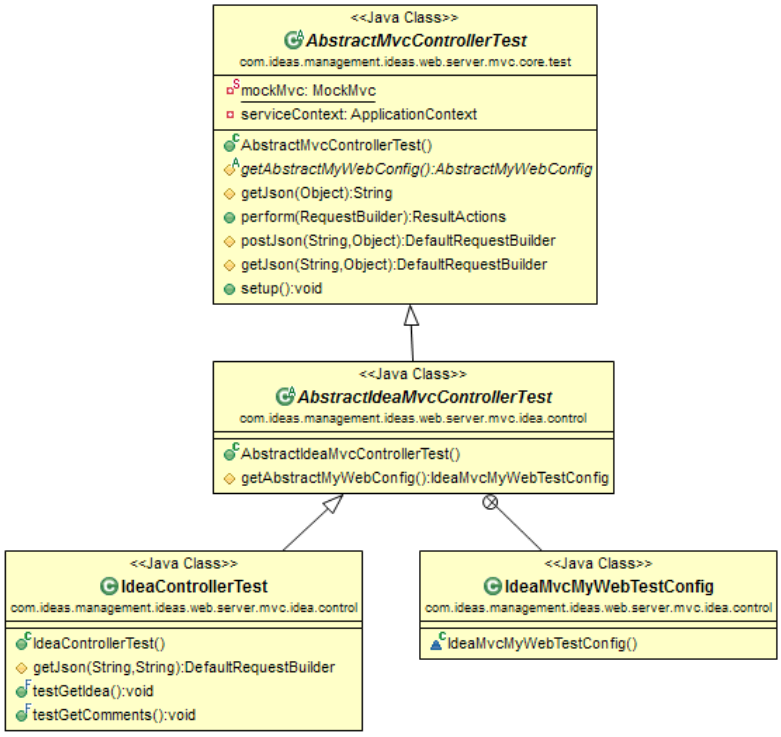


En la clase "AbstractMvcControllerTest" se centra todo el contenido, lo primero que se hace es definir dos clases internas, "MyServiceConfig" y "AbstractMyWebConfig", en ellas se habilitan las configuraciones de Spring MVC y se deja preparada las estructura para que cada controlador que las utilice, pueda añadir su propia configuración con los servicios que necesite utilizar.

Después se define el MockMvc, esta clase es la encargada de simular el servlet en el que se despliega el controlador. Será el encargado de devolver las respuestas a las peticiones simuladas que reciba de los test.

Además contiene los métodos postJson() y getJson() que son los que se utilizan para simular las peticiones a los controladores desde los test.

Para que cada controlador pueda utilizar esta estructura, se ha definido el siguiente modelo:



Este diseño es el definido para el controlador "IdeaController", como podemos observar, se define una clase abstracta ("AbstractIdeaMvcControllerTest") que extiende a la definida en el diagrama anterior. Gracias a esto, obtiene todos los métodos implementados en ella. Después se define una clase interna estática, dónde define toda su configuración (servicios que tiene que utilizar, etc).

Por último se crea una clase dónde se añaden los test para ese controlador, en este caso "IdeaControllerTest", vamos a ver un ejemplo:

```
@Test
public final void testGetIdea() throws Exception {

	DefaultRequestBuilder requestBuilder = getJson(
		"/api/categories/11/ideas/1",null);
	super.perform(requestBuilder).andDo(print()).
		andExpect(status().isOk());
}
```

Este es uno de los test que posee la clase, en él se define el tipo de método que se va a utilizar para la simulación de la petición, en este caso es de tipo HTTP GET (getJson()). A este método le asociamos la URI a la que queremos hacer la petición, en el ejemplo, la URI hacer referencia a una idea con identificador "1" que pertenece a una categoría con identificador "11". Estos datos deberán existir, lógicamente, para ello los habremos definido previamente en los "XML" de creación de datos de pruebas.

En el ejemplo se comprueba que el resultado obtenido sea un "200 ok", indicando que la petición se ha realizado satisfactoriamente, pero se pueden añadir todas las pruebas que queramos, como comprobar los resultados obtenidos en el JSON de respuesta.

8.4- Pruebas SmartGWT

Para las pruebas en la parte visual se ha utilizado un plugin de SmartGWT para eclipse. Este plugin ofrece la posibilidad de desplegar un servidor embebido para visualizar la vista que se quiera probar en un navegador web, a esto lo llamaremos "Hosted Mode".

Además, no es necesario compilar el código a JavaScript, directamente con el código JAVA se pueden realizar las pruebas o incluso depurar.

Cada una de las vistas desarrolladas contiene un proyecto de test, en el que se define un "Entry Point" o punto de entrada, tal y como se definió en el diseño técnico de los Portlets.

A partir de este punto de entrada, se inyecta la vista que queremos visualizar en el navegador web, con los parámetros que se requieran. Vamos a ver un ejemplo del punto de entrada para la vista idea:

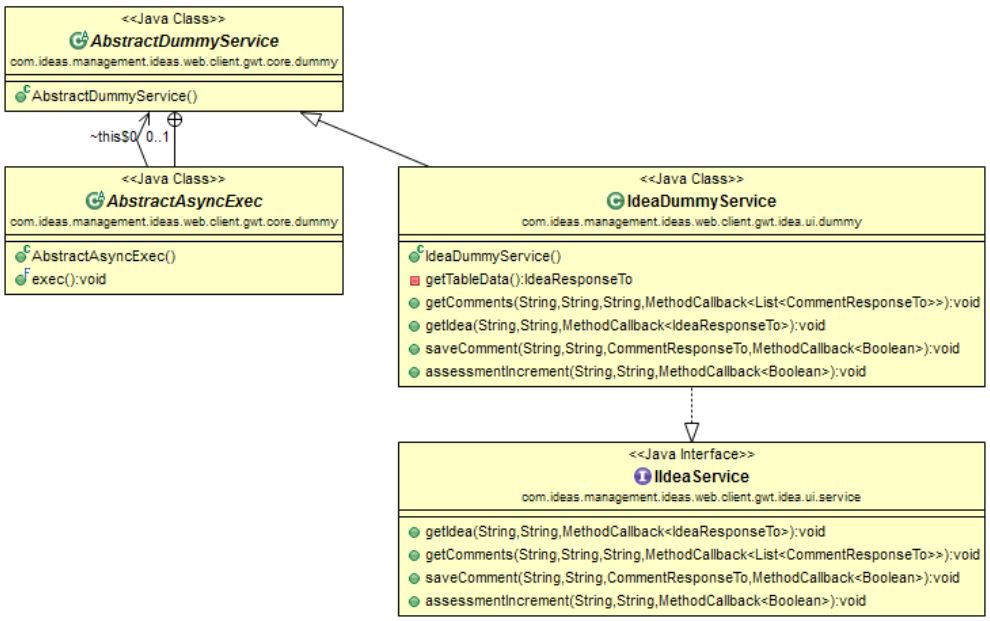
```
public class IdeaTestEntryPoint extends AbstractTestEntryPoint {

	private Idea idea;

	@Override
	public void onModuleLoad() {
		super.onModuleLoad();
		idea = new Idea("", "", "");
		RootPanel.get().add(idea);

	}

}
```

Como podemos observar, se instancia la vista Idea y se añade al contenedor principal.

Además del punto de entrada, es necesario generar unos ficheros de configuración propios de SmartGwt, dónde se indican los parámetros de configuración y librerías que tiene que utilizar para levantar correctamente el servidor embebido.

Para simular la llamada de las peticiones asíncronas que realizan las vistas, se ha implementado un "dummy" para cada vista, que devuelve datos de prueba, estos "dummy" sólo se utilizan en cuando estamos en el entorno de "Hosted Mode". Recordar, que esta estructura se definió en el apartado técnico con la factoría de servicios.

Cada uno de los dummy, posee exactamente los mismo métodos que la llamada real a la API RESTful del servidor, incluso simula un tiempo de retardo (1 segundo) para hacer lo más real las pruebas. Para la vista Idea, la estructura de "dummy" es la siguiente:



Como podemos observar, implementa los métodos de la interfaz del servicio, en ellos rellenan los objetos de transferencia con datos de prueba y se devuelve la petición simulada a la vista para que los utilice.

9.- Liferay

Como se ha mencionado durante toda la memoria, la aplicación cliente se basa en el portal de Liferay, en la que los Portlets desarrollados se incrustan en las páginas qué son creadas y configuradas dinámicamente a través del panel de control que ofrece Liferay.

A cada una de estas páginas se les puede asociar un "Tema" y un "Layout". El tema sirve para añadir los estilos a la página y el Layout se utiliza para definir la disposición de los Portlets en la misma.

También es necesario definir a través del panel de control los grupos de usuario en los que agrupamos a los usuarios para darles unos privilegios u otros.

9.1- Tema

El tema de Liferay desarrollado se compone de estilos (.css) y templates (.vm).

En los estilos se ha añadido la maquetación necesaria para diseña la cabecera de las páginas y el pie de página.

Este es el diseño:

Hay que destacar los siguientes contenidos de la cabecera:

- Logo: Este es el logotipo diseñado específicamente para la aplicación.
- Acceso: Este es un enlace al acceso del sistema.
- Redes Sociales: Se ofrecen posibilidades para compartir con amigos la aplicación, a través de redes sociales, como Facebook, Google+ etc.
- Menú de navegación: Con este componente nos podemos mover a través de las tres páginas que componen la aplicación.

A través de los "templates" se generan dinámicamente las páginas HTML que recibe el usuario. A estos "templates" se les han añadido mediante la API de Liferay Portlets incrustados. En concreto se ha utilizado el Portlet "Web Content Display" que sirve para añadir directamente desde el portal los contenidos HTML, JavaScript o CSS. Los contenidos que se han mencionado en la cabecera, están incluidos dentro de estos Portlets, con el objetivo de que un administrador que posee los roles para modificarlos, pueda en cualquier momento cambiar el logo, las redes sociales etc. directamente en el portal, sin necesidad de desplegar nada.

Para incrustar estos Portlets, se ha utilizado la siguiente llamada a la API de liferay:

```
$theme.runtime("56_INSTANCE_odCB4Xy6n4dY")
```

Con esto insertamos el Portlet en la posición dónde se encuentre la llamada dentro del "template". El parámetro que recibe es el identificador del Portlet.

Para el botón de acceso al sistema, se ha seguido el mismo procedimiento, solo que en ese caso se procedido a añadir un enlace a el Portlet "Login", a través del cual, los usuarios pueden acceder al sistema, registrarse o incluso recuperar una contraseña perdida.

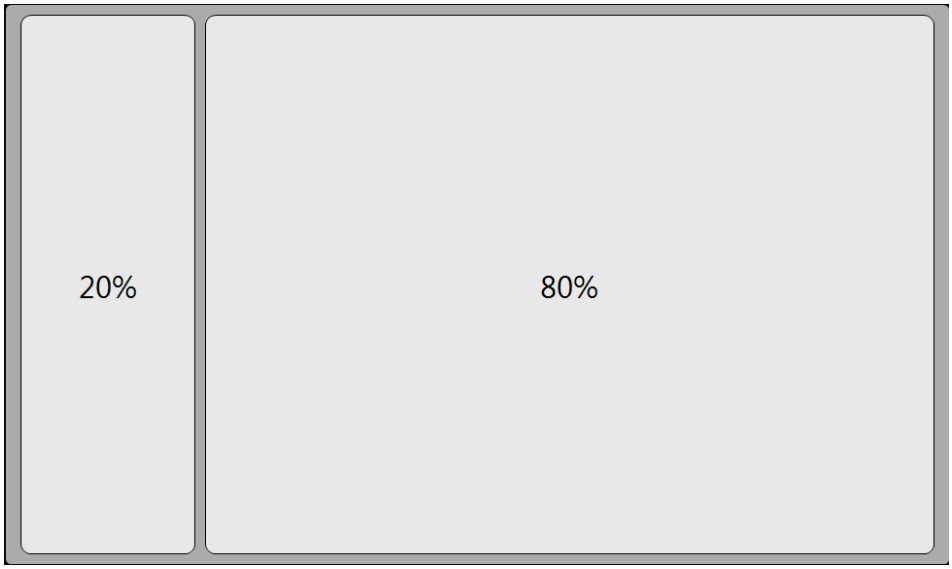
Hay que destacar que el diseño de las páginas se ha hecho pensando en una estructura estándar de 1024 pixeles de ancho.

9.2- Layout

Con los layouts, se define la disposición de los Porlets dentro de la página en la que se encuentran. Liferay posee un sistema de "drag and drop" mediante el cual, un usuario con los permisos adecuados puede arrastrar los Portlets de una posición a otra del layout directamente desde el portal sin necesidad de programar nada. Para la aplicación ha sido necesario el desarrollo de dos layouts, uno para la página "Inicio" y el segundo para la página "Mis ideas".

A continuación, vamos a ver la estructura de estos dos layouts:

Inicio



Se ha divido el contenedor de los Porlets en dos columnas, la primera tiene un tamaño de 20% (de 1024 píxeles) y la segunda columna tiene un porcentaje de 80% (respecto a 1024px).

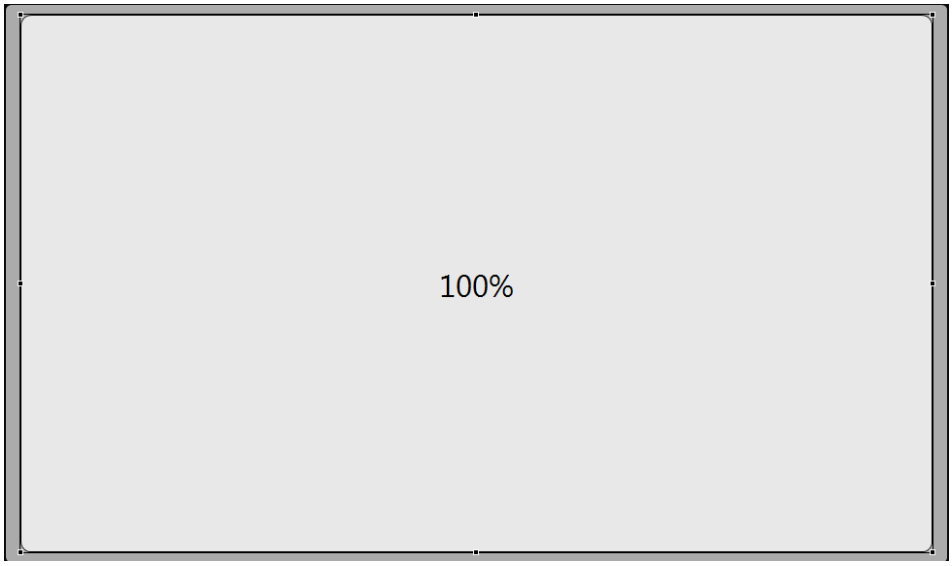
El código asociado es el siguiente:

```
<divclass="template"id="main-content"role="main">
<divclass="portlet-layout">
        <divclass="aui-w20 portlet-columnportlet-column-first"id="column-1">
                $processor.processColumn("column-1", "portlet-column-contentportlet-column-
                content-first")
        </div>
        <divclass="aui-w80 portlet-columnportlet-column-last"id="column-2">
                $processor.processColumn("column-2", "portlet-column-contentportlet-column-
                content-last")
        </div>
        </div>
</div>
```

Este es el código que se utiliza para generar dinámicamente los HTML, a partir de estos "templates".

En la primera columna se ha insertado el Portlet Toolbar, y en la segunda, los demás (excepto User Panel) según el orden en que aparecen en el diseño funcional que se describió anteriormente.

Mis ideas



En esta página sólo se encuentra el Portlet User Panel, es por esto que se le ha dado el espacio máximo, el 100% (de 1024px).

El código necesario para este diseño de layout es:

```
<divclass="template"id="main-content"role="main">
        <divclass="portlet-layout">
                <divclass="portlet-columnportlet-column-only"id="column-1">
                        $processor.processColumn("column-1", "portlet-column-contentportlet-
                        column-content-only")
                </div>
        </div>
</div>
```

Hay que destacar que tanto el tema, como los layouts desarrollados, son aplicaciones web, es decir, tienen sus ficheros de configuración propios de Liferay y el imprescindible "web.xml", en estos ficheros se ha añadido todas las funcionalidades necesarias para desplegarlos correctamente en el servidor de aplicaciones.

Una vez están en el servidor de aplicaciones, desde el panel de administración de Liferay, se configura en qué páginas se quiere poner cada componente.

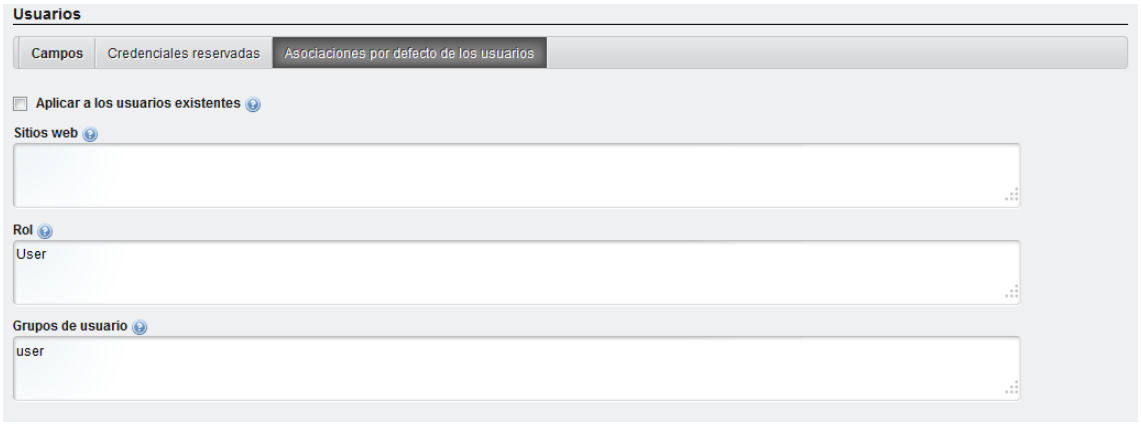
9.3- Configuración

Para utilizar correctamente la aplicación dentro del portal Liferay, es necesario configurar una serie de parámetros. Todas estas configuraciones las hacemos a través del panel de administración que ofrece Liferay para los usuarios que tengan los privilegios adecuados.

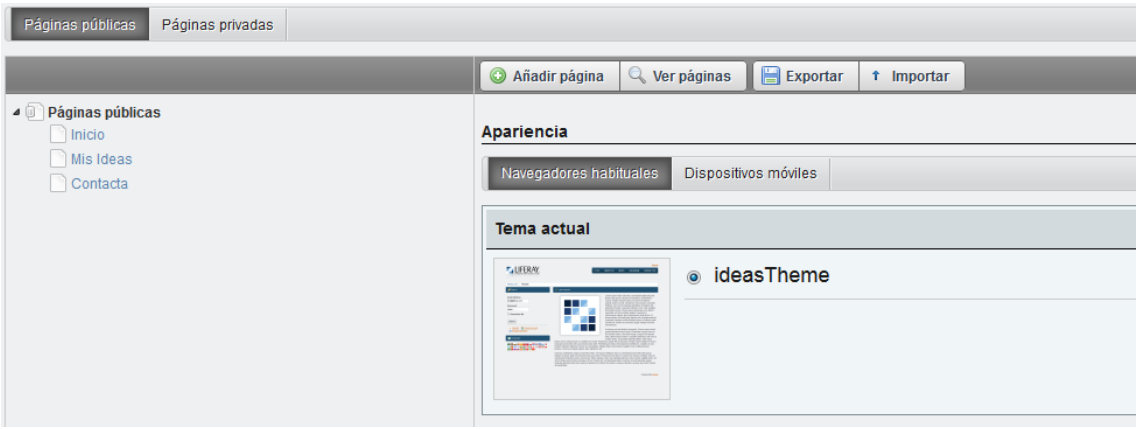
Lo primero es crear los grupos de usuarios, mediante los cuales vamos a gestionar los roles de los usuarios:

Como se puede ver en la imagen, se crean dos grupos, uno para los usuarios registrados ("user") y otro para los administradores de ideas ("admin").

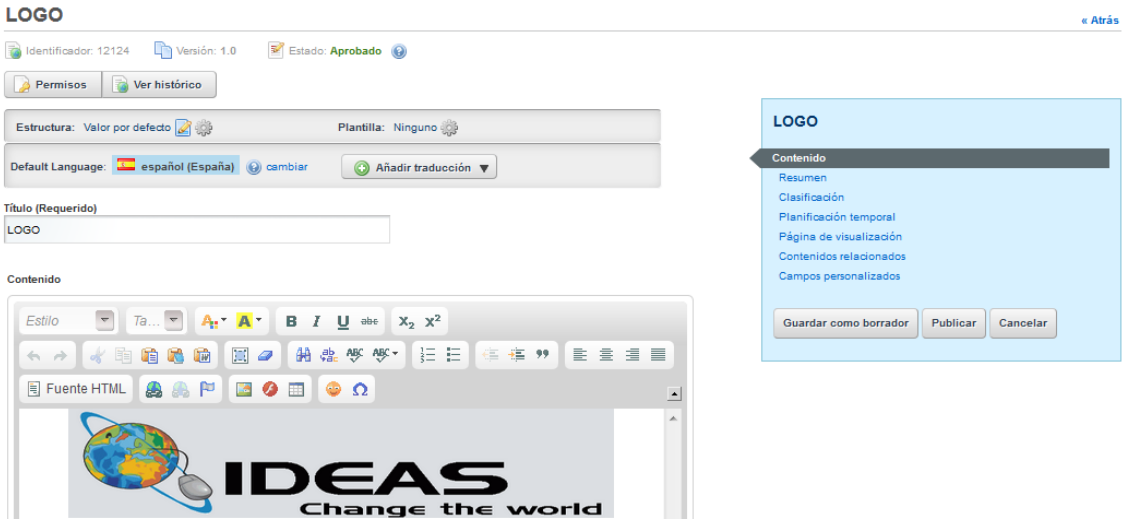
El siguiente paso, es definir el grupo por defecto que se asigna a los usuarios que se registren, en este caso el grupo"user". Para ello, en la siguiente ventana se indica como asociación por defecto el grupo "user":



Además, es necesario crear las páginas que aparecen en el menú de navegación principal, esto se realiza directamente desde el panel de administración de Liferay. A cada una de estas páginas se les asocia una "friendly-url" distintas además de un tema y un layout.



Para los contenedores web que se incrustan en el tema, existen un panel de administración desde dónde se puede añadir los contenidos al mismo. En la siguiente imagen se muestra el editor visual dónde añadimos el logo de la aplicación.



Para las redes sociales, se realiza la misma operativa.

10.- Seguridad

Para la autentificación en el sistema, se ha utilizado las tecnologías Portlet Security y Spring-Security. Es de vital importancia gestionar la autentificación correctamente, para que ningún usuario no autorizado pueda acceder a datos que no debería.

Spring-Security nos ayuda a gestionar los contenidos que se le muestran a casa usuario, sin darles la posibilidad de violar el acceso a la aplicación. Utilizando este conocido framework, nos aseguramos que nuestra aplicación va a ser segura.

Con Portlet-Security, se ha diseñado un "filtro", por dónde tienen que pasar todas las peticiones a la API RESTful. En este filtro, se identifica al usuario que está haciendo la petición y se le asignan los roles adecuados.

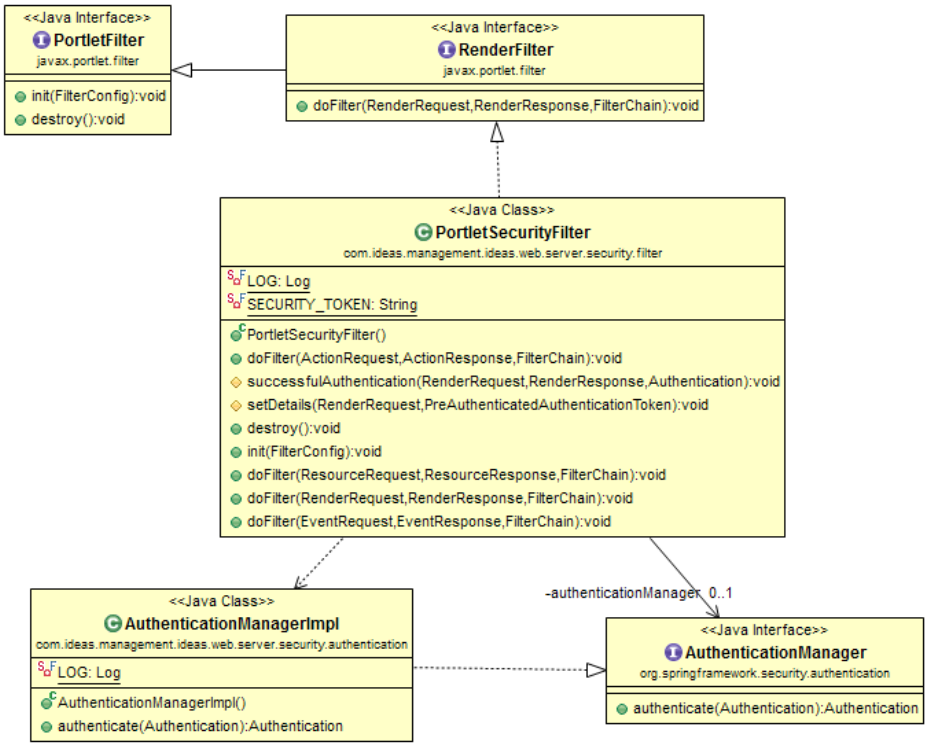
Vamos a ver estas dos implementaciones detalladamente.

10.1- Filtro

Según define el estándar JSR-286 de especificación de Portlets, los Portlets deben implementar un mecanismo para poder añadir filtros a las distintas fases de vida de un Portlet. En esta aplicación sólo nos vamos a centrar en la parte de "render" (pintado) ya que es la parte dónde añadiremos el filtro.

El objetivo de añadir este filtro es poder identificar al usuario que realiza la petición, y asignarle el rol correcto, para que los controladores reciban en el campo "UserData" los atributos correctos del usuario que realiza la petición.

Para este cometido se ha diseñado el siguiente diagrama de clases:



La clase "PorletSecurityFilter" compone la implementación del filtro, en concreto, el método que se ejecuta cada vez que se "renderiza" un Portlet, es el método "doFilter()", en el se realizan las siguientes acciones:

- Si el usuario no tiene sesión, se le asigna el rol "UNREGISTERED" (no registrado).
- Si tiene sesión de Liferay, se utiliza el método "authenticate()" de la clase "AuthenticationManagerImpl" para identificar al usuario, esto se realiza de la siguiente forma:

```
public Authentication authenticate(Authentication auth)
                        throws AuthenticationException {
                Collection<GrantedAuthority> authList = new ArrayList<GrantedAuthority>();
        try {
                LOG.debug("Roles del usuario " + auth.getName() + ": ");

                for (UserGroup userGroup : UserLocalServiceUtil.getUser(
                                Long.valueOf(auth.getName()).longValue()).getUserGroups()) {
                        authList.add(new SimpleGrantedAuthority(userGroup.getName()));
                        LOG.debug(userGroup.getName());
                        }
                } catch (NumberFormatException e) {
                        LOG.error("Error: No se pudieron obtener los roles del usuario.");
                } catch (SystemException e) {
                        LOG.error("Error: No se pudieron obtener los roles del usuario.");
                } catch (PortalException e) {
                        LOG.error("Error: No se pudieron obtener los roles del usuario.");
                }
                return new UsernamePasswordAuthenticationToken(auth.getName(),
                                auth.getCredentials(), authList);
        }
```

Utilizamos el servicio de Liferay "UserLocalServiceUtil" para obtener los grupos de usuario a los que pertenece el usuario, recordemos que son dos, "user" o "admin". Es decir, se utiliza el grupo de usuario como si fuera un rol. A partir de este grupo obtenido, se agrega un "token" de seguridad a su sesión, dónde se especifica el grupo al que pertenece.

En el apartado técnico se describió la función de la clase "SessionParamArgumentResolver", y se definió que obtenía los roles, pero no como los obtenía. Es a partir de este "token" que almacenamos en sesión, junto con las credenciales, de dónde se saca la información del grupo al que pertenece, y dependiendo del grupo se asigna un rol u otro.

Para que este filtro se despliegue correctamente en el contexto, tanto del Porlet como de Spring, es necesario realizar una serie de configuraciones. La principal es añadir al fichero de configuración "portlet.xml" (del Portlet que queremos que tenga el filtro) el siguiente código:

```
<filter>
        <display-name>idea</display-name>
        <filter-name>PortletSecurityFilter</filter-name>
        <filter-class>
                com.ideas.management.ideas.web.server.security.filter.PortletSecurityFilter
        </filter-class>
        <lifecycle>ACTION_PHASE</lifecycle>
        <lifecycle>EVENT_PHASE</lifecycle>
        <lifecycle>RENDER_PHASE</lifecycle>
        <lifecycle>RESOURCE_PHASE</lifecycle>
        <init-param>
                <name>message</name>
                <value>Security Filter</value>
        </init-param>
</filter>
<filter-mapping>
        <filter-name>PortletSecurityFilter</filter-name>
        <portlet-name>idea</portlet-name>
</filter-mapping>
```

Con esta configuración, indicamos al Portlet (en este caso el Portlet "Idea") que añada un filtro definido en la estructura anterior en todo su ciclo de vida.

10.2- Gestión de contenido

En el Portlet "User Panel" es necesario gestionar el contenido que se muestra a los usuarios, ya que si accede un usuario que no está registrado, no se tiene que inicializar la vista de SmartGwt, sino, queremos que se le muestra un mensaje informativo indicándole que debe registrarse para tener un panel de usuario propio.

Este control de acceso no lo tienen que realizar las vistas, ya que estas se tienen que abstraer de este tipo de lógica de negocio. Para este cometido se han utilizado los grupos de usuario que se resuelven en el filtro anterior, vamos a ver un ejemplo con el punto de entra de la vista "User Panel":

```
<%@ taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet"%>
<%@ taglib prefix="sec"
uri="http://www.springframework.org/security/tags"%>

…

< sec:authorize ifNotGranted="user,admin">
       <p><p>No estas registrado, registrate para tener un panel de control
       propio</p></p>
</sec:authorize >

<sec:authorize ifAnyGranted="user,admin">
<script type="text/javascript" language="javascript"
       src="<%=request.getContextPath()%>/UserPanelModule/UserPanelModule.n
ocache.js"></script>
<table align="center" width="100%">
       <tr>
              <td id="userPanelModuleContainer"></td>
       </tr>
</table>
</sec:authorize>
```

Como podemos observar, se utilizan los "taglibs" de Spring-Security para mostrar un contenido u otro en función del grupo al que pertenezca el usuario. En concreto, el "taglib" "sec:authorize ", muestra la sección de código que se encuentra entre las etiquetas si se cumple la condición indicada en el parámetro.

En el ejemplo mostrado, se divide en dos secciones. La primera, contiene un mensaje informativo para los usuarios no registrados. Como condición se utiliza el parámetro "ifNotGranted", y se le da valor "*user,admin*", esto quiere decir, que si el usuario que accede, no tiene ninguno de estos dos roles se le muestra este contenido.

En la segunda sección, se hace justo lo contrario, se utiliza el atributo "ifAnyGranted" para preguntar si el usuario que accede tiene uno de los dos roles. Si se cumple, se inyecta la vista de User Panel (SmartGwt).

De esta forma controlamos los contenidos que se muestran al usuario, sin darle la oportunidad de violar la seguridad del sistema.

11.- Conclusión y líneas futuras

Durante los últimos cinco meses que ha durado el proyecto de final de carrera, se han realizado multitud de acciones. Llegado a este punto marcado como final de proyecto, es necesario pararse y echar la vista atrás, para analizar el conjunto de componentes desarrollados, con el objetivo de ver si el trabajo realizado cumple los objetivos establecidos.

Como visión general, se puede decir que todos los objetivos marcados al inicio del proyecto han sido cumplidos, incluso me atrevo a decir que se ha superado con creces, debido a la ambición que se ha tenido a la hora de elegir las tecnologías para desarrollar la aplicación.

Como punto de partida, hay que mencionar que la metodología ágil utilizada (Scrum), ha contribuido enormemente a la hora de alcanzar los objetivos marcados, ya que la división del proyecto en tareas pequeñas y fáciles de estimar, ha facilitado establecer los objetivos para llegar al plazo de entrega previsto. Hay que destacar que la metodología que quería seguir al inicio del proyecto era de tipo "RUP", pero a la hora de estimar todo el contenido de la aplicación, me decante por "Scrum" por las razones mencionadas.

Funcionalmente, la aplicación diseñada ofrece una base muy sólida para contener todas las ideas y demás aspectos sociales propuestos. Las funcionalidades implementadas en los Portlets, junto con Liferay, ha posibilitado a que los usuarios puedan acceder al contenido de las ideas e interactuar con los demás usuarios de la aplicación, de una forma sencilla e intuitiva.

La idea sería probar con un conjunto de usuarios la aplicación, para encontrar defectos o añadir mejoras que se propongan. Este sería el primer punto en el caso de salir a producción con una BETA de la aplicación.

La API que ofrece ha seguido estrictamente el modelo arquitectónico RESTful, que estable que todos los recursos tienen que estar correctamente estructurados, para que el acceso a los mismo mediante el protocolo HTTP sea lo más intuitivo posible. Además, esta API es independiente del cliente en el que se utilice. Es por esto que si quisiéramos desarrollar una aplicación móvil, solo tendríamos que implementar las "vistas" correspondientes, ya que toda la parte del servidor se puede aprovechar.

Técnicamente, el proyecto ha sido muy ambicioso, por la cantidad de tecnologías utilizadas. Pero, esta es la idea de un proyecto real, "no hay que reinventar la rueda", para esto existen todos estos "frameworks" de desarrollo. Voy a destacar una vez más, la integración que se ha tenido que realizar con estas tecnologías, que no ha sido nada trivial.

La utilización de metodologías del software y patrones de diseño, ha dado un punto de calidad a la aplicación, dónde se ha intentado en todo momento diseñar módulos con bajo acoplamiento y una alta escalabilidad, intentando siempre diseñar estructuras que se puedan reutilizar.

Como líneas futuras, existe una cantidad de cosas que se pueden hacer para seguir ampliando la aplicación. Vamos a ver los caminos que se pueden tomar.

Como partes de la aplicación que se pueden ampliar, estaría la API RESTful y el modelo de datos. La idea en un futuro sería aumentar las colecciones del modelo de datos, con más aspectos sociales, retos, etc. e incluir en la API todos estos recursos, siguiendo la estructura base que se ha diseñado.

Otra posibilidad sería añadir a la aplicación algún tipo de "data mining", por ejemplo, se podría diseñar algún proceso para clasificar las ideas en subcategorías automáticamente en función de las palabras que tuviera el contenido de la idea. Para este cometido, se podría utilizar el motor de MongoDB, que se mencionó anteriormente, el cual dispone de multitud de usos y posibilidades. Una de ellas sería utilizar "Map/Reduce", para obtener estadísticas o datos informativos de las ideas que contenga la aplicación.

En cuanto al diseño del cliente, se podrían diseñar más Portlets que den la opción de utilizar otros recursos de la API REST. En el panel del usuario, recordemos que solamente se podía borrar ideas, este sería un Portlet al que añadir más acciones.

La comunicación de Portlets propuesta, hace que no se generen URLs identificativas, ya que toda la comunicación se realiza en la parte de cliente y las peticiones que se hacen son asíncronas. Para este tipo de aplicaciones se usa el objeto "hash (#)", incluido en el objeto "Location" del navegador. La idea sería que cada vez que se realice una acción, se añadan campos representativos en este "hash", y así tener una URL representativa de cada recurso.

Por ejemplo, si nos encontráramos viendo una idea, una URL descriptiva sería:

www.ideas.com#salud/hospitales/reducir_tiempo_de_consultas

En la URL mostrada se puede ver la categoría, subcategoría y nombre de la idea.

Otra función interesante sería añadir un módulo de auto respondedor, que mandara e-mails con las ideas más valoradas (por ejemplo), a todos los usuarios que quisieran suscribirse al boletín.

Como punto final, mencionar que toda la aplicación ha quedado preparada para ser internacionalizada, por lo que se podría traducir a cualquier idioma.

12.- Bibliografía

[1] Eric Freeman y Elisabeth Freeman, "Head First Design Patterns".

[2] Apache Maven, documentación oficial. [Documento WWW] URL http://maven.apache.org

[3] MongoDB, documentación oficial. [Documento WWW] URL http://www.mongodb.org/

[4] Spring-data-mongo, documentación oficial. [Documento WWW] URL http://static.springsource.org/springdata/mongodb/docs/current/reference/html/

[5] Liferay, documentación oficial. [Documento WWW] URL http://www.liferay.es/

[6] Spring-MVC, documentación oficial. [Documento WWW] URL http://static.springsource.org/spring/docs/3.0.x/reference/mvc.html

[7] Spring-REST, documentación oficial. [Documento WWW] URL http://static.springsource.org/spring/docs/3.0.0.M3/reference/html/ch18s02.html

[8] JUnit, documentación oficial. [Documento WWW] URL http://junit.org/

[9] Mockito, documentación oficial. [Documento WWW] URL http://code.google.com/p/mockito/

[10] Apache Log4J, documentación oficial. [Documento WWW] URL http://logging.apache.org/log4j/2.x/

[11] SmartGWT, documentación oficial. [Documento WWW] URL http://code.google.com/p/smartgwt/

[12] Resty-GWT, documentación oficial. [Documento WWW] URL http://restygwt.fusesource.org/

[12] Spring-Security, documentación oficial. [Documento WWW] URL http://static.springsource.org/spring-security/site/index.html

[13] Sonar, documentación oficial. [Documento WWW] URL http://www.sonarsource.org/

[14] Jenkins, documentación oficial. [Documento WWW] URL http://jenkins-ci.org/

[15] ObjectAid, documentación oficial. [Documento WWW] URL http://objectaid.com/

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Fri Feb 14 18:45:52 CET 2014
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)