ksike
FRAMEWORK

2010

Concepción tecnológica

Framework orientado a la construcción de aplicaciones Web, basadas en Ajax.

Autor:
Ing. Antonio Membrides Espinosa

Revisores:
Ing. Yampier Medina Tarancón
Lic. Sarily Porras De La Guardia

Versión: 1.0

Índice

Índice de Figuras

Índice de Tablas

Introducción

En la actualidad el desarrollo de las nuevas tecnologías de la información y las comunicaciones, con la incorporación de los ordenadores, los sistemas de comunicación por satélite, la telefonía celular y el fax ha llegado al punto tal que el término *web* ha comenzado a ganar su propio espacio como la forma hegemónica en la que circulan información y entretenimiento. Por tanto su crecimiento desenfrenado está ocasionando un impacto en la sociedad donde las personas tiendan a realizar la mayoría de sus actividades por esta vía. Provocándose como consecuencia el surgimiento de nuevos conceptos, tales como:

- ✓ **Web 1.0:** Internet básica
- ✓ **Web 1.5:** Sitios dinámicos.
- ✓ **Web 2.0:** La red social de colaboración.
- ✓ **Web 3.0:** La red semántica.
- ✓ **Web 4.0:** La red móvil.
- ✓ **Web 5.0:** La red sensorial o emotiva.

Independientemente de la puesta en práctica de estos conceptos, se puede afirmar que su facilidad de administración centralizada la hace ideal para el despliegue tanto en redes de amplio alcance como corporativas. La factibilidad de uso de sus interfaces y el hecho de que cada día más personas están acostumbradas a la navegación por Internet hace que el tiempo de aprendizaje se reduzca considerablemente respecto a las tradicionales aplicaciones de escritorio.

Este tipo de aplicaciones se le denomina a aquellos productos de software que los usuarios pueden utilizar accediendo a un servidor web a través de Internet o de una intranet. En otras palabras, es un sistema que se codifica en un lenguaje soportado por una especie de máquina virtual denominada navegador y es publicado a través de un protocolo estándar para la intercomunicación entre ordenadores. Para mayor comprensión véase la figura 1.

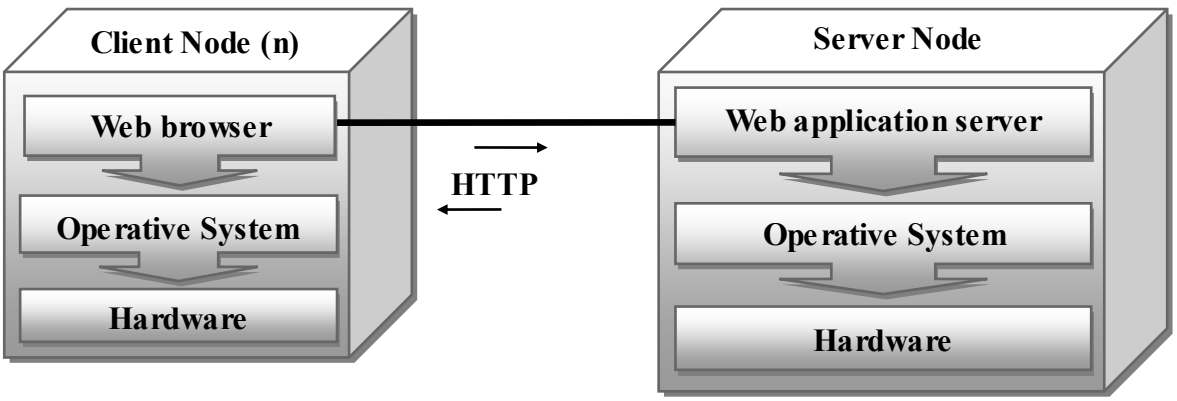


Figura 1: *Modelo Cliente-Servidor para aplicaciones Web*

En la actualidad estas tecnologías se combinan. Por tan solo citar un ejemplo, los celulares no se limitan a la función de comunicar a dos personas entre sí, sino que han evolucionado hasta incluir modalidades como el acceso a Internet en casi todos sus aspectos, incluyendo las teleconferencias, transmisión de archivos fotográficos, mp3, videos, etc.

Esto trae consigo innegables ventajas, pues se acelera el ritmo al cual se obtiene la información, facilitando las comunicaciones, reduciendo los tiempos de emisión y respuesta. Sin embargo se convierte en un desafío para los ingenieros del software. Como consecuencia se han creado enfoques disciplinados y metodologías donde se tuvieron en cuenta aspectos específicos de este nuevo medio.

Entre los que se destacan sin duda alguna el diseño gráfico y la organización estructural del contenido.

La aglomeración del gran cúmulo tecnológico que se emplea en la construcción de aplicaciones *web*, ha creado una brecha considerable entre el desarrollo de este tipo de producto y los orientados a entornos de escritorio. Trayendo como consecuencia que se hayan desechado muchas de las buenas prácticas arquitectónicas típicas de estas últimas, afectando directamente el soporte de los productos finales. Principalmente en la liberación de nuevas versiones, implicando la modificación de algunos atributos ausentes o pobremente diseñados. De igual forma el proceso de migración para determinados aplicativos de un entorno a otro, puede llegar a ser traumático y en extremo complejo, pues los conceptos que se manejan difieren en gran medida. Teniendo en cuenta que en el desarrollo de aplicaciones *web* la mayoría de los mecanismos de diseño están enfocados en la comunicación cliente-servidor, así como conectar las *GUI*[1] con la lógica de negocio.

En consecuencia a lo anteriormente expresado, se han desarrollado numerosos *framework* orientados a la construcción de aplicaciones *web*. Muchos de estos con el objetivo de minimizar la complejidad de su construcción. Definiéndose para ello una serie de mecanismos que han evoluciona en la medida que las tecnologías lo han permitido, tal es el caso de los motores de plantillas, el *Ajax*, etc. Pese a todos estos esfuerzos, aún persiste la gran divergencia entre la confección de aplicaciones orientadas a diferentes entornos. Por otra parte la tendencia de orientar los productos *web* a sistemas de gestión tampoco ha favorecido en este sentido y en aras de homogenizar las tecnologías que en estas intervienen, en ocasiones han exacerbado los costos por concepto de rendimiento en los productos finales.

Ksike surge como necesidad de obtener una plataforma para la construcción de aplicaciones orientadas a entornos web, enfocado en maximizar el rendimiento y agilizar el proceso de desarrollo. Sin embargo esto no debería de ir en decremento de la solidez del mismo, permitiendo soportar cualquier tipo de sistema indiferentemente de la lógica de negocio que este necesite gestionar. Otros de los elementos a tener en cuenta es que es sumamente extensible y fomenta la consistencia de código entre los desarrolladores. Partiendo de la idea de que potencia el desarrollo de productos de *software* basados en el paradigma de *OOP*[2] y por ende todas las ventajas que este provee.

Su objetivo fundamental consiste en potenciar la construcción ágil de *software*, permitiéndole al desarrollador concentrarse en la lógica de la aplicación y abstraerse de la comunicación cliente-servidor. Asumiendo como proyección la ruptura de las barreas entre el desarrollo de aplicaciones orientadas a entornos *web* y de escritorio.

Este proyecto se inició a mediados del mes de julio del 2010. Constituye el reflejo del interés profesional de sus integrantes en aras de lograr el objetivo propuesto. Inicialmente fue concebido como un método autodidacta para la comprensión a fondo del funcionamiento de otros *framework*, pero en la medida que se avanzó en la investigación se fueron incorporando nuevos conceptos, reorientándose de esta forma la meta a seguir.

En la tabla que aparece reflejada a continuación, están plasmados los fundadores del mismo. Se considera relevante destacar que el número de personas implicadas directamente en la construcción de esta tecnología no debe ser relevante. Teniendo en cuenta una serie de mecanismos que esta provee, potenciando en gran medida el desarrollo colaborativo del mismo, sin necesidad de tener grandes conocimientos sobre su arquitectura, ni modificar el propio núcleo.

[1] **GUI** acrónimo de Interfaz Gráfica de Usuario
2 OOP acrónimo de Programación Orientada a Objeto

No	Nombres y Apellidos	Rol o Responsabilidad	Entidad	País
1.	Antonio Membrides Espinosa	Arquitecto, Diseñador, Analista, Programador, Revisor técnico.	UCID	Cuba
2.	Hermes Lázaro Herrera Martínez	Diseñador, Programador, Revisor técnico.	UCID	Cuba
3.	Rolando Toledo Fernández	Revisor técnico.	UCID	Cuba
4.	Armando Batista Piñeda	Diseñador gráfico.	UCI	Cuba
5.	Joan Pablo Jiménez Milian	Revisor técnico.	DTS	Cuba

Tabla 1: *Listado de los autores de Ksike*

La *Tabla 2* describe la relación para aquellas herramientas, tecnologías y componentes que intervienen de una forma u otra en este producto. En la misma serán excluidos los elementos que se utilizaron en la fase de construcción, pues no tienen relevancia en el resultado final, dígase: *DotProject*, *Subversion*, *IDE*, editores de imágenes y sonido, etc. Por tanto comprende aquellos recursos que están incluidos o son distribuidos con este, teniéndose en cuenta para definir dichas clasificaciones los siguientes elementos:

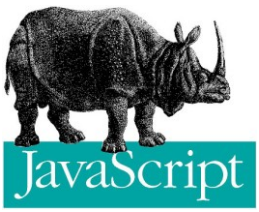
- ✓ **Incluida**: Se refiere a si el componente forma parte del producto final, sea en forma de código fuente, como librería de enlace dinámico.
- ✓ **Distribuida**: Se refiere a los componentes de software cuyo desarrollo es independiente de la plataforma y deben ser distribuidos de conjunto con la misma, excepto por los lenguajes de programación y sus intérpretes.

No	Nombre	versión	Descripción	Incluida	Distribuida
1.	Javascript	1.6.0	Lenguaje de programación interpretado por los navegadores web.	x	
2.	PHP	5.3.2	Lenguaje de programación	x	
3.	ConfigManager	1.0.0	Driver para la administración de ficheros de configuración	x	x
4.	OutManager	1.0.0	Driver para la administración de ficheros de configuración de los formatos de las respuestas.	x	x
5.	Doctrine	1.0.0	Driver para la administración del ORM Doctrine v1.2.2	x	
6.	PgSQL	1.0.0	Driver para la administración de bases de datos orientadas al SGBD PosgresSQL.	x	
7.	File	1.0.0	Driver para la administración de ficheros de texto.	x	
8.	Bhike	0.1 alfa	Entorno gráfico para desarrollo de aplicaciones web sobre la tecnología Ksike.		

Tabla 2: *Componentes del framework Ksike*

Aquellos que no aparecen en ninguna de las clasificaciones anteriormente explicadas, son comprendidos como valores agregados del marco de trabajo, muy similar a la clasificación *incluida*, pero la dependencia del mismo para con estos tiende a ser prácticamente nula.

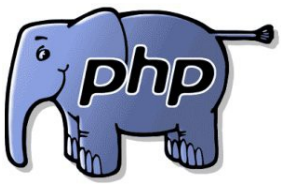
Para una mayor comprensión se expondrán los elementos que conforman el soporte tecnológico del *framework*:



Lenguaje de *script* basado en prototipos, con entrada dinámica y funciones de primera clase. Permitiendo el desarrollo de interfaces de usuario mejoradas y páginas web dinámicas, a través del DOM[3]. El cual es esencialmente una interfaz de programación de aplicaciones que proporciona un conjunto estándar de objetos para representar documentos HTML y XML, un modelo estándar sobre cómo pueden combinarse dichos objetos, y una interfaz estándar para acceder a ellos y manipularlos. En efecto, el DOM es una API para acceder, añadir y cambiar dinámicamente contenido estructurado en documentos con lenguajes como *ECMAScript*. (Javier Eguíluz Pérez)

Es una técnica de desarrollo web para crear aplicaciones interactivas o RIA[4]. Estas aplicaciones se ejecutan en el cliente, es decir, en el navegador de los usuarios mientras se mantiene la comunicación asíncrona con el servidor en segundo plano. De esta forma es posible realizar cambios sobre las páginas sin necesidad de recargarlas, lo que significa aumentar la interactividad, velocidad y usabilidad en las aplicaciones. Ajax [5]es una tecnología asíncrona, en el sentido de que los datos adicionales se requieren al servidor y se cargan en segundo plano sin interferir con la visualización ni el comportamiento de la página. JavaScript es el lenguaje interpretado en el que normalmente se efectúan las funciones de llamada de Ajax mientras que el acceso a los datos se realiza mediante XMLHttpRequest, objeto disponible en los navegadores actuales. En cualquier caso, no es necesario que el contenido asíncrono esté formateado en XML. (Mellado Domínguez, 2008)

Ajax es una técnica válida para múltiples plataformas y utilizable en muchos sistemas operativos y navegadores dado que está basado en estándares abiertos como JavaScript y DOM.



Lenguaje de programación interpretado, diseñado originalmente para la creación de páginas web dinámicas. Puede ser desplegado en la mayoría de los servidores web y en casi todos los sistemas operativos y plataformas sin costo alguno. El gran parecido que posee PHP con los lenguajes más comunes de programación estructurada, como C y Perl, permiten a la mayoría de los programadores crear aplicaciones complejas con una curva de aprendizaje muy corta. También les permite involucrarse con aplicaciones de contenido dinámico sin tener que aprender todo un nuevo grupo de funciones.

Aunque su diseño está orientado a facilitar la creación de página web, es posible crear aplicaciones orientadas a entornos de escritorio, utilizando la extensión *PHP-Qt* o *PHP-GTK*. También puede ser usado desde la línea de comandos, de la misma forma como Perl o Python pueden hacerlo, a esta versión de *PHP* se le conoce como *PHP CLI*[6].

Otro elemento a tener en cuenta es que permite la conexión a diferentes tipos de servidores de bases de datos tales como *MySQL*, *PostgresSQL*, *Oracle*, *ODBC*, *DB2*, *Microsoft SQL Server*, *Firebird* y *SQLite*. En el framework se brinda de forma nativa recursos para la gestión y administración sobre *PostgresSQL*. (PHP Group)

[3] **DOM** acrónimo de Document Object Model
[4] **RIA** acrónimo de Rich Internet Applications
[5] **Ajax** acrónimo de Asynchronous JavaScript And XML
[6] **CLI** acrónimo de Command Line Interface.

PostgreSQL es un sistema de gestión de base de datos relacional orientada a objetos y libre, publicado bajo la licencia *BSD* y desarrollado por una comunidad denominada *PGDG*[7]. El mismo provee recursos como la alta concurrencia mediante un sistema denominado *MVCC*[8], permite que mientras un proceso escribe en una tabla, otros accedan a la misma tabla sin necesidad de bloqueos. Cada usuario obtiene una visión consistente de lo último a lo que se le hizo commit. Esta estrategia es superior al uso de bloqueos por tabla o por filas común en otras bases de datos, eliminando la necesidad del uso de bloqueos explícitos. Permite integrase en un sistema distribuido formado por varios recursos gestionados por un servidor de aplicaciones donde el éxito de la transacción goblal es el resultado del éxito de las transacciones locales. (PostgreSQL Global Development Group)

1. Arquitectura

La arquitectura propuesta se enfoca en la descomposición del diseño en componentes funcionales o lógicos que expongan interfaces de comunicación bien definidas. Esto provee un nivel de abstracción mayor que los principios de orientación por objetos, evitando desviar el centro de atención en asuntos específicos como los protocolos de comunicación y la forma como se comparte el estado.

El estilo de arquitectura basado en componentes tiene las siguientes características:

- ✓ Es un estilo de diseño para aplicaciones compuestas de componentes individuales.
- ✓ Pone énfasis en la descomposición del sistema en componentes lógicos o funcionales que tienen interfaces bien definidas.
- ✓ Define una aproximación de diseño que usa componentes discretos, los que se comunican a través de interfaces que contienen métodos, eventos y propiedades.

1.1 Principios Fundamentales

Teniendo en cuenta que un componente es un objeto de software específicamente diseñado para cumplir con cierto propósito, es de vital importancia que en el diseño de estos, estén implícitos una serie de principios que son fundamentales para su correcto funcionamiento:

- ✓ **Reusable**: Los componentes son usualmente diseñados para ser utilizados en escenarios diferentes por diversas aplicaciones, sin embargo, algunos pueden ser diseñados para tareas específicas permitiéndole ser compartido entre varios sistemas.
- ✓ **Sin contexto específico**: Los componentes son diseñados para operar en diferentes ambientes y contextos. Información específica como el estado de los datos deben ser pasadas al componente en vez de incluirlos o permitir al componente acceder a ellos.
- ✓ **Extensible**: Un componente puede ser extendido desde un componente existente para crear un nuevo comportamiento.
- ✓ **Encapsulado**: Los componentes exponen interfaces que permiten al programa usar su funcionalidad. Sin revelar detalles internos, detalles del proceso o estado.
- ✓ **Independiente**: Los Componentes están diseñados para tener una dependencia mínima de otros componentes. Por lo tanto los componentes pueden ser instalados en el ambiente adecuado sin afectar otros componentes o sistemas.

[7] **PGDG** acrónimo de PostgreSQL Global Development Group.
[8] **MVCC** acrónimo de Acceso Concurrente Multiversión

1.2 Beneficios

Principales beneficios del estilo de arquitectura basado en componentes:

- ✓ **Facilidad de instalación:** Cuando una nueva versión esté disponible, usted podrá reemplazar la versión existente sin impacto en otros componentes o el sistema como un todo.
- ✓ **Costos reducidos:** El uso de componentes de terceros permite distribuir el costo del desarrollo y del mantenimiento.
- ✓ **Facilidad de desarrollo:** Los componentes implementan un interface bien definida para proveer la funcionalidad definida permitiendo el desarrollo sin impactar otras partes del sistema.
- ✓ **Reusable:** El uso de componentes reutilizables significa que ellos pueden ser usados para distribuir el desarrollo y el mantenimiento entre múltiples aplicaciones y sistemas.
- ✓ **Mitigación de complejidad técnica:** Los componentes mitigan la complejidad por medio del uso de contenedores de componentes y sus servicios. Ejemplos de servicios de componentes incluyen activación de componentes, gestión de la vida de los componentes, gestión de colas de mensajes para métodos del componente y transacciones.

1.3 Etimología

Este acápite esta dedicado a aquellos que son un poco más curiosos y llegan a interesarse por la procedencia etimológica de la marca para este producto, dicho elemento dentro de la ingeniería del *software* está estrechamente relacionado con la arquitectura de la información. Es importante tener presente que el mercado está constantemente bombardeado por propuestas atinentes a las distintas alternativas de satisfacción ante cada requerimiento por parte del individuo.

Un buen nombre se convierte en el pasaporte a la solución de la carencia y en dependecia de lo que este transmita constituye una ventaja competitiva respecto a otros productos alternativos. Es por ello que uno de los elementos importantes a tener en cuenta en la conformación del identificador para el producto radicó en la posibilidad del fácil memorización por parte del consumidor y al mismo tiempo la capacidad de haberle despertado una necesidad o movilización del interesado hacia la oferta.

Como es conocido la marca es un elemento que le atribuye identidad a determinado producto, en esta deben de estar presentes los siguientes elementos:

- **Nombre o Fonotipo**: Constituido por la parte de la marca que se puede pronunciar. Es la identidad verbal de la marca.
- **Logotipo**: Es la representación gráfica del nombre, la grafía propia con la que éste se escribe.
- **Isotipo**: Es la representación gráfica de un objeto, que es un signo- icono
- **Gama Cromática o Cromatismo**: Es empleo y distribución de los colores.
- **Diseño Gráfico o Grafismo**: Son los dibujos, ilustraciones, no pronunciables, que forma parte de la identidad visual de marca

Con el logotipo de este producto se intentó obtener un doble sentido en el mensage a transmitir y que este a su vez fuese de forma subliminal. A continuación se muestran algunos de los prototipos iniciales, así como la descripción del mensaje que se desea transmitir.

Se intenta inducir un sentido de pertenencia manteniendo algunos elementos tradicionales de la idiosincrasia de los países caribeños. Partiendo de la propia

fonética del nombre que hace referencia al idioma Taíno, perteneciente a culturas indígenas que habitaron estas zonas geográficas, principalmente la isla de Cuba. Su equivalente en idioma Español es cacique, especie de líder, gobernante o dictador.

Pretendiendo hacer ver a esta tecnología como impulsor o promotor de una nueva filosofía en el desarrollo de sistemas informáticos, teniendo en cuenta que provee una metáfora de desarrollo para aplicaciones *web* como si se construyera una aplicación de escritorio. De ahí que en el logo se refleje el rostro de un nativo o en el que a parece a continuación que sería con una pluma, aunque este último queda descartado por su similitud con el logotipo del servidor de aplicaciones web Apache.

Por otra parte las siglas en idioma inglés transmiten la idea de en qué consiste dicho framework, su descomposición sería la siguiente:

- ✓ **K** como contracción de *Kit* cuyo significado en el idioma Castellano seria: paquete, herramientas.
- ✓ **Sike** considerado como una forma arcaica de decir *gutter*, *ditch*, *small stream* cuyo significado en castellano corresponde a: canal, cuneta, arroyo, desagüe, zanja, zafarse de, deshacerse por, sacudirse a, abandonar a alguien, pequeño arroyo. También es considerado una forma antigua para referirse a términos como *sigh* o *sob* que significan suspiro, susurro, sentimental.

Por tanto la complementación de sus siglas generaría ideas tales como: **canal de desarrollo**, pequeño paquete de herramientas. Incluso algunas muy sugerentes y jocosas como: despojo tecnológico y paquete de herramientas sentimentales o románticas, en fin hay para todos los gustos, aunque optamos por la idea que aparece resaltada.

Independientemente del estudio realizado cabe resaltar que el acto de confeccionar la marca para determinado proyecto reviste una dificultad análoga a la disyuntiva que sufren los padres ante similar situación. Sin embargo el efecto que vierte el mismo es de vital importancia, pues como bien se había expresado anteriormente, el nombre es la representación verbal que asigna personalidad y singularidad al negocio o producto, conformando la llave para la entrada del proceso de selección del mismo ante la necesidad del cliente.

2. Estructura

Este acápite recoge toda la información asociada a la estructura física o esquema organizativo de los proyectos desarrollados sobre la tecnología *Ksike*. Como es evidente es de vital importancia que se comprenda, pues este apartado constituye el pilar fundamental para la asimilación de los sucesores.

2.1 Proyecto

Comúnmente se entiende por proyecto la planificación que consiste en un conjunto de actividades que se encuentran interrelacionadas y coordinadas. La razón del mismo consiste en alcanzar objetivos específicos dentro de los límites que imponen un presupuesto con la calidad requerida en un lapso de tiempo previamente definido. Empleándose para su gestión la aplicación de conocimientos, habilidades, herramientas y técnicas en función de satisfacer los requisitos del mismo, el cual apunta a lograr un resultado único, surgido como respuesta a una necesidad.

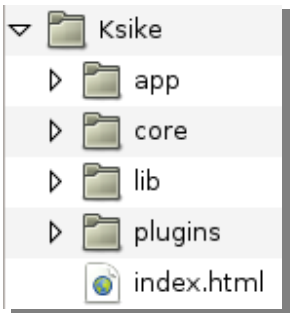


Figura 2: *Estructura de un proyecto simple*

La idea anterior hace referencia al concepto de proyecto, pero enmarcado en el área de la gestión. Pese a

las similitudes que puedan tener, existen elementos que marcan de una forma u otra la diferencia. En el contexto de la construcción de aplicaciones informáticas empleando determinada tecnología, está encaminado describir la composición física basado en la alineación de las distintas partes que la constituyen. Donde intervienen ciertas particularidades tales como la estructura jerárquica de sus entes, tipos de archivos, ficheros de configuración, etc. Teniendo en cuenta lo anteriormente expresado se puede asumir que en el *framework Ksike* un proyecto corresponde al esquema organizativo que conforma la infraestructura del producto en su totalidad, identificado dentro de un dominio específico o un negocio en particular, pudiendo existir varios en el contexto de dicha plataforma.

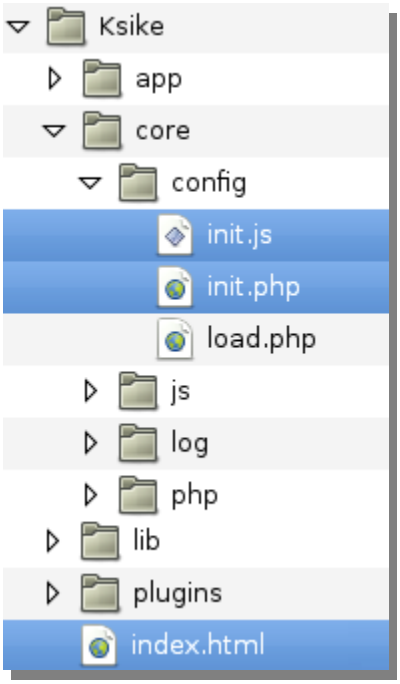


Figura 3: *Ficheros significativos del proyecto*

Los elementos que definen el comportamiento para determinado proyecto son:

- **Init.js:** Fichero de configuración para el *API* de *Ksike,* desarrollado en el lenguaje *Javascript*. Más adelante se procederá a explicar en detalles que debe contener el mismo.
- **Init.php:** Al igual que el anterior se encarga de gestionar los elementos configurativos del *API* desarrollada en lenguaje *php*.
- **Index.html:** Corresponde a la plantilla escrita en lenguaje *html* que debe contener básicamente las librerías que conforman la base del marco de trabajo**.** No obstante es posible desarrollar tantas páginas html se deseen incorporarles los elementos que persistirán durante la ejecución del producto final. Sin embargo a medida en que se avance en la lectura del documento se evidenciará que esta práctica no es la más adecuada en aras de lograr aplicaciones que sean fáciles de mantener en cuanto a soporte se refiere, por tan solo citar un ejemplo.

2.2 Aplicación

Teniendo en cuenta que una aplicación es una clase de programa informático diseñado como herramienta para permitir a los usuarios realizar uno o diversos procesos de negocio. Generalmente suele resultar una solución informática para la automatización de ciertas tareas complicadas. Como pueden ser la contabilidad, la redacción de documentos, o la gestión de un almacén. Por tanto constituye la materialización del proyecto enfocada al prototipo de solución.

Teniendo en cuenta lo anteriormente expresado, no existe conceptualmente un proyecto que no tenga definida su aplicación, conocida como *app* dentro del esquema organizativo que él mismo provee.

Nótese que cada proyecto tiene una única app, sin embargo esta correspondencia no tiene que cumplirse en sentido contrario. Puede que se dé el caso que dos proyectos la compartan, aunque este tipo de situaciones es bastante difícil que se den, a no ser en proyectos de gran similitud. Este tema será abordado con mayor profundidad en acápites posteriores cuando se explique el mecanismo de multiproyecto que provee esta tecnología.

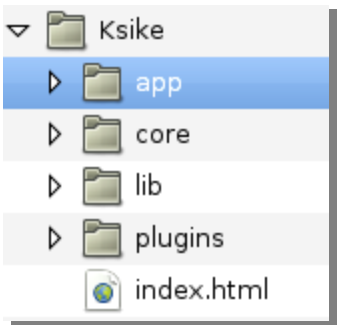


Figura 4: *La aplicación como módulo.*

El desarrollo de productos sobre la plataforma *Ksike* se sustenta fundamental en el módulo principal que se encuentra ubicado en el directorio **"app"**, esto significa que no es necesario la existencia de *plugins*

para que una aplicación pueda ser publicada, sin embargo la construcción de la misma separa por módulos potencia la reutilización de los mismos así como la escalabilidad del producto final.

2.3 Plugin

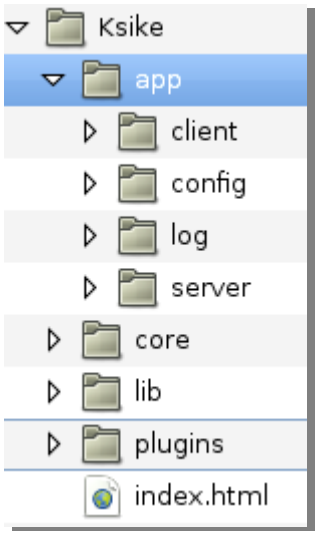


Figura 5: *Estructura del app.*

Los términos de *Plugins* o Módulos hacen referencias a componentes de software. El cual constituye un elemento autónomo, escrito dentro de un contexto que permita que sus funcionalidades sean útiles en la creación de distintas piezas de software. Permitiendo que sea desarrollado de manera independiente y ser modificado internamente ya sea por concepto de diseño o adición de nuevas funcionalidades lógicas, sin afectar significativamente el resto del sistema.

En el proceso de construcción de una pieza de software con componentes existentes, da origen al principio de reutilización del software, mediante el cual se promueve que los componentes sean implementados de una forma que permita su utilización funcional sobre diferentes sistemas en el futuro.

Un componente de software como bien se había expresado anteriormente se define típicamente como algo que puede ser utilizado como una caja negra, en donde se tiene de manera externa una especificación general, la cual es independiente de la especificación interna. De esta definición se presentan tres conceptos ligados con la definición de un componente:

- ✓ Interior del componente: Es una pieza de software que cumple con un conjunto de propiedades y que se encuentra conformada como un artefacto del cual se espera que sea reutilizable.
- ✓ Exterior del componente: Es una interface que cumple con un conjunto de propiedades y provee un servicio a los agentes humanos u otros artefactos de software.
- ✓ Relación interior-exterior: Es la que define el proceso de relación entre el interior y exterior el componente, a través de conceptos como especificación, implementación y encapsulación.

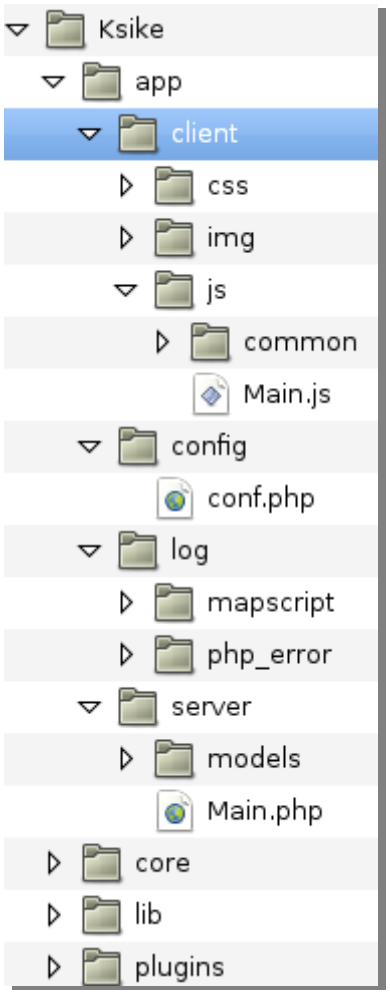


Figura 6: *Componentes del app*

Como bien se puede apreciar el *framework* consta de cuatro directorios principales, de ellos solo deberán ser modificados por los desarrolladores **"app"** y **"plugins"** en aras de garantizar la integridad y el correcto funcionamiento del mismo.

Elementos que conforman cualquiera de los módulos desarrollados para la plataforma *Ksike,* esto incluye también al *app* pues que este constituye el módulo principal.

- ✓ **Client**: Directorio en que se encuentran todo los elementos que son interpretados por el navegador en la PC cliente. Entiéndase por archivos en formato *js*, *css*, *xml*, *json*, imágenes, etc.
- ✓ **Config**: Al igual que en el core en este directorio se definen todos los elementos que tienen carácter variable y pueden ser cambiados en tiempo de ejecución sin necesidad de afectar el código fuente

del producto final. Entiéndase por configuraciones de servidores, etc. La plataforma por defecto utiliza este recurso en formato *php*, teniendo en cuenta las ventajas en cuanto a rendimiento, cediendo el grueso del procesamiento al propio intérprete de *php*.

- ✓ **Server**: En este directorio se encuentran los script que deben ser ejecutados en el servidor de aplicaciones web. Tales como ficheros en formato *php, so, dll, etc.*

- ✓ **Log**: Este directorio está reservado para el almacenamiento de las trazas del sistema. Incluye listado de los errores del propio lenguaje *php* organizados por las categorías definidas por su intérprete tales como *WARNING, NOTICE, DEPRECATED, STRICT* los cuales estarían ubicados en el directorio *php_error*. Cada traza es generada independiente para cada uno de los módulos del proyecto en ejecución, incluyendo la aplicación. Además se pueden incluir otros elementos que se consideren necesarios tener en cuenta para la gestión de trazas, claro está que esto se encuentra sujeto a la lógica del negocio que se desea automatizar.

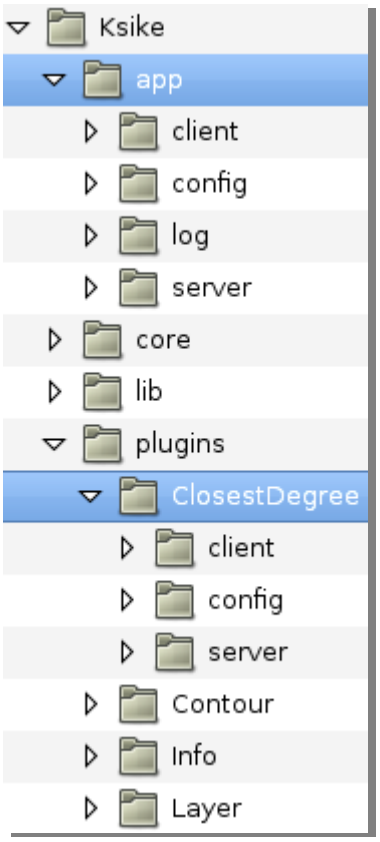


Figura 7: *Similitud entre el app y los plugins.*

Obsérvese la similitud en el directorio de **"*app*"** y el del *plugin* **"*ClosestDegree*"**, evidentemente no es casualidad, el objetivo es que sea lo menos traumático posible asimilar la teoría del desarrollo orientado a componentes, manteniendo para ello la misma estructura que se emplea en la aplicación en cada uno de los módulos o *plugins* que se desarrolle sobre esta plataforma.

A diferencia de la aplicación los *plugin* no gestionan **"log"**, una vez que estos se incorporan a la plataforma deben funcionar de forma autónoma y en aras de ganar en cuanto a legibilidad y comprensión del sistema de trazas o log se decidió delegar los mismos al módulo principal o aplicación. En posteriores versiones se liberará un sistema para administración de los mismos.

A pesar de que no se ilustra en el directorio del servidor de cualquiera de los módulos del proyecto se pueden definir los directorios **"include",** estos constituyen un recurso muy útil permitiendo sin importar la estructura organizativa que se encuentre en su interior, importar archivos escritos en lenguaje *php* a la aplicación. Sobre este recurso habrá un acápite dedicado que entraría más en detalles.

2.4 Multiproyecto

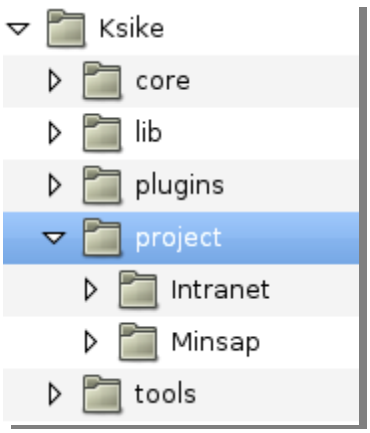


Figura 8: *Ksike multiproyecto*

Como bien se ilustraba anteriormente se puede liberar un producto que contenga los directorios principales en la raíz del directorio del proyecto, entiéndase por directorios principales: *core, lib, plugins, app*. Sin embargo esto no quiere decir que la tecnología limite a los desarrolladores en este sentido.

Este marco de trabajo al igual que muchos otros da la posibilidad de compartir para diferentes proyectos un *core* común, potenciándose de esta forma la facilidad y agilidad por concepto de soporte para productos informáticos.

También en este sentido se gana en cuanto a organización, estandarización tecnológica e incluso consumo óptimo de los recursos de hardware por concepto de buen aprovechamiento del espacio en discos duros, que si bien hoy en día no constituyen un problema tampoco es beneficioso utilizarlo incorrectamente.

Como bien se puede apreciar en la figura número 8, se define un directorio nombrado *project* en el cual conviven dos proyectos: *Intranet* y *Minsap*. Estos comparten los directorios principales solo deben redefinir el directorio *core/conf* en el cual a través de ficheros de configuración establecen las políticas de acceso y que será utilizado del exterior o no.

Nótese que el directorio *project* no es de carácter obligatorio, sino es una propuesta organizativa. Pues es realmente a través de los ficheros de configuración que definen las rutas de acceso. Incluso cada uno de los directorios pueden encontrarse en disimiles directorios. Sin embargo tampoco es recomendable como una buena práctica para el desarrollo de aplicaciones, por las razones anteriormente expuestas.

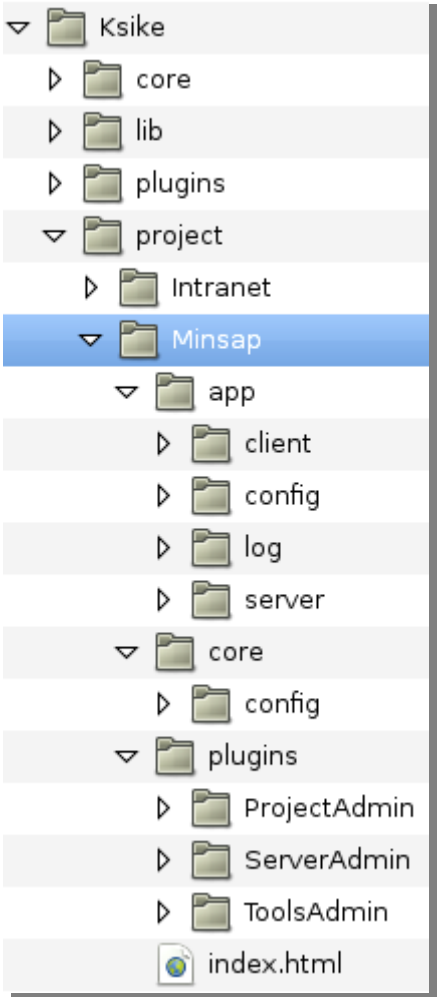


Figura 9: *Ejemplo de proyectos*

2.4 Core

El término *core* del idioma inglés hace referencia al núcleo o base y es precisamente lo que constituye para el framework *Ksike*. Esta plataforma está compuesta por dos APS una enfocada a la gestión en el cliente y la otra en el servidor. Según la filosofía que este propone podrían emplearse por separado. Lo que implicaría un uso parcial de esta tecnología en dependencia de las necesidades, simplemente manteniendo las entradas y salidas de cada una de ellas, aunque se recomienda que se utilice íntegramente.

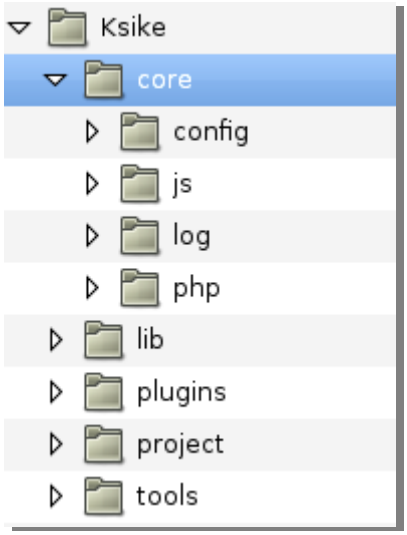


Figura 10: *Núcleo de Ksike*

Como se había expresado en acápites anteriores esta plataforma de desarrollo está orientada a la construcción de aplicaciones web basadas en el modelo Cliente-Servidor**.** Es por ello que el "**core**" se divide fundamentalmente en "**js**" para la gestión sobre el navegador y "**php**" para el control desde el servidor de aplicaciones web. Por otra parte se dedica el directorio "**config**" para los temas relacionados con la configuración del *framework*, donde se encuentran algunos elementos modificables tales como los módulos que serán cargados al inicio de la aplicación tanto para el cliente como para el servidor. Se aconseja que solo sea modificado por aquellos desarrolladores con conocimientos avanzados en el funcionamiento de la plataforma.

2.5 Lib

Ksike no brinda recursos para la generación de interfaces gráficas de usuario, teniendo en cuenta que en la actualidad existen muchos *framework* dirigidos principalmente al lenguaje *Javascript* tales como *ExtJs, JQuery, Yui, OpenLayer*. Por otra parte en un proyecto no solo se emplean para aquellas librerías que se encargan de la gestión de GUI en el cliente o navegador web sino que se pueden emplear otras

tales como *Doctrine, File, PgSQL, ConfigManager*, etc.

Por tal motivo se definió **"lib"** como el directorio donde deberán ser depositadas todas aquellas librerías cuyo desarrollo sea independiente a la plataforma y que por consiguiente puedan ser consideradas externas a la misma.

En el propio directorio **"lib"** se provee la librería *PgSQL* para la gestión de bases de datos soportadas sobre *PostgresSQL, ConfigManager* para la administración de ficheros de configuración en distintos formatos y *File* para la gestión de datos sobre ficheros en modo texto, las mismas se distribuyen con la plataforma, pese a que su desarrollo es independiente, tal y como se expresaba anteriormente.

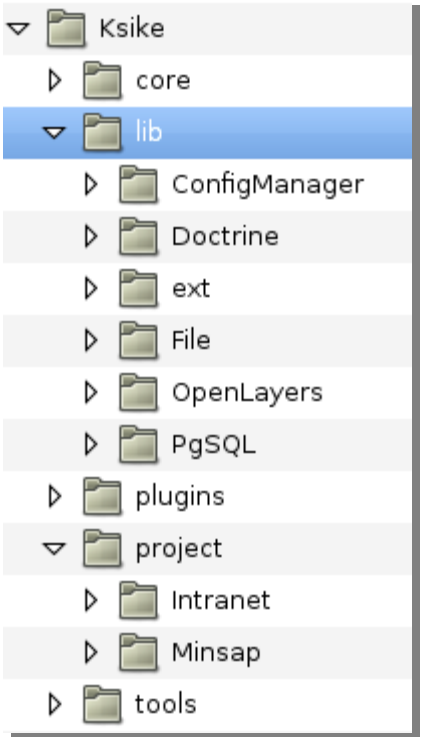


Figura 11: *Recursos externos*

2.6 Tools

El directorio *tools* no es de carácter obligatorio más bien es considerado como un conjunto de utilidades que se le entregan a los desarrolladores que decidan emprender la epopeya de la construcción de aplicaciones *web* sobre el *framework Ksike*. En aras de facilitar al trabajador y agilizar en alguna medida el proceso de desarrollo.

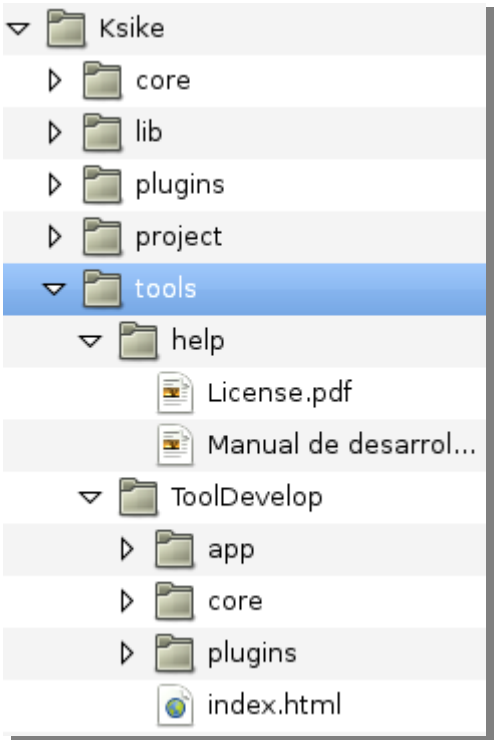


Figura 12: *Complementos de Ksike*

En el subdirectorio *help* podrán encontrar todos los materiales de ayuda que el equipo de desarrollo sea capaz de liberar en cada una de las versiones, contribuyendo a minimizar la curva de aprendizaje del mismo.

Por otra parte la herramienta *ToolDevelop* actualmente en desarrollo constituye un *demo* de empleo del *framework,* teniendo en cuenta que está desarrollada sobre la propia plataforma y cumple con lo que se mencionaba anteriormente sobre que los proyectos no tienen por que estar en un directorio obligatorio para que los mismos funcionen correctamente como es usual en otros marcos de trabajo.

En el caso de esta última herramienta su culminación será de vital importancia por el impacto que revertiría en el proceso de desarrollo de aplicaciones sobre *Ksike*. Teniendo en cuenta que tiene como objetivo gestionar de forma grafica todo el proceso de generación y administración tanto de proyectos como módulos por cada uno de estos.

3. API Cliente

Este apartado comprende todo los mecanismos de diseño relacionados con la capa de interfaz principalmente, todo el código está basado en el lenguaje *Javascript,* dándole un enfoque orientado a objetos. Un elemento importante a destacar es que como se venía explicando en acápites anteriores la gran mayoría de las funcionalidades brindadas están enfocadas a la comunicación cliente servidor y no a la generación de GUI.

3.1 STD

En la plataforma *Ksike* se define un espacio de nombres global denominado *std*, en el cual se incluyen las definiciones de todas las funciones y clases que conforman el módulo cliente. Entre los recursos nativos de este se provee la función *include* permitiendo de esta forma adicional en tiempo de ejecución tanto ficheros escritos en lenguaje *Javascript* como hojas de estilos a nuestras aplicaciones. También es posible cambiar la denominación del mismo en función de las necesidades de los desarrolladores, para lograr tal objetivo debe ser cambiado directamente en el fichero de configuración del cliente para determinado proyecto, remítase al acápite 3.8 para mayor comprensión.

3.1.1 Include

Esta es la funcionalidad que permite de forma dinámica y de carácter asincrónico, incluir tanto fichero de extensión *js* como *css* al *DOM*.

Sintaxis:

std.include(file, handle);

- **file**: Fichero ***js|css*** a incluir.
- **handle**: Función que se ejecuta una vez cargado el fichero y es de carácter opcional.

Ejemplo:

1. Incluir la hoja de estilos que lleva por nombre *ext*-all.

```
1. std.include("lib/ext/css/ext-all.css");
```

2. Incluir el archivo que contiene código escrito en el lenguaje Javascript y una vez que concluya su carga en memoria se deberá ejecutar una función anónima con el comportamiento que se desee.

```
2.  std.include("app/client/js/common/viewPort.js", function(){
3.          ...
4.  });
```

3.1.2 Require

Este mecanismo es similar al *Include* con la particularidad de que permite cargar sincrónicamente un listado de ficheros en formatos *js y css*. Es más ventajosa su utilización cuando se requiere cargar un grupo de ficheros que dependen unos de otros y donde el orden de carga es imprescindible.

Sintaxis:

std.require (file, handle);

- **file**: Fichero o listado ordenado de ficheros de extensión ***js|css*** a incluir.
- **handle**: Función que se ejecuta una vez cargado el fichero y es de carácter opcional.

Ejemplos:

1. Incluir la hoja de estilos que lleva por nombre ext-all.

```
5.  std.require("lib/ext/css/ext-all.css");
```

2. Cargar sincrónicamente las librerías (*Router, Factory, Loader*) e instanciar las clases *Router y Loader* una vez finalizada la carga.

```
6.  std.require([
7.          "Router.js",
8.          "Factory.js",
9.          "Loader.js",
10.         ],function(){
11.                 std.Router = new Router();
12.                 std.Loader = new Loader({"libs":std, "mods":mod});
13. });
```

3.1.3 Namespace

Los espacios de nombres son otro de los recursos más utilizados actualmente en el desarrollo de aplicaciones, pues constituyen un conjunto de identificadores en el cual todos son únicos. También puedes ser considerado como un contexto en el que un grupo de uno o más identificadores pueden existir.

Un identificador definido en un espacio de nombres está asociado con ese espacio de nombres. El mismo identificador puede independientemente ser definido en múltiples espacios de nombres, eso es, el sentido asociado con un identificador definido en un espacio de nombres es independiente del mismo identificador declarado en otro espacio de nombres. Los lenguajes que manejan espacio de nombres especifican las reglas que determinan a qué espacio de nombres pertenece una instancia de un identificador.

Sintaxis:

std.$ (ns);

- **ns**: Propiedad de tipo cadena que corresponde al identificador del espacio de nombre.

Ejemplos:

3. Definir la variable *name* en el espacio de nombres *myNs*

```
14. std.$("myNs");
15. myNs.name = "Willson";
```

4. Definir la variable *age* en el espacio de nombres *myNs.person*

```
1.  std.$("myNs.person").age = 122;
2.  var value = myNs.person.age;
```

3.2 FrontController

Uno de los recursos más relevantes que provee esta plataforma es la abstracción a la comunicación ciente servidor fundamentado sobre la tecnología *Ajax*. El *FrontController* es en efecto la librería que gestiona cada uno de los pedidos que se realizan al servidor así como las respuestas del mismo, permitiendo de esta forma que se ejecuten más una acción implementada en el servidor y se le dé respuesta a cada módulo por separado en un pedido unificado.

3.2.1 Send

Al utilizar dicho recurso deja de ser problemático el hecho de enviar estructuras de datos tan complejas como objetos, arreglos sean asociativos o indexados, así como cualquier combinación de ambos. Esto es posible debido a que la propia librería se encarga de la codificación al formato *JSOM*, optimizando por consiguiente cada uno de los pedidos y evitando errores de índole humano en lo que podrían incurrir los propios desarrolladores. Por tanto de la misma forma que se manipulan las estructuras de datos en el cliente son recibidas y gestionadas en cada una de las acciones del servidor a las cuales fueron enviadas.

Sintaxis:

std. FrontController**.send(**params**);**

- **params**: Objeto con toda la información necesaria para realizar una petición a determinada acción en el servidor, el cual deberá presentar la siguiente estructura:
    - **action:** Nombre o identificador de la acción que será ejecutada en el servidor.
    - **controller:** Nombre o identificador del módulo que contiene dicha acción, en caso de omitirse el *FrontController* asumirá que la *action* pertenece al *Main*.
    - **params:** Lista de argumentos en forma de objeto que será recibido en por parámetros en la acción implementada en el servidor.
    - **outFormat:** Identificador del formato para la respuesta que arrojara el servidor, por defecto toma valor *json*.
    - **outOption:** Hace referencia a un valor o listado de estos que sean requeridos por el *outFormat* definido para dicha petición, por defecto toma valor nulo.
    - **outInfo:** Propiedad que define si se le incorpora información adicional a la salida arrojada por el servidor, por defecto toma valor *embebed* permitiendo que se le incorpore datos como módulo y acción ejecutados, en caso de especificar *false* o *void* el valor retornado sería íntegro al retorno de la acción invocada.

Ejemplos:

1. Ejecutar la función *showRecord* del *plugin Distance*, enviado dos variables por parámetro: *name* y *age*.

```
3.  std.FrontController.send({
4.      action: 'showRecord ',
5.      controller: "Distance ",
6.      params: {
7.              name : “Lois”,
```

```
8.              age: 35
9.      }
10. });
```

2. Ejecutar la función *showRecord* de la aplicación *Main*, enviado dos variables por parámetro, en la que *person* se define como un objeto, *months* es una lista de valores numéricos y *enterprise* es una cadena de caracteres. Esta acción debe arrojar una imagen en formato png, sin que se le sea incorporada información adicional, ejemplo:

```
11. std.FrontController.send({
12.             action: 'showRecord',
13.             outFormat: "img",
14.             outOption: "png",
15.             outInfo: false,
16.             params: {
17.                     person: {"name":"Lois", "age":23},
18.                     months: [3,5,1,7,9],
19.                     enterprise: "IBM"
20.             }
21. });
```

3.2.2 GetRequest

Actualmente es muy común en el desarrollo de *GUI* para aplicaciones sobre entornos *Web*, utilizar *framework* desarrollados en el lenguaje *Javascript*. Estos en su gran mayoría traen mecanismos implícitos para llevar a cabo peticiones a páginas servidoras y llenar con los datos provenientes de estas los componentes propios, tales como formularios, tablas, árboles, etc. Con el objetivo de que no se violente la estructura arquitectónica del *framework* y de hacer transparente la comunicación cliente servidor en el empleo de otros mecanismos para comunicación. Se provee de una funcionalidad que permite generar en formato *string* los parámetros necesarios para realizar una petición al controlador frontal del *framework*.

Sintaxis:

std.FrontController.**getRequest** (action, controller, params)

Entradas:

- **action:** Nombre o identificador de la acción que será ejecutada en el servidor.
- **controller:** Nombre o identificador del módulo que contiene dicha acción, en caso de omitirse el *FrontController* asumirá que la *action* pertenece al *Main*.
- **params:** Lista de argumentos en forma de objeto que será recibido en por parámetros en la acción implementada en el servidor.

Salida: Petición en formato *string* correspondiente al GET

Ejemplos:

1. Definición de la propiedad *store* muy utilizada por la mayoría de los componentes del *framework ExtJs*, en el cual se le especifica que debe extraer la información a partir de los resultados que arroje la función *getTemplates* del controlador *ProjectAdmin,* pasando por parámetro la propiedad *id* cuyo valor es equivalente a 1101.

```
var tplStore = new Ext.data.JsonStore({
        autoDestroy: true,
        url: std.FrontController.getRequest("getTemplates", "ProjectAdmin", {id:1101}),
        storeId: 'myStore',
        root: 'data',
        idProperty: 'name',
        fields: ['namet', 'typet', 'datet']
});
```

2. Definición de una instancia de la clase *Layer* utilizada en el *framework OpenLayer* para la creación de mapas. En este caso se le especifica que la información es provista por los resultados arrojados de la función *getMap* del controlador principal o *Main*.

```
var lay = new OpenLayers.Layer.MapServer(
            "lay",
            std.FrontController.getRequest("getMap"),
            {
                        layers: []
            },{
                        isBaseLayer: 1,
                        gutter: 0,
                        buffer: 0,
                        singleTile: true,
                        transitionEffect: 'resize'
            }
);
```

3.3 OOP

Pese a que el *Javascript* no se define como un lenguaje basado en el paradigma de programación orientado a objetos si existen algunos recursos que permiten simularlo y aprovechar de esta forma las potencialidades que este brinda. Es por ello que en la plataforma se desarrollo el módulo ***OOP*** el cual provee de funcionalidades para solventar el problema planteado.

3.3.1 Conceptos fundamentales

La programación orientada a objetos es una forma de programar que trata de encontrar una solución a estos problemas. Introduce nuevos conceptos, que superan y amplían conceptos antiguos ya conocidos. Entre ellos destacan los siguientes:

- ✓ **Clase**: definiciones de las propiedades y comportamiento de un tipo de objeto concreto. La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ellas.
- ✓ **Objeto**: entidad provista de un conjunto de propiedades o atributos y de comportamiento o funcionalidad los mismos que consecuentemente reaccionan a eventos. Se corresponde con los objetos reales del mundo que nos rodea, o a objetos internos del sistema. Es una instancia a una clase.
- ✓ **Método**: Algoritmo asociado a un objeto, cuya ejecución se desencadena tras la recepción de un

"mensaje". Desde el punto de vista del comportamiento, es lo que el objeto puede hacer. Un método puede producir un cambio en las propiedades del objeto, o la generación de un "evento" con un nuevo mensaje para otro objeto del sistema.

- ✓ **Evento**: Es un suceso en el sistema. El sistema maneja el evento enviando el mensaje adecuado al objeto pertinente. También se puede definir como evento, a la reacción que puede desencadenar un objeto, es decir la acción que genera.

- ✓ **Mensaje**: una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.

- ✓ **Propiedad o atributo**: contenedor de un tipo de datos asociados a un objeto, que hace los datos visibles desde fuera del objeto y esto se define como sus características predeterminadas, y cuyo valor puede ser alterado por la ejecución de algún método.

- ✓ **Estado interno**: es una variable que se declara privada, que puede ser únicamente accedida y alterada por un método del objeto, y que se utiliza para indicar distintas situaciones posibles para el objeto. No es visible al programador que maneja una instancia de la clase.

- ✓ **Componentes de un objeto**: atributos, identidad, relaciones y métodos.

- ✓ **Identificación de un objeto**: un objeto se representa por medio de una tabla o entidad que esté compuesta por sus atributos y funciones correspondientes.

- ✓ **Abstracción**: denota las características esenciales de un objeto, donde se capturan sus comportamientos. Cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema sin revelar cómo se implementan estas características. Los procesos, las funciones o los métodos pueden también ser abstraídos y cuando lo están, una variedad de técnicas son requeridas para ampliar una abstracción. El proceso de abstracción permite seleccionar las características relevantes dentro de un conjunto e identificar comportamientos comunes para definir nuevos tipos de entidades en el mundo real. La abstracción es clave en el proceso de análisis y diseño orientado a objetos, ya que mediante ella podemos llegar a armar un conjunto de clases que permitan modelar la realidad o el problema que se quiere atacar.

- ✓ **Encapsulamiento**: Significa reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema. Algunos autores confunden este concepto con el principio de ocultación, principalmente porque se suelen emplear conjuntamente.

- ✓ **Principio de ocultación**: Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una interfaz a otros objetos que específica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas, solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no pueden cambiar el estado interno de un objeto de maneras inesperadas, eliminando efectos secundarios e interacciones inesperadas. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción. La aplicación entera se reduce a un agregado o rompecabezas de objetos.

- ✓ **Polimorfismo**: comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre, al llamarlos por ese nombre se utilizará el comportamiento correspondiente al

objeto que se esté usando. O dicho de otro modo, las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado. Cuando esto ocurre en "tiempo de ejecución", esta última característica se llama asignación tardía o asignación dinámica.

- ✓ **Herencia**: las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir y extender su comportamiento sin tener que volver a implementarlo. Esto suele hacerse habitualmente agrupando los objetos en clases y estas en árboles o enrejados que reflejan un comportamiento común. Cuando un objeto hereda de más de una clase se dice que hay herencia múltiple.

En comparación con un lenguaje imperativo, una "variable", no es más que un contenedor interno del atributo del objeto o de un estado interno, así como la "función" es un procedimiento interno del método del objeto.

3.3.2 Clases

Como bien se había expresado en acápites anteriores cuando se escribe un programa en un lenguaje orientado a objetos, definimos una plantilla o clase que describe las características y el comportamiento de un conjunto de objetos similares. Sin embargo Javascript no es un lenguaje orientado a objetos, son que es basado en prototipos. Un prototipo es un objeto abstracto, capaz de contener otros objetos dentro, los cuales pueden ser distintos tipos de datos: variables, vectores, funciones e inclusive otros grupos de objetos.

Entonces, en vez de programar una clase, para estar orientados a objetos en JS definimos un prototipo y por consiguiente las variables dentro de este serán las propiedades, así como las funciones serán los métodos.

```
43. var className = function(params)
44. {
45.         ...
46. }
```

Obsérvese cómo la declaración de clases en *Javascript* se funciona con la declaración del constructor, si se asocia con la filosofía impuesta por lenguajes de programación como C++, C# o Java por tan solo citar algunos ejemplos, donde el identificador de la clase y el de su propio constructor son iguales, por tanto *className* seria el nombre de la clase, *params* constituye la lista de parámetros que requiere el constructor de la misma, de igual forma todo lo que se encuentre dentro de las llaves se ejecutara en el momento que se instancie la misma.

Otro elemento importante a tener en cuenta son los conocidos niveles de accesibilidad para cada uno los recursos de la clase, sean propiedades o funcionalidades. En lenguajes como los mencionados anteriormente se apreciaban tres clasificaciones diferentes *public, protected* y *prívate*. En Javascript existen aunque no de manera explícita. Si se desea declarar un recurso de carácter privado basta con declararlo utilizando el operador ***var.***

```
1.  var className = function(params)
2.  {
3.      var propertyPrivate = arguments[0];
4.      var functionPrivate = function(params)
5.      {
6.              …
7.      }
8.  }
```

Por el contrario si se requiere de un nivel de acceso público entonces se deberá utilizar el operador ***this***, la única limitante es que dicho operador se ve afectado por el ámbito de declaración, esto implica que se utiliza dentro de una función propia de la clase entonces no se estaría haciendo referencia a la misma sino a la función miembro.

```
1.  var className = function(params)
2.  {
3.      var propertyPrivate  = arguments[0];
4.      this.propertyPublic  = arguments[1];
5.      this.functionPublic  = function(params)
6.      {
7.              var tmp = this.propertyPublic;
8.              …
9.      }
10. }
```

Para solucionar el caso anterior se define como propiedad privada a la clase el propio ***this*** denominándola ***_this*** tal y como se muestra en el ejemplo a continuación.

```
1.   var className = function(params)
2.   {
3.       var _this = this;
4.       var propertyPrivate  = arguments[0];
5.       this.propertyPublic  = arguments[1];
6.       this.functionPublic  = function(params)
7.       {
8.              var tmp = _this.propertyPublic ;
9.              …
10.      }
11.  }
12.
13.   className.prototype.dynamicFunction = function()
14.   {
15.             var tmp = this.propertyPublic;
16.             …
17.    }
```

Otro problema que podría acarrear esta declaración es en caso de adicionar recurso fuera del constructor de la clase utilizando el recurso ***prototype,*** entonces al igual que en el caso anterior se perdería la referencia al ***this***. Para solucionarlo se deben declarar las propiedades públicas a través del ***prototype***.

```
1.  var className = function(params)
2.  {
3.      var _this = this;
4.      var _that = className.prototype;
5.
6.      var propertyPrivate   = arguments[0];
7.      _that.propertyPublic  = arguments[1];
8.      _that.functionPublic  = function(params)
9.      {
10.         var tmp = _this.propertyPublic;
11.         …
12.     }
13.
14.     className.prototype.dynamicFunction = function()
15.     {
16.         var _that = className.prototype;
17.         var tmp = _that.propertyPublic;
18.         …
19.     }
20. }
```

3.3.3 Objetos

Una instancia u objeto de una clase es una representación concreta y específica de una clase y que reside en la memoria del ordenador. A continuación se presenta la sintaxis de su utilización:

Sintaxis:

var objName = **new** className**(**params**);**

- **objName:** Nombre que identificará a la instancia.
- **className:** Nombre que identifica a la clase o plantilla.
- **params:** Listado de argumentos externos a la clase y que son utilizados en el constructor de la misma.

En este caso el operador ***new*** reserva espacio en memoria para una nueva instancia de tipo *className* y devuelve su referencia la cual queda almacenada en una variable *objName*, que no es más que una ubicación utilizada para almacenar y manipular valores por su nombre.

3.3.4 Initialice

Teniendo en cuenta que *Javascript* no es orientado a objeto y no existe el concepto ***class,*** prácticamente no es posible inicializar los parámetros de la clase base una vez que se pone en práctica el recurso de herencia, puesto que el mismo se realiza fuera de la declaración del constructor de la hija. Es por ello que la plataforma brinda un mecanismo que permite solventar dicha problemática.

Sintaxis:

std.OOP.**initialice** (objChild, parent, params);

- **objChild**: Instancia de la clase que adoptará como propias todas las propiedades y funcionalidades publicas de la clase *parent*.
- **parent:** Clase en la que se definen un conjunto de propiedades o funcionalidades que pueden ser reutilizadas por otras teniendo en cuenta su marcado carácter generalizador.
- **params:** Lista de parámetros en caso de ser necesarios para el constructor de la clase antecesora.

Ejemplo:

1. En el siguiente ejemplo se procede a inicializar el constructor de la clase base en la propia declaración de la hija.

```
21. var Animal = function(name, age)
22. {
23.         this.name = name;
24.         this.age  = age;
25.         …
26. }
27.
28. var Cat = function(place, name, age)
29.  {
30.         std.OOP.initialice (this, Feline, [name, age]);
31.         this.place = place;
32.         …
33. }
34.
35. std.OOP.extend(Cat, Animal);
36.
37. var misu = new Cat (“house”, “misu”, 5);
```

3.3.5 Extend

A través de este recurso es posible apropiarse de todos aquellos atributos y funcionalidades de carácter público definidos en la clase padre. Solo es posible heredar de una sola clase y automáticamente se crea en la hija una propiedad denominada *parent* que evita que se pierda el comportamiento del padre por concepto de redefinición de funciones.

Sintaxis:

std.OOP.**extend**(child, parent);

- **child**: Clase que adoptará como propio todas las propiedades y funcionalidades públicas de la clase *parent*.
- **parent:** Clase en la que se definen un conjunto de propiedades o funcionalidades que pueden ser reutilizadas por otras teniendo en cuenta su marcado carácter generalizador.

Ejemplo:

1. Crear una clase *Cat* que extienda su comportamiento a partir de las funcionalidades definidas en *Animal*:

```
38. std.OOP.extend(Cat, Animal);
```

2. Redefinir la función *getName* de la clase *Animal* en la clase *Cat* manteniendo parte del comportamiento del padre.

```
39. var _that = Cat.prototype;
40. _that.getName = function()
41. {
42.     var p = _that.parent.getName();
43.     return p + "Cats";
44. }
```

3.3.6 Implement

Este recurso permite manejar los conceptos de interfaces muy utilizados en lenguajes de programación como C# y Java. El objetivo principal de la misma consiste en validar que la clase hija redefina las funciones definidas en la clase interface, en caso contrario se lanza una excepción.

3.3.7 Imitate

Este recurso es similar al funcionamiento del *extend,* sin embargo este permite apropiarse de los recursos sean propiedades o funciones de varias clases padres. Como limitante se tiene que en caso de que se le incorpore algún recurso al *parent* posteriormente a la instanciación de la clase hija, esta no se percataría de dicha actualización. Teniendo en cuenta que esta funcionalidad se asigna directamente al objeto de la clase descendiente a diferencia de la anterior que se realiza sobre el prototipo de función.

Sintaxis:

std.OOP.**imitate** (objChild, parent, params);

- **objChild**: Instancia de la clase que adoptará como propias todas las propiedades y funcionalidades públicas de la clase *parent*.
- **parent:** Clase en la que se definen un conjunto de propiedades o funcionalidades que pueden ser reutilizadas por otras teniendo en cuenta su marcado carácter generalizador.
- **params:** Lista de parámetros en caso de ser necesarios para el constructor de la clase antecesora.

Ejemplo:

1. Crear una clase *Cat* que extienda su comportamiento a partir de las funcionalidades definidas en *Animal* e implemente *Feline*:

```
45. var Cat = function()
46. {
47.         std.OOP.imitate(this, Feline, arguments);
48. }
49. std.OOP.extend(Cat, Animal);
```

```
50. var misu = new Cat(“misu”, 5);
```

3.3.8 Class

El empleo del paradigma orientado a objetos trae consigo muchísimas ventajas repercutiendo directamente en el rendimiento de los productos finales, teniendo en cuenta que al evitarse la repetición de código se minimiza también el volumen en *byte* de los ficheros contenedores del código *Javascript* que son cargados en el cliente. Este *framework* provee un recurso que permite englobar todos los conceptos expresados y puede ser considerado como la base de metáfora de trabajo que simula la programación *web* como si fuese el desarrollo de aplicaciones de escritorio.

Sintaxis:

Kcl.Class(name, prototype)

- **name**: Propiedad de tipo cadena que corresponde al identificador de la clase, en caso de especificar el espacio de nombre se crea automáticamente.
- **prototype:** Objeto que define el prototipo de la clase que se desea definir. Esta contiene una serie de palabras reservadas las cuales se exponen a continuación:
  - **extend:** Define el nombre de la clase de la cual se desea extender.
  - **imitate:** Define el nombre o listado de clases de la cual se desea imitar en cuanto a comportamiento.
  - **implement:** Define el nombre o listado de clases de las cuales se desea implementar.
  - **patterns:** Nombre o listado de los identificadores de cada uno de los patrones de diseño que se desea incorporar a la definición de la clase.
  - **property:** Listado de atributos o propiedades que conforman la clase.
  - **behavior:** Listado de funciones o métodos que conforman la clase.

Este recurso está sobrecargado permitiendo crear una clase sin especificar su nombre, imitando de esta forma la declaración literal de funciones que provee el lenguaje *Javascript*, para mayor comprensión refiérase a el ejemplo 3, la sintaxis del mismo es la que aparece a continuación: Kcl.Class(prototype).

Ejemplo:

1. Crear una clase *Cats* que extienda su comportamiento a partir de las funcionalidades definidas en *Animal*, contenidas en el espacio de nombre *App*:

```
1.  Kcl.Class( "App.Animal",
2.  {
3.          property:{
4.                  name: “”,
5.                  age: “”
6.          },
7.          behavior: {
8.                  construct:function( name, age){
9.                          this.name = name;
10.                         this.age = age;
11.                 },
12.                 getName:function(){
13.                         return this.name;
14.                 }
```

```
15.     }
16. });
17.
18. Kcl.Class( "App.Cat",
19. {
20.         extend: App.Animal
21.         property: {
22.             iris: “”
23.         },
24.         behavior: {
25.                 construct:function(iris, name, age){
26.                         var _this = std. Cat.prototype;
27.                         _this.parent.construct.call(this, name, age);
28.                         this.iris = iris;
29.                 },
30.                 getIris:function(params){
31.                         return this.iris;
32.                 }
33.         }
34. });
35.
36. var misu = new App.Cat(“line”, “misu”, 5);
37. var iris = misu.getIris();
38. var name = misu.getName();
```

2. Crear una clase nombrada *Feline* que sin importar el número de veces que esta sea instanciada los objetos resultantes sean el mismo, contenidas en el espacio de nombre *App*:

```
39. Kcl.Class( "App.Feline",
40. {
41.         patterns: "Singleton",
42.         property: {
43.             tail: “”,
44.             glue: “”
45.         },
46.         behavior:{
47.                 construct:function( tail, glue){
48.                         this.glue = glue;
49.                         this.tail =tail;
50.                 },
51.                 attack: function(){
52.                         return "do Attack ";
53.                 }
54.          }
55. });
```

3. Crear una clase nombrada *Barcino* que extienda de *Cat* e imite el comportamiento de *Feline* y *Cuadrúpedo*, de igual forma que el ejemplo anterior sin importar el número de veces que

esta sea instanciada los objetos resultantes sean el mismo, contenidas en el espacio de nombre *App,* con la particularidad de que en este ejemplo se utilizara la declaración literal de clases:

```
std.$( "App").Barcino  = Kcl.Class(
{
        extend: App.Cat,
        imitate: [App.Feline, App.Cuadrupedo],
        patterns: "Singleton",
        property:{
                tail: “”,
                glue: “”
        },
        behavior:{
                construct:function( tail, glue, iris, name, age){
                        var _this = App.Barcino.prototype;
                        _this.parent.construct.apply(this, arguments);
                        this.glue = glue;
                        this.tail =tail;
                },
                attack: function(){
                        var _this = App.Barcino.prototype;
                        return "do Attack " + _this.parent.attack();
                }
        }
});

var misus = new App.Barcino();
var attac = misus.attack();
var iris  = misus.getIris();
```

Como se puede apreciar en la mayoría de los ejemplos para acceder a determinadas propiedades o funciones de cada clase se define la propiedad ***_this***, que no es más que la referencia en memoria del prototipo de dicha clase. Esto es muy importante que se tenga en cuenta partiendo de que también es posible acceder desde el ***this,*** recurso nativo del *Javascript* para acceder al objeto actual, sin embargo este se ve afectado dependiendo de quién fue el ejecutor de la llamada.

Este problema se puede apreciar principalmente cuando se emplea la herencia, en el momento que el constructor de la clase hija ejecuta el del padre para inicializar los atributos heredados, una vez dentro del propio constructor del padre el objeto ***this*** no se comporta como tal, sino que sigue siendo el objeto de la clase invocadora o hija, produciéndose en este caso un ciclo infinito. Este es un elemento que debe ser tomado con mucho cuidado, no todo debería ser accedido a través del prototipo de la clase, porque de lo contrario se verían afectados todos los objetos que se hallan instanciado a partir de la clase en cuestión. Se recomienda principalmente el empleo del *_this* en caso de que se desee utilizar propiedades como *parent,* para evitar problemas como el anteriormente explicado, partiendo del hecho que el this se comporta diferente en dependencia del contexto de invocación.

Otro elemento a tener en cuenta es el mecanismo que se emplee para inicializar los atributos de la clase padre, básicamente se consta de dos recursos fundamentales: ***call*** y ***apply.*** Se recomienda el empleo del *apply* cuando se tiene cierta incertidumbre en la cantidad de argumentos que se necesitan pasar al padre

y en caso contrario sería el *call* en el cual se deja plasmado explícitamente cuales serian los argumentos separados por el carácter de coma.

Ejemplo:

1. Utilizando el recurso *call* para invocar el constructor de *Cats* considerada como clase padre o superclase de *Barcino*:

```
1. construct:function( tail, glue, iris, name, age){
2.         var _this = App.Barcino.prototype;
3.         _this.parent.construct.apply(this, iris, name, age);
4. },
```

2. Utilizando el recurso *apply* para invocar el constructor de *Cats* considerada como clase padre o superclase de *Barcino*:

```
5. construct:function( tail, glue, iris, name, age){
6.         var _this = App.Barcino.prototype;
7.         _this.parent.construct.apply(this, [iris, name, age]);
8. },
```

3. Utilizando el recurso *apply* para invocar el constructor de *Cats* considerada como clase padre o superclase de *Barcino*. En este caso se desconoce la cantidad de parámetros o esta es variable, así que se utiliza el recurso *arguments* para introducirle a la clase padre todos los parámetros recibidos en la clase hija :

```
9. construct:function( tail, glue, iris, name, age){
10.         var _this = App.Barcino.prototype;
11.         _this.parent.construct.apply(this, arguments);
12. },
```

Como se ha podido apreciar todo los objetos creados a través de este recurso están provistos de una propiedad nombrada ***parent*** que permite el acceso a los recursos heredados. También es posible conocer algunos elementos de carácter informativo a través de la propiedad ***_inf,*** esta no es más que un objeto que brinda información como ***ns*** la cual retorna el valor del espacio de nombre y *type* que hace referencia del tipo de construcción en caso de tomar valor *"_class"* indica que ha sido creada bajo este recurso.

Uno de los elementos más significativos de este recurso es la posibilidad de emplear patrones de diseño en la definición de la clase sin necesidad de dedicar tiempo a su implementación. Estos pueden ser desarrollados de forma independiente como si fuesen *drivers* y ser empleados posteriormente a través de la propiedad *patterns*.

Ejemplo:

1. Definición del patrón de diseño *Singleton*

```
1. Kcl.$("DesignerPatterns").Singleton = function(_class, _obj)
2. {
3.     if(_class.prototype.instance) return _class.prototype.instance;
4.     else _class.prototype.instance = _obj;
```

```
5. }
```

Como se puede apreciar en el ejemplo anterior la implementación de determinado patrón no es nada extraordinario. Constituye una simple función que recibe por parámetro la clase y la instancia que se está creando donde podrá ser incorporado el comportamiento deseado. Como premisa si debe estar declarado dentro del espacio de nombres ***DesignerPatterns.***

3.4 Router

La librería *Router* es la responsable de conocer las direcciones relativas para cada uno de los componentes que conforman el *framework*.

Sintaxis:

var objName = **new** Router(params);

- **params:** Lista de parámetros compuesto por propiedades como *uri* y *proj*.

Ejemplo:

1. Tanto el *core* como el proyecto se encuentran en el mismo directorio.

```
6. std.Router = new Router();
```

2. En este caso se especifica que el *core* a utilizar en el proyecto nombrado *ToolDevelop* se encuentra dos niveles hacia arriba y que el mismo se encuentra contenido en el directorio *tools*

```
7. std.Router = new Router({uri:'../../', proj:'tools/ToolDevelop/'});
```

En los ejemplos se expone el proceso de instanciación de esta librería, sin embargo no es algo necesario teniendo en cuenta que el *framework* brinda un objeto global *std.Router*, mediante el cual se puede acceder a todas las funcionalidades que la misma brinda.

3.4.1 Uri

Esta propiedad permite identificar la ruta relativa del proyecto en ejecución hasta el *core* que este utilizará para su correcto funcionamiento. Esto es uno de los mecanismos que aseguran el carácter multiproyecto del *framework* y de forma transparente para los desarrolladores que lo utilizan.

3.4.2 Proj

Esta propiedad es similar a la que se describe en el apartado anterior pero en sentido inverso, permite identificar la ruta relativa del *core* hasta el proyecto.

3.4.3 ModulePath

Este mecanismo permite obtener la ruta relativa a cualquiera de los módulos definidos para el proyecto en cuestión. Es ventajoso su empleo en el sentido de que tiene en cuenta las rutas relativas al *core*, pudiéndose mover el proyecto de directorio sin afectar el código que depende de estos.

Sintaxis:

std.Router.**getModulePath**(libName, parent)

- **libName:** Nombre del módulo de cual se desea obtener la ruta relativa, por defecto toma valor *Main*.
- **parent:** Propiedad booleana que define si se toma en cuenta la *uri*, por defecto toma valor *false*

Ejemplo:

1. Incluir el fichero *icons.css* que se encuentra ubicado en el módulo principal del proyecto en cuestión.

```
8. var path = std.Router.getModulePath()
9. std.include(path+'css/icons.css');
```

3.4.4 CorePath

Este mecanismo permite obtener la ruta relativa al módulo cliente del *core*.

Sintaxis:

std.Router. **getCorePath** (parent)

- **parent:** Propiedad booleana que define si se toma en cuenta la *uri*, por defecto toma valor *true*

Ejemplo:

1. Incluir los ficheros *Factory.js* y *Loader.js* una vez concluida su carga se procede a instanciar e *Loader* y se comienza a cargar sincrónicamente una serie de librerías propias del *core* del framework.

```
10. std.include(std.Router.getCorePath()+'js/Factory.js',function(){
11.         std.include(std.Router.getCorePath()+'js/Loader.js',function(){
12.             std.Loader = new Loader({'libs':std, 'mods':mod});
13.             std.Loader.coreLibs([
14.                 'OOP',
15.                 'Communicator',
16.                 'FrontController',
17.                 {'name':'Primal','build':false},
18.                 {'name':'Plugin','build':false},
19.                 {'name':'App', 'build':false,
20.                 'callback': function(){
21.                     std.Loader.plugins(['Main']);
22.                 }}
23.             ]);
24.         });
25.  });
```

3.4.5 LibPath

Es similar a la función explicada en el apartado anterior, pero centrada en el directorio *lib*, donde se

encuentra las librerías externas al *framework*.

3.4.6 Path

Este mecanismo provee de una abstracción sobre sus predecesoras permitiendo obtener la ruta relativa conociendo el tipo de componente que se desea encontrar.

Sintaxis:

std.Router. **getPath** (libName, type)

- **libName:** Nombre del módulo de cual se desea obtener la ruta relativa, por defecto toma valor *Main*.
- **type:** Propiedad booleana que define el tipo de componente a utilizar, valores permitidos *core/lib*, en caso de no especificarse asume que es un módulo.

Ejemplo:

1. Incluir dinámicamente tanto módulos como librerías del *core*, en este caso serian el módulo *Route* y la librería de *core Factory*.

```
26.  var load = function(name, type)
27.  {
28.          var path = std.Router.getPath(name, type);
29.          std.include(path +"/js"+ name+".js");
30.  }
31.  load("Route", "plugin");
32.  load("Factory", "core");
```

3.5 Loader

La librería *Loader* se define como un cargador dinámico de librerías y módulos o *plugins*, abstrayendo a los desarrolladores de esta tediosa tarea, evitando de esta forma errores de tipo factor humano, que usualmente se incurren principalmente cuando se trata de *url* y direcciones relativas.

3.5.1 Plugins

Este mecanismo permite cargar en memoria las clases de cada uno de los módulos especificados sin necesidad de especificar la *url* de los mismos.

Sintaxis:

std.loader.**plugins**(plugins);

- ✓ **plugins:** Listado ordenado de los nombres de cada uno de los módulos que se desean cargar.

Ejemplo:

1. Cargar una lista de *plugins* o módulos con aproximadamente siete elementos:

```
33.  std.loader.plugins([
```

```
34.         "Layer",
35.         "Map",
36.         "Info",
37.         "Navigation",
38.         "ThematicRaster",
39.         "Rutas",
40.         "Localizador"
41. ]);
```

3.5.2 CoreLibs

Este mecanismo permite cargar en memoria un listado de librerías del *core* de forma asincrónica, pero con la particularidad de que es posible especificar el árbol de dependencia por cada una de estas.

Sintaxis:

std.loader.**coreLibs** (plugins)**;**

- ✓ **plugins:** Listado de las libreras del *core* que se desean cargar. Cada uno de los elementos que componen este listado pueden ser de tipo cadena o sea el nombre del la librería o un objeto con la siguiente estructura:
  - **name:** Nombre de la librería
  - **build**: Propiedad de tipo booleana que define la posibilidad de instanciar automáticamente la librería, en caso de no ser especificada toma valor true.
  - **callback**: Función que se llama una vez que termine la carga de la librería en cuestión.
  - **params**: Listado de atributos que son pasados por parámetros tanto por la función de *callback* como el constructor en caso de ser instanciada.
  - **child**: Listado de nombres u objetos que representan de forma arbolea la dependencia de ficheros de tipo *js y css* que serán cargados.

Ejemplo:

2. Cargar las librerías *OOP*, *FrontController* y *Primal*, una vez cargada esta última se procederá a cargar entonces *Plugin* y *App,* como se puede observar solo serán instanciadas dentro del espacio de nombre u objeto global las dos primeras y por otro lado al culminarse la carga de la clase *App* se invocara una función anónima que implícitamente manda a cargar el *plugin Main :*

```
42. std.Loader.coreLibs([
43.         "OOP",
44.         "FrontController",
45.         {
46.             "name":"Primal",
47.             "build":false,
48.             "child":[
49.                         {"name":"Plugin","build":false},
50.                         {       "name":"App", "build":false,
51.                                 "callback": function(){
```

```
52.                                         std.Loader.plugins(["Main"]);
53.                                 }}
54.                         ],
55.                 }
56.     ]);
```

3.6 Main

Esta constituye la clase principal del *framework* y de hecho es la primera en ejecutarse, mucho antes que el propio módulo servidor del mismo. La gran mayoría de los marcos de trabajo dedican esfuerzos en la generación y gestión de *GUI*, construyéndolo generalmente desde el código *php*. Esta filosofía se invierte con el objetivo de ganar en cuanto al rendimiento y el aprovechamiento de interfaces. De esta forma es posible crear un sitio completamente estático e incorporarle posteriormente elementos dinámicos, sin que este sufra cambios traumáticos en su estructura arquitectónica y de igual forma seria posible integrar este tipo de producto con otras aplicaciones a nivel de interfaz, aunque los elementos de integración se explicaran en acápites posteriores.

3.6.1 Definición de la aplicación o Main

Este módulo asegura de forma automática la carga de los *plugins* que se encuentren especificados en el fichero de configuración nombrado *conf* que se encuentra en el directorio *./app/config/,* tal y como se muestra a continuación*:*

```
1.  <?php
2.      $config["tpl"]["path"]  = 'server/templates/';
3.
4.      $config["plugins"][]    = "ProjectAdmin";
5.      $config["plugins"][]    = "TemplateAdmin";
6.      $config["plugins"][]    = "DoctrineAdmin";
7.      $config["plugins"][]    = "Help";
8.
9.      return $config;
10. ?>
```

En el ejemplo anterior se especifican en formato *php* que se deben cargar una vez inicializada la aplicación los módulos *ProjectAdmin, TemplateAdmin, DoctrineAdmin y Help.* Teniendo en cuenta esto se debe ser muy cuidadoso en el momento de definir la estructura de la clase *Main* y si se desea mantener este comportamiento debe ser especificado a través de la propiedad *parent* tanto para la función *construct* como *serverResponse,* de lo contrario podría hacerse de forma manual a través de la librería *Loader* o la funcionalidad *loadPlugins*. Para más detalles refiérase al ejemplo que aparece a continuación.

Un elemento importante a resaltar en este punto es que si no se define la variable que contiene los módulos que deberán ser cargados durante el proceso de auto load, entonces el *API* del servidor se encargara de escanear el directorio de *Plugin* y se cargarían todos los que en dicho directorio yacen.

Ejemplo:

1. Definición básica del módulo principal, manteniendo el comportamiento del padre tanto para el constructor como para la respuesta del servidor, en esencia esto garantiza la carga dinámica de los módulos especificados en el fichero de configuración**:**

```
1.
2.  Kcl.Class( "Qba.Main",
3.  {
4.      extend: Qba.App,
5.      behavior: {
6.                          construct : function(eve, loadType){
7.                                  var _this = Qba.Main.prototype;
8.                                  _this.parent.construct.apply(_this, [_this]);
9.                          },
10.                         serverResponse : function(objResponse){
11.                                 var _this = Qba.Main.prototype;
12.                                 _this.parent.serverResponse(objResponse, _this);
13.                                 switch(objResponse.action)
14.                                 {
15.                                         case '':
16.                                         break;
17.                                 }
18.                         },
19.                         onLoadPlugins : function(params){
20.                                 this.buildGUI({"gui": _this.gui});
21.                         }
22.             }
23. });
24.
```

3.6.2 Build GUI

A través del recurso *buildGUI* es posible insertar la información necesaria para que cada uno de los módulos de la plataforma, sean capaces de construir o modificar las interfaces gráficas de usuario, desde su ámbito y sin dependencia alguna con otros *plugins* o la propia aplicación.

Sintaxis:

this.buildGUI (params)**;**

- ✓ **params:** Listado de propiedades que serán pasadas por parámetro a cada una de las funciones *buildGUI* perteneciente a los módulos que componen la aplicación.

Ejemplo:

1. En el siguiente ejemplo se invoca la función *buildGUI* en cada uno de los módulos de la plataforma enviándoles la propiedad *gui y map*.

```
25.  this.buildGUI({"gui":this.gui, "map": "mod.Map.obj"});
```

3.6.3 Server Response

Este recurso es el responsable de recoger la respuesta del servidor, la cual se captura a través del

parámetro que aparece en su propia definición. Teniendo en cuenta que este recurso gestiona la totalidad de eventos sobre el controlador de la aplicación se recomienda que en la respuesta se incorpore una propiedad ***"action"***, la cual deberá ser evaluada en un ***switch*** ganando de esta forma en cuanto a claridad y comprensión del código, tal y como se muestra a continuación.

```
26.  serverResponse : function(objResponse)
27.  {
28.      switch(objResponse.action)
29.      {
30.          case "update":
31.              ...
32.          break;
33.          case "load":
34.              ...
35.          break;
36.          default:
37.              ...
38.          break;
39.      }
40.  }
```

Este mecanismo permite centralizar las peticiones para determinado módulo, incluyendo el *app*. Esto provoca el efecto conocido como de cuello de botella o embudo. No es más que un esquema organizativo que permite identificar con rapidez el resultado arrojado por el módulo servidor, agilizándose de cierta forma el proceso de revisiones por concepto de pruebas de unitarias para *software*. Pese a las ventajas que ese pueda proveer no constituye el único mecanismo para la recepción de los datos provenientes del servidor, aunque si el que viene definido por defecto.

Tanto los *plugin* como el *app* tienen definida una propiedad *typeResponse* cuyo valor por defecto es *centralize,* el cual puede ser redefinido permitiendo cambiar dicho esquema organizativo y redirigir la respuesta hacia la función del propio módulo cuyo nombre corresponda con la acción invocada en el servidor, en este caso debería especificarse como *decentralize*. Para mayor comprensión refiérase al ejemplo que se muestra a continuación:

Ejemplo: arraigar

1. En el siguiente ejemplo se representa la declaración del módulo *ToolsAdmin* el cual realiza una petición *ajax* en su propio constructor, con la singularidad de que al redefinir la propiedad *typeResponse* esta redirecciona la respuesta de la acción invocada en el módulo servidor nombrada *holaMundo* a su homóloga en el cliente, nótese que en este *plugin* no existe la necesidad de redefinir la función *serverResponse*.

```
1.  Kcl.Class( 'ToolsAdmin',
2.  {
3.      extend: Kcl.Plugin,
4.      behavior: {
5.                  construct:function(params){
6.                      var _this = ToolsAdmin.prototype;
7.                      _this.parent.construct.apply(this, []);
8.                      this.typeResponse = " decentralize";
9.
```

```
10.                     std.FrontController.send({
11.                                         action: 'holaMundo',
12.                                         controller: 'ToolsAdmin',
13.                                         params: {
14.                                                 'name':'ToolsAdmin obj',
15.                                                 'age': 23
16.                                         }
17.                     });
18.             },
19.             holaMundo:function(params){
20.                     alert(params.result+" >>holaMundo");
21.             }
22.         }
23. });
```

3.6.4 LoadPlugins

Este mecanismo permite cargar en memoria un listado de módulos ubicados en el directorio especificado para almacenar los *plugin* del proyecto. Es similar al que provee la librería *Loader* pero esta puede asumirse como un comportamiento propio de la clase.

Sintaxis:

this.loadPlugins (list, params, callback)**;**

- ✓ **list:** Listado de los nombres correspondientes para cada uno de los módulos que se desean cargar.
- ✓ **callback**: Función que se llama una vez que termine la carga de la librería en cuestión.
- ✓ **params:** Listado de atributos que son pasados por parámetros a la función de callback.

Ejemplo:

1. En el siguiente ejemplo se invoca la carga en memoria de forma estática de los *plugins Route* y *Layer,* una vez finalizado dicho proceso se invocara el evento *onLoadPlugins*.

```
24. var _plugins = ["Route", "Layer"];
25. var iterators = function(_length){
26.         if(_iter < _length-1)
27.             _iter++;
28.         else if(this.eve.onLoadPlugins)
29.          this.eve.onLoadPlugins(_plugins);
30. }
31. this.loadPlugins(_plugins, _plugins.length, iterators);
```

3.6.4 Construct

Teniendo en cuenta que los constructores permiten al programador establecer valores predeterminados, limitar la creación de instancias posibilitando de esta forma escribir código flexible y fácil de interpretar. Se decidió incorporar este recurso tanto para el *Main* como para los *plugins,* esto implica que cada vez

que se cree una instancia por el cargador dinámico de la plataforma denominado *loader* se hará una llamada intrínseca al la función ***construct***.

3.6.5 OnLoadPlugins

Este es el evento que es emitido automáticamente una vez que se haya finalizado la carga en memoria del último módulo. Sin embargo este comportamiento para ser utilizado como es lógico debe ser redefinido en la propia clase *Main,* tal y como queda reflejado en el ejemplo que se muestra a continuación.

```
1.  Kcl.Class( "Qba.Main",
2.  {
3.          extend: Qba.App,
4.          behavior: {
5.                  construct : function(eve, loadType){
6.                          var _this = Qba.Main.prototype;
7.                          _this.parent.construct.apply(_this, [_this]);
8.                          this.gui = new MainGUIDevelop();
9.                          this.gui.buildGUI();
10.                         this.gui.region.center.add({
11.                                 title:'Start Page',
12.                                 html:"<br><center><h1>Welcome to Mermas Tools Develop</h1>"
13.                         });
14.                 },
15.                 serverResponse:function(objResponse){
16.                         var _this = Qba.Main.prototype;
17.                         _this.parent.serverResponse(objResponse, _this);
18.                         switch(objResponse.action)
19.                         {
20.                                 case '':
21.                                 break;
22.                         }
23.                 },
24.                 onLoadPlugins : function(params){
25.                         this.buildGUI({"gui": this.gui});
26.                 }
27.         }
28. });
29.
30. Kcl.Main.require = [
31.         std.Router.getLibPath()+"ext/css/ext-all.css",
32.         std.Router.getLibPath()+"ext/css/misc/ux-all.css",
33.         std.Router.getLibPath()+"ext/js/ext-base.js",
34.         std.Router.getLibPath()+"ext/js/ext-all.js",
35.         std.Router.getLibPath()+"ext/js/ext-all-debug.js",
36.         std.Router.getModulePath()+'css/init.css',
37.         std.Router.getModulePath()+'js/common/MainMenuBar.js',
38.         std.Router.getModulePath()+'js/common/MainToolBar.js',
39.         std.Router.getModulePath()+'js/common/MainGUIDevelop.js'
```

```
40. ];
```

3.7 Plugins

En esencia un módulo se comporta de forma similar al *Main,* con la diferencia que este desconoce la existencia de otros y se responsabiliza únicamente de las propiedades y comportamientos asociados a lo que se desea implementar en dicho apartado.

Ejemplo:

1. Sintaxis básica para la declaración de un módulo, en este caso llamado *Map.*

```
1.  Kcl.Class( "Qba.Map",
2.  {
3.              extend: Qba.Plugin ,
4.              property: {
5.                      lay: null
6.              },
7.               behavior: {
8.                      construct : function(){
9.                          …
10.                     },
11.                     serverResponse:function(objResponse){
12.                         …
13.                     },
14.                     buildGUI:function(params){
15.                         …
16.                     }
17.             }
18.  });
```

2. Ejemplo en el que se combinan los mecanismos para conformar la lógica del módulo *Map*

```
19.  std.include("lib/OpenLayers/theme/default/style.css");
20.  std.include("lib/OpenLayers/examples/style.css");
21.  //-----------------------------------------------------------
22.  Kcl.Class( "Qba.Map",
23.  {
24.              extend: Qba.Plugin,
25.              property:{
26.                      lay: null
27.              },
28.              behavior: {
29.                      onResize : function(){
30.                              var _this = Qba.Map.prototype;
31.                              _this.obj.updateSize();
32.                     },
33.                     refresh : function(){
```

```
34.                         var _this = Qba.Map.prototype;
35.                         _this.lay.redraw(true);
36.                 },
37.                 serverResponse:function(objResponse){
38.                         var _this = Qba.Map.prototype;
39.                         switch(objResponse.action)
40.                         {
41.                           case "update":
42.                                   _this.refresh();
43.                           break;
44.
45.                           case "load":
46.                                   _this.obj.setOptions({
47.                                     projection: objResponse.projection
48.                                   });
49.                           break;
50.                         }
51.                 },
52.                 buildGUI:function(params){
53.                         var _this = Qba.Map.prototype;
54.                         params.gui.region.center.obj.addListener("resize",_this.onResize);
55.                         _this.obj = new OpenLayers.Map("msmap",{
56.                           allOverlays: true,
57.                           maxResolution: 'auto',
58.                           numZoomLevels: 9,
59.                           controls: [new OpenLayers.Control.KeyboardDefaults()]
60.                         });
61.                         std.FrontController.send({
62.                           action: 'load',
63.                           controller: "Map"
64.                         });
65.                 }
66.         }
67. });
68. Qba.Map.require = "lib/OpenLayers/OpenLayers.js";
```

3.8 Configuración

Este apartado no por ser el último deja de ser importante de hecho fue intencional. Pues es aquí donde se definen qué librerías se emplearán para determinado proyecto y cuáles no, así como las direcciones relativas para que se pueda acceder al núcleo compartido por varios proyectos y viceversa, En caso que dichas direcciones relativas sean omitidas se asumirá que los directorios principales se encuentran en el directorio raíz del *framework,* entre los que se destacan por su funcionalidad el *core y lib*.

Ejemplo:

1. En este ejemplo se ilustra el fichero de configuración para el proyecto nombrado *ToolDevelop*, el cual se encuentra en una ruta relativa de acceso hacia el core compartido identificada por ../../ y

como elemento de direccionamiento desde el núcleo al mismo *tools/ToolDevelop/*.

```
1.  var uri = '../../';
2.  var std  = new Kcl.Core();
3.   std.require([
4.              uri+'core/js/DesignerPatterns/Singleton.js',
5.              uri+'core/js/Class.js',
6.              uri+'core/js/Factory.js',
7.              uri+'core/js/Router.js',
8.              uri+'core/js/Loader.js',
9.              uri+'core/js/Base.js',
10.                         uri+'core/js/Format.js',
11.                         uri+'core/js/Format.JSON.js',
12.                         uri+'core/js/Communicator.js',
13.                         uri+'core/js/Communicator.Ajax.js',
14.                         uri+'core/js/Primal.js',
15.                         uri+'core/js/Plugin.js'
16.             ],function(){
17.                         std = new Kcl.Base({'uri':uri, 'proj':'tools/ToolDevelop/'});
18.                         std.load([
19.                                     'OOP',
20.                                     'FrontController',
21.                                     {'name':'App',  'build':false},
22.                                     {'name':'Main', 'type':"plugin"}
23.                         ]);
24.  });
```

Obsérvese que en ejemplo anterior se incluyen todas las librerías del *API* de *Javascript* que provee el *framework*, sin embargo esto debe ir enfocado a las necesidades de cada proyecto. Por concepto de rendimiento es posible que solo sea necesario cargar algunas en función de resolver determinado negocio que no requiera tanta complejidad o el empleo de las mismas. De todas formas se recomienda que se haga un estudio exhaustivo de las dependencias de cada una de las librerías en aras de asegurar el correcto funcionamiento de *Ksike*.

4. API Servidor

Este apartado se refiere a los mecanismos de diseños implementados por este *framework* para su correcto funcionamiento. Es código tal y como se describía en acápites anteriores está escrito en *PHP* en su versión 5 y teniendo en cuenta que este tipo de script se ejecuta en los nodos que funcionan como servidores de aplicaciones se le deja todo el procesamiento de lógica de negocio y acceso a datos para de esta forma aprovechar al máximo los recursos de hardware del mismo y aliviar las máquinas clientes.

4.1 Main

Esta constituye la clase controladora que se encarga de la gestión de las peticiones provenientes del cliente. Como característica fundamental y propia de la arquitectura definida para este marco de trabajo es la única que tiene conocimiento de la existencia de los módulos que componen la aplicación, de ahí la posibilidad de crear árboles de dependencia entre ellos, mecanismo del cual se estará explicando en acápites posteriores.

Al igual que demás módulos que componen la aplicación en desarrollo es posible extender sus funcionalidades implementando un grupo de interfaces de las cuales se abundará más en acápites posteriores, para una mejor apreciación refiérase al ejemplo que aparece a continuación.

Ejemplo:

1. Definición de la aplicación *o Main con* seguridad a nivel de acción

```
1.  class Main extends App implements Security
2.  {
3.      public function __construct()
4.      {
5.        parent::__construct();
6.        $this->linker->connect("update","ThematicRaster","update","Map",'pos', null);
7.        $this->linker->connect("update","MapTemplate","load", "Layer",'pos', null);
8.      }
9.
10.     public function showData($params)
11.     {
12.       return array(
13.             'first'=>$params->first,
14.             'last'=>$params->last,
15.             'file'=>'firstIn'
16.       );
17.     }
18.
19.     public function allow($certy, $acction){return true;}
20.     public function allowFailed(){}
21. }
```

4.2 Plugin

Al igual que el *Main*, cada *plugin* cuenta con una clase que se encarga de la gestión de las peticiones provenientes del cliente, así como la lógica del comportamiento que sea de interés para el desarrollador del mismo que este incorpore. La única restricción que debe cumplir esta es que su nombre debe ser exactamente el mismo que el del módulo, para una mejor apreciación refiérase al ejemplo que se expone a continuación.

Ejemplo:

1. Definición básica de un módulo o *plugin*

```
1.  class Route extends Plugin
2.  {
3.              public function __construct() {}
4.              public function showRecord ($params)
5.              {
6.                      $confobj = DriverManager::factory("ConfigManager");
7.                      $mainconf = $confobj->loadConfig("Main");
```

```
8.                      $rutaconf = $confobj->loadConfig();
9.
10.                     $db = DriverManager::factory("PgSQL", $mainconf ["db"]["name"]);
11.                     $result = $db->executeSQL("SELECT nombre, edad, ci FROM person",1);
12.
13.                     $modulePath = Router::getModulePath("Test");
14.                     $path  = $modulePath."server/common/Raster.bit";
15.                     $file  = DriverManager::factory("File", $path);
16.                     $dataf = $file->reed();
17.
18.                     return array(
19.                             'first'=>$rutaconf["tree"],
20.                             'last'=>$rutaconf["node"],
21.                             'file'=>$dataf,
22.                             'dbout'=>$result
23.                     );
24.             }
25.
26.             public function admninRecord ()
27.             {
28.                     return "info en texto plano";
29.             }
30.     }
```

4.3 Drivers

Un *Driver* o controlador es un paquete, biblioteca o repositorio de funcionalidades encapsuladas, que permite al operario interactuar con elementos externos. Provee una abstracción de la manipulación de estos, proporcionando una interfaz estandarizada para su empleo. Se puede esquematizar como un manual de instrucciones que le indica al componente de software cómo debe orientar, regular y comunicarse con un dispositivo en particular.

Este es otro de los conceptos que se introducen en el *framework*, permitiéndole a los desarrolladores de hacer uso de librerías externas a la plataforma. Esta potencialidad al igual que los directorios establecidos para la inclusión dinámica, sea a nivel de *plugin* o de aplicación, se encarga de cargar en memoria una sola vez el código *php*, pero en este caso no se limitaría desde el punto de vista organizacional, teniendo en cuenta que estas no tendrían que estar orientadas específicamente al negocio en el cual serian empleadas sino que se convertirían en soluciones más genéricas, además la librería *DriverManager* se encararía de gestionar una instancia única de los mismos en aras de minimizar costos por concepto de rendimiento.

Sintaxis:

DriverManager::factory($driverInfo, $params);

- **driverInfo:** Puede tomar dos posibles valores:
  - ✓ **String:** Se introduce el nombre del *driver* en forma de cadena, en este caso el nombre del directorio debe corresponder con el de la clase controladora.
  - ✓ **Array:** Este caso se emplea cuando la estructura física de almacenamiento del *driver* es más compleja que el caso anterior, permitiéndose de esta forma especificar el nombre del fichero y su ruta relativa en que se encuentra declarada la clase controladora a través de el índice "*file*" y el nombre de dicha

clase a través del índice "*class*"

- **params**: Parámetro o Listado de parámetros requeridos por el driver en cuestión, es de carácter opcional y por defecto toma valor vacío.

Ejemplos:

1. Instanciar el *driver WsdlWrite,* esta se encuentra definida con un constructor por defecto sin parámetros, el fichero en que se encuentra definida es "***../lib/WsdlWrite/WsdlWrite.php"***

```
31.  $ wsdlWrite = DriverManager::factory("WsdlWrite");
```

2. Instanciar el *driver WsdlWrite,* esta se encuentra definida con un constructor parametrizado por un único elemento, el fichero en que se encuentra definida es "***../lib/WsdlWrite/WsdlWrite.php"***

```
32.  $ wsdlWrite = DriverManager::factory("WsdlWrite", "publication/services");
```

3. Instanciar el *driver WsdlWrite,* esta se encuentra definida con un constructor parametrizado, el fichero en que se encuentra definida es "***../lib/WsdlWrite/Static/Write.php"***

```
33.  $ wsdlWrite = DriverManager::factory(
34.              array(
35.                        "file"=>"WsdlWrite/Static/Write.php ",
36.                        "class"=>" WsdlWrite "
37.             ),array(
38.                        ".wsdl",
39.                       "publication/services/",
40.                       "installations"
41.  ));
```

En correspondencia con lo anteriormente expresado, los *driver* deben de estar ubicados en el directorio ***lib***, la idea principal es que los propios desarrolladores definan sus *drivers* y de esta forma se retroalimentaría el *framework*, no obstante se brindan algunos ya desarrollados tales como: *ConfigManager*, *PgController*, *FileController*, etc.

4.3.1 Configuración

Este manejador se encarga de la gestión dinámica de los ficheros de configuración sobre el framework *Ksike*. Su funcionamiento se basa en un listado de formatos soportados por el mismo el cual debe estar organizado por prioridad. Es uno de los driver que potencian la escalabilidad del marco de trabajo teniendo en cuenta que los formatos pueden ser desarrollados por los propios desarrolladores en caso que soporte dicha tecnología no esté contemplado entre las necesidades de los usuarios finales.

Partiendo de este recurso se hace posible acceder a la configuración local para determinado *plugin*, para ello es necesario que se solicite una instancia del controlador *ConfigManager*. De esta forma se garantiza que solo sea cargado en memoria aquellos drivers que realmente serán utilizados por el desarrollador ganando por concepto de rendimiento.

Para poder emplearlo como es lógico lo primero es crear una instancia del mismo a través de administrador de *driver,* tal y como se presenta a continuación.

Sintaxis:

$ ObjConfigManager = DriverManager::factory("ConfigManager", $order);

- **order:** Hace alusión al orden que se tendrá en cuenta en el momento que se desee hacer la gestión de la carga de los ficheros de configuración, lo identificadores se refieren a los formatos o extensiones de los archivos. Por defecto asume que el orden es: *php, json, xml, ini, txt*

Para acceder a la configuración contenida en los ficheros ubicados dentro del contexto definido por el marco de trabajo tanto para los módulos como el *app* véase la sintaxis que se muestra a continuación. También es posible acceder a la configuración de la aplicación desde cualquiera de los módulos que integren la plataforma, la sintaxis es similar a la anterior, con la particularidad de al cargar la misma se le debe especificar el indicador *"Main",* para mayor comprensión refiérase al ejemplo #2.

Sintaxis:

ObjConfigManager -> loadConfig($module);

- **module:** Identifica el módulo que requiere la carga en memoria del fichero de configuración, por defecto asume el nombre del plugin que lo invoca.

Este driver provee un recurso que permite acceder a un fichero de configuración que se encuentre ubicado fuera del contexto definido por *Ksike* para la gestión de los mismos sea por *plugins* o *app*. La función *loadConfigFrom* permite solventar este tipo de necesidad, potenciando el empleo del mismo desde librerías u otros *drivers*.

Sintaxis:

ObjConfigManager -> loadConfigFrom ($path);

- **module:** Identifica el módulo que requiere la carga en memoria del fichero de configuración, por defecto asume el nombre del *plugin* que lo invoca.

El orden prioritario que define la búsqueda por concepto de extensión de ficheros de configuración puede ser modificada dinámicamente a través de la función *setOrderPriory* tal y como se muestra en la sintaxis a continuación, para mayor comprensión refiérase al ejemplo #4.

Sintaxis:

ObjConfigManager -> setOrderPriory ($order);

- **order**: Hace alusión al orden que se tendrá en cuenta en el momento que se desee hacer la gestión de la carga de los ficheros de configuración, lo identificadores se refieren a los formatos o extensiones de los archivos**.**

Ejemplos:

1. Almacenar en la variable *rutaconf* los datos definidos en el fichero *conf* ubicado en el directorio *config* del *plugin Route,* es necesario esclarecer que esta llamada se hace desde el controlador *php* del propio módulo.

```
42. $confobj = DriverManager::factory("ConfigManager");
43. $rutaconf = $confobj->loadConfig();
```

2. Almacenar en la variable *mainconf* toda la información de tipo configuración correspondiente al módulo *Main* del *framework,* en forma de arreglo multidimensional o

matriz.

```
44. $confobj = DriverManager::factory("ConfigManager");
45. $mainconf = $confobj->loadConfig("Main");
```

3. Almacenar en la variable *externalconf* la configuración definida en un directorio externo o fuera del contexto definido para el almacenamiento de este tipo de archivos sobre la plataforma. Por tanto ni siquiera tiene porque ser llamado desde el contexto de un *plugin* o *app*, pudiendo emplearse en driver y librerías.

```
1. $confobj = DriverManager::factory("ConfigManager");
2. $path = __FILE__."config/conf.";
3. $externalconf = $driv->loadConfigFrom($path);
```

4. Modificar el orden de búsqueda del fichero de configuración priorizando los formatos *XML y JSON*. En este caso quedarían descartados los demás formatos, esto podría ser útil en caso que se desee optimizar la búsqueda reduciéndola a un espectro más pequeño.

```
4.$confobj = DriverManager::factory("ConfigManager");
5.$confobj-> setOrderPriory(array("xml","json"))
6.$rutaconf = $confobj->loadConfig();
```

4.3.1.2 Creando nuevos formatos

Para la generación de nuevos formatos de interpretación para ficheros que gestionen en alguna medida la configuración para determinado contexto, solo es necesario tener en cuenta tres elementos básicos.

- ✓ **Restricción de nomenclatura**: Hace referencia a que el nombre del formato debe tener implícito como prefijo *ConfigDriver* y seguido en mayúscula completamente el identificador del mismo.
- ✓ **Directorio de publicación**: Se refiere al directorio donde debe ser depositados estas clases, el cual debe ser dentro del directorio del propio driver *lib/ConfigManager/configDrivers/*.
- ✓ **Herencia**: El nuevo formato desarrollado deberá extender de *ConfigDriver* y redefinir la función *getConfig*, la cual se encarga de interpretar el formato contenido en el fichero especificado y retornar en forma de arreglo asociativo la información que en el mismo subyace, en función de que sea realmente homogéneo su empleo.

Sintaxis:

class ConfigDriver<id> **extends** ConfigDriver

- **id**: identificador del formato a interpretar**.**

Ejemplos:

1. Desarollar un formato que permita leer un fichero cuyo contenido se encuentre en formato json.

```
5.  class ConfigDriverJSON extends ConfigDriver
6.  {
7.          private $fileConf;
8.          //.....
9.          public function __construct($path="")
10.         {
11.                 parent::__construct($path);
```

```
12.                     $this->fileConf = 0;
13.             }
14.             //.....
15.             public function getConfig()
16.             {
17.                     if(!$this->fileConf) $this->fileConf = include $this->file;
18.                     return  json_decode($this->fileConf, true);
19.             }
20.     }
```

4.3.2 Formato de Salida

OutManager constituye un manejador de formatos de salida para el *API* que provee el *framework*, permitiendo potenciar la escalabilidad del mismo en dicho sentido. Por defecto *Ksike* asume como formato para el envío y recepción de datos entre cliente y servidor el json. Sin embargo no es una limitante a través de este driver es posible que los desarrolladores exploten su habilidades creativas e implementen sus propios formatos, de forma tal que la acción invocada les retorne el resultado esperado de la forma menos traumática posible.

Ejemplos:

1. Pese a que su empleo es prácticamente transparente a los desarrolladores se exponen un ejemplo de cómo se pude poner en práctica el mismo, aunque su verdadero impacto radica en la posibilidad de desarrollar nuevos formatos.

```
1.  $outFormat = 'img';
2.  $outOption = 'png';
3.  $OutManager = DriverManager::factory('OutManager', array($outFormat, $outOption));
4.  die( $OutManager->getOut($this->out) );
```

4.3.2.1 Creando nuevos formatos

Para desarrollar nuevos formatos tan solo deber crearse una clase ubicada en el directorio *OutManager/outDrivers* que contiene como restricción de nombre *OutDriver* seguido del nombre del formato en mayúscula. Esta debe redefinir el método getOut de la clase *OutDriver*. Para mayor comprensión refiérase al ejemplo que se expone a continuación:

Sintaxis:

class OutDriver<id> **extends** OutDriver

- **id**: identificador del formato a interpretar**.**

Ejemplos:

5. En este ejemplo se expone cómo se podría desarrollar un formato de salida *json* simple:

```
7.  class OutDriverJSON extends OutDriver
8.  {
9.          public function  __construct($option="")
```

```
10.         {
11.                 parent::__construct($option);
12.         }
13.
14.         public function getOut($data=false)
15.         {
16.                 $this->buildHead("txt/json");
17.                 if($data) return  json_encode($data);
18.         }
19.  }
```

Como bien se puede apreciar es un ejemplo bien sencillo, sin embargo ilustra claramente todo lo que podría desarrollarse partiendo de este mecanismo. Por ejemplo en caso que se requiera desarrolla una aplicación que contenga gestión documental, o se desee exportar simplemente determinada información a *pdf*. Una solución podría ser desarrollar una librería que se encargue de la generación del mismo, sin embargo sería mucho más factible si se construyera un formato que en su salida implementara la conversión a *pdf,* de forma tal que sin importar qué acción se esté invocando, la plataforma permitirá transformar el valor retornado a la salida deseada, con tan solo especificar el formato y las opciones de configuración que el mismo requiera.

Este es tan solo una de las aplicaciones que podría brindarse, sin embargo puede convertirse en un mecanismo fuerte para la integración con otras tecnologías. Tal es el caso de *OpenLayers*, librería desarrollada en lenguaje *Javascript* para la manipulación de mapas en el cliente. Esta tiene como particularidad que interpreta algunos formatos muy específicos del negocio que intenta gestionar, como es el caso de *gml* y *kml* variaciones del *xml* pero en función del almacenamiento de datos georeferenciados al igual que el *geojson* del propio *json*.

4.3.3 Bases de Datos

Debido al desarrollo tecnológico de campos como la informática y la electrónica, la mayoría de las bases de conocimientos se han almacenado en formato digital. Sumado a esto el efecto generado por el empleo de la red de redes con respecto al incremento exponencial de información, implica una insostenibilidad en la gestión de los datos generados. En consecuencia a dicho fenómeno surgen los *DBMS*[9].

En sentido general las db[10] pueden clasificarse de varias formas, de acuerdo al contexto que se manipulen, la utilidad de las mismas o las necesidades que satisfagan, sin embargo el término más relevante lo define su modelo de administración. Teniendo en cuenta que un modelo de datos es básicamente una abstracción que permite la implementación de un sistema eficiente de base de datos y que por lo general se refiere a algoritmos o conceptos matemáticos. Algunas de estas clasificaciones se muestran a continuación:

- ✓ **Jerárquicas:** Estas almacenan su información en una estructura jerárquica. En este modelo los datos se organizan en una forma similar a un árbol, en donde un nodo padre de información puede tener varios hijos. El nodo que no tiene padres es llamado raíz, y a los nodos que no tienen hijos se los conoce como hojas. Estas son especialmente útiles en el caso de aplicaciones que manejan un gran volumen de información y datos muy compartidos permitiendo crear estructuras estables y de gran rendimiento. Una de las principales limitaciones de este modelo es

[9] **DBMS** acrónimo de *Database Management System*
[10] **DB** acrónimo de *Database*

su incapacidad de representar eficientemente la redundancia de datos.

- ✓ **Red:** Este es un modelo ligeramente distinto del anterior, su diferencia fundamental es la modificación del concepto de nodo. Permite que un mismo nodo tenga varios padres. De esta forma se ofrece una solución eficiente al problema de redundancia de datos; pero, aún así, la dificultad que significa administrar la información en una base de datos de red ha significado que sea un modelo utilizado en su mayoría por programadores más que por usuarios finales.

- ✓ **Transaccionales:** Son bases de datos cuyo único fin es el envío y recepción de datos a grandes velocidades, estas bases son muy poco comunes y están dirigidas por lo general al entorno de análisis de calidad, datos de producción e industrial, es importante entender que su fin único es recolectar y recuperar los datos a la mayor velocidad posible, por lo tanto la redundancia y duplicación de información no es un problema como con las demás bases de datos, por lo general para poderlas aprovechar al máximo permiten algún tipo de conectividad a bases de datos relacionales.

- ✓ **Relacionales:** Uno de los más utilizados en la actualidad para modelar problemas reales y administrar datos dinámicamente. Tanto el lugar y la forma en que se almacenen los datos no tienen relevancia. Los datos se almacenan y se accede a ellos por medio de relaciones. El lenguaje más habitual para construir las consultas a bases de datos relacionales es *SQL*[11].

- ✓ **Multidimensionales**: Son bases de datos ideadas para desarrollar aplicaciones muy concretas, como creación de *Cubos OLAP*. Básicamente no se diferencian demasiado de las *db* relacionales, la diferencia está más bien a nivel conceptual. En este modelo los campos o atributos de una tabla pueden ser de dos tipos, o bien representan dimensiones de la tabla, o bien representan métricas que se desean estudiar.

- ✓ **Documentales:** Permiten la indexación a texto completo, y en líneas generales realizar búsquedas más potentes.

- ✓ **Deductivas:** Permite hacer conjeturas a través de inferencias. Se basa principalmente en reglas y hechos que son almacenados en la base de datos. También conocidas como bases de datos lógicas, a raíz de que se basa en lógica matemática.

- ✓ **Distribuida**: Tanto la *db* como el *SGBD* pueden estar distribuidos en múltiples sitios conectados por una red. Surgen debido a la existencia física de organismos descentralizados.

Partiendo de necesidad para emplear controladores que permitan la gestión de datos alfanuméricos, se decidió proveer de dos mecanismos fundamentales para tal fin y que evidentemente tengan en cuenta los aspectos mencionados: *PgSQL y Doctrine.*

4.3.3.1 PgSQL

Constituye un driver básico, basado en las funciones nativas del lenguaje *PHP 5* en función de gestionar consultas a bases de datos soportadas en *ProstgersSQL*. Se recomienda su empleo para aplicaciones cuya gestión de información no se encuentre representada por grandes volúmenes de tablas y relaciones. Teniendo en cuenta esto las recuperaciones de datos no se caracterizarán por consultas complejas en lenguaje *SQL*, además de que el rendimiento es mucho mayor que en el empleo de un *ORM*.

[11] **SQL** acrónimo de *Structured Query Language*

Sintaxis:

DriverManager::factory("PgSQL", $params);

- **params**: Listado ordenado de parámetros requeridos por el driver en cuestión, su estructura corresponde al listado que aparece a continuación.
  - **dbname**: Nombre de la base de datos a utilizar,
  - **user**: Usuario para establecer la conexión, es de carácter opcional y por defecto toma valor *'postgres'*,
  - **password**: Contraseña correspondiente al usuario definido por el parámetro anterior, es de carácter opcional y por defecto toma valor *'postgres'*,
  - **host**: Identificador de la dirección *IP* o *DNS* del servidor de bases de datos donde se encuentra definida la *db* especificada, es de carácter opcional y por defecto toma valor *'localhost'*,
  - **port**: Valor del puerto para establecer conexión con el servidor, es de carácter opcional y por defecto toma valor *'5432'*

Es importante destacar que en caso de invocarse el mismo driver por segunda vez con otros parámetros configurativos, estos no surtirán efecto sobre el objeto obtenido. En su lugar se obtendrá el anterior, teniendo en cuenta que el objetivo de esta clase es entregar una instancia única sin importar cuantas veces sea invocada, en aras de minimizar costos por concepto de rendimiento. Para resolver dicha necesidad se debe acceder a los atributos a través de los propios mecanismos que provea el *driver,* para una mayor comprensión véase los ejemplos 1,2 y 3 de esta sección.

El driver de PsSQL provee un mecanismo para la ejecución de consultas *SQL*, para mayor comprensión véanse los ejemplos 4 y 5 se muestran a continuación.

Sintaxis:

$objPgSQL->**executeSQL** ($syntaxSQL, $SELECT);

- **syntaxSQL**: Cadena de texto con la estructura definida en lenguaje *sql* de la consulta que se desea ejecutar en el sistema gestor de bases de datos.
- **SELECT**: Propiedad booleana que define el tipo de consulta a ejecutar en caso de tomar valor *true* o *1* retornará un arreglo asociativo donde la clave del mismo corresponde con el nombre del campo de la tabla en la base de datos, es de carácter opcional y toma por defecto valor *false*.

Ejemplos:

1. Obtener el driver de PgSQL para establecer conexión con una base de datos basada en los siguientes elementos configurativos: dbname=*'DataSpatial'*, user=*'postgres'*, password=*'postgres'*, host=*'localhost'*, port=*'5432'*

```
1. $db = DriverManager::factory("PgSQL", "DataSpatial");
```

2. Obtener el driver de PgSQL para establecer conexión con una base de datos basada en los siguientes elementos configurativos: dbname=*'SpatialSig'*, user=*'gsig'*, password=*'sig2010'*, host=*'10.171.1.7'*, port=*'5432'*

```
1. $db = DriverManager::factory("PgSQL", array(
2.     'SpatialSig'
```

```
3.      'gsig'
4.      'sig2010'
5.      '10.171.1.7'
6.  ));
```

3. Obtener por segunda vez el driver de PgSQL para establecer conexión con una base de datos basada en los siguientes elementos configurativos: dbname=*'SpatialRepo'*, user=*'postgres'*, password=*'postgres'*, host=*'database.uci.cu'*, port=*'5433'*

```
7.  $db = DriverManager::factory("PgSQL");
8.  $db->dbname = 'SpatialRepo';
9.  $db->user = 'postgres';
10. $db->password = 'postgres';
11. $db->host = 'database.uci.cu';
12. $db->port = '5433';
```

4. Incrementar en *200* unidades el valor del campo *amount* perteneciente a la tabla *account*, donde se cumpla que el valor del campo *id* de dicha tabla sea igual a *333*:

```
1.  $db = DriverManager::factory("PgSQL", $mainconf ["db"]["name"]);
2.  $db->executeSQL("UPDATE account SET amount = amount + 200 WHERE id > 333");
```

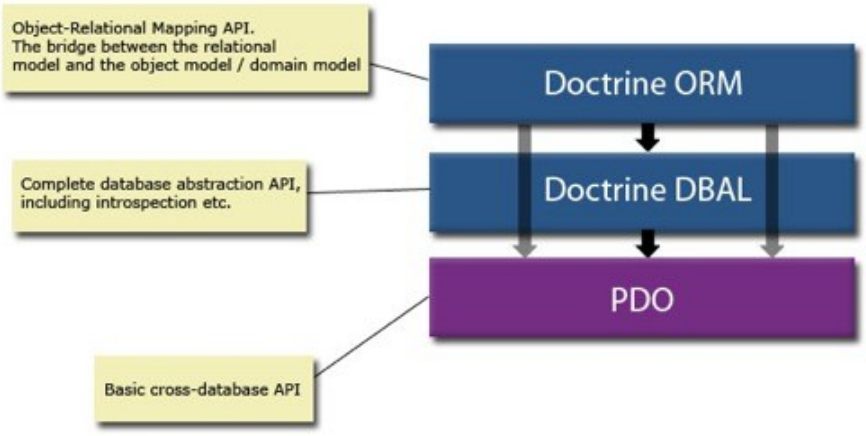
5. Obtener todos los registros con la información asociada por *nombre, edad y ci* perteneciente a la tabla *person*:

```
3.  $result = $db->executeSQL("SELECT nombre, edad, ci FROM person", 1);
4.  print_r($result);
5.
out: array(
        array("nombre"=>"iban", "edad"=>"12","ci"=>"85103111703"),
        array("nombre"=>"tusa", "edad"=>"22","ci"=>"76103111713"),
        array("nombre"=>"dali", "edad"=>"33","ci"=>"66103111704"),
)
```

4.3.3.2 Doctrine

Doctrine es una librería para *PHP* que nos permite trabajar con un esquema de base de datos como si fuese un conjunto de objetos. Está inspirado en *Hibernate*, uno de los ORM[12] más utilizados brindando una capa de abstracción para bases de datos muy completa. La característica más importante es que da la posibilidad de escribir consultas de base de datos en un lenguaje propio llamado *DQL*[13].

[12] **ORM** acrónimo de Object Relational Mapping
[13] **DQL** acrónimo de Doctrine Query Language



Elementos de interés a tener en cuenta para su empleo:

- ✓ **Generación automática del modelo:** Cuando se trabaja con *ORM*, necesitas crear el conjunto de clases que representa el modelo de la aplicación, luego estas clases serán vinculadas al esquema de la base de datos de forma automática con un motor *ORM*.

  Aunque son cosas diferentes, cuando diseñas un modelo relacional y un modelo de clases, suelen ser muy parecidos. Doctrine se aprovecha de esta similitud y nos permite generar de forma automática el modelo de clases basándose en el modelo relacional de tablas.

- ✓ **YAML**: Como se comenta en el apartado anterior, Doctrine puede generar de forma automática el modelo, pero también deja la posibilidad de que el propio desarrollador pueda definir el mapeo de tablas y sus relaciones. Esto se puede hacer con código *PHP* o con *YAML*, que es un formato de serialización de datos legible por humanos muy usado para este fin.

- ✓ **Magic finders:** En Doctrine, se pueden buscar registros basándose en cualquier campo de una tabla. Al igual que se pueden hacer búsquedas a través de la función *find*(), se podría hacer *findByX*() en donde *X* es el nombre de la columna, por ejemplo si existen los campos llamados *name* y *email*, se emplearían las funciones *findByName*() y *findByEmail*(). Además este provee el método *findAll*(), que obtiene todos los registros de la tabla.

- ✓ **Lenguaje DQL**: Creado para abstraer al programador sobre la gestión de información de la base de datos. Entre las ventajas de usar este lenguaje se encuentran:

  - Está diseñado para extraer objetos, no filas, que es lo que nos interesa.
  - Entiende las relaciones, por lo que no es necesario escribir los *joins* de forma manual.
  - Portable con diferentes sistemas gestores de bases de datos.

  Es importante considerar el uso de *DQL* para obtener la información a cargar en lugar de utilizar la forma automática de Doctrine para mejorar el rendimiento.

  Ejemplo: Teniendo en cuenta que se cuenta con dos tablas, *User* y *Comments*, con relación de 1-N elementos

```
6.  $q = Doctrine_Query::create()
7.      ->select('c.*,u.name')
8.      ->from('Comments c')
9.      ->innerJoin('c.User u');
```

```
10.  $comments = $q->execute();
```

- ✓ **Relaciones entre entidades:** En Doctrine, una vez definido el modelo con las tablas y sus relaciones, resulta fácil acceder y moverse por entidades relacionadas.

```
11.  $commentsTable = Doctrine_Core::getTable('Comments');
12.  $comments = $commentsTable->findAll();
13.  foreach($comments as $comment){
14.    echo  $comment->User->name;
15.  }
```

A través de $comment->User estamos accediendo a un nuevo objeto del tipo *User* que se carga de forma dinámica, de esta forma podemos acceder a las propiedades y métodos de dicha clase olvidándonos de tener que lanzar código *SQL*.

Un elemento importante a resaltar es la posibilidad de definir modelos para bases de datos por *plugins,* asegurando de esta forma la autonomía de estos a nivel de dato. Los ejemplos 1, 2 y 3 que se exponen a continuación refjejan claramete la puesta en práctica de lo expresado. Para instanciar este driver es necesario que se tengan en cuenta algunos aspectos como los que se muestran a continuación y al igual que el anterior para modificar los atributos de carácter configurativo se debe hacer mediante los mecanismos que el mismo provee. Para mayor comprensión refiérase al ejemplo 5 de este acápite.

Sintaxis:

DriverManager::factory("Doctrine", $params);

- **params**: Parámetro o Listado ordenado de parámetros requeridos por el driver en cuestión, su estructura corresponde al listado que aparece a continuación.
  - **dbname**: Nombre de la base de datos a utilizar.
  - **options:** Listado asociativo de carácter opcional, a continuación se muestra cada una de las claves que pueden ser especificadas:
    - ✓ **user**: Usuario para establecer la conexión, es de carácter opcional y por defecto toma valor *'postgres'*,
    - ✓ **password**: Contraseña correspondiente al usuario definido por el parámetro anterior, es de carácter opcional y por defecto toma valor *'postgres'*,
    - ✓ **host**: Identificador de la dirección IP o DNS del servidor de bases de datos donde se encuentra definida la db especificada, es de carácter opcional y por defecto toma valor *'localhost'*,
    - ✓ **port**: Valor del puerto para establecer conexión con el servidor, es de carácter opcional y por defecto toma valor *'5432'*
    - ✓ **driver:** Propiedad que define el tipo de gestor de bases de datos al cual se le establecerá la conexión, valores permitidos (*pgsql/mysql*), es de carácter opcional y por defecto toma valor *'pgsql'*:
  - **inv:** Propiedad que define el módulo contenedor del directorio donde se encuentran definido los modelos a cargar, es de carácter opcional y toma por defecto valor *Main.*

Ejemplos:

1. Obtener el driver de *Doctrine* para establecer conexión con una base de datos basada en los siguientes elementos configurativos: dbname=*'SpatialSig'*, user=*'postgres'*, password=*'postgres'*, host=*'localhost'*, port=*'5432',* driver=*'pgsql'* y con el modelo definido en el *plugin Main*.

```
1. $db = DriverManager::factory("Doctrine", 'SpatialSig');
```

2. Obtener el driver de *Doctrine* para establecer conexión con una base de datos basada en los siguientes elementos configurativos: dbname=*'SpatialSig'*, user=*'postgres'*, password=*'postgres'*, host=*'localhost'*, port=*'5432',* driver=*'pgsql'* y con el modelo definido en el *plugin Route*.

```
1. $db = DriverManager::factory("Doctrine", array('SpatialSig', null, 'Route'));
```

3. Obtener el driver de *Doctrine* para establecer conexión con una base de datos basada en los siguientes elementos configurativos: dbname=*'SpatialSig'*, user=*'gsig'*, password=*'sig2010'*, host=*'10.171.1.7'*, port=*'5432',* driver=*'pgsql'* y con el modelo definido en el *plugin Main*.

```
1. $db = DriverManager::factory("Doctrine", array(
2.     'SpatialSig',
3.     array(
4.         'user'=>'gsig'
5.         'password'=>'sig2010'
6.         'host'=>'10.171.1.7'
7.     )
8. ));
```

4. Insertar los datos asociados a la tabla *Users* cuyos campos correspondientes son *ci, name, email*.

```
1.  $db = DriverManager::factory("Doctrine", $mainconf["db"]["name"]);
2.
3.  $user = new Users();
4.  $user->ci = 85103111703;
5.  $user->name = "tusa";
6.  $user->email = "tusa@mail.cu";
7.  $user->save();
8.
9.   $commentsTable = $db->getTable("Users");
10. $tpl = $commentsTable->findAll();
```

5. Obtener por segunda vez el driver de PgSQL para establecer conexión con una base de datos basada en los siguientes elementos configurativos: dbname=*'SpatialRepo'*, user=*'postgres'*, password=*'postgres'*, host=*'database.uci.cu'*, port=*'5433"5432',* driver=*'pgsql'* y con el modelo definido en el *plugin Route*.

```
11. $db = DriverManager::factory("Doctrine");
12. $db->dbname = 'SpatialRepo';
13. $db->user = 'postgres';
```

```
14.  $db->password = 'postgres';
15.  $db->host = 'database.uci.cu';
16.  $db->port = '5433';
17.  $this->connect();
18.  $this->loadClassModel('Route');
19.
20.  $q = Doctrine_Query::create()
21.      ->update('Account')
22.      ->set('amount', 'amount + 200')
23.      ->where('id > 200');
24.
25.  $q->set('amount', '?', 500);
26.  echo $q->getSqlQuery();

OUT: UPDATE account SET amount = amount + 200 WHERE id > 200
```

4.3.4 Administración de ficheros

Este *framework* provee un driver básico que abstrae a la gestión y manipulación de ficheros basada en las funciones estándar definidas en el php 5.

```
27.  $path = $modulePath."server/common/Raster.bit";
28.  $file  = DriverManager::factory("File", $path);
29.  $data  = $file->reed();
```

4.4 Enrutador

Este recurso permite a los desarrolladores abstraerse del direccionamiento dentro de la plataforma.

Sintaxis:

Router::**getModulePath**($module);

- **module:** Nombre o identificador del módulo al cual se desea obtener la ruta relativa, en caso de omitirse la librería *Router* asume que se trata del *Main*.

Ejemplos:

2. Obtener el camino relativo al plugin *Distance*

```
30.  $modulePath = Router::getModulePath("Distance");
```

3. Obtener el camino relativo de la aplicación o main

```
31.  $modulePath = Router::getModulePath();
```

4.5 Signals/Slots

Ksike proporciona una *API* de programación con una orientación similar a las aplicaciones de escritorio. En este tipo de aplicaciones es importante proveer de un mecanismo claro de comunicación entre

objetos para minimizar el acoplamiento entre los diferentes módulos. Al igual que otros *framework* como Qt, se proporciona un mecanismo basado en "*Signals*" y "*Slots*".

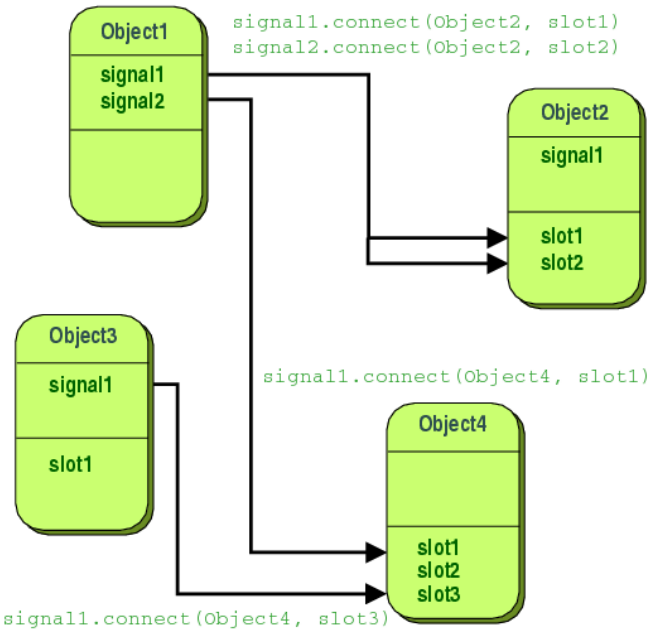


Figura 13: *Signals and Slots*

Cuando se produce un evento en un objeto, éste puede emitir una señal. Esa señal se puede conectar con un slot el cual no deja de ser un método de otro objeto que responde a la misma, estas podrían interpretarse como los valores de retorno que se enviaran al método que está conectado a dicha señal.

El recurso *linker* o conector es una propiedades de la clase *Main* de la aplicación. Posibilitando de esta forma vincular entre si los distintos *plugins* y reutilizar sus comportamientos con la menor dependencia posible.

Las *signals* pueden ser conectadas con uno o varios *slots* y estos a su vez pueden estar vinculados con diversas *signals*.

Existen dos peculiaridades que son necesarias destacar, la primera es que dicho comportamiento es recursivo, lo que quiere decir es que si se produce el grafo de dependencia que se representa a continuación se induciría un ciclo recursivo el cual provocaría un desbordamiento del buffer de memoria existente en el servidor de aplicaciones *Web*. Por tal motivo es de vital importancia representar gráficamente el mismo con el objetivo de evitar comportamientos no deseados, de esta forma sería más fácil de detectar.

Grafo de dependencia que provoca ciclo infinito:

- A → B
- B → C
- C → A

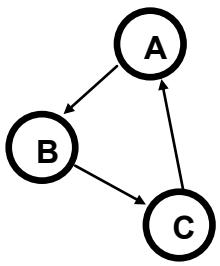


Figura 14: Grafo de dependencia cíclica.

La segunda característica es que el vínculo se crea de forma unidireccional, en otras palabras si A → B implica que una vez que se ejecute A se invoca B pero no de forma viceversa.

Sintaxis:

$this->linker->connect(signals, class, slots, class, option, handle);

- signals : Función que activa el evento
- class : Clase en la que se encuentra definida la signals
- slots : Función dependiente del evento
- class : Clase en la que se encuentra definida el slots
- option : Modo de ejecución ***pre|pos*** (*antes de | después de)*
- handle : Función intermediaria

Ejemplo:

1. En caso de que se conozca estáticamente la dependencia de los *plugins* se podrá establecer los enlaces en el propio constructor del *Main,* en este ejemplo se procederá a:

    - Conectar como poscondición la función *update* del *plugin Map* con la señal *update* del *plugin ThematicRaster.*
    - Conectar como precondición la función *load* del *plugin Layer* con la señal *update* del *plugin MapTemplate.*
    - Conectar como poscondición la función *update* del *plugin Map* con la señal *update* del *plugin Layer.*

```
1.  class Main extends App implements Security
2.  {
3.   public function __construct()
4.   {
5.       parent::__construct();
6.       $this->linker->connect("update", "ThematicRaster",  "update", "Map", 'pos', null);
7.       $this->linker->connect("update", "MapTemplate",  "load", "Layer", 'pre', null);
8.       $this->linker->connect("update", "Layer",  "update", "Map",  'pos', null);
9.   }
10.  }
```

Como bien se puede apreciar en el ejemplo anterior esta definición es "*estática*", teniendo en cuenta que en caso que se desee modificar dicha configuración se debe acceder directamente en el código fuente de la aplicación. Teniendo en cuenta dicha necesidad esta librería provee de un recurso que permite a través de un fichero de configuración definir cómo estarían interrelacionados los diferentes módulos de la misma.

Sintaxis:

$this->linker->**load** ($file);

- **file:** Nombre o identificador del fichero contenedor de la configuración que permite la conexión dinámica entre *plugins* ubicado en el directorio definido para el almacenamiento de este tipo de archivos que por defecto se asume como *app/config*, en caso de omitirse este buscaría la aparición de la variable con índice *$config["linker"]* en el fichero *conf.php,* en caso de no estar definida en la misma o de no existir dicho fichero la buscaría en el archivo denominado *linker.php.*

Ejemplos:

2. Cargar la interrelación entre módulos de la aplicación desde el fichero *conf.php*, esto lo que trae como consecuencia es un aumento en memoria, partiendo del hecho de que en más de una ocasión sería necesaria el empleo de los datos de configuración global que en este subyacen siendo innecesaria la carga de los datos del linker.

```
11. class Main extends App
12. {
13.     public function __construct()
14.     {
15.       parent::__construct();
16.       $this->linker->load();
17.     }
```

```
18. }
```

Fichero *conf.php*

```
19. <?php
20.         $config["db"]["server"] = 'localhost';
21.         $config["db"]["name"]   = 'work';
22.         $config["db"]["port"]   = '5432';
23.         $config["db"]["user"]   =  'postgres';
24.         $config["db"]["passw"]  = 'postgres';
25.
26.         $config["linker"]["pos"]["ServerAdmin"]["doIt"][] = array(
27.               "handle" => 0, "slot" => "hola", "class" => "ToolsAdmin"
28.         );
29.         $config["linker"]["pos"]["ThematicRaster"]["update"][] = array(
30.               "handle" => 0, "slot" => "update", "class" => "Map"
31.          );
32.         $config["linker"]["pos"]["Rutas"]["showRegistro"][]              = array(
33.               "handle" => 0, "slot" => "admninRout", "class" => "Rutas"
34.         );
35.         return $config;
36. ?>
```

3. En función de minimizar el costo de consumo por concepto de complejidad temporal, teniendo en cuenta la carga de información innecesaria, se podría separar dicha información en otro fichero dedicara para este fin, tal y como aparece a continuación:

```
1.  Class Main extends App
2.  {
3.      public function __construct()
4.      {
5.         parent::__construct();
6.         $this->linker->load();
7.      }
8.  }
```

Fichero *conf.php*

```
9.  <?php
10.         $config["db"]["server"] = 'localhost';
11.         $config["db"]["name"]   = 'work';
12.         $config["db"]["port"]   = '5432';
13.         $config["db"]["user"]   = 'postgres';
14.         $config["db"]["passw"]  = 'postgres';
15.         return $config;
16. ?>
```

Fichero *linker.php*

```
17. <?php
18.     $config["linker"]["pos"]["ServerAdmin"]["doIt"][] = array(
19.               "handle" => 0, "slot" => "hola", "class" => "ToolsAdmin"
20.     );
21.     $config["linker"]["pos"]["ThematicRaster"]["update"][] = array(
22.               "handle" => 0, "slot" => "update", "class" => "Map"
```

```
        );
        $config["linker"]["pos"]["Rutas"]["showRegistro"][] = array(
                "handle" => 0, "slot" => "admninRout", "class" => "Rutas"
        );
        return $config;
?>
```

4. Estos recursos tanto la carga desde un fichero de configuración externa o a través de la función *connect* no son solapados el uno por el otro, por tanto pueden ser utilizados de conjunto, permitiendo definir de forma "estática" ciertos niveles de dependencia y "dinámicamente" otros. En este caso se define el fichero contenedor de la configuración por el nombre *linkconf.php*.

```
class Main extends App
{
  public function __construct()
  {
        parent::__construct();
        $this->linker->load("linkconf");
        $this->linker->connect("setDat", "ThematicRaster", "getInf", "Map", 'pre', null);
        $this->linker->connect("getInst", "MapTemplate", "load", "Layer", 'pos', null);
  }
}
```

Fichero *conf.php*

```
<?php
        $config["db"]["server"] = 'localhost';
        $config["db"]["name"]   = 'work';
        $config["db"]["port"]   = '5432';
        $config["db"]["user"]   = 'postgres';
        $config["db"]["passw"]  = 'postgres';
        return $config;
?>
```

Fichero *linkconf.php*

```
<?php
        $config["linker"]["pos"]["ServerAdmin"]["doIt"][] = array(
            "handle" => 0, "slot" => "hola", "class" => "ToolsAdmin"
        );
        $config["linker"]["pos"]["ThematicRaster"]["update"][] = array(
            "handle" => 0, "slot" => "update", "class" => "Map"
        );
        $config["linker"]["pos"]["Rutas"]["showRegistro"][] = array(
            "handle" => 0, "slot" => "admninRout", "class" => "Rutas"
        );
        return $config;
?>
```

4.6 Log

Durante el desarrollo y la instalación de las aplicaciones, los programadores necesitan toda la información posible para determinar si la aplicación está funcionando como debería. Normalmente, esta

información se obtiene mediante los archivos de *log* y las herramientas de depuración o *debug*. Los *framework* como *Symfony* son el núcleo de las aplicaciones, por lo que es esencial que el propio *framework* disponga de las herramientas necesarias para asegurar un desarrollo eficiente de las aplicaciones. *Ksike* no intenta desechar esta teoría sino todo lo contrario es por ello que provee una librería que permite la administración de un recurso tan valioso como este.

PHP dispone de una directiva llamada *error_reporting*, que se define en el archivo de configuración php.ini, y que especifica los eventos de *PHP* que se guardan en el archivo de log. Partiendo de esto en *Ksike* se decide generar log de errores para php en un directorio denominado *php_error* organizándolo por las diferentes categorías que el mismo intérprete define para ello, tales como: *E_ERROR, E_WARNING, E_PARSE, E_NOTICE, E_CORE_ERROR, E_CORE_WARNING, E_COMPILE_ERROR, E_COMPILE_WARNING, E_RECOVERABLE_ERROR, E_USER_DEPRECATED, E_USER_ERROR, E_USER_ERROR, E_USER_WARNING, E_USER_NOTICE, E_DEPRECATED, E_STRICT.*

De igual forma es posible definir las llamadas trazas o *log* desde cualquier entorno dentro de *Ksike*, entiéndase por ello, los *plugins*, *app*, *driver* o cualquier librería que se emplee dentro del *API* servidora que el mismo provee. A continuación se exponen las sintaxis que permiten lograr tales objetivos.

Sintaxis:

Log::**save**($struct, $name, $path);

- **struct:** Corresponde a la estructura de datos que se desea almacenar en forma de log, esta debe ser un arreglo asociativo, donde el índice determina la característica a cual hace referencia su valor
- **name:** Se refiere al identificador del fichero que será generado o al cual se le adicionará la traza que se desea almacenar.
- **path:** Indica la ruta relativa en la cual se encuentra el fichero de log en cuestionamiento.

Por defecto esta librería trae definido un formato para la generación de trazas a partir de la estructura organizativa que contiene la información a almacenar. La cual se refleja a continuación:

- propiedad1 **:** << valor1 >>**,** propiedad2 **:** << valor2 >>**,** propiedad **:** << valorn >>

Teniendo en cuenta que la estructura organizativa no es más que un arreglo asociativo, entonces se debe asumir que la propiedad corresponde al índice y el valor sería evidentemente el resultado almacenado que le corresponde a la misma.

Sin embrago lo expresado anteriormente no limita a que los desarrolladores tengan que asumirla, teniendo que interpretar los registros de *log* basados en una estructura rígida, que en determinado contexto resulte en extremo tedioso o ineficiente, así que partiendo de dicha necesidad se provee de un mecanismo que permite modificarlo de forma dinámica. Cuidado, una vez modificada esta todas las llamadas posteriores a la misma se generarían de la misma forma, recuerden que esta librería se comporta como un singleton, a continuación se presenta la sintaxis y para mayor comprensión dirigirse al ejemplo #2 de esta sección.

Sintaxis:

Log::**$format** = $newForma;

- **newForma:** Constituye un arreglo asociativo que debe contener 3 índices, los cuales se explican a continuación:
  - **in:** Índice que define el carácter inicio de la descripción para

determinada característica o propiedad dentro del *log*.

- **out:** Índice que define el carácter fin de la descripción para determinada característica o propiedad dentro del *log*.
- **|:** Índice que define el carácter que hace la función de separador entre una característica o propiedad de su adyacente.

Independientemente de la estructura organizativa definida para el almacenamiento de una traza o *log,* siempre se incluye la propiedad *Date,* la cual hace referencia a la fecha y hora en que se invocó dicho procedimiento. Teniendo en cuenta que el propio intérprete de *php* da soporte para una serie de formatos de salida para representar la fecha y hora, se decidió proveer un mecanismo que haga posible la gestión en tiempo de ejecución de esta propiedad.

Por defecto el formato implícito es: *Y/m/d H:i:s*, el cual corresponde a año/mes/día seguido de la hora:minutos:segundos, de hecho un ejemplo práctico podría ser: *2011/02/27 22:35:56.* A continuación se muestra la sintaxis que hace posible resolver esta problemática, para mayor comprensión consulte el ejemplo #3:

Sintaxis:

Log::**$dateFormat** = $newDateFormat;

- **newDateFormat:** Constituye una cadena en formato de texto que define la estructura organizativa en la cual será representada los datos de la fecha y hora, para determinado *log*.

Por otra parte el formato o extensión en la cual los ficheros de *log* o trazas son generados también es modificable. Por defecto asume valor .log, sin embargo a través de la sintaxis que aparece a continuación eso podría ser variado:

Sintaxis:

Log::**$ext**= $newExt;

- **newExt:** Constituye una cadena en formato de texto que define la extensión del archivo generado, para determinado *log*:

Pese a que estos valores pueden ser modificados a través de las distintas vías anteriormente reflejadas, es posible encapsular toda esta configuración en un método que lo resuma todo, en caso de que se omitan algunos de lo parámetros entonces serían restablecidos al mismo estado en que se encontraban al inicio de su definición. De esta forma se pueden evitar comportamientos no deseados a partir de un control inadecuado de este recurso, para mayor comprensión véanse los ejemplos 4 y 5.

Sintaxis:

Log::setFormat($format, $dateFormat="trace", $ext='../log/')

- **format:** Constituye un arreglo asociativo que debe contener 3 índices explicados anteriormente, por defecto toma valor *array('in'=>': <<', 'out'=>'>>', '|'=>', ')*.
- **dateFormat**: Constituye una cadena en formato de texto que define la estructura organizativa en la cual será representada los datos de la fecha y hora, para determinado *log*, por defecto toma valor *'Y/m/d H:i:s'*.
- **ext:** Constituye una cadena en formato de texto que define la extensión del archivo generado, para determinado *log*, por defecto toma valor *.log*

Ejemplos:

1. Generar un *log* en el directorio *../log/php_error/* con el nombre de *E_WARNING* y extensión *.log* tomando como formato de traza el que trae por defecto la propia librería, permitiendo especificar que se detectó un error en determinado fichero incluyendo la línea en que él mismo fue lanzado.

```
37. $errfile = '/var/www/demo/core/php/FrontController.php';
38. $errline = 62;
39. $errstr  = 'call_user_func_array() expects parameter 1 to be a valid callback';
40. $conten = array(
41.     'File'=> $errfile,
42.     'Line'=> $errline,
43.     'Description'=> $errstr
44. );
45. $name = 'E_WARNING';
46. $path = '../log/php_error'
47. Log::save($conten, $name, $path);
48.
```

Fichero *../log/php_error/E_WARNING.log*

```
1. Date: <<2011/02/27 22:35:56>>, File: <</var/www/demo/core/php/FrontController.php>>, Line: <<62>>, Description:
   <<call_user_func_array() expects parameter 1 to be a valid callback>>
```

2. Retomando el ejemplo #1, se desea incorporar al mismo fichero de log otra traza con el mismo objetivo, pero con la particularidad de que el formato de salida no sería el mismo, en este caso se necesitaría que la información quede recogida entre los caracteres ": #" y "#", teniendo como separador "| ".

```
49. Log::$format = array(
50.     'in'=>': #',
51.     'out'=>'#',
52.     '|'=>'| '
53. );
54. $errfile = '/var/www/demo/core/php/FrontController.php';
55. $errline = 73;
56. $errstr  = 'class "Map" does not have a method getMap?outInfo=false';
57. $conten = array(
58.     'File'=> $errfile,
59.     'Line'=> $errline,
60.     'Description'=> $errstr
61. );
62. $name = 'E_WARNING';
63. $path = '../log/php_error'
64. Log::save($conten, $name, $path);
65.
```

Fichero *../log/php_error/E_WARNING.log*

```
1. Date: <<2011/02/27 22:35:56>>, File: <</var/www/demo/core/php/FrontController.php>>, Line: <<62>>, Description:
   <<call_user_func_array() expects parameter 1 to be a valid callback>>
2. Date: #2011/02/27 21:35:56#| File: #/var/www/demo/core/php/FrontController.php#| Line: #73#| Description: #class "Map" does
   not have a method getMap?outInfo=false#
```

3. Retomando el ejemplo #2, se desea incorporar al mismo fichero de log otra traza con el mismo objetivo, pero con la particularidad que el formato de salida para la propiedad fecha no sería la

que dicha librería trae por defecto.

```
66. Log::$dateFormat= 'l dS \of F Y h:i:s A';
67. $errfile = '/var/www/demo/core/php/FrontController.php';
68. $errline = 78;
69. $errstr  = 'expects parameter 1 to be a valid callback';
70. $conten = array(
71.     'File'=> $errfile,
72.     'Line'=> $errline,
73.     'Description'=> $errstr
74. );
75. $name = 'E_WARNING';
76. $path = '../log/php_error'
77. Log::save($conten, $name, $path);
78.
```

Fichero ../log/php_error/*E_WARNING.log*

```
1. Date: <<2011/02/27 22:35:56>>, File: <</var/www/demo/core/php/FrontController.php>>, Line: <<62>>, Description: <<call_user_func_array() expects parameter 1 to be a valid callback>>
2. Date: #2011/02/27 21:35:56#| File: #/var/www/demo/core/php/FrontController.php#| Line: #73#| Description: #class “Map” does not have a method getMap?outInfo=false#
3. Date: #Monday 28th of August 2011 03:12:46 PM#| File: #/var/www/demo/core/php/FrontController.php#| Line: #73#| Description: #expects parameter 1 to be a valid callback #
```

4. Retomando el ejemplo #3, se desea incorporar al mismo fichero de log otra traza con el mismo objetivo, pero con la particularidad que la extensión del archivo de salida no sería la que dicha librería trae por defecto, esto traería como consecuencia que se generaría un archivo nuevo.

```
79. Log::setFormat(
80.     array(
81.         'in'=>': #',
82.         'out'=>'#',
83.         '|'=>'| '
84.     ),
85.     'l dS \of F Y h:i:s A',
86.     '.inf',
87. );
88. $errfile = '/var/www/demo/core/php/FrontController.php';
89. $errline = 78;
90. $errstr  = 'expects parameter 1 to be a valid callback';
91. $conten = array(
92.     'File'=> $errfile,
93.     'Line'=> $errline,
94.     'Description'=> $errstr
95. );
96. $name = 'E_WARNING';
97. $path = '../log/php_error'
98. Log::save($conten, $name, $path);
99.
```

Fichero ../log/php_error/*E_WARNING.inf*

```
1. Date: #Monday 28th of August 2011 03:12:46 PM#| File: #/var/www/demo/core/php/FrontController.php#| Line: #73#| Description: #expects parameter 1 to be a valid callback #
```

5. Retomando el ejemplo #3, se desea incorporar al mismo fichero de log otra traza con el mismo

objetivo, pero en este caso se restauraría la configuración inicial para tal objetivo.

```
100.  Log:: setFormat();
101.  $errfile = '/var/www/demo/core/php/FrontController.php';
102.  $errline = 80;
103.  $errstr  = 'expects parameter 2 to be a valid callback';
104.  $conten = array(
105.     'File'=> $errfile,
106.     'Line'=> $errline,
107.     'Description'=> $errstr
108.  );
109.  $name = 'E_WARNING';
110.  $path = '../log/php_error'
111.  Log::save($conten, $name, $path);
112.
```

Fichero ../log/php_error/*E_WARNING.log*

1. **Date**: <<2011/02/27 22:35:56>>, **File**: <</var/www/demo/core/php/FrontController.php>>, **Line**: <<62>>, **Description**: <<call_user_func_array() expects parameter 1 to be a valid callback>>
2. **Date**: #2011/02/27 21:35:56#| **File**: #/var/www/demo/core/php/FrontController.php#| **Line**: #73#| **Description**: #class "Map" does not have a method getMap?outInfo=false#
3. **Date**: #Monday 28th of August 2011 03:12:46 PM#| **File**: #/var/www/demo/core/php/FrontController.php#| **Line**: #73#| **Description**: #expects parameter 1 to be a valid callback #
4. **Date**: <<2011/06/29 22:35:56>>, **File**: <</var/www/demo/core/php/FrontController.php>>, **Line**: <<80>>, **Description**: <<expects parameter 2 to be a valid callback>>

4.7 Interfaces

Las interfaces constituyen otros de los mecanismos proveídos por este *framework* para extender las funcionalidades o comportamientos tanto para los módulos como para la aplicación en sentido general. Este recurso es muy utilizado en lenguajes como *C#* y *Java*, de hecho es la vía que escogieron para solventar los conflictos de herencia múltiple que podrían darse en lenguajes como *C++* partiendo de una mala base arquitectónica de los productos finales.

4.7.1 Seguridad

Uno de los elementos más importantes en el desarrollo de aplicaciones *web* es la correcta implementación de una capa de seguridad. De forma tal que sea posible mantener la integridad y confiabilidad de información que será representada gráficamente sobre la misma, así como asegurar la disponibilidad de los servicios publicados en el producto final.

En la primera versión de este *framework* no se incorpora ningún módulo o driver para gestionar la seguridad a nivel de aplicación. Este deja dicho recurso a decisión de los desarrolladores que lo utilicen, permitiendo que estos puedan emplear algunos subsistemas de gestión de seguridad implementada e incorporárselos al proyecto de *Ksike* en forma de driver o por inclusión.

No obstante el controlador frontal correspondiente a la *API* del lenguaje *PHP5* si provee de una interfaz en la que se deben redefinir dos métodos principales que permite gestionar la seguridad a nivel de acción por módulos.

Sintaxis:

Security**::allow** ($certy, $acction);

- **$certy**: Esta propiedad hace referencia al certificado o credencial que identifica al usuario que esta logueado en la sesión actual y está intentando de acceder a la acción *acction*.
- **acction**: Esta propiedad define el identificador de la función que se está accediendo en ese instante de tiempo.

Una vez realizada la validación de si determinado usuaria tiene los privilegios suficientes para invocar determinada acción y en caso de que sea denegado se invocaría la función *Security::allowFailed($certy, $acction)*

Sintaxis:

Security**::allowFailed** ($certy, $acction);

- **$certy**: Esta propiedad hace referencia al certificado o credencia que identifica al usuario que esta logueado en la sesión actual y está intentando de acceder a la acción *acction*.
- **acction**: Esta propiedad define el identificador de la función que se está accediendo en ese instante de tiempo.

Ejemplos:

1. A continuación se representa un ejemplo de cómo se gestionaría la seguridad en el módulo servidor del *plugin Route*

```
1.  <?php
2.      class Route extends Plugin implements Security
3.      {
4.          public function __construct() {}
5.
6.          public function reedRegistro($params)
7.          {
8.              $confobj = DriverManager::factory("ConfigManager");
9.              $mainconf = $confobj->loadConfig("Main");
10.             $rutaconf = $confobj->loadConfig();
11.
12.             $dbf = DriverManager::factory("PgSQL", $mainconf["db"]["name"]);
13.             $result = $dbf->executeSQL("SELECT nombre, edad, ci FROM person",1);
14.
15.             return array(
16.                 'first'=>$rutaconf["tree"],
17.                 'last'=>$rutaconf["node"],
18.                 'file'=>$dataf,
19.                 'dbout'=>$result
20.             );
21.         }
22.
23.         public function showRegistro($params)
```

```
24.             {
25.                     $vector = new Vector();
26.                     return array(
27.                             'first' =>$params->first,
28.                             'last'  =>$params->last,
29.                             'rgb'   =>$ vector->getRaster()->getRGB()
30.                     );
31.             }
32.
33.             public function admninRout()
34.             {
35.                     return "info en texto plano";
36.             }
37.
38.             public function allow($certy, $acction)
39.             {
40.                     switch($acction)
41.                     {
42.                             case "admninRout" : return true; break;
43.                             case "showRegistro" :
44.                                     if($certy == "65645645")
45.                                             return true;
46.                                     else return false;
47.                             break;
48.                             case "reedRegistro" :
49.                                     if($certy == "232324877")
50.                                             return true;
51.                                     else return false;
52.                             break;
53.                             default: return false; break;
54.                     }
55.             }
56.
57.             public function allowFailed($certy, $acction) {
58.                     return "El usuarion {$certy} No está autorizado a ejecutar {$ acction}";
59.             }
60.     }
61. ?>
```

Uno de los modelos más utilizados para lograr sistemas seguros es la implementación del *AAA*[14]. El cual tiene como base los siguientes conceptos:

- ✓ La **autenticación** es el proceso por el que una entidad prueba su identidad ante otra. Se consigue mediante la presentación de una propuesta de identidad y la demostración de estar en posesión de las credenciales que permiten comprobarla.

- ✓ La **autorización** se refiere a la concesión de privilegios específicos a una entidad o usuario

[14] **AAA** acrónimo de Autenticación, Autorización y Contabilización. En no pocas ocasiones se combina el término con auditoria, convirtiéndose entonces en *AAAA*.

basándose en su identidad autenticada, los privilegios que solicita, y el estado actual del sistema. Pueden también estar basadas en restricciones, tales como horarias, sobre la localización de la entidad solicitante, la prohibición de realizar logins múltiples simultáneos del mismo usuario, etc. La mayor parte de las ocasiones el privilegio concedido consiste en el uso de un determinado tipo de servicio.

- ✓ La **contabilización** se refiere al seguimiento del consumo de los recursos de red por los usuarios. Esta información puede usarse posteriormente para la administración, planificación, facturación, u otros propósitos. La contabilización en tiempo real es aquella en la que los datos generados se entregan al mismo tiempo que se produce el consumo de los recursos. En contraposición la contabilización por lotes consiste en la grabación de los datos de consumo para su entrega en algún momento posterior. La información típica que un proceso de contabilización registra es la identidad del usuario, el tipo de servicio que se le proporciona, cuando comenzó a usarlo, y cuando terminó.

- ✓ La **auditoría** consiste en recoger, agrupar y evaluar evidencias para determinar si un sistema salvaguarda determinado activo, mantiene la integridad de los datos y utiliza eficientemente los recursos, cumpliendo de esta forma con las leyes y regulaciones establecidas. Permiten detectar de forma sistemática el uso de los recursos y los flujos de información dentro de una organización y determinar qué información es crítica para el cumplimiento de su misión y objetivos, identificando necesidades, duplicidades, costes, valor y barreras, que obstaculizan flujos de información eficientes.

Si se lleva a cavo un análisis detallado de los elementos que componen este apartado, se podrán identificar ciertos mecanismos que permiten implementar dicho modelo de seguridad. La autenticación es el modo más simple y puede ser implementado empleando los recursos del propio del lenguaje *php* tales como sesiones por usuarios. La autorización puede ser garantizada a través de la implementación de la interfaz *Security,* garantizando la veracidad de la aplicación de los patrones *GRASP*[15]*,* teniendo en cuenta que los módulos son los responsables de gestionar su propia seguridad. Por otra parte tanto la contabilización como la auditoría pueden ser solventadas a través del recurso *Log* descrito en la sección 4.6, permitiendo generar trazas que garanticen dichos principios.

4.8 Configuración

De la misma forma que se expresaba la definición de valores significativos que podían ser modificables en función de determinados proyectos. El módulo del servidor requiere de algunos ficheros de configuración que le permita identificar qué proyecto es el que está en ejecución y cuál es su distribución física.

El *framework Ksike* define en el módulo servidor una serie de constantes que definen el comportamiento a lo largo de la ejecución del mismo, estas son las que aparecen a continuación:

- ✓ **STD_EXT**: Esta propiedad hace referencia al tipo de extensión permitida para los ficheros escritos en el lenguaje que se ejecuta el servidor de aplicaciones *web*, por defecto toma valor *.php*
- ✓ **STD_NEXT**: Constituye variaciones no permitidas de la extensión anteriormente definida, valor por defecto que toma es: *.php~*
- ✓ **STD_MAIN**: Esta propiedad hace referencia al identificador del módulo principal que se debe encontrar en el directorio app, por defecto toma valor: *Main*
- ✓ **STD_CORE**: Define el directorio en que se encontrara ubicado el núcleo del *framework*, por

[15] **GRASP** acrónimo de General Responsibility Assignment Software Patterns

defecto toma valor: *core/php/*

- ✓ **STD_PLUGINS**: Define el directorio en que se encontrarán ubicados los módulos o *plugin* que conforman la arquitectura de componente para determinado proyecto, por defecto toma valor: *plugins/*
- ✓ **STD_LIBS**: Define el directorio en que se encontrarán ubicados las librerías externas a la plataforma, esto incluye a los drivers, valor por defecto que toma es: *lib/*
- ✓ **STD_APP**: Define el directorio en que se encontrara ubicado el módulo principal del proyecto, por defecto toma valor: *app/*
- ✓ **STD_LOG**: Define el directorio en que se encontrarán ubicados las trazas concernientes al funcionamiento del core del framework, valor por defecto que toma es: *log/*
- ✓ **STD_INCLUDE**: Define el directorio en que se encontrarán ubicadas aquellas librerías que serán cargadas dinámicamente, por defecto toma valor*:* *server/include/*
- ✓ **STD_INTERFACES**: Define el directorio en que se encontrarán ubicadas las interfaces que podrán implementar tanto los módulos como la *app*, por defecto toma valor*:* *interfaces/*
- ✓ **STD_PHPERROR**: Define el directorio en que se encontrarán ubicados los log de errores provenientes del lenguaje php, por defecto toma valor*:* *php_error/*
- ✓ **STD_ACCESS:** Esta propiedad define la identidad del framework, por defecto toma valor*:* *Ksike*

Cada una de estas constantes pueden ser redefinidas en dependencia del proyecto a desarrollar aquellas que sean omitidas entonces tomarán los valores que se presentaron como que asumen por defecto.

Ejemplo:

1. Fichero de configuración que refleja los valores tomados por defecto, para cualquier proyecto.

```
1.  define('STD_EXT',         '.php');
2.  define('STD_NEXT',        '.php~');
3.  define('STD_MAIN',        'Main');
4.  define('STD_CORE',        'core/php/');
5.  define('STD_PLUGINS',     'plugins/');
6.  define('STD_LIBS',        'lib/');
7.  define('STD_APP',         'app/');
8.  define('STD_CONFIG',      'config/');
9.  define('STD_LOG',         'log/');
10. define('STD_INCLUDE',     'server/include/');
11. define('STD_INTERFACES',  'interfaces/');
12. define('STD_PHPERROR',    'php_error/');
13. define('STD_ACCESS',      'Ksike');
```

2. En consecuencia con el ejemplo presentado en el módulo cliente este ilustra el fichero de configuración para el proyecto nombrado *ToolDevelop*, el cual se encuentra en una ruta relativa de acceso hacia el core compartido identificada por *../../* y como elemento de direccionamiento desde el núcleo al mismo *tools/ToolDevelop/*.

```
14. define('STD_PLUGINS',     'tools/ToolDevelop/plugins/');
15. define('STD_LIBS',        'lib/');
16. define('STD_APP',         'tools/ToolDevelop/app/');
```

Nótese que en el caso anterior solo se redefinen los directorios de *plugin* y *app* especificando que estos pertenecen al directorio raíz del proyecto, siendo todo lo contrario para el directorio *libs* que será

tomado como el general para otros proyectos.

5. Seguridad

Entiendase por seguridad como la ausencia de riesgo. Por tanto se basa en la protección de activos. Estos pueden ser elementos tan tangibles como un servidor de aplicaciones o de bases de datos. Generalmente es posible evaluar el nivel de protección para determinado ente en base a tres aspectos principales tales como: integridad, disponibilidad, confidencialidad.

5.1 Amenazas

Son múltiples los ataques externos a los que puede estar expuesto un sistema desarrollado para entornos *web*. Estos son usualmente clasificados en seis categorías principales, las cuales se detallan a continuación especificando los mecanismos empleados en función de lograr sus objetivos propuestos.

- ✓ **Autenticación**: Dirigidas a explotar el método de validación para la identidad de un usuario, servicio o aplicación.
  - Fuerza Bruta
  - Autenticación insuficiente
  - Débil validación de recuperación de *password* o contraseña

- ✓ **Autorización**: Explotan el mecanismo de un sitio *web* para determinar si un usuario o servicio tiene los permisos necesarios para ejecutar una acción.
  - Predicción de Credenciales o Sesión
  - Autorización insuficiente
  - Expiración de Sesión insuficiente
  - Fijado de Sesión

- ✓ **Ataques Lógicos:** Explotan la lógica de la aplicación, empleando el flujo procedural utilizado por la aplicación para efectuar cierta operación.
  - Abuso de funcionalidad
  - Denial of Service
  - Insuficiente Anti-Automatismo
  - Insuficiente validación de procesos
  - Manipulación de entradas

- ✓ **Ataques al cliente:** Atacan al usuario de la aplicación.
  - Content Spoofing
  - Cross-Site Scripting

- ✓ **Ejecución de comandos:** Ataques diseñados para ejecutar comandos remotos en el servidor.
  - Buffer Overflow
  - Format String
  - LDAP Injection
  - Ejecucuón de Comandos
  - SQL Injection
  - SSI Injection
  - XPath Injection

- ✓ **Robo de Información**: Ataques que apuntan a adquirir información específica sobre el sitio *web*.
  - Indexado de directorio

- Caminos transversales
- Predicción de ubicación de recursos
- Escape de información

En sentido general cada una de estas vulnerabilidades logran de una forma u otra un comportamiento no deseado en los productos finales. Por tanto diseñar un plan de mitigación es de vital importancia y será objeto de seguimiento por el equipo de desarrollo de *Ksike,* en función de incorporar mecanismos que abstraigan a los desarrolladores a este tipo de problemas. A continuación se describen los ataques considerados críticos y que son empleados con mayor frecuencia.

5.1.1 SQL Injection

SQL Injection es una vulnerabilidad que afecta aplicaciones a nivel de base de datos. Esta consiste en enviar instrucciones descritas en el lenguaje *SQL* adicionales a partir de parámetros entrada ingresados por el usuario. Al inyectar el código malicioso dentro de estos campos, el *script* invasor se ejecuta dentro del propio de la aplicación con el objetivo de alterar su funcionamiento normal, de acuerdo con el propósito del atacante.

Es un problema de seguridad que debe ser tomado en cuenta por el programador para prevenirlo. La vulnerabilidad ocurre cuando la aplicación ejecuta una sentencia *SQL* que utiliza el valor de campos de entrada sin validarlos correctamente. Permiten al atacante saltar restricciones de acceso, elevar privilegios del usuario, extracción de información de la base de datos, ejecución de comandos dentro del servidor. Incluso es posible destruir parte la base de datos de la aplicación.

Los mensajes de error a menudo revelan demasiada información que puede ser útil al atacante, por tanto no se deberá exponer al usuario final a este tipo de notificaciones, principalmente las devueltas por el gestor de la base de datos. Estos deberían ser notificados solamente a los administradores de la aplicación.

Un mecanismo muy empleado en la construcción de aplicaciones *web* principalmente orientadas a la gestión, es el empleo de *framework* enfocados en el tratamiento del acceso a datos tales como *ORM*, permitiendo tener un control prácticamente absoluto de los mismos. Estos incluyen mecanismos validadores certificados por la comunidad internacional.

5.1.2 Cross Site Scripting (XSS)

Esta amenaza surge de los riesgos inherentes de permitir a un servidor web enviar código a un *browser*. Cuando a un *script* de una página se le permite acceder a datos de otra página u objeto, existe la posibilidad de que un sitio *web* malicioso, lo utilice para obtener información confidencial. Aprovechando de esta forma las vulnerabilidades en los mecanismos de validación de interacción entre páginas y objetos y en lenguajes scripting que ejecutan en el cliente.

Existe tres tipos conocidos que vulnerabilidades que implican:

- ✓ **Tipo 1 – XSS local**: En esta categoría el problema reside en el propio script del lado del ciente de la página. Por ejemplo si un fragmento de JavaScript accede un parámetro de un request HTML y lo utiliza para escribir HTML en su propia página, no codificándolo como entidades HTML, se presenta la vulnerabilidad. Estos datos serán reinterpretados por los browsers como HTML, que podría incluir código adicional (maligno).

- ✓ **Tipo 2 – XSS reflejado**: Este es el tipo más común, también es conocido como no-persistente. La vulnerabilidad se presenta cuando los datos provistos por el cliente es

utilizada inmediatamente por scripts del lado del servidor para generar la página de resultados. Si datos no validados provistos por el cliente son incluidos en la página de resultados sin una codificación apropiada, es posible insertar código desde el lado del cliente. Basta con un poco de ingeniería social para llevar a un usuario desprevenido a seguir un link malicioso que inserte este código en la página de resultados, obteniéndose acceso total a su contenido.

- ✓ **Tipo 3 – XSS persistente**: Este tipo incluye los ataques más poderosos. La vulnerabilidad existe cuando los datos provistos por el cliente a la aplicación es almacenada persistentemente en el servidor, y accesible a varios usuarios del sitio sin codificación de entidades *HTML* – por ejemplo message boards. El atacante puede insertar un script una única vez, y con esto le basta para alcanzar a un gran número de usuarios, sin requerir mucho esfuerzo de ingeniería social. Los métodos de inserción son variados y no es necesario utilizar la aplicación misma para explotar la vulnerabilidad.

Preferentemente no debería permitirse código *HTML* en los campos de entrada, en caso de que se necesitara, deberían de validarse los tags, descartando aquellos que pudieran ser peligrosos, adicionalmente también se deben descartar caracteres potencialmente peligrosos como -,",",?,&,<,>,etc.

5.1.3 Manipulación de parámetros

Es un conjunto de técnicas que atacan la lógica de negocio de la aplicación, tomando ventaja del uso de campos ocultos o fijos para la transferencia de información sensible entre browser y servidor. En particular, tags ocultos en un form, *cookies* o parámetros anexados a la *URL* son fácilmente modificables por un atacante.

- ✓ **Manipulación de campos ocultos**: Cualquiera de los valores que se almacenan en los campos de un form pueden ser manipulados por un atacante. En particular los campos ocultos son usualmente atractivos para su manipulación ya que muchos desarrolladores los utilizan para información confidencial de estatus de algún objeto sobre el que se está trabajando. La manipulación de estos campos es tan simple como salvar la página, editar el valor de estos campos en su código *HTML* y recargarla en el *browser*.

- ✓ **Manipulación de URL**: Los forms *HTML* envían sus resultados usando uno de dos métodos posibles: *GET* o *POST*. Si el método es *GET*, todos los parámetros del *form* y sus valores correspondientes aparecen en cadena de búsqueda del siguiente *URL* que el usuario ve. Esta cadena puede ser fácilmente manipulable.

5.1.4 Ejecución de comandos

Las técnicas de manipulación de entrada vistas son sólo algunas que llevan a la posibilidad de ejecutar remotamente comandos del Sistema Operativo de la víctima. Determinados caracteres especiales pueden ser interpretados por *scripts* de validación poco seguros como una instrucción al Sistema Operativo de esperar un comando arbitrario a continuación. En particular, el punto y coma o la barra | son en *Unix* caracteres que permiten encadenar comandos. Por tanto incluir en el campo de entrada un **;** seguido del comando que se desea ejecutar puede tener éxito si el mecanismo de validación no es lo bastante robusto.

En la mayoría de los ataques de inyección de comandos, es necesario tener conciencia de que todo dato hacia y desde un *browser* puede ser modificado. Por ende, la validación propiamente dicha de cada dato de entrada debe darse en el servidor, fuera del control del usuario. Además el servidor debería configurarse para requerir una autenticación debida al nivel del directorio para cada archivo en el mismo. Aún así esta solución no es perfecta, por lo que es conveniente que el usuario utilizado por la aplicación sobre el servidor tenga los permisos mínimos necesarios, para minimizar el posible impacto.

5.2 Mitigación

Como bien se pudo apreciar en el acápite anterior y aunque se cubrió tan solo una pequeña parte del tema, fue suficiente para comprobar la facilidad con que una aplicación puede ser vulnerable, cuando no se le asigna una prioridad adecuada a los controles de seguridad en las distintas etapas de desarrollo.

Ksike no provee un componente completo para salvaguardar una aplicación *web*. Aunque el término anterior parezca un poco crudo, la realidad es que muy pocas tecnologías logran dicho objetivo. Uno de los mecanismo más empleado por la gran mayoría de los *framework* orientados al desarrollo de este tipo de aplicaciones es la definición de un directorio de publicación, donde se encontrarán todos aquellos ficheros que obligatoriamente debes ser entregados a los clientes, tales como archivos *js, html, css, imágenes*. De esta forma podrán ser "protegidos" los demás elementos que conforma el producto, sin embargo la limitación de acceso no hace realmente un sistema seguro, aunque evidentemente resuelven en alguna medida dicha problemática o por lo menos es el camino más fácil.

Por las características propias la tecnología propuesta y los objetivos que la misma se impone, en su primera versión no se incorporan los métodos tradicionales, como el explicado anteriormente. En cambio se sugieren una serie de mecanismos que en su momento han gosado de gran aceptación, hasta tanto no pueda ser solventado dicha problemática en futuras y no muy lejanas versiones.

5.2.1 SSI

Las directivas de *SSI*[16] constituyen poderosas herramientas para aumentar la productividad del programador, le permite al servidor modificar automáticamente el documento solicitado antes de ser enviado al navegador del cliente. Su propósito principal consiste en insertar dentro de las páginas *Web* el contenido de ficheros de texto, información dinámica sobre archivos o la salida resultante de la ejecución de ciertas instrucciones preseleccionadas del sistema. Los *Server Side Includes* pueden contener a su vez otros *Server Side Includes*, de forma que se puede llegar a introducir una cantidad enorme de texto en una página con la inclusión de un único comando.

5.2.2 SSL

El protocolo *SSL*[17] es un sistema diseñado y propuesto por *Netscape Communications Corporation,* al igual que *TLS*[18] constituye un protocolo criptográfico que proporciona comunicaciones seguras por una *red*, comúnmente Internet. Se ejecuta en la capa de aplicación del modelo *OSI* de conjunto con *HTTP, SMTP, NNTP* y sobre el protocolo de transporte *TCP*, que forma parte de la familia *TCP/IP*. En la mayoría de los casos se emplea junto a *HTTP* para formar *HTTPS*.

Implica una serie de fases básicas, las cuales se reflejan a continuación:

[16] **SSI** acrónimo de Server Side Includes
[17] **SSL** acrónimo de Secure Sockets Layer
[18] **TLS** acrónimo de Transport Layer Security

- ✓ Negociar entre las partes el algoritmo que se usará en la comunicación
- ✓ Intercambio de claves públicas y autenticación basada en certificados digitales
- ✓ Cifrado del tráfico basado en cifrado simétrico

Durante la primera fase, el cliente y el servidor negocian qué algoritmos criptográficos se van a usar. Las implementaciones actuales proporcionan las siguientes opciones:

- ✓ **Para criptografía de clave pública**: *RSA, Diffie-Hellman, DSA*[19] o Fortezza;
- ✓ **Para cifrado simétrico**: *RC2, RC4, IDEA*, *DES*[20], *Triple DES o AES*[21]

- ✓ **Con funciones hash**: MD5 o de la familia SHA.

La clave de sesión es la que se utiliza para cifrar los datos que vienen del y van al servidor seguro. Generándose una clave de sesión distinta para cada transacción, lo cual permite que aunque sea reventada por un atacante en una transacción dada, no sirva para descifrar futuras transacciones.

Proporciona cifrado de datos, autenticación de servidores, integridad de mensajes y opcionalmente, autenticación de cliente para conexiones *TCP/IP*. Cuando el cliente pide al servidor una comunicación segura, el servidor abre un puerto cifrado, gestionado por un software llamado Protocolo *SSL Record*, situado encima de *TCP*. Será el software de alto nivel, Protocolo *SSL Handshake*, quien utilice el Protocolo *SSL Record* y el puerto abierto para comunicarse de forma tangible con el cliente.

5.2.3 HTTPS

HTTPS[22] es un protocolo de aplicación basado en el protocolo *HTTP*, destinado a la transferencia segura de datos de hipertexto. Su objetivo principal consiste en crear un canal seguro sobre una red insegura. Esto proporciona una protección razonable contra ataques de tipo *eavesdropping* y *man-in-the-middle*, siempre que se empleen métodos de cifrado adecuados y que el certificado del servidor sea verificado, resultando de confianza.

Utiliza un cifrado basado en *SSL/TLS* para crear un canal más apropiado para el tráfico de información sensible que el protocolo *HTTP*. Este último opera en la capa más alta del modelo *OSI*, específicamente la de Aplicación, pero el de seguridad opera en una inferior, cifrando un mensaje previo a la transmisión y descifrandolo una vez recibido.

Independientemente de sus fortalezas es vulnerable cuando se aplica a contenido estático de publicación disponible. El sitio entero puede ser indexado usando un mecanismo de tipo *Araña web*, y la *URI* del recurso cifrado puede ser adivinada conociendo solamente el tamaño de la petición y respuesta. Esto permite a un atacante tener acceso al texto plano o contenido estático de publicación, así como al texto cifrado, permitiendo un ataque criptográfico.

Debido a que *SSL* opera bajo *HTTP* y no tiene conocimiento de protocolos de nivel más alto, los servidores *SSL* solo pueden presentar estrictamente un certificado para una combinación de puerto/IP en particular. Esto quiere decir, que en la mayoría de los casos, no es recomendable usar *Hosting virtual name-based* con *HTTPS*. Por otra parte existe una solución llamada *SIN*[23], que envía el *hostname* al servidor antes de que la conexión sea cifrada, sin embargo muchos navegadores antiguos no soportan esta extensión. El soporte para *SNI* está disponible desde *Firefox 2, Opera 8, e Internet Explorer 7 sobre*

[19] **DSA** acrónimo de Digital Signature Algorithm
[20] **DES** acrónimo de Data Encryption Standard
[21] **AES** acrónimo de Advanced Encryption Standard
[22] **HTTPS** acrónimo de Hypertext Transfer Protocol Secure
[23] **SNI** acrónimo de Server Name Indication

Windows Vista.

5.2.4 OpenSSL

OpenSSL es un proyecto de *software* desarrollado por los miembros de la comunidad *Open Source*, basado en *SSLeay*, desarrollado por Eric Young y Tim Hudson. Consiste en un robusto paquete de herramientas de administración y librerías relacionadas con la criptografía, que suministran mecanismos a otros paquetes como *OpenSSH* y navegadores *web* para acceso seguro a sitios *HTTPS*. Este tipo de herramientas facilitan y potencian en gran medida la implementación del *SSL*, así como otros protocolos relacionados con la seguridad, como el *TLS*[24].

5.2.5 Htaccess

Un fichero *.htaccess*[25], también conocido como archivo de configuración distribuida, es un fichero especial, popularizado por el Servidor *HTTP* Apache que nos permite definir diferentes directivas de configuración para cada directorio, así como sus respectivos subdirectorios, sin necesidad de editar el archivo de configuración principal de Apache.

Este ofrece un universo de posibilidades, detallándose a continuación los usos más frecuentes:

- ✓ Restringir el acceso a directorios
- ✓ Restringir el acceso a *IP* o *ISP*
- ✓ Creación de *URL* amigables semánticas
- ✓ Manejar errores del servidor.
- ✓ Crear redirecciones estáticas
- ✓ Controlar *Cache*
- ✓ Evitar *hotlink*
- ✓ Forzar Dominio sin *WWW*

Es evidente que estas técnicas por separado constan de disímiles vulnerabilidades, sin embargo la puesta en práctica de las mismas de forma conjunta podrían generarnos un sistema realmente seguro. Independientemente de esto existen una serie de principios básicos que cualquier aplicación o servicio web debe cumplir:

- ✓ **Validación de la entrada y salida de información:** La entrada y salida de información es el principal mecanismo que dispone un atacante para enviar o recibir código malicioso contra el sistema. Por tanto, siempre debe verificarse que cualquier dato entrante o saliente es apropiado y en el formato que se espera. Las características de estos datos deben estar predefinidas y debe verificarse en todas las ocasiones.

- ✓ **Diseños simples**: Los mecanismos de seguridad deben diseñarse para que sean los más sencillos posibles, huyendo de sofisticaciones que compliquen excesivamente la vida a los usuarios. Si los pasos necesarios para proteger de forma adecuada una función o módulo son muy complejos, la probabilidad de que estos pasos no se ejecuten de forma adecuada es muy elevada.

- ✓ **Utilización y reutilización de componentes de confianza:** Debe evitarse reinventar la rueda constantemente. Por tanto, cuando exista un componente que resuelva un problema de forma correcta, lo más inteligente es utilizarlo.

[24] **TLS** acrónimo de Transport Layer Security
[25] **htaccess** acrónimo de Hypertext Access

- ✓ **Defensa en profundidad:** Nunca confiar en que un componente realizará su función de forma permanente y ante cualquier situación. Hemos de disponer de los mecanismos de seguridad suficientes para que cuando un componente del sistema falle ante un determinado evento, otros sean capaces de detectarlo.

- ✓ **Tan seguros como el eslabón más débil**: La frase "garantizamos la seguridad, pues se utiliza *SSL*" es realmente muy popular, pero también es muy inexacta. La utilización de *SSL* certifica que el tráfico en tránsito entre el servidor y el cliente se encuentra cifrado, pero no garantiza nada acerca de los mecanismos de seguridad existentes. Por tanto, los desarrolladores no se deben fiar únicamente de los elementos de seguridad "exteriores", sino que es primordial identificar cuáles son los puntos precisos en los que deben establecerse las medidas de seguridad.

- ✓ **La "seguridad gracias al desconocimiento" no funciona:** El simple hecho de ocultar algo no impide que, a medio o largo plazo, llegue a ser descubierto. Tampoco es ninguna garantía de que tampoco será descubierto a corto plazo.

- ✓ **Verificación de privilegios:** Los sistemas deben diseñarse para que funcionen con la menor cantidad de privilegios posibles. Igualmente, es importante que los procesos únicamente dispongan de los privilegios necesarios para desarrollar su función, de forma que queden compartimentados.

- ✓ **Ofrecer la mínima información:** Ante una situación de error o una validación negativa, los mecanismos de seguridad deben diseñarse para que faciliten la mínima información posible. De la misma forma, estos mecanismos deben estar diseñados para que una vez denegada una operación, cualquier operación posterior sea igualmente denegada.

- ✓ **Otros aspectos**: Consideraciones de arquitectura, mecanismos de autenticación, gestión de sesiones de usuario, control de acceso, registro de actividad, prevención de problemas comunes, consideraciones de privacidad y criptografía.

Para mayor comprensión de la puesta en práctica de muchos de los elementos planteados sobre el *framework Ksike,* por concepto de construcción de aplicaciones orientadas a entornos *web* seguros, consulte el acápite *4.7.1*.

Conclusiones

Debe quedar bien claro que la intención de *Ksike* no implica para nada el concepto de competencia frente a otras tecnologías como *Symfony*, *Codeigniter*, *PradoPHP*, *Zend Framework*, etc. Evidentemente estos cuentan con comunidades estables de desarrollo que han invertido recursos a lo largo de varios años siendo lo que les ha permitido llegar a ser lo que representan hoy en día en el desarrollo de aplicaciones web.

Tampoco es objetivo desechar por completo la filosofía que estas proponen, sin embargo no es menos cierto que en estos últimos años se ha potenciado tanto el *HTML* como el *Javascript*. Negar esto también sería rechazar de cierta forma el desarrollo. Es por ello que *Ksike* trata de aprovecharlo al máximo, en aras de impulsar una nueva perspectiva en el desarrollo de aplicaciones de este tipo. Proveyendo un marcado enfoque orientado a objetos y delegando la construcción de interfaces gráficas de usuarios a *framework* como *ExtJs*, *OpenLayers*, *Yui*, *JQuery*, entre otros.

Está claro que muchos *framework* como los mencionados no pueden darse el lujo de dar estos saltos tecnológicos y desechar muchos de los mecanismos que estos proveen para la gestión de vistas. Teniendo en cuenta que estos comenzaron a desarrollarse mucho antes que las tecnologías del cliente tomaran el rol protagónico que hoy se están ganando, por ende se invirtieron muchos esfuerzos en solventar los problemas que para en ese entonces constituían una realidad. Que van desde el empleo de motores de plantillas como *Smarty* hasta propias implementaciones a través de patrones de diseños que optimizan el proceso de renderizado del código *html* generado por cada una de las vistas.

Pero aún así esto trae inconvenientes, tales como la reconstrucción de toda la aplicación por cada vez que se necesite ejecutar una petición al servidor. En esta dirección también se ha trabajado mucho en estos *framework* como *Symfony,* el cual incluye librerías para hacer estas peticiones por ajax, sin embargo falta mucho camino por recorrer en este sentido, teniendo en cuenta que en ocasiones se llega a violar estas arquitecturas cuando se emplean librerías como *ExtJs* por tan solo citar un ejemplo.

Por otra parte también existe un factor humano, muchos han sido los años en que el desarrollo de aplicaciones *web* se ha mantenido utilizado el patrón *MVC*[26] basado en el renderizado de plantillas y básicamente empleándose un estilo de programación estructurada. Precisamente es esto lo que *Ksike* intenta suplir. Estamos concientes que el cambio genera incertidumbre lo que traen consigo desconcierto y de seguro habrán muchos detractores, pero el camino más largo nunca comienza si al menos no se da el primer paso.

Como se puede apreciar este *framework* provee mecanismos que potencian la escalabilidad del mismo, sobre todo a partir del trabajo colaborativo o lo que se conoce como comunidades de desarrollo, sin la necesidad de tener conocimientos avanzados sobre su arquitectura interna y mucho menos asumir la modificación del propio núcleo. Este es un elemento realmente importante partiendo del hecho que el desarrollo del marco de trabajo no se limita al entorno de sus autores, sino que rebasa sus fronteras permitiendo que el tiempo de actualizaciones se reduzca de forma prácticamente exponencial.

Entre los más significativos que se pueden destacar principalmente en el *API* cliente es el empleo de patrones de diseño para las clases en el *Javascrip* y por parte del servidor el desarrollo de drivers, e incluso dentro de ellos se encuentran *ConfigManager* y *OutManager,* que implementan sus propios recursos en función de que puedan ser extendidos con facilidad.

[26] **MVC** acrónimo de Modelo Vista Controlador

Bibliografía

- ✓ Cesar de la Torre Llorente, U. Z. (2010). *Guia de Arquitectura N-Capas Orientada al dominio con .Net*. Krasis Consulting S. L.
- ✓ Fabien Potencier, F. Z. (2008). *Symfony la guía definitiva.*
- ✓ Perez, J. E. (2009). *CSS Avanzado.*
- ✓ Javier Eguíluz Pérez. Librosweb. Introducción a JavaScript. [En línea] [Citado el: 15 de septiembre de 2010.] http://www.librosweb.es/javascript/.
- ✓ Stefanov, S. (July 2008). *Object Oriented JavaScript*. (G. Singh, Ed.) Packt Publishing Ltd.
- ✓ Wage, K. V. (2008). *Doctrine Manual.*
- ✓ PHP Group. php. [En línea] [Citado el: 12 de septiembre de 2010.] http://www.php.net/.
- ✓ PostgreSQL Global Development Group. PostgreSQL. [En línea] [Citado el: 19 de octubre de 2010.] http://www.postgresql.org/.
- ✓ Javier Eguíluz Pérez. Librosweb. Introducción a JavaScript. [En línea] [Citado el: 15 de septiembre de 2010.] http://www.librosweb.es/javascript/.
- ✓ Orchard, Leslie M., y otros. 2009. *Professional JavaScript Frameworks: Prototype, JQuery, YUI, ExtJS, Dojo, and MooTools.* 2009. 047038459X ISBN-13: 9780470384596.
- ✓ Goodman, Danny. 2008. JavaScript y DHTML. 2008. 8441523886 ISBN-13: 9788441523883.
- ✓ Mellado Domínguez, Javier. 2008. *Ajax*. 2008. 8441524149 ISBN-13: 9788441524149.
- ✓ Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. 2005. *Design Patterns: Elements of Reusable Object-Oriented Software.* s.l. : Addison Wesley, 2005. 0-201-63361-2.

Glosario de Términos

- ✓ **AAA:** Acrónimo de *Authentication, Authorization and Accounting*, en castellano: Autenticación, Autorización y Contabilización.
- ✓ **Ajax:** Acrónimo de *Asynchronous JavaScript And XML*
- ✓ **API:** Acrónimo de Interfaz de Programación de Aplicaciones.
- ✓ **CLI:** Acrónimo de *Command Line Interface*.
- ✓ **CMS:** Acrónimo de *Content Management System*, en castellano: Sistema de gestión de contenidos.
- ✓ **CSS:** Acrónimo de *Cascading Style Sheets,* en castellano: Hojas de Estilo en Cascada.
- ✓ **DB:** Acrónimo de *Database*, en castellano: Bases de datos. Entiéndase como el conjunto de informaciones pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior empleo.
- ✓ **DBMS:** Acrónimo de *Database Management System*, en castellano: Sistemas de Gestión de Bases de Datos. Constituyen un tipo de software muy específico, cuyo propósito general consiste en manejar de manera clara, sencilla y ordenada un conjunto de datos que posteriormente se convertirán en información relevante para determinada organización.
- ✓ **DOM:** Acrónimo de *Document Object Model*.
- ✓ **DTS:** Acrónimo de Dirección de Tecnologías y Sistemas, centro de desarrollo de *software*.
- ✓ **RIA:** Acrónimo de *Rich Internet Applications*.

- ✓ **PGDG:** Acrónimo de *PostgreSQL Global Development Group*.
- ✓ **MVCC:** Acrónimo de Acceso Concurrente Multiversión.
- ✓ **FTP**: Acrónimo de *File Transfer Protocol*, en castellano: Protocolo de transferencia de archivos.
- ✓ **GRASP:** Acrónimo de *General Responsibility Assignment Software Patterns*, en castellano: Patrones Generales de Software para Asignación de Responsabilidades.
- ✓ **GNU GPL**: Licencia Pública General de GNU, es una licencia creada por la *Free Software Foundation* en 1989, orientada principalmente a proteger cinco principios del software libre. Su propósito consiste en proteger determinado producto de intentos de apropiación que restrinjan dichas libertades a los usuarios.
- ✓ **UCI:** Universidad de las Ciencias Informáticas.
- ✓ **UCID**: Unidad de Compatibilización e Integración para la defensa, centro de desarrollo de software que radica dentro de la UCI.
- ✓ **URL:** Acrónimo de *Uniform Resource Locators*, en castellano: Localizador de Recursos Uniforme, de acuerdo a un formato estándar se emplea en función de nombrar recursos en Internet para su localización o identificación.
- ✓ **Licencia**: Documento rector en forma de contrato mediante el cual un individuo o entidad recibe de otra el derecho de uso de varios de sus bienes, pudiendo darse a cambio del pago de un monto determinado por el uso de los mismos.
- ✓ **Metadatos**: Son un enfoque importante para construir un puente sobre el intervalo semántico. Constituyen un conjunto de datos estructurados y codificados que describen características para determinada información, conteniendo indicadores para identificar, descubrir, valorar y administrar las instancias. Análogo al empleo de índices para localizar recursos.
- ✓ **Ontología**: El término en informática hace referencia a la formulación de un exhaustivo y riguroso esquema conceptual dentro de uno o varios dominios dados; con la finalidad de facilitar la comunicación y el intercambio de información entre diferentes sistemas y entidades.
- ✓ **HTML:** Acrónimo de *Hypertext Markup Language,* en castellano: Lenguaje de Etiquetado de Hipertexto.
- ✓ **HTTP**: Acrónimo de *Hypertext Transfer Protocol*, define la sintaxis y la semántica que utilizan los elementos de software de la arquitectura web para comunicarse. Es un protocolo orientado a transacciones y sigue el esquema petición-respuesta entre un cliente y un servidor.
- ✓ **HTTPS**: Acrónimo de *Hypertext Transfer Protocol Secure,* en castellano: Protocolo seguro de transferencia de hipertexto.
- ✓ **JSON:** Acrónimo de *JavaScript Object Notation*.
- ✓ **Script**: Guión o conjunto de instrucciones definidos por determidado lenguaje, que permiten la automatización de tareas creando pequeñas utilidades, usualmente son archivos de texto.
- ✓ **SSI:** Acrónimo de *Server Side Includes.*
- ✓ **SSL:** Acrónimo de *Secure Sockets Layer.*
- ✓ **W3C:** Acrónimo de *World Wide Web Consortium*.
- ✓ **WWW**: Acrónimo de *World Wide Web*, es considerado un sistema de distribución de información basado en hipertexto o hipermedios enlazados y accesibles a través de Internet.

Con un navegador web, un usuario visualiza sitios web compuestos de páginas, que pueden contener texto, imágenes, videos u otros contenidos multimedia, y navega a través de ellas usando hiperenlaces.

- ✓ **Web browser**: En idioma español se denomina navegador web. Es un programa informático que permite visualizar e interactuar con la información contenida en una aplicación web. Entre los más utilizados se encuentran *Google Chrome, Safari, Internet Explorer, Mozilla Firefox*, etc.
- ✓ **Web 1.0:** Internet básica, basaba en páginas cuyo contenido era de carácter estático, programadas en *HTML* provocando que no fueran actualizadas frecuentemente, donde los usuarios se limitan a la visualización pasiva de información que se les proporciona.
- ✓ **Web 1.5:** Denominados sitios dinámicos, donde los *CMS* jugaron un papel preponderante, permitiendo enviar páginas *HTML* de forma dinámica, estas eran creadas al vuelo desde una actualizada base de datos.
- ✓ **Web 2.0:** La red social de colaboración está comúnmente asociado con un fenómeno social, basado en la interacción que se logra a partir de diferentes aplicaciones web, que facilitan el compartir información, la interoperabilidad, el diseño centrado en el usuario o DCU. Son las comunidades, los servicios de red social, los servicios de alojamiento de videos, las wikis, blogs, etc. Permitiéndole a sus usuarios interactuar con otros o cambiar contenido del mismo.
- ✓ **Web 3.0:** La red semántica se basa en la idea de añadir metadatos semánticos y ontológicos a la *WWW*. Proveyendo lógica descriptiva mediante diversos lenguajes más elaborados como *SPARQL*, *POWDER* u *OWL*, facilitando la evaluación automática del contenido a través de máquinas de procesamiento. El objetivo es mejorar Internet ampliando la interoperabilidad entre los sistemas informáticos basándose en agentes inteligentes, permitiendo hacer búsquedas de información sin operadores humanos.
- ✓ **Web 4.0:** La red móvil sentará sus bases a partir de la proliferación de la comunicación inalámbrica, tanto para personas como objetos. Estos podrán conectarse en cualquier momento y desde cualquier lugar del mundo físico o virtual. Con más entes en la red, se suma un nuevo nivel de contenido generado por los usuarios, y con él, otro nivel de análisis. Por ejemplo, el GPS que guía al automóvil y facilitándole al conductor a mejorar la ruta prevista o a ahorrar combustible, en poco tiempo le evitará el trámite de manejarlo.
- ✓ **Web 5.0:** La red sensorial o emotiva, parte del hecho de que independientemente del debate eufórico que pueda generar un *blog* o la reacción en cadena provocada por un video publicado en *YouTube*, la *Web* se mantiene emocionalmente neutra, pues es incapaz de percibir el estado anímico de los usuarios. La empresa *Emotiv Systems* ha creado neurotecnología, mediante auriculares que permiten al usuario interactuar con el contenido que responda a sus emociones o cambiar en tiempo real la expresión facial de un *avatar*. Si se pueden personalizar las interacciones para crear experiencias que emocionen a los usuarios, la *Web 5.0* será, sin duda, más afable que sus antecesoras.

Licencia

Este apartado se refiere a las licencias y términos de empleo tanto para este documento como para la tecnología en sentido general, así como las herramientas que esta provee. Entiéndase por ello como documento rector en forma de contrato mediante el cual un individuo o entidad recibe de otra el derecho de uso de varios de sus bienes, pudiendo darse a cambio del pago de un monto determinado por el uso de los mismos.

En aras de asegurar los principios de *software* libre se decide distribuir bajo la licencia que propone la fundación de software libre. Esta cuenta con un marcado carácter de protección heredada lo que implica que todo aquello que se desarrolle bajo esta tecnología mantiene el mismo sistema de licenciamiento. A continuación se exponen lo que plantea dicha licencia

GNU LESSER GENERAL PUBLIC LICENSE
Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages--typically libraries--of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need toencourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a) The modified work must itself be a software library.

   b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

   c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

   d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

   (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is

unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e) Verify that the user has already received a copy of these
materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR EDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the library's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU

Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library `Frob' (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990 Ty Coon, President of Vice
That's all there is to it!