Cuadro de mando para iPad

Consultores: Jordi Ceballos Villach
Jordi Almirall López

Alumno: Javier Ángel Caballero Gimeno

1. Introducción

El presente proyecto trata de cubrir una necesidad específica: dotar a los posibles usuarios de un cuadro de mando personalizable para un dispositivo iPad.

Existen múltiples escenarios de uso para esta aplicación: desde un usuario que quiera visualizar información de bolsa a otro que busque controlar un servicio como un centro de atención a usuarios. El denominador común es la necesidad de mostrar información de forma clara y gráfica para de un vistazo conocer el estado y poder tomar decisiones rápidas en base a ello, aprovechando las ventajas de un dispositivo móvil como es el iPad, que permite acceder a esa información en cualquier momento y lugar.

En el App Store existen otras aplicaciones que tratan de resolver este mismo problema, pero en general su el enfoque es ser herramientas a medida para servicios concretos publicados por los mismos desarrolladores de la aplicación, lo cual las hace inútiles fuera de esos casos de uso. Como ventaja algunas de estas aplicaciones disponen de funcionalidad que quedan fuera del alcance del presente proyecto.

Este proyecto busca realizar una herramienta sencilla, clara y adaptable a las necesidades y prioridades de varios perfiles distintos de usuarios. Específicamente se busca que no sea dependiente de orígenes de datos predeterminados sino que pueda trabajar con cualquiera que produzca un formato adecuado.

2. Objetivo del proyecto

El objetivo del proyecto consiste en desarrollar una aplicación para iPad cuya función principal es ser un cuadro de mando. De esta manera permite ofrecer información relevante sobre diversos aspectos de forma visual y personalizada a las necesidades del usuario.

Para ello se dispondrá de un conjunto de visualizadores, que se podrán colocar en cualquier punto del área de trabajo de la aplicación. Estos elementos serán de varios tipos:

- Gráficos de barras
- Gráficos de tarta
- Semáforos con indicadores numéricos y visuales
- Mapas con posicionamiento de elementos
- Tablas de datos

De esta forma el usuario podrá componer un cuadro de mando a medida con la información que le interese disponer en cada momento.

Se busca realizar una herramienta sencilla, clara y adaptable a las necesidades y prioridades de varios perfiles distintos de usuarios.

La plataforma a la que está destinada esta aplicación es el Apple iPad. En concreto se va a desarrollar sobre el SDK de la versión de iOS 8.3 (versión más actual en el momento de comenzar el proyecto), lo cual limitará su ejecución a los modelos que soporten esta versión. En las fechas actuales el único iPad que no puede ejecutarla es el modelo de primera generación.

3. Planificación del proyecto

3.1. Planificación docente

Desde el punto de vista docente se marcan las siguientes fechas para las entregas parciales:

	Fecha inicio	Fecha fin
PEC 1	mié, 25 feb 2015	mié, 11 mar 2015
PEC 2	jue, 12 mar 2015	mié, 8 abr 2015
PEC 3	jue, 9 abr 2015	mié, 20 may 2015
Entrega final	jue, 21 may 2015	dom, 21 jun 2015

Estos hitos se añaden al plan de proyecto general que detalla en mayor medida las distintas fases y tareas del proyecto.

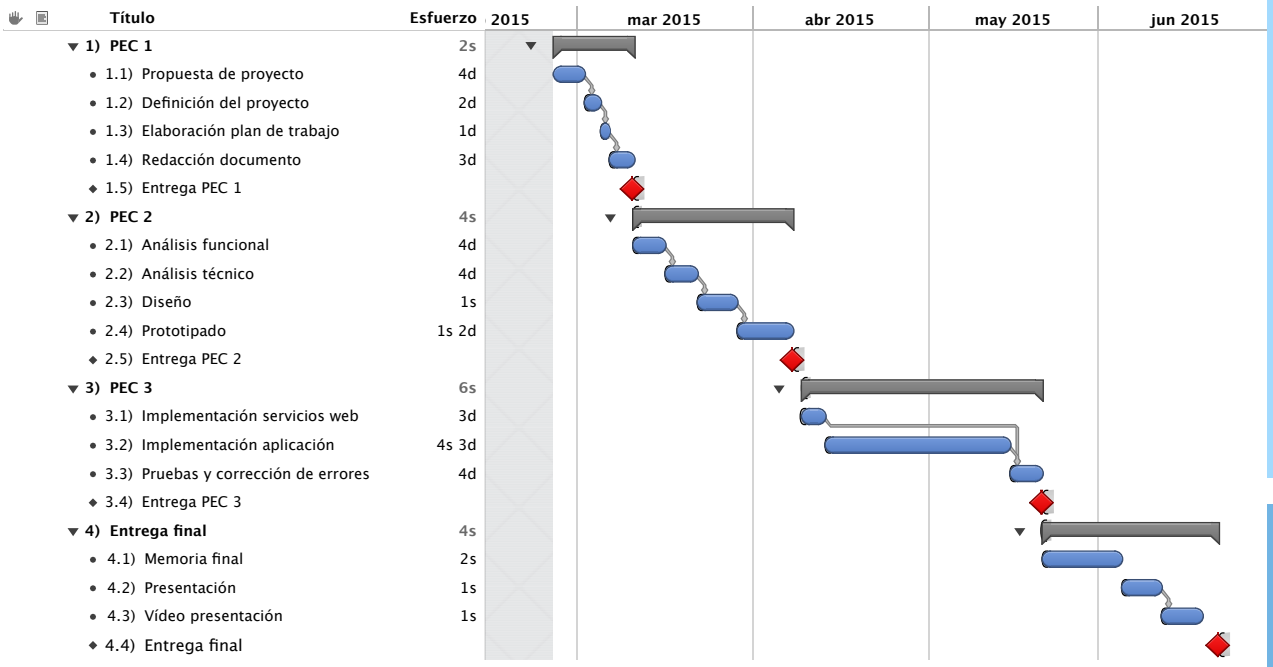
3.2.Planificación proyecto

De cara a la planificación se ha considerado una semana de trabajo estándar de cuarenta horas, trabajando ocho horas diarias. Realmente la dedicación semanal si será de cuarenta horas pero la distribución de las mismas será irregular contando incluso con fines de semana.

La planificación del proyecto ha tenido en cuenta como hitos las entregas docentes previstas en el apartado anterior. Además de las entregas como tal, las distintas fases se han articulado alrededor de las fechas de cada una de las actividades docentes propuestas.

En el diagrama de Gantt se pueden observar las distintas tareas asociadas a cada fase, así como su duración y dependencias. Al sólo existir un recurso dedicado a la ejecución del proyecto este resulta totalmente secuencial en el tiempo.

Diagrama de Gantt:



4. Recursos y tecnologías

4.1. Lenguaje de programación

El lenguaje de programación utilizado ha sido Swift. Justo al inicio del desarrollo se publicó la nueva versión de iOS 8.3 junto al nuevo SDK correspondiente, que entre otros cambios actualizaba el lenguaje de programación Swift a la versión 1.2.

En algunos aspectos se consideró que pudiera ser necesario usar Objective-C, particularmente en las vistas con elementos gráficos de la librería Core Plot (https://github.com/core-plot/core-plot). Esto finalmente no ha sido necesario y el proyecto se ha realizado íntegramente en Swift.

Se ha seleccionado Swift para el desarrollo principalmente por dos razones:

- Se buscaba hacer una aplicación nativa en el entorno de Apple, lo que sólo es posible mediante Objective-C o Swift.
- Swift es un nuevo lenguaje más moderno que Objective-C, con las ventajas que ello conlleva, además de ser el lenguaje que propone Apple a futuro para su plataforma.

Al margen de la propia aplicación para iPad, objetivo principal del proyecto, se han creado unos servicios web RESTful sobre plataforma Java 1.8. Estos servicios se han creado con el objeto de servir de origen de datos para las pruebas con la aplicación. En este caso se ha usado Java por el conocimiento previo y experiencia en este entorno.

4.2. Entorno de desarrollo

El entorno de desarrollo utilizado ha sido Xcode. Es la herramienta proporcionada por Apple para el desarrollo tanto para Mac OS X como para iOS. Sus orígenes se encuentran en el entorno de desarrollo Project Builder, suministrado con el sistema operativo NeXT, el cual es el predecesor del actual OS X y base también de iOS.

Inicialmente estaba previsto usar la versión 6.2, pero antes de comenzar el desarrollo se publicó la versión 8.3 del SDK de iOS, el cual incluía la versión 6.3 de Xcode.

Además del propio entorno de desarrollo se han utilizado otras herramientas que forman parte del SDK: Simulador iOS e Instruments. La primera es necesaria para poder ejecutar en entorno de pruebas sin dispositivo físico iOS, la segunda para evaluar el correcto funcionamiento de la aplicación (consumo de recursos, rendimiento, fugas de memoria, etc).

Se ha utilizado Xcode por ser prácticamente la única alternativa existente para el desarrollo de aplicaciones para iOS con lenguaje Swift, además de disponer de una completa documentación y el soporte del propio fabricante.

En cuanto al entorno de desarrollo para los servicios web de Java se ha utilizado NetBeans. Aunque en principio se consideró utilizar IntelliJ, las herramientas integradas para creación de servicios web de NetBeans han resultado más cómodas y sencillas de utilizar.

4.3. Frameworks

En este aspecto la premisa ha sido basarse todo lo posible en los elementos estándar proporcionados dentro del SDK 8.3 de iOS. En cierta medida esto ayuda a que futuras versiones de iOS puedan ejecutar adecuadamente la aplicación y que la migración de esta a versiones superiores, si fuera necesario, resulte más sencilla.

Al margen de UIKit y Foundation, estándar en todas las aplicaciones de iOS, se han utilizado los siguientes frameworks:

- Accelerate
- QuartzCore
- MapKit
- CorePlot

Todos ellos, salvo CorePlot, forman parte del SDK 8.3 de iOS. MapKit se utiliza para la representación de mapas y tanto Accelerate como QuartzCore son requisitos de Core Plot.

CorePlot se ha utilizado para la generación de los gráficos de tarta y barras. Dado que desarrollar una librería propia para esta necesidad quedaba fuera del alcance del proyecto, se realizó una búsqueda de posibles alternativas para cubrir esta necesidad.

Se seleccionó CorePlot por ser el proyecto de código abierto más avanzado y completo de los existentes. Aunque el hecho de utilizar un proyecto de código abierto no era una prioridad per se, si lo era que el coste fuera mínimo o inexistente, dejando de lado alternativas comerciales como Shinobicharts (https://www.shinobicontrols.com/).

En concreto se ha usado la actual versión en desarrollo, rama 2.0, por ser la que mejor interoperabilidad proporciona con el lenguaje Swift.

Para los test unitarios de ha utilizado XCTest, también parte del SDK 8.3 de iOS. Al igual que ha ocurrido con el resto de componentes se ha preferido el uso de los disponibles directamente en el SDK de Apple, por lo que en este caso no se ha considerado ningún otro.

4.4. Otras herramientas

Además de las herramientas de desarrollo se han utilizado otras para la elaboración de la documentación, tales como la suite iWork (procesador de texto, hoja de cálculo y presentaciones), OmniPlan (planificación del proyecto), Omnigraffle (diseño de interfaces y diagramas) y Pixelmator (tratamiento de imágenes).

Para poder ejecutar los web service de prueba se ha preparado una máquina virtual con sistema operativo Debian. Sobre está máquina se ha instalado el servidor Apache Tomcat

(contenedor de servlets) y un servidor MySQL (base de datos). Esta máquina virtual se ha ejecutado usando VMware Fusion.

4.5. Recursos hardware

Para la ejecución del proyecto son necesarios diversos recursos, tanto hardware como software:

- Equipos de desarrollo: Retina MacBook Pro 15” e iMac 27”.
- Dispositivos iOS: iPad 4 e iPad mini.

5. Riesgos del proyecto

Como todo proyecto, se presentan una serie de riesgos que pueden comprometer el correcto desarrollo del mismo. Para garantizar el éxito del proyecto todos estos riesgos deben ser identificados, así como las posibles acciones correctivas a realizar en caso de producirse.

La siguiente tabla resume los riesgos identificados:

Riesgo	Descripción	Probabilidad	Impacto	Acciones mitigadoras
Avería hardware	Avería de los de los equipos de desarrollo	Baja	Bajo	Al contar con dos ordenadores para desarrollo es improbable que ambos fallaran simultáneamente. En cualquier caso se podría disponer de otros ordenadores capaces de realizar el trabajo.
Disponibilidad por trabajo	Pueden surgir requerimientos en el trabajo, como viajes, que reduzcan el tiempo disponible para el desarrollo del proyecto.	Media	Muy alto	Viajar con ordenadores aptos para desarrollo, para poder aprovechar los tiempos disponibles. Tratar de mejorar siempre la planificación para reducir el impacto de lapsos de tiempo sin dedicación al proyecto.
Enfermedades y accidentes	Enfermedad o accidentes que impidan cumplir con los hitos	Baja	Muy alto	Evitar actividades de riesgo.
Falta de conocimientos	La implementación de alguna funcionalidad puede resultar dificultosa o imposible por la falta de conocimientos.	Media	Alto	Buscar alternativas en caso de ser necesario. Consultar en foros de desarrollo.
Pérdida de datos	Pérdida de documentos o archivos de desarrollo	Baja	Alto	Sincronización de todos los ficheros entre los equipos de desarrollo y servidor NAS. Copia de seguridad local (Time Machine) en ordenador sobremesa (iMac).

6. Análisis funcional

6.1. Requerimientos funcionales

La función principal de la aplicación es presentarse como un lienzo en blanco donde el usuario podrá añadir diversos elementos de visualización para componer un cuadro de mando totalmente personalizado. Estos elementos podrán ser añadidos, movidos o eliminados por el usuario con total libertad sobre el área de visualización.

Los elementos a disposición del usuario son los siguientes:

- Gráficos de barras
- Gráficos de tarta
- Semáforos con indicadores numéricos y visuales
- Mapas con posicionamiento de elementos
- Tablas de datos

Cada elemento dispondrá de una configuración propia con parámetros tales como el origen de datos, título o intervalo de actualización. En función del tipo elegido estas configuraciones proporcionarán unas u otras opciones disponibles.

El usuario podrá componer un cuadro de mando a medida añadiendo y configurando estos elementos a su gusto. En cualquier momento podrá eliminar u ordenar estos elementos según su criterio.

6.2. Requerimientos no funcionales

El interfaz está destinado a dispositivos iPad. Aunque esta aplicación podría ser adaptada para funcionar en dispositivos con pantalla más pequeña, como un iPhone o un iPod, está prevista sólo para iPad por considerarse el dispositivo óptimo para este tipo de aplicación.

La aplicación muestra una barra de iconos similar al dock de OS X, donde cada icono corresponde a un tipo de visualizador a añadir a la pantalla. En el momento de adicionarlo se presentarán las opciones de configuración del mismo.

Los elementos podrán reubicarse por el área de pantalla de la aplicación arrastrándolos para que de esta forma el usuario pueda componer la información a su gusto. Podrán ser eliminados pulsando un botón de cierre.

Los datos de origen deberán ser publicados en un formato basado en JSON previamente definido y deberán poder ser obtenidos mediante protocolo HTTP.

6.3. Casos de uso

En esta aplicación sólo participa un perfil de usuario en un momento dado. Por su orientación y diseño los casos de uso son reducidos:

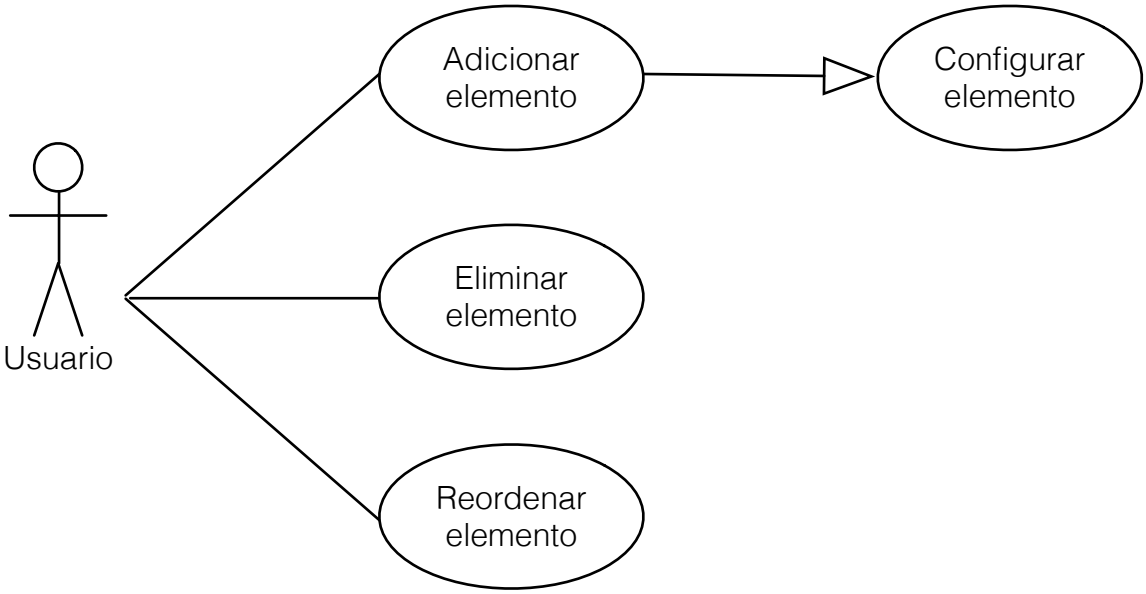


Figura: Diagrama de casos de uso

6.3.1. Caso de uso 1

Identificador	CU-001
Nombre	Adicionar elemento.
Prioridad	Normal.
Descripción	El usuario toca uno de los botones de la barra inferior de la aplicación, que representa un tipo concreto de elemento de visualización.
Actores	Usuario.
Precondiciones	No estar adicionando otro elemento.
Iniciado por	
Flujo	El usuario pulsa el botón del elemento deseado y espera la aparación de la ventana de configuración.
Postcondiciones	
Notas	

6.3.2.Caso de uso 2

Identificador	**CU-002**
Nombre	Configurar elemento.
Prioridad	Normal.
Descripción	El usuario completa los datos necesarios para realizar la configuración del elemento.
Actores	Usuario.
Precondiciones	Haber presionado previamente el elemento deseado para adicionar.
Iniciado por	Adicionar elemento.
Flujo	El usuario inicia primero la adición del elemento (CU-001), se presentan las opciones de configuración
Postcondiciones	
Notas	Los parámetros varían en función del tipo de elemento a adicionar.

6.3.3. Caso de uso 3

Identificador	**CU-003**
Nombre	Eliminar elemento
Prioridad	Normal.
Descripción	El usuario quita un elemento del área de visualización.
Actores	Usuario.
Precondiciones	Existencia del elemento a eliminar.
Iniciado por	
Flujo	El usuario presiona el botón de eliminar situado en la esquina superior derecha del elemento que desea quitar.
Postcondiciones	
Notas	

6.3.4.Caso de uso 4

Identificador	**CU-004**
Nombre	Reordenar elemento.
Prioridad	Normal.
Descripción	El usuario mueve un elemento en el área de visualización.
Actores	Usuario.
Precondiciones	Existencia del elemento a mover.
Iniciado por	
Flujo	El usuario arrastra el elemento sobre el área de visualización hasta que consigue la ubicación deseada.
Postcondiciones	No debe salir del área de visualización.
Notas	

7. Diseño técnico

7.1. Tecnologías

El objetivo del proyecto consiste en desarrollar una aplicación para iPad. Existen diversas formas de acometer esta tarea: usando el entorno de desarrollo de Apple, creando una aplicación web específica para estos dispositivos o con entornos de terceros como Adobe AIR.

En este caso se ha optado por crear una aplicación nativa usando el entorno de desarrollo de Apple Xcode (versión 6.3). Se ha considerado que el rendimiento es el más óptimo y además permite acceder a todas las prestaciones ofrecidas por los dispositivos iPad.

En cuanto al lenguaje de programación se ha optado por Swift, salvo en circunstancias en las que fuera necesario interactuar con elementos no accesibles desde este, usando entonces Objective-C.

El patrón de programación a utilizar es MVC, el cual define desde los roles hasta la forma de interactuar de los objetos. Tanto la documentación como las librerías de Apple responden a este patrón, con lo que es lógico aplicarlo en los diseños propios, ya que esta es la orientación prevista.

Todas las prestaciones a nivel de entorno de desarrollo se encuentran incluidas en el entorno oficial de Apple (iOS SDK 8.3), salvo las relacionadas con la creación de gráficas de barras y tartas. Para este aspecto concreto se usará la librería open source Core Plot (https://github.com/core-plot/core-plot), que entre los muchos tipos de gráficos soportados por la misma dispone de implementaciones tanto para gráficas de tarta como de barras . En particular se va a usar la rama de desarrollo (versión 2.0), ya que mejora notablemente la compatibilidad con el lenguaje Swift.

Como esta aplicación consume servicios web genéricos para representar la información que estos proporcionan, se crearan unos servicios web (RESTful) a modo de ejemplo para

poder usar la aplicación. Para ello se hará uso de un servidor Apache Tomcat que interrogará una base de datos MySQL mediante JDBC.

7.2. Esquema general de la aplicación

La aplicación consume datos generados por uno o diversos servicios web simultáneamente. Los datos se obtendrán tanto a través de internet como de posibles recursos en red local, pudiendo combinar ambos en cualquier momento dado. El siguiente diagrama muestra el funcionamiento típico de la aplicación consultando un servicio web y mostrando la información obtenida:

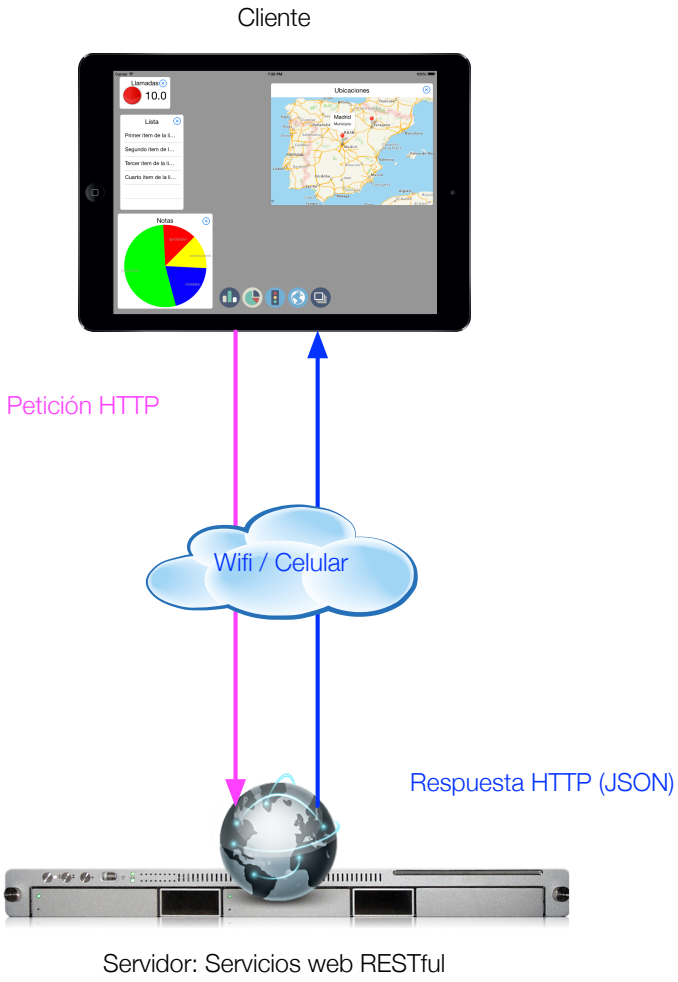


Figura 1: Aplicación obteniendo información a través de un servicio web productor de JSON.

7.3. Estructura de clases

Esta aplicación está desarrollada aplicando el patrón modelo vista controlador (MVC). Esto ha dado lugar a que cada elemento esencial realmente está compuesto por tres: la vista, el controlador que maneja a la vista y al modelo y el propio modelo que representa la información. En la siguiente figura se muestra el diagrama UML de las clases de la aplicación:

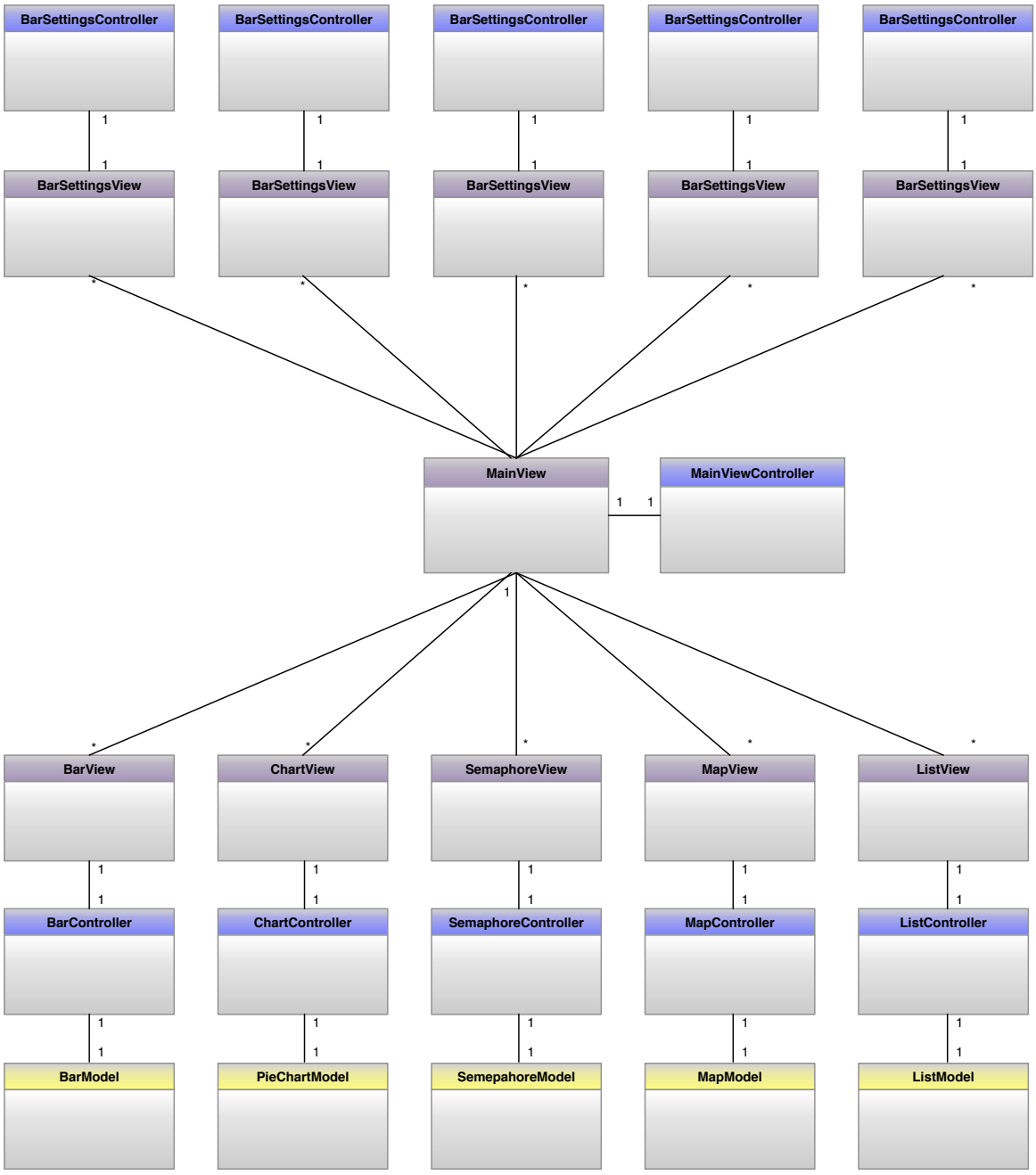


Figura 2: Diagrama de clases UML de la aplicación. En color morado las clases de tipo vista, en color azul las de tipo controlador y en color amarillo las de tipo modelo.

7.4. Elementos y clases relacionadas

Área de visualización

Se implementa mediante dos clases: MainView y MainViewController.

MainView es la clase con los componentes visuales. Sobre esta se muestran los botones para utilizar la aplicación y se añaden los elementos, que son conjuntos de vista - controlador - modelo.

MainViewController contiene el código actúa sobre la vista, siendo el responsable de permitir la adición de los nuevos elementos, así como el movimiento y la eliminación de los mismos.

Visualizadores

Al igual que el área de visualización estos se implementan usando el patrón MVC. Existen cinco tipos de elementos de visualización con sus clases relacionadas:

- BarView - BarViewController - BarModel
- ChartView - ChartViewController - ChartModel
- SemaphoreView - SemaphoreViewController - SemaphoreModel
- MapView - MapViewController - MapModel
- ListView - ListViewController - ListModel

Las clases *View* aportan los componentes visuales. Según el tipo muestran un tipo de información u otra. También incluyen los botones para por ejemplo eliminar el componente.

Las clases de tipo *Controller* reciben las notificaciones del modelo y actúan sobre la vista para que esta represente los datos.

Las clases *Model* interrogan los servicios web y almacenan la información procedente de estos.

Elementos para configurar y adicionar visualizadores

Como el resto de elementos, se implementan usando el patrón MVC, aunque por su simplicidad carecen de clases separadas para los modelos. Existen tantos tipos como elementos de visualización:

• BarSettings - BarSettingsController

• ChartSettings - ChartSettingsController

• SemaphoreSettings - SemaphoreSettingsController

• MapSettingsView - MapSettingsController

• ListSettingsView - ListSettingsController

Las clases *View* aportan los componentes visuales. Muestran los distintos parámetros disponibles para configurar el elemento designado, además del botón para adicionarlo a la vista principal.

Las clases *Controller* recogen los parámetros introducidos en la vista e implementan la acción necesaria para añadir el nuevo elemento al área de visualización principal.

7.5. Formato de datos

La aplicación consulta servicios web RESTful para obtener los datos a representar. Cada tipo de visualizador requiere un conjunto de datos concreto para poder mostrar la información, así que los orígenes de datos deben adaptarse a las convenciones que se detallan a continuación.

Visualizador gráfico de barras

Este visualizador espera una lista de elementos con los valores “name” y “value”, representado el primero al identificador del mismo y el segundo al valor correspondiente. Ejemplo:

```
[{"name":"aprobados","value":4},{"name":"suspensos","value":9},
{"name":"notables","value":7},{"name":"sobresalientes","value":2},{"name":"no
presentados”,"value":3}]
```

Visualizador gráfico de tarta

Este visualizador espera un formato idéntico al anterior: una lista de elementos con los valores “name” y “value”, representado el primero al identificador del mismo y el segundo al valor correspondiente.

Ejemplo:

```
[{“name":"aprobados","value":2},{"name":"suspensos","value":8},
{"name":"notables","value":3},{"name":"sobresalientes","value":2}]
```

Visualizador gráfico de mapa

Este visualizador espera una lista de elementos a posicionar en un mapa, requiriendo en total cuatro parámetros para cada uno:

- “title”: Texto principal que se muestra sobre el punto del mapa.
- “subtible”: Subtítulo asociado al texto principal.
- “latitude”: Latitud del punto en formato decimal
- “longitude”: Longitud del punto en formato decimal

Ejemplo:

```
[{"title":"Zaragoza","subtitle":"Municipio","latitude":
41.6488,"longitude":-0.8896},
{"title":"Madrid","subtitle":"Municipio","latitude":
40.4168,"longitude":-3.7038}]
```

Visualizador gráfico de semáforo

Este visualizador sólo espera un par de valores “name” y “value”, representado el primero al identificador del mismo y el segundo al valor correspondiente

Ejemplo:

```
[{"name":"Llamadas espera","value":20}]
```

Visualizador gráfico de lista

Este visualizador espera una lista de elementos con un único valor a representar: “value”.

Ejemplo:

```
[{"value":"Primer item de la lista"},{"value":"Segundo item de la lista"},
{"value":"Tercer item de la lista"},{"value":"Cuarto item de la lista"}]
```

7.6. Servicios web

Aunque se encuentra fuera del alcance del proyecto, el cual se ciñe a la creación de una aplicación para iPad, se crearán unos servicios web para poder desarrollar y probar la aplicación. El siguiente esquema representa todos elementos participantes:

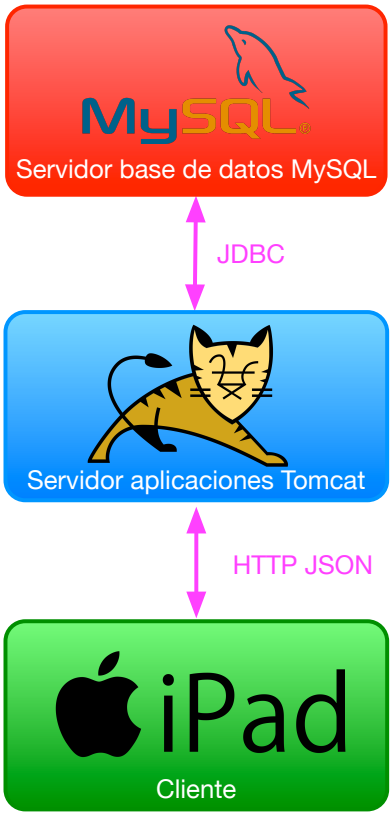


Figura 3: Origen de datos, servicio web y cliente.

Se publican cinco servicios web diferentes, uno para cada tipo de visualizador disponible, los cuales proveen los tipos de datos descritos anteriormente:

- http://tomcat/WebServiceiOS/resources/bar
- http://tomcat/WebServiceiOS/resources/pie
- http://tomcatWebServiceiOS/resources/map
- http://tomcat/WebServiceiOS/resources/semaphore
- http://tomcat/WebServiceiOS/resources/list

Estos datos se obtienen de la base de datos mediante consultas SQL sin ninguna otra capa de intermediación. Dado que los datos de origen están en una base de datos se pueden modificar utilizando herramientas externas y ver como esos cambios se reflejan en la aplicación para iPad.

Se han creado cinco tablas en la base de datos que se ajustan al formato JSON esperado por la aplicación. De esta forma el JSON generado automáticamente satisface los requisitos de la aplicación para iPad. A continuación se muestran las tablas creadas en la base de datos:

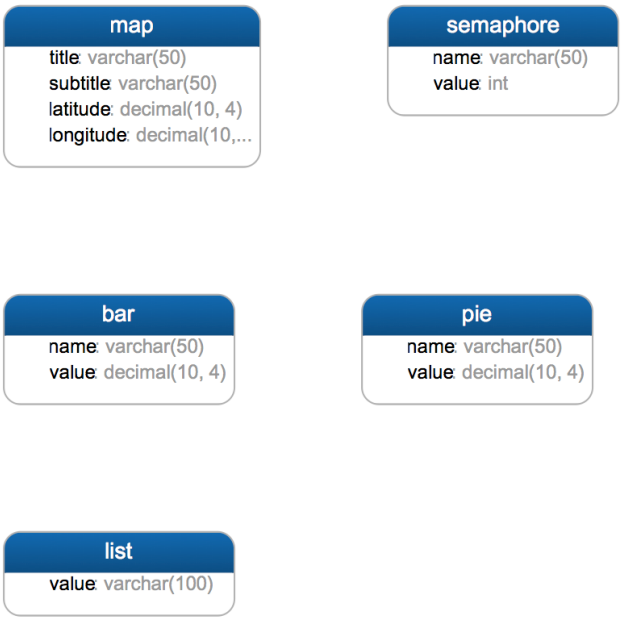


Figura 4: Modelo de datos.

La decisión de usar el servidor de aplicaciones Tomcat junto con la base de datos MySQL ha venido dada por la familiaridad con ambos productos además de la simplicidad a la hora de instalarlos y configurarlos.

8. Prototipo

El diseño de la aplicación se realizó en varias etapas: partiendo desde diseños sobre papel, se pasó a dibujar prototipos de alta fidelidad mediante aplicaciones de diseño gráfico, para finalmente implementar el diseño final sobre el entorno de desarrollo.

8.1. Skecth mano alzada

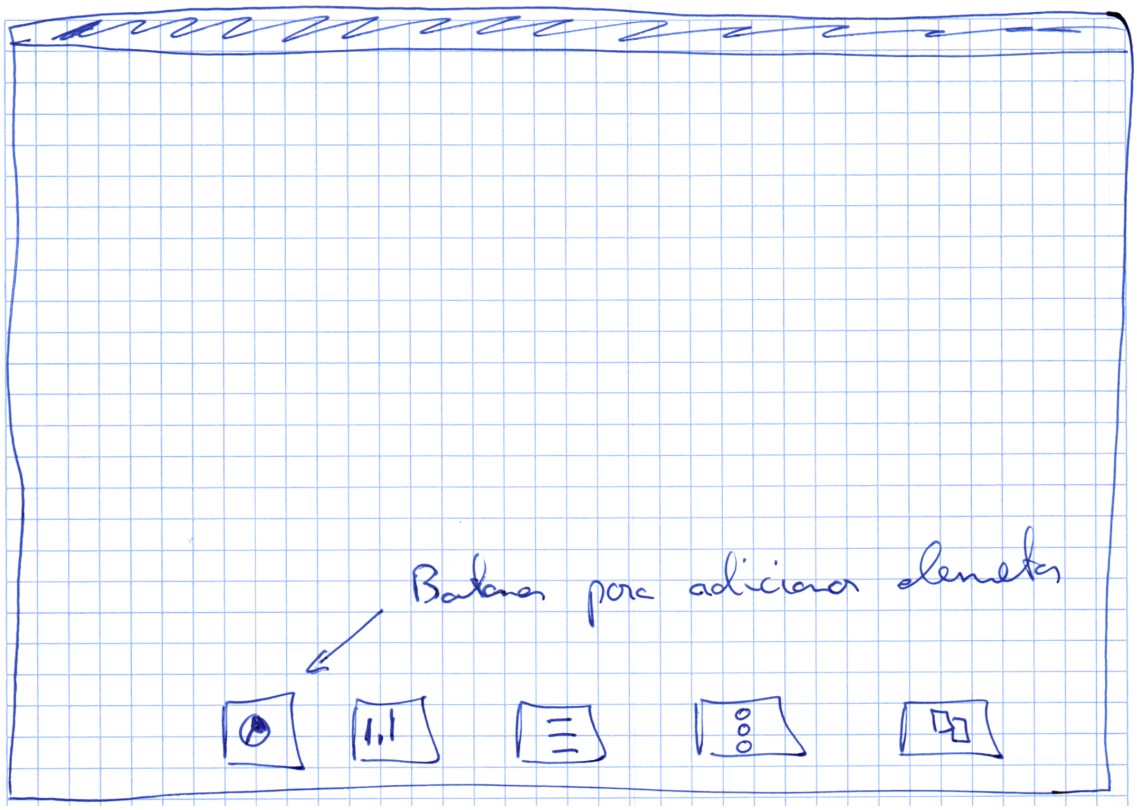


Figura 1: Aspecto inicial de la aplicación, en la que se muestra la barra de iconos que permiten adicionar los distintos elementos de visualización al área de trabajo. Es un planteamiento similar a la barra de aplicaciones de OS X (Dock), donde los usuarios pueden encontrar elementos para conseguir llevar a cabo sus objetivos.

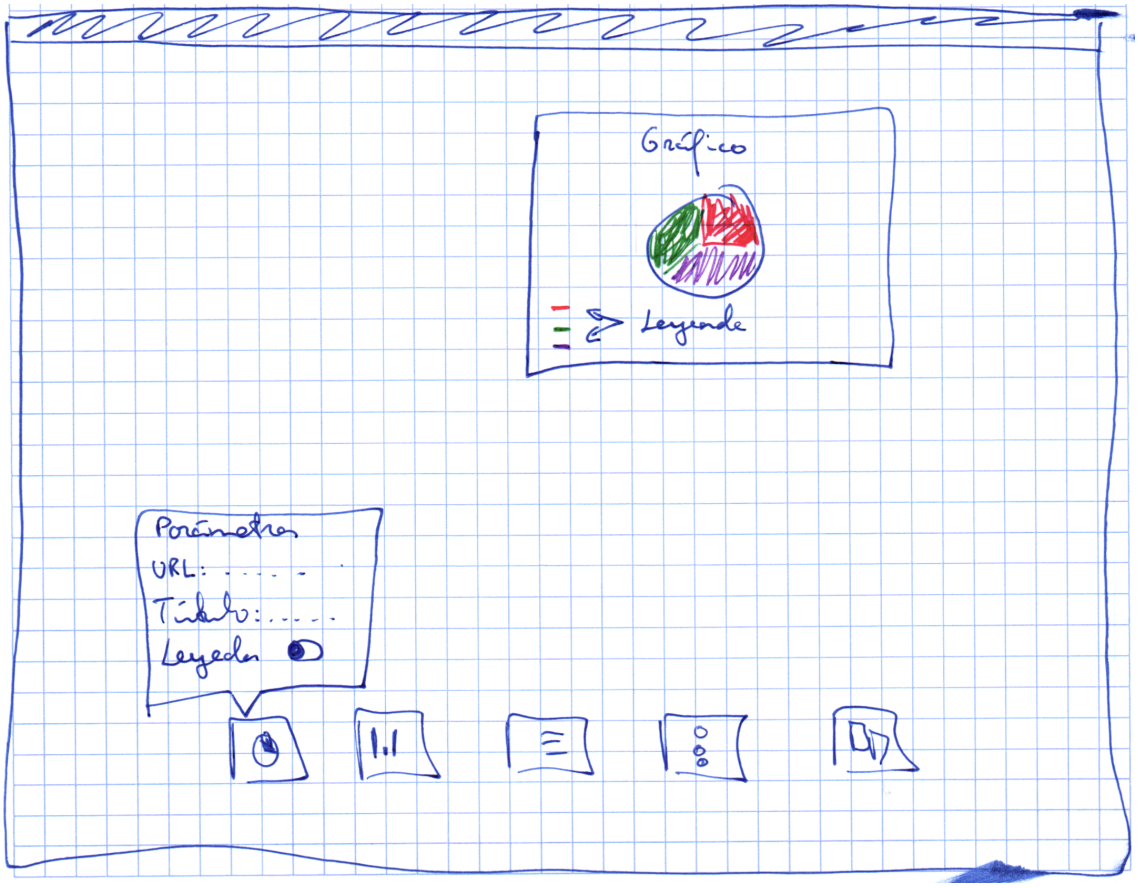


Figura 2: Aplicación ejecutando un elemento de visualización y en proceso de adicionar otro. Una vez que se decide que elemento quiere añadir, el usuario toca sobre este y se le presentan las opciones necesarias para configurarlo. Cuando completa la configuración sólo resta pulsar el botón *Añadir* para que aparezca sobre el área de visualización.

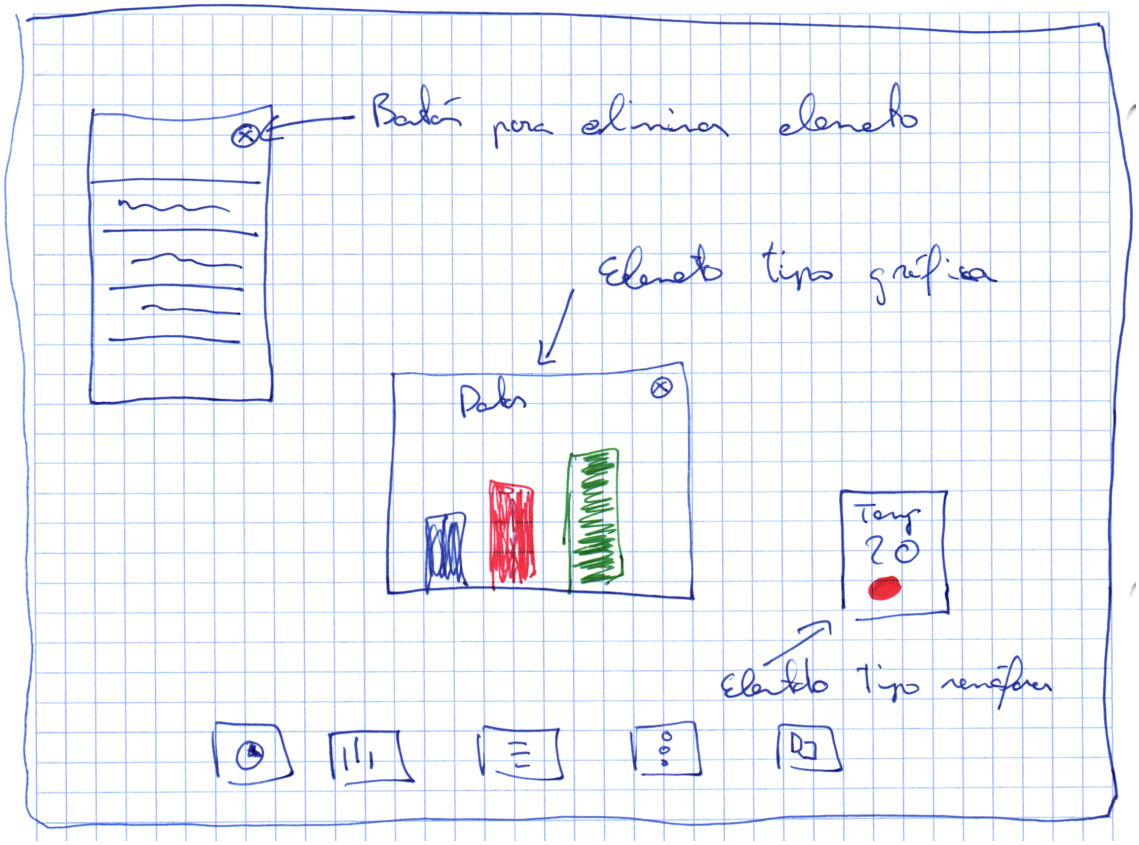


Figura 3: Aplicación mostrando varios tipos de elementos de visualización junto con el método previsto para eliminarlos del área de visualización. Además de poder añadir y eliminar elementos estos se pueden reorganizar simplemente arrastrándolos por el área de visualización de la aplicación.

8.2. Prototipo alta fidelidad

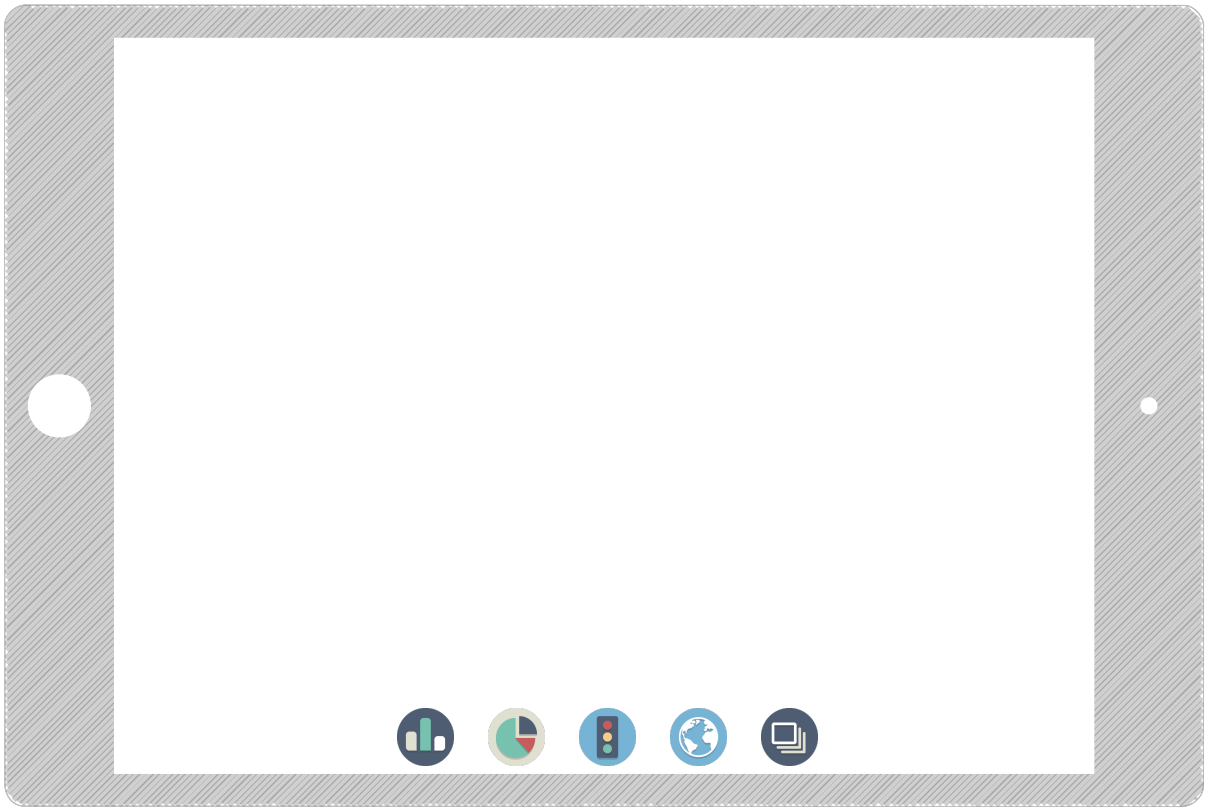


Figura 4: Aspecto inicial de la aplicación, en la que se muestra la barra de iconos que permiten adicionar los distintos elementos de visualización al área de trabajo. Emula el aspecto de la barra de aplicaciones de OS X (Dock), con estos botones se pueden añadir los distintos elementos disponibles para personalizar el cuadro de mando.

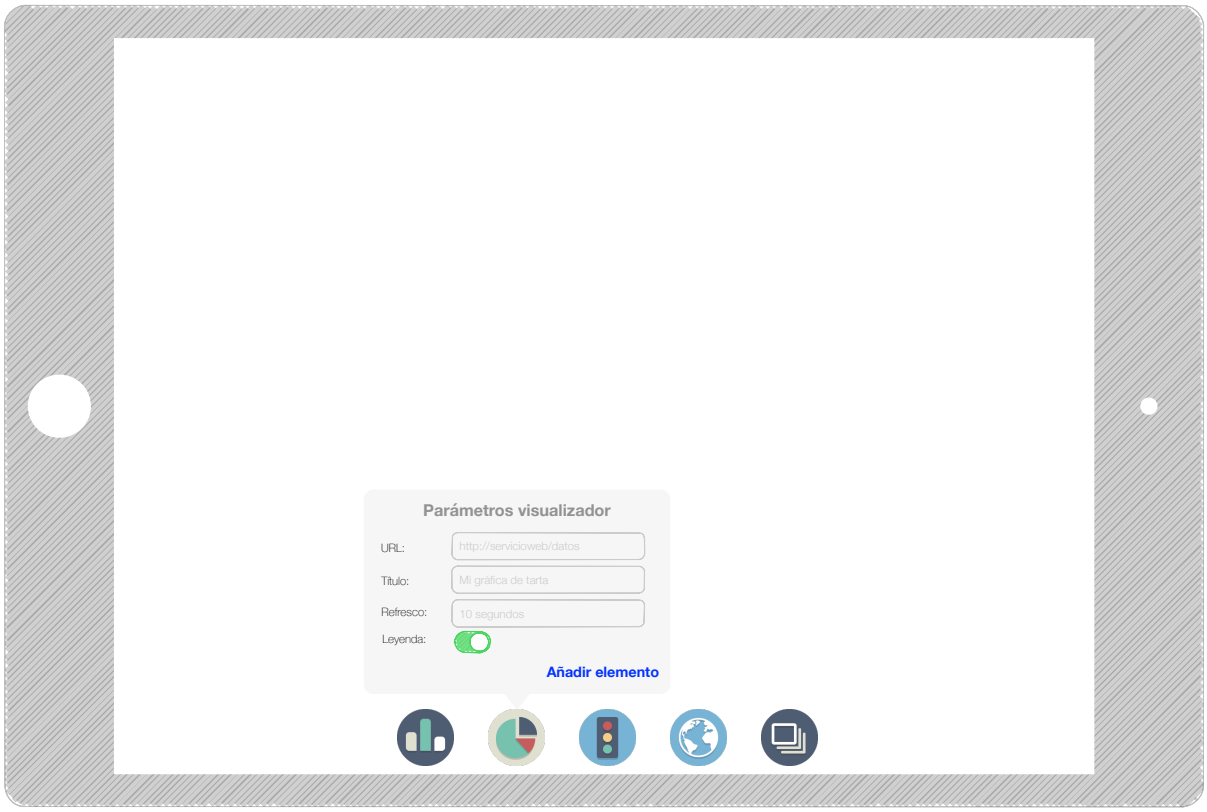


Figura 5: Aplicación mostrando la configuración a la hora de adicionar un nuevo elemento. En concreto se está configurando un elemento de tipo gráfico de tarta. Para ello se solicita un título (opcional), la dirección del servicio web, el tiempo de refresco y la activación / desactivación de la leyenda.

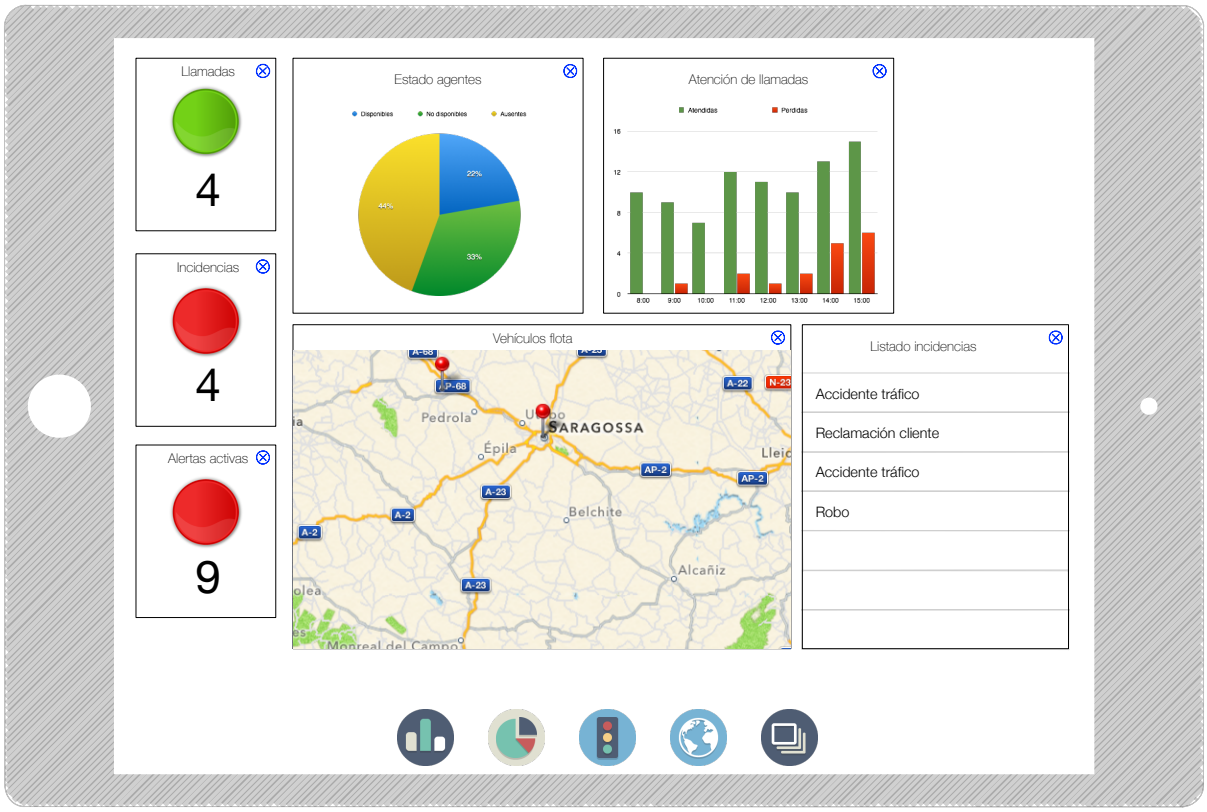


Figura 6: Aplicación en ejecución mostrando todos los tipos de elementos de visualización disponibles. Los diversos elementos se han añadido y reorganizado Notarrastrándolos por el área de visualización de la aplicación.

8.3. Prototipo funcional

Figura 7: Aspecto final de la aplicación ejecutándose sobre el simulador del entorno de desarrollo.

El prototipo final difiere en varios aspectos de los diseñados previamente:

- En lugar de admitir un valor libra para el tiempo de refresco, este se implementa con un deslizador. De esta manera se impiden valores extremos o inválidos.
- En los elementos gráficos de tarta la leyenda se añade siempre, ya de otra forma no se identifica la serie de valores.
- El elemento semáforo es horizontal, pudiendo ser más compacto.
- Los elementos tienen esquinas redondeadas y sin líneas delimitadas.
- Se utilizan los tipos de letra y juegos de colores estándar del sistema.

9. Implementación

9.1. Características generales

La creación de la aplicación para iPad se ha realizado íntegramente con el entorno de desarrollo de Apple Xcode, junto con el SDK de la versión de iOS 8.3 (versión más actual en el momento de comenzar el proyecto). Se ha buscado realizar una aplicación fácil de usar, fácilmente personalizaste por el usuario y robusta.

9.2. Área principal

El área principal de la aplicación se presenta como un lienzo en blanco puesto a disposición del usuario para que este cree una composición a su medida. El interfaz es sencillo: el usuario tiene a su disposición cinco botones en la parte inferior de la pantalla con los que puede configurar y adicionar nuevos elementos de visualización.

Estos botones muestran un elemento de tipo **UIPopoverController** específico para cada tipo de visualizador. El siguiente código muestra como el botón destinado a añadir un nuevo mapa presenta este **UIPopoverController**:

```
@IBAction func graphButtonTouchUpInside(sender: UIButton) {
       var popView = BarSettingsController(nibName: “BarSettingsController",
       bundle: nil)
       var popViewController = UIPopoverController(contentViewController:
       popView)
       popViewController.popoverContentSize = CGSize(width: 250, height: 225)
       popViewController.presentPopoverFromRect(sender.frame, inView: self.view,
       permittedArrowDirections: UIPopoverArrowDirection.Any, animated: true)
}
```

Como se puede observar, se crea un nuevo objeto de tipo **BarSettingsController**, el cual carga su interfaz gráfica desde un archivo **.xib**. Se ha utilizado este esquema a lo largo de todo el proyecto ya que el flujo de funcionamiento de la aplicación no se adaptaba al trabajo con StoryBoards, el cual sólo se ha utilizado en el área principal.

Estos objetos son los que presentan el interfaz gráfico en el que se definen las propiedades del elemento de visualización que se desea añadir al área de visualización. Para ello realizan una llamada al controlador principal (**ViewController**) para que adicione el nuevo objeto con las características deseadas. En el siguiente código de ejemplo se muestra como se añade un nuevo visualizador:

```
func addBarViewController(title:String,url:String,refresh:Float){
       var bar = BarViewController(nibName: "BarViewController", bundle: nil)
       bar.view.frame = CGRectMake(20, 20, bar.view.frame.width,
       bar.view.frame.height)
       self.addChildViewController(bar)
       bar.view.layer.cornerRadius = 5
       bar.view.layer.masksToBounds = true
       self.view.addSubview(bar.view)
       bar.configureSettings(title, url: url, refresh: refresh)
       registerPan(bar)
}
```

Como se ha visto con anterioridad, se utilizan ficheros **.xib** para el interfaz gráfico de los distintos elementos. Cada uno de estos ficheros **.xib** es de una clase específica:

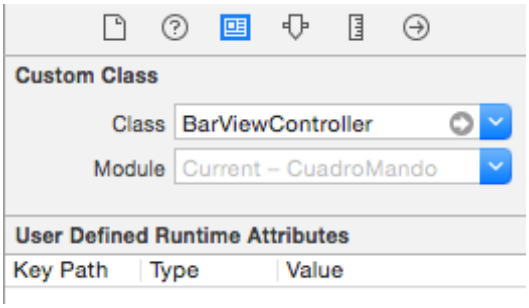


Figura 1: Asociación del tipo de clase de un archivo .**xib**.

Una vez creado el nuevo objeto este se añade como elemento hijo al área principal **(ViewController**) y su vista también se incorpora a la jerarquía de vistas de la aplicación, como una subvista de la vista de **ViewController**.

Como se buscaba una gran capacidad de personalización se tomo la decisión de que cada uno de estos elementos pudieran moverse libremente por el área de visualización. Para ello se implementaron los siguientes métodos, que añaden el reconocimiento del gesto de arrastrar el elemento y trasladarlo a otra ubicación:

```
func registerPan(sender: UIViewController){
        var panGesture = UIPanGestureRecognizer(target: self, action:
        "handlePan:")
        panGesture.maximumNumberOfTouches = 1
        panGesture.minimumNumberOfTouches = 1
        sender.view.addGestureRecognizer(panGesture)
}
func handlePan(sender:UIPanGestureRecognizer){
        self.view.bringSubviewToFront(sender.view!)
        var translation = sender.translationInView(self.view)
        sender.view!.center = CGPointMake(sender.view!.center.x + translation.x,
        sender.view!.center.y + translation.y)
        sender.setTranslation(CGPointZero, inView: self.view)
}
```

9.3. Elementos de configuración

Existen cinco elementos destinados a la configuración y creación de nuevos elementos de visualización. Estos elementos consisten en un fichero **.xib** y una clase de tipo **UIViewController** asociada:

Figura 2: Diseño de un elemento de configuración.

Por su simplicidad no necesitan un modelo de datos, ya que la información de configuración no es almacenada en estos sino que se pasa a los elementos de visualización.

El aspecto es similar en todos los casos aunque varía en función de los distintos parámetros de configuración disponibles para cada visualizador. Todos disponen de un botón “Añadir” el cual realiza una llamada al controlador principal (**ViewController**) para que adicione el nuevo objeto con las características deseadas.

En el siguiente código se puede observar como se accede al controlador principal pasando los datos de configuración del nuevo elemento de visualización y se cierra el propio controlador una vez añadido este último:

```
@IBAction func addElement(sender: UIButton) {
       (self.presentingViewController as!
       ViewController).addBarViewController(visualizerTitle.text!, url:
       url.text, refresh: refreshValue)
        self.presentingViewController?.dismissViewControllerAnimated(true,
       completion: {})
}
```

9.4. Elementos de visualización

Todos los elementos de visualización siguen el patrón MVC:

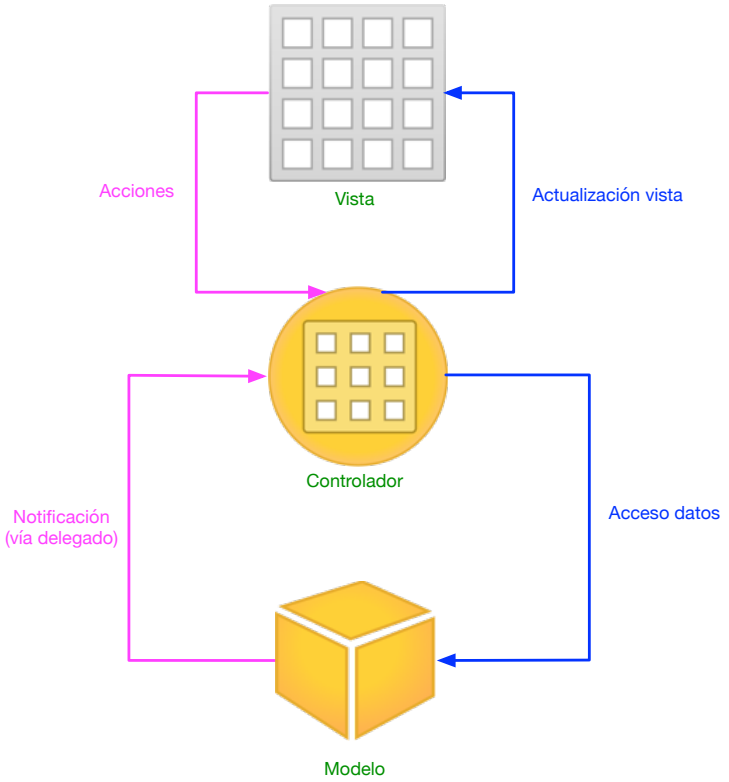


Figura 3: MVC en los elementos de visualización.

La vista y el controlador se crean mediante un fichero **.xib** y una clase de tipo **UIViewController** asociada:

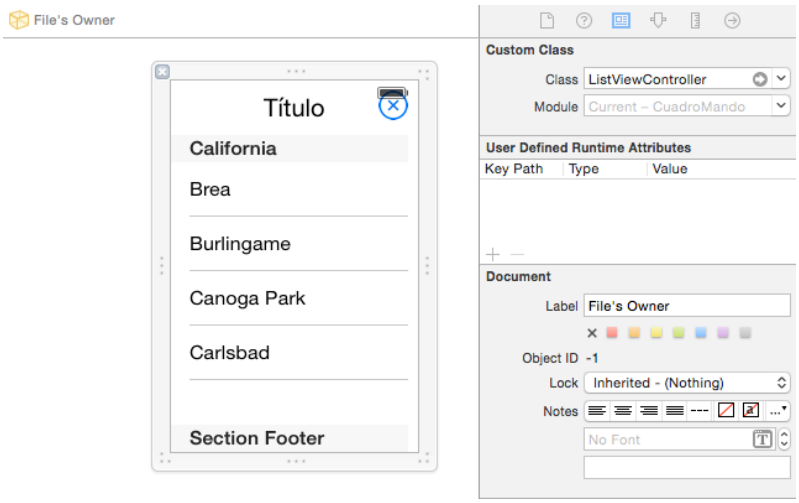


Figura 4: Diseño de un elemento de visualización.

El modelo se encarga de obtener la información, consultando servicios web, así como de almacenarla. El controlador contiene al modelo, siendo además delegado del mismo, así cuando el modelo tiene datos nuevos puede notificar al controlador para que este realice las aciones necesarias para actualizar la vista. El controlador actúa sobre la vista para que esta muestre la información actualizada, bien manipulando esta directamente, bien indicándole que consulte al modelo los datos.

En todos los casos el modelo accede a los web service cada cierto lapso de tiempo, el cual se establece en la configuración previa. Este tiempo oscila entre un mínimo de diez segundos y un máximo de dos minutos. Para la implementación se utiliza un objeto **NSTimer**:

```
func startRequestingData () {
      requestTimer = NSTimer.scheduledTimerWithTimeInterval(refresh!,
       target:self, selector: Selector("requestData"), userInfo: nil, repeats:
       true)
}

func stopRequestingData() {
       requestTimer?.invalidate()
}
```

Se controla tanto el inicio como la finalización de este, ya que debe detenerse para poder liberar recursos cuando el elemento es eliminado del área de visualización.

La obtención de los datos accediendo al web service se realiza de forma asíncrona para no interrumpir la ejecución normal de la aplicación. Una vez recibidos los datos se valida que el contenido al menos es formato JSON (la verificación del formato concreto es específica en cada visualizador y se realiza en el momento de usar los datos para representarlos).

Cuando se ha comprobado que los datos recibidos están en formato JSON se realiza la notificación al controlador, utilizando para ello un delegado específico para cada tipo de visualizador.

En el siguiente fragmento de código se puede observar como se realiza la petición, validación del formato JSON y notificación mediante el delegado:

```
func requestData () {

       let url:NSURL = NSURL(string: self.url!)!

       var request: NSURLRequest = NSURLRequest(URL: url)

       let queue:NSOperationQueue = NSOperationQueue()

       NSURLConnection.sendAsynchronousRequest(request, queue: queue,
completionHandler:{ (response: NSURLResponse!, data: NSData!, error: NSError!) -
> Void in

       if (!(nil==data)) {

             if let json = NSJSONSerialization.JSONObjectWithData(data,
options: NSJSONReadingOptions.MutableContainers, error: nil) as?
Array<NSDictionary> {

                   self.dataRecived = json

                   if let maxValue = (self.dataRecived as
AnyObject).valueForKeyPath("@max.value") as? Double {

                       self.delegate?.dataAvailable(maxValue)

                   }

               }

           }

        })

    }
```

9.4.1. Visualizador gráfico de barras

Está constituido por tres elementos:

- Vista: **BarViewController.xib**
- Controlador: **BarViewController**
- Modelo: **BarModel**

El siguiente diagrama muestra la relación entre los distintos elementos y como interactúan entre ellos:

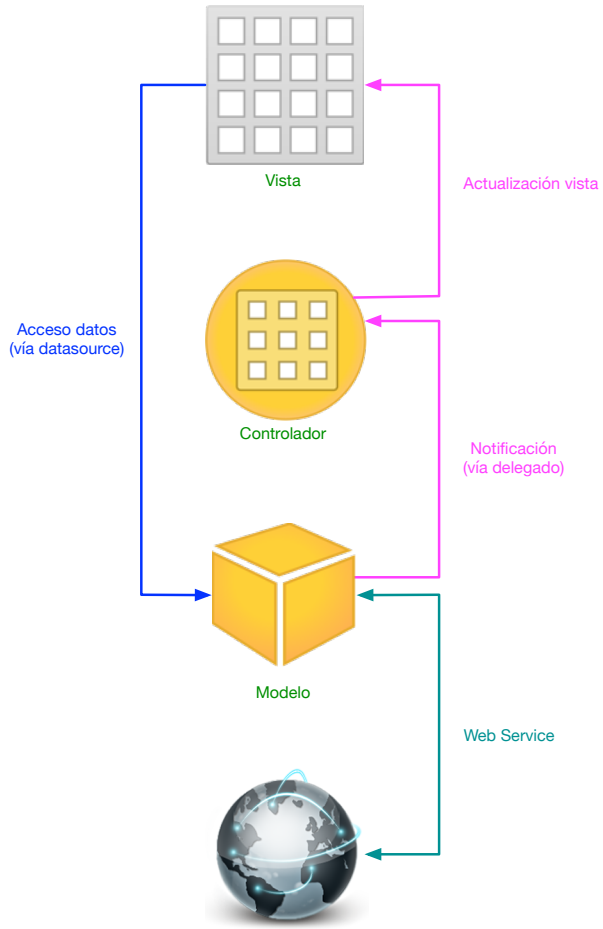


Figura 5: Flujo de funcionamiento.

Para la implementación de este elemento se utiliza el framework Core Plot, en concreto el tipo de gráfica **CPTBarPlot**.

La obtención de los datos para crear el gráfico se realiza mediante un datasource (**CPTBarPlotDataSource**), el cual es implementado por el modelo. Así pues cuando existen datos el modelo notifica al controlador y este actúa sobre el objeto de tipo **CPTBarPlot** indicándole que recargue los datos:

```
func dataAvailable(maxVaulue: Double){
       dispatch_async(dispatch_get_main_queue(), {
            if !self.graphAdded {
                self.addChart()
                self.graphAdded = true
       }
       let plotSpace = self.barPlot.graph.defaultPlotSpace as! CPTXYPlotSpace
       plotSpace.yRange = CPTPlotRange(location:0.0, length:maxVaulue+5)
       plotSpace.xRange = CPTPlotRange(location:-0.5,
       length:self.model.dataRecived.count)
self.barPlot.reloadPlotData()
})
    }
```

Como se realiza una actualización del interfaz gráfico es necesario asegurar que esta se realiza en el hilo principal de la aplicación. De otra manera el refresco del interfaz puede ocurrir con mucho retraso o directamente no realizarse.

En este tipo de gráfico es necesario calcular el rango de valores a representar. Desde el modelo se obtiene el valor máximo existente para asociarla a la gráfica:

```
if let maxValue = (self.dataRecived as AnyObject).valueForKeyPath("@max.value")
as? Double {
       self.delegate?.dataAvailable(maxValue)
}
```

Dado que el modelo es el origen de datos del gráfico, implementa las funciones necesarias para que este pueda pintar el gráfico de barras:

```
func numberForPlot(plot: CPTPlot!, field: UInt, recordIndex: UInt) -> AnyObject!
{
        if Int(recordIndex) > self.dataRecived.count {
            return nil
        } else {
            switch CPTBarPlotField(rawValue: Int(field))! {
            case .BarLocation:
                return recordIndex as NSNumber
            case .BarTip:
                if let value =
self.dataRecived[Int(recordIndex)].objectForKey("value") as? NSNumber {
                    return value
                } else {
                    return NSNumber(float: 0.0)
                }
            default:
                return nil
            }
        }
    }
```

9.4.2. Visualizador gráfico de tarta

Está constituido por tres elementos:

- Vista: **ChartViewController.xib**
- Controlador: **ChartViewController**
- Modelo: **ChartModel**

El siguiente diagrama muestra la relación entre los distintos elementos y como interactúan entre ellos:

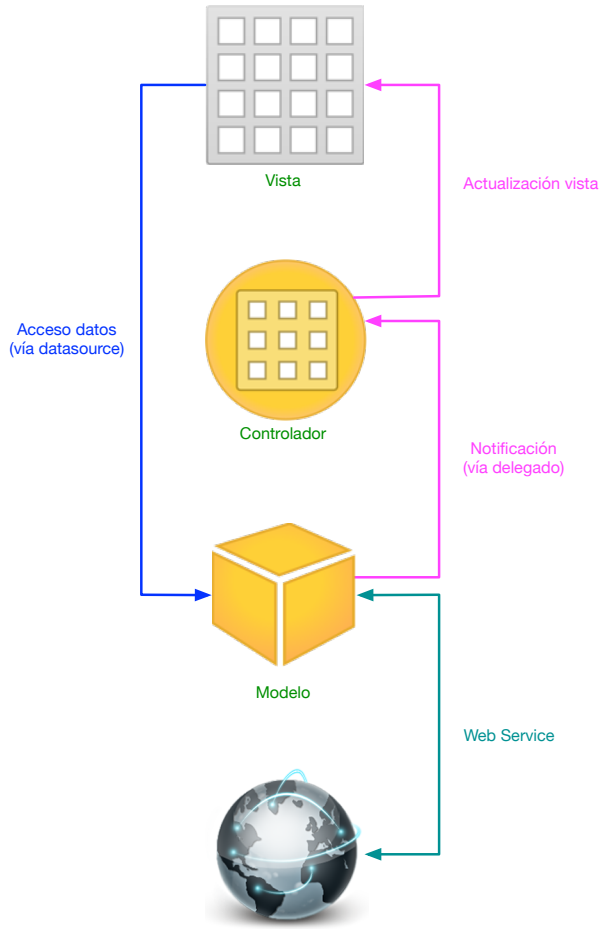


Figura 6: Flujo de funcionamiento.

Para la implementación de este elemento se utiliza el framework Core Plot, en concreto el tipo de gráfica **CPTPieChart**.

Su funcionamiento es análogo al anteriormente descrito para el elemento de visualización de tipo gráfica de barras. La diferencia principal radica en la configuración del propio tipo de gráfica y la obtención de datos para esta, siendo en este caso necesario implementar el protocolo **CPTPieChartDataSource**:

```
func numberOfRecordsForPlot(plot: CPTPlot!) -> UInt {
       return UInt(self.dataRecived.count)
}

func numberForPlot(plot: CPTPlot!, field: UInt, recordIndex: UInt) -> AnyObject!
{
       if Int(recordIndex) > self.dataRecived.count {
              return nil
        } else {
              switch CPTPieChartField(rawValue: Int(field))! {
              case .SliceWidth:
                 if let value =
                 self.dataRecived[Int(recordIndex)].objectForKey("value") as?
                 NSNumber {
                     return value
                 } else {
                     return NSNumber(float: 0.0)
                 }
                 default:
                 return recordIndex as NSNumber
            }
        }
    }
```

```
func dataLabelForPlot(plot: CPTPlot!, recordIndex: UInt) -> CPTLayer!
{
       var label = CPTTextLayer(text: "")
        if let textLabel =
       self.dataRecived[Int(recordIndex)].objectForKey("name") as? String {
              label.text = textLabel
        }
        let textStyle = label.textStyle.mutableCopy() as! CPTMutableTextStyle
        textStyle.color = CPTColor.lightGrayColor()
        label.textStyle = textStyle
        return label
    }
}
```

9.4.3. VISUALIZADOR SEMÁFORO

Está constituido por tres elementos:

- Vista: **SemaphoreViewController.xib**
- Controlador: **SemaphoreViewController**
- Modelo: **SemaphoreModel**

El siguiente diagrama muestra la relación entre los distintos elementos y como interactúan entre ellos:

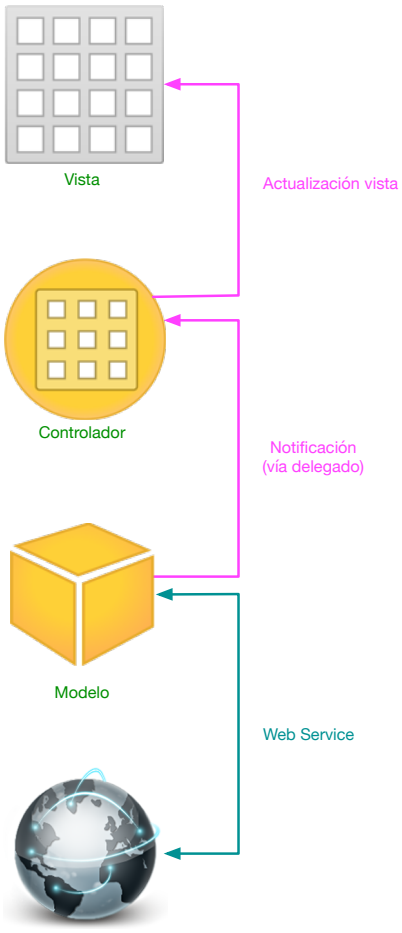


Figura 7: Flujo de funcionamiento.

Este elemento difiere en varios aspectos con los anteriores:

- No es necesaria la librería Core Plot
- El controlador manipula la vista directamente y esta no consulta los datos directamente åal modelo.

Cuando el modelo notifica al controlador de la existencia de datos, este los verifica y manipula los elementos contenidos en la vista para mostrar la información que representan:

```
func dataAvailable(){
       if let value:Float = (model.dataRecived[0].objectForKey("value") as?
       Float) {
       dispatch_async(dispatch_get_main_queue(), {
        self.visualizerValue.text! = String(stringInterpolationSegment: value)
        if value<self.model.minValue  {
              self.semaphoreImage.image = UIImage(named:"CircleGreen")!
       }
       else if (value>self.model.minValue && value<self.model.maxValue) {
              self.semaphoreImage.image = UIImage(named:"CircleYellow")!
       } else if (value>self.model.maxValue){
              self.semaphoreImage.image = UIImage(named:"CircleRed")!
      }
     })
    }
}
```

En el código mostrado se puede observar la comprobación de la existencia de datos denominados “value” y de tipo **Float** en el JSON recibido y almacenado por el modelo. Esto se realiza mediante el patrón **if - let** para evitar errores si estos no están presentes o no son del tipo adecuado.

9.4.4. VISUALIZADOR MAPA

Está constituido por tres elementos:

- Vista: **MapViewController.xib**
- Controlador: **MapViewController**
- Modelo: **MapModel**

El siguiente diagrama muestra la relación entre los distintos elementos y como interactúan entre ellos:

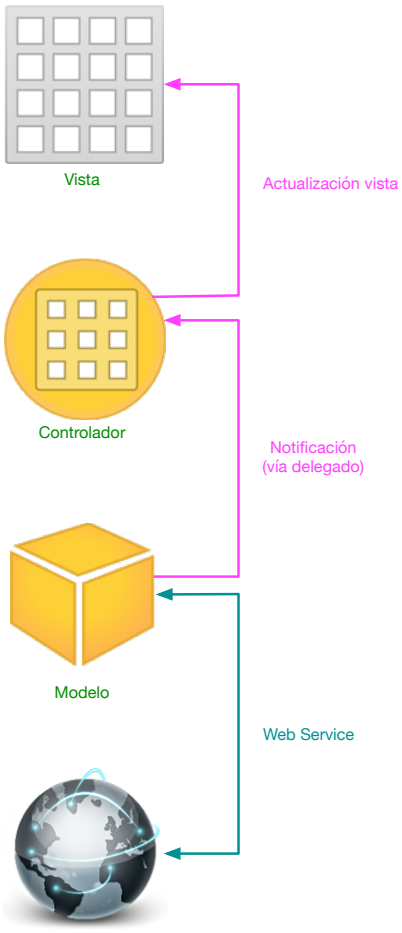


Figura 8: Flujo de funcionamiento.

Este elemento se comporta de forma similar al anterior difiriendo principalmente en la información mostrada y la forma de tratarla. En este caso se utiliza un objeto de tipo **MKMapView** para representar el mapa.

Sobre el objeto de tipo **MKMapView** se añaden las anotaciones, las cuales son objetos de tipo **MKPointAnnotation**. Los datos para generar estas proceden del modelo, que el controlador interroga para recuperar la información y mostrarla sobre el mapa:

```
func dataAvailable(){
       dispatch_async(dispatch_get_main_queue(), {
       self.mapa.removeAnnotations(self.mapa.annotations)
       for register in self.model.dataRecived {
if let latitude:CLLocationDegrees = register.objectForKey("latitude") as?
CLLocationDegrees,longitude:CLLocationDegrees =
register.objectForKey("longitude") as? CLLocationDegrees, title:String =
register.objectForKey("title") as? String,subTitle:String =
register.objectForKey("subtitle") as? String  {
var latitude:CLLocationDegrees = register.objectForKey("latitude") as!
CLLocationDegrees
var longitude:CLLocationDegrees = register.objectForKey("longitude") as!
CLLocationDegrees
var title:String = register.objectForKey("title") as! String
var subTitle:String = register.objectForKey("subtitle") as! String
var annotation = MKPointAnnotation()
annotation.coordinate = CLLocationCoordinate2D(latitude:
latitude,longitude:longitude)
annotation.title = title
annotation.subtitle = subTitle
self.mapa.addAnnotation(annotation)
                }
            }
        })
    }
```

Para ello recorre todos los datos existentes buscando los valores de longitud, latitud, título y subtítulo. Un detalle interesante, introducido en Swift 1.2, es la posibilidad de utiliza el patrón **if - let** con múltiples constantes en una sola línea. Antes era necesario anidar estas comprobaciones, resultando en un código más complejo.

9.4.5. VISUALIZADOR LISTA

Está constituido por tres elementos:

- Vista: **ListViewController.xib**
- Controlador: **ListViewController**
- Modelo: **ListModel**

El siguiente diagrama muestra la relación entre los distintos elementos y como interactúan entre ellos:

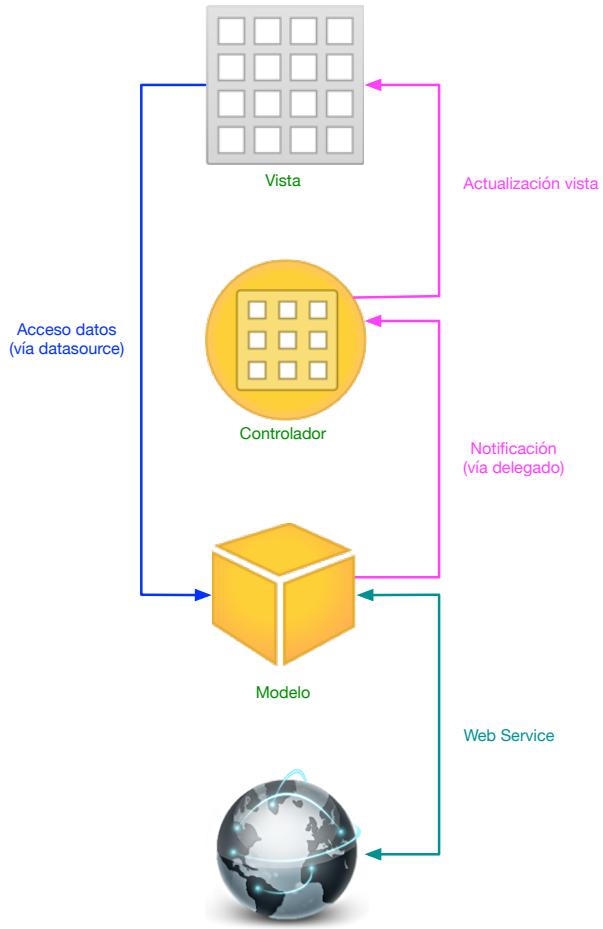


Figura 9: Flujo de funcionamiento.

Su funcionamiento es similar al de los elementos de visualización de gráficas. A diferencia de estos no utiliza la libreria Core Plot, pero la vista si obtiene los datos directamente desde el modelo mediante un datasource. En concreto se implementa el protocolo **UITableViewDataSource** en el modelo:

```
func numberOfSectionsInTableView(tableView: UITableView) -> Int {
        return 1
}

func tableView(tableView: UITableView, numberOfRowsInSection section: Int) ->
Int {
        return dataRecived.count
    }

func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath:
NSIndexPath) -> UITableViewCell {
       let cell = UITableViewCell()
       if let value = (dataRecived[indexPath.row] as
       NSDictionary).objectForKey("value") as? String {
       cell.textLabel?.text = value
      }
       return cell
    }
```

9.5. Servicios web de ejemplo

Para la implementación de los web services de ejemplo se ha buscado la máxima sencillez, ya que su único propósito es servir como origen de datos para las pruebas y demostraciones de la aplicación para iPad.

Sólo son necesarias dos clases para la implementación:

- **DataAccess**: generadora de los servicios web en si mismos.
- **DataBaseHandler**: encargada de gestionar la conexión con la base de datos a través de un pool de conexiones.

Se implementan únicamente los métodos GET de los web services, siguiendo en el siguiente planteamiento en todos los casos:

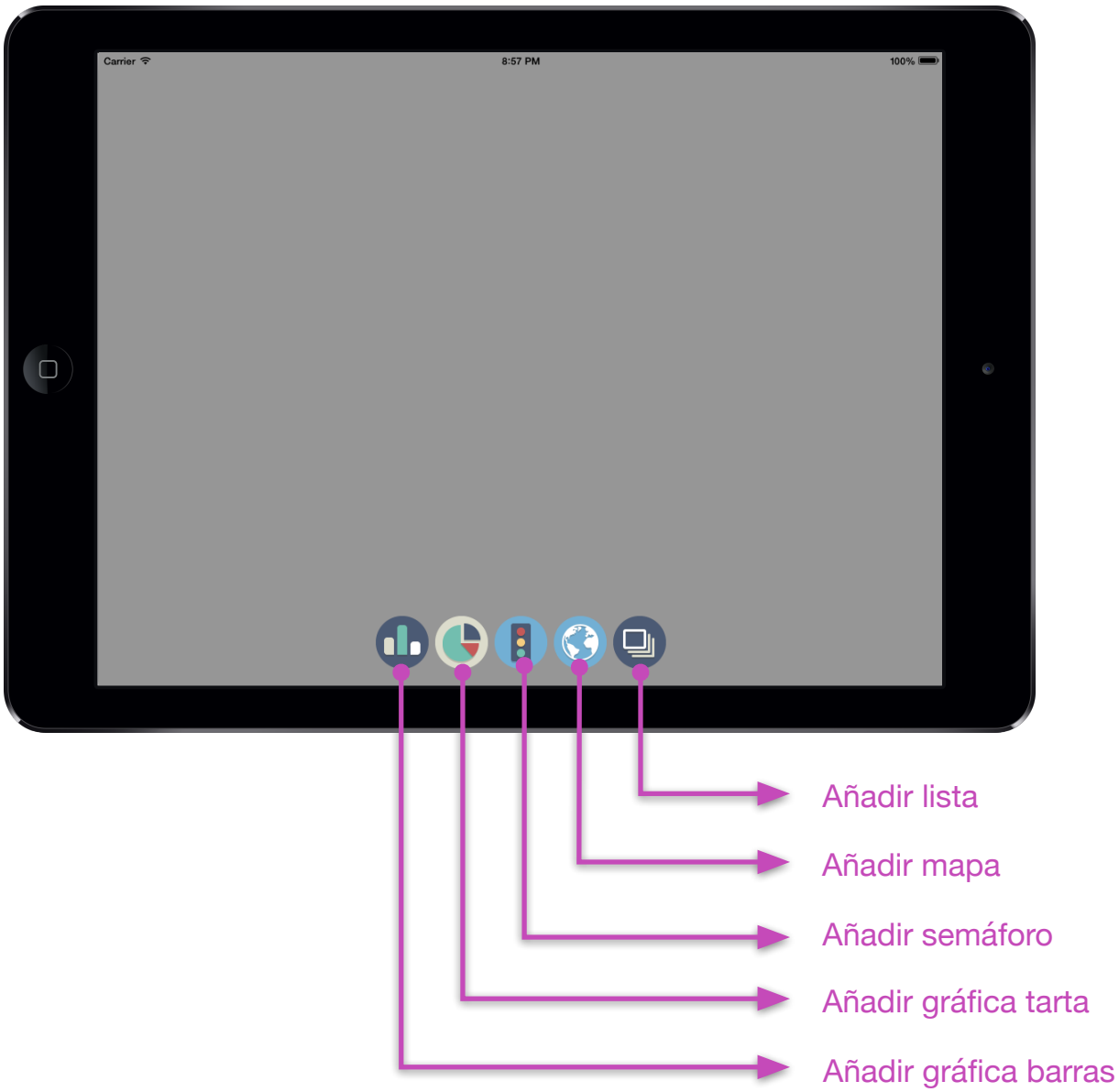
```
@GET
@Path("semaphore")
@Produces("application/json")
public String getSemaphore() {
        try {
            ResultSet rs = db.executeSQLwithResultSet("SELECT * FROM semaphore");
            String result = getJson(rs).toString();
            db.closeConnection();
            return result;
        } catch (Exception ex) {
            Logger.getLogger(DataAccess.class.getName()).log(Level.SEVERE, null, ex);
            db.closeConnection();
            return null;
        }
    }
```

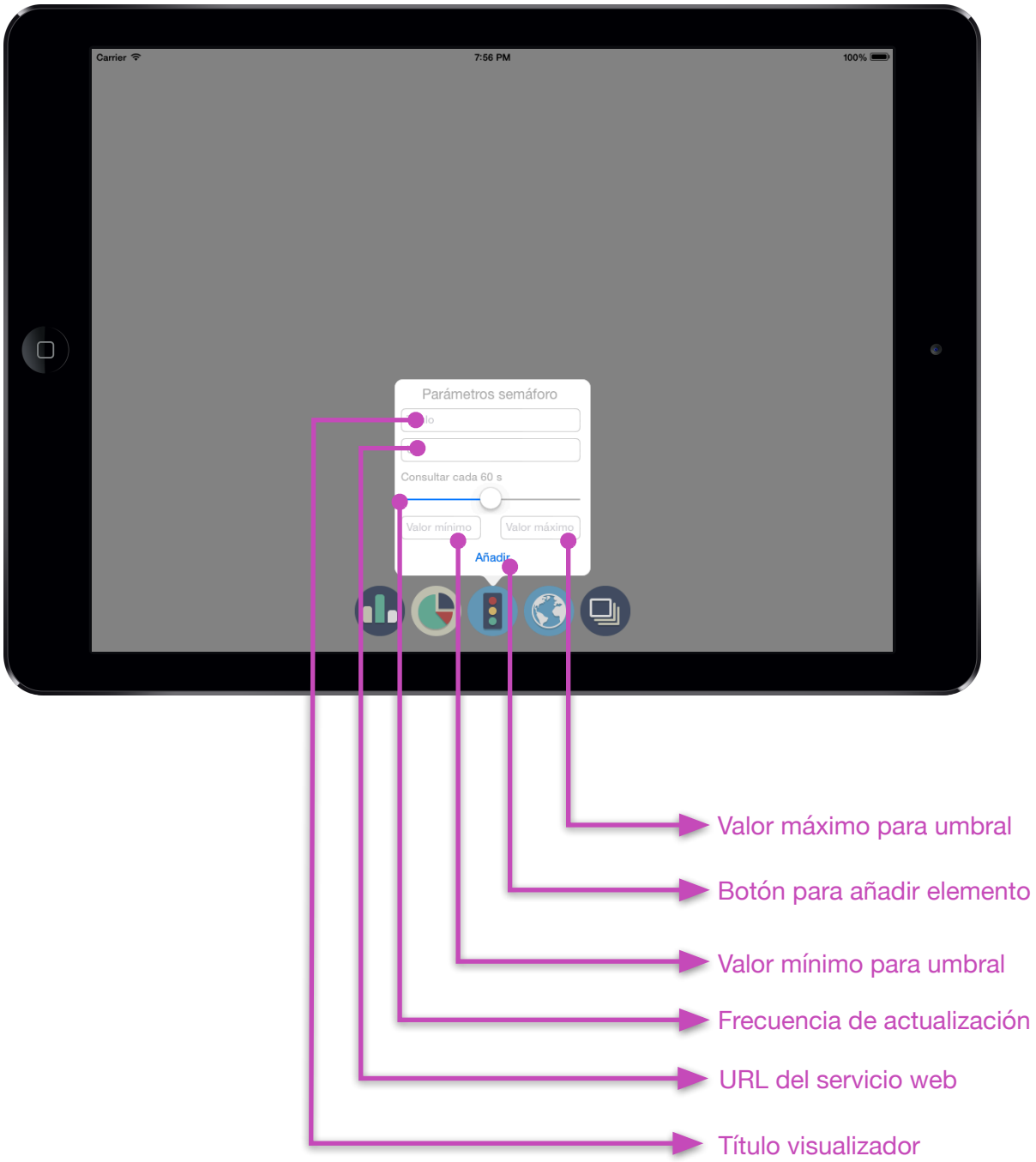
Como las tablas se han creado teniendo en cuenta el formato descrito en el diseño técnico el JSON generado automáticamente satisface los requisitos de la aplicación para iPad.

10. Uso

El manejo de la aplicación es muy sencillo. A lo largo de las siguientes capturas de pantalla se describe su aspecto y uso.

En la siguiente imagen se puede observar el aspecto inicial de la aplicación, en el que se muestra la barra de iconos que permiten adicionar los distintos elementos de visualización al área de trabajo:

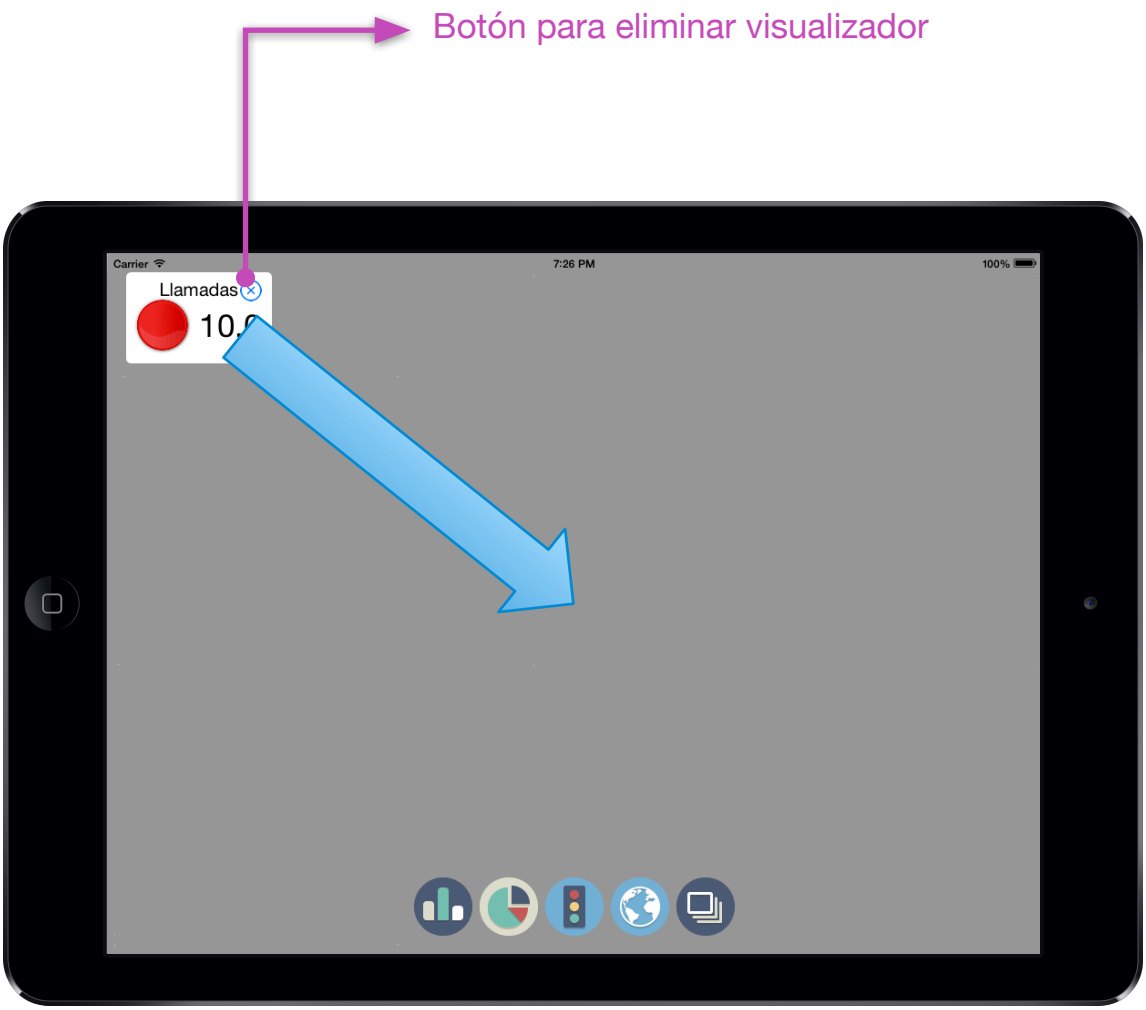




Aquí se puede observar la configuración y adición de un nuevo elemento de visualización, en concreto uno de tipo semáforo.

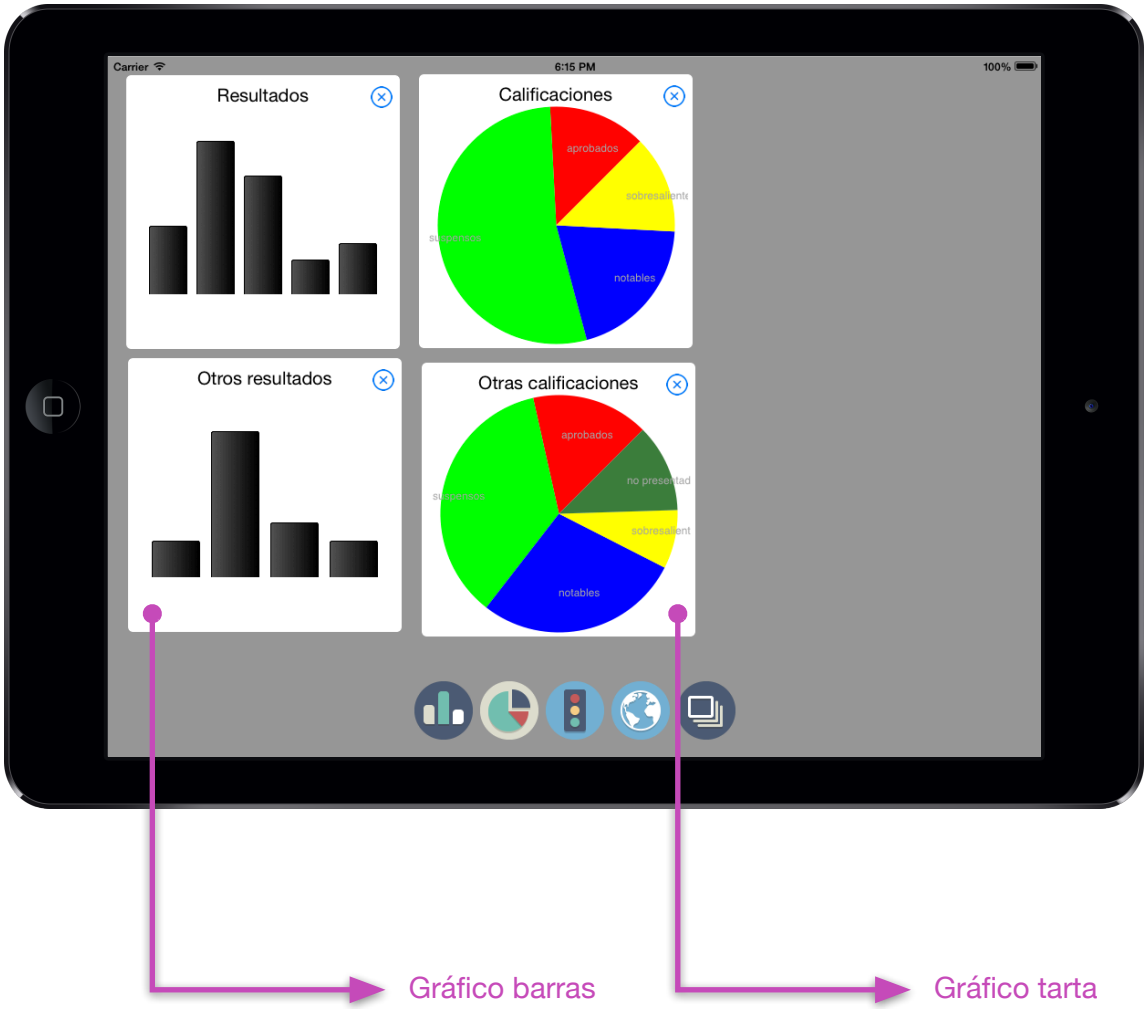
Este elemento muestra un indicador de color verde (por debajo del umbral), amarillo (entre el valor mínimo y máximo del umbral) o rojo (valor por encima del umbral), además del valor numérico. Es necesario al menos indicar la URL del servicio web junto con los valores máximo y mínimo del umbral. Si no se varía la frecuencia esta toma el valor por defecto de refrescar los datos cada minuto. El título también es opcional, si no está presente no se muestra ninguno.

Los elementos pueden redistribuirse libremente por la pantalla. Para ello basta realizar el gesto de arrastrar y soltar:



Así mismo cualquier elemento puede eliminarse en un momento dado haciendo uso del botón superior derecho con forma de aspa.

Cada elemento configura su origen de datos de forma independiente, pudiendo añadirse cuantos se deseen de cualquier tipo. Aquí se pueden observar los tipos de visualizadores de gráfica de barras y gráfica de tarta:



El único factor limitante es el propio espacio de visualización, aunque los elementos pueden solaparse y reordenarse en un momento dado para consultar la información.

Además de los tipos de visualizadores de gráfica de barras y gráfica de tarta hay otros tres disponibles, que se pueden ver a continuación:

Los de tipo semáforo muestran un valor numérica así como una representación visual con código de colores indicando en que umbral se sitúa el valor mostrado. Los de tipo lista muestran un conjunto de elementos en una tabla, pudiendo ser los datos de cualquier tipo. Los mapas colocan anotaciones con los datos recibidos. Estas anotaciones deben proporcionar la longitud, latitud, un título y un subtítulo (opcional).

Aquí se puede observar la aplicación ejecutando un total de nueve visualizadores, incluyendo al menos uno de cada tipo:

11. Resultado del proyecto

11.1. Resumen

El desarrollo de la aplicación ha sido muy interesante y educativo. Era un reto doble ya que apenas conocía con anterioridad el entorno de desarrollo, a lo que se sumaba un lenguaje de programación totalmente nuevo y por ende desconocido. En general ha resultado más complejo de lo inicialmente previsto, pero por otro lado ha resultado gratificante.

En particular la integración y uso del framework para generación de gráficos Core Plot ha sido bastante complejo. Respecto al primer aspecto, ha sido necesario vincular el proyecto Core Plot con el proyecto de la propia aplicación, además al estar este programado en Objective-C se han tenido que crear elementos “puente” para que fuera posible acceder desde la aplicación programada en Swift. A su vez el propio framework no resulta sencillo de utilizar, contando con escasos ejemplos en lenguaje Swift.

11.2. Cumplimiento de los objetivos

Se ha logrado cumplir los objetivos definidos al comienzo del proyecto, tanto desde el punto de vista de prestaciones previstas e implementadas en la propia aplicación como desde el punto de vista de adquisición de conocimientos.

Todos los hitos definidos se han cumplido en plazo, aunque algunas tareas han resultado más difíciles de lo inicialmente previsto. No obstante no se ha podido realizar un control exhaustivo del tiempo empleado siendo bastante probable que se hayan superado las estimaciones de dedicación inicialmente previstas.

Un aspecto que se ha tenido muy en cuenta ha sido la robustez y buen funcionamiento de la aplicación. No se ha buscado simplemente implementar las prestaciones planteadas al comienzo del proyecto, sino que estas estuvieran muy probadas y que la aplicación pudiera tolerar errores como datos mal formados o servicios web erróneos (por ejemplo por una equivocación en la URL). Para ello se han utilizado tanto herramientas de análisis de rendimiento (Instruments) como test unitarios (XCTest).

11.3. Posibles mejoras

Durante el desarrollo de la aplicación se ha observado que existen diversas vías de posible mejora o ampliación como por ejemplo:

Añadir nuevos elementos de visualización.

Existen muchas posibilidades en este aspecto, desde elementos similares a los ya existentes como por ejemplo otros tipos de gráfica hasta componentes para comprobar el correo o mostrar información de redes sociales.

Permitir otros orígenes de datos como XML.

Aunque JSON está muy extendido en la actualidad, existen muchos servicios que en su día se desarrollaron e implementaron mediante XML, ya que era lo más habitual en ese momento. Además no sólo hay que tener en cuenta el formato en si de los datos, sino el tipo de servicio web: es muy habitual encontrar servicios web SOAP productores de XML.

Incrementar las posibilidades de personalización de cada uno de los visualizadores, como por ejemplo poder variar el tamaño.

Probablemente lo más claro en este aspecto sea la modificación del tamaño, pero además se podrían alterar otros aspectos como el juego de colores utilizado por las gráficas, tipos de letra, temas, etc.

Añadir otros medios de comunicación como sockets TCP.

La aplicación parte de la premisa de consultar iterativamente un servicio web para obtener datos. Esto es adecuado si los datos no varían con frecuencia o si con los intervalos de actualización previstos la información es suficientemente actual, pero puede darse el caso de necesitarse una mayor inmediatez o directamente querer acceder a los datos en tiempo real. Para ello sería útil adicionar medios de comunicación adicionales como sockets TCP, UDP o incluso web sockets.

Añadir múltiples áreas de visualización para no verse limitado a una única pantalla.

De esta forma un usuario podría componer un cuadro de mando aún más extendido. Aunque no toda la información fuera visible simultáneamente sería muy útil disponer de los visualizadores ya funcionando y con información actualizada para poder revisarlos simplemente cambiando de pantalla.

Implementar persistencia de la configuración para que se esta se recupere al volver a arrancar la aplicación.

Actualmente la aplicación pierde su configuración al cerrarse. Poder conservarla resultaría muy útil, llegado al caso incluso se podría implementar un sistema para gestionar y almacenar configuraciones o perfiles de uso, para adaptarse más rápidamente a un escenario de uso

Notificaciones.

Particularmente los elementos como los semáforos podrían generar notificaciones locales a nivel de sistema operativo, por ejemplo al superar el umbral definido.

Todos estas posibles mejoras son muy interesantes y convertirían el producto en uno con muchas más posibilidades, pero debido al tiempo disponible no han podido llevarse a cabo.

12. Fuentes de información

12.1. Bibliografía

The Swift Programming Language. **Apple Inc.**

Using Swift with Cocoa and Objective-C. **Apple Inc.**

iOS Human Interface Guidelines. **Apple Inc.**

12.2. Cursos iTunes

Developing iOS 8 Apps with Swift. **Paul Hegarty, Stanford University.**

12.3. Recursos web

Apple Developer Connection: https://developer.apple.com. **Apple Inc.**

Core Plot: https://github.com/core-plot/core-plot. **Eric Skroch y otros.**

Ray Wenderlich: http://www.raywenderlich.com. **Diversos autores.**

Stackoverflow: http://stackoverflow.com. **Stack Exchange Network.**

We ❤ Swift: https://www.weheartswift.com **Diversos autores.**

Wikipedia: http://www.wikipedia.org. **Diversos autores.**