

Desarrollo para dispositivos iOS

Pau Ballada

PID_00209911



Los textos e imágenes publicados en esta obra están sujetos –excepto que se indique lo contrario– a una licencia de Reconocimiento-NoComercial-SinObraDerivada (BY-NC-ND) v.3.0 España de Creative Commons. Podéis copiarlos, distribuirlos y transmitirlos públicamente siempre que citéis el autor y la fuente (FUOC. Fundació para la Universitat Oberta de Catalunya), no hagáis de ellos un uso comercial y ni obra derivada. La licencia completa se puede consultar en <http://creativecommons.org/licenses/by-nc-nd/3.0/es/legalcode.es>

Índice

1. Introducción al Xcode, Objective-C e iOS.....	5
1.1. Introducción iOS	5
1.1.1. Distribución de aplicaciones iOS	6
1.2. Introducción iOS SDK	10
1.2.1. Core OS	11
1.2.2. Core Services	11
1.2.3. Media	15
1.2.4. Cocoa Touch	17
1.3. Introducción Xcode	20
1.4. Introducción Objective-C	21
2. Desarrollo de una aplicación sencilla en iOS.....	32
2.1. Xcode	32
2.2. Wireframes	40
2.3. Desarrollo con Xcode	41
2.3.1. Pantalla de login	42
2.3.2. Pantalla de email	51
2.3.3. Pantalla de Students	60
2.3.4. Pantalla de Mapa	65
2.3.5. Pantalla de Contacto	66
2.3.6. Retoques	71
3. Testeo de aplicaciones en iOS.....	75
4. Depuración de aplicaciones en iOS.....	80

1. Introducción al Xcode, Objective-C e iOS

Empezaremos introduciendo los conceptos básicos para poder desarrollar aplicaciones nativas para iOS. Más adelante, aplicaremos estos conceptos de forma práctica en el desarrollo de una pequeña aplicación, y por último, propondremos fuentes donde ampliar nuestros conocimientos.

1.1. Introducción iOS

iOS (<http://en.wikipedia.org/wiki/ios>), antes conocido como iPhone OS, es el sistema operativo que utilizan los dispositivos móviles de Apple como el iPod Touch, el iPhone y el iPad; la primera versión se presentó junto con el primer iPhone en junio del 2007; desde entonces, se han ido añadiendo nuevas funcionalidades a las nuevas versiones, pero prácticamente no ha variado conceptualmente ni en funcionamiento desde el primer día. Esto tiene un gran mérito, puesto que demuestra que fueron capaces de crear un gran producto que prácticamente no ha sufrido cambios pasados todos estos años.

Una de las características principales más reconocidas de iOS es la capacidad de interactuar a través de gestos multitáctiles, es decir, con más de un dedo a la vez; esto ha permitido crear muchos gestos nuevos que nos ayudan a hacer mucho más intuitiva y usable la experiencia en estos dispositivos móviles. Aparte de esta funcionalidad muy táctil, también se han introducido elementos como el acelerómetro, el giroscopio, la brújula magnética, el GPS, etc. Tener todas estas funciones dentro de un mismo dispositivo ha provocado una gran variedad de aplicaciones y usos impensables utilizando solamente un solo dispositivo.

En cuanto a los orígenes del iOS, se trata de una adaptación del sistema Mac OS X, pero a pequeña escala y adaptado a dispositivos móviles; tanto el uno como el otro siguen teniendo unas raíces muy parecidas a un sistema UNIX.

Las herramientas para programar iOS y Mac OS X son exactamente las mismas, y muchas librerías de programación son prácticamente idénticas. En iOS, por eso, suelen haber bastante más restricciones que con una aplicación de escritorio. Por ejemplo, las aplicaciones de iOS disponen de un sandbox, por lo que no pueden ver los ficheros otras aplicaciones; no existe por ahora un lugar común donde dejar archivos, cada aplicación dispone de una serie de directorios propios para trabajar. En Mac OS podemos instalar la aplicación que deseemos sin tener que pasar ninguna validación por parte de Apple; en cambio, con iOS todas las aplicaciones de la App Store han sido validadas previamente por Apple.

Apple ha ido mejorando tanto iOS como Mac OS en los últimos años; está integrando cada vez más los dos sistemas; iOS ha evolucionado gracias a todo el trabajo hecho ya con Mac OS, y Mac OS, en las últimas versiones, ha ganado muchas cosas de iOS, como las notas, recordatorios, notificaciones, iMessage, etc. El ritmo de actualizaciones en iOS es aproximadamente de una nueva versión cada año.

A pesar de que existen gran cantidad de versiones diferentes de iOS desde su lanzamiento, los usuarios de iOS suelen actualizar sus dispositivos en la última versión disponible, siempre y cuando tengan un terminal compatible. En parte es debido a las alertas que aparecen cuando hay una versión nueva disponible y a las nuevas funcionalidades que estas aportan a los usuarios. Esto hace que no haya una fragmentación importante de versiones iOS y facilita mucho el trabajo al desarrollador.

Hasta el día de hoy, el lanzamiento de cada versión de iOS ha ido acompañado del lanzamiento de un nuevo iPhone; cada nuevo iPhone ha venido con nuevas funcionalidades y mejoras respecto al anterior, a veces con cambios muy importantes, como la pantalla retina con más resolución o la nueva pantalla con una medida más alargada de 4" de los últimos dispositivos; estos son cambios que han modificado la manera de desarrollar las aplicaciones.

Tampoco tenemos que olvidarnos de la existencia de la familia del iPod Touch, la cual, a pesar de tener muchas funciones similares, no siempre ha tenido las mismas características que la misma generación de la del iPhone, puesto que es una gama de producto diferente y generalmente más asequible, por lo que hay generaciones sin cámara, sin GPS o con características no tan avanzadas que cabe tener en cuenta al desarrollar.

Y finalmente, nos encontramos con la familia de las tabletas con el iPad, el cual, con una medida tan diferente de pantalla, hace que sea necesario, la mayoría de veces, crear una versión de aplicación específica para dar soporte también a la tableta de Apple.

De cara al desarrollo, es importante, por lo tanto, tener claro con qué versiones de iOS queremos ser compatibles, con qué dispositivos, con qué medidas de pantalla, si todos comparten las mismas características, etc.

1.1.1. Distribución de aplicaciones iOS

Con la salida del iPhone, no existía todavía la App Store, solo se podían utilizar las aplicaciones que venían con el dispositivo; inicialmente, no se pensaba en la posibilidad de aplicaciones de terceros nativas, solo existía la posibilidad de realizar web apps, es decir, aplicaciones que funcionaban solo utilizando el

Enlace recomendado

Podéis encontrar una descripción de todas las características de los dispositivos actuales con iOS en:

http://en.wikipedia.org/wiki/list_of_ios_devices#iPhone

navegador. Fue un año después del 2008 cuando se presentó, finalmente, un paquete para desarrollar aplicaciones de terceros de forma nativa; era el primer iOS SDK y con él se hizo realidad también la tienda de aplicaciones App Store.

Desde su nacimiento ha significado un éxito rotundo; hoy en día la App Store de Apple es un nuevo mercado que mueve miles de millones de dólares en beneficios y que ha provocado también la creación de mercados de aplicaciones en otras plataformas.

Al contrario que en otras plataformas, Apple revisa cada aplicación antes de estar disponible en la App Store; la aplicación tiene que cumplir una serie de normas establecidas por Apple, y que, con cierta polémica, han ido variando durante el tiempo. Hay que destacar que se han flexibilizado bastante desde sus inicios, permitiendo por ejemplo muchas aplicaciones que hacen la competencia directa a aplicaciones de Apple como Google Chrome, Google Maps, etc.

En la App Store encontramos aplicaciones gratuitas y de pago; los beneficios de las ventas de aplicaciones de pago se comparten entre Apple y el desarrollador, Apple se lleva el 30% de la venta y el desarrollador el 70%.

Para poder distribuir nuestra aplicación o instalarla en nuestro dispositivo, necesitamos estar dados de alta como desarrolladores. Por eso, no es necesario para empezar a aprender a programar aplicaciones iOS, puesto que podemos descargar el programa Xcode desde la Mac App Store y probar nuestras aplicaciones en un simulador sin necesidad de darnos de alta.

Existen tres tipos de programas de Developer; el más común es el llamado Developer Program, que da ciertas ventajas, como disponer de las herramientas cuando todavía están en fase beta, acceso a los foros de desarrolladores, acceso a versiones de iOS beta, soporte técnico limitado y posibilidad de subir aplicaciones a la App Store, distribución limitada de aplicaciones a clientes usuarios limitada a 100 dispositivos (se denomina distribución Adhoc), etc. El coste de darse de alta en este programa es de 99 dólares anuales.

Después encontramos el Enterprise Developer Program, pensado para las empresas, con el cual podemos distribuir aplicaciones a nuestros empleados sin restricciones, pero no podremos publicar en la App Store; el coste es de 299 dólares anuales.

Y por último, encontramos el Developer University Program, con el cual podremos instalar las aplicaciones que desarrollamos en nuestro dispositivo; solo está disponible si demostramos que somos estudiantes; el precio es gratuito.

Provisioning

Enlace recomendado

Podéis consultar las normas en:
<https://developer.apple.com/appstore/guidelines.html>

Una de las otras características de la tienda de aplicaciones App Store es que estas están cifradas y firmadas por temas de seguridad; para publicar una aplicación en la App Store, tendremos que generar y utilizar una serie de archivos.

Toda esta gestión se realiza desde el Provisioning Portal, dentro del portal de desarrolladores de Apple; actualmente, mucha de esta gestión también se puede realizar directamente desde el Xcode de una forma mucho más sencilla.

La información que tendremos que gestionar será:

1) Certificados

Es el certificado que necesitamos para firmar y distribuir nuestras aplicaciones; existen de dos tipos: los de desarrollo para instalar la aplicación en nuestro dispositivo, y los de distribución para publicar la aplicación en la App Store.

2) App ID

Es un identificador de una sola aplicación o un grupo de aplicaciones; es una cadena de texto que nos indica las aplicaciones asociadas a un fichero mobileprovision. Podemos encontrar APP ID con wildcards (*) indicando un grupo de aplicaciones o explicit APP ID sin wildcards, indicando solo una aplicación. Un APP ID está formado por una serie de cadenas de caracteres separadas por puntos; la primera se llama Bundle Seed ID y es generada por Apple de forma aleatoria, sirve por si queremos compartir información privada entre diferentes aplicaciones. Después encontramos el dominio escrito de forma inversa y, por último, el nombre de la aplicación.

A continuación, veremos algunos ejemplos en el caso de la UOC (<http://www.uoc.edu>); la parte de XXXXXX corresponde al Bundle Seed ID.

XXXXXX.*

Xxxxxx.edu.uoc.*

Xxxxxx.edu.uoc.uocapp

Con el APP ID podríamos firmar cualquier aplicación; solo lo podremos utilizar para desarrollo, pero nos será útil tener un solo App ID para generar cualquier aplicación.

El segundo APP ID nos permite firmar todas las aplicaciones de la UOC, tanto de desarrollo como de distribución.

El tercer APP ID solo permite firmar la aplicación llamada edu.uoc.uocapp; se denomina explicit App ID.

Para utilizar algunos de los servicios de Apple, como las notificaciones PUSH, In App Purchase, iCloud, Passbook, etc., estaremos obligados a utilizar la tercera opción explicit App ID.

3) Devices

En este apartado, podemos dar de alta dispositivos útiles para el desarrollo o dispositivo donde queramos distribuir nuestra aplicación. Para identificar un dispositivo concreto se utiliza el UDID, que es una cadena de caracteres con la que Apple puede identificar cualquier dispositivo iOS.

4) Provisioning Profiles

Estos ficheros son los que nos permiten firmar las aplicaciones, los ficheros donde se agrupa toda la información anterior; nos indica el certificado que se utiliza, el App ID válido y los dispositivos donde se puede instalar la aplicación; también se llaman mobileprovision, puesto que es la extensión que utilizan.

Pueden ser de dos tipos: de desarrollo o de distribución. Estos últimos pueden ser de distribución en la App Store o de distribución AdHoc. La distribución AdHoc permite distribuir nuestra aplicación en una serie de dispositivos para probar nuestra aplicación.

Realizar todos los pasos y generar todos los archivos supone bastante tiempo y dolores de cabeza habitualmente, por lo que siempre es recomendable seguir al pie de la letra las instrucciones que encontraremos en el Provisioning Portal.

5) Jailbreak

Apple siempre ha ejercido un fuerte control sobre las aplicaciones de la App Store; el hecho de ser un ecosistema cerrado, en el que Apple decide las normas, le ha hecho recibir bastantes críticas.

Este cierre es el que impulsó el llamado jailbreak, que permite saltarse esta restricción y permitir instalar aplicaciones de terceros sin que tengan que estar en la App Store. El jailbreak dispone de un ecosistema propio de aplicaciones con diferentes stores, donde se pueden encontrar aplicaciones que nos modifican el iOS, aplicaciones rechazadas por Apple, etc. También existen stores con aplicaciones pirateadas, etc. El problema del jailbreak es que acostumbra a hacer más inestable nuestro teléfono, además de la posibilidad de que, debido a un error, no seamos capaces de recuperarlo. El porcentaje de uso de jailbreak ha ido disminuyendo de forma progresiva, puesto que las nuevas versiones de iOS ya incorporan muchas de las cosas que antes solo eran posibles a través de jailbreak.

1.2. Introducción iOS SDK

El programa Xcode es el programa oficial de desarrollo de Apple; es el IDE con que nos provee Apple para desarrollar aplicaciones.

El iOS SDK (http://en.wikipedia.org/wiki/ios_sdk) es el paquete de librerías que nos da Apple para poder desarrollar aplicaciones para iOS; en cada nueva versión de iOS hay una nueva versión del SDK, con nuevas funcionalidades y librerías disponibles. Es importante contar siempre con la última versión del SDK y estar al día de las novedades posibles, ya que Apple recomienda que se use siempre la última versión de iOS SDK disponible para compilar las aplicaciones del itunes appStore.

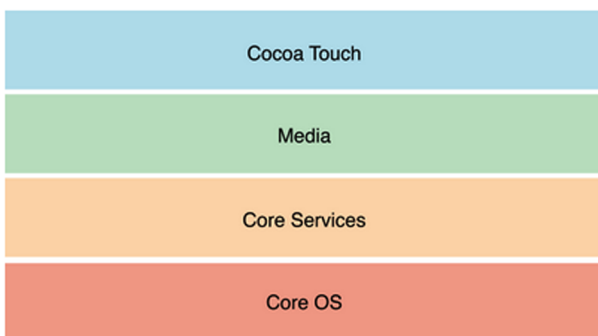
Actualmente, el Xcode y el iOS SDK están bastante interrelacionados, debido a que con la instalación del Xcode ya nos viene por defecto instalada la última versión del iOS SDK, por lo que es posible que nos pase desapercibido el iOS SDK. Con cada versión de SDK también se incluye el simulador de esa versión; podemos instalar otras versiones anteriores del SDK para disponer de los simuladores de versiones anteriores y así poder probar nuestra aplicación con el máximo de versiones posibles.

Como hemos dicho, el iOS SDK es la herramienta de desarrollo del iOS; por lo tanto, para utilizar el iOS SDK nos será de utilidad entender mejor la arquitectura interna del iOS.

A continuación, presentaremos la arquitectura interna, que a la vez nos servirá para introducir algunas de las tecnologías que tenemos disponibles en el iOS.

iOS se divide en cuatro capas o componentes, del componente de más bajo nivel (más cercano al hardware), Core OS, al de más alto nivel (más cercano al usuario), Cocoa Touch. Todos ellos disponen de los llamados frameworks, librerías compartidas que contienen clases, métodos y documentación que nos ayudan a acceder a las características de los dispositivos.

Estructura interna con las diferentes capas que forman el sistema iOS



1.2.1. Core OS

Core OS, por ejemplo, dispone de los frameworks de más bajo nivel, como el CoreBluetooth.framework o el ExternalAccessory.framework, para dar comunicación con hardware de terceros, el Security.framework con librerías de criptografía, el Accelerate.framework para funciones algebraicas, procesamiento de imágenes, etc.

El Core OS también contiene el System o kernel, con los drivers, interfaces UNIX, sistema de ficheros, gestión de memoria, etc.

El Core OS, al ser la capa de más bajo nivel, es usada habitualmente por las capas superiores o frameworks de forma interna.

1.2.2. Core Services

En la capa de Core Services encontramos algunos servicios como el iCloud, ARC, los Blocks, Grand Central Dispatch, In-App purchase, etc.

iCloud

Es un servicio introducido en el iOS 5 que permite leer y escribir datos y ficheros utilizando la nube de Apple. De este modo, el usuario puede disponer de sus archivos si pierde el dispositivo o utiliza más de uno.

ARC (Automated Reference Counting)

Es quizás una de las funcionalidades más importantes en cuanto a desarrollo para iOS; se introdujo en el iOS 5.

Para explicar qué significa, tenemos que introducir lo que se conoce como garbage collector; se trata de un proceso que se dedica a liberar los recursos que ya no utiliza el programa para que puedan ser utilizados de nuevo. Esto facilita enormemente la tarea al programador, ya que no necesita pedir y liberar memoria constantemente; desgraciadamente, el iOS no dispone de garbage collector, por lo que hasta hace poco el código de iOS estaba lleno de instrucciones como release o retain, para liberar o evitar liberar variables.

ARC es una mejora añadida al compilador; este ahora es suficientemente inteligente para añadir él mismo las instrucciones para gestionar la memoria, por lo que no necesitaremos llevar la cuenta de las variables creadas y liberadas.

Cada instrucción alloc tenía que tener un release; también existía una función dealloc donde liberar todas las variables; todo esto ya no es necesario.

Esto puede parecer una tontería, pero era seguramente la causa más importante de bugs en las versiones anteriores, además de ser bastante pesado tener que estar siempre pendiente de liberar toda la memoria de forma correcta.

Actualmente todavía hay bastantes códigos, librerías y clases que todavía no utilizan ARC; Xcode dispone de un conversor para actualizar código sin ARC en ARC; también da la posibilidad de, en un mismo proyecto, tener código con y sin ARC.

Blocks

Se introdujeron con el iOS 4; básicamente, permiten definir una función de modo anónimo con sus correspondientes instrucciones. Los escenarios más habituales donde los usaremos son los que se conocen como *callbacks*, funciones de cierre, etc. Para declararlos, se utiliza el carácter `^`.

Un ejemplo sería este:

```
^[NSLog(@"This is a block");]
```

GCD (Grand Central Dispatch)

Es una tecnología que se introdujo en el iOS 4 que nos facilita poder expresar al máximo los diferentes núcleos del procesador; en vez de tener que programar threads o diferentes hilos de ejecución de forma manual, podemos utilizar una serie de colas y prioridades indicando en el código qué partes queremos ejecutar y de qué manera, siendo el sistema el encargado de ejecutarlas de la manera más eficiente posible.

In-App Purchase

Se presentó con el iOS 3; permite a los desarrolladores vender productos o servicios dentro de la app, y se acostumbra a utilizar para vender nuevos niveles, suscripciones, etc. Al usuario se le carga el pago a su cuenta de iTunes, por lo que es una plataforma de pago muy sencilla de implementar; por eso Apple se lleva un porcentaje de cada compra.

Gracias al In-App Purchase, y debido a la barrera que supone a veces el pago de entrada de una aplicación, han aumentado significativamente las aplicaciones que utilizan lo que se denomina *free-to-play*, que permite descargar las aplicaciones de forma gratuita y solo cobrar ya desde dentro de la aplicación cuando el usuario se ha enganchado a ello.

En cuanto a los frameworks, la capa de Core Services incluye algunos bastante importantes.

CoreData.framework

Con el lanzamiento del iOS 3.0, Apple desarrolló su propio sistema de base de datos; hasta entonces se utilizaba SQLite; este nuevo sistema se conoce como Core Data, y está integrado también en el Xcode. Se trata de una base de datos de alto nivel, que nos permite poder trabajar con los diferentes objetos sin tenernos que preocupar de generar las sentencias SQL. Se encarga de la parte de Model de una aplicación Model-View-Controller (MVC).

CoreLocation.framework

Este framework es el encargado de darnos información sobre posicionamiento, ya sea utilizando GPS, señales de las antenas de teléfono o Wi-Fi. También permite obtener información de la brújula interna si está disponible en el modelo.

CoreMotion.framework

Con este framework, seremos capaces de obtener la información del acelerómetro interno y del giroscopio para saber, exactamente, en qué posición está el terminal o cómo se está moviendo en el espacio.

EventKit.framework

Introducido por primera vez con el iOS 4, con este framework seremos capaces de acceder a los diferentes events del calendario interno; podremos consultar events de la agenda o crear nuevos. Con el iOS 6 también se pueden añadir recordatorios a la nueva aplicación Reminders. Al consultar datos privados, como por ejemplo la agenda, fotos, contactos, etc., el sistema alerta siempre al usuario informándole y preguntando si le da permiso para realizar la acción.

Foundation.framework

Es uno de los frameworks más importantes que nos facilita el trabajo con un conjunto de tipos básicos con los que empezar a trabajar. Es un conjunto de clases escritas Objective-C con las cuales nos será más fácil trabajar.

Como ejemplo encontramos las clases NSString, NSNumber, NSDate, NSArray, NSDictionary, etc.

Son clases con muchos métodos disponibles y que nos ahorrarán mucho trabajo en vez de utilizar variables de C como char, int, int[10], etc.

Algunos de los tipos de datos más habituales son NSString, NSNumber, NSArray, NSDictionary.

Los comentaremos en profundidad más adelante, cuando tratemos el lenguaje Objective-C.

También encontramos, dentro del framework Foundation, otros objetos relacionados en los hilos de ejecución, localización, comunicación, manipulación de URL, gestión de preferencias, etc. Podríamos definir este framework como la caja de herramientas que utilizaremos habitualmente.

NewsstandKit.framework

Este framework presentado junto al iOS 5 nos permite utilizar el Newsstand, que viene a ser un quiosco virtual donde poder leer y comprar revistas y prensa; si nuestra aplicación está orientada al mundo de la lectura, tendremos que dar un vistazo a este framework.

PassKit.framework

Disponible a partir del iOS 6, nos permite interactuar con la aplicación Passbook, que permite la descarga, creación y distribución de tickets, entradas, descuentos, billetes de avión, etc., utilizando solo el terminal móvil. Cada vez es soportado por más empresas al ahorrar la impresión y distribución de tickets, y hace que el usuario pueda usarlo solo con su terminal.

QuickLook.framework

Este framework se puede utilizar a partir del iOS 4; dispone de las herramientas para visualizar gran cantidad de formatos de ficheros diferentes; la idea es permitir a las aplicaciones previsualizar, de una forma fácil, diferentes formatos de archivo sin que la aplicación tenga que incluir un visualizador propio.

Social.framework

Con este framework presentado con el iOS 6 se permite una integración más fácil con algunas redes sociales como Twitter o Facebook. Hasta entonces, para hacerlo se tenían que usar las API correspondientes de las redes sociales. Ahora, dependiendo de lo que se tenga que implementar, se puede optar por utilizar este framework.

StoreKit.framework

Es el framework encargado de dar soporte al servicio In-App Purchase que hemos visto anteriormente. Nos ahorra tener que gestionar nosotros mismos una pasarela de pagos, haciéndolo todo mucho más sencillo tanto para el desarrollador como para el usuario; por el contrario, Apple se lleva un porcentaje de cada compra.

SystemConfiguration.framework

Es el framework que nos da información sobre la conectividad de nuestro dispositivo, si está conectado a internet, si lo hace con Wi-Fi o utilizando la red móvil, etc.

1.2.3. Media

Por encima de Core Services nos encontramos la capa de Media, que incluye las tecnologías de soporte tanto de tratamiento gráfico como librerías de audio y vídeo.

Dispone de frameworks muy completos que nos permiten gestionar experiencias multimedia con el nivel de complejidad que deseamos, ya que hay frameworks de más alto nivel y otros de más bajo nivel.

Analizando la cantidad de frameworks incluidos en la capa Media, queda clara la importancia que Apple da a los contenidos multimedia.

Una de las tecnologías destacables de esta capa es el AirPlay, que permite que los usuarios con el iOS5 puedan reproducir tanto el audio como el vídeo que están visualizando directamente en una pantalla externa; esto mejora mucho la experiencia del usuario, por ejemplo, al jugar un juego, ver un vídeo de YouTube o reproduciendo música. Para hacerlo, es necesario un AppleTV o un dispositivo compatible con AirPlay.

Algunos de los frameworks más destacables son:

AssetsLibrary.framework

Con este framework podemos acceder a las fotos y vídeos del usuario, del mismo modo que la aplicación de fotos del terminal. Por supuesto, es necesario el permiso explícito del usuario.

AVFoundation.framework

Disponibile desde el iOS 2.2, ha recibido muchas mejoras a lo largo del tiempo; actualmente, se utiliza para la captura como reproducción tanto de audio como de vídeo. También es el framework encargado de da soporte a AirPlay.

Core Audio

Core Audio se entiende como un conjunto de diferentes frameworks: CoreAudio.framework, encargado de definir los diferentes tipos o formatos de audio utilizados; AudioToolbox.framework, encargado de la reproducción de sonidos del sistema, de hacer vibrar el dispositivo; AudioUnit.framework, para

el procesamiento de audio; CoreMIDI.framework, para hacer aplicaciones con soporte de MIDI; y Mediatoolbox.framework, para dar soporte a interfaces de audio.

CoreGraphics.framework

Este framework contiene la librería Quartz, una librería de Apple de dibujo vectorial, tratamiento de imágenes, visualización de PDF, etc. Es una librería con mucha evolución y, por lo tanto, bastante potente.

CoreImage.framework

Introducido con el iOS 5, permite la manipulación tanto de imágenes como de vídeos, y añadir funcionalidades, como el retoque de imágenes o detección de caras.

CoreText.framework

Fue introducido con el iOS 3.2; permite una mejor gestión para dibujar texto y gestionar fuentes. Está pensado para aplicaciones que hagan un uso intensivo de textos y tipografías.

ImageIO.framework

Presentado con el iOS 4, es un framework que permite la importación y exportación de datos relacionados con las imágenes, como por ejemplo metadata EXIF o IPTC, incluida dentro de muchas imágenes.

GLKit.framework

Este framework, presentado con el iOS 5, aporta una serie de utilidades para simplificar la implementación de aplicaciones OpenGL.

MediaPlayer.framework

Es un framework de alto nivel pensado para la reproducción de audio y vídeo en nuestro dispositivo. Al ser de alto nivel, las funcionalidades que aporta son limitadas, pero suficientes para la mayoría de aplicaciones que necesitan reproducir archivos multimedia.

OpenAL.framework

Framework que da soporte a la especificación OpenAL, muy utilizado en la reproducción de sonido para juegos.

OpenGLES.framework

Este framework da soporte a OpenGL; actualmente soporta la versión 1.1 y 2.0 en la mayoría de dispositivos actuales. OpenGL es un lenguaje estándar para la creación de entornos 2D y 3D, utilizado sobre todo para desarrollar juegos.

QuartzCore.framework

Incluye lo que se conoce como Core Animation, que es una serie de clases que nos permitirán realizar animaciones y aplicar efectos de una manera bastante sencilla.

1.2.4. Cocoa Touch

Y finalmente, encontramos la capa de más alto nivel de todas, la capa Cocoa Touch, que agrupa las tecnologías y frameworks que forman la interfaz básica de una aplicación.

En cuanto a tecnologías, encontramos las siguientes:

1) AutoLayout

Introducido recientemente con el iOS 6, permite establecer reglas de posicionamiento en los diferentes elementos de la interfaz de una aplicación; con la aparición de diferentes medidas de pantalla en iOS, cada vez era más necesario mejorar este punto. Anteriormente, cada objeto se posicionaba por él mismo con unas reglas simples; AutoLayout permite definir reglas respecto a otros objetos.

2) Storyboards

Se introdujeron con el iOS 5; es una de las mejoras más destacables de las últimas versiones; se trata de una mejora en la creación de interfaces gráficas. Antes de los storyboards, para cada pantalla se creaba un fichero donde estaban todos los elementos de esa pantalla; la navegación entre pantallas se tenía que programar mediante código. Los storyboards permiten definir en un solo fichero todas las pantallas, así como la navegación entre ellas; esto nos ahorra escribir mucho código, simplificar enormemente el desarrollo y poder visualizar en una sola pantalla toda la aplicación.

3) Document Support

Introducido con el iOS 5, en parte debido a la inclusión del iCloud, permite mejorar la gestión de archivos por parte de las aplicaciones.

4) Multitarea

Con el iOS 4 se añade la posibilidad de multitarea; en realidad no es una multitarea real como la de un ordenador de escritorio, sino que la aplicación se guarda en memoria pero no puede ejecutar instrucciones; esto permite volver a abrirla tal como la habíamos dejado anteriormente.

También se establece una serie de acciones que las aplicaciones pueden pedir al sistema que continúe ejecutando, como la reproducción de música, el GPS, etc. Esto permite la sensación de multitarea en un dispositivo móvil que no tiene las características técnicas para realizarla.

5) Air Print

Esta tecnología permite la impresión en impresoras conectadas en la misma red Wi-Fi de forma wireless y sin necesidad de disponer de ningún driver. La impresora, por eso, tiene que ser compatible con Air Print.

6) Notificaciones PUSH

Con el iOS 3.0 también aparece la funcionalidad de las notificaciones PUSH, que nos permiten recibir avisos cuando no tenemos la aplicación abierta, tanto de texto como alertas sonoras, o simplemente mostrar un número rojo en el icono de la aplicación.

Estas pueden ser generadas localmente; por ejemplo, una alarma programada o remotamente, como por ejemplo un mensaje de WhatsApp. Para enviar notificaciones PUSH, es necesario disponer de un servidor configurado para permitir el envío de PUSH; muchas veces es más sencillo utilizar servicios gratuitos de envío como urbanairship.com, boxcar.io, etc.

7) Gestos multitáctiles

Se añadieron con el iOS 3.2; son un conjunto de gestos que podemos enlazar a los objetos de nuestra interfaz para poderlos identificar y hacer la acción pertinente. Se introdujeron en parte debido a la aparición del iPad, pues una pantalla más grande permitía realizar gestos con más dedos.

8) Peer-to peer (P2P)

Introducido a partir del iOS 3.0, permite la funcionalidad de interactuar con otros dispositivos cercanos a través de Bluetooth; está pensado para poder jugar a juegos varias personas a la vez, cada cual con su terminal.

9) External Display Suport

Disponibile para el iOS 3.2 y superior, permite conectar el dispositivo a través de un cable a una pantalla externa; con la nueva tecnología AirPlay, que lo hace Wireless, ya no se utiliza mucho.

En cuanto a los frameworks de la capa Cocoa Touch, disponemos de los siguientes:

1) **AddressBookUI.framework**

Se trata del framework que nos permite gestionar los contactos de la agenda, ya sea consultarlos o crear nuevos; es necesario el consentimiento del usuario para algunas acciones.

2) **GameKit.framework**

Incluido desde el iOS 3, es el encargado de dar soporte a la tecnología Peer-to-peer; utiliza Bonjour, que es el protocolo de detección de dispositivos de Apple. A partir del iOS 4 también incluye el Game Center, que permite a los usuarios de los juegos compartir sus logros en los diferentes juegos, así como visualizar los progresos de todos los juegos de forma centralizada, etc. Es una pequeña red social orientada a los juegos de iOS.

3) **iAd.framework**

Es el framework que da soporte a la plataforma iAd de Apple. iAD es una plataforma de anuncios para dispositivos iOS creada por Apple; permite visualizar anuncios interactivos dentro de las aplicaciones; el desarrollador cobra por cada visualización.

4) **MapKit.framework**

Introducido con la versión iOS 3, permite la visualización de mapas, indicar al usuario elementos, dar instrucciones para llegar a algún lugar específico, etc.

Inicialmente utilizaba los mapas de Google, ahora utiliza los propios de Apple.

5) **MessageUI.framework**

Es el framework que permite la gestión de correo electrónico y mensajería instantánea o SMS. Se acostumbra a utilizar sobre todo a la hora de redactar un correo electrónico.

6) **UIKit.framework**

El framework más importante de Cocoa Touch; contiene todos los elementos gráficos básicos de una interfaz de iOS; contiene clases como UIView, UIButton, UILabel, etc. Estos son los objetos habituales con que construiremos nuestra interfaz.

También incluye funcionalidades de multitarea, gestión de impresión, copy-paste, accesibilidad y otras muchas funcionalidades.

1.3. Introducción Xcode

Xcode (<http://en.wikipedia.org/wiki/xcode>) es la herramienta oficial para programar aplicaciones tanto para iOS como para Mac OS. Ha ido evolucionando con el tiempo.

Hay otras herramientas para desarrollar aplicaciones de Apple, pero solo una es la IDE oficial. El resto de herramientas acaban utilizando, de una u otra manera, el Xcode para poder generar los binarios finales; generalmente acaban haciendo uso de la línea de pedidos del Xcode para firmar y generar los binarios finales.

Xcode ha ido integrando diferentes herramientas; hasta hace relativamente poco eran necesarias dos aplicaciones para programar aplicaciones iOS, el Xcode y otro programa llamado Interface Builder; el primero era para programar código, mientras que con el Interface Builder se creaban las interfaces de forma gráfica o GUI. A partir de la versión 4, la Interface Builder se integró también dentro del Xcode.

Con la última versión, la 5, se han añadido cambios significativos a la interfaz del Xcode 4, un poco siguiendo la idea del iOS7 de simplificar la interfaz al máximo.

Algunas de las mejores son:

Se ha facilitado la configuración de servicios como iCloud, Passbook, GameCenter. Ahora el Xcode permite crear bots; estos permiten programar compilaciones y tener un mayor control para encontrar bugs en las diferentes versiones (es necesario para eso disponer de un servidor). También ha mejorado la herramienta de AutoLayout, que nos permite indicar cómo se comportan los elementos con diferentes tamaños de pantalla. Ha mejorado el soporte de control de versiones interno y de repositorios externos. Y hay otros muchos cambios que mejoran la versión anterior.

Hay que destacar que la programación con Xcode sigue el patrón de MVC (model-view-controller); esta estructura es común a muchos lenguajes de programación; se trata de un modelo que separa los elementos en tres tipos. El primero es el Model; en este apartado está la información de la app, las funciones y las diferentes clases. El segundo es el View, donde encontramos la interfaz de la aplicación, pantallas, vistas, botones, controles, etc. Y por último, encontramos el Controller, que contiene la lógica de la app y, como tiene que actuar, modifica el View con las herramientas del Model.

Por lo que respecta al compilador utilizado, este ha ido evolucionando también; actualmente se utiliza uno llamado LLVM; hasta hace no mucho se utilizaba el LLVM-GCC, que era un híbrido de los dos, y antes de este el GCC, seguramente el compilador más conocido. LLVM nos aporta ciertas mejoras ya que está más optimizado y nos ayudará a encontrar errores en nuestro código más fácilmente.

En la parte práctica veremos, de forma más detallada, el funcionamiento del Xcode.

1.4. Introducción Objective-C

Objective-C (<http://en.wikipedia.org/wiki/objective-c>) es históricamente el lenguaje asociado a la programación de programas para los sistemas operativos de Apple, al menos desde la aparición de Mac OS X.

Objective-C es un lenguaje evolucionado del C, del mismo modo que C++; los dos son lenguajes orientados a objetos (OOP); el principal objetivo de estos es la reutilización de código, y permitir crear programas más complejos con estructuras más sencillas.

Objective-C, además de la sintaxis, tiene algunas diferencias estructurales con C++, puesto que han evolucionado de manera diferente; Objective-C ha recibido mucha influencia de otro lenguaje llamado Smalltalk.

Objective-C es totalmente compatible con el código C; Objective-C supone solo una capa por encima de C, por lo que podremos utilizar librerías, tipos de variables y funciones de C sin problemas.

Para aprovechar las ventajas de un lenguaje orientado a objetos, no se utiliza mucho código en C, a pesar de que nos puede ser útil esta característica para integrar código C cuando sea necesario.

Una de las características de Objective-C es el envío de mensajes; cuando se llama a un método en otros lenguajes, el nombre del método establece un punto fijo a ejecutar; en Objective-C, el método a llamar se resuelve en tiempo

Enlace recomendado

Más información en:
<http://en.wikipedia.org/wiki/model-view-controller>

Enlace recomendado

Podéis encontrar más información en:
http://en.wikipedia.org/wiki/object-oriented_programming

de ejecución, por lo que no está asegurada la respuesta. Podemos expresar, por lo tanto, que cuando se llama a un método se manda un mensaje, pues no tenemos asegurada la respuesta.

Nomenclatura

Existe una nomenclatura de nombres en Objective-C que ayuda a identificar variables, métodos, clases, etc.

Las clases empiezan siempre con una letra mayúscula; si está formada por más de una palabra, las siguientes palabras también empiezan por una letra mayúscula. Los nombres de clases suelen ir en singular.

Ejemplos de nombres de clases

AddressBook, MultipleContact, Contact

En cuanto a las variables, estas empiezan siempre con una letra minúscula o guion bajo; si está formada por más de una palabra, se suele poner con mayúscula la segunda palabra o se mete un guion bajo entre ellas.

Ejemplos de nombres de variables

counter, currentValue, contact_name

Y por último, encontramos los métodos que comienzan con una palabra en minúscula, que tiene que ser una acción; si tienen más palabras, se ponen en mayúscula a continuación.

Ejemplos de métodos

print, addContact, removeList

Declaración de variables

Antes de entrar de lleno en la sintaxis del lenguaje Objective-C, es importante notar que Objective-C utiliza punteros cuando trabaja con objetos; se ha de tener en cuenta cuando se crea un objeto o cuando se pasa uno por parámetro; tendremos que indicar que se trata de un objeto con el carácter *. No explicaremos el funcionamiento de los punteros, simplemente tenemos que acordarnos de añadir el carácter * cuando declaremos objetos o recibamos un objeto en una función.

Antes que nada, observemos cómo declaramos un objeto: notamos el * debido a que se trata de un objeto.

```
Class *object;
```

Por el contrario, para crear una variable básica de C no será necesario el *.

```
int a;
```

Para llamar a un método o mandar un mensaje a un objeto, la sintaxis es la siguiente:

```
[object method];
```

Lo que se hace es indicar primero el receptor y después indicar el método que se quiere llamar, todo entre corchetes.

El receptor puede ser tanto un objeto como una clase, puesto que también existen los métodos de clases, llamados métodos static en otros lenguajes; en este caso, la llamada sería:

```
[Class method];
```

Volviendo a la creación de objetos con Objective-C, es importante entender que, para trabajar con objetos, no tenemos suficiente con declararlos sino que tendremos que pedir también memoria e inicializarlos. Esto serían tres instrucciones:

La primera, la declaración de un objeto:

```
Class *object;
```

La segunda sería pedir memoria con la función alloc; este es el método que se encarga de pedir la memoria necesaria para poder guardar el objeto y nos devuelve un objeto de la clase.

```
object = [Class alloc];
```

Y por último, inicializaríamos el objeto con la función init; este método se encarga de ejecutar las instrucciones necesarias al crear nuestro objeto, para tenerlo listo para utilizar.

```
object = [object init];
```

Podemos realizar estas tres acciones con una misma línea:

```
Class *object = [[Class alloc] init];
```

De este modo, se incluye el resultado de una llamada dentro de otra; el resultado es idéntico, pero esta última variante es más sencilla y corta.

Los métodos `alloc` e `init` mencionados son comunes a todos los objetos, debido a que todas las clases heredan de una clase raíz llamada `NSObject`; esta dispone de una serie de métodos comunes a todos los objetos que nos serán de gran utilidad habitualmente.

Cuando declaramos un objeto, este tiene asignado el valor `nil`, que viene a ser el `null` utilizado en otros lenguajes de programación. Una vez hayamos pedido la memoria e inicializado el objeto, este dejará de valer `nil`, y estará listo para ser utilizado.

Pase de parámetros

A continuación, veremos cómo funciona el paso de parámetros y el retorno de valores; es importante decir que Objective-C no permite la sobrecarga de funciones o `function overloading`. Esto quiere decir que no puede haber dos métodos con el mismo nombre. A continuación, veremos cómo Objective-C soluciona esta carencia.

Imaginémonos, por ejemplo, un método de una clase llamada `API` que nos comprueba que hemos hecho el login correctamente.

Sin ningún parámetro, la declaración sería:

```
+ (BOOL) doLogin;
```

El símbolo `+` del principio nos indica que se trata de un método de clase (llamado método `static` en otros lenguajes); el símbolo `-` por otro lado nos indica que se trata de un método de un objeto. A continuación, entre paréntesis, encontramos el tipo devuelto, aquí, un booleano `BOOL`, en este caso para indicarnos si se ha hecho el login correctamente. Si el método no devuelve nada se utiliza (`void`).

Para llamar a este método haríamos la llamada:

```
BOOL result=[API doLogin];
```

En este caso, recibiremos un booleano con el resultado de llamar al método de clase `doLogin`.

Para poder pasar parámetros a un método se añade el símbolo `“:”`.

En este caso, pasaremos una cadena de caracteres, de tipos `NSString`.

```
+ (BOOL) doLoginWithUserName: (NSString*) user;
```


En la declaración después de los dos puntos, se añade entre paréntesis el tipo; si es un objeto se añade el símbolo de puntero *. Y después se añade el nombre del parámetro.

Para llamar a este método, el código es el siguiente.

```
BOOL result=[API doLoginWithUserName:username];
```

Como vemos, solo tenemos que añadir dos puntos y la variable que queremos enviar. Es importante advertir que el nombre del método ha cambiado; esto es habitual en Objective-C, ya que es más comprensible de este modo.

```
[API doLoginWithUserName:username]
```

que no

```
[API doLogin:username]
```

A continuación, lo complicaremos algo más, ahora pasando dos parámetros.

La declaración es la siguiente:

```
+(BOOL)doLoginWithUserName:(NSString *)user andPassword:(NSString *)password;
```

Cuando se pasa más de un parámetro se suele añadir un texto descriptivo para el segundo parámetro que acompañe la acción del método, en este caso andPassword. No es obligatorio; también lo podríamos definir así:

```
+(BOOL)doLoginWithUserName:(NSString *)user :(NSString *)password;
```

En estos dos casos, los nombres de variables que recibe la función son user y password.

Y la llamada quedará:

```
BOOL result=[API doLoginWithUserName:username andPassword:password];
```

o

```
BOOL result=[API doLoginWithUserName:username :password];
```

Queda mucho más comprensible la primera opción, es por eso que se añade este nombre.

Esta manera de trabajar con los parámetros hace que no se eche de menos la posibilidad de sobrecargar las funciones.

Por lo tanto, podemos tener definidos estos métodos a la vez sin problemas:

```
+ (BOOL) doLogin;  
+ (BOOL) doLoginWithUserName: (NSString*) user;  
+ (BOOL) doLoginWithPassword: (NSString*) password;  
+ (BOOL) doLoginWithUserName: (NSString *) user andPassword: (NSString *) password;
```

Puede parecer complicado de entrada, pero más adelante nos ayudará a leer mejor el código.

Tipo de variables y clases

Como hemos comentado antes, programando con Objective-C podemos utilizar para trabajar tanto variables de C, como int o float, como clases de más alto nivel, como NSString o NSNumber, que veremos más adelante.

Pero también es importante decir que existen unos equivalentes a los tipos int, uint, float, etc., con los que es más recomendable trabajar; se llaman NSInteger, NSUInteger, NSDecimal, etc. Estos nuevos tipos se encargan, automáticamente, de escoger la medida ideal cuando se trabaja con una máquina de 32 bits o 64 bits, por lo que siempre son más recomendables de utilizar.

A continuación, presentaremos algunas de las clases del framework Foundation que utilizaremos más habitualmente programando con el lenguaje Objective-C. Existen muchas, cada una por un objetivo determinado; a continuación veremos las más destacadas.

1) NSString

Se trata de una clase encargada del tratamiento de strings inmutables¹; dispone de muchos métodos para el tratamiento de strings o cadenas de caracteres que nos serán de gran utilidad, métodos de busca, comparación, etc.

⁽¹⁾ **Inmutable:** no se puede modificar el contenido una vez creado.

Para cadenas de caracteres mutables disponemos de la clase que hereda de NSString, que se llama NSMutableString, con la cual dispondremos además de métodos para modificar nuestras cadenas de caracteres.

Para definir cadenas de caracteres inmutables, con Objective-C se utiliza el símbolo @ delante de todo de la cadena.

En este ejemplo guardamos el texto “cadenadetext” dentro de la variable cadena.

```
NSString *cadena = @"cadenadetext";
```

A continuación llamamos al método `length` de `NSString` para obtener la longitud de este.

```
NSUInteger llargada = [cadena length];
```

2) NSNumber

Es la clase con la que podemos almacenar y trabajar con números, ya sean `int`, `float`, `NSInteger`, etc.

Para crear un objeto `NSNumber` donde guardar el valor 1, haríamos lo siguiente:

```
NSNumber *number = [NSNumber numberWithInt:1];
```

Puede parecer incómodo realizar operaciones con esta clase, pero son clases pensadas para homogeneizar el tratamiento de los diferentes tipos de variables números y no tanto para hacer operaciones; son clases contenedoras de números.

Para sumar dos valores se suelen sumar las clases de `C` y después se guarda en un `NSNumber`.

```
NSInteger a=2,b=3;  
NSNumber *resultat = [NSNumber numberWithInt:a+b];
```

Para crear un `NSNumber` nuevo y darle el valor 1 tendríamos que utilizar las funciones `alloc` e `init`:

```
NSNumber *numero = [[NSNumber alloc] initWithInteger:1];
```

De todos modos, hay excepciones donde esto no es necesario.

Utilizar el `alloc` implicaba, antes de la tecnología ARC, que nosotros éramos los responsables de esa variable y, por lo tanto, tendríamos que tener una instrucción `release` para liberar el espacio; para facilitar el trabajo había métodos de conveniencia, donde era la propia clase la que se encargaba de gestionar la memoria de la variable. Un ejemplo es la instrucción que hemos visto antes:

```
NSNumber *numero = [NSNumber numberWithInt:1];
```

Con ARC activado a efectos prácticos, las dos sentencias son completamente idénticas. Sin ARC, la cosa cambiaría, pues tendríamos que liberar la memoria pedida en el primer caso llamando al método `release`. Los métodos de conveniencia, por lo tanto, eran muy útiles para no tenernos que preocupar de liberar después las variables.

Y últimamente, con la última versión del compilador utilizada por Xcode, se pueden usar los llamados Objective-C literales; son declaraciones mucho más cortas que las que hemos comentado anteriormente; las identificaremos porque el primer carácter es también una @ igual que cuando definimos cadenas de caracteres.

Las instrucciones siguientes son todas equivalentes.

```
NSNumber *numero =@3U;
NSNumber *numero = [NSNumber numberWithInt:3];

NSNumber *numero2 =@YES;
NSNumber *numero2 = [NSNumber numberWithBool:YES];
```

3) NSArray

Es la clase encargada de tratar con arrays o conjuntos de objetos; nos permite tratar los arrays como pilas y colas. Dentro de un NSArray solo podemos guardar objetos de Objective-C como NSString, NSNumber, NSDate, etc.; no puede almacenar variables de C como int o float.

Para indicar el final de un NSArray, lo indicaremos con el valor nil; si necesitaríamos indicar una casilla vacía de un array, podemos guardar [NSNull null], puesto que si no, estaríamos indicando el final del array en esa casilla.

NSArray, del mismo modo que NSString, es inmutable, por lo que no lo podremos modificar; si queremos modificar sus valores, disponemos de la clase NSMutableArray, que dispone de métodos especiales para poder modificar los diferentes objetos dentro del array.

Algunos ejemplos de creación de NSArray serían:

```
NSArray *array = [NSArray arrayWithObjects:a,b,[NSNull null],c,nil];

NSArray *array = @[a,b,[NSNull null],c];

NSMutableArray *array = [[NSMutableArray alloc] init];
```

4) NSDictionary

Es una clase contenedora de otras como NSArray; lo que pasa es que en un NSDictionary los objetos se guardan con una clave y no tienen por qué estar ordenados. Al igual que las otras clases, también disponemos de una clase mutable, en este caso NSMutableDictionary.

Un ejemplo sería:

```
NSMutableDictionary *dictionary = [NSMutableDictionary dictionaryWithObjectsAndKeys:a, @"clau-a",
    b, @"claub", nil];
```

O

```
NSMutableDictionary *dictionary = @{@"clau-a" : a,
    @"clau-b" : b };
```

Esto nos guardaría las variables a y b.

Para consultar a solo tendríamos que hacer:

```
NSNumber *a = [dictionary valueForKey:@"clau-a"];
```

Hay muchas más clases disponibles en Foundation; las podemos consultar todas desde la documentación incluida en el iOS SDK.

Custom Classes

Para desarrollar una aplicación, necesitaremos definir nuestras propias clases; las clases del lenguaje Objective-C necesitan dos ficheros, un primer fichero con extensión .h con la declaración, y un segundo fichero con extensión .m con la implementación.

En el fichero .h es donde encontraremos la clase de quien hereda nuestra clase, las variables que utiliza la clase y los métodos que implementa. Todo va dentro del elemento @interface e indicamos el final con @end.

Un ejemplo de fichero de archivo .h sería:

```
@interface MyClass : NSObject

-(void)methodFunction;
+(void)classFunction;

@end
```

En este fragmento MyClass es el nombre de nuestra clase, NSObject es la clase de quien heredamos, en este caso de la clase raíz NSObject. A continuación nos encontramos con la declaración de los métodos de nuestra clase; para ser accesible desde otras clases tendremos que declararlo aquí, si no, el método no será accesible desde el exterior.

Advertimos que un método tiene el signo - y el otro +, que indica, como hemos comentado anteriormente, si se trata de un método de una clase o es un método de los objetos de la clase.

En el caso del método de clase, llamaríamos al método de la siguiente manera:

```
[MyClass classFunction];
```

En cambio, para llamar al método de un objeto de la clase, inicializaríamos primero el objeto y después ya podríamos llamar al método.

```
MyClass *myObject = [[MyClass alloc] init];  
[myObject methodFunction];
```

A continuación, veremos cómo añadir variables dentro de nuestra clase.

```
@interface MyClass : NSObject <MyDelegate>{  
  
    NSString *myString;  
    NSInteger myNumber;  
}  
  
@property (readwrite, nonatomic) int myNumber;  
  
-(void)methodFunction;  
+(void)classFunction;  
@end
```

Las variables que añadimos dentro de las claves del elemento @interface se llaman instance variables; son todas protected, que quiere decir que serán accesibles desde la misma clase y también subclasses. En C++ por defecto serían privadas, con lo que solo serían accesibles desde la misma clase pero no por las subclasses.

Para hacerlas accesibles de forma externa a cualquier clase tenemos que crear una @property; estas tienen diferentes atributos, que veremos más adelante. Cuando definimos una @property automáticamente tendremos dos métodos para acceder a la variable: uno para consultarla (getProperty) y otro para modificarla (setProperty). Antes del Objective 2.0, se tenía que añadir también la instrucción @synthesize dentro del archivo .m, el cual creaba estos dos métodos para cada propiedad; ahora ya no es necesario, pues todas las @property tendrán estos métodos setters y getters de forma automática.

En el ejemplo anterior, por lo tanto, tendremos una variable llamada myString que será accesible desde la propia clase y subclasses. Y una segunda variable, llamada myNumber, que será accesible desde cualquier otra clase directamente o utilizando los métodos setMyNumber y getMyNumber, creados de forma automática, tal como hemos dicho, aunque no aparezcan en nuestra declaración.

También es necesario comentar que, cuando definimos un `@property`, entre paréntesis tenemos que indicar sus propiedades. Existen muchas, y no entraremos a detallarlas todas.

Lo más común es utilizar `(readwrite, nonatomic)` para variables como `int`, `float`, etc., que no son objetos, y para objetos las propiedades `(strong, nonatomic)`.

Cabe hablar también de los delegates, que nos indican qué funcionalidades implementa nuestra clase. Los trataremos más adelante, pero por ahora veamos que, para declararlos, los ubicamos junto a la clase de la que heredamos entre los símbolos `<>`.

```
@interface MyClass : NSObject <MyDelegate1, MyDelegate2>{

    NSString *myString;
    int myNumber;
}

@property (readwrite, nonatomic) int myNumber;

- (void)methodFunction;
+ (void)classFunction;

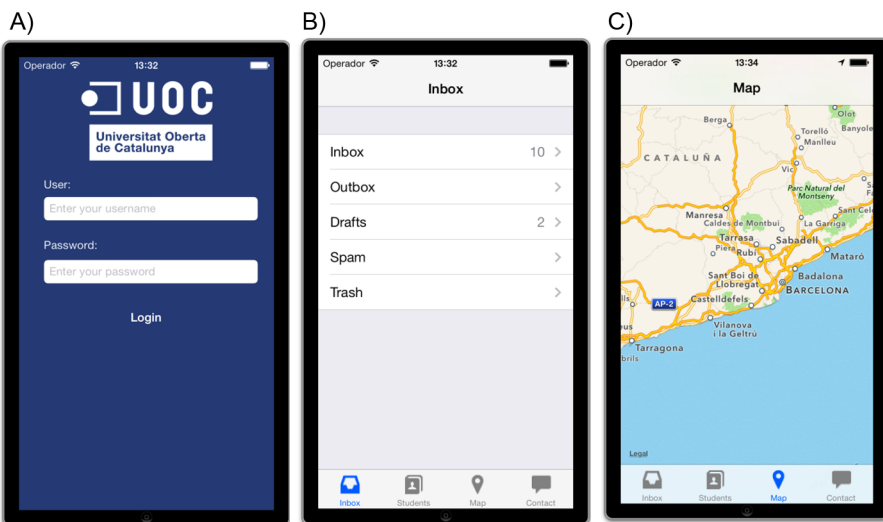
@end
```

2. Desarrollo de una aplicación sencilla en iOS

Empezaremos creando un nuevo proyecto para introducir el Xcode, y después utilizaremos este proyecto nuevo para desarrollar nuestra primera aplicación iOS.

En este apartado, veremos cómo desarrollar una aplicación sencilla utilizando los conocimientos aprendidos hasta ahora. La aplicación será una maqueta de una nueva posible aplicación iPhone de la UOC orientada a los estudiantes. Dispondría de una primera pantalla de acceso, que una vez entrada la información correcta nos permitiría consultar el correo del campus, disponer de un listado con los alumnos que cursan los estudios con nosotros para poder hacer chat, un mapa con la localización de las sedes o puntos de interés, y una última pantalla donde contactar con la UOC. Recordemos que se trata de una aplicación sencilla, por lo que solo simulará un posible acceso al campus y un listado de estudiantes también como ejemplo. Para que funcionara, se tendría que implementar toda la comunicación con el servidor de la UOC, pero en esto no entraremos.

A continuación, veamos unas pantallas de lo que sería la aplicación ya acabada.



A) Pantalla de acceso a la aplicación con los datos de acceso del estudiante. B) Pantalla que simula el buzón de un estudiante. C) Pantalla del mapa donde ubicarnos y donde mostraríamos los puntos importantes para los estudiantes

2.1. Xcode

Si no tenemos todavía el Xcode instalado, lo podremos instalar accediendo a la aplicación App Store de nuestro ordenador; buscaremos el Xcode y lo instalaremos.

Podemos descargarlo desde la URL:

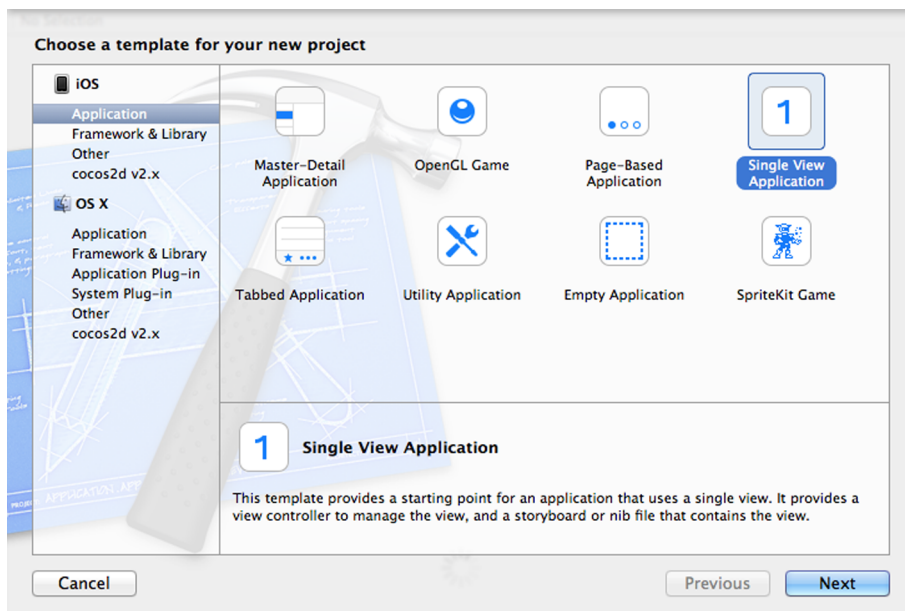
<https://itunes.apple.com/es/app/xcode/id497799835?mt=12>

Una vez finalizada la instalación, lo abriremos y acabaremos de instalar los componentes que faltan.

Si accedemos a las preferencias del Xcode en el apartado Downloads, veremos los componentes instalados; es recomendable tener instaladas todas las versiones posibles del simulador iOS, de este modo podremos probar de forma fácil nuestra aplicación con diferentes versiones del iOS.

Empezaremos creando un nuevo proyecto, seleccionaremos la opción New > Project para crear nuestro primer proyecto y nos aparecerá una serie de opciones.

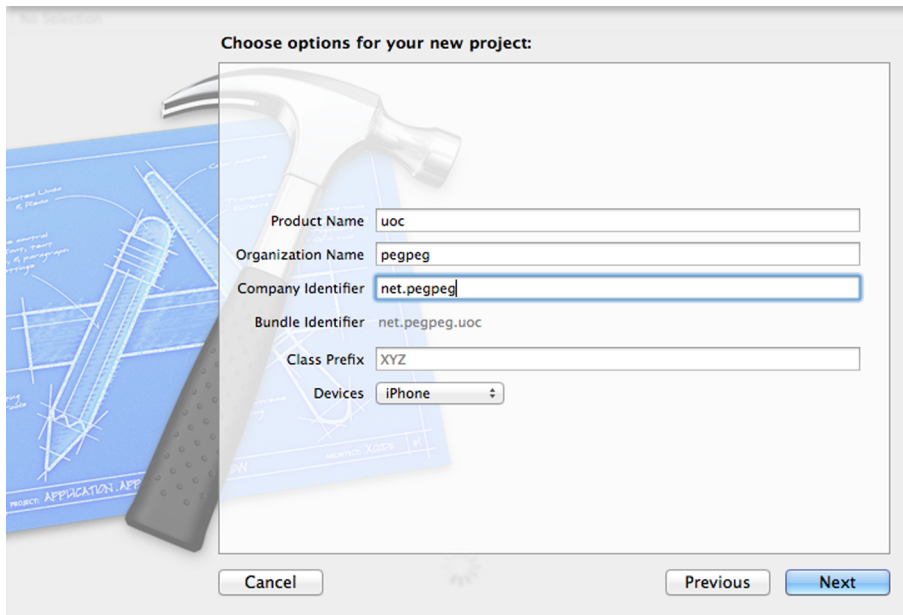
Pantalla del Xcode de creación del proyecto



Si hacemos clic en las diferentes opciones, veremos una pequeña explicación de cada una en la parte inferior. Para nuestra aplicación, escogeremos la opción de Single View Application, con la cual crearemos el esqueleto de la app con un storyboard asociado donde estará definida una sola vista o pantalla.

Le daremos a Next y nos aparecerá una pantalla donde se nos pedirá que le pongamos un nombre a nuestra aplicación.

Pantalla del Xcode de configuración de un proyecto nuevo



El nombre de nuestra aplicación es el Product Name (lo podremos cambiar más adelante); también tenemos que dar un nombre de organización y un Company Identifier. El Company Identifier nos definirá el Bundle Identifier; este es el nombre interno de la aplicación que tiene que ser único en el iTunes.

Es como una dirección web inversa, donde primero está la terminación del dominio, después el nombre del dominio y por último el nombre de la app.

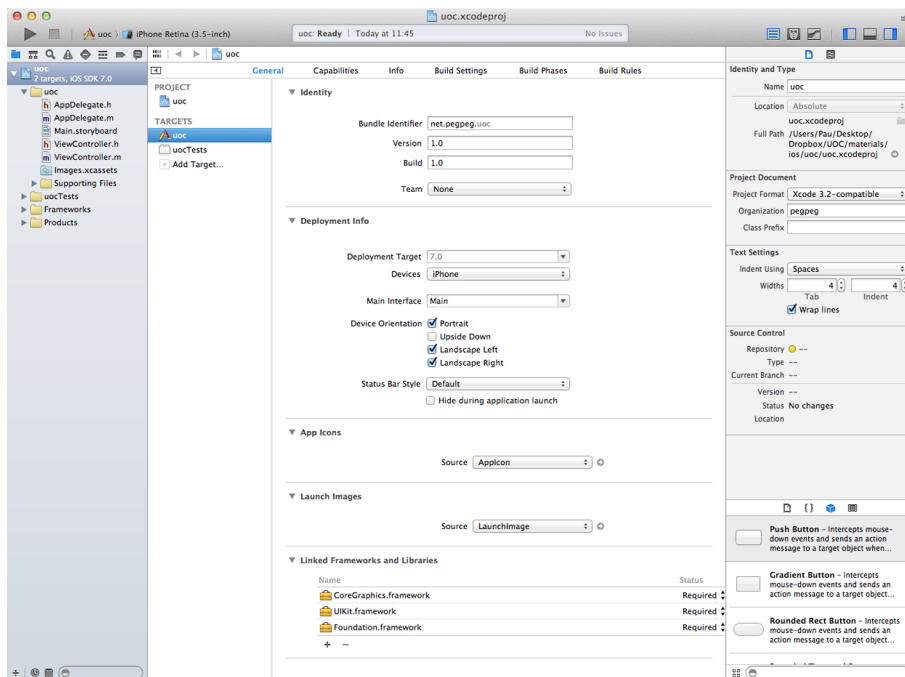
En nuestro caso, vemos que el Bundle Identifier nos ha quedado como net.pegpeg.uoc.

Para hacer pruebas en el simulador el Bundle Identifier no es importante, sí que lo sería a la hora de instalar la app en un dispositivo o subir la aplicación al iTunes, puesto que para hacerlo el Xcode tiene que firmar la app y solo la puede firmar con nuestra identidad, que tiene que concordar con el Bundle Identifier. En este caso, el Bundle Identifier tendría que concordar con la App ID que hemos definido en el fichero mobileprovision con que firmamos la aplicación.

Por último, en Devices podemos definir si nuestra app será compatible solo para iPhone, para iPad o será Universal y, por lo tanto, compatible con los dos dispositivos. Para facilitar el ejemplo, solo la haremos compatible para iPhone, le daremos a Next y escogeremos la carpeta donde guardar el proyecto.

Finalmente, se nos abrirá la siguiente pantalla:

Pantalla de configuración principal de nuestra aplicación

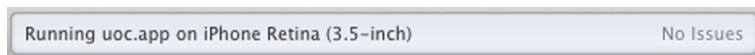


En la parte superior encontramos el icono de Play y Stop, con los que podremos ejecutar y parar la ejecución de nuestra aplicación. Haciendo clic en el lado donde pone uoc o iPhone Retina podremos seleccionar qué simulador queremos utilizar (si tuviéramos un iPhone o iPad conectados también veríamos la opción de instalar la app en este dispositivo).

Si le damos a Play con uno de los simuladores seleccionados, se nos abrirá este y nos mostrará nuestra aplicación; por ahora es solo una pantalla con fondo blanco.

En la barra superior, en el centro, encontraremos un recuadro donde veremos las acciones que está realizando el Xcode; normalmente veremos las acciones de compilación y el mensaje de si ha habido errores.

Captura de la barra de información del Xcode



En la parte derecha de la barra superior, las tres primeras opciones nos permiten modificar la visualización del código; podemos verlo en una sola pantalla, en dos pantallas (la segunda pantalla nos la muestra de forma automática según cuál sea la primera), o visualizar los cambios realizados en las diferentes versiones.

Captura con los diferentes modos de visualización de la pantalla



Los otros tres iconos nos permiten mostrar o esconder las diferentes barras.

Captura con los accesos a las diferentes barras de herramientas disponibles en el Xcode



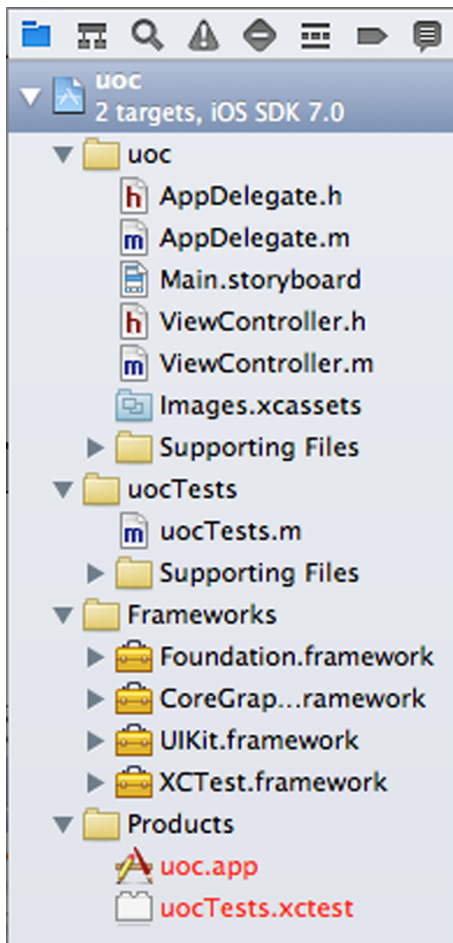
La barra izquierda o Navigator nos permite navegar entre ficheros, clases, buscas, errores, puntos de depuración, etc. Podemos movernos de pestaña en la barra Navigator, con la combinación de teclas `Cmd (#) + número de pestaña`.

La barra inferior o Debug, donde veremos la información del depurador y la información de la consola.

Y por último, la barra derecha o Utilities, donde encontramos las propiedades de los elementos, elementos UI disponibles, etc. Al igual que la barra de Navigator, esta tiene una serie de pestañas; también podemos movernos con la combinación `Cmd (#) + Opt + número de pestaña`.

En el Navigator aparecen una serie de botones en la parte superior que hacen la función de pestañas; la primera nos muestra un árbol desplegable con los ficheros que utiliza actualmente nuestra aplicación.

Captura donde se muestra el árbol de ficheros creados de forma automática por el Xcode



Los ficheros creados más importantes son:

1) AppDelegate

Es la clase encargada del funcionamiento de la app y de recibir sus mensajes. Implementa el delegate <UIApplicationDelegate>, por lo que recibe los mensajes de este como el mensaje cuando se inicia la app, cuando el usuario la cierra, cuando el usuario la vuelve a abrir desde la multitarea, etc.

2) Main.storyboard

Este es el fichero del StoryBoard, el fichero donde crearemos la interfaz de nuestra aplicación. En el supuesto de que la aplicación fuera universal, tendríamos uno para iPhone y otro para iPad. Como veremos más adelante, es una herramienta bastante intuitiva que nos permite crear interfaces de forma muy rápida.

3) ViewController

Es el controlador de la única vista de nuestra aplicación. Recordemos que hemos elegido la opción Single View Application, por lo que nos ha creado una vista por defecto.

Otros ficheros a comentar serían la carpeta uocTests, donde almacenar el código de los tests, en el caso de desarrollar usando TDD o Test Driven Development.

En la carpeta de frameworks vemos los frameworks que tenemos actualmente enlazados. Los frameworks son librerías que podemos utilizar; en nuestro caso, Xcode nos ha enlazado las imprescindibles: Foundation, de la que hemos hablado anteriormente y donde tenemos los tipos básicos para trabajar, como NSString, NSNumber, etc.; UIKit, con las clases de todos los elementos de la interfaz o UI; CoreGraphics, con herramientas de dibujo; y XCTest, para poder hacer tests.

Por último, encontramos la carpeta Products, donde veremos los ficheros que estamos generando, en nuestro caso la aplicación uoc.app.

A continuación, volveremos a abrir la pantalla de preferencias del proyecto que hemos visto inicialmente; para hacerlo, seleccionaremos el proyecto en la parte superior de la barra Navigator.

Es una de las pantallas más importantes ya que podemos modificar todas las preferencias de nuestro proyecto y de nuestra aplicación.

Tenemos que distinguir entre proyecto y targets; un proyecto puede tener uno o muchos targets. Tanto el proyecto como los targets tienen sus propias preferencias.

Las preferencias generales del Project son heredadas por todos los targets.

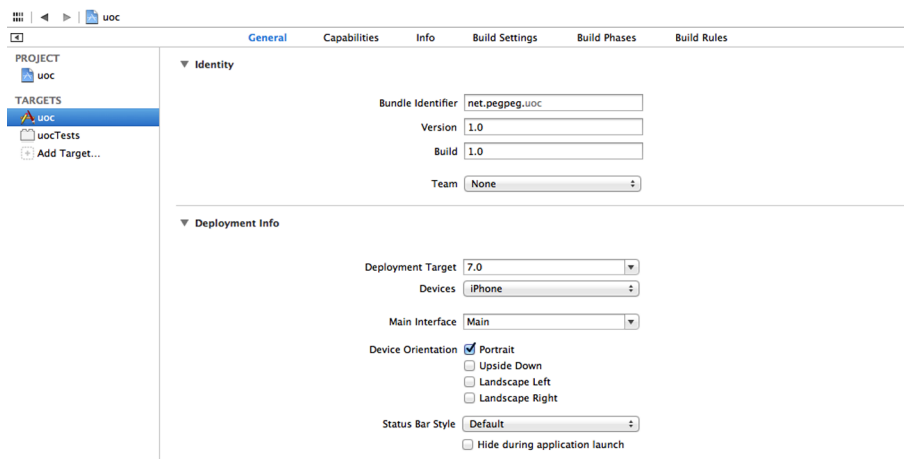
Un target es una determinada aplicación; nos puede interesar que varias aplicaciones compartan el mismo código, si varían muy ligeramente unas de otras. Por ejemplo, ofrecemos una aplicación de venta de coches a diferentes empresas, con las mismas funcionalidades pero diferente diseño.

Cada target hereda las preferencias del proyecto si no tiene sus propias preferencias. En nuestro caso, con un solo target, lo habitual es modificar las preferencias del target y no del Project, y dejar la configuración del Project para proyectos más complejos que utilicen muchos targets.

Enlace recomendado

Más información en: http://en.wikipedia.org/wiki/test-driven_development

Captura con la pantalla de configuración principal de nuestra aplicación



Por lo tanto, seleccionaremos nuestro target uoc en la barra izquierda.

Se nos mostrará una serie de opciones; no entraremos en detallarlas, pero destacaremos las más importantes.

1) Pestaña General

En esta pestaña podemos definir gran cantidad de propiedades, como la versión de nuestra app, el deployment target (es la versión del iOS mínima que soporta la app), si la app es universal, las orientaciones que soporta, los iconos, los splash screens o pantallas iniciales, los frameworks que utiliza, etc.

Sobre el deployment target, podemos compilar utilizando por ejemplo el iOS SDK 7.0 y definir un deployment target del 6.0; con esto estamos indicando que nuestra app puede funcionar con el iOS 6.0, pero somos nosotros los que tendremos que comprobar que esto sea verdad y que no haya ningún problema.

Habitualmente, Apple permite definir como deployment una o dos versiones del iOS por debajo. Por ello, no podremos compilar para el iOS 7 y hacer una app compatible para el iOS 3, por ejemplo; compilar con el iOS7, por ejemplo, solo permite hacer deployment hasta el iOS6.

Esto puede ser problemático en algunos casos, pero como ya hemos comentado, por suerte el iOS sufre muy poca fragmentación entre versiones del iOS. En este proyecto, dejaremos el deployment en 7.0, por lo que la app solo será compatible con el iOS 7.

En esta pestaña también desactivaremos las opciones de Device orientation que no soportará la aplicación que desarrollaremos más adelante; solo dejaremos la opción de Portrait seleccionada.

2) Pestaña Capabilities (disponible a partir de la versión 5 de Xcode)

Aquí encontramos algunos de los servicios que nuestra app puede utilizar; nos automatiza algunas de las gestiones que antes se tenían que hacer de forma manual al utilizar servicios como iCloud, Game Center, In-App Purchase, Passbook, etc.

3) Pestaña Info

Esta pestaña es en realidad un fichero muy importante llamado Info.plist, que acompaña siempre a la app y nos permite configurar algunas cosas importantes como el nombre de la app, los iconos, el idioma por defecto, etc. Algunas preferencias de la pestaña General se aplican en este fichero de forma automática.

Utilizando la pestaña Info nos es más fácil visualizarlo o modificarlo que con un editor de texto.

4) Pestaña Build Settings

Aquí encontramos las preferencias de compilación de nuestra app; pueden asustar un poco inicialmente, pero no se acostumbran a modificar muchas.

Las más importantes serían: Product Name, con que podemos modificar el nombre de nuestra aplicación, y Code Signing Identity, con que definimos con qué perfil (mobileprovision) queremos firmar la aplicación.

5) Pestaña Build Phases

En esta sección encontramos los ficheros utilizados por parte de nuestra aplicación cuando se compila; a veces nos puede ser de ayuda para arreglar problemas, como duplicados, ficheros no encontrados, etc.

2.2. Wireframes

Antes de empezar a programar es muy importante tener claro cuál es el objetivo del desarrollo, lo que supone tener muy definidas las diferentes pantallas y el funcionamiento de la app.

Para hacerlo, se utiliza lo que se denominan wireframes; hay muchas herramientas que podemos utilizar para crearlos; actualmente, existen muchos servicios web que nos pueden facilitar el trabajo; podéis encontrar un listado de ellos en:

<http://mashable.com/2013/04/02/wireframing-tools-mobile/>

<http://webdesignledger.com/tools/13-super-useful-ui-wireframe-tools>

También podemos utilizar programas de diseño como Photoshop o FireWorks, para facilitarnos el trabajo. Si escogemos esta opción, hay plantillas GUI que podemos utilizar para facilitar el trabajo.

Estas plantillas disponen de todos los elementos de la interfaz de un dispositivo en un fichero PSD.

Algunas de las más conocidas son las de teehan.

- **iPhone iOS 6:** <http://www.teehanlax.com/blog/ios-6-gui-psd-iphone-5/>
- **iPhone iOS 7:** <http://www.teehanlax.com/tools/ios7/>
- **iPad:** <http://www.teehanlax.com/tools/ipad/>

O incluso podemos hacer un borrador con lápiz y papel.

Lo importante es que, cuanto más definidos tengamos nuestros wireframes, menos problemas y dudas nos encontraremos a la hora del desarrollo.

2.3. Desarrollo con Xcode

Como hemos comentado anteriormente, la programación con Objective-C con el Xcode sigue la estructura Model-View-Controller.

La parte de Model corresponde a los datos, los cuales suelen estar en una base de datos Core Data, un fichero XML, un fichero PLIST (variante de XML que utiliza Objective-C) o cualquier otro fichero local o remoto donde guardamos los datos de la aplicación.

La parte de Controller es la que se encarga de definir el funcionamiento y la lógica de nuestro programa; en el Xcode los controllers están definidos mediante clases, que heredan de la clase UIViewController. En el proyecto que hemos creado, el Xcode nos ha creado los ficheros ViewController.h y ViewController.m, que definen el controlador de nuestra vista.

Y por último está la parte de View, que en nuestro caso estará definida toda dentro de un solo fichero storyboard. Anteriormente, los elementos se definían en ficheros separados con el formato XIB. Con los storyboards, esto ya no es necesario.

Cuando definimos una interfaz añadiendo elementos como buttons, desplegables, etc., siempre lo haremos sobre una vista o UIView; podemos entender un UIView como un contenedor de objetos o un folio en blanco.

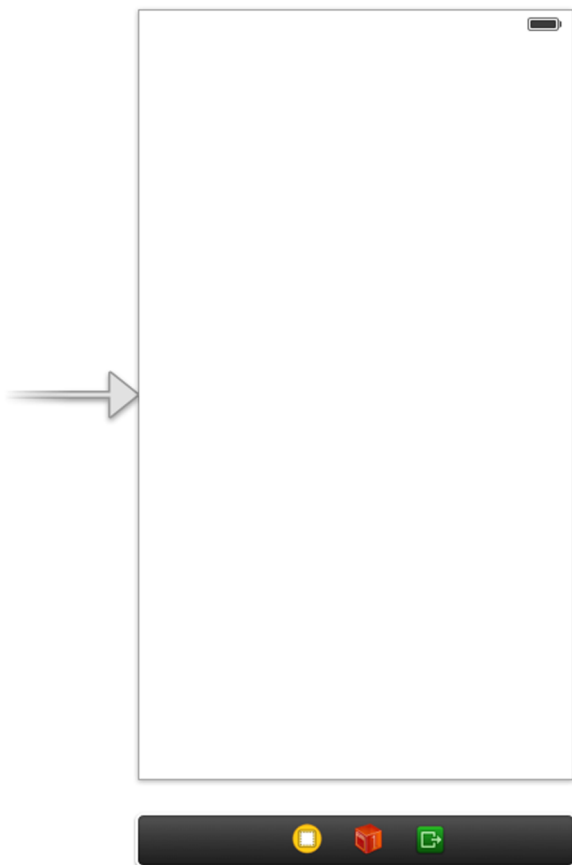
Cada UIViewController está enlazado a un UIView, y es desde el UIViewController como podemos modificar el comportamiento de elementos dentro del UIView.

2.3.1. Pantalla de login

Con esto explicado empezaremos construyendo la interfaz de nuestra aplicación; para hacerlo, seleccionaremos el fichero Main.storyboard en el Navigator (barra izquierda).

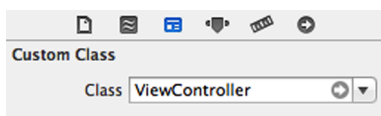
Se nos abrirá el storyboard; dentro de este solo encontraremos una vista asociada a la clase ViewController que nos ha creado el Xcode de forma automática. También veremos una flecha que apunta a esta Vista; la flecha es lo que indica la primera Vista que se cargará al iniciarse la aplicación.

Captura del elemento creado de forma automática dentro de nuestro storyboard



Seleccionaremos esta Vista y, en la barra derecha o Utilities, iremos a la tercera pestaña (Identity Inspector). Aquí es donde está definido el UIViewController que define esta vista; en nuestro caso encontraremos el nombre ViewController, puesto que es como se llama nuestro controlador.

Captura del Identity Inspector donde podemos definir la clase a que pertenece cada objeto del storyboard




Esta Vista la aprovecharemos para hacer una pantalla donde pedir los datos de acceso a nuestra aplicación.

En la parte inferior de Utilities encontraremos todos los elementos y controles que podemos utilizar para crear nuestra interfaz; los primeros con una redonda son View Controller, Table View Controller, Navigation Controller, etc.; a pesar de que el nombre nos puede confundir, son Vistas que enlazan a un tipo de Controller determinado. Si no definimos ningún Custom Class, el encargado de gestionar estas vistas será el controlador por defecto de cada una.

Más abajo ya encontramos los controles que podemos utilizar; para empezar, arrastraremos una serie de elementos a este ViewController.

Captura con todos los elementos UI que tenemos a nuestra disposición para crear la interfaz de nuestra aplicación



View Controller – A controller that supports the fundamental view-management model in iPhone OS.







Table View Controller – A controller that manages a table view.




Collection View Controller – A controller that manages a collection view.




Navigation Controller – A controller that manages navigation through a hierarchy of views.




Tab Bar Controller – A controller that manages a set of view controllers that represent tab bar...



Page View Controller – Presents a sequence of view controllers as pages.



GLKit View Controller – A controller that manages a GLKit view.



Object – Provides a template for objects and controllers not directly available in Interface Builder.

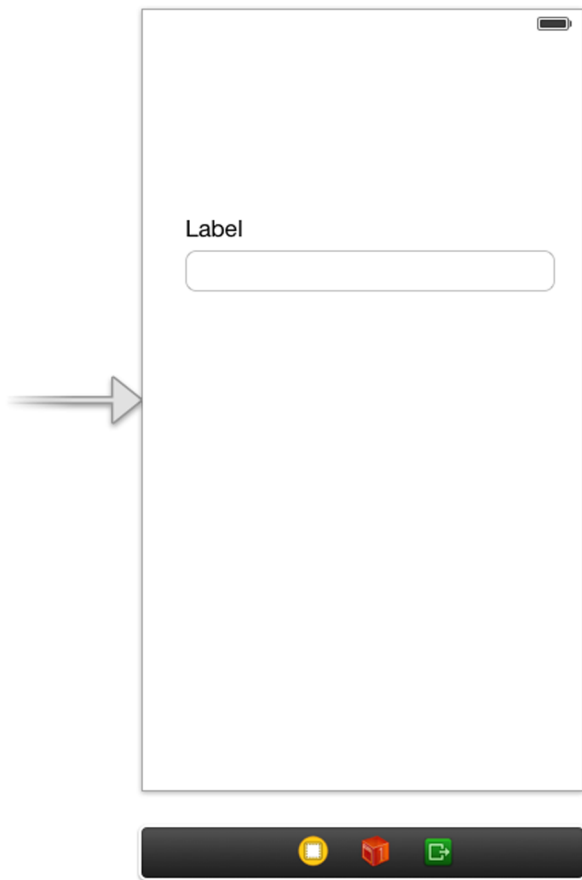
Label **Label** – A variably sized amount of static text.

Button **Button** – Intercepts touch events and sends an action message to a target object when it's tapped.

1 2 **Segmented Control** – Displays multiple segments, each of which functions as a discrete button.

Empezaremos arrastrando un elemento UILabel y otro UITextField; tendríamos que tener un resultado como el siguiente:

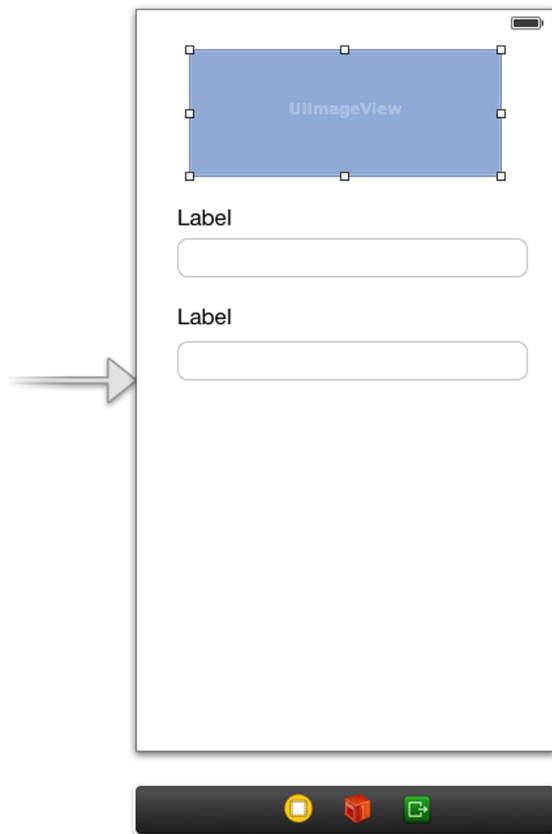
Captura con el View inicial con los primeros objetos creados



A continuación, con la tecla Cmd, seleccionaremos los dos elementos y después, con la tecla Opción, arrastrando la selección podremos duplicar los elementos. Si nos fijamos, cuando movemos los elementos aparece una serie de guías e información de píxeles que nos pueden ser útiles a la hora de alinear nuestros objetos.

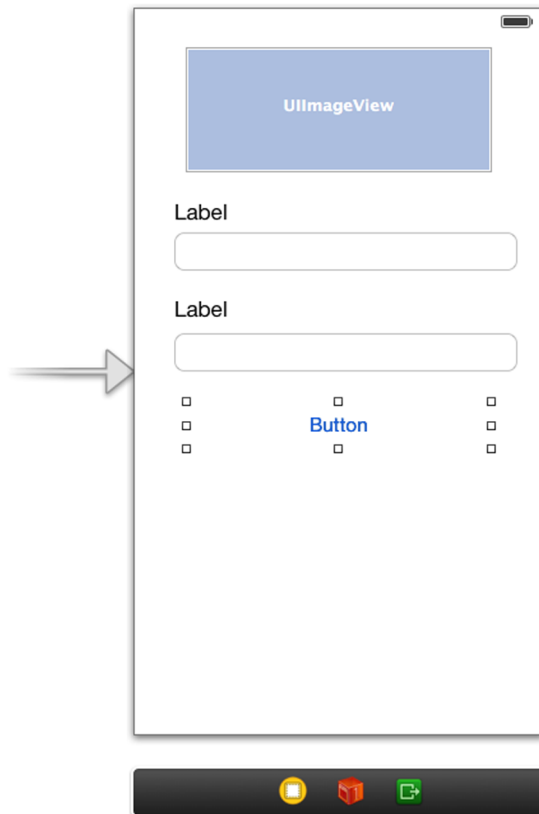
También añadiremos un elemento Image View en la parte superior, lo escalaremos desplazando los cuadrados blancos que aparecen al seleccionarlo y lo centraremos en la Vista.

Captura con el resultado de colocar los primeros objetos en la Vista inicial



Por último, incluiremos un componente Button, para poder tener un botón para hacer el login. Cuando lo añadamos también lo escalaremos para ganar más área, puesto que si no, sería demasiado pequeño para que el usuario hiciera clic con el dedo.

Captura con la colocación del botón de aceptar



Cuando seleccionamos un control, podemos editar sus atributos con el Atributes Inspector; es la cuarta pestaña en la barra derecha de Utilities; podemos cambiar de pestaña en la barra Utilities, con la combinación de teclas Cmd + Opción + número de pestaña.

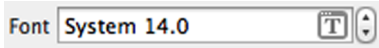
Ahora acabaremos de modificar la apariencia de esta pantalla por la que nos guste más.

Empezaremos modificando el color de fondo de la Vista por uno azul; para hacerlo, seleccionaremos en la barra izquierda del storyboard el objeto View, y en la barra de Utilities cambiaremos la propiedad Background por un color azul.

Seguidamente, modificaremos los textos de las etiquetas poniendo "User" y "Password"; poniendo como placeholder una frase como "Enter your username" y "Enter your password", modificaremos la medida de los labels para que se muestre todo el texto y reduciremos un poco el tamaño de la tipografía con el atributo Font; por último, cambiaremos el color de los labels por el blanco.

En cuanto al botón, también haremos algunos cambios: le modificaremos el texto por “Login”, le cambiaremos el color del texto también por el blanco, y, para resaltarlo más, lo meteremos en negrita; para hacerlo, lo cambiaremos pulsando sobre el cuadrado con la letra “T” que aparece junto al atributo Font, dentro de la barra Utilities.

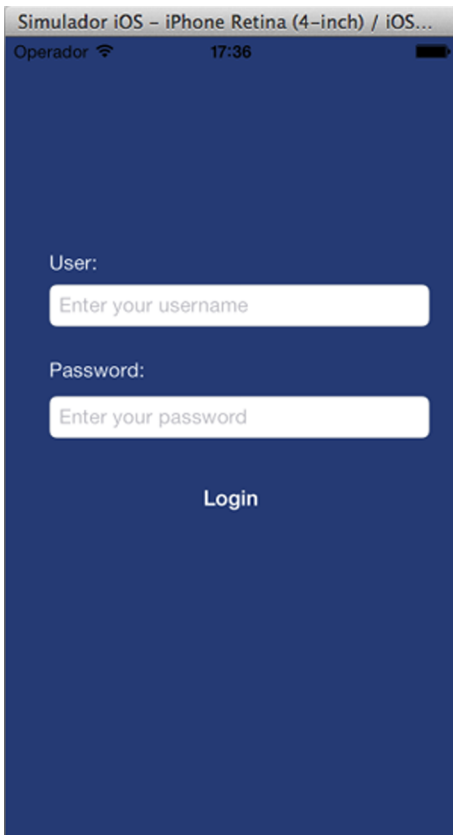
Captura detallada con la herramienta de modificación de textos



Seleccionaremos en la parte superior uno de los emuladores iPhone con la última versión del iOS disponible y probaremos cómo se visualiza nuestra aplicación; nos interesa siempre probar nuestra app con el último iOS disponible primero; después ya haremos los cambios necesarios para que también funcione correctamente con las versiones anteriores.

Veremos cómo ya empieza a coger forma nuestra aplicación y que, si hacemos tapón dentro de alguno de los campos de introducción de texto, se nos mostrará el teclado de entrada de datos.

Captura con los últimos cambios aplicados a los elementos



A continuación, añadiremos una imagen en la parte superior, y buscaremos una imagen para colocar en la parte superior; en este caso, utilizaremos un logo de la UOC. Podemos utilizar diferentes formatos compatibles de imagen,

pero el formato mejor soportado es el formato PNG a 24 bits, con el cual podemos trabajar con transparencias sin problemas. Para hacer los elementos de interfaz, por lo tanto, utilizaremos el formato PNG siempre que sea posible.

Respecto a las imágenes, también es importante tener en cuenta el soporte de las pantallas “retina”; con el lanzamiento de las pantallas retina, era necesario disponer siempre de dos imágenes, una para la pantalla normal y otra al doble de tamaño para la pantalla retina; las dos tenían que tener el mismo nombre de fichero, excepto que las imágenes retina acaban con la terminación @2x antes de la extensión del fichero.

```
<nomdelaimatge>.png (imatge pantalles normal)
<nomdelaimatge>@2x.png (imatges pantalles retina)
```

Esto es importante tenerlo en cuenta al empezar el diseño de nuestra aplicación, y diseñarla ya con la medida de pantallas retina.

Actualmente, ya muchos dispositivos con iOS disponen de pantalla retina; los dispositivos que se siguen utilizando que todavía no tienen son el iPad Mini y el iPhone 3GS. Si nuestra aplicación tiene que dar soporte a alguno de estos dispositivos, es recomendable añadir también imágenes con versión no retina.

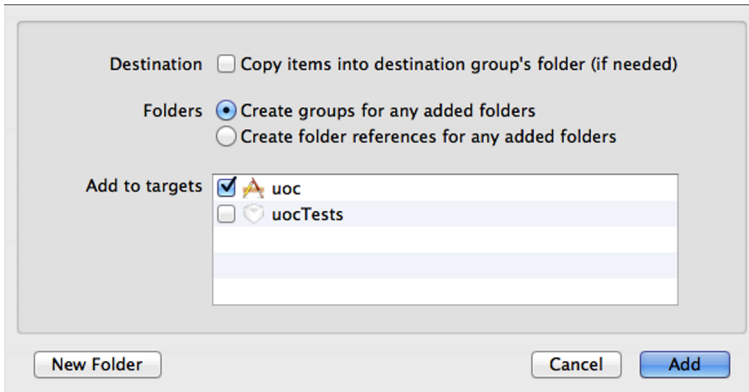
Apple recomienda incluir las dos versiones de la imagen; de todos modos, iOS, si no dispone de alguna versión de la imagen, escalará la que tenga disponible para mostrarla de forma automática con un coste computacional por eso. Por lo tanto, si necesitamos reducir la medida de nuestra app por alguna razón, siempre tenemos la posibilidad de incluir solo las imágenes retina y que iOS escale a la mitad las imágenes para los dispositivos no retina.

En nuestra aplicación crearemos dos imágenes, una para cada versión, guardaremos una al doble de tamaño con la extensión @2x en la carpeta de nuestro proyecto y otra en el tamaño original sin la extensión retina. A continuación, las añadiremos a nuestro proyecto arrastrándolas a la barra Navigator de nuestro proyecto o con la opción de “Add File to <project>” que encontraremos en el menú File.

En nuestro caso, la opción de “Copy items into destination group” tiene que estar desactivada, puesto que habremos movido manualmente ya los archivos a nuestro proyecto.

También nos tenemos que asegurar de tener seleccionado nuestro target a la hora de incluir los ficheros. En este caso, uocTests no es necesario seleccionarlo ya que para este proyecto no utilizaremos los test de Test-Driven-Development (TDD).

Captura del diálogo para añadir elementos a nuestro proyecto



Le daremos a Add y se nos añadirán en la barra izquierda de Navigator. A continuación, seleccionaremos el UIImageView en nuestro storyboard y en el Attributes Inspector, en el campo imagen, seleccionaremos la imagen que acabamos de añadir; por defecto nos estirará la imagen para que ocupe toda la medida del UIImageView.

Lo que haremos es dar al UIImageView la misma medida de la imagen; para hacerlo con el UIImageView seleccionado, podemos utilizar la combinación Cmd += o seleccionar la opción Size to Fit Content del menú Editor.

Con esto el resultado tendría que ser parecido al siguiente:

Captura con la pantalla de acceso ya acabada



A continuación, haremos algunas modificaciones más en los campos de entrada de usuario y contraseña; en el panel attributes inspector, el campo Correction lo pondremos en No; para evitar que se active la autocorrección al escribir, la opción de Return Key la dejaremos en Done. Y para el campo de contraseña activaremos la opción “Secure”, para indicarle que se trata de una contraseña y nos la oculte mientras la vamos escribiendo.

Si probamos de nuevo la aplicación, veremos aplicados los cambios, pero si probamos el teclado veremos que este, una vez se ha mostrado, no desaparece cuando le damos al botón de Done. Para hacerlo, tendremos que utilizar un delegate; esto significa asignar al TextField una clase que se encargue de gestionar su funcionamiento; lo más común es que el delegate sea la clase que contiene el objeto, por lo tanto ViewController.

Para indicar que la clase ViewController implementa este Delegate, abriremos el archivo ViewController.h, y modificaremos el código:

```
@interface ViewController : UIViewController
```

por el código

```
@interface ViewController : UIViewController <UITextFieldDelegate>
```

Con esto estamos indicando que esta clase implementa los métodos del delegate UITextFieldDelegate. Para saber qué métodos son necesarios implementar, tendremos que consultar la documentación o la declaración de UITextFieldDelegate. La manera más sencilla de hacer esto es hacer clic sobre el texto en cuestión, con la tecla Cmd apretada para ver la declaración o la tecla Option para abrir la documentación.

Si consultamos la declaración de UITextFieldDelegate, veremos que todos los métodos son opcionales, por lo que no es obligatorio implementar ninguno; pero para implementar la funcionalidad del botón Done del teclado, necesitaremos aplicar el siguiente método.

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField;
```

Lo que haremos será incluir este código en el fichero ViewController.m, con la implementación del método.

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField{  
  
    [textField resignFirstResponder];  
    return YES;  
}
```

Este método se llama al tocar sobre el botón de Return. Como vemos por la declaración, este método recibe el textField que ha generado la acción y devuelve un booleano indicando que se ha introducido correctamente la información; por ahora no implementaremos ningún tipo de validación, solo esconderemos el teclado; para hacerlo llamamos al método `resignFirstResponder`, con lo cual indicamos al textField que ya no tiene el foco y por lo tanto se esconde el teclado de forma automática. No existe ninguna función para mostrar y esconder el teclado directamente, sino que, cuando un campo de entrada del texto recibe el foco, el teclado se muestra, y cuando lo pierde, el teclado se esconde.

Si probamos la aplicación de nuevo, veremos que todavía sigue sin funcionar; esto es debido a que nos falta asignar el delegate a los Textfields en el storyboard; para hacerlo abriremos el storyboard y seleccionaremos uno de los Textfields, abriremos el Connections Inspector `Cmd+Opt+6` y, en la sección Outlet, veremos que no hay ningún delegate asignado. Para asignarlo, arrastraremos el círculo del delegate hasta encontrar la redonda amarilla en el Document Outline del ViewController donde acabamos de implementar el delegate. Haremos lo mismo para el otro TextField, para tener los dos TextFields apuntando al delegate de nuestra clase.

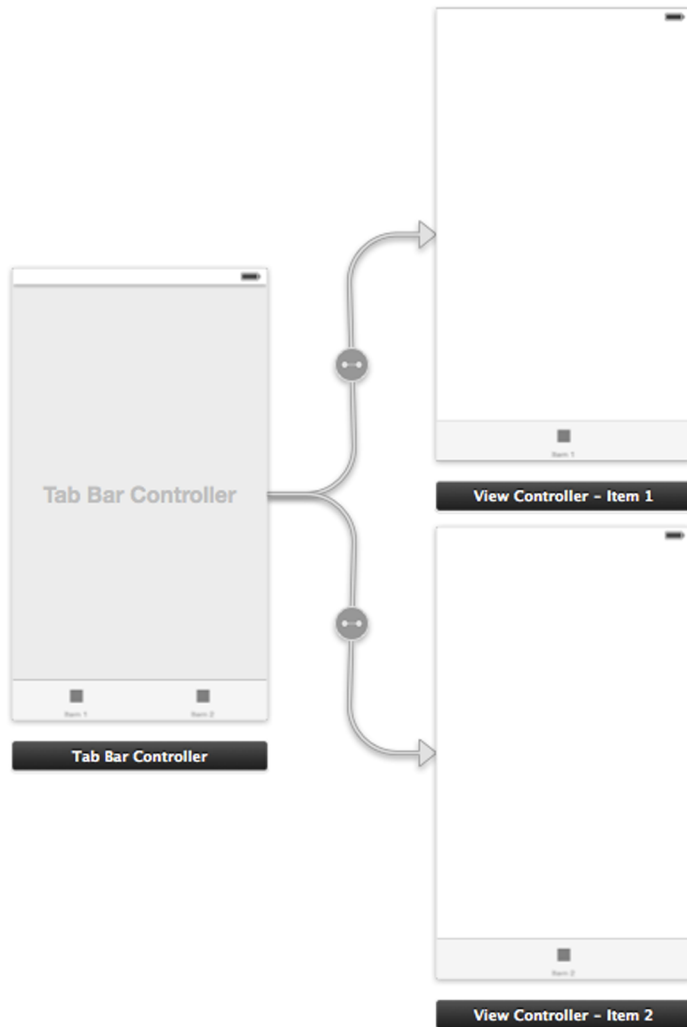
Con esto, si probamos la aplicación, finalmente obtendremos el resultado deseado, es decir, al pulsar el botón Done del teclado, nos desaparecerá el teclado, y con el botón Done podremos saltar a la siguiente pantalla.

2.3.2. Pantalla de email

Para continuar con nuestra aplicación añadiremos una nueva pantalla, que se mostraría después de hacer el login correctamente. Dispondrá de una botonera inferior donde poderse mover entre diferentes pantallas. Esta botonera se llama `UITabBar` y existe un controlador llamado `UITabBarController` que nos facilita el trabajo.

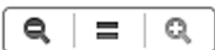
Por lo tanto, el siguiente paso será buscar en el listado de componentes Object Library de la barra derecha de Utilities el controlador llamado “Tab Bar Controller” y arrastrarlo dentro de nuestro storyboard; lo colocaremos en la derecha de nuestra pantalla de login.

Captura del elemento Tab Bar Controller con los View Controllers asociados por defecto



Para trabajar mejor dentro de un storyboard, podemos utilizar la lupa para aumentar o disminuir el zoom; lo que hay que tener en cuenta es que solo podemos editar los objetos de un controlador con el zoom al máximo. Podemos encontrar la lupa en la parte inferior derecha de la pantalla.

Captura en detalle de la herramienta lupa



Como vemos, una vez arrastrado el controlador, nos crea de forma automática un Tab Bar Controller con dos botones, y dos View Controllers enlazados con unas flechas. Estas flechas se llaman segues y nos permiten definir la navegación de nuestra aplicación.

En este caso, cada botón enlaza un segue² a un View Controller diferente; recordemos que podemos modificar también las propiedades de los segues haciendo clic sobre ellos y modificando sus atributos como si se tratara de un objeto.

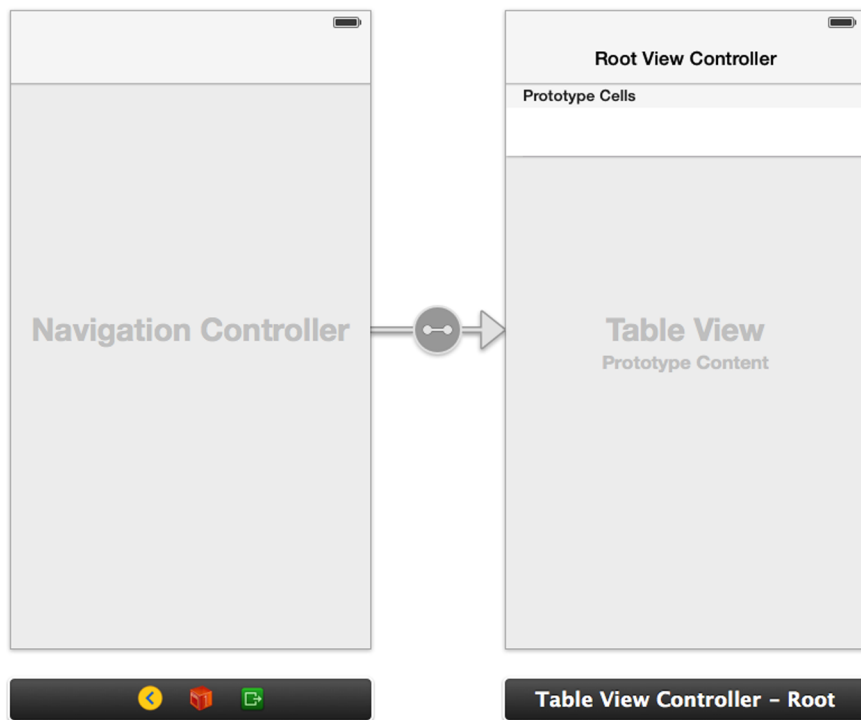
⁽²⁾Segue: transición entre pantallas.

Empezaremos creando un nuevo segue para enlazar nuestra pantalla de login con el Tab Bar Controller; de este modo, al tocar sobre el botón se nos abrirá la pantalla con la barra inferior.

Para hacerlo, seleccionaremos el botón Login y con el botón Ctrl apretado arrastraremos desde el botón Acceder hasta el controlador Tab Bar Controller; mientras arrastramos tendríamos que ver una línea azul que nos indica el enlace. Una vez seleccionado el Tab Bar controller, nos pedirá el estilo del segue: puede ser Push, Modal o Custom; comentaremos los diferentes modos a continuación.

El estilo Push implica que hay una jerarquía o niveles de View Controllers; en la navegación habitual, un ViewController abre segundo y este tiene un botón de atrás en la parte superior para volver al primero.

Captura de un Navigation Controller con un Table Controller asociado por defecto



Para utilizar la opción Push es necesario la existencia de un Navigation Controller para gestionarlo; este es el encargado de controlar la navegación en la aplicación, nos permite cargar diferentes View Controllers en serie utilizando una barra de navegación superior, que nos permite realizar acciones o volver atrás; es uno de los controladores más utilizados habitualmente, puesto que la

barra de navegación superior la encontramos prácticamente en todas las aplicaciones. Es posible por eso utilizar un Navigation Controller sin tener que mostrar la barra de navegación.

La opción Modal es diferente, pues funciona como un Popup, muestra el View-Controller por encima del actual y sin modificar la jerarquía que hubiera hasta entonces.

Para enlazar el login al Tab Controller lo haremos con el tipo Modal, puesto que no consideramos la pantalla de login como la primera de la jerarquía; así, para hacer el logout, solo tendremos que esconder la ventana modal. También modificaremos el tipo de animación; haciendo clic sobre el nuevo segue escogeremos como transición la opción Flip Horizontal.

Y finalmente, encontramos el tipo Custom, con el que podremos definir el tipo de transición nosotros mismos.

Si probamos la aplicación, veremos que ya tenemos los dos controladores enlazados y al tocar sobre el botón se nos abrirá la siguiente pantalla.

Ahora nos meteremos a trabajar en esta nueva pantalla; para ahorrarnos tiempo cuando probemos la aplicación dentro del storyboard, podemos mover la flecha que indica la primera pantalla de la aplicación a la pantalla que estemos trabajando; de este modo no tendremos que hacer clic en acceder cada vez cuando probemos la aplicación.

Empezaremos eliminando los nuevos View Controllers asociados al Tab Bar Controller y creando uno nuevo a la derecha del Tab Controller. Este primer controlador será un Navigation Controller como el que hemos introducido anteriormente.

El siguiente paso será indicar al Tab Controller que queremos utilizar este Navigation Controller como primera sección en nuestra barra; para hacerlo, realizaremos la misma acción que antes: con la tecla Ctrl haremos clic sobre el Tab Controller arrastrando hasta el Navigation Controller, se nos mostrará el tipo de relación que queremos asignarle y escogeremos la opción de "Relationship Segues view controllers". Seleccionando el icono de la barra inferior del Navigation Controller, podremos modificar tanto el texto como el icono a mostrar; en este caso, añadiremos un nuevo icono al proyecto y modificaremos el nombre por "Inbox".

Vemos que nos ha creado también un Table View Controller asociado; este ya nos va bien, el Table View Controller es un View Controller puesto que hereda de UIViewController, pero está pensado para contener Tables; una Table o tabla no deja de ser un listado de elementos de forma vertical; cada uno de estos elementos se denomina Cell o celda.

Ejemplos de mesa:

Captura donde podemos ver dos ejemplos de Table View Controller



Las tablas están formadas por secciones, que contienen celdas o filas. Un ejemplo de secciones serían las separaciones de letras de la aplicación Agenda; también podemos definir una cabecera o header y un pie o footer en la tabla si lo deseamos.

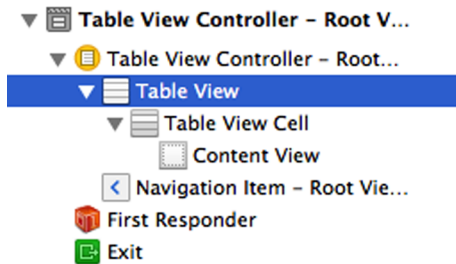
Otra característica importante a tener en cuenta es su funcionamiento: una tabla puede tener un número muy grande de celdas, por lo que para cada fila no se crea un nuevo objeto sino que se reaprovecha alguna fila que haya desaparecido de la pantalla.

Con los storyboards también podemos definir las celdas de forma cómoda, gracias a las Prototype Cells; básicamente, encontramos de dos tipos: las estáticas y las dinámicas. Las dinámicas están pensadas para recibir la información a mostrar de forma dinámica, mientras que en las estáticas la información no varía.

Las tablas están compuestas de una serie de elementos; cada tabla puede tener diferentes secciones, donde cada sección tiene sus celdas. También puede tener una cabecera o header y un pie o footer.

Definiremos las celdas como estáticas; para hacerlo, seleccionaremos Table View en el Document Outline o barra izquierda del storyboard.

Detalle donde vemos la selección solo del elemento Table View



A continuación, en el Attributes Inspector, modificaremos el Content a Static Cells, cambiaremos el atributo Style a Grouped y dejaremos Sections con el valor a 1, ya que solo queremos una sola sección en este caso.

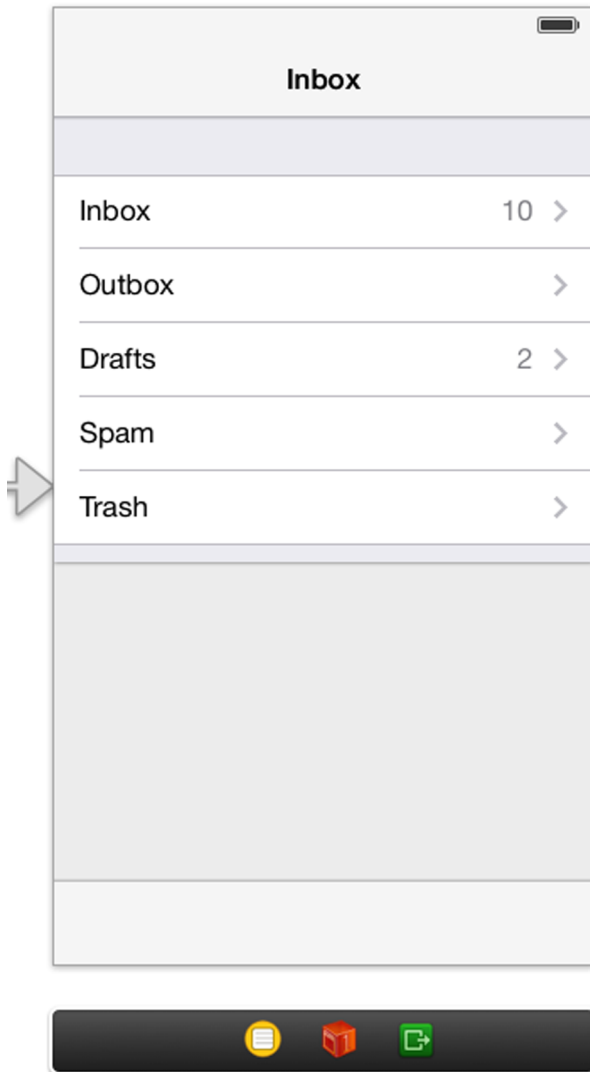
Por defecto, solo tenemos una sección con tres filas, pero necesitamos cinco; para añadir dos más, seleccionaremos Table View Section en el Document Outline y a rows le daremos el valor de cinco.

Ahora ya podremos modificar el diseño de las celdas para conseguir el efecto deseado; podemos arrastrar nosotros mismos los controles que deseamos a las celdas o utilizar un diseño predefinido si se tercia. En nuestro caso, utilizaremos el estilo ya predefinido Right Detail; para cambiar el estilo de una celda, la seleccionaremos en el Document Outline y modificaremos el atributo Style. También modificaremos el accessory a Disclosure indicator. El accessory es el icono que aparece a la derecha del todo de una celda; el disclosure en concreto es el símbolo > y nos indica que al tocar encima iremos a otra pantalla o View.

Y por último, cambiaremos el título superior del Table Controller; cambiaremos “Root View Controller” por “Inbox”.

El resultado que queremos conseguir es este:

Detalle con el resultado de la pantalla del buzón del storyboard



Una serie de celdas que simulen una pantalla donde poder acceder a las diferentes secciones de una cuenta de correo electrónico. A continuación, lo que haremos será que, al pulsar en alguna de estas celdas, se abre una nueva pantalla donde se cargará la web de la UOC. En la versión final de la aplicación, podríamos cargar las webs correspondientes a cada uno de los apartados, pero al ser una maqueta, con cargar la web de la UOC ya se ve cómo sería el proceso.

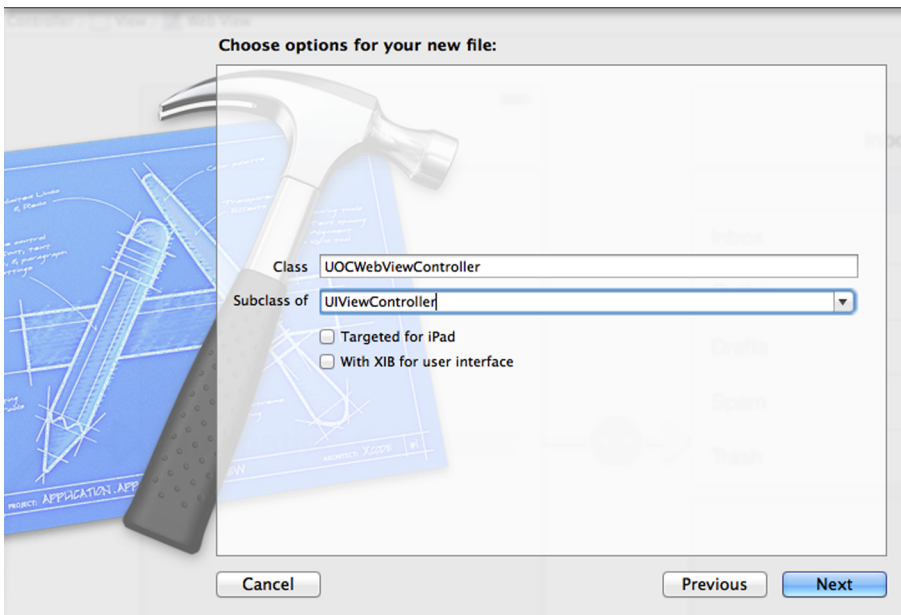
Para hacerlo, añadiremos un nuevo View Controller a nuestro storyboard, a la derecha del Table View Controller. Iremos seleccionando cada una de las celdas y creando los segues correspondientes en el nuevo View Controller; escogeremos la opción Selection segue push, puesto que queremos hacer enlazable toda la celda y no solo el icono derecho o accesorio.

A continuación, para cargar una web necesitaremos añadir el control Web View, que nos permite cargar webs en nuestras aplicaciones de forma sencilla; arrastraremos este componente al nuevo View Controller; para poder cargar

una dirección web tendremos que poder acceder al componente `UIWebView` desde el código, para indicarle la dirección a cargar, puesto que no se puede definir desde los atributos del storyboard.

Primero de todo, crearemos un nuevo archivo apretando el botón derecho sobre nuestro proyecto y escogiendo `New File`, seleccionando la opción del menú `File New File` o simplemente con `Cmd+ N`. Escogeremos el tipo de archivo `iOS > Cocoa Touch > Objective-C class`, le daremos al botón `Next` y le pondremos como `Class` el nombre de `UOCWebViewController` y como `Subclass of` `UIViewController`.

Ventana de diálogo de la creación de una nueva clase



Le daremos a `Next`, le indicaremos que queremos guardar los archivos de la clase dentro de nuestra carpeta con el proyecto de la aplicación y nos aseguraremos de nuevo de tener seleccionado el target con que estamos trabajando.

Una vez tengamos ya nuestra clase creada, lo enlazaremos al storyboard; para hacerlo, abriremos el archivo del storyboard y seleccionaremos el `View Controller` con el `WebView`; para seleccionarlo, tenemos que hacer clic en la zona negra inferior donde pone "View Controller"; en la barra de `Utilities` iremos al `Identity Inspector`, y donde pone `Class` escribiremos `UOCWebViewController`. De este modo, será esta clase la que haga de controlador de esta vista.

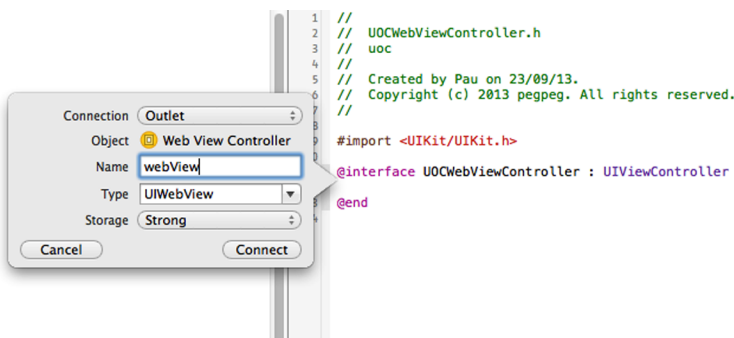
A continuación, lo que tenemos que hacer es añadir un enlace o instancia del objeto `Web View` para poder acceder desde nuestro código; esto es lo que se denomina un `outlet`. Para crearlo, seguramente la manera más sencilla es seleccionar el `Web View` en el storyboard y activar la vista `Assistant Editor`; es un icono cuadrado con un dibujo de un frac en la parte superior derecha. Con

esto nos aparecerá la pantalla dividida en dos, en la pantalla derecha abriremos el archivo header UOCWebViewController.h; es posible que se haya abierto de forma automática, si no, lo escogeremos utilizando la barra superior.

◀ ▶ | 📄 Automatic > 📄 UOCWebViewController.h > 📄 @interface UOCWebViewController

Una vez tengamos los dos archivos abiertos, apretaremos la tecla Ctrl y arrastraremos el componente Web View hacia el fichero de header abierto en la parte derecha, entre los tags @interface y @end. Nos aparecerá un cuadro de diálogo, dándonos la posibilidad de crear un outlet; le pondremos de nombre webView.

Captura donde vemos la ventana de creación de un outlet



Como vemos, al darle a Connect nos ha creado una nueva @property IBOutlet del tipo UIWebView, llamada webView, con la cual acceder al objeto del storyboard.

Los outlets son esenciales al desarrollar aplicaciones para poder enlazar los controles que nos interesan del storyboard y trabajar con ellos desde el código de nuestra aplicación.

Para cargar una página en el componente Web View, añadiremos a este una llamada a la función loadRequest dentro del método viewDidLoad.

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view.
    [self.webView loadRequest:[NSURLRequest requestWithURL:[NSURL URLWithString:@"http://
    www.uoc.edu"]]];
}
```

En este fragmento de código llamamos al método loadRequest del UIWebView para cargar una dirección web; este necesita un objeto NSURLRequest y este último un objeto NSURL; es por eso por lo que nos queda una instrucción tan larga.

Cabe decir también que el método `viewDidLoad` se llama en todos los `View-Controllers` cuando se han acabado de cargar; se llaman solo cuando este se acaba de cargar en memoria; si por el contrario quisiéramos ejecutar el código cada vez que se visualizara el `View Controller`, utilizaríamos `viewWillAppear`. Existen otros muchos métodos asociados a la clase `UIViewController` que nos pueden ser útiles; lo mejor es consultar la documentación para irlos conociendo.

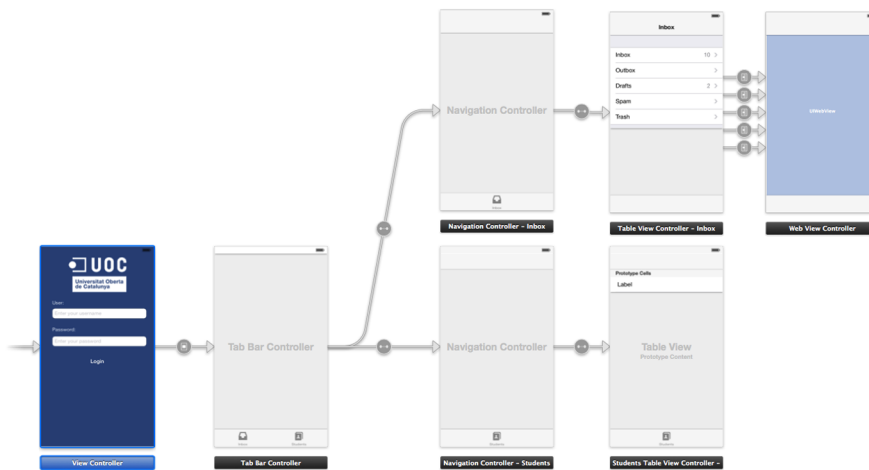
Si ejecutamos la aplicación, veremos que ahora sí que se carga la web de la UOC cuando accedemos a cualquiera de las celdas de la tabla.

2.3.3. Pantalla de Students

A continuación, crearemos una nueva pantalla en nuestro storyboard, del tipo `Navigation Controller`, incluyendo también el `TableController` que nos viene por defecto. Igual que hemos hecho antes, lo enlazaremos al `Tab Bar Controller` con la tecla `Ctrl`, por lo que ahora tendremos ya dos botones en la barra inferior. Le cambiaremos el nombre a `Students`, y si lo deseamos también le añadiremos un icono adecuado.

El resultado tendría que ser como este:

Árbol del storyboard donde vemos la navegación de nuestra aplicación



En esta tabla lo que haremos es mostrar un listado de nombres de alumnos, por lo que definiremos un solo tipo de celda, aunque esta vez de tipo dinámico. Por defecto ya se nos crea una sola celda `prototype` dinámica, por lo que no tendremos que hacer ningún cambio.

Esta vez, en lugar de utilizar un estilo predefinido, crearemos nosotros mismos los controles de las celdas. Solo añadiremos un único control `UILabel` a la celda; para hacerlo, lo arrastraremos a la celda y estiraremos el componente para que ocupe prácticamente todo el ancho.

Detalle donde vemos la creación de los elementos dentro de una celda



A continuación, tenemos que hacer otra cosa: darle un identificador a nuestra celda, pues en este caso solo tenemos de un tipo, aunque podríamos tener diferentes celdas dinámicas y las tenemos que poder diferenciar desde nuestro código. Por lo tanto, seleccionaremos en el Document Outline la celda y en el Atributes Inspector, en Identifier, escribiremos “StudentCell”.

Para las celdas dinámicas es necesario que utilicemos una clase controladora, puesto que es desde el código desde donde gestionaremos la información de esta tabla.

Crearemos una nueva clase que herede de UITableViewController y le daremos el nombre de StudentsTableViewController; como hemos hecho antes, también será necesario enlazar la vista del storyboard a su controlador; seleccionaremos la vista StudentsTableController del storyboard y en el Identity Inspector le daremos a Class el nombre de StudentsTableViewController.

Si abrimos el archivo .m recientemente creado, veremos una serie de métodos que nos ha generado por defecto, que nos servirán para definir la información que mostraremos en la tabla, en concreto nos encontraremos tres métodos del protocolo UITableViewDataSource.

A continuación, comentaremos los cambios que haremos, en el primer método llamado numberOfSectionsInTableView, donde devolveremos el número de secciones de nuestra tabla, en nuestro caso un 1.

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    // Return the number of sections.
    return 1;
}
```

En el segundo método tenemos que devolver el número de filas que deseamos que muestre la tabla; en nuestro caso, estableceremos que muestre por ejemplo 10 filas.

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    // Return the number of rows in the section.
    return 10;
}
```

Y finalmente, encontramos el tercer método, en el que podemos definir de qué manera se dibujará cada una de las filas. Este método llama a la función `dequeueReusableCellWithIdentifier` con un identificador para obtener una celda que se pueda reutilizar; si no hay ninguna se creará una nueva de forma automática.

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"StudentCell";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier: CellIdentifier
                           forIndexPath:indexPath];

    // Configure the cell...

    return cell;
}
```

Por ahora, solo modificaremos el identificador `@"Cell"` por `@"StudentCell"`, que es el identificador que hemos dado a la celda en el storyboard.

Si ejecutamos la aplicación, veremos que al acceder a la sección de Students se nos muestran 10 filas con el texto "Label". El siguiente paso, por lo tanto, será mostrar el nombre de cada alumno; para hacerlo, crearemos un array donde guardar esta información. En este caso, al ser una maqueta, no tiene importancia guardar la información en el fichero del controlador, pero en la aplicación final esta tendría que estar en un fichero de base de datos, fichero externo, cogerla de forma remota o cualquier otra opción para cumplir con el paradigma Model-View-Controller.

Crearemos una variable de tipo array; para hacerlo, abriremos el fichero header de Students y haremos los cambios para que nos quede de la siguiente manera:

```
@interface StudentsTableViewController : UITableViewController{

    NSArray *tableArray;
}
```

Hemos creado una variable llamada `tableArray`, la cual solo será accesible desde la propia clase debido a que no es ninguna `@property`.

Lo inicializaremos dentro del método `viewDidLoad` y le añadiremos algunos estudiantes de ejemplo.

```
- (void)viewDidLoad
{
    [super viewDidLoad];
```

```
tableArray = @[@"Marta Pérez",@"John doe",@"Enrique García",@"Fernando Fernández",@"Luís Rojas"];  
}
```

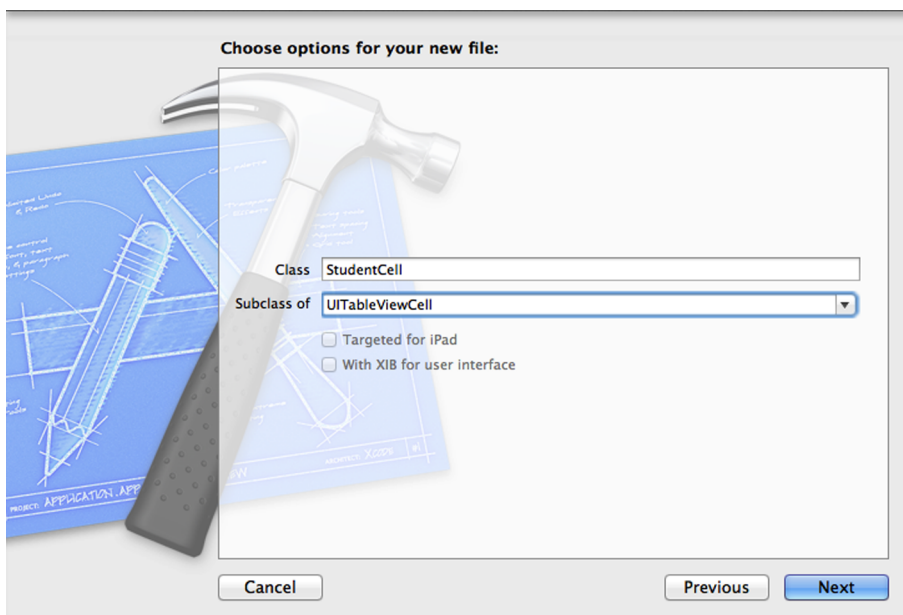
Actualizaremos también el método que nos da el número de filas para que nos dé el total de elementos del array.

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section  
{  
    // Return the number of rows in the section.  
    return [tableArray count];  
}
```

Y por último, el método `cellForRowAtIndexPath` también lo tendremos que modificar para actualizar el nombre en cada fila con el contenido del array; para hacerlo, antes tenemos que crear una clase nueva para poder acceder a los controles de la fila desde el código.

Crearemos una nueva clase de tipo Objective-C Class, de nombre de clase le pondremos "StudentCell" y que sea una subclase de `UITableViewCell`.

Ventana de diálogo para la creación de una nueva clase de tipo `UITableViewCell`



A continuación, tenemos que asociar la nueva clase a la celda del storyboard; para hacerlo dentro del storyboard, seleccionaremos la celda en el Document Outline y al Identity Inspector en Class le daremos el valor de `StudentCell`.

A continuación, nos crearemos un outlet del Label de la celda para poderla modificar desde el código.

Para hacerlo, activaremos de nuevo la vista Assistant Editor, y en la ventana derecha visualizaremos el fichero StudentCell.h. Arrastraremos con el Ctrl apretado desde el Label de la celda hasta la ventana derecha entre @interface y @end y escogeremos la opción Outlet, dándole como nombre “studentName”.

A continuación, ya podremos acabar de modificar el código de la función cellForRowAtIndexPath de StudentsTableViewController para poder modificar las diferentes celdas con el contenido del array.

Para empezar, tendremos que añadir arriba de todo la instrucción para la importación del header de la clase StudentCell; en este caso, lo podemos hacer en StudentsTableViewController.h como StudentsTableViewController.m, pues no necesitamos acceder a la clase desde StudentsTableViewController.h, solo desde el archivo .m.

```
#import "StudentCell.h"
```

A continuación, modificaremos el código de la función por el siguiente fragmento:

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"StudentCell";
    StudentCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier
                      forIndexPath:indexPath];

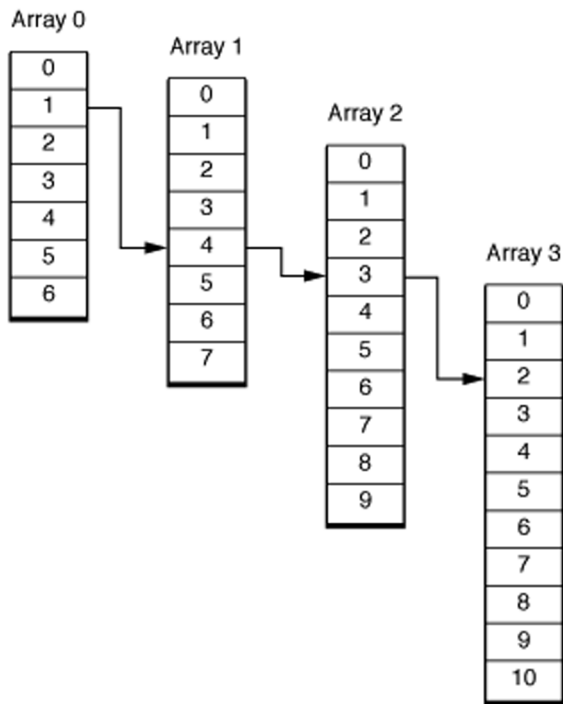
    NSInteger rowNum=[indexPath row];
    cell.studentName.text=[tableView objectAtIndex:rowNum];
    return cell;
}
```

Lo primero que hacemos es modificar la clase UITableViewCell, que nos devuelve la función dequeueReusableCellWithIdentifier por nuestra clase StudentCell, ya que en esta tabla todas las celdas son instancias de esta clase. En el supuesto de que hubiera más clases, tendríamos que quitar de la cola la clase que correspondiera según la fila que queremos dibujar.

Seguidamente, obtendremos el número de fila, mediante el objeto indexPath; la clase NSIndexPath es la clase que nos da el camino o path dentro de un árbol; de este modo, un mismo objeto NSIndexPath nos puede indicar la sección, la fila, etc. en la que se encuentra el elemento que nos interesa.

Ejemplo de IndexPath

Figure 1 Index path 1.4.3.2



En la siguiente instrucción obtendremos el objeto correspondiente a la fila de nuestro array y lo asignaremos al Label de la celda; en concreto, lo tenemos que asignar al atributo texto para poder modificar el texto.

Si ejecutamos la aplicación a continuación, veremos que finalmente se nos presenta la tabla listando todos los alumnos del array.

2.3.4. Pantalla de Mapa

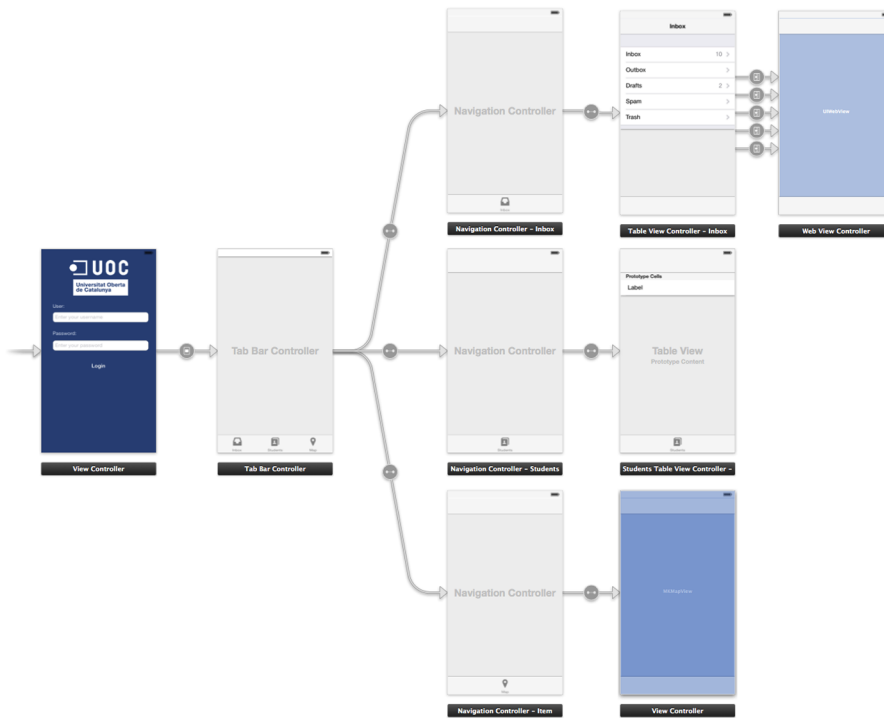
A continuación, crearemos una sección donde visualizar un mapa y donde ubicaríamos las diferentes sedes de la UOC; en este ejemplo solo nos ubicará a nosotros.

Para implementar esta pantalla, volveremos al storyboard y crearemos un nuevo Navigation Controller, pero esta vez borraremos el Table Controller asociado y añadiremos un View Controller. Asociaremos el Navigation Controller como una nueva sección del Tab Controller y el View Controller como root view controller del Navigation controller.

Desarrollar el mapa es bastante sencillo, solo tenemos que arrastrar el componente Map View al View Controller. Seleccionaremos el Map View que acabamos de añadir, y en la pantalla con los atributos, seleccionaremos la opción “Shows User Location”, para ubicar de forma automática al usuario; esto también implicará que aparezca una alerta de forma automática pidiendo permiso al usuario para ubicar su posición.

Nos tendría que quedar la siguiente estructura:

Estructura de nuestro storyboard con el nuevo elemento para visualizar el mapa



Si ejecutamos la aplicación e intentamos ver la pantalla del mapa, veremos que la aplicación se cierra y en la consola nos aparecerá el error “Could not instantiate class named MKMapView”; esto es debido a que, para utilizar los mapas, necesitaremos añadir el framework MapKit.framework. Para hacerlo, pulsaremos el icono del proyecto en la barra izquierda y en la pestaña General, en la parte inferior, veremos todos los frameworks utilizados en nuestro proyecto, le daremos al símbolo + y añadiremos el framework MapKit.framework.

Si volvemos a ejecutar la aplicación, veremos que ya se visualiza el mapa de forma correcta, y que nos aparece también la alerta pidiendo permiso para ubicarnos.

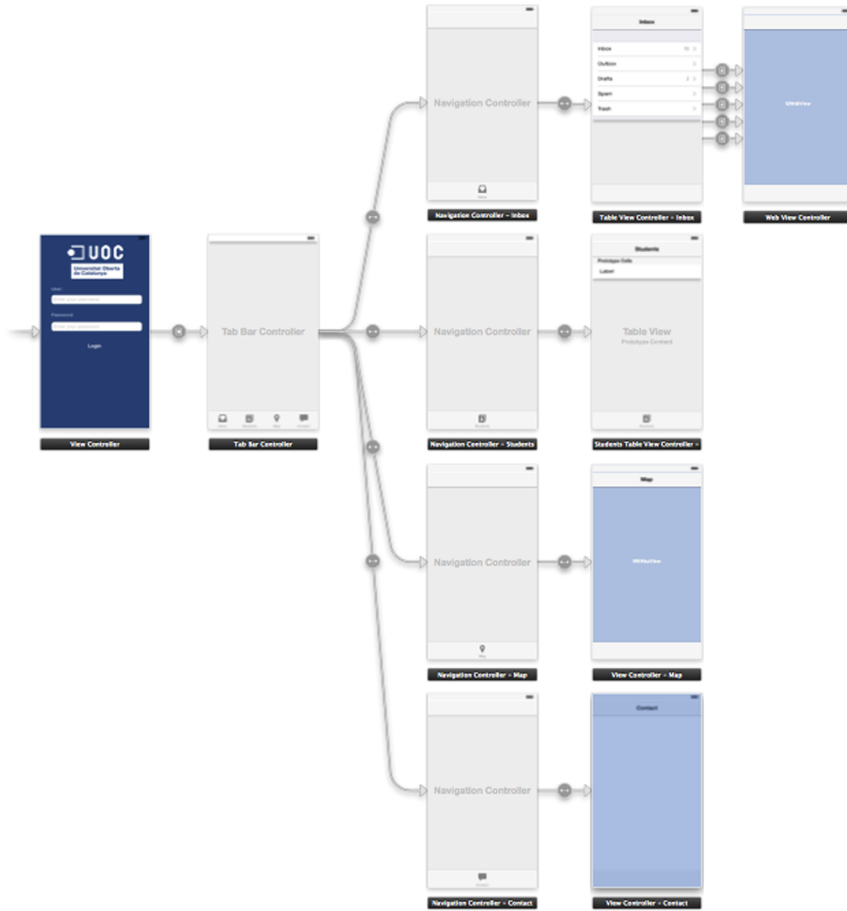
2.3.5. Pantalla de Contacto

Finalmente, haremos una última pantalla para permitir contactar vía email con los servicios de la UOC desde la misma aplicación.

Para hacerlo, volveremos a añadir un nuevo Navigation Controller asociado a un View Controller y enlazaremos el Navigation Controller al Tab Bar, como hemos hecho con la pantalla anterior.

El esquema que nos quedará finalmente será este:

Estructura del storyboard con la nueva pantalla de Contacto

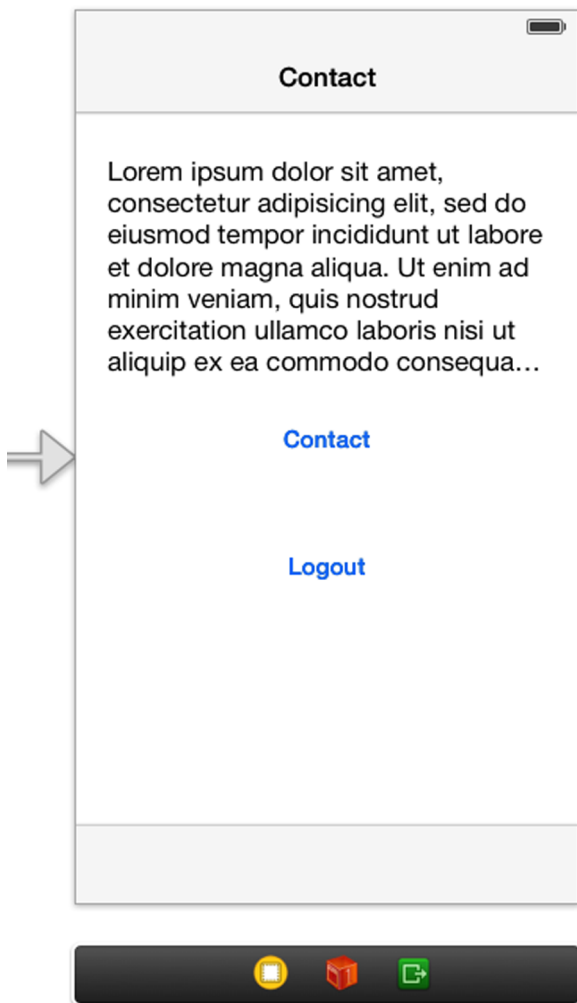


Dentro de este View Controller añadiremos un Label, aumentaremos su área y le escribiremos el texto que deseamos; por defecto un Label solo muestra una línea de texto; si queremos que muestre tantas como sean necesarias, le asignaremos un 0 al número de líneas.

Más abajo añadiremos un Button con el que enviaremos el correo electrónico y más abajo un segundo Button para poder hacer salir de la aplicación y volver a la pantalla inicial de acceso.

El resultado tendría que ser como el siguiente dibujo:

Pantalla de contacto con un texto opcional, botón de contactar y posibilidad de hacer logout de la aplicación



El siguiente paso será crear la clase controladora de esta pantalla; igual que hemos hecho anteriormente, crearemos una nueva subclase de UIViewController que se denomine ContactViewController; una vez creada, la asociaremos al View Controller que hemos creado en el storyboard; para hacerlo, le asignaremos el valor ContactViewController al atributo Class del mismo modo como lo hemos anteriormente.

A continuación, añadiremos una función que se llamará al tocar sobre el botón de contactar; para hacerlo activaremos de nuevo la Vista Assistant Editor y cargaremos en la parte derecha el archivo ContactViewController.h. Realizaremos la misma acción como si creáramos un outlet, es decir, arrastrando desde el botón hasta la ventana derecha, pero esta vez escogeremos como Connection "Action" para indicarle que quieren un IBAction; como event nos interesa la opción por defecto Touch Up Inside, le daremos de nombre contactAction y le daremos a Connect.

Nos tendría que quedar el archivo así:

```
@interface ContactViewController : UIViewController
```

```
- (IBAction) contactAction:(id) sender;

@end
```

Seguiremos implementando la opción de Contactar; para hacerlo dentro de la función que nos acaba de crear el Xcode, tendremos que colocar el siguiente fragmento de código.

```
- (IBAction) contactAction:(id) sender {

    MFMailComposeViewController *picker = [[MFMailComposeViewController alloc] init];
    picker.mailComposeDelegate=self;
    [picker setSubject:@"need help"];
    [picker setMessageBody:@"Please, I need help" isHTML:NO];
    [self presentViewController:picker animated:YES completion:nil];
}
```

Con las primeras versiones del iOS, para enviar un correo electrónico desde nuestra aplicación lo tendríamos que hacer abriendo la aplicación de Email y, por lo tanto, saliendo de nuestra aplicación. Con el iOS 3 apareció un View-Controller llamado MFMailComposeViewController, que nos permite redactar un email sin tener que salir de nuestra aplicación y que el iOS se encargue del envío.

Lo que hace este fragmento es primero crear una instancia de un MFMailComposeViewController.

La segunda instrucción establece la clase actual como delegate; esto nos servirá para saber qué acción ha escogido finalmente el usuario y esconder el View-Controller.

Las dos siguientes instrucciones establecen el Subject y el Body del email, a pesar de que el usuario lo puede cambiar si lo desea.

Y por último, la última instrucción presenta el ViewController del email de forma "Modal", o sea, estilo Popup.

Si ejecutamos la aplicación, veremos que nos aparecerá un error; una vez más tendremos que añadir un framework para hacer uso de una funcionalidad, en este caso el envío de emails.

Añadiremos el framework MessageUI.framework e importaremos las siguientes librerías en clase ContactViewController.

```
#import <MessageUI/MessageUI.h>
```

```
#import <MessageUI/MFMailComposeViewController.h>
```

Si probamos ahora la aplicación, veremos que ya se ejecuta correctamente, pero al querer cerrar la redacción del email, la pantalla no desaparece; para arreglarlo, implementaremos el delegate de MFMailComposeViewController.

Añadiremos el delegate:

```
@interface ContactViewController : UIViewController <MFMailComposeViewControllerDelegate>
```

E implementaremos el método que se llama al finalizar la redacción del email; en este método cerraremos la ventana modal.

```
-(void)mailComposeController:(MFMailComposeViewController *)controller didFinishWithResult:(MFMailComposeResult)result error:(NSError *)error{
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

A continuación haremos que, al pulsar el botón de Logout, volvamos a la pantalla de login; si hiciésemos un segue, lo que haríamos sería crear una nueva instancia; lo que queremos es volver a la instancia ya creada; esto se hace mediante un unwind segue. Para crear uno, tendremos que declarar una función al View Controller donde queremos volver, en este caso será nuestra clase ViewController del login.

En el ViewController.h añadiremos la declaración:

```
-(IBAction)returnActionForSegue:(UIStoryboardSegue *)returnSegue;
```

Y en el fichero .m la función:

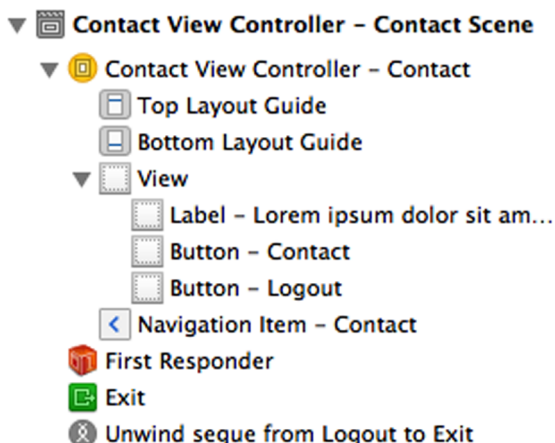
```
-(IBAction)returnActionForSegue:(UIStoryboardSegue *)returnSegue {
    // do useful actions here.
}
```

Esta función nos puede servir en otras ocasiones para pasar parámetros, en este caso solo necesitamos declararla para poder crear el unwind segue.

Ahora volveremos al storyboard y, con la tecla Ctrl activada, tenemos que trazar una línea desde el botón de Logout hasta el icono verde de Exit del ViewController de login. (Es posible que hayamos de mover los ViewControllers; si no, lo podemos hacer en el Document Outline).

En el Document Outline de Contact nos aparecerá un icono indicando el unwind segue creado; por defecto ejecutará la transición inversa.

Detalle donde vemos el elemento Unwind Segue que acabamos de crear



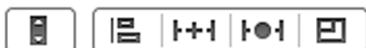
Si probamos la aplicación, veremos que ya funcionan todas las pantallas y podemos volver a la pantalla de login para empezar de nuevo; el siguiente paso pasaría por testear su funcionamiento.

Una de las primeras cosas que tenemos que contemplar es cómo se visualiza nuestra aplicación en la medida de pantalla antigua del iPhone, puesto que hemos desarrollado todo hasta ahora pensando en la medida alargada de cuatro pulgadas.

Para probarlo, utilizaremos el Auto Layout, un sistema que nos permite definir cómo se comportan los controles con diferentes pantallas.

Encontramos los iconos en la parte inferior derecha de la pantalla del storyboard.

Detalle con los elementos de Auto Layout



Con el primer icono podemos ver la visualización con las diferentes medidas de pantalla de forma rápida; de este modo, podemos detectar rápidamente si tenemos que modificar las propiedades de Auto Layout de algún elemento.

En el caso de nuestra aplicación, parece que no hay ningún problema de ajuste; si fuera necesario tendríamos que utilizar las herramientas de Auto Layout y definir las restricciones necesarias para cada objeto que necesitemos ajustar.

2.3.6. Retoques

A continuación, acabaremos de ajustar algunos elementos, como por ejemplo definir el icono de la aplicación, la pantalla inicial, etc.

De cara a diseñar el icono, tenemos que tener en cuenta hacerla de un tamaño suficientemente grande, ya que ahora mismo, al subir una aplicación a la App Store, ya se pide el icono a una medida de 1024x1024 píxeles.

Y respecto a la pantalla inicial en iOS, podemos definir una imagen inicial que se mostrará mientras se carga nuestra aplicación; suele verse poco tiempo, pero es la manera que utiliza iOS para mostrar al usuario que se está cargando la aplicación. Esta pantalla es una imagen PNG y no puede animarse; las pantallas animadas o vídeos que muestran algunas aplicaciones se cargan *a posteriori* de esta imagen inicial.

Tanto para el icono como para la pantalla de inicio, necesitaremos generar diferentes medidas para las diferentes medidas de pantalla o versiones de iOS.

Empezaremos por el icono; con el Xcode 5 tenemos una nueva herramienta para gestionar los assets o materiales gráficos como iconos y pantallas de inicio.

Si seleccionamos el icono de nuestro proyecto en la pestaña General, encontraremos la opción App Icons, que nos permite escoger si queremos utilizar assets o el sistema tradicional. Confirmaremos que tenemos la opción AppIcon escogida; AppIcon es el nombre del asset para los iconos.

Si abrimos el archivo Images.xcassets, veremos dos assets ya definidos: uno llamado AppIcon y el otro LaunchImage; cada uno de ellos nos informa de las imágenes necesarias que necesitamos para definir tanto el icono como la splash screen.

Lo que tendremos que hacer es generar las diferentes imágenes con algún programa de edición de imágenes y guardarlas en la carpeta de assets correspondiente: Images.xcassets/AppIcon.appiconset, para los iconos, e Images.xcassets/LaunchImage.launchimage para la pantalla de inicio. Una vez las tengamos todas guardadas desde la pantalla de assets, podemos asignar a cada imagen cuál es. Hay que tener en cuenta que las imágenes en las que pone 2x quiere decir que son imágenes retina, por lo que tienen que estar al doble del tamaño que indica; es habitual que el nombre de estas acabe con @2x para indicar que son retina.

Para el AppIcon deberíamos tener un resultado como este:

Detalle con todos los assets de los iconos asignados

AppIcon



Para la pantalla inicial haremos el mismo procedimiento: la medida para pantallas de 4" es de 640x1136, y la de pantallas tradicionales retina de 640x960.

Detalle con los assets de las pantallas iniciales asignadas



Por último, haremos otro cambio gráfico: con el iOS 7 podemos modificar el color con que se muestra la status bar, que es la barra superior donde aparece la cobertura, la hora y la batería; en las versiones anteriores del iOS 7, era un espacio separado en el espacio de la aplicación; con el iOS 7 forma parte de la aplicación, por lo que, dependiendo del color de nuestros elementos, puede quedar escondida. Es el caso de la pantalla de inicio y la de login, que al ser de fondo azul y la status bar negra no se acaba de ver correctamente, lo que haremos será que en estas dos pantallas se muestre la status bar con color blanco.

Para empezar, tenemos que añadir un parámetro al fichero Info.plist para indicar que queremos que la status bar aparezca inicialmente blanca; la manera más sencilla de hacerlo es seleccionar nuestro proyecto, seleccionar nuestro target y, en la pestaña de Info, veremos todas las entradas del fichero Info.plist. Daremos al botón derecho sobre cualquiera de ellas y seleccionaremos la opción Add row; dentro de la nueva fila creada, añadiremos el parámetro `UIStatusBarStyle` con el valor `UIStatusBarStyleLightContent`. Con esto, conseguiremos que se muestre en la pantalla de inicio la status bar en blanco, pero no en la de login.

Tendremos que añadir otro parámetro al Info.plist llamado `UIViewControllerBasedStatusBarAppearance`; este parámetro nos permite definir la apariencia de la status bar en cada ViewController, a través de un método. Así que añadiremos este parámetro con el valor de YES, y en el ViewController.m añadiremos el siguiente método:

```
- (UIStatusBarStyle)preferredStatusBarStyle
```

```
{  
    return UIStatusBarStyleLightContent;  
}
```

Con este fragmento de código ya habremos conseguido nuestro objetivo, puesto que, para la pantalla de login, forzaremos la status bar en blanco y para el resto seguirá apareciendo el estilo negro por defecto.

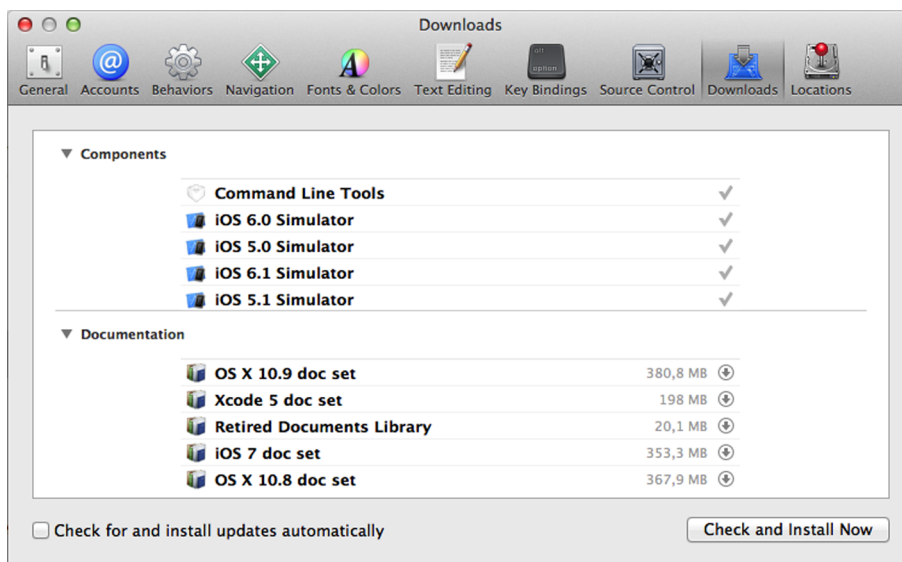
Con este último detalle ya daremos el desarrollo de la aplicación por acabado. El siguiente paso será el testeo.

3. Testeo de aplicaciones en iOS

Respecto al testeo de aplicaciones desarrolladas por el iOS, lo más importante es utilizar, siempre que sea posible, el simulador mientras dure el desarrollo, pues es más fácil y rápido que probarlo en un dispositivo, y además, no es necesario estar dado de alta como desarrollador.

Es indispensable probar siempre la aplicación con todas las versiones con las que estará disponible; esto quiere decir ejecutarla con el simulador con el SDK con el que se compilará, y también probarla con las versiones de deployment; para hacerlo, tendremos que tener instaladas todas las versiones del iOS posibles en nuestro simulador. Podemos descargar nuevas versiones en el menú Xcode>Preferences en el apartado Downloads.

Detalle de las preferencias de Xcode con los paquetes descargados



Una vez hayamos probado y solucionado los problemas detectados en el simulador, entonces haremos las pruebas en el máximo de dispositivos físicos que tengamos disponibles. El simulador del iOS es bastante fiable, y es extraño que el funcionamiento en el dispositivo sea diferente en el simulador.

La única diferencia importante entre el simulador y el dispositivo es que el simulador es case insensitive, y el dispositivo es case sensitive; esto quiere decir que para el simulador no hay diferencia entre una mayúscula y una minúscula, y con el dispositivo sí que la hay, por lo que tendremos que ir especialmente con cuidado cuando trabajemos con nombres de imágenes, de ficheros, etc.

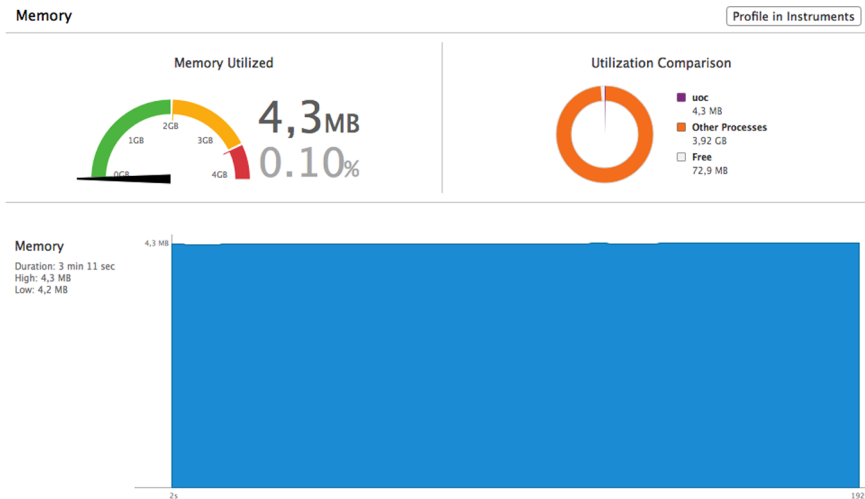
El uso del simulador no será posible en todos los proyectos ya que hay ciertas cosas que solo podremos testear en el dispositivo, como aplicaciones donde se utilice la cámara, acelerómetro, etc.

A continuación, analizaremos algunas de las cosas que podemos utilizar de cara al testeo de nuestras aplicaciones.

Debug Navigator/Instruments

Con el Xcode 5 podemos visualizar mientras ejecutamos la aplicación la cantidad de CPU y memoria utilizados; al ejecutar la aplicación, si hacemos Cmd +6, iremos a la pestaña Debug Navigator; aquí nos mostrará la información básica de los recursos utilizados.

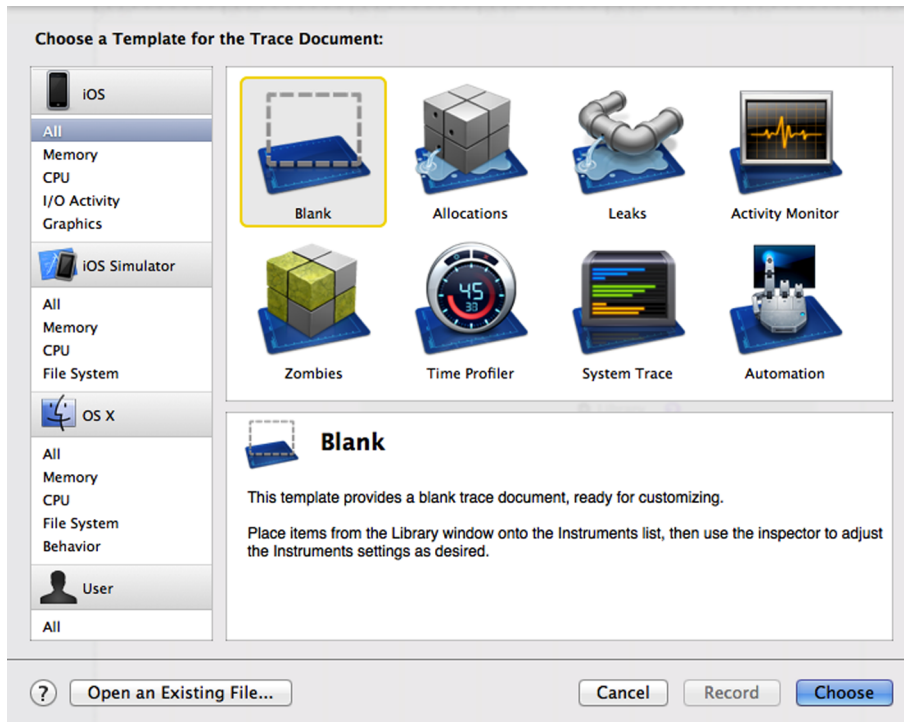
Detalle del Debug Navigator del Xcode 5



Si queremos información más específica u optimizar nuestra aplicación, entonces tenemos la opción de ejecutar Instruments.

Este es un software muy completo que nos permite analizar el comportamiento de una aplicación mientras se está ejecutando; en el nivel más bajo, podemos ver el funcionamiento de la memoria en tiempo real y saber qué se está ejecutando en cada momento, podemos detectar leaks o memoria que no utilizamos correctamente, podemos monitorizar la actividad del disco, CPU, red, etc.

Ventana con algunos ejemplos de lo que podemos hacer con la herramienta Instruments



Para hacer funcionar Instruments, podemos utilizar el acceso desde la pantalla de Debug Navigator, o con la opción de Menú Product > Profile o Cmd +I.

Logging

Para hacer un buen testeo es muy importante disponer de un buen logging o registro de events, ya que es una de las mejores maneras de saber qué está pasando o qué ha pasado.

Habitualmente, cuando hablamos de logging son registros enviados a la consola para saber qué está pasando en nuestro código; normalmente se utiliza la función por defecto NSLog.

Una instrucción muy habitual de NSLog es:

```
NSLog(@"%s", __PRETTY_FUNCTION__);
```

Esta instrucción es muy común verla a comienzo de muchas funciones, pues manda un mensaje a la consola con el nombre de la función; esto nos puede servir para saber en qué momento ha fallado nuestra aplicación comprobando los últimos mensajes enviados a la consola.

Enlace recomendado

Si queremos profundizar más en el funcionamiento de Instruments, existe un muy buen tutorial en:

<http://www.raywenderlich.com/es/31024/como-utilizar-instruments-en-xcode>

Sobre el registro de events, hay que destacar también una librería muy útil llamada CocoaLumberjack, la cual nos da muchísimas más opciones que NSLog; nos permite definir diferentes niveles de events, por lo que podemos configurar recibir solo los de un determinado tipo; también podemos guardar fácilmente este registro en un fichero o enviarlo a un servidor, etc.

Podéis encontrarla aquí:

<https://github.com/robbiehanson/cocoalumberjack>

Crash Reports

Con el iOS también existen los llamados Crash Reports; estos son volcados de memoria en el momento en que una aplicación ha tenido un error fatal y se ha tenido que cerrar; analizando estos ficheros podemos intentar averiguar qué es lo que ha podido fallar.

Podemos descargarnos Crash Reports de nuestros usuarios desde nuestra cuenta de iTunes Connect, para poder detectar errores que nos han pasado por alto. Actualmente, el mismo iTunes Connect se encarga de darnos la información más importante, es decir, si es posible nos indicará el tipo de error y la función que se ha producido.

Podemos descargarnos de todas maneras estos archivos y analizarlos más detenidamente si es necesario; para hacerlo, es importante siempre guardar el binario de cada versión que subimos al iTunes y el fichero .dSYM generado, pues necesitamos exactamente el mismo binario y fichero dSym, para traducir el fichero de crash y saber qué función ha generado el crash. La traducción de valores de memoria a nombre de funciones es lo que se denomina Symbolication.

TestFlight

Finalmente, hablaremos de un servicio muy útil llamado TestFlight, un servicio de terceros pero que es bastante conocido y utilizado por los desarrolladores iOS; empezó siendo una plataforma para distribuir de forma fácil las versiones beta entre las personas que deseamos a través de la distribución AdHoc. Recordemos que este tipo de distribución nos permite distribuir nuestra aplicación a los clientes o usuarios hasta un límite de 10 dispositivos por año; es necesario, por eso, disponer del Developer Program.

TestFlight nos facilita enormemente esta tarea a la vez que nos integra una serie de servicios que nos pueden interesar, como la posibilidad de informarnos de las excepciones, symbolication de crashreports de los usuarios de nuestras

aplicaciones beta, la posibilidad de hacer logs remotos, permitir la instalación de nuevas versiones de beta de forma transparente, la posibilidad de recibir feedback de los testers, etc.

Y hasta ahora todos los servicios son gratuitos.

Enlace recomendado

Podés encontrar más información en:
<http://testflightapp.com/>

4. Depuración de aplicaciones en iOS

La mejor manera para depurar nuestro código será primero utilizar una buena herramienta de logging, como el CocoaLumberjack comentado anteriormente, y después aprender a hacer uso del depurador del Xcode, que como buen IDE también dispone de un depurador integrado.

El depurador del Xcode permite tanto depurar las aplicaciones utilizando el simulador como depurar una aplicación en nuestro dispositivo; tendremos que tener conectado el dispositivo al ordenador a través de un cable USB.

Para activar el depurador, podemos hacerlo a través del menú Debug > Activate Breakpoints, con la combinación de teclas Cmd + Y, o utilizando el segundo icono por la derecha, en las opciones del Area Debug, situadas encima de la consola, para activar o desactivar los breakpoints o puntos de depuración.

Detalle de la barra de depuración del Xcode



Esto nos activa el depurador, pero tendremos que definir los puntos que queremos estudiar dentro de nuestro código. Para hacerlo, haremos clic sobre los números de línea que nos interesen. Nos aparecerá una flecha azul indicando que en esa posición el depurador se parará.

Captura donde vemos un breakpoint o punto de depuración en nuestro código

```
37 - (BOOL)textFieldShouldReturn:(UITextField *)textField{
38
39
40     NSLog(@"%s", __PRETTY_FUNCTION__);
41     [textField resignFirstResponder];
42     return YES;
43 }
```

Si volvemos a hacer clic encima, el punto quedará desactivado; para eliminarlo del todo, tendremos que apretar el botón derecho encima y escoger la opción Delete breakpoint.

Para depurar nuestro código, tendremos que activar el depurador y ejecutar la aplicación en el simulador o en el dispositivo; podemos activar el depurador en cualquier momento de la ejecución.

Podemos añadir tantos puntos de depuración como necesitemos; los podremos visualizar y modificar todos desde un mismo lugar en la pantalla Break-Point Navigator Cmd + 7.

Si activamos el depurador marcando un punto donde sepamos seguro que pasará el hilo del programa y le damos a ejecutar, el programa se ejecutará y, al llegar a ese punto, se parará. En este punto, se nos mostrará con una línea resaltada de color verde el punto de ejecución actual del programa.

Captura donde vemos el programa ejecutando una instrucción determinada

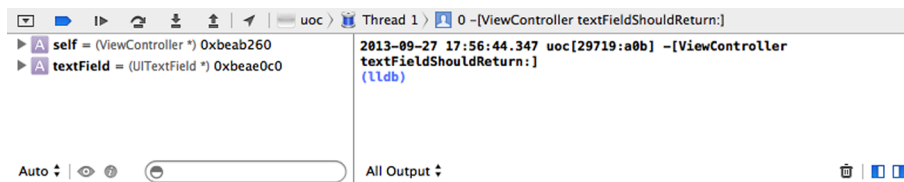
```

38 - (BOOL)textFieldShouldReturn:(UITextField *)textField{
39
40     NSLog(@"%s", __PRETTY_FUNCTION__);
41     [textField resignFirstResponder];
42     return YES;
43 }
44

```

El Area de Debug se compone de la barra con las opciones de Debug en la parte superior, una ventana en la parte izquierda con la zona de Variables y en la parte derecha la Consola.

Captura donde vemos la barra de Debug a la izquierda y la Consola a la derecha



En la barra con las herramientas de Debug encontramos una serie de iconos; los comentaremos de izquierda a derecha.



El primer icono simplemente nos sirve para mostrar o esconder las herramientas de Debug.



El segundo icono es el que activa el depurador o lo desactiva; cuando el icono aparece azul es que el depurador está activado.



El tercer icono nos sirve para saltar al punto de depuración actual y continuar la ejecución del programa.



El cuarto icono “Step Over” nos permite saltar a la siguiente instrucción dentro de la misma función.



El siguiente icono “Step Into” nos permite acceder a la siguiente instrucción, que se ejecutará dentro de la instrucción actual. Baja un nivel.



El último icono de las opciones de Debug es “Step Out”, con el que podemos subir o volver atrás un nivel.

Explicada la barra de Debug, comentaremos la zona de Variables; aquí encontramos las variables que Xcode piensa que nos pueden interesar; podemos escoger también visualizar solo las variables locales o todas las variables incluyendo registros globales, etc. Desplegando estas variables, podemos visualizar el valor de todas sus propiedades.

Si escogemos una variable y hacemos clic en el pequeño icono con una y en la parte baja de Variables, se imprimirá en la consola un resumen de sus propiedades, lo que se denomina description del objeto. También podemos conseguir el mismo resultado ejecutando la instrucción `po <variable>` en la consola, `po` o `print object`; nos sirve para mostrar la información importante de un objeto de una manera rápida.

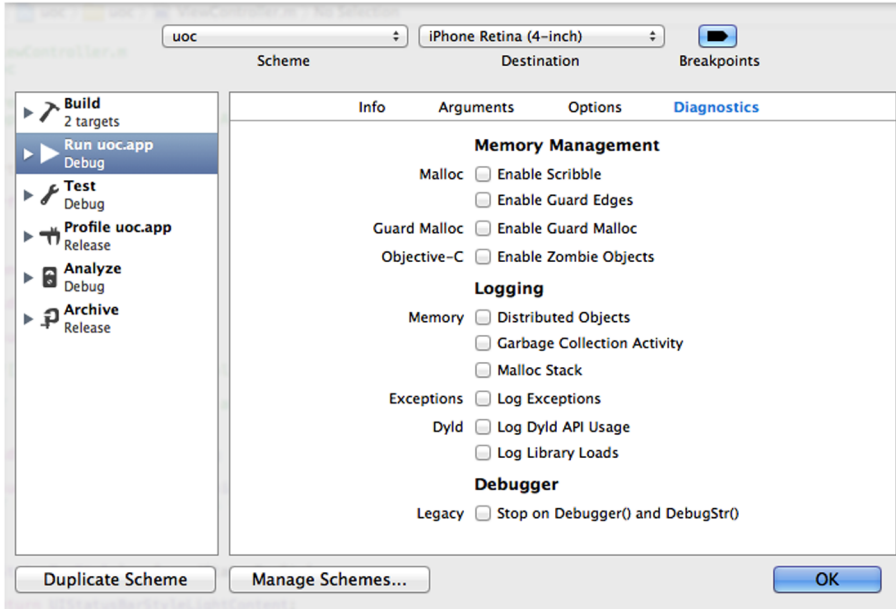
Para primitivas como `int`, `float`, etc., podemos utilizar las instrucciones `po` o `print`.

Con esto ya estaremos en cuanto al uso del depurador de Xcode, pero veamos algunas características más que nos pueden servir para depurar nuestras aplicaciones.

Xcode dispone de lo que se conoce como NSZombies; sirve para depurar problemas con objetos en memoria; en el funcionamiento normal, cuando un objeto deja de estar instanciado, este se libera de memoria y desaparece; a veces podemos encontrarnos errores donde un objeto nos ha desaparecido, pero Xcode no es capaz de indicar de qué objeto se trata. En estos casos podemos utilizar los NSZombies; con los NSZombies habilitados, las variables, al dejarse de utilizar, no desaparecen sino que se transforman en un NSZombie, con lo cual, cuando accedemos, la consola nos informará del objeto que está provocando el problema.

Podemos activar esta opción en las opciones de nuestro Scheme o esquema de ejecución; podemos encontrarlo haciendo clic en el nombre de nuestro proyecto junto al botón Play y Stop; seleccionamos Edit Scheme a Diagnostics, encontraremos la siguiente pantalla con la opción de Enable Zombie Objects.

Ventana de configuración de los schemes para configurar opciones avanzadas de depuración y ejecución



Es importante acordarnos de desactivar esta opción cuando no hagamos depuración.

Y por último comentaremos la depuración de excepciones; nuestro código, en un momento, seguramente generará alguna excepción; muchas veces Xcode nos mostrará un mensaje con el tipo de excepción, pero no nos sabrá indicar la línea donde se ha producido la excepción. Para evitar esto, podemos indicar que queremos un punto de depuración en el punto del código que ha producido la excepción. Para activarlo, en la pantalla Breakpoint Navigator Cmd +7, en la parte inferior veremos un símbolo + y - para añadir o eliminar puntos de depuración; si le damos al símbolo + podemos seleccionar la opción Add Exception Breakpoint con la cual, cuando depuremos si se produce una excepción, nos indicará exactamente el punto exacto.

