Guías de estilo y personalizaciones avanzadas

Índice

En esta sesión hablaremos de los patrones de diseño que *Apple* nos recomienda seguir para nuestras aplicaciones. En muchos casos estos patrones son obligatorios cumplirlos a la hora de diseñar nuestras aplicaciones y el hecho de no cumplirlos puede ser motivo de rechazo a la hora de querer publicar en la *App Store*. Comentaremos las distintas características de cada plataforma iOS. Por último detallaremos distintas técnicas que existen para personalizar los controladores y vistas más usados y, de esta forma, conseguir un aspecto más atractivo para el usuario final. ¡Comenzamos!

1. Guías de estilo en iOS

Tanto el iPhone como el iPad han supuesto un revolucionario giro en cuanto a diseño de interfaz se refiere. Los usuarios de este tipo de dispositivos pueden realizar múltiple tareas como navegar por internet, leer y escribir correos, realizar fotos y videos, navegar por las distintas aplicaciones, etc. Para evitar el "caos" a la hora de diseñar cualquier tipo de aplicación, en *Apple* han planteado una serie de guías de estilo que todo diseñador debe seguir en la medida de lo posible a la hora de diseñar una aplicación. Estas guias de estilo no son válidas, como es lógico, a la hora de diseñar la interfaz de un juego, sólo son útiles en aplicaciones.

En este módulo vamos a aprender a diseñar aplicaciones de iPhone / iPad que sean usables, accesibles y en las que el usuario perciba una grata experiencia al hacer uso de ella. Hay que tener en cuenta que un usuario de iPhone está habituado a una serie de elementos y situaciones básicas como la navegación dentro de vistas mediante *Navigation Controllers*, navegación por las opciones fundamentales mediante los *Tab Bar Controllers*, uso de botones claros, etc. Aquí haremos un resumen de todo lo que Apple propone en sus guias de estilo.

Vamos a dividir el módulo en 3 puntos claramente diferenciados:

- Características principales de la plataforma iOS
- Estrategias de diseño de aplicaciones
- Guías de uso de las principales tecnologías disponibles en iOS

1.1. La pantalla

Todo usuario que usa una aplicación accede mediante la pantalla, es por ello que este es el componente más importante que podemos encontrar en cualquier dispositivo iOS. Existen tres distintos tamaños de pantalla según cada dispositivo, estos son:

- *iPhone 4 (Retina Display)*: 640 x 960 pixels
- *iPad*: 768 x 1024 pixels
- *iPhone 3G, 3GS o iPod Touch*: 320 x 480 pixels

Detección de contacto (touch events)
El tamaño de la zona mínima para que el evento de contacto funcione correctamente debe de ser

de al menos 44 x 44 puntos.

1.2. Orientación del dispositivo

Uno de los requesitos a la hora de publicar una aplicación de iPhone/iPad en *App Store* es la compatibilidad con las distintas posiciones que puede adoptar el dispositivo iOS, estas posiciones son vertical (*portrait*) y horizontal (*landscape*). Cualquier aplicación debe de estar adaptada a ambas posiciones y esto se hace para mejorar la usabilidad y la comodidad a la hora de utilizar la aplicación. La programación de la orientación en una aplicación iOS es relativamente sencilla, excepto casos puntuales y no implementarla puede suponer, como hemos comentado, el rechazo a la hora de publicar en *App Store*.

Según el dispositivo que dispongamos, la pantalla inicial (*Home Screen*) acepta una orientación vertical, horizontal o ambas. Por ejemplo, un iPhone o iPod Touch muestra su pantalla de inicio sólo en vertical, mientras que en el iPad se puede mostrar en vertical y en horizontal.

1.3. Los multigestos

Un punto que hay que tener muy presente a la hora de diseñar y programar una aplicación en cualquier dispositivo iOS es el uso de los gestos, llamamos "gestos" a los distintos movimientos que hace el usuario sobre la pantalla para realizar distintas acciones. Cuantos más gestos se implementen en una aplicación, más agradable e intutiva será de usar para el usuario final. Los distintos gestos que una aplicación puede adoptar son los siguientes:

- Toque (*tap*): Consiste en presionar o hacer "click" sobre un botón o cualquier objeto que esté en pantalla.
- Arrastre (*drag*): Mover el pulgar sobre la pantalla en una dirección, puede usarse para navegar sobre los elementos de una tabla, por ejemplo.
- Arrastre rápido (*flick*): Como el anterior, pero más rápido. Sirve para moverse por la pantalla de forma rápida.
- Arrastre lateral (*Swipe*): Mover el pulgar en dirección horizontal, sirve para mostrar el botón de "Eliminar" en una fila de una tabla.
- Doble toque (double tap): Presionar dos veces seguidas y de forma rápida la pantalla. Sirve para aumentar una imagen o un mapa, por ejemplo.
- Pellizco exterior (*pinch open*): Gesto de pellizco sobre la pantalla que sirve para aumentar una imagen o un mapa.
- Pellizco interior (*pinch close*): El inverso del anterior.
- Toque continuo (*touch and hold*): Como el toque básico, pero manteniendo el pulgar sobre la pantalla. Sirve para mostrar un menu contextual sobre la zona que se pulsa.

1.4. La multitarea

La multitarea es una característica que está disponible a partir de iOS 4. Mediante esta nuestro dispositivo puede ejecutar más de dos aplicaciones al mismo tiempo: siempre estará una ejecutándose en pantalla, mientras que el resto estarán almacenadas en memoria, en *background*. *Apple* nos recomienda que nuestra aplicación tenga dicha característica ya que de esta forma el usuario puede estar realizando otras tareas al mismo tiempos sin necesidad de cerrarla.

Captura multitarea iOS

La gestión e implementación de la multitarea es muy sencilla, se realizará desde la clase delegada, en los métodos `applicationDidEnterBackground` y `applicationWillEnterForeground`. Esto se explicará en las sesiones correspondientes.

1.5. Preferencias y ayuda

Si la aplicación utiliza preferencias propias, estas estas deben de estar en la medida de lo posible en el panel de preferencias que se encuentra en el menu de general de configuración de la aplicación de preferencias del dispositivo.

Por otro lado, la el texto de ayuda, si existe, debe de ser lo más claro y compacto posible, si se pueden utilizar imágenes mejor. Hay que tener en cuenta que el usuario no tendrá tiempo ni ganas de estar leyendo la ayuda antes de utilizar tu aplicación, además el

espacio que ocupa la ayuda se puede emplear para otros contenidos que sean de mayor importancia. La ayuda no debería ser necesaria se se utilizan las guias de estilo establecidas por *Apple* y que tienen por finalidad, como hemos comentado anteriormente, establecer una interfaz simple e intuitiva de usar.

1.6. Estrategias de diseño: Documento descriptivo de la aplicación

Existen distintas estrategias de diseño de aplicaciones. *Apple* nos recomienda redactar un pequeño documento que describa la aplicación de la manera más clara y concisa posibla siguiendo cuatro pasos simples:

- 1) Listar todas las tareas que debe implementar la aplicación sin importar que esta lista sea grande, después puede reducirse.
- 2) Determinar quiénes son los usuarios que van a usar la aplicación.
- 3) Filtrar el listado de tareas del primer punto según los tipos de usuario definidos en el segundo punto.
- 4) Usar el listado de tareas final para definir el tipo de interfaz gráfica a utilizar, los controles y terminología, etc.

1.7. Estrategias de diseño: Diseño según el dispositivo

La aplicación que diseñes tiene que estar totalmente adaptada a un dispositivo iOS, no a otro tipo ni a Web, el usuario lo agradecerá. La inmensa mayoria de usuarios de iOS están acostumbrados al uso de botones, barras de navegación, *Tab Bars*, etc. Debemos de, en la medida de lo posible, hacer uso de toda esta serie de componentes que son diseñados de forma específica para dispositivos iOS y con los que los usuarios están muy familiarizados.

Otro tema importante es que si la aplicación es universal, esta debe de funcionar correctamente tanto en iPhone como en iPad. Hay que asegurarse antes de publicarla en *App Store* de esto ya que si se llega a publicar y falla en alguno de estos dispositivos, el usuario lo tendrá en cuenta a la hora de puntuarla o comprar otras del mismo desarrollador. Existen una serie de puntos a tener en cuenta en este caso:

- Hay que "moldear" cada aplicación según el dispositivo en el que se use teniendo en cuenta especialmente las capas de vistas, ya que estas cambian bastante.
- Se debe de adaptar todo el arte al dispositivo adecuado. El iPad y el iPhone 4 Retina diplay tendrá un arte con mayor resolución que un iPhone 4, 3G, 3GS y iPod Touch.
- Las características de la aplicación deben de conservarse a pesar del tipo de dispositivo que se use.
- Intentar, al menos, diseñar la aplicación universal y no únicamente para iPhone y que por defecto aparezca aumentada 2x en iPad. El usuario lo agracederá.

1.8. Guías de uso de las principales tecnologías disponibles en iOS

La API de iOS nos da acceso a múltiples tecnologías nativas que nos permiten realizar distintas tareas dignas de destacar y que posiblemente sean de utilidad a la hora de usar la aplicación que desarrollemos. A continuación se comentan las principales:

1.8.1. Multitarea

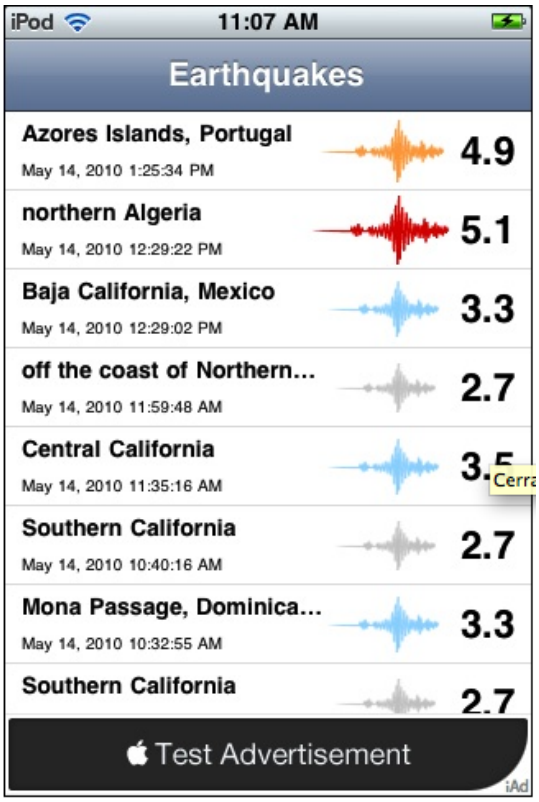
La multitarea aparece a partir de la versión de iOS 4, es muy importante tener en cuenta esta característica a la hora de desarrollar una aplicación ya que el usuario en cualquier momento puede cambiar entre ellas y si no está contemplado puede llegar a producir errores o inconsistencias. La aplicación debe de estar preparada para gestionar interrupciones de audio en cualquier momento, pararse y reiniciarse sin ninguna complicación ni lag y de forma "suave", por último debe comportarse de forma adecuada cuando se encuentra en *background*. La multitarea es una tecnología que se utiliza muy a menudo a la hora de usar apliaciones iOS y es por ello que debemos tenerla muy presente a la hora de programar nuestras aplicaciones. La gestión del paso de un estado *activo* a *inactivo* se suele programar dentro de la clase *Delegada*.

1.8.2. Imprimir

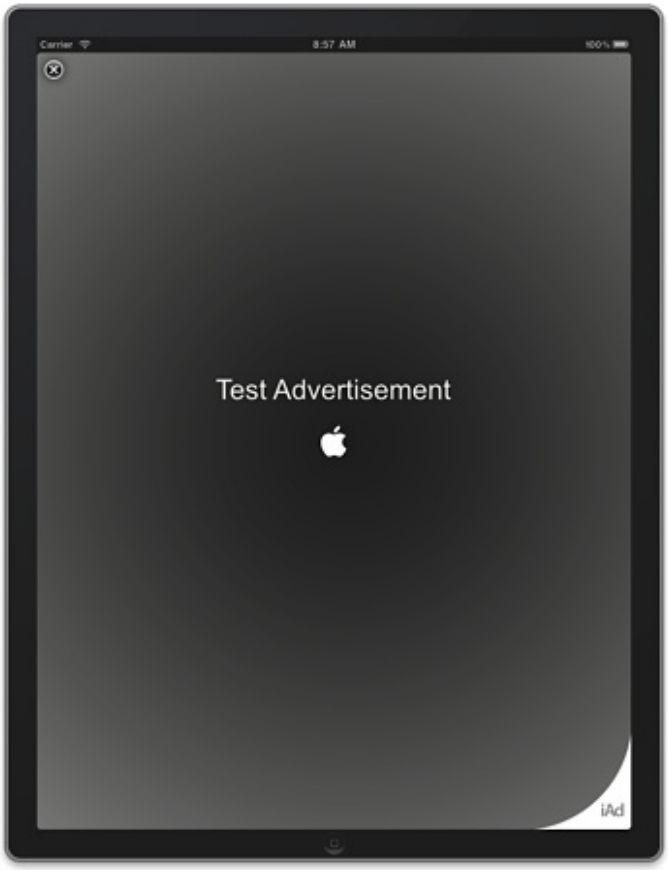
A partir de la versión iOS 4.2 aparece la tecnología de imprimir. Mediante esta ya se puede enviar a imprimir cualquier documento que deseemos desde nuestras aplicaciones. La API de iOS se encarga de gestionar la localización de impresoras y la ejecución de tareas de impresión, el desarrollador se encarga de especificar qué elemento/s desea imprimir (normalmente PDF o imágenes) y gestionar el botón de *imprimir*.

1.8.3. iAD

A partir de iOS 4.0 se pueden mostrar banners de publicidad gestionados por *Apple* en nuestras aplicaciones. Existen varias formas y tamaño de mostrar estos banners dependiendo del dispositivo que utilicemos. El sistema iAD es un método a tener en cuenta para obtener beneficios extra y se utiliza bastante a menudo. La publicidad de iAD no es intrusiva, cuando un usuario pulsa sobre uno de estos banners, la aplicación se debe de *pausar* y la publicidad se abrirá en una ventana dentro de la aplicación. Cuando el usuario cierra esta ventana, la aplicación que se estaba usando seguirá desde el mismo punto donde se quedó. Esta característica le distingue enormemente de otros sistemas de publicidad disponibles en iOS como el famoso *AdMob* de *Google*. A continuación muestro una serie de ejemplos de este tipo de publicidad:



Ejemplo iAD



Ejemplo iAD pantalla completa iPad

1.8.4. Vista rápida de documentos

Desde la versión 4 de iOS los usuarios pueden acceder a una vista prévia de documentos descargados desde las aplicaciones. Dependiendo del dispositivo a utilizar, los documentos se presentarán dentro de una ventana modal si estamos hablando de un iPad o dentro de una vista completa si es un iPhone/iPod Touch.

1.8.5. Sonidos

Puede que como desarrollador estés interesado en reproducir sonidos dentro de tú aplicación, esto puede producir un efecto agradabe en el usuario aunque hay que tener en cuenta una serie de puntos para evitar justamente lo contrario. Si el usuario activa el modo *silencio* en el dispositivo, este no debe de emitir ningún tipo de sonido, por lo tanto esto es algo a tener muy en cuenta cuando se desarrolle una aplicación con sonido. En principio no debe haber problemas, pero conviene comprobarlo siempre. En la medida de lo posible, dejar al usuario la opción de ajustar los volúmenes de sonido que desee. El tipo de sonido a escoger a la hora de desarrollar las distintas características de la

aplicación es muy importante, estos sonidos deben de ser los más adecuados y se deben de ajustar al tipo de acción que se esté desarrollando. Otro punto a tener en cuenta es la gestión de las interrupciones de audio cuando, por ejemplo, se produce una llamada entrante o la aplicación pasa a estado de inactiva (*background*), etc. Si estamos hablando de una aplicación musical podemos tener en cuenta, para mejorar la experiencia de usuario, el uso de control remoto de *Apple* o incluso el nuevo sistema de *AirPlay*.

1.8.6. Accesibilidad

El sistema de control por voz *VoiceOver* está diseñado para aumentar la accesibilidad a personas con discapacidades visuales o incluso para usuarios con interés en el aprendizaje de una lengua extranjera.

1.8.7. Menú de edición

Existe un menú contextual de edición básico que incorpora iOS por defecto, este contiene las opciones de "copiar", "cortar" y "seleccionar". Este menú los desarrolladores podemos modificarlo a nuestro antojo eliminando o incluyendo nuevas opciones que se ajusten más a nuestra aplicación.

1.8.8. Teclado

El teclado virtual es totalmente modificable según nuestros requerimientos, por ejemplo, si queremos utilizar un teclado númerico sólo debermos indicarlo al cargar el teclado dentro del método adecuado.

1.8.9. Servicios de localización

Los servicios de localización permiten a los usuarios acceder a una situación geográfica en cualquier momento y lugar. Si nuestra aplicación hace uso de los mapas, por ejemplo, estos intentarán a su vez hacer uso de los servicios de localización del dispositivo para poder situar de una forma bastante certera la posición actual del usuario en el mapa. Al intentar hacer uso de estos servicios, la aplicación lanzará una ventana emergente (*AlertView*) preguntando por los permisos de acceso. Los sistemas de localización están centrados en la brújula, en el GPS y wifi.

1.8.10. Notificaciones Push

Las notificaciones push permiten a los usuarios estar avisados en cualquier momento, independientemente si la aplicación está funcionando en ese mismo instante. Esto es muy útil en aplicaciones que hagan uso de un calendario, para avisar de futuros eventos, o aplicaciones con algún tipo de recordatorio, etc. También se utiliza muy a menudo en juegos, sobre todo en juegos sociales. La implementación de este servicio por parte del desarrollador es algo compleja y requiere de un servidor propio, aunque se pueden

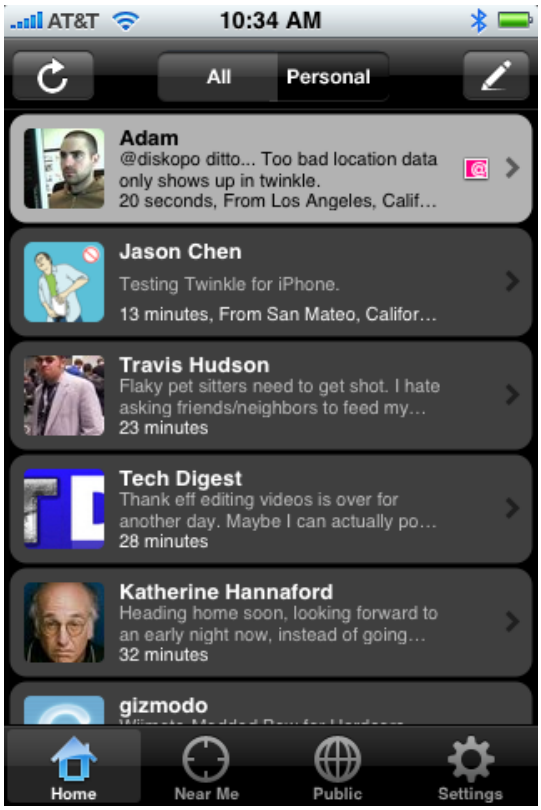
contratar servicios externos que realicen esa tarea de forma bastante económica. Al mismo tiempo que se activa la notificación push se muestra un número en la esquina superior derecha del icono de la aplicación, a esto se le llama *badge* y es muy sencillo de implementar en nuestras aplicaciones.

Notificación push

Ejemplo badge en app

2. Personalización avanzada: celdas

La captura siguiente muestra la pantalla principal del cliente de Twitter para iPhone *Twinkle* y, aunque no lo parezca, es simplemente una tabla con las celdas totalmente personalizadas.



Ejemplo personalización Twinkle

La personalización de las celdas de una tabla es algo muy habitual en la programación de aplicaciones iOS, con ello podremos hacer que nuestras tablas se distingan del resto usadas comunmente así como ofrecer un "toque" característico y concordante con nuestras aplicaciones. Pero, ¿cómo podemos hacer este tipo de celdas en nuestra aplicación iOS? Muy simple, a continuación veremos mediante un ejemplo paso a paso el diseño y programación de celdas personalizadas mediante el *Interface Builder* de XCode.

2.1. Creando el proyecto y las clases básicas

Al terminar este ejemplo tendremos una aplicación formada únicamente una vista de tabla *UITableView* con celdas personalizadas. Estas celdas tendrán una imagen en el lado izquierdo, un texto en negrita en la parte superior, un texto normal en la parte central y otro texto pequeño en la parte inferior. Las celdas tendrán un tamaño algo mayor al que viene por defecto y, para finalizar, la tabla tendrá estilo zebra, esto es que el fondo de las celdas se intercalará de color.

Comenzaremos creando el proyecto, para ello abrimos XCode y creamos un nuevo proyecto de tipo *Navigation-based Application* sólo para iPhone. Lo guardamos con el nombre *uisesion02-ejemplo1*. Con esto ya tenemos el esquema básico del proyecto: una vista de tabla contenida en un controlador de navegación. Ahora creamos una nueva vista

subclase de *Table View Cell*, para ello hacemos click en *New > New File*, seleccionamos *UIViewController Subclass* y Subclass of *UITableViewCell* **desmarcando** "With XIB for user interface" y "Targeted for iPad". La guardamos como *TableViewCell*.

2.2. Diseñando la celda desde Interface Builder

Una vez que hemos creado el proyecto y la clase de la celda ahora vamos a diseñarla, para ello creamos un nuevo archivo *New > New File* de tipo *User Interface* y seleccionamos la plantilla *Empty*. Guardamos el fichero con el nombre *TableViewCell*, lo abrimos y arrastramos desde la libreria de objetos un *Table View Cell*:

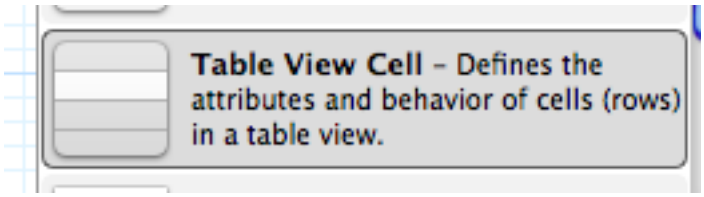
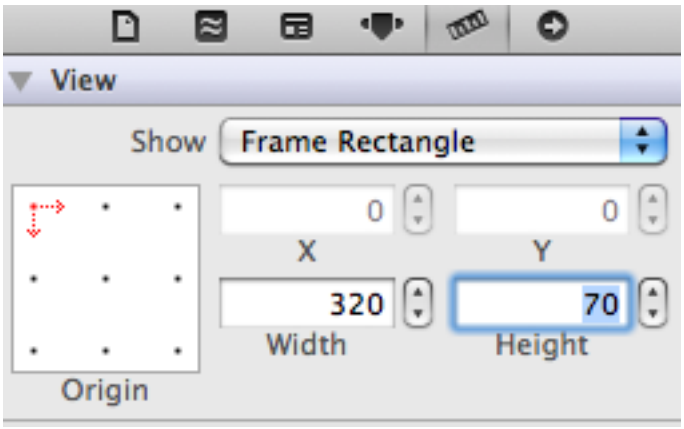


Table View Cell

Vamos a modificar su tamaño cambiándolo desde el *size inspector*, escribimos de altura: 70:

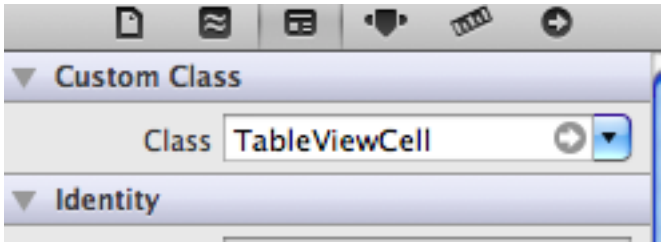


Ajustes de tamaño vista celda

Ahora queda arrastrar los objetos que queramos que aparezcan en la celda: 3 labels y una imagen por el momento. La celda queda de la siguiente manera en el *Interface builder*:

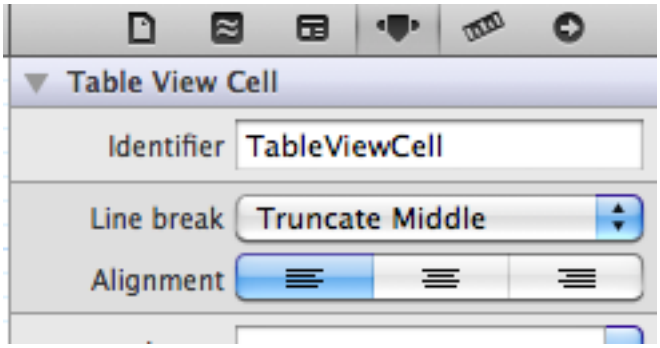
Vista celda

Ahora, en la pestaña de *Identity Inspector* escribimos la clase que corresponde a la vista, en nuestro caso será *TableViewCell*:



Asignamos la clase TableViewCell en Interface Builder

En la pestaña de *Attributes Inspector* escribimos un identificador para la celda: *TableViewCell*. Esto servirá más adelante para referenciar a ella desde el controlador de la tabla y así poder utilizar distintas celdas en la misma tabla si queremos:



Identificador en la celda

Con esto tenemos ya la vista de la celda creada. Ahora vamos a programar la clase.

2.3. Programando la celda

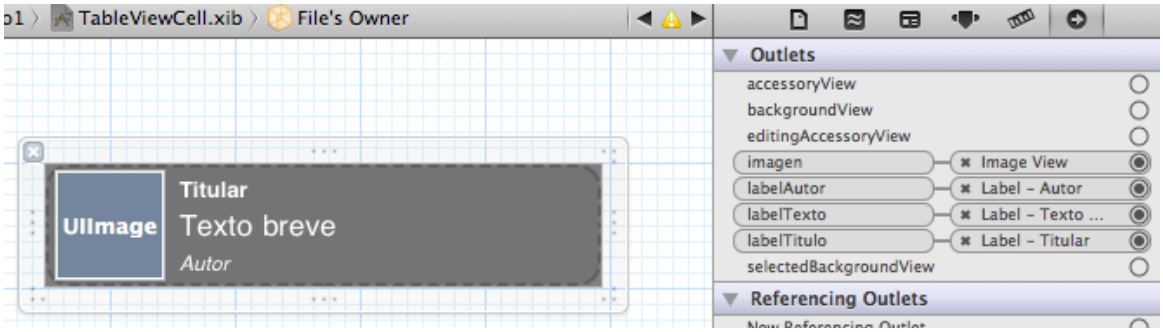
Dentro de la clase de la celda tendremos que añadir los atributos Outlet que hemos creado antes en la vista, para ello abrimos el fichero *TableViewCell.h* y escribimos lo siguiente:

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


@interface TableViewCell : UITableViewCell {
    UILabel *_labelTitulo;
    UILabel *_labelTexto;
    UILabel *_labelAutor;
    UIImageView *_imagen;
}

@property (nonatomic, retain) IBOutlet UILabel *labelTitulo;
@property (nonatomic, retain) IBOutlet UILabel *labelTexto;
@property (nonatomic, retain) IBOutlet UILabel *labelAutor;
@property (nonatomic, retain) IBOutlet UIImageView *imagen;

@end
```

En el fichero *TableViewCell.m* añadimos lo siguiente:

```
//Debajo de @implementation:

@synthesize labelTitulo=_labelTitulo;
@synthesize labelTexto=_labelTexto;
@synthesize labelAutor=_labelAutor;
@synthesize imagen=_imagen;
```

Ahora volvemos a la vista de la celda *TableViewCell.xib* y enlazamos los Outlets creados en la clase con los objetos de la vista:



Outlets de la vista celda

Una vez hecho esto abrimos de nuevo la clase controladora de la tabla *RootViewController.m* y modificamos los siguientes métodos:

```
- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:
(NSIndexPath *)indexPath {
    return 70.0f;
}

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

```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
    return 3; // para pruebas
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:
(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"TableViewCell";

    TableViewCell *cell = (TableViewCell*)[tableView
dequeueReusableCellWithIdentifier:
    CellIdentifier];
    if (cell == nil) {
            // Cargamos el Nib de la celda con ese identificador
        NSArray *array = [[NSBundle mainBundle]
loadNibNamed:@"TableViewCell"
        owner:self
        options:nil];
        cell = [array objectAtIndex:0];
    }

    // Configure the cell.

    cell.labelTitulo.text = [NSString stringWithFormat:
    @"Título noticia número %d", indexPath.row+1];
    cell.labelTexto.text = @"Texto de pruebas...";
    cell.labelAutor.text = @"Autor/es de la noticia";

    cell.imagen.image = [UIImage imageNamed:@"logo_mvl.png"];

    return cell;
}
```

La imagen *logo_mvl.png*la puedes descargar desde aqui y la debes de arrastrar al directorio de *Supporting files del proyecto*. Ahora ya podemos ejecutar la aplicación y nos debe de aparecer la tabla con las celdas que acabamos de programar.

Una vez que tenemos las celdas hechas vamos a aumentar un nivel más la personalización añadiendo un fondo determinado a las celdas pares y otro a las impares, para hacer un efecto "zebra". Para hacer esto necesitaremos dos imágenes más que las puedes descargar aquí.

Abrimos el nib de la celda *TableViewCell.xib* y arrastramos un *ImageView* al fondo de la celda ocupando todo el espacio. En el fichero *TableViewCell.h* añadimos un *UIImageView* como Outlet y después lo enlazamos desde la vista, como hemos hecho con el resto de elementos.

Ahora en el fichero *RootViewController.m*, dentro del método `cellForRowAtIndexPath` añadimos lo siguiente justo antes de `return cell;`

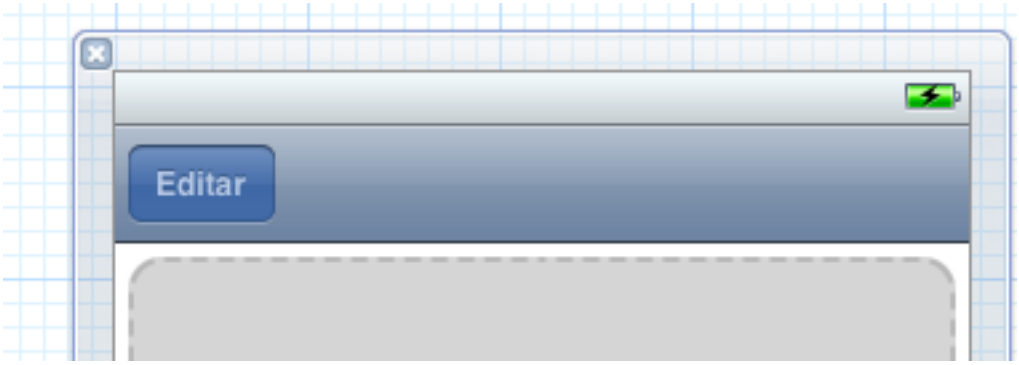
```
if (indexPath.row%2){
        cell.fondo.image = [UIImage imageNamed:@"fondo_celda1.png"];
    }
```

```
    else {
        cell.fondo.image = [UIImage imageNamed:@"fondo_celda2.png"];
    }
```

Después de modificar alguna propiedad de la tabla desde el *Interface Builder*, ejecutamos ahora el proyecto tendremos nuestra tabla con las celdas personalizadas.

3. Modo edición y personalización de tablas

Después de ver en el apartado anterior la forma de personalizar las filas de una tabla vamos a aprender como implementar y usar el modo edición de estas. La API del controlador de tabla (*TableView Controller*) dispone de varios métodos encargados de gestionar la edición, borrado e inserción de elementos. Esto es muy útil y bastante intuitivo sobre todo en aplicaciones de gestión como por ejemplo *ToDos*. Para entender el funcionamiento y aprender a implementar esto vamos a realizar un pequeño ejemplo paso a paso. Realizaremos una aplicación de gestión de tareas (ToDo) muy simple en la que podremos eliminar, editar y añadir tareas.



Botón editar en barra superior

Comenzamos creando un proyecto en XCode de tipo *Navigation-based application* que llamaremos *uisesion02-ejemplo2*. Hemos elegido este tipo de plantilla para aprovechar la vista de tabla y el control de navegación que vienen integrados desde el principio.

Ahora creamos un array dentro de la clase `RootViewController` que será nuestra fuente de datos para la tabla. Este array contendrá el listado de tareas pendientes de realizar. El fichero *RootViewController.h* queda como sigue:

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

@interface RootViewController : UITableViewController {
    NSMutableArray *_arrayTareas;
    UINavigationItem *_botonEditar;
}

@property (nonatomic, retain) NSMutableArray *arrayTareas;
@property (nonatomic, retain) IBOutlet UINavigationItem *botonEditar;
```

```
-(void) editarTabla:(id)sender;

@end
```

En el fichero *RootViewController.m*, dentro del método `viewDidLoad` creamos el botón de editar, lo añadimos a la barra de navegación e inicializamos el array. También implementamos el método que se llamará cuando se pulse el botón "editar":

```
@synthesize arrayTareas=_arrayTareas;
@synthesize botonEditar=_botonEditar;

- (void)viewDidLoad
{


    // Creamos el boton de editar a la izquierda de la barra de navegacion
    UIBarButtonItem *botonEditar = [[UIBarButtonItem alloc]
            initWithTitle:@"Editar"
            style:UIBarButtonSystemItemAction
            target:self
            action:@selector(editarTabla:)];

    self.navigationItem.leftBarButtonItem = botonEditar;
    [botonEditar release];


    // Iniciamos el array
    self.arrayTareas = [[NSMutableArray alloc] init];

    // Creamos las tareas
    [self.arrayTareas addObject:@"Hacer la compra"];
    [self.arrayTareas addObject:@"Cambiar de seguro de coche"];
    [self.arrayTareas addObject:@"Hacer deporte"];
    [self.arrayTareas addObject:@"Ir al banco"];
    [self.arrayTareas addObject:@"Confirmar asistencia cumple"];
    [self.arrayTareas addObject:@"Llamar a Juan"];


    [super viewDidLoad];
}

-(void) editarTabla:(id)sender {

    if (self.editing)
    {

        [super setEditing:NO animated:YES];
        [self.tableView setEditing:NO animated:NO];
        [self.navigationItem.leftBarButtonItem setTitle:@"Editar"];
        [self.navigationItem.leftBarButtonItem
setStyle:UIBarButtonItemStylePlain];

    }
    else
    {

        [super setEditing:YES animated:YES];
        [self.tableView setEditing:YES animated:YES];
        [self.navigationItem.leftBarButtonItem setTitle:@"Hecho"];
        [self.navigationItem.leftBarButtonItem
setStyle:UIBarButtonItemStyleDone];

    }
```

```
}
```

También debemos de completar los métodos `numberOfRowsInSection` y `cellForRowAtIndexPath` para mostrar los elementos del array en la tabla:

```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
        // una fila adicional al final de todas para añadir nuevas filas
    return [self.arrayTareas count]+1;
}

// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:
(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleDefault
        reuseIdentifier:CellIdentifier] autorelease];
    }

    // Configure the cell.
    if(indexPath.row == [self.arrayTareas count])
    {

        cell.textLabel.text = @"NUEVA";

    }
    else {
        cell.textLabel.text = [self.arrayTareas
objectAtIndex:indexPath.row];
    }

    return cell;


}
```

Una vez hecho esto ya podemos probar la aplicación. Veremos que al pulsar el botón de editar este cambia y se añaden unos pequeños botones a la izquierda de cada una de las celdas, si los pulsamos aparecerá un nuevo botón a la derecha el cual, al pulsarlo se debe de borrar la fila. Esto será lo siguiente que implementaremos.

Para implementar el borrado de elementos de la tabla debemos de descomentar el método `commitEditingStyle` y completarlo con el siguiente fragmento de código:

```
if (editingStyle == UITableViewCellEditingStyleDelete)

    {
```

```
        [self.arrayTareas removeObjectAtIndex:indexPath.row];
        [self.tableView deleteRowsAtIndexPaths:[NSArray
arrayWithObject:indexPath]
withRowAnimation:UITableViewRowAnimationRight];
        [self.tableView reloadData];

    } else if (editingStyle == UITableViewCellEditingStyleInsert)

    {

        [self.arrayTareas insertObject:@"Otra tarea" atIndex:
        [self.arrayTareas count]];
        [self.tableView insertRowsAtIndexPaths:[NSArray arrayWithObject:
        [NSIndexPath indexPathForRow:[self.arrayTareas count]
inSection:0]]
withRowAnimation:UITableViewRowAnimationFade];
        [self.tableView reloadData];
    }
```

Para asignar un modo de edición de la fila de la tabla en la que nos encontremos debemos de incluir el siguiente código dentro del método `editingStyleForRowAtIndexPath`:

```
if (self.editing == NO || !indexPath)
        return UITableViewCellEditingStyleNone;

    if (self.editing && indexPath.row == ([self.arrayTareas count]))
    {

        return UITableViewCellEditingStyleInsert;

    }
    else
    {

        return UITableViewCellEditingStyleDelete;

    }

    return UITableViewCellEditingStyleNone;
```

Por último, para implementar el reorden de las filas debemos de implementar los siguientes métodos:

```
// Override to support rearranging the table view.
- (void)tableView:(UITableView *)tableView moveRowAtIndexPath:
(NSIndexPath *)fromIndexPath toIndexPath:(NSIndexPath *)toIndexPath
{
    NSString *item = [[self.arrayTareas objectAtIndex:fromIndexPath.row]
retain];

    [self.arrayTareas removeObject:item];

    [self.arrayTareas insertObject:item atIndex:toIndexPath.row];

    [item release];
}



// Override to support conditional rearranging of the table view.
- (BOOL)tableView:(UITableView *)tableView
```

```
canMoveRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Return NO if you do not want the item to be re-orderable.
    return YES;
}
```

Ahora ya podemos ejecutar nuestro proyecto y comprobar que todo funciona correctamente:

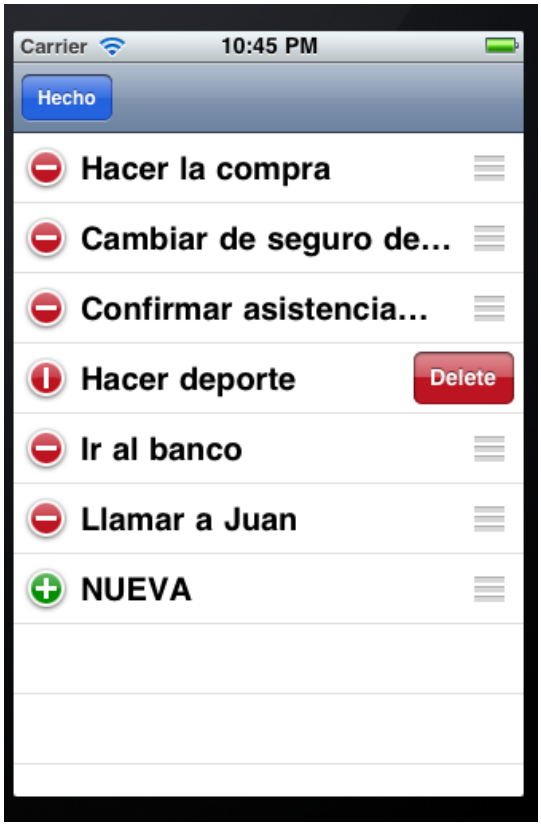


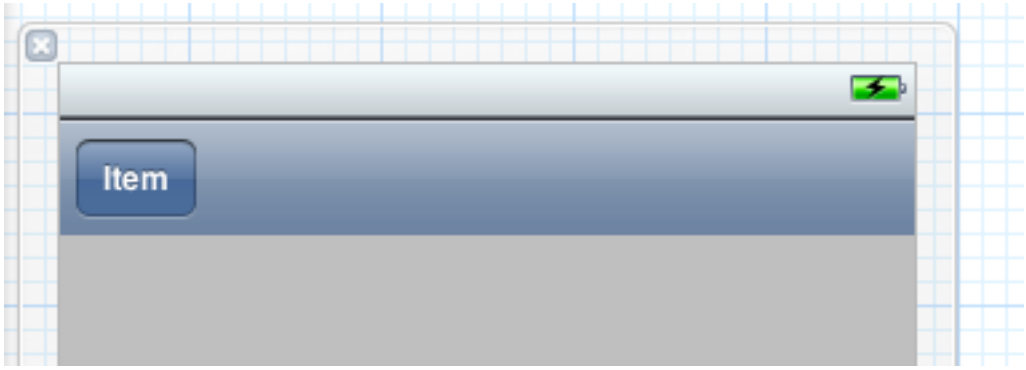
Tabla en modo de edición

4. Personalización de ToolBars

El *ToolBar*, al igual que la gran mayoría de componentes, se puede personalizar bastante mediante código para de esta forma conseguir las funcionalidades que deseemos en nuestras aplicaciones de iOS. En el siguiente ejemplo vamos a crear una vista *ToolBar* con un estilo determinado usando una imagen de fondo, estilos para los botones, añadiendo botones de distinto tipo, etc.

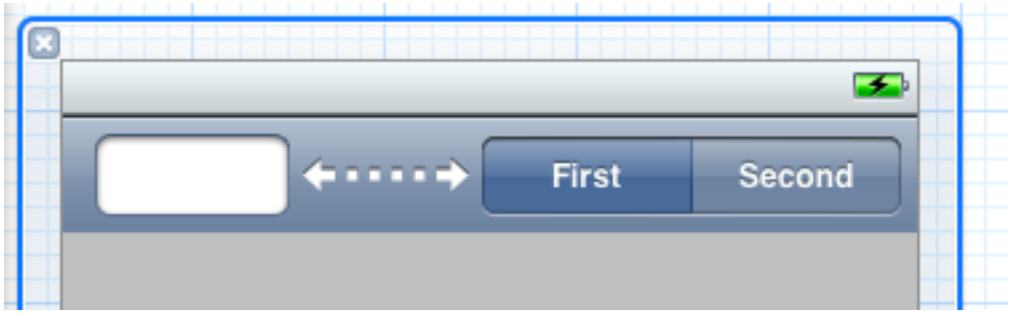
Comenzamos creando un nuevo proyecto de tipo *View-based application* con el nombre *uisesion02-ejemplo3*. Abrimos la vista *uisesion02_ejemplo3ViewController.xib* y arrastramos un objeto *ToolBar* a la vista, la situamos en la parte superior de ventana.

También situamos un objeto *Label* en el centro de la vista.



ToolBar básico

Ahora vamos a situar los elementos que deseemos sobre el *ToolBar*, en nuestro caso arrastramos por el siguiente orden desde el listado de objetos de la columna de la derecha un *Text Field*, un *Flexible Space Bar* y un *Segmented Control*. El *ToolBar* quedará de la siguiente manera:



ToolBar con botones y campos de texto

Como podemos observar, la función del objeto *Flexible Space* no es más que añadir un espacio flexible entre dos objetos dentro de un *ToolBar*. Una vez que tenemos la barra con todos sus elementos vamos a definir los elementos dentro de la clase. El fichero *uisesion02_ejemplo3ViewController.h* debe de quedar de la siguiente manera:

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

@interface uisesion02_ejemplo3ViewController : UIViewController {
    UIToolbar *_toolBar;
    UITextField *_textField;
    UISegmentedControl *_segmentedControl;
    UILabel *_segmentLabel;
}

@property (nonatomic, retain) IBOutlet UIToolbar *toolBar;
@property (nonatomic, retain) IBOutlet UITextField *textField;
@property (nonatomic, retain) IBOutlet UISegmentedControl
*segmentedControl;
@property (nonatomic, retain) IBOutlet UILabel *segmentLabel;
```

```
@end
```

Y en el fichero *uisesion02_ejemplo3ViewController.m*:

```
// Justo debajo del @implementation
@synthesize segmentedControl=_segmentedControl;
@synthesize textField=_textField;
@synthesize toolBar=_toolBar;
@synthesize segmentLabel=_segmentLabel;
```

Seguidamente tenemos que enlazar los *Outlets* dentro de la vista. En la vista nos situamos dentro del objeto *File's Owner* y arrastramos cada uno de sus Outlets hasta el elemento que corresponda. De esta forma ya tenemos "conectados" los elementos de la clase con los de la vista, ahora vamos a implementar la acción del *Segmented Control*, para ello escribimos la siguiente definición del método dentro de la clase *uisesion02_ejemplo3ViewController.h* y lo implementamos en el *.m*:

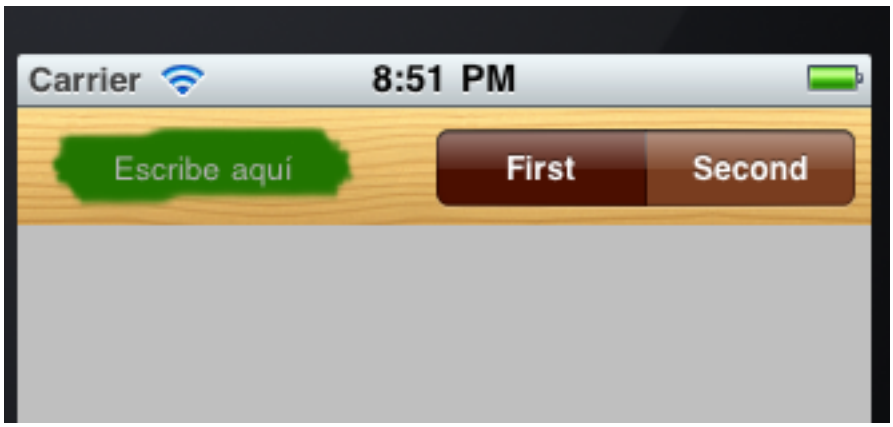
```
//uisesion02_ejemplo3ViewController.h

-(IBAction) segmentedControlIndexChanged;


//uisesion02_ejemplo3ViewController.m

-(IBAction) segmentedControlIndexChanged{
        switch (self.segmentedControl.selectedSegmentIndex) {
                case 0:
                        self.segmentLabel.text = @"Segmento 1
seleccionado.";
                        break;
                case 1:
                        self.segmentLabel.text = @"Segmento 2
seleccionado.";
                        break;

                default:
                         break;
         }

}
```

Para que el método se llame cuando se pulsa un botón del *Segmented Control* debemos enlazarlo dentro de la vista, para ello seleccionamos con la tecla cntrl pulsada el objeto *Segmented Control* y arrastramos hasta el objeto *File's Owner*. Ahi seleccionamos el método que hemos declarado justo antes: `segmentedControlIndexChanged`. En este punto ya podemos ejecutar el proyecto por primera vez y comprobar que al pulsar sobre uno de los botones del *Segmented Control* la etiqueta *Label* cambia.

Ya tenemos un objeto *ToolBar* con una personalización básica funcionando. Ahora vamos a personalizarlo un poco más: vamos a añadirle una imagen de fondo, a asignar un color de fondo al *Segmented Control* y a modificar el diseño del *TextField*. Para hacer todo esto debemos de modificar el método `viewDidLoad` de la clase *uisesion02_ejemplo3ViewControll*er y añadir el siguiente código:

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

    // Asignamos la imagen de fondo para el toolbar
    UIImageView *iv = [[UIImageView alloc] initWithImage:
    [UIImage imageNamed:@"fondo_madera.png"]];
    iv.frame = CGRectMake(0, 0, _toolBar.frame.size.width,
_toolBar.frame.size.height);
    iv.autoresizingMask = UIViewAutoresizingFlexibleWidth;

    // Añadimos la imagen al toolbar. Distinguimos si estamos en iOS 4 o
en iOS 5
    if([[[UIDevice currentDevice] systemVersion] intValue] >= 5)
        [_toolBar insertSubview:iv atIndex:1]; // iOS5 atIndex:1
    else
        [_toolBar insertSubview:iv atIndex:0]; // iOS4 atIndex:0


    // Añadimos color al Segmented Control
    [_segmentedControl setTintColor:[UIColor brownColor]];

    // Personalizamos el text field:
    _textField.textAlignment = UITextAlignmentCenter; //centramos el texto
    _textField.textColor = [UIColor whiteColor]; //texto de color blanco
    _textField.borderStyle = UITextBorderStyleNone; //quitamos el borde
del campo
    _textField.background = [UIImage imageNamed:@"fondo_textfield.png"];
//fondo
    [_textField setPlaceholder:@"Escribe aquí"]; //texto inicial
}
```

Para que el código funcione debemos de descargarnos las imágenes desde aquí e insertarlas en la carpeta *Supporting Files* del proyecto. Una vez hecho esto ya podemos ejecutar el proyecto y veremos como ha cambiado. De esta forma tendremos nuestro componente *ToolBar* bastante personalizado, lo que le da un aspecto visual bastante más agradable a la aplicación.



ToolBar personalizado

Por último comentar que el método que acabamos de implementar nos sirve tanto para personalizar el componente *TabBar* como el *Navigation Bar*.