Controladores

Índice

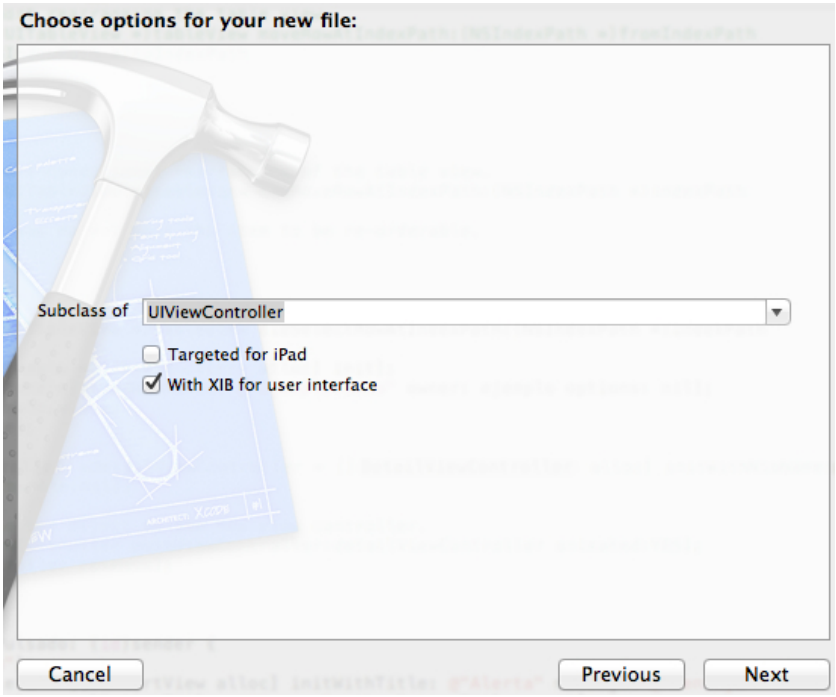
En la sesión anterior hemos visto cómo crear la interfaz con *Interface Builder* y como cargar las vistas definidas en el fichero NIB desde la aplicación. Sin embargo, lo más frecuente será crear un controlador asociado a cada fichero NIB que se encargue de gestionar la vista. Los controladores serán clases que heredarán de `UIViewController`, que tiene una propiedad `view` que se encargará de referenciar y retener la vista que gestiona.

Este controlador se utilizará además como *File's Owner* del NIB. Podríamos utilizar cualquier tipo de objeto como *File's Owner*, pero utilizar un objeto de tipo `UIViewController` nos aportará numerosas ventajas, como por ejemplo incorporar métodos para controlar el ciclo de vida de la vista y responder ante posibles cambios en el dispositivo.

Cuando mostremos en la pantalla la vista asociada al controlador (propiedad `view`), por ejemplo añadiéndola como subvista de la ventana principal, los eventos del ciclo de vida de dicha vista (carga, cambio de orientación, destrucción, etc) pasarán a ser gestionados por nuestro controlador.

1. Creación de un controlador propio

Para crear un nuevo controlador seleccionaremos *File > New File ... > iOS > Cocoa Touch > UIViewController subclass*. Nos permitirá crear una subclase de `UIViewController` o de `UITableViewController`, que es un tipo de controlador especializado para la gestión de tablas. También nos permite elegir si queremos crear un controlador destinado al iPad, o si queremos que cree automáticamente un fichero NIB asociado. Tras esto tendremos que introducir el nombre del fichero en el que guardaremos el controlador (que coincidirá con el nombre de la clase y del fichero NIB si hemos optado por crearlo).



Nuevo controlador

Normalmente crearemos el fichero NIB (XIB) junto al controlador. De esta forma ya creará el fichero NIB con el *File's Owner* configurado del tipo del controlador que estamos creando, y con una vista (`UIView`) raíz vinculada con la propiedad `view` del controlador. Dicha propiedad está definida en `UIViewController`, y se utilizará para referenciar la vista raíz de la pantalla. Por ejemplo, si hemos creado un controlador llamado `EjemploViewController` el *File's Owner* del NIB asociado será de dicha clase (atributo *Class* del inspector de identidad).

Podemos cargar un controlador, junto a su vista asociada, de la siguiente forma:

```
EjemploViewController *ejemploViewController =
   [[EjemploViewController alloc]
       initWithNibName:@"EjemploViewController" bundle:nil];
```

De esta forma el controlador se encarga de cargar la vista del fichero NIB automáticamente (no tenemos que cargarlo manualmente como hicimos en la sesión anterior). Cuando mostremos la vista de dicho controlador en pantalla (propiedad `ejemploViewController.view`) se irán ejecutando una serie de métodos del controlador que nos permitirán gestionar el ciclo de vida de dicha vista.

Nota

Por defecto Xcode le da el mismo nombre al fichero NIB que al controlador, en nuestro caso `EjemploViewController`. Sin embargo, no parece adecuado ponerle sufijo `Controller` a un fichero que sólo contiene la vista, sería más conveniente llamarlo `EjemploView.xib`. Podemos cambiarle el nombre manualmente. Incluso la API de Cocoa Touch tiene esto en cuenta, y aunque busquemos un NIB con sufijo `Controller`, si no lo encuentra buscará si

existe el fichero sin ese sufijo. De esta forma, al cargar el controlador podríamos no especificar ningún nombre de fichero NIB (pasamos `nil` como parámetro), y como buscará un NIB que se llame como el controlador, debido a la característica que acabamos de comentar será capaz de localizar el NIB aunque no lleve el sufijo `Controller`.

Si queremos que cargue el NIB por defecto, también podemos utilizar simplemente su inicializador `init`, que será equivalente a llamar al inicializador indicado anteriormente (que es el inicializado designado) pasando `nil` a sus dos parámetros.

```
EjemploViewController *ejemploViewController =
   [[EjemploViewController alloc] init];
```

Nota

Si el fichero NIB por defecto asociado al controlador no existiese, el controlador creará automáticamente una vista vacía con el tamaño de la ventana principal (`applicationFrame`).

Una vez inicializado el controlador, podemos hacer que su contenido se muestre en pantalla añadiendo su vista asociada (propiedad `view`) a la ventana principal que esté mostrando la aplicación, o a alguna de sus subvistas.

```
[self.window addSubview: ejemploViewController.view];
```

También deberemos retener el controlador en memoria, en alguna propiedad de nuestra clase, que en el caso anterior hemos supuesto que es *Application Delegate*. Por ese motivo también teníamos una propiedad que hacía referencia a la ventana principal. Si estuviésemos en otra clase, recordamos que podríamos hacer referencia a esta ventana con `[[UIApplication sharedApplication] keyWindow]`.

Atajo

Si nuestra aplicación está destinada sólo a iOS 4.0 o superior, podemos mostrar la vista de nuestro controlador en la ventana principal y retener el controlador en una única operación, sin tener que crear una nueva propiedad para ello. Esto es gracias a la propiedad `rootViewController` de `UIWindow`, a la que podemos añadir directamente el controlador y se encargará de retenerlo y de mostrar su vista en la ventana.

Una vez es mostrada en pantalla la vista, su ciclo de vida comenzará a gestionarse mediante llamadas a métodos del controlador al que está asociada (y que nosotros podemos sobrescribir para dar respuesta a tales eventos).

1.1. Ciclo de vida de los controladores

Los métodos básicos que podemos sobrescribir en un controlador para recibir notificaciones del ciclo de vida de su vista asociada son los siguientes:

```
- (void)loadView
- (void)viewDidLoad
- (void)viewDidUnload
```

```
- (void)didReceiveMemoryWarning
```

El más importante de estos métodos es `viewDidLoad`. Este método se ejecutará cuando la vista ya se haya cargado. En él podemos inicializar propiedades de la vista, como por ejemplo establecer los textos de los campos de la pantalla:

```
- (void)viewDidLoad {
        descripcionView.text = asignatura.descripcion;
        descripcionView.editable = NO;
}
```

Normalmente no deberemos sobrescribir el método `loadView`, ya que este método es el que realiza la carga de la vista a partir del NIB proporcionado en la inicialización. Sólo lo sobrescribiremos si queremos inicializar la interfaz asociada al controlador de forma programática, en lugar de hacerlo a partir de un NIB. En este método tendremos que crear la vista principal gestionada por el controlador, y todas las subvistas que sean necesarias:

```
- (void)loadView {
    UIView *vista = [[UIView alloc]
        initWithFrame: [[UIScreen mainScreen] applicationFrame]];
    ...
    self.view = vista;
    [vista release];
}
```

Podríamos también redefinir este método para cargar el contenido de un NIB de forma manual, tal como vimos en la sesión anterior, y asignarlo a la propiedad `view`.

Por último, el método `viewDidUnload` podrá ser llamado en situaciones de escasa memoria en las que se necesite liberar espacio y la vista no esté siendo mostrada en pantalla. Realmente, en situaciones de baja memoria se llamará al método `didReceiveMemoryWarning`. Este método comprobará si la vista no está siendo mostrada en pantalla (es decir, si la propiedad `view` del controlador tiene asignada como supervista `nil`). En caso de no estar mostrándose en pantalla, llamará a `viewDidUnload`, donde deberemos liberar todos los objetos relacionados con la vista (otras vistas asociadas que estemos reteniendo, imágenes, etc) que puedan ser recreados posteriormente.

Cuando la vista se vaya a volver a mostrar, se volverá a llamar a los métodos `loadView` y `viewDidLoad` en los que se volverá a crear e inicializar la vista.

Consejo

Un práctica recomendable es liberar los objetos de la vista en `viewDidUnload` y los objetos del modelo en `didReceiveMemoryWarning`, ya que este segundo podría ejecutarse en ocasiones en las que no se ejecuta el primero (cuando la vista esté siendo mostrada). Si sucede esto, no se volverá a llamar a `loadView` ni `viewDidLoad`, por lo que no podemos confiar en estos métodos para reconstruir los objetos del modelo. Lo recomendable es crear *getters* que comprueben si las propiedades son `nil`, y que en tal caso las reconstruyan (por ejemplo accediendo a la base de datos). Si sobrescribimos `didReceiveMemoryWarning`, no debemos olvidar llamar a `super`, ya que de lo contrario no se hará la llamada a `viewDidUnload` cuando la vista no sea visible.

A parte de estos métodos, encontramos otros métodos que nos avisan de cuándo la vista va a aparecer o desaparecer de pantalla, o cuándo lo ha hecho ya:

```
- viewWillAppear:
- viewDidAppear:
- viewWillDisappear:
- viewDidDisappear:
```

Todos ellos reciben un parámetro indicando si la aparición (desapación) se hace mediante una animación. Si los sobrescribimos, siempre deberemos llamar al correspondiente `super`.

1.2. Control de la orientación

El controlador también incorpora una serie de métodos para tratar la orientación de la vista. Esto nos facilitará gestionar los cambios de orientación del dispositivo y adaptar la vista a cada caso concreto. El método principal que deberemos sobrescribir si queremos permitir y controlar el cambio de orientación es el siguiente:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
       (UIInterfaceOrientation)interfaceOrientation
```

El tipo del parámetro `UIInterfaceOrientation` es una enumeración con las cuatro posibles orientaciones del dispositivo que reconocemos (vertical hacia arriba, horizontal izquierda, horizontal derecha y vertical hacia abajo). Cuando el usuario cambie la orientación del dispositivo se llamará a este método con la orientación actual para preguntarnos si la soportamos o no. Si devolvemos `YES`, entonces cambiará la vista para adaptarla a dicha orientación. Podemos utilizar las macros `UIInterfaceOrientationIsLandscape()` o `UIInterfaceOrientationIsPortrait()` para comprobar si la orientación es horizontal o vertical independientemente de su sentido.

Ayuda

Podemos probar los cambios de orientación en el simulador seleccionando *Hardware* > *Girar a la izquierda* (*cmd+Cursor izq.*) o *Hardware* > *Girar a la derecha* (*cmd+Cursor der.*). Si en el método anterior devolvemos siempre `YES`, veremos como la vista se adapta a todas las orientaciones posibles.

Si hemos configurado correctamente la vista para que el tamaño se ajuste automáticamente, no debe haber ningún problema al hacer la rotación. De todas formas, en `UIViewController` tenemos métodos adicionales para controlar la forma en la que se realiza la rotación con más detalle:

- `willRotateToInterfaceOrientation:duration:` Se va a realizar la rotación, pero todavía no se han actualizado los límites de la pantalla
- `willAnimateRotationToInterfaceOrientation:duration:` Los límites de la pantalla ya se han actualizado (se ha intercambiado el ancho y el alto), pero todavía

no se ha realizado la animación de la rotación.

- `didRotateFromInterfaceOrientation:` La rotación se ha completado. Se puede utilizar tanto el método anterior como este para ajustar el contenido de la vista al nuevo tamaño de la pantalla.

Nota

La orientación inicial con la que arranque la aplicación podrá ser una de las indicadas en el fichero `Info.plist`. Si es distinta a la orientación vertical, deberemos llevar cuidado al inicializar la vista, ya que en `viewDidLoad` todavía no se habrá ajustado el tamaño correcto de la pantalla. El tamaño correcto lo tendremos cuando se llame a `willAnimateRotationToInterfaceOrientation: duration:` o `didRotateFromInterfaceOrientation:`.

1.3. Tablas y fuente de datos

En el caso de las tablas (`UITableView`), la gestión de los datos que muestra el componente es más compleja. Para rellenar los datos de una tabla debemos definir una fuente de datos, que será una clase que implemente el protocolo `UITableViewDataSource`. Este protocolo nos obligará a definir al menos los siguientes métodos:

```
- (NSInteger) tableView:(UITableView *)tabla
  numberOfRowsInSection: (NSInteger)seccion
- (UITableViewCell *) tableView:(UITableView *)tabla
         cellForRowAtIndexPath: (NSIndexPath *)indice
```

En el primero de ellos deberemos devolver el número de elementos que vamos a mostrar en la sección de la tabla indicada mediante el parámetro `numberOfRowsInSection`. Si no indicamos lo contrario, por defecto la tabla tendrá una única sección, así que en ese caso podríamos ignorar este parámetro ya que siempre será `0`. Podemos especificar un número distinto de secciones si también definimos el método opcional `numberOfSectionsInTableView`, haciendo que devuelva el número de secciones de la tabla.

El segundo es el que realmente proporciona el contenido de la tabla. En él deberemos crear e inicializar cada celda de la tabla, y devolver dicha celda como un objeto de tipo `UITableViewCell`. En el parámetro `cellForRowAtIndexPath` se nos proporciona el índice de la celda que tendremos que devolver. La creación de las celdas se hará bajo demanda. Es decir, sólo se nos solicitarán las celdas que se estén mostrando en pantalla en un momento dado. Cuando hagamos *scroll* se irán solicitando los nuevos elementos que vayan entrando en pantalla sobre la marcha. Esto podría resultar muy costoso si cada vez que se solicita una celda tuviésemos que crear un nuevo objeto. Por este motivo realmente lo que haremos es reutilizar las celdas que hayan quedado fuera de pantalla.

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
```

```
    // (1) Reutilizamos una celda del pool
    static NSString *CellIdentifier = @"Cell";

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

    // (2) Introducimos los datos en la celda reutilizada
    cell.textLabel.text =
        [NSString stringWithFormat: @"Item %d", indexPath.row];

    return cell;
}
```

En la primera parte simplemente buscamos una celda disponible en la cola de celdas reutilizables que incorpora el objeto `UITableView`. Si no encuentra ninguna disponible, instancia una nueva y le asigna el identificación de reutilización con el que las estamos referenciando, para que cuando deje de utilizarse pase a la cola de reutilización de la tabla, y pueda reutilizarse en sucesivas llamadas.

Ejemplo de UITableView

En la segunda parte configuramos la celda para asignarle el contenido y el aspecto que deba tener el elemento en la posición `indexPath.row` de la tabla. En el ejemplo anterior cada elemento sólo tiene una cadena en la que se indica el número de la fila. Una implementación común consiste en mapear a la tabla los datos almacenados en un `NSArray`. Esto se puede hacer de forma sencilla:

```
- (NSInteger) tableView:(UITableView *)tabla
  numberOfRowsInSection: (NSInteger)seccion
{
    return [asignaturas count];
}

- (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];
    }

    // Configuramos la celda
    cell.textLabel.text =
        [[asignaturas objectAtIndex: indexPath.row] nombre];

    return cell;
}
```

Como vemos, mostrar los elementos de un `NSArray` en una tabla es tan sencillo como devolver el número de elementos del *array* (`count`) en `tableView:numberOfRowsInSection:`, y en `tableView:cellForRowAtIndexPath:` mostrar en la celda el valor del elemento en la posición `indexPath.row` del *array*.

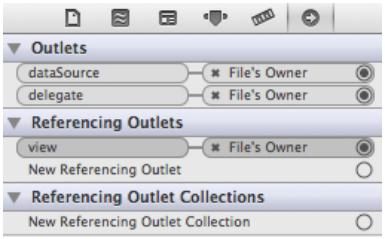
Al utilizar una tabla normalmente necesitaremos además un objeto delegado que implemente el protocolo `UITableViewDelegate`, que nos permitirá controlar los diferentes eventos de manipulación de la tabla, como por ejemplo seleccionar un elemento, moverlo a otra posición, o borrarlo.

La operación más común del delegado es la de seleccionar un elemento de la tabla. Para ello hay que definir el método `tableView:didSelectRowAtIndexPath:`, que recibirá el índice del elemento seleccionado.

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    UAAsignatura *asignatura =
            [asignaturas objectAtIndex: indexPath.row];
    UIAlertView *alert =
        [[UIAlertView alloc] initWithTitle:asignatura.nombre
                                   message:asignatura.descripcion
                                  delegate: nil
                         cancelButtonTitle: @"Cerrar"
                         otherButtonTitles: nil];
    [alert show];
    [alert release];

}
```

En este caso al pulsar sobre un elemento de la tabla, nos aparecerá una alerta que tendrá como título el nombre de la asignatura seleccionada, y como mensaje su descripción, y además un botón que simplemente cerrará la alerta.

Dado que normalmente siempre necesitaremos los dos objetos anteriores (fuente de datos y delegado) al utilizar tablas, para este tipo de componentes tenemos un controlador especial llamado `UITableViewController`, que hereda de `UIViewController`, y además implementa los protocolos `UITableViewDataSource` y `UITableViewDelegate`. En este controlador la propiedad `view` referenciará a una vista de tipo `UITableView`, y esta vista a su vez tendrá dos *outlets* (`delegate` y `dataSource`) que deberán hacer referencia al controlador, que es quien se comporta a su vez como delegado y fuente de datos.



Conexiones de UITableView

Si cambiamos el conjunto de datos a mostrar en la tabla, para que los cambios se reflejen en la pantalla deberemos llamar al método `reloadData` del objeto `UITableView`.

```
[self.tableView reloadData];
```

2. Controladores contenedores

Como hemos comentado, normalmente tendremos un controlador por cada pantalla de la aplicación. Sin embargo, esto no quiere decir que en un momento dado sólo pueda haber un controlador activo, sino que podremos tener controladores dentro de otros controladores. Es muy frecuente encontrar un controlador que se encarga de la navegación (se encarga de mostrar una barra de navegación con el título de la pantalla actual y un botón para volver a la pantalla anterior), que contiene otro controlador que se encarga de gestionar la pantalla por la que estamos navegando actualmente. También tenemos otro tipo de controlador contenedor que se encarga de mostrar las diferentes pantallas de nuestra aplicación en forma de pestañas (*tabs*).

Otra forma de contención se da cuando mostramos una vista modal. Utilizaremos nuestro controlador para mostrar la vista modal, que a su vez estará gestionada por otro controlador.

2.1. Controladores modales

Los controladores modales son la forma más sencilla de contención. Cuando mostramos un controlador de forma modal, su contenido reemplazará al del contenedor actual. Para hacer que un controlador muestre de forma modal otro controlador, llamaremos a su método `presentModalViewController:animated:`

```
- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    UAAsignatura *asignatura =
            [asignaturas objectAtIndex: indexPath.row];

    DetallesViewController *controladorModal =
        [[DetallesViewController alloc]
             initWithAsignatura: asignatura];

    [self presentModalViewController: controladorModal
                            animated: YES];
```

```
    [controladorModal release];
}
```

En el ejemplo anterior tenemos un controlador de una tabla, en el que al seleccionar un elemento muestra un controlador de forma modal con los detalles del elemento seleccionado. Al presentar el controlador modal, éste se almacena en la propiedad `modalViewController` del controlador padre (la tabla en nuestro ejemplo). Por otro lado, en el controlador modal (detalles en el ejemplo) habrá una propiedad `parentViewController` que hará referencia al controlador padre (tabla en el ejemplo). El controlador modal es retenido automáticamente por el controlador padre cuando lo presentamos.

Nota

En el iPhone/iPod touch el controlador modal siempre ocupará toda la pantalla. Sin embargo, en el iPad podemos hacer que esto no sea así. Podemos cambiar esto con la propiedad `modalPresentationStyle` del controlador modal.

Hemos de remarcar que un controlador modal no tiene porque ser un controlador secundario. Se puede utilizar este mecanismo para hacer transiciones a pantallas que pueden ser tan importantes como la del controlador padre, y su contenido puede ser de gran complejidad. Cualquier controlador puede presentar otro controlador de forma modal, incluso podemos presentar controladores modales desde otros controladores modales, creando así una cadena.

Cuando queramos cerrar la vista modal, deberemos llamar al método `dismissModalViewControllerAnimated:` del padre. Sin embargo, si llamamos a este método desde el controlador modal también funcionará, ya que hará un *forward* del mensaje a su padre (propiedad `parentViewController`).

La llamada a `dismissModalViewControllerAnimated:` funcionará de la siguiente forma según sobre qué controlador la llamemos:

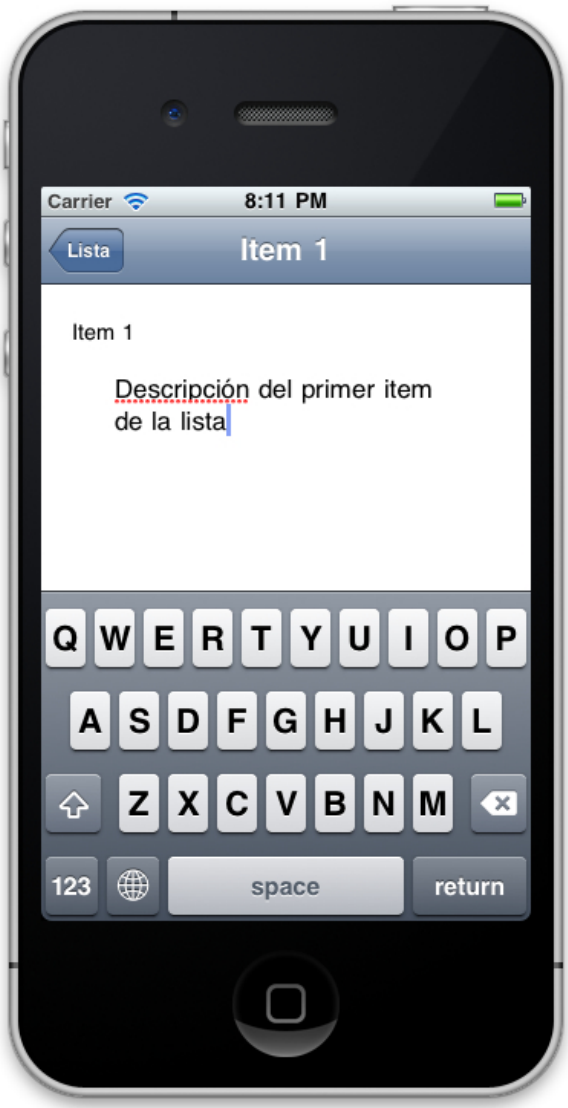
- Controlador con `modalViewController==nil` (el controlador modal que se está mostrando actualmente): Se redirije la llamada a su propiedad `parentViewController`.
- Controlador con `modalViewController!=nil` (controlador que ha presentado un controlador modal hijo): Cierra su controlador modal hijo, y todos los descendientes que este último tenga. Cuando se cierra con una única operación toda la cadena de controladores modales, veremos una única transición de la vista modal que se estuviese mostrando a la vista padre que ha recibido el mensaje para cerrar la vista modal.

2.2. Controlador de navegación

El tipo de aplicación más común que encontramos en iOS es el de las aplicaciones

basadas en navegación. En ellas al pasar a una nueva pantalla, ésta se apila sobre la actual, creando así una pila de pantallas por las que hemos pasado. Tenenemos una barra de navegación en la parte superior, con el título de la pantalla actual y un botón que nos permite volver a la pantalla anterior.

Pantalla raíz de la navegación

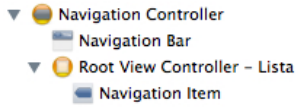


Pantalla de detalles en la pila de navegación

Para conseguir este funcionamiento, contamos con la clase `UINavigationController` que ya implementa este tipo de navegación y los elementos anteriores. Dentro de este controlador tenemos una vista central que es donde se mostrará el contenido de cada pantalla por la que estemos navegando. Cada una de estas pantallas se implementará mediante un controlador.

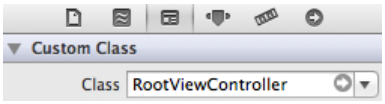
No será necesario crear ninguna subclase de `UINavigationController`, ya que en ella contamos ya con todos los elementos necesarios para implementar este esquema de navegación, pero si que tendremos que tener en cuenta que estamos utilizando este controlador contenedor en los controladores que implementemos para cada pantalla.

Cuando arrastramos un objeto de tipo `UINavigationController` a nuestro fichero NIB, veremos que crear bajo él una estructura con los siguientes elementos:



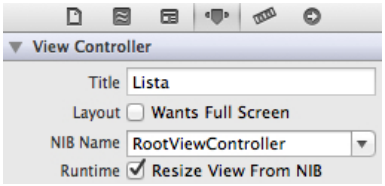
Elementos de Navigation Controller

- *Navigation Bar*: Define el fondo de la barra de navegación en la parte superior de la pantalla.
- *Navigation Item*: Define el contenido que se mostrará en la barra de navegación.
- *Root View Controller*: Controlador raíz que se utilizará para mostrar el contenido de la vista central. Este controlador podrá ser de un tipo propio en el que definamos el contenido de la pantalla raíz de la navegación. Para definir el tipo del controlador utilizaremos la propiedad *Class* del inspector de identidad.



Inspector de identidad del controlador raíz

Si queremos que cargue dicho controlador con un NIB distinto al NIB por defecto, podemos especificar el nombre del NIB en el inspector de atributos.



Inspector de atributos del controlador raíz

Atención

Si nuestro controlador (*Root View Controller* en el caso anterior) se crea dentro del NIB, nunca se llamará a su inicializador designado, ya que los objetos del NIB son objetos ya construidos que cargamos en la aplicación. Por lo tanto, si queremos realizar alguna inicialización, deberemos utilizar dentro de él el método `awakeFromNib`.

De forma alternativa, también podemos crear nuestro `UINavigationController` de forma programática. Para ello en primer lugar crearemos el controlador raíz, y después inicializaremos el controlador de navegación proporcionando en su constructor dicho controlador raíz:

```
RootViewController *rootViewController =
    [[RootViewController alloc] initWithNibName:@"RootViewController"
                                         bundle:nil];
UINavigationController *navController =
    [[UINavigationController alloc]
                      initWithRootViewController:rootViewController];
```

Vamos a centrarnos en ver cómo implementar estos controladores propios que mostraremos dentro de la navegación.

En primer lugar, `UIViewController` tiene una propiedad `title` que el controlador de navegación utilizará para mostrar el título de la pantalla actual en la barra de navegación. Cuando utilicemos este esquema de navegación deberemos siempre asignar un título a nuestros controladores. Cuando apilemos una nueva pantalla, el título de la pantalla anterior se mostrará como texto del botón de volver atrás.

Además, cuando un controlador se muestre dentro de un controlador de navegación, podremos acceder a dicho controlador contenedor de navegación mediante la propiedad `navigationController` de nuestro controlador.

Nota

Podemos personalizar con mayor detalle los elementos mostrados en la barra de navegación mediante la propiedad `navigationItem` de nuestro controlador. Esta propiedad hace referencia al objeto que veíamos como *Navigation Item* en Interface Builder. Los elementos que podremos añadir a dicha barra serán de tipo *Bar Button Item* (`UIBarButtonItem`). En Interface Builder podemos añadirlos como hijos de *Navigation Item* y conectarlos con *outlets* de dicho elemento.

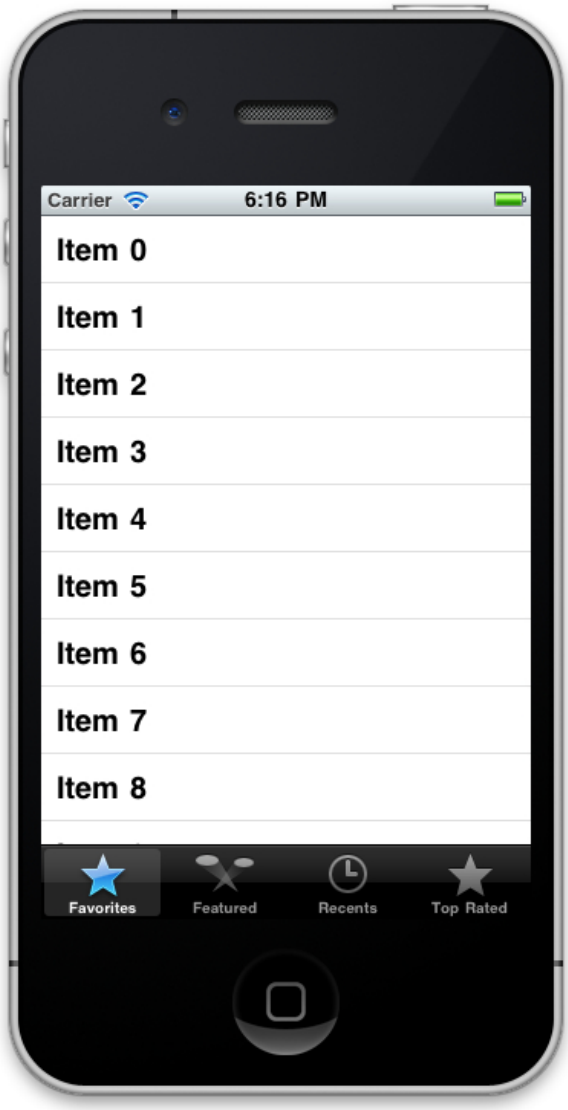
Cuando queramos pasar a la siguiente pantalla dentro de un controlador de navegación, utilizaremos el método `pushViewController:animated:` del controlador de navegación:

```
- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    DetailViewController *detailViewController =
        [[DetailViewController alloc]
            initWithNibName:@"DetailViewController"
                    bundle:nil];
    NSString *titulo = [NSString
        stringWithFormat: @"Item %d", indexPath.row];
    detailViewController.title = titulo;

    [self.navigationController
        pushViewController:detailViewController animated:YES];

    [detailViewController release];
}
```

Tras esta llamada se apila, se retiene, y se muestra la siguiente pantalla, que pasa a ser la cima de la pila. En la nueva pantalla veremos un botón para volver atrás en la barra de navegación. Al pulsarlo se desapilará la pantalla y se librerará la referencia. También podemos desapilar las pantallas de forma programática con `popViewControllerAnimated:`, que desapilará la pantalla de la cima de la pila. De forma alternativa, podemos utilizar `popToRootViewControllerAnimated:` o `popToViewController:animated:` para desapilar todas las pantallas hasta llegar a la raíz, o bien hasta llegar a la pantalla especificada, respectivamente.

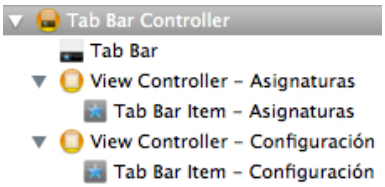
2.3. Controlador de barra de pestañas

Otro tipo de controlador contenedor predefinido que podemos encontrar es el controlador que nos permite organizar las pantallas de nuestra aplicación en forma de pestañas (*tabs*) en la parte inferior de la pantalla:



Aspecto de un Tab Bar Controller

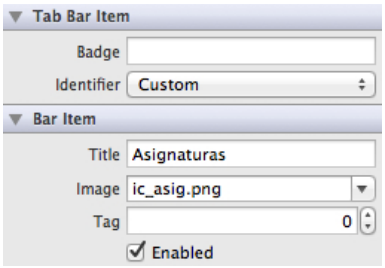
Dicho controlador se implementa en `UITabBarController`, y al igual que en el caso anterior, tampoco deberemos crear subclases de él. Cuando añadamos un elemento de tipo *Tab Bar Controller* a nuestro NIB se creará una estructura como la siguiente:



Elementos de Tab Bar Controller

- *Tab Bar*: Representa la barra de pestañas que aparecerá en la parte inferior de la pantalla, y que contendrá una serie de elementos.
- *Tab Bar Item*: Representa cada uno de los elementos de la barra de pestañas. Habrá un *item* por cada pestaña que tengamos en la barra. Aquí configuraremos el título y el icono que se mostrará en la pestaña.
- *View Controller*: Tendremos tantos controladores como pestañas en la barra. Podemos añadir nuevos controladores como hijos de *Tab Bar Controller* para añadir nuevas pestañas. Cada controlador tendrá asociado un *Tab Bar Item* con la configuración de su correspondiente pestaña.

Para configurar el aspecto de cada pestaña en la barra, seleccionaremos en el *dock* el correspondiente *Tab Bar Item* y accederemos a su inspector de atributos.



Inspector de atributos de Tab Bar Item

Como vemos, podemos o bien utilizar un aspecto predefinido (propiedad *Identifier*), o bien poner un título (*Title*) e imagen (*Image*) propios.

Iconos de las pestañas

Como iconos para las pestañas deberemos utilizar imágenes PNG de 30 x 30 px con transparencia. Los colores de la imagen (RGB) se ignorarán, y para dibujar el icono sólo se tendrá en cuenta la capa *alpha* (transparencia).

Podremos utilizar para cada pestaña cualquier tipo de controlador, que definirá el contenido a mostrar en dicha pestaña. Para especificar el tipo utilizaremos la propiedad *Class* de dicho controlador en el inspector de identidad (al igual que en el caso del controlador de navegación). Incluso podríamos poner como controlador para una pestaña un controlador de navegación, de forma que cada pestaña podría contener su propia pila de navegación.

Nota

> Si abrimos un controlador modal desde una pestaña o desde una pantalla de navegación, perderemos las pestañas y la barra de navegación.

Podemos crear también este controlador de forma programática. En este caso, tras instanciar el controlador de pestañas, tendremos que asignar a su propiedad `viewController` un *array* de los controladores a utilizar como pestañas. Tendrá tantas pestañas como elementos tenga la lista proporcionada:

```
PrimerViewController *controller1 = [[PrimerViewController alloc]
                initWithNibName:@"PrimerViewController" bundle:nil];
SegundoViewController *controller2 = [[SegundoViewController alloc]
                initWithNibName:@"SegundoViewController" bundle:nil];
TercerViewController *controller3 = [[TercerViewController alloc]
                initWithNibName:@"TercerViewController" bundle:nil];

UITabBarController *tabBarController =
                        [[UITabBarController alloc] init];
tabBarController.viewControllers = [NSArray arrayWithObjects:
                        controller1, controller2, controller3, nil];
```

El título de cada pestaña y el icono de la misma se configurarán en cada uno de los controladores incluidos. Por ejemplo, en `PrimerViewController` podríamos tener:

```
- (id)initWithNibName:(NSString *)nibNameOrNil
                bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil
                           bundle:nibBundleOrNil];
    if (self) {
        // Custom initialization
        self.title = @"Asignaturas";
        self.tabBarItem.image = [UIImage imageNamed:@"icono_asig"];
    }
    return self;
}
```

2.4. Controlador de búsqueda

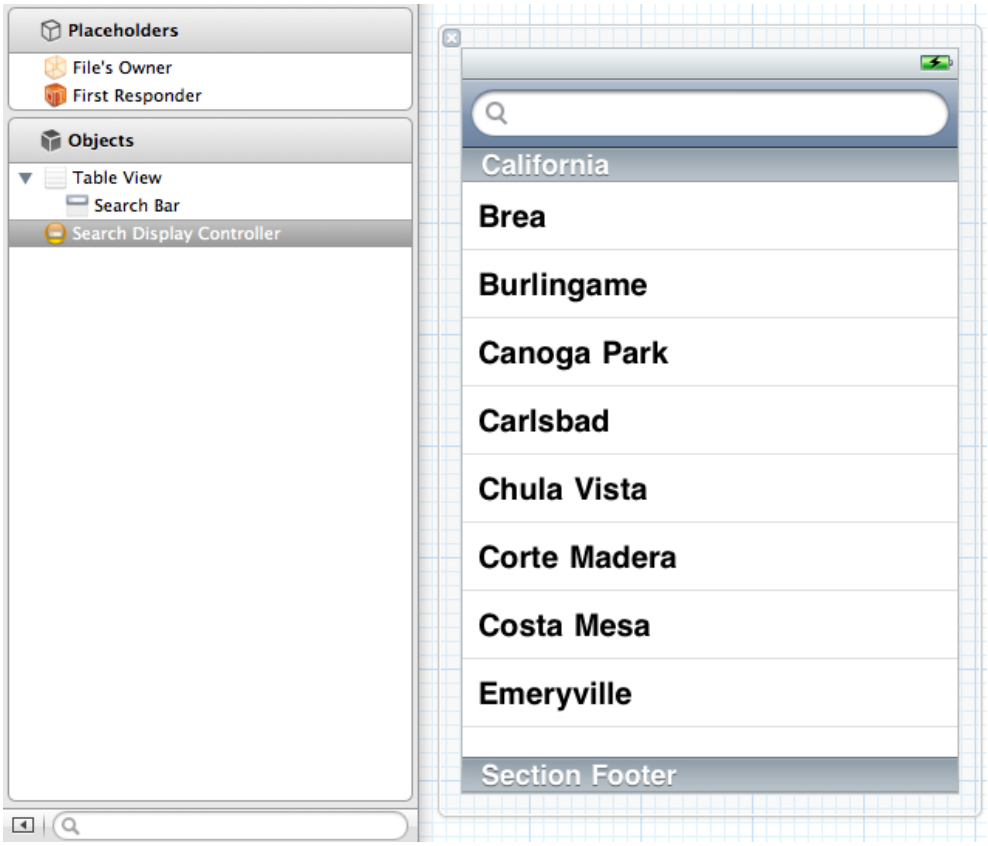
Es común encontrar una barra de búsqueda en las aplicaciones iOS, asociada a una tabla en la que se muestran los resultados de la búsqueda. Este comportamiento estándar se define en el controlador `UISearchDisplayController`. Este controlador es más complejo que los anteriores, por la coordinación necesaria entre los distintos elementos de la interfaz que intervienen en la búsqueda.

Los principales elementos que deberemos proporcionar al controlador de búsqueda son:

- **Barra de búsqueda**: Es la barra de búsqueda en la que el usuario introduce el texto a buscar, y se define con la clase `UISearchBar`. Se guardará en la propiedad `searchBar` del controlador de búsqueda.
- **Controlador de contenido**: Es el controlador en el que se mostrará el contenido resultante de la búsqueda. Normalmente este controlador será de tipo `UITableViewController`, y tendrá asociada una vista de tipo tabla donde se

mostrarán los resultados. Se guardará en la propiedad `searchContentsController` del controlador de búsqueda.

Habitualmente la barra de búsqueda se incluye como cabecera de la tabla, y cuando realizamos una búsqueda, permanecerá fija en la parte superior de la pantalla indicando el criterio con el que estamos filtrando los datos actualmente. Este comportamiento es el que se produce por defecto cuando arrastramos un elemento de tipo *Search Bar* sobre una vista de tipo *Table View*. Como alternativa, podemos también arrastrar sobre ella *Search Bar and Search Display Controller*, para que además de la barra nos cree el controlador de búsqueda en el NIB. En ese caso, el controlador de búsqueda será accesible mediante la propiedad `searchDisplayController` de nuestro controlador (podemos ver esto en los *outlets* generados).



Barra de busqueda en Interface Builder

Si queremos hacer esto mismo de forma programática, en lugar de utilizar Interface Builer, podemos incluir un código como el siguiente:

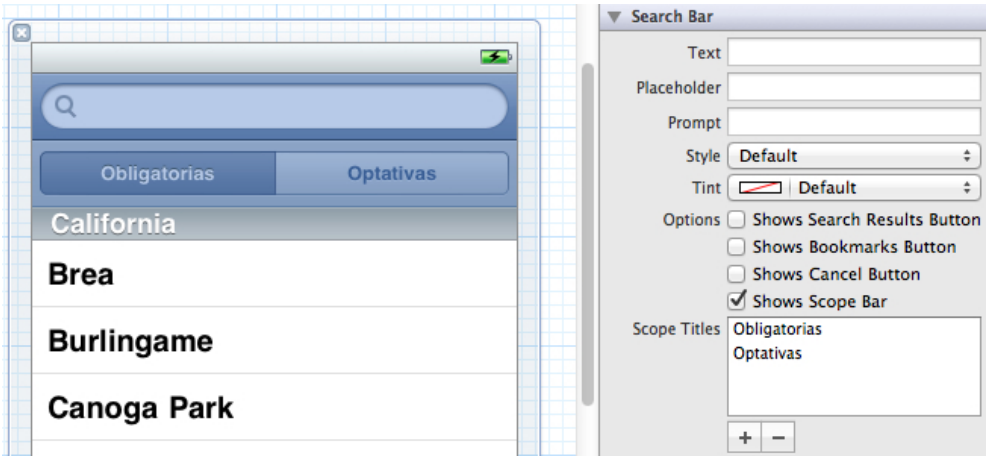
```
UISearchBar *searchBar = [[UISearchBar alloc] init];
[searchBar sizeToFit];
searchBar.delegate = self;
self.tableView.tableHeaderView = searchBar;
```

```
UISearchDisplayController *searchController =
          [[UISearchDisplayController alloc]
              initWithSearchBar:searchBar contentsController:self];
searchController.delegate = self;
searchController.searchResultsDataSource = self;
searchController.searchResultsDelegate = self;
self.searchController = searchController;
```

Cuidado

Aunque según la documentación de Apple la política de la propiedad `searchDisplayController` es `retain`, al crear el controlador de forma programática se asigna pero no se retiene, por lo que deberemos utilizar una propiedad creada por nosotros para que funcione correctamente. La propiedad `searchDisplayController` sólo nos servirá cuando hayamos creado el controlador mediante Interface Builder.

En la barra podemos introducir un texto que indique al usuario lo que debe introducir (*Placeholder*). De forma opcional, podemos añadir una barra de ámbito (*scope bar*) en la que indicamos el criterio utilizado en la búsqueda. Para ello activaremos esta barra en el inspector de atributos e introduciremos el nombre de cada elemento.



Barra de scope

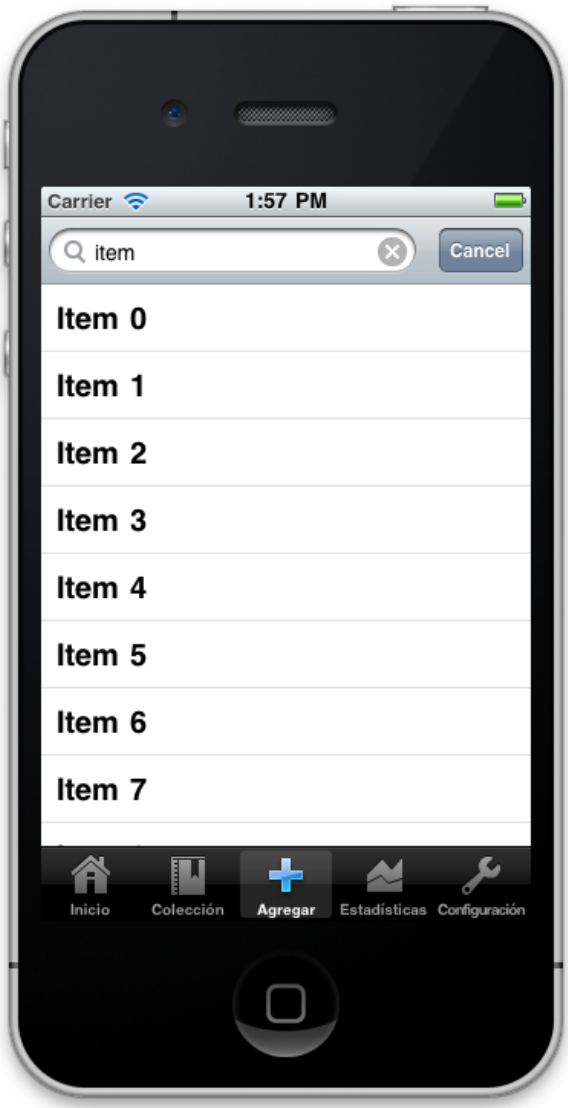
Es también habitual que la barra de búsqueda permanezca inicialmente oculta, y que tengamos que tirar hacia abajo de la lista para que aparezca. Esto podemos conseguirlo haciendo *scroll* hasta la primera posición de la vista en `viewDidLoad`:

```
[self.tableView
  scrollToRowAtIndexPath: [NSIndexPath indexPathForRow:0
                                             inSection:0]
        atScrollPosition:UITableViewScrollPositionTop
                animated:NO];
```

Cuando el usuario pulsa sobre la barra de búsqueda, el controlador de búsqueda entra en acción, situando la barra de búsqueda en una posición fija en la parte superior de la pantalla, y sombreando el resto de contenido, para que así el usuario centre su atención en rellenar el campo de búsqueda.

Foco en la barra de búsqueda

Una vez introducida la cadena de búsqueda, la barra quedará fija en la parte superior de la pantalla, indicando así que lo que estamos viendo son los resultados de la búsqueda, no la tabla original. Podemos volver a la tabla original pulsando sobre el botón *Cancel*.



Modo de filtrado

Hemos de destacar que realmente existen dos tablas distintas:

- **Tabla original**: Es la tabla que nosotros hemos creado en el NIB, y contiene la colección de datos completa. La encontramos en la propiedad `tableView` de nuestro controlador (`UITableViewController`), al igual que en cualquier tabla.
- **Tabla de resultados**: Es la tabla que crea el controlador de búsqueda para mostrar los resultados producidos por la búsqueda. Se encuentra en la propiedad `searchResultsTableView` del controlador de búsqueda. El controlador de encargará de crearla automáticamente.

Entonces, ¿por qué por defecto estamos viendo los mismos datos en ambos casos? Esto se

debe a que nuestro controlador se está comportando como delegado y fuente de datos de ambas tablas, por lo que resultan idénticas en contenido, aunque sean tablas distintas. Podemos ver en los *outlets* del controlador de búsqueda, que nuestro controlador (*File's Owner*), se comporta como `searchResultsDatasource` y `searchResultsDelegate` (además de ser `dataSource` y `delegate` de la tabla original).

La forma de determinar en la fuente de datos si debemos mostrar los datos originales y los resultados de la búsqueda, será comprobar qué tabla está solicitando los datos:

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    if(tableView ==
        self.searchDisplayController.searchResultsTableView) {
        return 1;
    } else {
        return [_items count];
    }
}

- (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];
    }

    if(tableView ==
        self.searchDisplayController.searchResultsTableView) {
        cell.textLabel.text = @"Item resultado";
    } else {
        cell.textLabel.text =
            [_items objectAtIndex: indexPath.row];
    }

    return cell;
}
```

La tabla de resultados aparecerá en cuanto introduzcamos texto en el campo de búsqueda. Si hacemos esto en el ejemplo anterior, veremos una tabla como la siguiente:

Resultados de la búsqueda

Sin embargo, lo normal será que cuando se introduzca un texto se realice un filtrado de los datos y esos datos filtrados sean los que se muestren en la tabla de resultado. ¿Cuándo deberemos realizar dicho filtrado? Lo ideal será hacerlo cuando el usuario introduce texto, o bien cuando pulsa el botón *Search*. Podremos estar al tanto de estos eventos de forma sencilla, ya que nuestro controlador, además de todo lo anterior, se comporta también como delegado (campo `delegate`), tanto de la barra de búsqueda (`UISearchBarDelegate`), como del controlador de búsqueda (`UISearchDisplayDelegate`).

Lo habitual será utilizar el método

`searchDisplayController:shouldReloadTableForSearchString:` de `UISearchDisplayDelegate` para realizar la búsqueda. Este método se ejecutará cada vez que cambie la cadena de búsqueda (al teclear en el campo). Deberemos devolver `YES` si con la cadena introducida queremos que se recargue la vista de resultados, o `NO` en caso contrario. Por ejemplo, podemos hacer que sólo se realice la búsqueda cuando la cadena introducida tenga una longitud mínima, para así no obtener demasiados resultados cuando hayamos introducido sólo unos pocos carácteres.

```
- (BOOL)searchDisplayController:
          (UISearchDisplayController *)controller
      shouldReloadTableForSearchString:(NSString *)searchString {

    self.itemsFiltrados = [self filtrarItems: _items
                                    busqueda: searchString];

    return YES;
}
```

Si hemos incluido una barra de ámbito, podemos responder a cambios del ámbito de forma similar, con el método `searchDisplayController:shouldReloadTableForSearchScope:`. Es recomendable definir un método genérico para realizar el filtrado que pueda ser utilizado desde los dos eventos anteriores.

Nota

En los métodos anteriores sólo deberemos devolver `YES` si hemos actualizado la lista de items dentro del propio método. Si queremos hacer una búsqueda en segundo plano, deberemos devolver `NO` y actualizar la lista una vez obtenidos los resultados.

En algunos casos puede que realizar la búsqueda sea demasiado costoso, y nos puede interesar que sólo se inicie tras pulsar el botón *Search*. Para ello podemos definir el método `searchBarSearchButtonClicked:` de `UISearchBarDelegate`:

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
    NSString *cadBusqueda = searchBar.text;
    self.itemsFiltrados = [self filtrarItems: _items
                                    busqueda: cadBusqueda];
}
```

Filtraremos los elementos de la vista según la cadena introducida (propiedad `text` de la barra de búsqueda). Guardamos el resultado del filtrado en una propiedad del controlador, y en caso de que se estén mostrando los resultados de la búsqueda, mostraremos dicha lista de elementos:

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    if(tableView ==
        self.searchDisplayController.searchResultsTableView) {
        return [_itemsFiltrados count];
    } else {
        return [_items count];
```

```
    }
}

- (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];
    }

    if(tableView ==
        self.searchDisplayController.searchResultsTableView) {
        cell.textLabel.text =
            [_itemsFiltrados objectAtIndex: indexPath.row];;
    } else {
        cell.textLabel.text =
            [_items objectAtIndex: indexPath.row];
    }

    return cell;
}
```

También podemos saber cuándo cambia el texto de la bara de búsqueda con el método `searchBar:textDidChange:`, o cuando cambia el ámbito seleccionado con `searchBar:selectedScopeButtonIndexDidChange:`.

3. Uso de storyboards

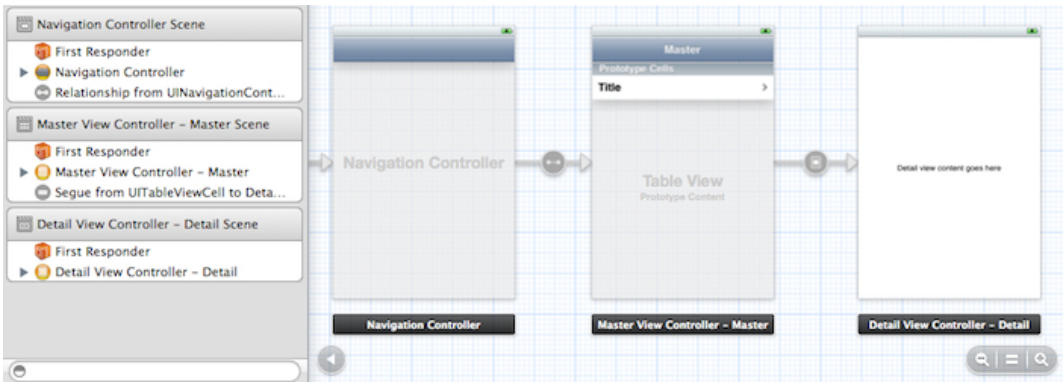
Los *storyboards* son una característica aparecida en iOS 5.0 y Xcode 4.2, que nos dará una forma alternativa para crear la interfaz de nuestras aplicaciones. Nos permitirán crear en un único fichero de forma visual las diferentes pantallas de la aplicación y las transiciones entre ellas. Están pensados principalmente para definir el flujo de trabajo de aplicaciones basadas en navegación o en pestañas.

Advertencia

Los *storyboards* no son compatibles con versiones de iOS previas a la 5.0.

Se definen en ficheros con extensión `.storyboard`, y podemos crearlos desde Xcode con la opción *New File ... > User Interface > Storyboard*. Por convenio, el *storyboard* principal de la aplicación se llamará `MainStoryboard.storyboard`.

Para que la aplicación arranque con el contenido del *storyboard*, deberemos especificar dicho fichero en la propiedad *Main Storyboard* del proyecto. Además, en `main.m` deberemos especificar como cuarto parámetro de `UIApplicationMain` la clase delegada de la aplicación, y en dicha clase delegada deberemos tener una propiedad de tipo `UIWindow` llamada `window`.



Editor del storyboard

Sobre el *storyboard* podremos arrastrar diferentes tipos de controladores. Cada uno de estos controladores aparecerán como una pantalla de la aplicación. Para cada controlador deberemos especificar su tipo en el atributo *Class* del inspector de identidad (podrán ser tipos definidos en Cocoa Touch, como `UINavigationController`, o tipos propios derivados de `UIViewController`). En las pantallas del *storyboard* no existe el concepto de *File's Owner*, sino que la relación de las pantallas con el código de nuestra aplicación se establecerá al especificar el tipo de cada una de ellas.

3.1. Segues

Las relaciones entre las distintas pantallas se definen mediante los denominados *segues*. Podemos ver los *segues* disponibles en el inspector de conexiones. Encontramos diferentes tipos de *segues*:

- *Relationship*: Define una relación entre dos controladores, no una transición. Nos sirve para relacionar un controlador contenedor con el controlador contenido en él.
- *Modal*: Establece una transición modal a otra pantalla (controlador).
- *Push*: Establece una transición de navegación a otra pantalla, que podrá utilizarse cuando la pantalla principal esté relacionada con un controlador de tipo `UINavigationController`.
- *Custom*: Permite definir una relación que podremos personalizar en el código.

Por ejemplo, si en una pantalla tenemos una lista con varias celdas, podemos conectar mediante un *segue* una de las celdas a un controlador correspondiente a otra pantalla. De esa forma, cuando el usuario pulse sobre dicha celda, se hará una transición a la pantalla con la que conecta el *segue*. Normalmente el origen de un *segue* será un botón o la celda de una tabla, y el destino será un controlador (es decir, la pantalla a la que vamos a hacer la transición).

En el primer controlador que arrastremos sobre el *storyboard* veremos que aparece una flecha entrante. Este es un *segue* que no tienen ningún nodo de origen, y que indica que es la primera pantalla del *storyboard*.

Si al arrancar la aplicación queremos tener acceso a dicho controlador raíz, por ejemplo para proporcionale datos con los que inicializarse, podemos acceder a él a través de la propiedad `rootViewController` de la ventana principal:

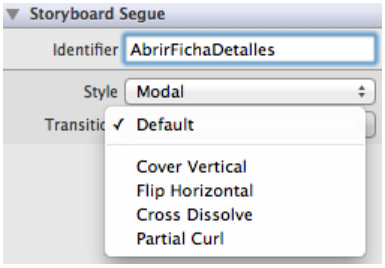
```
- (BOOL)application:(UIApplication *)application
      didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    UIViewController *controlador =
        (UIViewController *)self.window.rootViewController;
    ...
    return YES;
}
```

Cuando se ejecute un *segue*, se avisará al método `prepareForSegue:sender:` del controlador donde se originó. Sobrescribiendo este método podremos saber cuándo se produce un determinado *segue*, y en ese caso realizar las acciones oportunas, como por ejemplo configurar los datos de la pantalla destino.

```
-(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
  NSIndexPath *indexPath = [self.tableView indexPathForSelectedRow];
  [segue.destinationViewController setDetailItem:
          [NSString stringWithFormat:@"Item %d", indexPath.row]];
}
```

También podemos lanzar de forma programática un *segue* con el método `performSegueWithIdentifier:sender:`. Cada *segue* tendrá asociado un identificador, que se le asignará en la propiedad *Identifier* de la sección *Storyboard Segue* de su inspector de atributos.

En el caso de tener un *segue* de tipo Modal, podremos especificar en su propiedad *Transition* el estilo de la transición a la siguiente pantalla.



Atributos de los segues

Si queremos crear una transición personalizada, podemos crear una subclase de `UIStoryboardSegue`, establecer el tipo de *segue* a *Custom* (propiedad *Style*), y especificar la clase en la que hemos implementado nuestro propio tipo de *segue* en la propiedad *Segue Class*.

3.2. Tablas en el storyboard

Con el uso de los *storyboards* la gestión de las celdas de las tablas se simplifica

notablemente. Cuando introduzcamos una vista de tipo tabla, en su inspector de atributos (sección *Table View*), veremos una propiedad *Content* que puede tomar dos posibles valores:

- *Dynamic Prototypes*: Se utiliza para tablas dinámicas, en las que tenemos un número variable de filas basadas en una serie de prototipos. Podemos crear de forma visual uno o más prototipos para las celdas de la tabla, de forma que no será necesario configurar el aspecto de las celdas de forma programática, ni cargarlas manualmente.
- *Static Cells*: Se utiliza para tablas estáticas, en las que siempre se muestra el mismo número de filas. Por ejemplo, nos puede servir para crear una ficha en la que se muestren los datos de una persona (nombre, apellidos, dni, telefono y email).

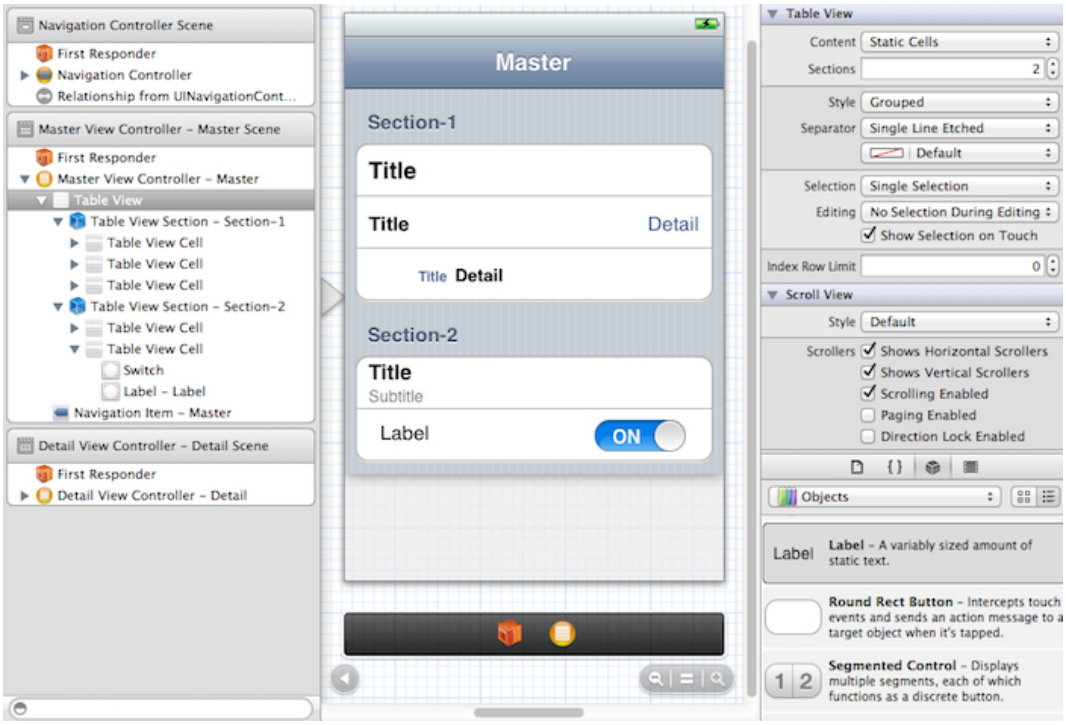


Tabla estática en el storyboard

En el caso de las tablas dinámicas, si definimos más de un prototipo, podemos dar a cada uno de ellos un identificador de reutilización diferente, para así en el código poder seleccionar el tipo deseado (esto lo especificaremos en el inspector de atributos de la celda prototipo, sección *Table View Cell*, propiedad *Identifier*).

Además, en el código se nos asegura que el método `dequeueReusableCellWithIdentifier:` siempre nos va a dar una instancia de una celda correspondiente al prototipo cuyo identificador se especifique como parámetro. Por lo tanto, ya no será necesario comprobar si nos ha devuelto `nil` y en tal caso instanciarla nosotros, como hacíamos anteriormente, sino que podremos confiar en que siempre nos devolverá una instancia válida.

```
UITableViewCell *cell = [tableView
       dequeueReusableCellWithIdentifier:CellIdentifier];

cell.textLabel.text = @"Item";
```

Las tablas estáticas podrán crearse íntegramente en el *storyboard*. Cuando creemos una tabla de este tipo podremos especificar el número de secciones (propiedad *Sections* en *Table View*), y en el *dock* aparecerá un nodo para cada sección, que nos permitirá cambiar sus propiedades. Dentro de cada sección podremos también configurar el número de filas que tiene (propiedad *Rows* en *Table View Section*), y veremos cada una de estas filas de forma visual en el editor, pudiendo editarlas directamente en este entorno.

Para cada celda podemos optar por utilizar un estilo predefinido, o bien personalizar su contenido (*Custom*). Esto lo podemos configurar en la propiedad *Style* de *Table View Cell*. En el caso de optar por celdas personalizadas, deberemos arrastrar sobre ellas las vistas que queramos que muestren.

También podremos conectar las celdas mediante *segues*, de forma que cuando se pulse sobre ellas se produzca una transición a otra pantalla.