Controladores

Índice

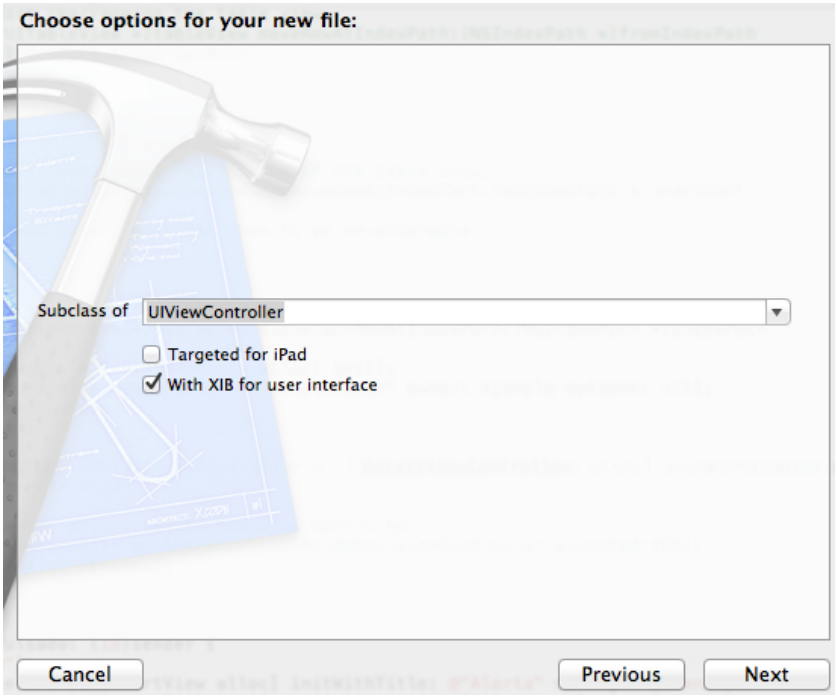
En la sesión anterior hemos visto cómo crear la interfaz con *Interface Builder* y como cargar las vistas definidas en el fichero NIB desde la aplicación. Sin embargo, lo más frecuente será crear un controlador asociado a cada fichero NIB que se encargue de gestionar la vista. Los controladores serán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

2. Controlador para las tablas

2.1. 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.

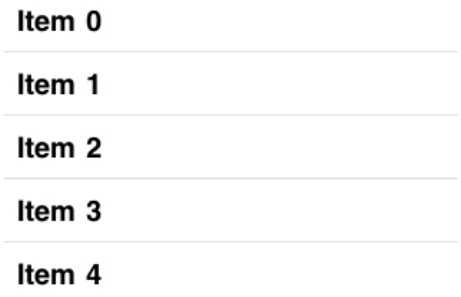
```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // (1) Reutilizamos una celda del pool
    static NSString *CellIdentifier = @"Cell";

    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*.

2.2. Delegado de la tabla

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.

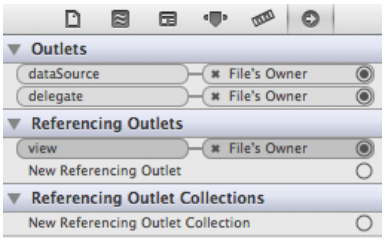
```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    UAAsignatura *asignatura =
            [asignaturas objectAtIndex: indexPath.row];
    UIAlertView *alert =
        [[UIAlertView alloc] initWithTitle:asignatura.nombre
                             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

2.3. Modo de edición

Las tablas cuentan con un modo de edición en el que podemos modificar los elementos de la tabla, cambiarlos de orden, eliminarlos, o añadir nuevos. Para gestionar este modo de edición bastará con añadir algunos métodos adicionales del *data source* y del delegado de la tabla. En el controlador de la tabla (*TableView Controller*) también disponemos de facilidades para trabajar con este modo de edición. Esto es muy útil y bastante intuitivo por ejemplo para aplicaciones de gestión de tareas. Para entender el funcionamiento y aprender a implementar esto vamos a realizar un ejemplo muy sencillo de aplicación en la que podremos eliminar, editar, añadir y reordenar tareas.

Para esta aplicación crearemos primero los métodos vistos anteriormente (`numberOfRowsInSection` y `cellForRowAtIndexPath`) para poblar la tabla de datos:

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

    // 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"];
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return [self.arrayTareas 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];
    }

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

    return cell;
}
```

En la plantilla que Xcode crea de un `UITableViewController`, en `viewDidLoad` veremos que aparece comentada una línea que se encarga de añadir un botón de edición a la barra de navegación.

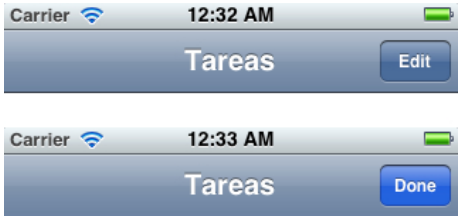
```
- (void)viewDidLoad
{
    [super viewDidLoad];

    // Uncomment the following line to preserve selection
    // between presentations.
    // self.clearsSelectionOnViewWillAppear = NO;

    // Uncomment the following line to display an Edit button in
    // the navigation bar for this view controller.
    self.navigationItem.rightBarButtonItem = self.editButtonItem;

    ...
}
```

Este botón se encargará de alternar entre el modo de edición y el modo normal, cambiando de aspecto según el modo actual.



Botón de edición

Si queremos realizar alguna acción cada vez que cambiemos entre el modo de edición y el modo normal, podemos sobrescribir el método `setEditing:animated:` del controlador:

```
- (void)setEditing:(BOOL)editing animated:(BOOL)animated {
    [super setEditing:editing animated:animated];
}
```

Alternativa. Podemos alternar también manualmente entre el modo de edición y el

normal llamando manualmente al método anterior. Por ejemplo, podríamos crear un botón propio que haga esto:

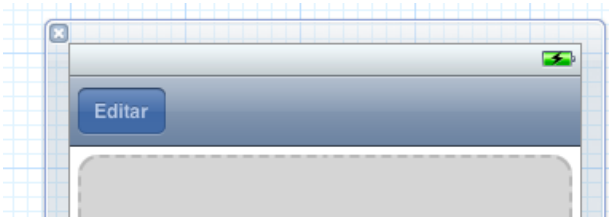
```
- (void)viewDidLoad
{
    [super viewDidLoad];

    // 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];

    ...
}

-(void) editarTabla:(id)sender {
    if (self.editing) {
        [super setEditing:NO animated:YES];
        [self.navigationItem.leftBarButtonItem setTitle:@"Editar"];
        [self.navigationItem.leftBarButtonItem
                                setStyle:UIBarButtonItemStylePlain];
    } else {
        [super setEditing:YES animated:YES];
        [self.navigationItem.leftBarButtonItem setTitle:@"Hecho"];
        [self.navigationItem.leftBarButtonItem
                                setStyle:UIBarButtonItemStyleDone];
    }
}
```



Botón editar en barra superior

En la implementación anterior hemos realizado de forma manual lo mismo que hace de forma automática el botón `self.editButtonItem` proporcionado por el sistema.

Una vez hecho esto ya podemos probar la aplicación. Veremos que al pulsar el botón de editar éste 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 deberá borrar la fila. Esto será lo siguiente que implementaremos.

2.4. Borrado

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];
} else if (editingStyle == UITableViewCellEditingStyleInsert) {
}
```

Como vemos, con este método indicamos la acción a realizar cuando se realiza alguna acción de edición. Por el momento sólo hemos indicado lo que debe hacer cuando se pulsa sobre el botón de borrar un elemento de la tabla, que es el que aparece por defecto.

Podemos fijarnos también en que en primer lugar debemos eliminar el *item* correspondiente a la fila de nuestro almacen de datos (en nuestro caso el *array* de tareas en memoria). Tras esto, llamamos a `deleteRowsAtIndexPaths:withRowAnimation:` sobre la tabla para que la fila que contenía el dato eliminado desaparezca mediante una animación.

Importante

Al eliminar una fila de la tabla, se comprueba que tras la eliminación el *data source* proporcione un elemento menos que antes. Por lo tanto es fundamental haber borrado el *item* de nuestro *array* de tareas, ya que de no haberlo hecho se producirá un error en tiempo de ejecución y se cerrará la aplicación.

Al implementar `commitEditingStyle` también se habilita el modo *swipe to delete*. Esto consiste en que si hacemos un gesto de barrido horizontal sobre una de las filas de la tabla, nos dará la opción de borrarla.

2.5. Reordenación

Dentro del modo de edición también tenemos la posibilidad de reordenar las filas de la tabla, arrastrándolas mediante un icono que aparece en su parte derecha. Para poder realizar esto debemos implementar los siguientes métodos del delegado de la tabla:

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

// 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;
}
```

Con esto podremos especificar la forma en la que la reordenación de la tabla se refleja en los datos almacenados, y también podemos indicar qué elementos son ordenables y cuáles no lo son.

2.6. Inserción

Por último, vamos a ver cómo insertar nuevos elementos en la tabla. Para ello tenemos diferentes opciones, entre las que destacamos:

- Añadir un botón en la barra de navegación que sirva para insertar un nuevo *item* en la tabla. Podemos utilizar el botón proporcionado por el sistema para añadir elementos.
- Añadir una nueva entrada en la tabla sólo cuando estemos en modo de edición, que nos permita añadir un nuevo *item*. Esta entrada tendrá como icono el símbolo '+', en lugar de '-' como ocurre por defecto para las filas cuyo estilo de edición es borrado.

Si optamos por el botón en la barra de navegación, la inserción de elementos es sencilla. En el evento de pulsación del botón añadiremos el nuevo *item* a nuestro almacén interno de datos, y tras esto añadiremos el nuevo elemento a la tabla mediante una animación:

```
- (void)addTarea:(id)sender {
    NSIndexPath *indexPath =
        [NSIndexPath indexPathForRow:[self.arrayTareas count]
                           inSection:0];
    [self.arrayTareas addObject:@"Tarea nueva"];
    [self.tableView
        insertRowsAtIndexPaths: [NSArray arrayWithObject:indexPath]
              withRowAnimation:UITableViewRowAnimationTop];
}
```

Si optamos por la opción de añadir una fila adicional para la inserción, en primer lugar deberemos hacer que el *data source* proporcione dicha fila en el caso en el que estemos en modo de edición:

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

// 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];
    }

    // Configure the cell.
    if(indexPath.row == [self.arrayTareas count]) {
        // Fila de inserción al final de la tabla
        cell.textLabel.text = @"NUEVA";
    } else {
        cell.textLabel.text =
            [self.arrayTareas objectAtIndex:indexPath.row];
    }

    return cell;
}
```

Además, deberemos hacer que esta fila aparezca o desaparezca cuando entremos y salgamos del modo de edición. Para esto podemos sobrescribir el método que cambia entre el modo de edición y el normal, y según a qué modo estemos cambiando insertar o eliminar mediante una animación la fila de inserción:

```
- (void)setEditing:(BOOL)editing
                          animated:(BOOL)animated {
    [super setEditing:editing animated:animated];

    NSArray *rows = [NSArray arrayWithObject:
        [NSIndexPath indexPathForRow:[self.arrayTareas count]
                           inSection:0]];
    if(editing) {
        [self.tableView insertRowsAtIndexPaths:rows
                          withRowAnimation:UITableViewRowAnimationTop];
    } else {
        [self.tableView deleteRowsAtIndexPaths:rows
                          withRowAnimation:UITableViewRowAnimationTop];
    }
}
```

Nota

Al insertar la fila de esta forma, también nos aparecerá cuando entremos en modo *swipe to delete*, ya que este modo también ejecuta el método anterior para hacer que el controlador entre en modo de edición. Podemos saber cuando entramos en modo *swipe to delete* mediante los métodos `tableView:willBeginEditingRowAtIndexPath:` y `tableView:didEndEditingRowAtIndexPath:` del delegado. Podemos implementarlos y utilizarlos para activar y desactivar un *flag* que nos indique si la edición es de tipo *swipe to delete*. En ese caso, podríamos decidir no mostrar la fila de inserción. Otra opción es no mostrar esta fila en `setEditing:animated:`, sino entrar en modo de edición con un botón propio e insertar la fila en el evento de pulsación de dicho botón.

Para que en esta fila aparezca el símbolo de inserción ('+') en lugar de borrado ('-'), deberemos definir el siguiente método del *data source*:

```
- (UITableViewCellEditingStyle)tableView:(UITableView *)tableView
           editingStyleForRowAtIndexPath:(NSIndexPath *)indexPath {
    if(indexPath.row < [self.arrayTareas count]) {
        return UITableViewCellEditingStyleDelete;
    } else {
        return UITableViewCellEditingStyleInsert;
```

```
    }
}
```

Para finalizar, haremos que al pulsar sobre el icono de inserción se realice la acción correspondiente:

```
- (void)tableView:(UITableView *)tableView
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
     forRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (editingStyle == UITableViewCellEditingStyleDelete) {
        // Delete the row from the data source.
        [self.arrayTareas removeObjectAtIndex:indexPath.row];
        [tableView
            deleteRowsAtIndexPaths: [NSArray arrayWithObject:indexPath]
                  withRowAnimation:UITableViewRowAnimationFade];
    } else if (editingStyle == UITableViewCellEditingStyleInsert) {
        // Create a new instance of the appropriate class, insert it
        // into the array, and add a new row to the table view.
        [self.arrayTareas addObject:@"Tarea nueva"];
        [tableView
            insertRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
                  withRowAnimation:UITableViewRowAnimationTop];
    }
}
```

Con esto ya funcionará correctamente la inserción de nuevas tareas. Ahora ya podemos ejecutar nuestro proyecto y comprobar que todo funciona correctamente:

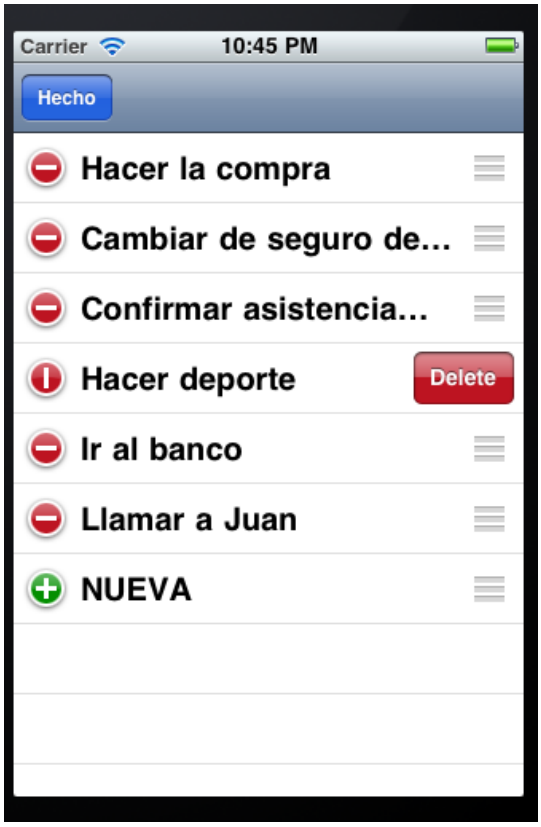


Tabla en modo de edición

Sin embargo, encontramos un problema al combinar la fila de inserción con la reordenación de filas, ya que esta fila no pertenece al *array* de tareas y no puede ser reordenada. Debemos añadir código adicional para evitar este problema. En primer lugar, haremos que la fila de inserción no sea reordenable:

```
// 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 indexPath.row < [self.arrayTareas count];
}
```

Aun así, podría ocurrir que el usuario intentase mover una tarea a una posición posterior a la fila de inserción. Para evitar que esto ocurra definiremos el siguiente método del delegado:

```
- (NSIndexPath *)tableView:(UITableView *)tableView
 targetIndexPathForMoveFromRowAtIndexPath:(NSIndexPath *)sourceIndexPath
 toProposedIndexPath:(NSIndexPath *)proposedDestinationIndexPath {
    if(proposedDestinationIndexPath.row >= [self.arrayTareas count]) {
        return [NSIndexPath indexPathForRow:([self.peliculas count]-1)
                                  inSection:0];
    } else {
        return proposedDestinationIndexPath;
    }
}
```

Este método nos permite modificar la posición destino en la reordenación de elementos. Cuando el usuario arrastre a una posición, se llama a este método con esa posición como propuesta, pero nosotros podemos cambiarla devolviendo una posición distinta. En nuestro caso, si la posición destino propuesta es posterior a la de la fila especial de inserción, devolvemos como posición de destino definitiva la posición anterior a esta fila, para evitar que la película pueda moverse a una posición inválida. Con esto la aplicación funcionará correctamente.

3. Controlador de búsqueda

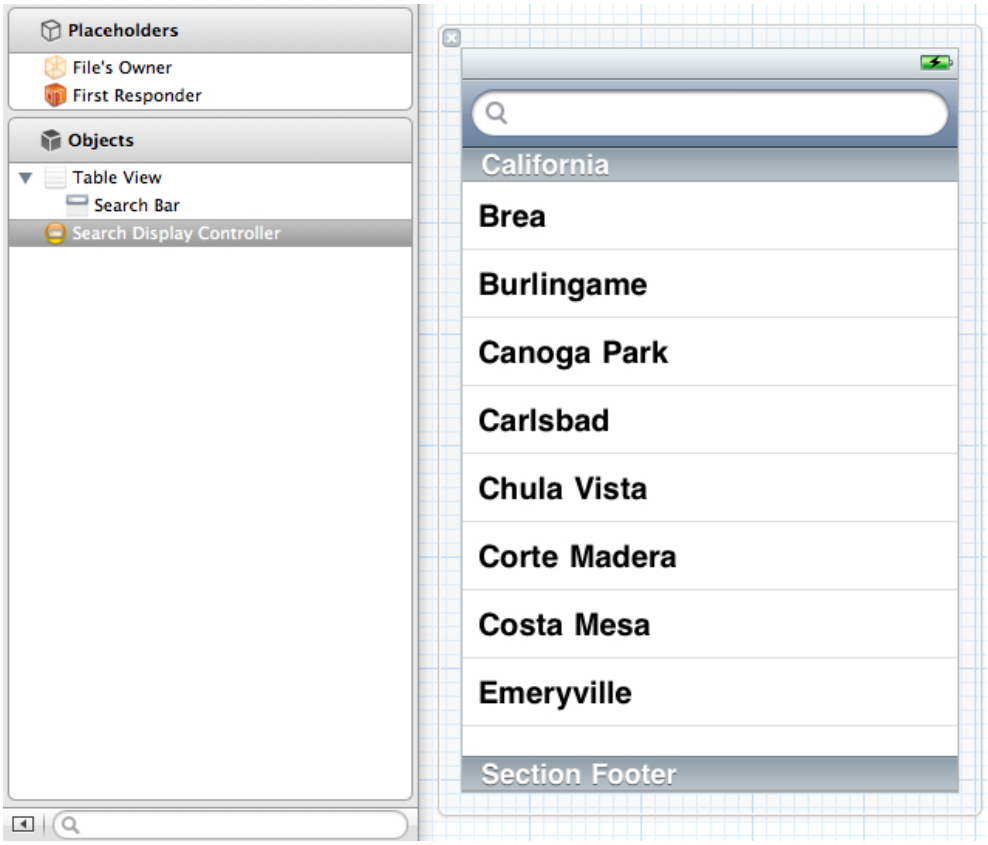
Es común encontrar una barra de búsqueda en las aplicaciones iOS, asociada a una tabla en la que se muestran los resultados de la búsqueda. Este comportamiento estándar se define en el controlador `UISearchDisplayController`. Este controlador es más complejo que los anteriores, por la coordinación necesaria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

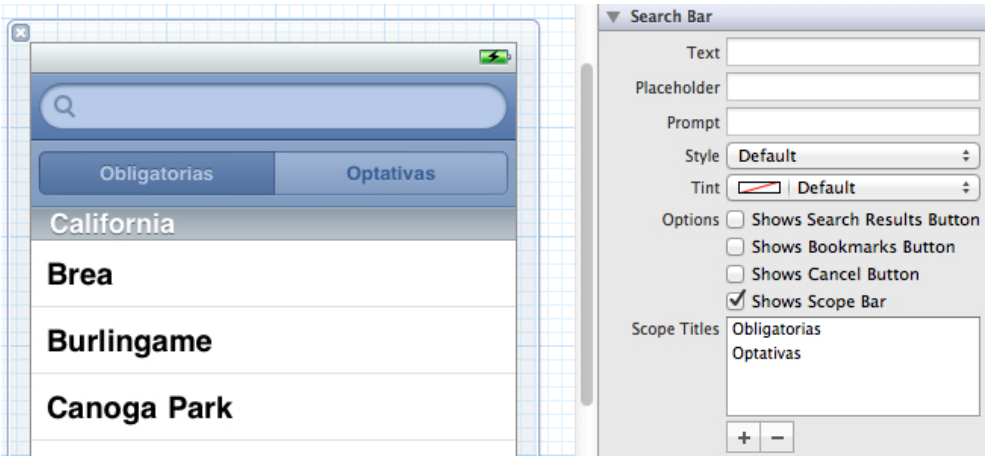
```
searchBar.delegate = self;
self.tableView.tableHeaderView = searchBar;

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`:

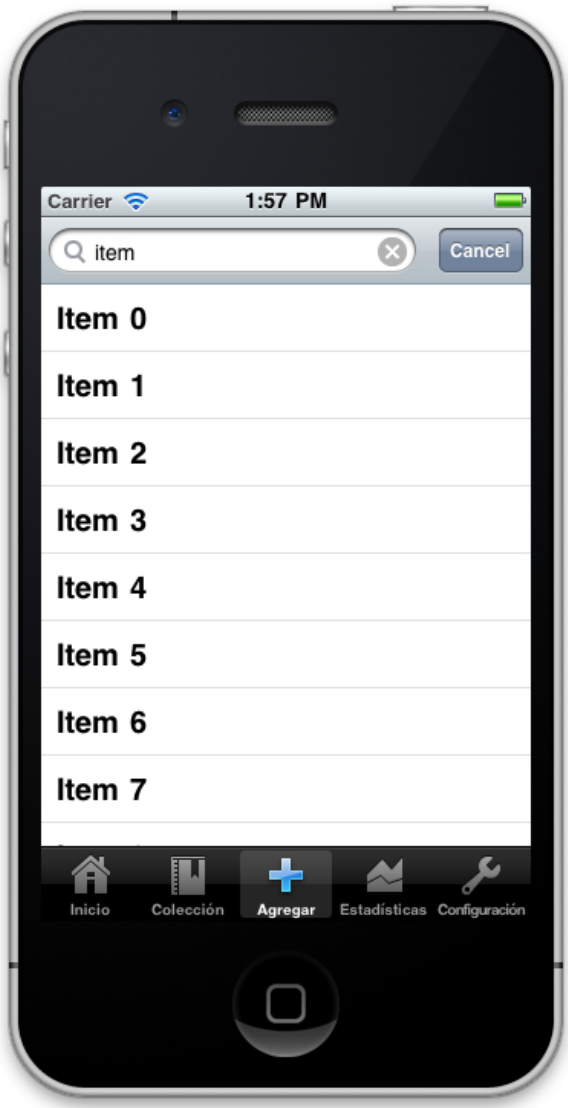
```
[self.tableView
  scrollToRowAtIndexPath: [NSIndexPath indexPathForRow:0
                                             inSection:0]
        atScrollPosition:UITableViewScrollPositionTop
                animated:NO];
```

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:`.