Recetario para iOS

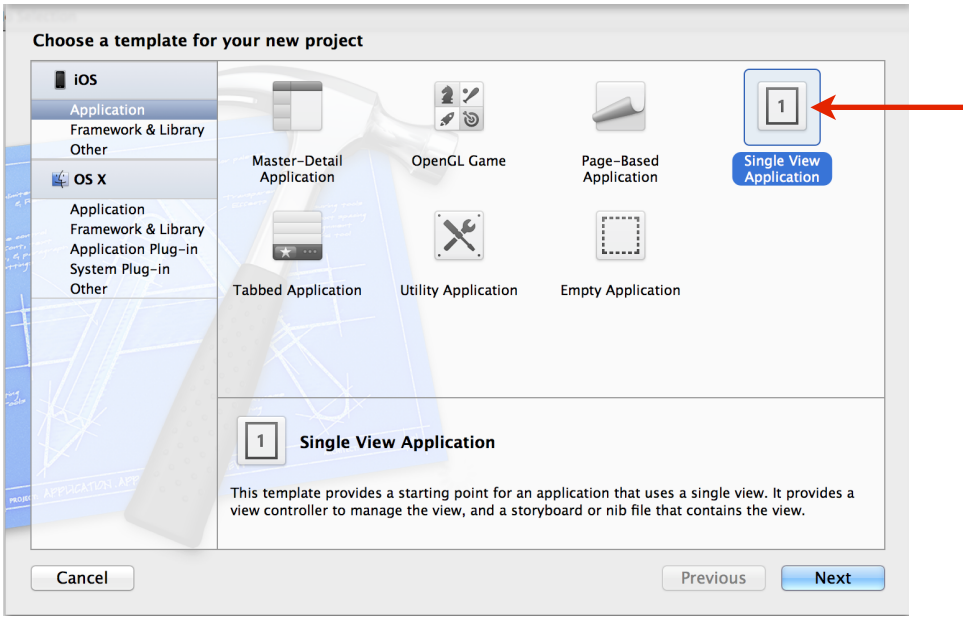
Indice

Crear una nueva aplicación

En este tutorial, se va a utilizar la versión numero 4.6.3 de Xcode y a continuación están los pasos necesarios para crear un nuevo proyecto, el cual utilizaremos para los ejemplos.

Al abrir Xcode vemos la siguiente ventana, en la cual hay que seleccionar la opción Create a new Xcode project.

Luego seleccionamos Single View Application

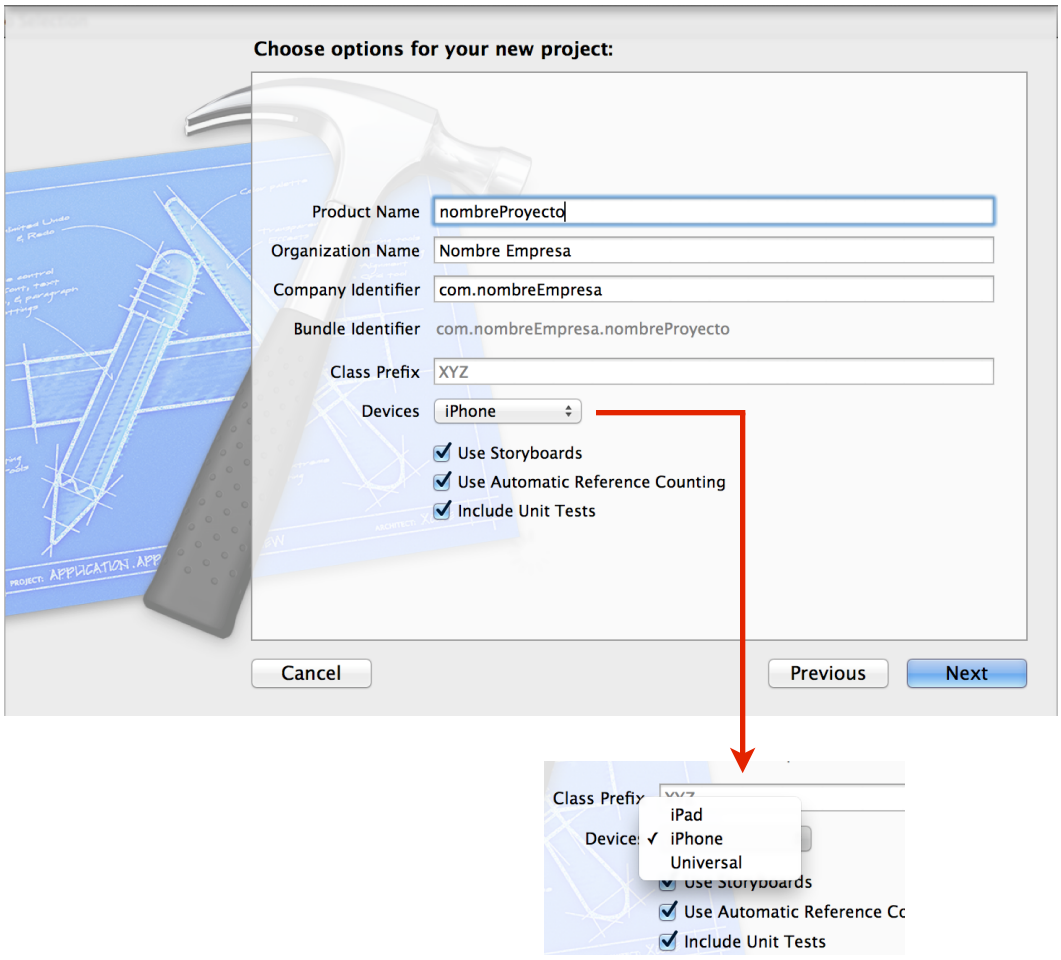


En la siguiente pantalla podemos seleccionar las opciones generales del proyecto, entre las que tenemos Product Name, que será el nombre de la aplicación una vez la publiquemos, Organization Name, que es el nombre de la empresa o persona que la desarrolla y Company Identifier, el cual por lo general tiene el formato de una URL al inverso con el nombre de la compañía, por ejemplo: com.nombreEmpresa.

El campo Class Prefix es opcional y lo podemos utilizar si queremos que todas nuestras clases tengan un prefijo predeterminado.

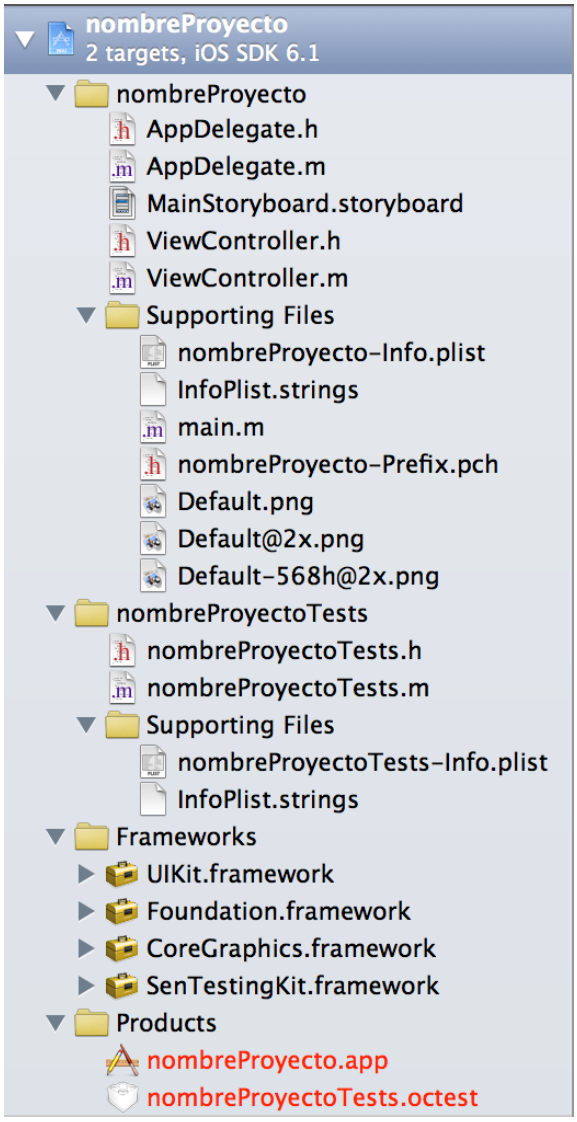
El menú desplegable Devices nos permite determinar si la aplicación va a estar desarrollada para iPhone, iPad o ambos (Universal).

Finalmente tenemos tres campos seleccionables que para funciones de este tutorial vamos a seleccionarlos, el primero nos permite hacer uso de los Storyboards para diseñar interfaces, el segundo activa la referencia automática de objetos, mejorando así el desempeño de nuestra aplicación y el tercero nos crea un esqueleto para las pruebas unitarias, que nos será de gran ayuda para probar nuestra aplicación.

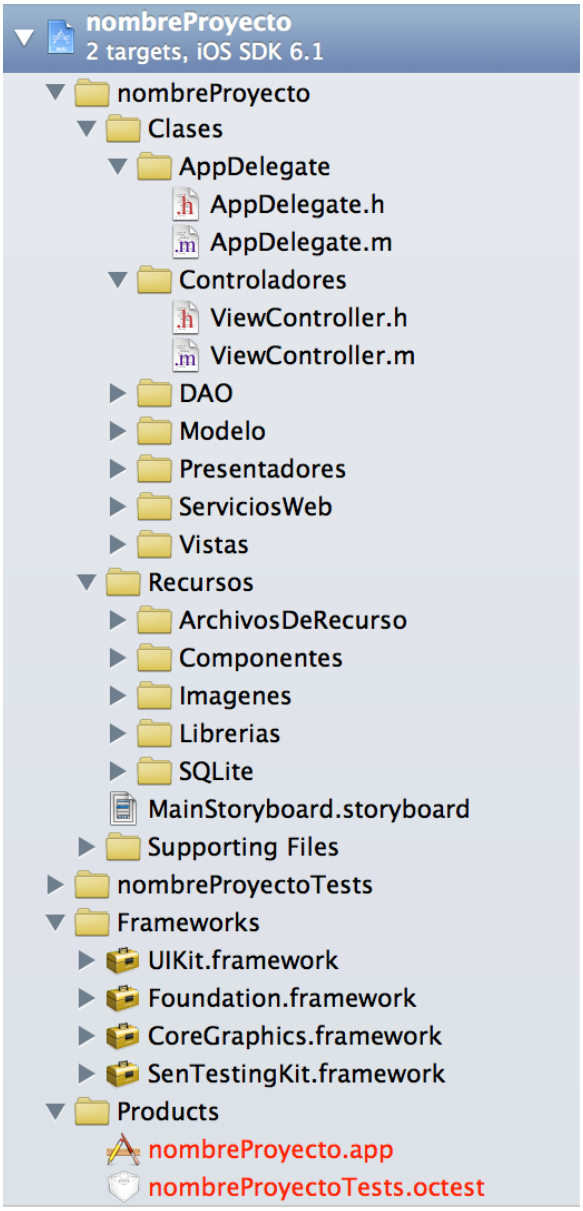


Una vez concluidos estos pasos tenemos la estructura base para un proyecto, pero esta arquitectura no está completamente estructurada y necesitamos crear algunas carpetas para ordenar y almacenar el código que vamos a generar en este tutorial, al igual que el código auto generado.

Arquitectura Original



Arquitectura Deseada



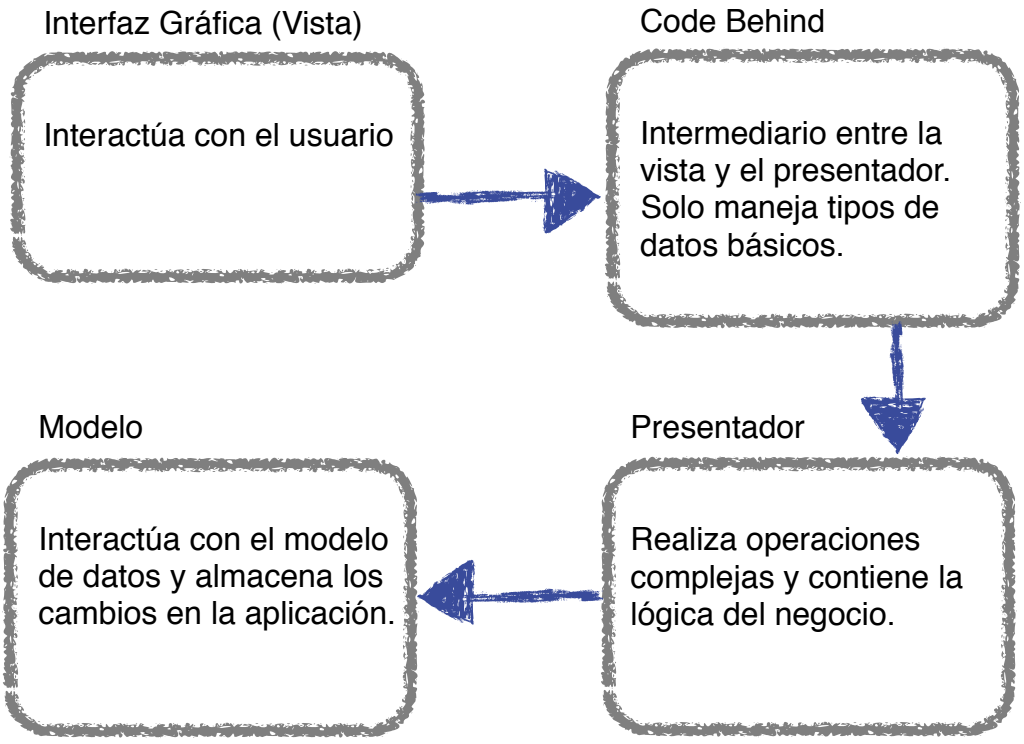
Es importante destacar que ésta estructura es solo lógica, si se desea tener el mismo orden en las carpetas físicas (es lo recomendable), es necesario crear las mismas carpetas manualmente en la ruta del proyecto.

Arquitectura de una aplicación

Anteriormente creamos un número de carpetas para organizar nuestro código y hay varias carpetas que deben su existencia a la utilización del patrón de diseño Modelo Vista Presentador (MVP), que es el patrón que utilizaremos.

Originalmente el patrón de diseño que debían utilizar todas las aplicaciones de iOS era el Modelo Vista Controlador (MVC), en donde el código que procesa la información está ubicado en el Controlador, y la vista (Interfaz gráfica) delega todas las funciones y procesamiento a esta capa, el problema es que con una sola capa de abstracción en las clases del ViewController, estas clases tendían a crecer mucho y a convertirse en monolíticas, por lo que para muchos las siglas MVC pasaron a significar Massive View Controller.

El patrón de diseño Modelo Vista Presentador se basa en lo siguiente:



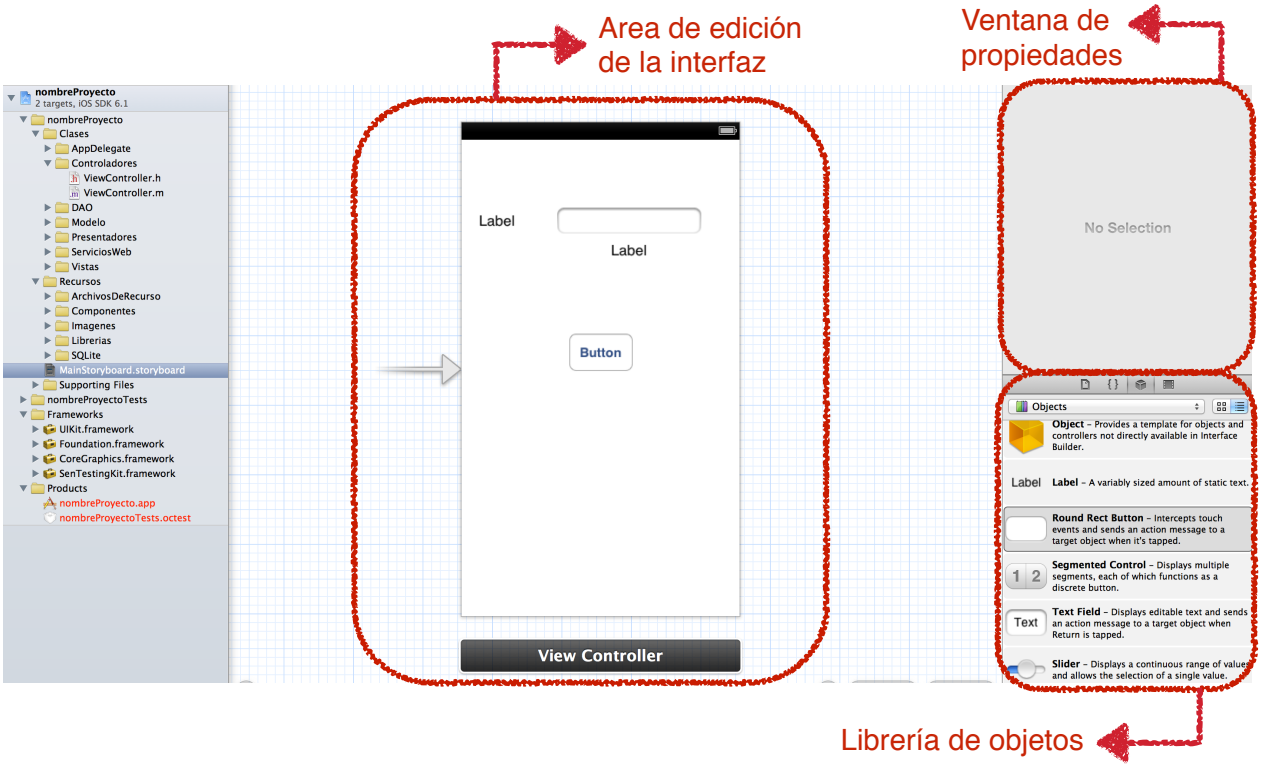
Modelo

En la arquitectura que tenemos ahora, las clases que interactúan con el modelo de datos de la aplicación se encuentran agrupadas en las carpetas DAO y Modelo. Ésta capa la estudiaremos mas adelante, por lo que ahondaremos en ella ahora.

Vista

La vista está conformada por tres elementos, el StoryBoard que contiene el diseño de la interfaz gráfica y la navegación entre las pantallas de la aplicación. Los ViewControllers que toman el papel del Code Behind de cada pantalla y las Vistas, que no son mas que una interfaz o contrato, que publica la funcionalidad de los View Controllers para que los presentadores la puedan consumir.

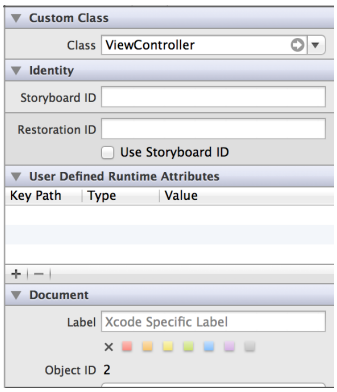
El primer componente que vamos a analizar es el Storyboard, en él podemos crear nuestras interfaces de usuario simplemente arrastrando los componentes, desde la librería de objetos que tenemos abajo a la derecha hacia el lugar donde los queremos dentro del área de edición de la interfaz. A continuación se muestra una imagen que explica la distribución de componentes dentro del Storyboard.



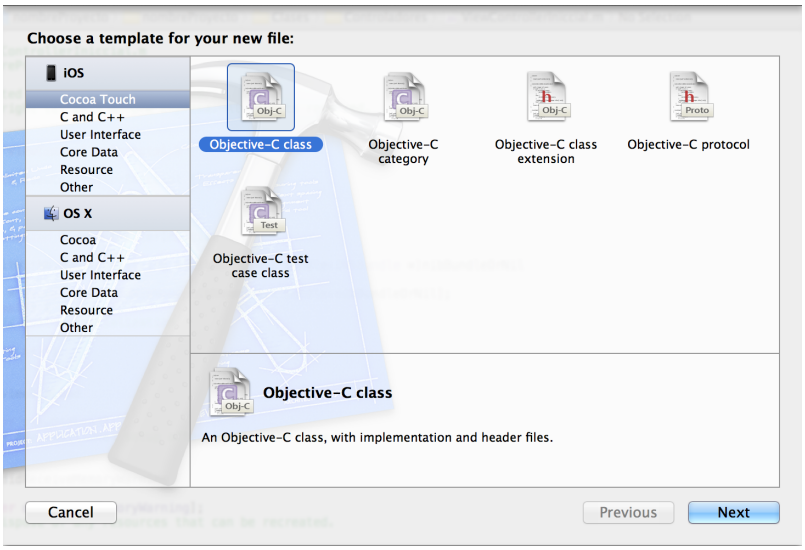
En el área de edición de la interfaz encontramos una vista previa del resultado final de nuestra aplicación, en la cual podemos ordenar y alinear nuestros componentes, aquí también podemos ver el flujo de navegación entre las pantallas, el cual esta marcado con flechas grises que van de una ventana a otra. Por ultimo tenemos recuadro negro que nos indica el nombre del ViewController que maneja los eventos de cada ventana.

La librería de objetos contiene todos los elementos que podemos agregar a nuestra aplicación y sólo basta con tomar uno y arrastrarlo con el mouse para que este forme parte de la ventana en la que lo coloquemos.

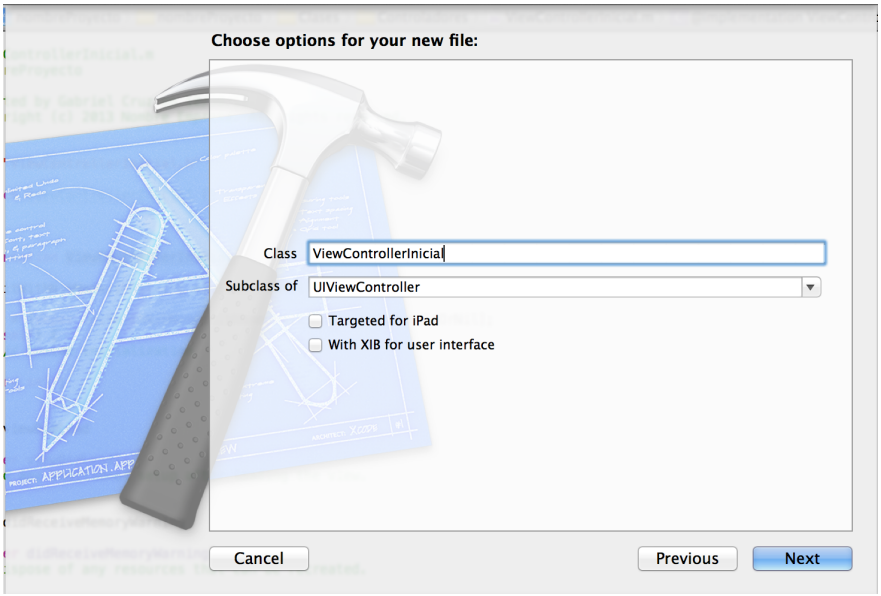
La ventana de propiedades nos muestra todas las propiedades del objeto que esté seleccionado en el StoryBoard en ese momento, por ejemplo si seleccionamos la ventana podremos ver algo como esto:



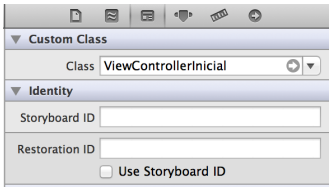
Para cada ventana que tenga nuestra aplicación es necesario crear un nuevo ViewController, lo que podemos hacer haciendo click derecho sobre la carpeta Controladores y seleccionando New File, esto nos abre una ventana donde podemos elegir el tipo de archivo a crear y en la que seleccionaremos Objective-C class:



Luego debemos colocar sus propiedades básicas, como nombre y clase de la que hereda, para nuestro caso la clase se llamará ViewControllerInicial y debe heredar de UIViewController.



Heredar de la clase UIViewController permite que nuestra clase pueda interactuar con el StoryBoard, así como también podremos ver nuestra clase en el menú desplegable de Custom Class en las propiedades de la ventana en el StoryBoard, donde debemos cambiar la selección de ViewController a nuestra clase.



Una vez que tenemos nuestra clase es hora de agregarle funcionalidad, inicialmente ella solo tendrá dos métodos, viewDidLoad y didReceiveMemoryWarning, los cuales permiten extender la funcionalidad de la clase base.

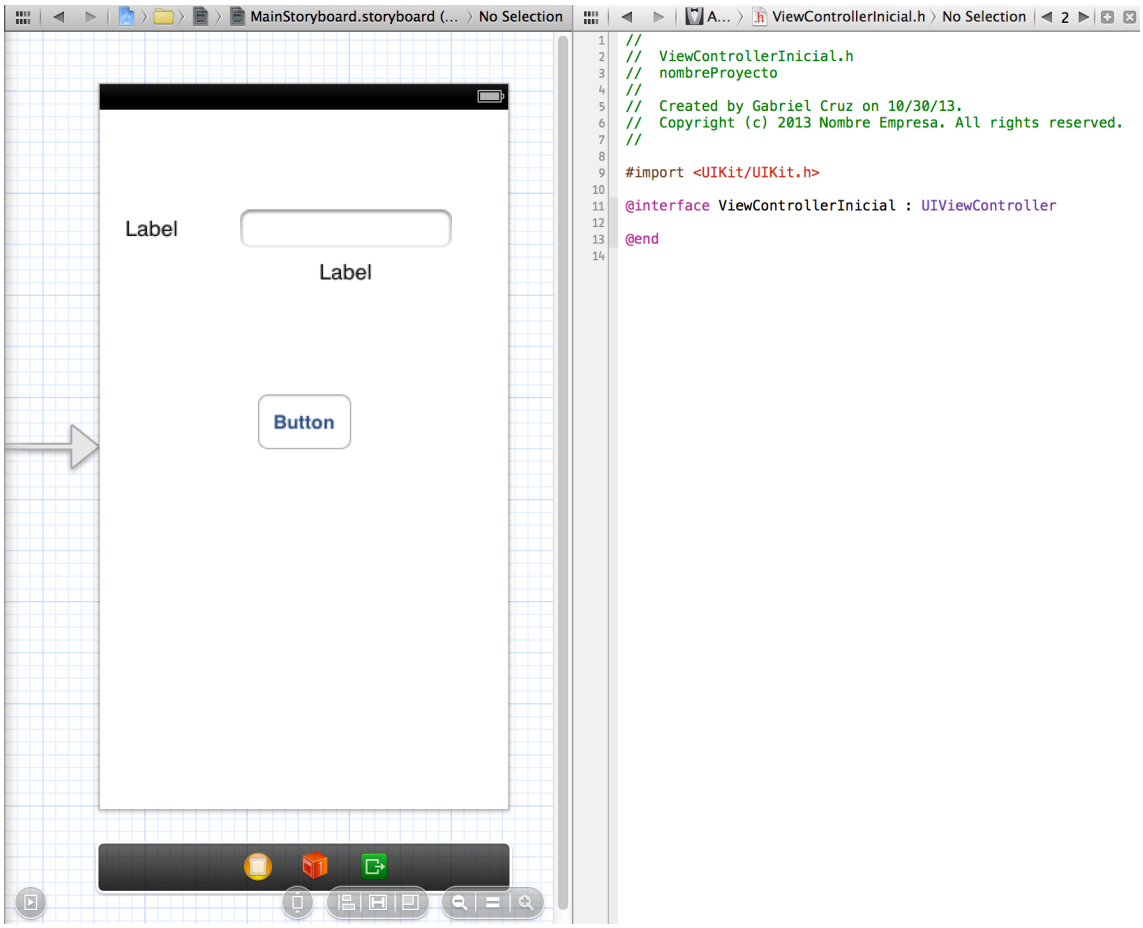
```
- (void)viewDidLoad
{
    [super viewDidLoad];
        // Do any additional setup after loading the view.
}
```

Este método es un evento que se dispara apenas la pantalla se termina de cargar y nos permite realizar una carga personalizada de la pagina, como lo es llenar algún combo box o colocar contenido en un label.

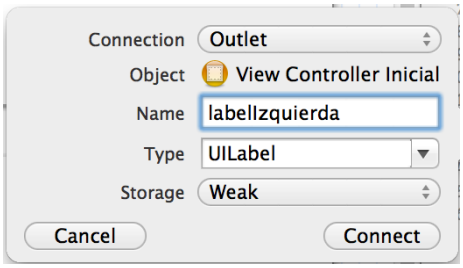
```
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
```

Este otro método es un evento igualmente, que se dispara al recibir una alerta de memoria, ya que la memoria es un recurso escaso en un teléfono, en este método podemos liberar memoria manualmente en caso que sea necesario, como por ejemplo eliminando resultados de consultas a base de datos u objetos no necesarios que ocupen mucha memoria.

Para conectar nuestra interfaz del StoryBoard con el ViewController es necesario crear Outlets o conectores, para hacer esto es recomendable utilizar la vista del Asistente, la que podemos activar presionando el botón  ubicado en la parte superior derecha de la ventana, esto nos permite una vista mas conveniente de ambas clases.



Una vez en esta vista, debemos presionar el botón Control de nuestro teclado y arrastrar el componente desde el Storyboard hacia el ViewController, lo que nos generará el Outlet necesario para interactuar con el componente mostrando la siguiente ventana en la que debemos especificar el nombre del Outlet junto a otras propiedades.



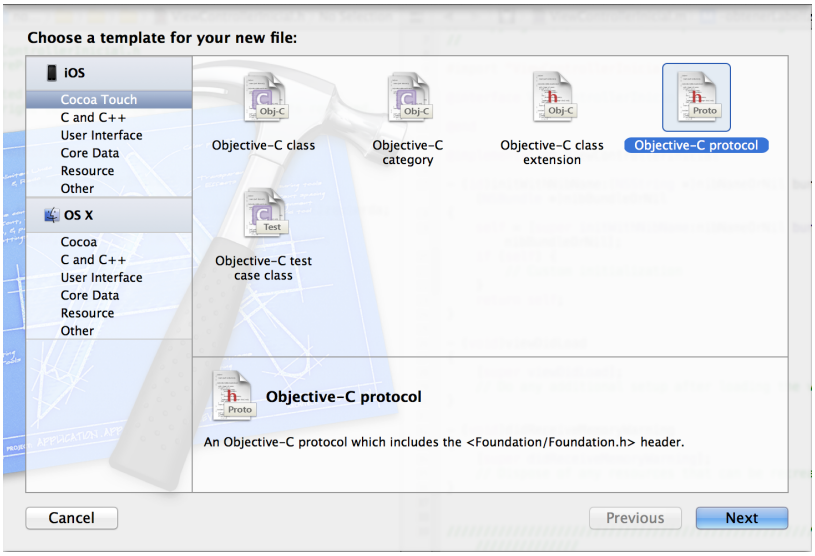
Al presionar el botón Connect, estaremos generando el siguiente código en el contrato o cabecera de nuestro ViewController:

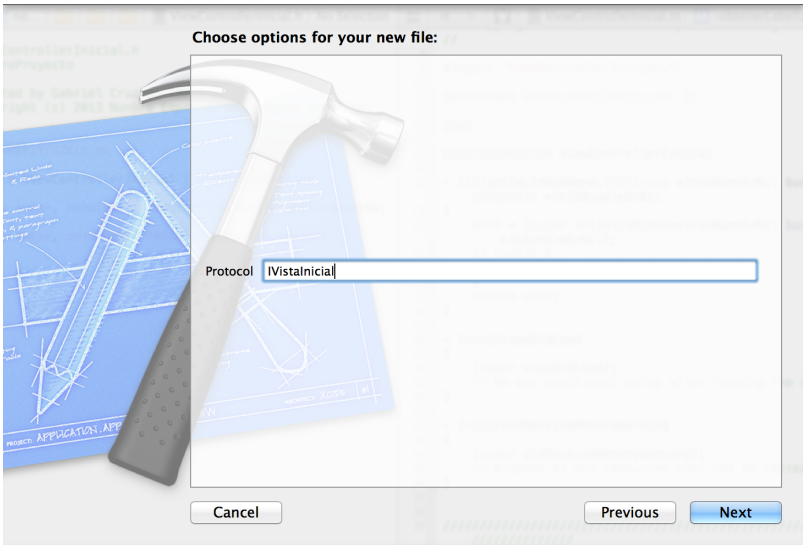
```
@property (weak, nonatomic) IBOutlet UILabel *labelIzquierda;
```

Pero ya que estamos trabajando con una arquitectura en capas, es necesario crear métodos para acceder y colocar valores en nuestros componentes y no permitir que los presentadores interactúen directamente con ellos, por lo que es necesario crear los siguientes métodos getter/setter en la implementación de nuestro ViewController.

```
-(NSString *) obtenerLabelIzquierda
{
   return self.labelIzquierda.text;
}

-(void) setearLabelIzquierda:(NSString *)contenidoLabel
{
   self.labelIzquierda.text = contenidoLabel;
}
```

Finalmente el último componente de nuestra capa Vista son las interfaces o contratos de la interfaz gráfica, las cuales publican los métodos que conforman la implementación de los ViewControllers a los Presentadores. Para crear una interfaz es necesario crear un nuevo archivo en la carpeta Vistas, pero este nuevo archivo no será un Objective-C class, sino mas bien un Objective-C protocol, el cual para este tutorial se llamará IVistaInicial





La clase IVistaInicial.h solo contiene las cabeceras de los métodos que se quieran publicar, por lo que su código es el siguiente:

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

@protocol IVistaInicial <NSObject>

-(NSString *) obtenerLabelIzquierda;

-(void) setearLabelIzquierda:(NSString *) contenidoLabel;

@end
```

Una vez listo el protocolo IVistaInicial, debe ser implementado por el ViewController de la siguiente manera:

```
#import “IVistaInicial.h"

@interface ViewControllerInicial : UIViewController <IVistaInicial>
```

Presentador

Los presentadores son las clases que contienen la lógica del negocio y las que realizan las operaciones mas complejas en toda la aplicación, ya que estas conocen el modelo de datos de la misma. Al igual que los ViewControllers, debemos contar con un presentador por pantalla, por lo que crearemos una nueva Objective-C class en la carpeta presentadores, que herede de NSObject y le daremos el nombre PresentadorInicial.

Esta clase debe tener un método constructor, el cual recibe como parámetro la interfaz o protocolo de la pantalla y lo almacena en una variable local, esto le permitirá luego poder interactuar con ella fácilmente a través de los métodos publicados en el Protocolo.

A continuación el código del PresentadorInicial, tanto de su contrato, como de su implementación:

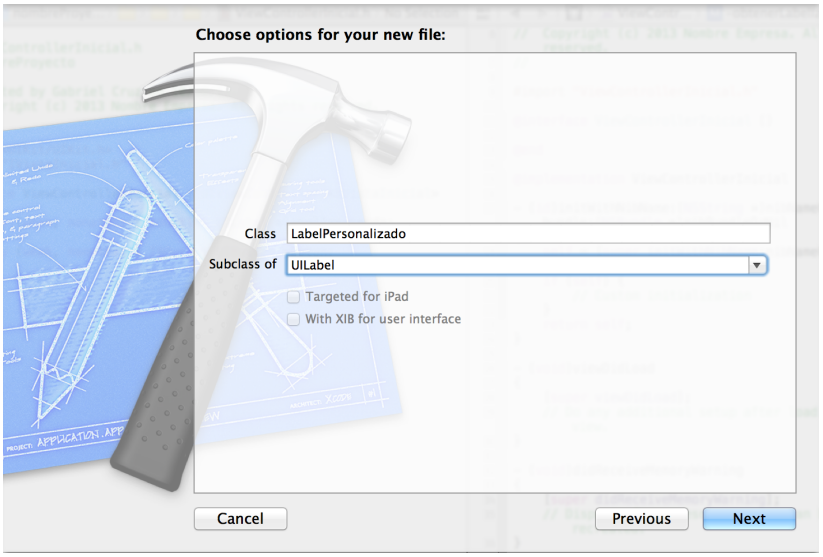
```
//  PresentadorInicial.h

#import <Foundation/Foundation.h>
#import "IVistaInicial.h"

@interface PresentadorInicial : NSObject

@property (nonatomic,strong) id<IVistaInicial> pantalla;

- (id) initWithView:(id<IVistaInicial>) pantalla;

@end
```

```
//  PresentadorInicial.m

#import "PresentadorInicial.h"

@implementation PresentadorInicial

@synthesize pantalla = _pantalla;

- (id)initWithView:(id<IVistaInicial>) pantalla
{
    if (self = [super init])
    {
        _pantalla = pantalla;
    }

    return self;
}

@end
```

Componentes personalizados

Al crear componentes personalizados, estamos creando unas estructuras que nos permiten alterar el look & feel y el comportamiento de cualquiera de los componentes que incluye el framework base de Objective-C, lo cual es muy útil en el caso que estemos desarrollando una aplicación para un cliente y éste luego de que la aplicación esta desarrollada nos pide que todos los labels ahora sean de color rojo, al tener un label personalizado, el cambio esta localizado en un solo sitio, y no es necesario modificar manualmente cada label de la aplicación.

Para crear nuestro label personalizado, debemos crear una nueva Objective-C class en la carpeta componentes, le daremos el nombre LabelPersonalizado y será una subclase de UILabel.



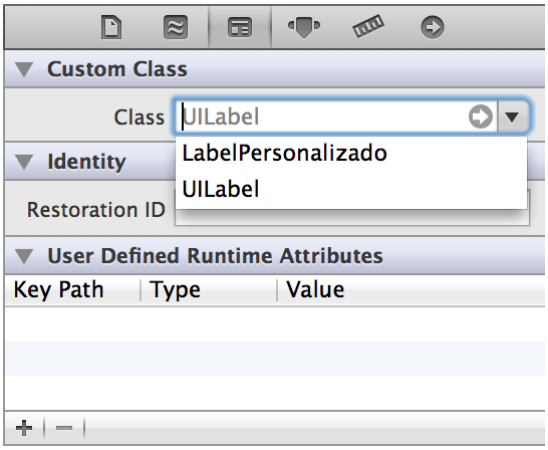
Ya que nuestra clase hereda de UILabel, en su implementación (archivo .m) tendrá el método drawRect, que nos permite modificar la manera en la que esta clase se dibuja en la interfaz gráfica, y este es el método que debemos modificar:

```
- (void)drawRect:(CGRect)rect
{
    self.textColor = [UIColor redColor];

    [super drawRect : rect];
}
```

Como se ve en el ejemplo, se esta colocando la propiedad textColor con el valor [UIColor redColor] y es importante que esto se haga antes de llamar al método drawRect de la clase padre, ya que sino los cambios no tendrán efecto.

Este tipo de personalización se puede realizar con cualquier otro componente base de la interfaz gráfica como lo son los textbox, botones, listas, celdas, etc.

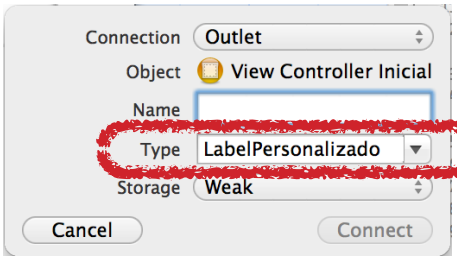
Una vez tengamos creada nuestra clase personalizada, debemos ir al StoryBoard y seleccionar el elemento que queremos que aplique esta personalización, que en este caso es un label, y en su ventana de propiedades vamos a encontrar lo siguiente:



En el campo Class seleccionamos nuestra clase personalizada y de ahora en adelante éste label se va a comportar de la manera que le digamos en nuestra clase LabelPersonalizado. Es importante que si ya hemos generado algún Outlet de este label debemos actualizarlo para que también implemente nuestra clase.

```
@property (weak, nonatomic) IBOutlet LabelPersonalizado *labelIzquierda;
```

De ahora en adelante, cualquier nuevo Outlet que generemos va a implementar nuestra clase por defecto.



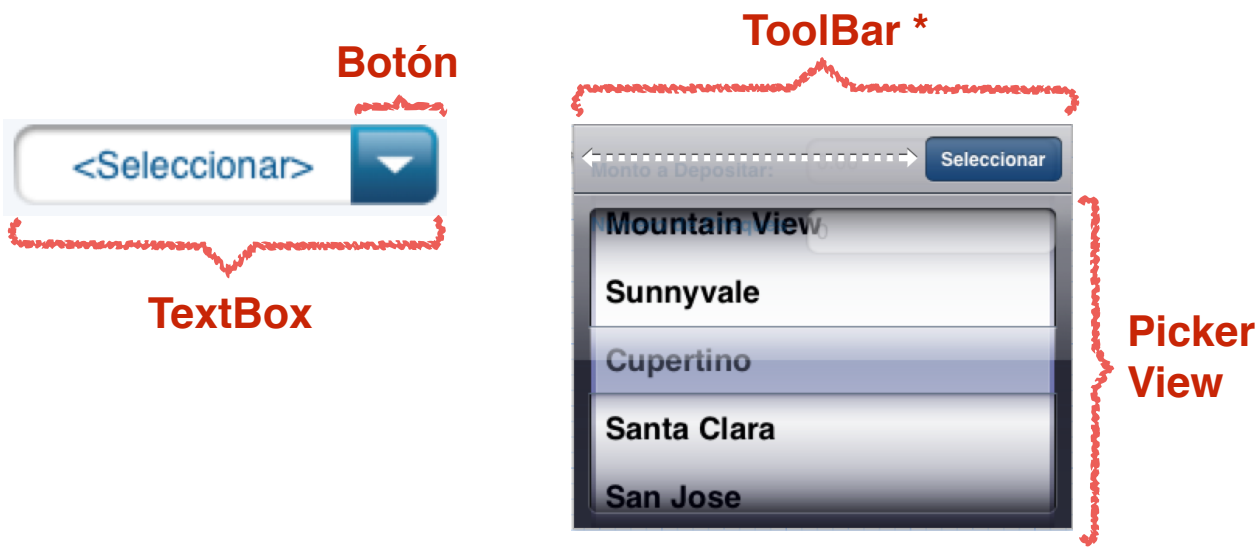
Implementación de componentes comúnmente utilizados

Ésta sección del tutorial comprende los componentes que mas utilizamos en nuestras aplicaciones, pero que su implementación en Objective-C no es tan directa (aunque no es complicada). Debido a esto, la mayor parte de lo que viene a continuación son ejemplos muy básicos de código para cada caso y su explicación.

Para implementar la funcionalidad de un componente por lo general sólo se necesita implementar su protocolo y saber que métodos hay que sobre-escribir para lograr la funcionalidad requerida.

ComboBox

Para iOS no existe directamente un control para el comboBox, pero se puede simular fácilmente utilizando 4 componentes. Necesitamos un TextBox, dos Botones, un PickerView y un Toolbar.



El ToolBar junto con su botón son opcionales, pero hace que la selección de un elemento sea mucho mas intuitiva para el usuario. Su uso depende mucho del tipo de aplicación que se esta desarrollando y las lineas de diseño general.

En términos generales, es un TextBox no seleccionable ni editable que tiene un botón (con una imagen) superpuesto, el evento del click en el botón nos muestra el PickerView con los datos cargados para poder seleccionar uno y luego el botón seleccionar nos oculta el PickerView y setea en el TextBox el valor del campo seleccionado.

Código Necesario en el ViewController:

```
// Método que oculta o muestra los componentes que conforman el PickerView
-(void) ocultarPickerView: (BOOL) esOculto
{
   [self.pickerView setHidden :esOculto];
   [self.toolbarPickerView setHidden :esOculto];
}


// Acción del botón del ComboBox
- (IBAction)mostrarPickerView:(id)sender
{
   self.pickerView.dataSource = self.presentador;
   self.pickerView.delegate = self.presentador;

   [self ocultarPickerView: NO];
}


// Acción del botón Seleccionar del Toolbar
- (IBAction)seleccionarPresionadoEnToolbar:(id)sender
{
   [self ocultarPickerView:YES];
}
```

Protocolos que hay que implementar en el presentador:

UIPickerViewDataSource
UIPickerViewDelegate

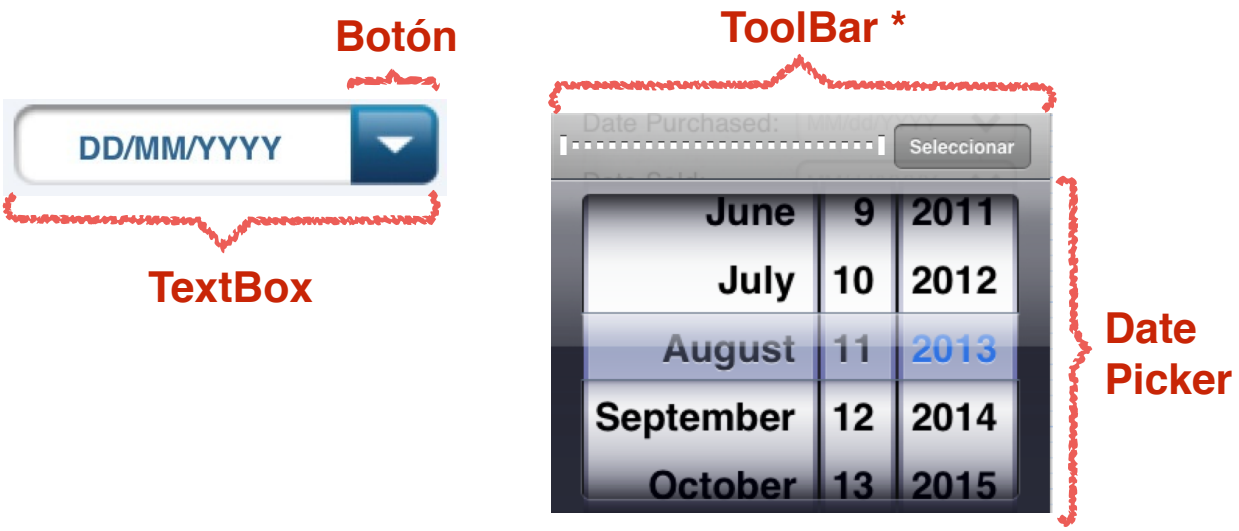
Código necesario en el presentador:

Es recomendable colocar el Pragma mark para separar el código de los diferentes delegados que se pueden implementar en cada presentador

```
////////////////////////////////////////////////////////////////////////////////////////////
#pragma mark -
#pragma mark PickerView Delegate
#pragma mark -
////////////////////////////////////////////////////////////////////////////////////////////


//  En este metodo se determina la cantidad de columnas que tendrá el PickerView
//  se suele utilizar 1 como resultado por defecto
- (NSInteger)numberOfComponentsInPickerView: (UIPickerView *)pickerView
{
   return 1;
}


//  En este metodo se determina la cantidad de filas que tendrá el PickerView
- (NSInteger)pickerView: (UIPickerView *)pickerView numberOfRowsInComponent: (NSInteger)component
{
   return self.listaQueContieneLosDatos.count;
}


//  En este metodo se determina el titulo para cada fila del PickerView
- (NSString *)pickerView: (UIPickerView *)pickerView titleForRow: (NSInteger)row forComponent:
(NSInteger)component
{
   return [self.listaQueContieneLosDatos objectAtIndex:row];
}


//  En este metodo se determina el titulo para cada fila del PickerView
-(void)pickerView:(UIPickerView *)pickerView didSelectRow: (NSInteger)row
     inComponent: (NSInteger)component
{
  [self.vista setearTextoEnComboBox: [self.listaQueContieneLosDatos objectAtIndex:row]];
}
```

DatePicker

La manera de utilizar un DatePicker es muy similar a la del ComboBox, por lo que se necesitan básicamente los mismos elementos para crearlo y el funcionamiento es similar, pero no es necesario utilizar delegados ya que el componente DatePicker se encarga automáticamente de llenarse con la data que necesita para mostrar las fechas.



Al igual que en el ComboBox, la utilización del ToolBar es opcional.

Código Necesario en el ViewController:

```
//  Metodo que oculta o muestra los componentes que conforman el DatePicker
-(void) ocultarDatePicker: (BOOL) esOculto
{
    [self.datePicker setHidden :esOculto];
    [self.toolbarDatePicker setHidden :esOculto];
    [self.setearFechaSeleccionadaEnDatePicker: [[NSDate alloc]init]];
}


//  Metodo getter del DatePicker
- (NSDate *) obtenerFechaSeleccionadaEnDatePicker
{
    return self.datePicker.date;
}
```

```
// Metodo setter del DatePicker
- (void) setearFechaSeleccionadaEnDatePicker :(NSDate *) fecha
{
    self.datePicker.date = fecha;
}

// Acción del botón del ComboBox
- (IBAction)mostrarDatePicker:(id)sender
{
    [self ocultarDatePicker:NO];
}

// Accion del botón Seleccionar del Toolbar
- (IBAction) seleccionarPresionadoEnToolbar:(id)sender
{
    [self.presentador setSelectedDateFromPicker];
    [self ocultarDatePicker:YES];
}
```

Código del presentador

```
- (NSString *) transformarDateEnString : (NSDate *) fecha
{
    NSDateFormatter *formatoFecha =[[NSDateFormatter alloc] init];
    [formatoFecha setDateFormat:@"dd-mm-yyyy"];
    return [formatoFecha stringFromDate : fecha];
}

- (NSDate *) transformarStringEnDate : (NSString *) dateString
{
    NSDate * fecha;

    if(![dateString isEqualToString:@""])
    {
        NSDateFormatter*formatoFecha =[[NSDateFormatter alloc] init];
        [formatoFecha setDateFormat:@"dd-mm-yyyy"];
        fecha = [formatoFecha dateFromString : dateString];
    }

    return fecha;
}

- (void) setSelectedDateFromPicker
{
    [self.vista setearFechaEnTextBox :[self transformarDateEnString :[self.vista
obtenerFechaSeleccionadaEnDatePicker]]];
}
```

Consideraciones importantes con los PickerView y DatePicker

Ya que ambos componentes ocupan el mismo espacio que ocupa el teclado en iOS, es importante que estos no se solapen uno al otro o incluso entre ellos si es una pantalla que contenga ambos controles. También es una buena practica hacer que un componente que ocupa gran parte de la pantalla como estos o el teclado se oculte automáticamente si el usuario interactúa con otra parte de la pantalla.

Es necesario crear un IBOutlet en el ViewController para el elemento que está mas al fondo en la pantalla (UIScrollView) e implementar el protocolo de su delegado UIScrollViewDelegate

```
@property (weak, nonatomic) IBOutlet UIScrollView *scrollViewPrincipal;
```

También es necesaria una variable que almacene cual es el TextBox seleccionado:

```
@property (nonatomic, assign) UITextField *activeTextField;
```

Luego hay que sobre-escribir los gestos que detectan cuando el teclado se muestra y oculta

```
- (void) keyboardWasShown:(NSNotification *)notification
{
    [self ocultarPickerView:YES];  // Si se va a mostrar el teclado debo ocultar los PickerView

    // 1: Obtener el tamaño del teclado
    CGSize keyboardSize = [[[notification userInfo] objectForKey:UIKeyboardFrameBeginUserInfoKey]
CGRectValue].size;
    // 2: Ajustar el borde inferior del contenido a la altura del teclado.
    UIEdgeInsets contentInsets = UIEdgeInsetsMake(0.0, 0.0, keyboardSize.height, 0.0);
    self.scrollViewPrincipal.contentInset = contentInsets;
    self.scrollViewPrincipal.scrollIndicatorInsets = contentInsets;
    // 3: Mover la pantalla hasta el campo en el que quiero escribir.
    CGRect aRect = self.view.frame;
    aRect.size.height -= keyboardSize.height;
    if (!CGRectContainsPoint(aRect, self.activeTextField.frame.origin) ) {
        CGPoint scrollPoint = CGPointMake(0.0, self.activeTextField.frame.origin.y - (keyboardSize.height-15));
        [self.scrollViewPrincipal setContentOffset:scrollPoint animated:YES];
    }
}


-(void)dismissKeyboard
{
    [self.textBox resignFirstResponder];  //  Esto debe hacerse con cada TextBox de la pantalla

    [self ocultarPickerView:YES];  //  También oculto el PickerView ya que este metodo se utiliza para ambos casos
}
```

```
- (void) keyboardWillHide: (NSNotification *)notification
{
   UIEdgeInsets contentInsets = UIEdgeInsetsZero;
   self.scrollViewPrincipal.contentInset = contentInsets;
   self.scrollViewPrincipal.scrollIndicatorInsets = contentInsets;
}
```

Finalmente es necesario configurar los gestos que sobre-escribimos para que sean reconocidos por la pantalla, el siguiente método los configura y debe ser llamado en el viewDidLoad, igual que debe ser configurado el delegado del ScrollView para que sea el mismo.

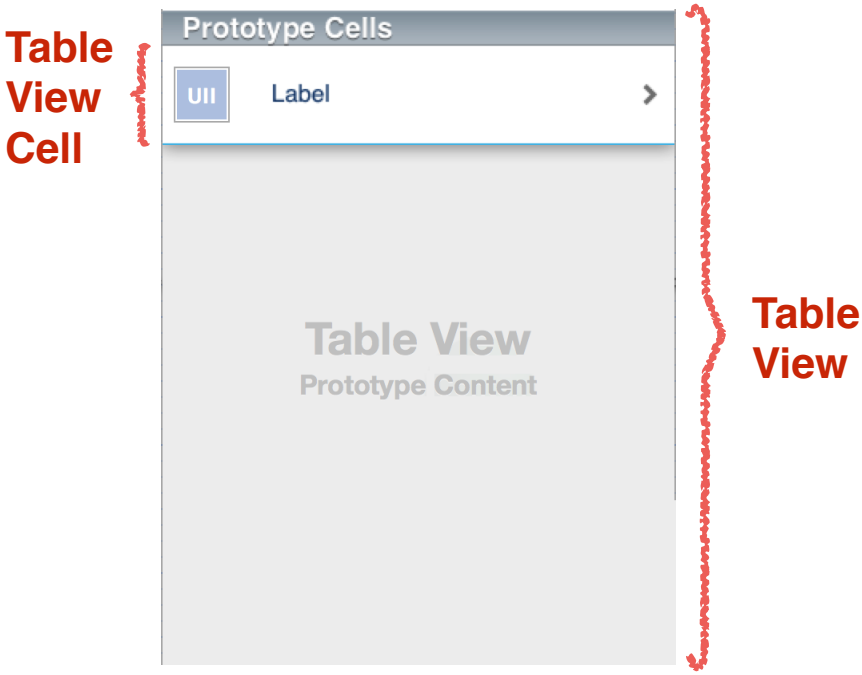
```
self.scrollViewPrincipal.delegate = self;

- (void) configurarReconocedoresDeGestos
{
        [[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(keyboardWasShown:)
name:UIKeyboardDidShowNotification object:nil];

   [[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(keyboardWillHide:)
name:UIKeyboardWillHideNotification object:nil];

   UITapGestureRecognizer *tap = [[UITapGestureRecognizer alloc] initWithTarget:self
action:@selector(dismissKeyboard)];

   [self.scrollViewPrincipal addGestureRecognizer:tap];
}
```

Es importante eliminar los observadores de gestos que programamos al salirnos de la ventana ya que estos ocupan memoria y no deberían estar activos mientras que la pantalla no se está mostrando, por lo cual es necesario sobre-escribir el método viewDidUnload

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

        [[NSNotificationCenter defaultCenter] removeObserver:self];
}
```

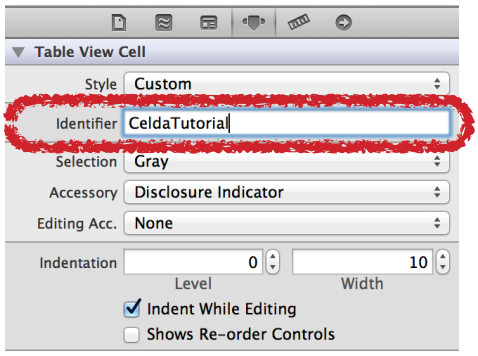
Tablas

Las tablas son componentes que nos ayudan a listar contenido y nos permiten que cada celda sea una especie de botón que nos lleva a otra pantalla, donde podemos ver toda la información detallada de ese elemento que seleccionamos. Las tablas están compuestas por un TableView, que es el contenedor principal y una serie de TableViewCells que son los elementos a mostrar.



Una de las cosas que hay que tener presente al trabajar con tablas es que cada vez que la pantalla se vaya a mostrar, ya sea por primera vez o luego de regresar de una navegación, hay que refrescar la información de la tabla porque ésta pudo haber cambiado.

Otro detalle muy importante a tener en cuenta, es que una vez tengamos nuestra celda creada en la tabla del StoryBoard debemos ir a la pestaña de propiedades y asignarle un identificador único, ya que éste será utilizado para luego poder crear las celdas que llenarán la tabla.



Código necesario en el ViewController:

```
// Método que se llama la primera vez que carga la pantalla
- (void)viewDidLoad
{
    [super viewDidLoad];

    [self instanciarPresentador];

    self.tabla.dataSource = _presentador;;
    self.tabla.delegate = _presentador;
}

// Método que se llama cada vez que la pantalla se va a mostrar luego de estar cargada
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];

    [self instanciarPresentador];

    self.tabla.dataSource = _presentador;;
    self.tabla.delegate = _presentador;
}


// Método que instancia el presentador
-(void) instanciarPresentador
{
    self.presentador = [[ClaseDelPresentador alloc]initWithView:self];
}


// Método que se llama cada vez que la aplicación va a navegar de una pantalla a otra, en este
// momento puedo capturar la navegación y enviarle a la pantalla destino el objeto que el usuario
// seleccionó en la tabla
-(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    if ([[segue identifier] isEqualToString:@"mostrarDetalle"])
    {
        DatosViewController *viewControllollerDetalles = [segue destinationViewController];

        viewControllollerDetalles.objetoQueDeseoEnviarALaOtraPantalla = [self.presentador
obtenerObjetoEnPosicion:[self.tabla indexPathForSelectedRow].row];
    }
}
```

Protocolos que hay que implementar en el presentador:

UITableViewDataSource
UITableViewDelegate

Código necesario en el presentador:

```
////////////////////////////////////////////////////////////////////////
#pragma mark -
#pragma mark Métodos del UITableViewDelegate
#pragma mark -
////////////////////////////////////////////////////////////////////////


//  En este metodo se determina la cantidad de columnas que tendrá el PickerView
//  se suele utilizar 1 como resultado por defecto
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
   return 1;
}


//  Metodo que determina la cantidad de filas que tendrá la tabla
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
   return self.listaObjetosAMostrar.count;
}


//  Metodo que crea cada celda que será incluida en la tabla
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
   static NSString *cellIdentifier = @“CeldaTutorial";  // Este es el identificador que creamos en el StoryBoard
   UIImage *imagenCelda = [UIImage imageNamed: @"ImagenCeldaTutorial.png"];

   CeldaPersonalizada *celda = [tableView dequeueReusableCellWithIdentifier:cellIdentifier];

   if(!celda)
      celda = [[CeldaPersonalizada alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:cellIdentifier];

   ObjetoAMostrar *objeto = [self.listaObjetosAMostrar objectAtIndex:indexPath.row];

   celda.label.text =  objeto.propiedadQueQuieroMostrar;
   [celda.imagen  setImage:imagenCelda];

   return celda;
}
```

```
//  Metodo que le agrega la funcionalidad de mostrara un botón para eliminar al desplazar lateralmente una celda
// (swipe lateral)
-(NSString *) tableView:(UITableView *)tableView titleForDeleteConfirmationButtonForRowAtIndexPath:
(NSIndexPath *)indexPath
{
   return @"Eliminar";
}


//  Metodo que captura el evento de seleccionar el botón de eliminar en la lista con el Swipe y lo elimina
- (void)tableView:(UITableView *)tableView commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath
{
   if (editingStyle == UITableViewCellEditingStyleDelete)
   {
      NSError *error;

      //  Se elimina el objeto de la Base de Datos
      Objeto *objetoParaEliminar = [self obtenerObjetoEnPosicion:[tableView indexPathForSelectedRow].row];
      [daoObjeto eliminarObjeto:objetoParaEliminar :&error];

      if(error == nil)
      {
         // Actualizo la tabla con la nueva lista de objetos
         [self.listaObjetosAMostrar removeObjectAtIndex:indexPath.row];
         [tableView reloadData];
      }
   }
}
```

Mensajes de notificación

En esta sección veremos los diferentes tipos de notificaciones que podemos tener en iOS, aunque principalmente se dividen en dos grupos, las Alerts que se muestran en estilo de popup y los ActionSheet que se muestran como un mensaje que se desliza desde la parte inferior de la pantalla.

Alerts

Los alerts están diseñados para notificar al usuario de algún evento que está ocurriendo en la aplicación o en el dispositivo en forma de un popup. Por lo general los Alerts son inesperados, porque ellos notifican al usuario sobre un problema o cambio de la situación actual que puede requerir que el usuario tome alguna acción.

Una buena practica es tener un método público en el ViewController Base, para generar notificaciones genéricas y que pueda ser llamado por todas las pantallas, para así controlar la manera en la que se muestran todas las alertas:

Protocolos que hay que implementar en el ViewController:

UIAlertViewDelegate

Código necesario en el ViewController:

```
// Método genérico para mostrar notificaciones
// Si el delegate es nil, el alert se cierra y no desencadena mas acciones, en cambio si el delegate es self
// al presionar un botón se va a invocar al método didDismissWithButtonIndex
-(void) mostrarMensajeInformativoConTitulo: (NSString *) titulo ConMensaje:(NSString *) mensaje
yNavegarAtras:(BOOL) navegarAtras
{
   UIAlertView *alerta;

   if(navegarAtras)
      alerta = [[UIAlertView alloc] initWithTitle:titulo
                                    message:mensaje
                                    delegate:self
                                    cancelButtonTitle:@"Aceptar"
                                    otherButtonTitles:nil];
   else
      alerta = [[UIAlertView alloc] initWithTitle:titulo
                                    message:mensaje
                                    delegate:nil
                                    cancelButtonTitle:@"Aceptar"
                                    otherButtonTitles:nil];

   [alerta show];
}


// Método que captura el evento del botón presionado en el Alert
- (void)alertView:(UIAlertView *)alertView didDismissWithButtonIndex:(NSInteger)buttonIndex
{
   [self navegarPaginaAnterior];
}


// Método que invoca el motor de navegación del teléfono y lo hace regresar a la pantalla anterior
-(void) navegarPaginaAnterior
{
   [self.navigationController popViewControllerAnimated:YES];
}
```

También es posible utilizar los Alerts para mostrar mensajes de espera mientras alguna acción que toma mucho tiempo se ejecuta. Es necesario contar con una variable global de tipo Alert en el ViewController Base, para que la alerta pueda ser activada y desactivada por cualquier pantalla.


```
@property UIAlertView *alertaCargando;


// Método que instancia la alerta con un mensaje y la muestra
- (void) mostrarAlertaCargandoConMensaje :(NSString *) mensaje
{
    self.alertaCargando = [[UIAlertView alloc] initWithTitle: mensaje
                                                     message:nil
                                                     delegate:nil
                                                     cancelButtonTitle:nil
                                                     otherButtonTitles: nil];
    [self.alertaCargando show];

    UIActivityIndicatorView *indicator = [[UIActivityIndicatorView alloc]
initWithActivityIndicatorStyle:UIActivityIndicatorViewStyleWhiteLarge];

    // Se ajusta el indicador para que esté a unos pocos pixeles del borde inferior de la alerta
    indicator.center = CGPointMake(self.alertaCargando.bounds.size.width / 2,
self.alertaCargando.bounds.size.height - 50);
    [indicator startAnimating];

    [self.alertaCargando addSubview:indicator];
}


// Método que oculta la alerta de carga
- (void) ocultarAlertaCargando
{
    if(self.alertaCargando != nil)
        [self.alertaCargando dismissWithClickedButtonIndex:0 animated:YES];
}
```

Action Sheet

Las Action Sheets le dan al usuario opciones adicionales relacionadas a la acción que están realizando actualmente. Los usuarios esperan la aparición de un action sheet cuando presionan un botón que podría iniciar una acción destructiva (como borrar un registro) o también cuando la acción que desencadena el botón puede ser completada de diferentes maneras.

Protocolos que hay que implementar:

UIActionSheetDelegate

Código necesario:

```
//    El delegate no siempre es self, solo debe ser la clase que implementa el protocolo UIActionSheetDelegate
UIActionSheet *sheet = [[UIActionSheet alloc]
                        initWithTitle:@"Que desea hacer con el objeto?"
                        delegate:self
                        cancelButtonTitle:@"Cancelar"
                        destructiveButtonTitle:@"Eliminar"
                        otherButtonTitles:@"Guardar",@"Modificar",nil];

[sheet showInView:self.view];
```

```
//    Código que captura el evento de presionar un botón en el ActionSheet
- (void)actionSheet:(UIActionSheet *)actionSheet didDismissWithButtonIndex:(NSInteger)buttonIndex
{
    if (buttonIndex == actionSheet.destructiveButtonIndex)
    {
        //    Código del botón eliminar
    }
    else if (buttonIndex == actionSheet.cancelButtonIndex)
    {
        //    Código del botón cancelar
    }
    else if (buttonIndex == actionSheet.firstOtherButtonIndex)
    {
         //    Código del primer botón de la lista de otherButtonTitles
    }
}
```

Archivos de recursos

Una manera sencilla de crear archivos de recursos es crear un protocolo, para luego definir en el constantes de strings y enumerables que pueden ser utilizadas a lo largo de toda la aplicación. A continuación un ejemplo de un archivo de recurso para una ser utilizado en el DAO.

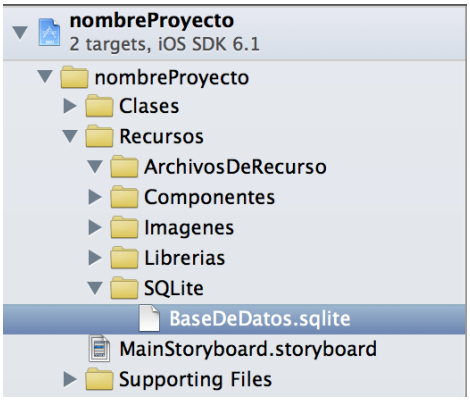
```
//
//  CodigosDeErrorDao.h
//  Recetario para iOS
//
//  Created by Gabriel Cruz on 11/1/13.
//  Copyright Gabriel Cruz. All rights reserved.
//

static NSString *const dominioErroresDao = @“com.empresa.proyecto.dao";

static NSString *const mensajeErrorGeneral = @"Ha ocurrido un error consultando la base de datos";
static NSString *const mensajeErrorUbicandoBaseDeDatos = @"No se puede conectar con la BD. Error: '%s'";
static NSString *const mensajeErrorPreparandoStatement = @"Ha ocurrido un error preparando el statement.
Error:'%s'";
static NSString *const mensajeErrorEjecutandoStatement = @"Ha ocurrido un error ejecutando el statement.
Error:'%s'";
static NSString *const mensajeErrorObtenerUltimoId = @"Error obteniendo el ultimo id insertado";


enum {
    ErrorGeneral,
    ErrorUbicandoBaseDeDatos,
    ErrorPreparandoStatement,
    ErrorEjecutandoStatement,
    ErrorObteniendoUltimoIdInsertado
};
```

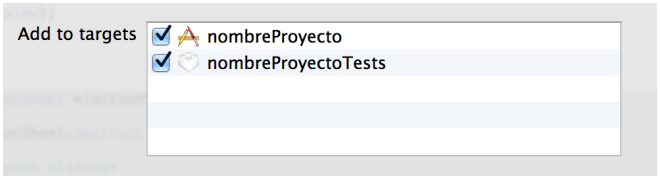
Conexión a base de datos

Las aplicaciones de iOS que necesiten guardar cualquier tipo de información pueden utilizar bases de datos SQLite para hacerlo, estas se comportan como cualquier base de datos relacional y no ocupan mucho espacio en el teléfono.

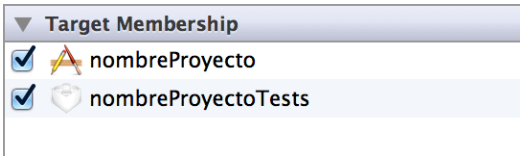
Existen dos maneras de inicializar la base de datos en el teléfono, una es tener un método que ejecute los scripts necesarios y cree las estructuras de datos, la otra es crear la base de datos inicial en un editor gráfico como SQLite Manager o Valentina Studio, para luego importar el archivo SQLite al proyecto y hacer que la aplicación copie este archivo en caso que no exista. Cada manera tiene sus pros y sus contras y cada una será la mejor opción en ciertos tipos de proyecto, en este tutorial daré todo el código necesario para la segunda opción, pero con unas pequeñas modificaciones al código se podrá aplicar la primera opción.

Primero vamos a importar el archivo .SQLite al proyecto dentro de la carpeta SQLite en los recursos, es importante seleccionar en la sección de targets nuestro proyecto y sus pruebas unitarias al momento de importarlo, ya que si no esta esa opción seleccionada, el archivo no será copiado al ejecutable y tendremos errores al momento de ejecutar nuestra aplicación.





En caso que olvidemos seleccionar el target al momento de importar el archivo, podemos hacerlo luego, solo debemos seleccionar el archivo .sqlite y marcar las opciones en la pestaña Target Membership de la sección de propiedades.



La primera clase que se ejecuta al iniciar la aplicación es AppDelegate, por lo que en ella debemos verificar si es necesario copiar el archivo de base de datos, eso lo haremos llamando al siguiente método dentro del método - (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

```
- (void) copiarBaseDeDatosSiEsNecesario
{
    NSError *error;
    BOOL existeBaseDeDatos;

    NSFileManager *fileManager = [[NSFileManager alloc] init];
    NSArray *rutas = NSSearchPathForDirectoriesInDomains(NSLibraryDirectory, NSUserDomainMask, YES);
    NSString *directorioDeDocumentos = [rutas objectAtIndex:0];
    NSString *rutaDeLaBaseDeDatos = [directorioDeDocumentos stringByAppendingPathComponent: @"/
BaseDeDatos.sqlite"];

    existeBaseDeDatos = [fileManager fileExistsAtPath:rutaDeLaBaseDeDatos];

    if (! existeBaseDeDatos)
    {
        // La base de datos no existe, así que copiamos la base de datos en la ubicación apropiada.
        NSString *defaultDBPath = [[[NSBundle mainBundle] resourcePath] stringByAppendingPathComponent: @"/
BaseDeDatos.sqlite"];
        existeBaseDeDatos = [fileManager copyItemAtPath:defaultDBPath toPath:rutaDeLaBaseDeDatos
error:&error];
    }

    if (! existeBaseDeDatos)
        NSLog(@"Ocurrió un error al crear la base de datos:'%@'.", [error localizedDescription]);
}
```

En nuestra computadora el simulador instala la aplicación en el siguiente directorio: /Users/{usuario}/Library/Application Support/iPhone Simulator/{versionSO}/Applications cada aplicación tiene un identificador único y la carpeta que la contiene tendrá este identificador, dentro de esta carpeta encontraremos otra con el nombre Documents, y en esta se encuentra el archivo de la base de datos.

Una vez tengamos la base de datos ubicada en el directorio, haremos uso de las siguientes clases para interactuar con ellas, en estas clases DAO podremos ver la implementación del archivo de recursos creado anteriormente.

Primero veremos la clase DAOBase que implementa todos los métodos genéricos, para luego ver DAOPersona, la cual personaliza el DAO para una clase.

```
//
```

```
//  DAOBase.h
//  Recetario para iOS
//
//  Created by Gabriel Cruz on 11/1/13.
//  Copyright Gabriel Cruz. All rights reserved.
//

#import <Foundation/Foundation.h>
#import "CodigosDeErrorDao.h"
#import "sqlite3.h"

@interface DAOBase : NSObject
{
   sqlite3 *bd;
}

- (NSString *) obtenerRutaBD;

- (BOOL) ejecutarSentenciaSql:(NSString *)sentenciaPorEjecutar : (NSError **) error;
- (int) obtenerIdInsertadoEnlaClase : (Class) nombreDeLaClase : (NSError **) error;

- (NSString *) convertirDateEnString : (NSDate *) date;
- (NSDate *) convertirStringEnDate : (NSString *) stringFecha;

@end



//
//  DAOBase.m
//  Recetario para iOS
//
//  Created by Gabriel Cruz on 11/1/13.
//  Copyright Gabriel Cruz. All rights reserved.
//

#import "DAOBase.h"

@implementation DAOBase

- (NSString *) obtenerRutaBD
{
   NSString *dirDocs;
   NSArray *rutas;
   NSString *rutaBD;

   rutas = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);
   dirDocs = [rutas objectAtIndex:0];

   NSFileManager *fileMgr = [NSFileManager defaultManager];

   rutaBD = [[NSString alloc] initWithString:[dirDocs stringByAppendingPathComponent: @"BaseDeDatos.sqlite"]];

   if([fileMgr fileExistsAtPath:rutaBD] == NO){
      [fileMgr copyItemAtPath:[[[NSBundle mainBundle] resourcePath] stringByAppendingPathComponent:
@"BaseDeDatos.sqlite"] toPath:rutaBD error:NULL];
```

```
    }
    return rutaBD;
}


-(BOOL)ejecutarSentenciaSql:(NSString *)sentenciaPorEjecutar : (NSError **) error
{
    BOOL respuesta = NO;
    int codigoError;
    NSDictionary *info;

    NSString *ubicacionDB = [self obtenerRutaBD];

    if(sqlite3_open_v2([ubicacionDB UTF8String], &bd,SQLITE_OPEN_READWRITE, NULL) == SQLITE_OK)
    {
        const char *sentenciaSQL = [sentenciaPorEjecutar UTF8String];
        sqlite3_stmt *sqlStatement;

        if(sqlite3_prepare_v2(bd, sentenciaSQL, -1, &sqlStatement, NULL) == SQLITE_OK)
        {
            if (sqlite3_step(sqlStatement) == SQLITE_DONE)
                respuesta = YES;
            else
            {
                info = @{NSLocalizedDescriptionKey: [NSString stringWithFormat:mensajeErrorEjecutandoStatement,,
sqlite3_errmsg(bd)]};
                codigoError = ErrorEjecutandoStatement;
            }
        }
        else
        {
            info = @{NSLocalizedDescriptionKey: [NSString stringWithFormat:mensajeErrorPreparandoStatement,
sqlite3_errmsg(bd)]};
            codigoError = ErrorPreparandoStatement;
        }
    }
    else
    {
        info = @{NSLocalizedDescriptionKey: [NSString stringWithFormat:mensajeErrorUbicandoBaseDeDatos,
sqlite3_errmsg(bd)]};
        codigoError = ErrorUbicandoBaseDeDatos;
    }

    if (!respuesta)
        if (error != NULL) *error = [NSError errorWithDomain:dominioErroresDao code:codigoError userInfo:info];

    return respuesta;
}


-(int) obtenerIdInsertadoEnlaClase : (Class) clase : (NSError **) error
{
    int respuesta = 0;
    int codigoError = ErrorObteniendoUltimoIdInsertado;
    NSDictionary *info = @{NSLocalizedDescriptionKey: mensajeErrorObtenerUltimoId};

    NSString *ubicacionDB = [self obtenerRutaBD];
```

```
    if(sqlite3_open_v2([ubicacionDB UTF8String], &bd,SQLITE_OPEN_READWRITE, NULL) == SQLITE_OK)
    {
        const char *sentenciaSQL = [[NSString stringWithFormat:@"SELECT max(id) FROM '%@'", [clase description]]
UTF8String];

        sqlite3_stmt *sqlStatement;

        if(sqlite3_prepare_v2(bd, sentenciaSQL, -1, &sqlStatement, NULL) == SQLITE_OK)
        {
            while(sqlite3_step(sqlStatement) == SQLITE_ROW)
            {
                respuesta = sqlite3_column_int(sqlStatement, 0);
            }
        }
        else
        {
            info = @{NSLocalizedDescriptionKey: [NSString stringWithFormat:mensajeErrorPreparandoStatement,
sqlite3_errmsg(bd)]};
            codigoError = ErrorPreparandoStatement;
        }
    }
    else
    {
        info = @{NSLocalizedDescriptionKey: [NSString stringWithFormat:mensajeErrorUbicandoBaseDeDatos,
sqlite3_errmsg(bd)]};
        codigoError = ErrorUbicandoBaseDeDatos;
    }

    if (respuesta == 0)
    {
        if (error != NULL) *error = [NSError errorWithDomain:dominioErroresDao code:codigoError userInfo:info];
    }
    return respuesta;
}

//  Este metodo es necesario ya que SQLite no maneja tipos de dato Date, por lo que es necesario guardarlo como un string
- (NSString *) convertirDateEnString : (NSDate *) date
{
    NSDateFormatter*dateFormat =[[NSDateFormatter alloc] init];
    [dateFormat setDateFormat:@"yyyy-MM-dd HH:mm:ss"];
    return [dateFormat stringFromDate : date];
}

//  Con este metodo recuperamos un date guardado en formato string en la base de datos y lo convertimos nuevamente a un Date
- (NSDate *) convertirStringEnDate : (NSString *) stringFecha
{
    NSDateFormatter*dateFormat =[[NSDateFormatter alloc] init];
    [dateFormat setDateFormat:@"yyyy-MM-dd HH:mm:ss"];
    return [dateFormat dateFromString : stringFecha];
}

@end
```

```
//
//  DAOPersona.m
//  Recetario para iOS
//
//  Created by Gabriel Cruz on 11/1/13.
//  Copyright Gabriel Cruz. All rights reserved.
//

#import "DAOPersona.h"

@implementation DAOPersona

-(Persona *) obtenerPersona : (int )idPersona : (NSError **) error;
{
   Persona *persona = nil;
   int codigoError = ErrorGeneral;
   NSDictionary *info;
   NSString *ubicacionDB = [super obtenerRutaBD];

   if(sqlite3_open_v2([ubicacionDB UTF8String], &bd,SQLITE_OPEN_READWRITE, NULL) == SQLITE_OK)
   {
      const char *sentenciaSQL = [[NSString stringWithFormat:@"select * from Persona where id = %i", idPersona]
UTF8String];

      sqlite3_stmt *sqlStatement;

      if(sqlite3_prepare_v2(bd, sentenciaSQL, -1, &sqlStatement, NULL) == SQLITE_OK)
      {
         persona = [[Persona alloc] init];

         while(sqlite3_step(sqlStatement) == SQLITE_ROW)
         {
            persona.identificador = sqlite3_column_int(sqlStatement, 0);
            persona.nombre = [NSString stringWithUTF8String:(char *) sqlite3_column_text(sqlStatement, 1)];
            persona.fechaNacimiento = [super convertirStringEnDate:[NSString stringWithUTF8String:(char *)
sqlite3_column_text(sqlStatement, 2)]];
         }
      }
      else
      {
         info = @{NSLocalizedDescriptionKey: [NSString stringWithFormat: mensajeErrorPreparandoStatement,
sqlite3_errmsg(bd)]};
         codigoError = ErrorPreparandoStatement;
      }
   }
   else
   {
      info = @{NSLocalizedDescriptionKey: [NSString stringWithFormat: mensajeErrorUbicandoBaseDeDatos,
sqlite3_errmsg(bd)]};
      codigoError = ErrorUbicandoBaseDeDatos;
   }

   if (persona == nil)
   {
      if (error != NULL)
         *error = [NSError errorWithDomain:dominioErroresDao code:codigoError userInfo:info];
   }
   return persona;
}
```

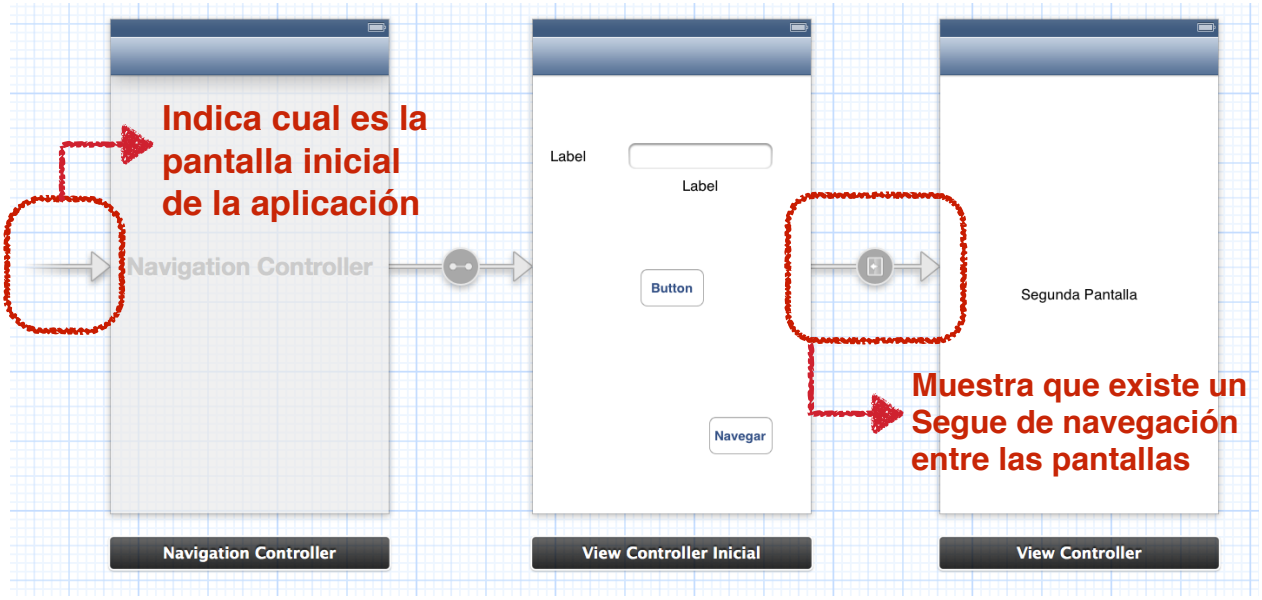
```
-(BOOL) insertarPersona: (Persona *) persona : (NSError **) error
{
   NSError *errorInterno;

   BOOL respuesta = [super ejecutarSentenciaSql:[NSString stringWithFormat:@"insert into Persona (nombre, fechaNacimiento)
values ('%@','%@')", persona.nombre, [super convertirDateEnString : persona.fechaNacimiento]]: &errorInterno];

   if(!respuesta)
   {
      *error = errorInterno;
   }
   else
   {
      if (errorInterno != nil)
      {
         persona = nil;
         *error = errorInterno;
      }
   }

   return (errorInterno == nil);
}


-(BOOL)eliminarPersona: (Persona *) persona : (NSError **) error
{
   NSError *errorInterno;

   BOOL respuesta = [super ejecutarSentenciaSql:[NSString stringWithFormat:@"delete from Persona where id = %i",
persona.identificador]: &errorInterno];

   if(!respuesta)
      *error = errorInterno;

   return respuesta;
}


-(BOOL)modificarPersona:(Persona *) persona : (NSError **) error
{
   NSError *errorInterno;

   BOOL respuesta = [super ejecutarSentenciaSql:[NSString stringWithFormat:@"UPDATE Persona SET nombre =
'%@',fechaNacimiento = '%@ WHERE id = %i", persona.nombre, [super convertirDateEnString : persona.fechaNacimiento],
persona.identificador]: &errorInterno];

   if(!respuesta)
      *error = errorInterno;

   return respuesta;
}


@end
```

Navegación entre pantallas de una aplicación

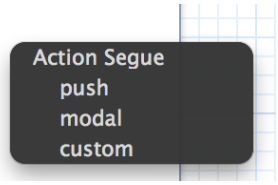
Cuando tenemos una aplicación con múltiples pantallas, debemos ser capaces de movernos entre ellas, esto lo podemos hacer de dos maneras diferentes, utilizando los Segues y mediante la barra de navegación. Ambos métodos de navegación se pueden utilizar en conjunto en una misma aplicación.

Segues

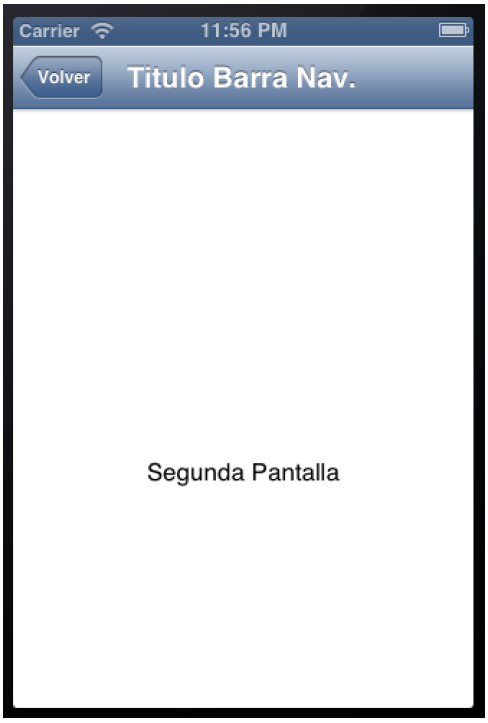
El Segue es la manera mas fácil de navegar en nuestra aplicación si estamos utilizando StoryBoards, ya que estos se agregan gráficamente y pueden ser iniciados por cualquier elemento clickeable de nuestra interfaz como botones o celdas de una tabla.

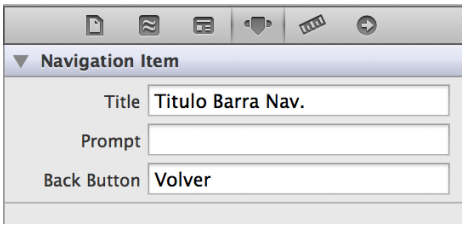


Para agregar un Segue de una pantalla a otra, sólo debemos hacer click en el elemento que queremos que inicie la navegación mientras mantenemos presionado el botón control en el teclado, y sin liberar el click, arrastramos hasta la ventana que será el destino de la navegación. Al liberar el click, nos mostrará las opciones que tenemos para la navegación:

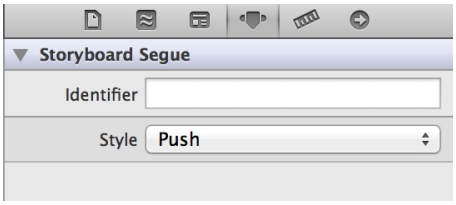


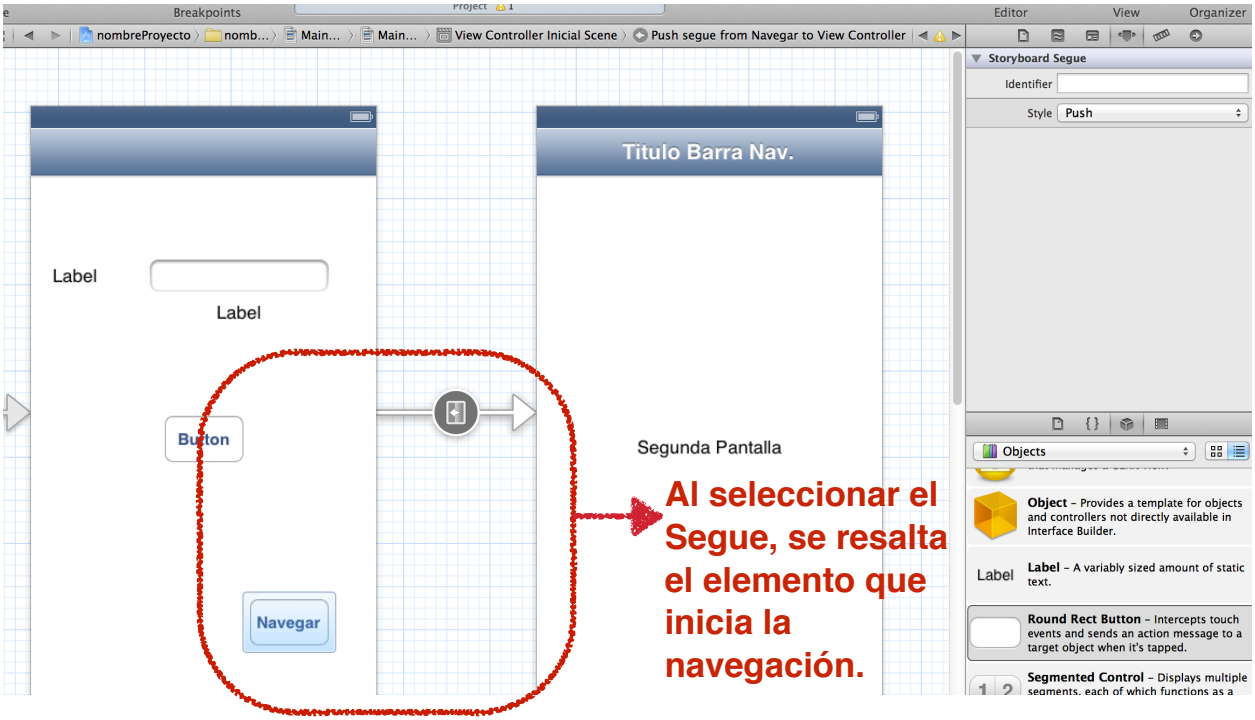
La opción push crea una navegación donde la pantalla destino se desplaza lateralmente y agrega una barra de navegación en la parte superior de la pantalla que nos permite volver a la pantalla anterior de manera intuitiva, pero es necesario que la pantalla que origina la navegación tenga una relación con un Navigation Controller, de lo contrario al intentar navegar la aplicación producirá un error.





El texto de la barra de navegación es completamente configurable en la ventana de configuración dentro del StoryBoard si seleccionamos la barra. De igual forma, si seleccionamos el Segue en el StoryBoard, le podemos asignar un identificador, el cual es necesario al momento de interceptar el segue para manualmente modificar el comportamiento de la navegación, como se pudo ver en el ejemplo de Tablas.

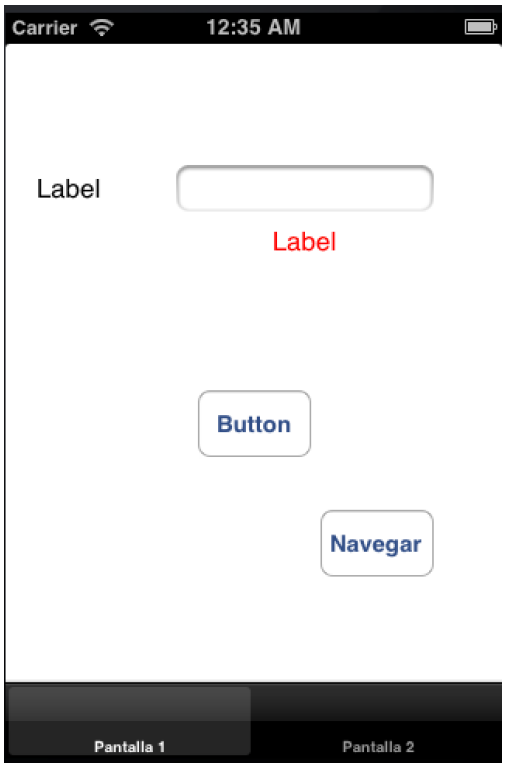




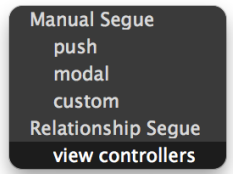
A diferencia de la opción push, la opción modal, crea una navegación donde la pantalla destino emerge de la parte inferior de la ventana y no se tiene la barra superior de navegación.

Tab Bar

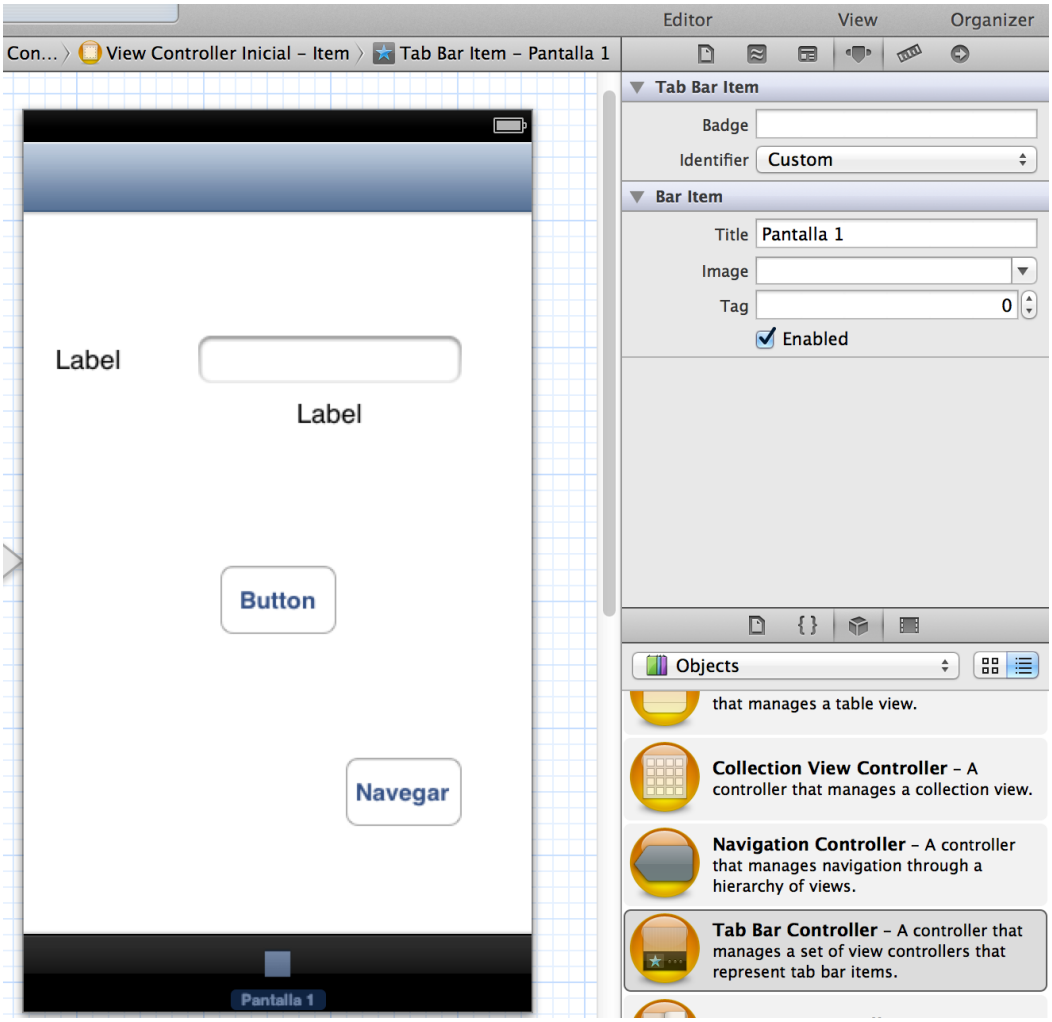
Tab Bar es la barra de navegación que se puede ver en numerosas aplicaciones de iOS en la parte inferior de la pantalla, ésta barra nos ayuda a navegar rápidamente entre las diferentes secciones de una aplicación.



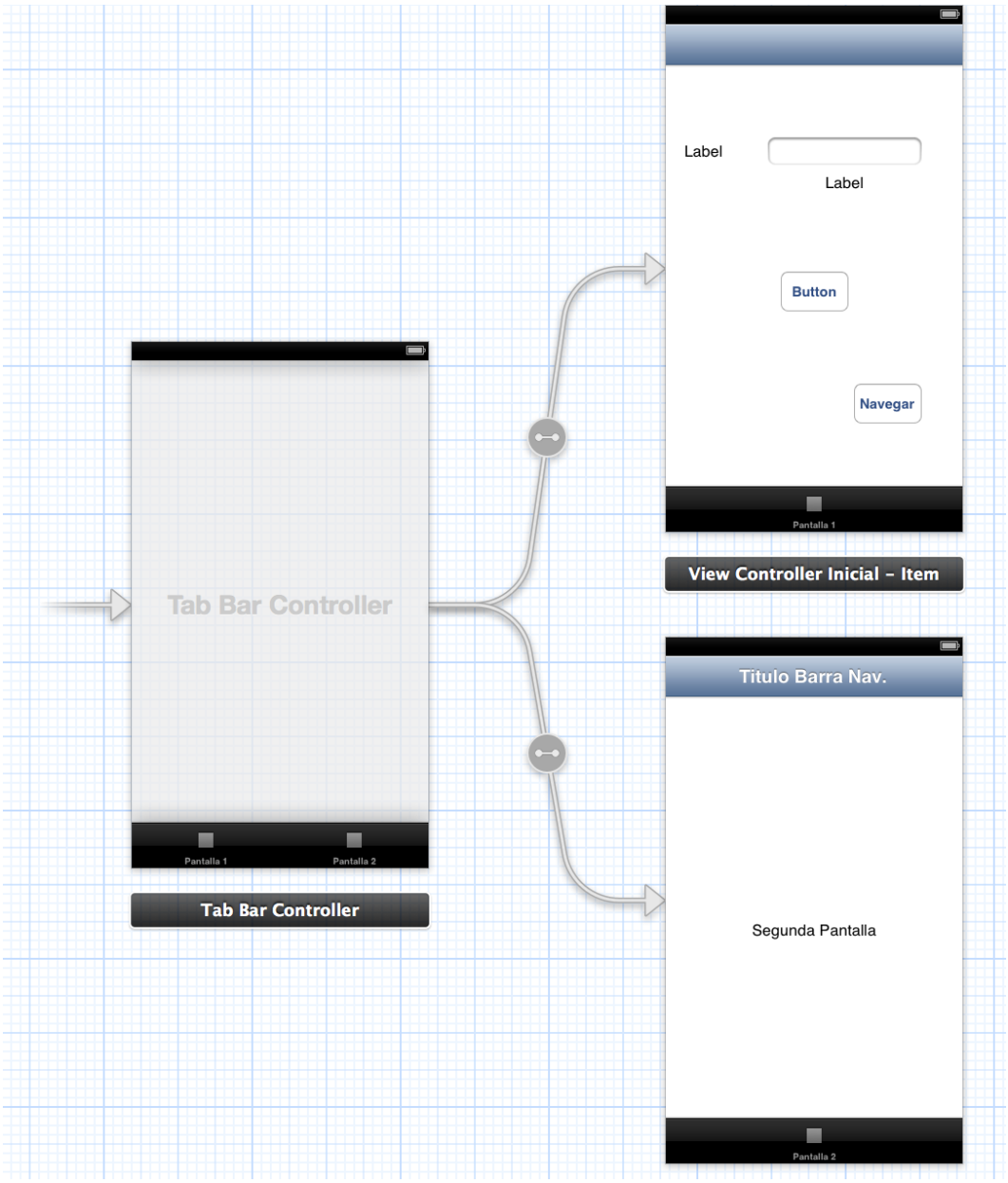
Para implementar un Tab Bar es necesario agregar al StoryBoard un Tab Bar Controller, una vez hecho esto agregar un nuevo tab es muy similar a como se agregan los Segues, sólo se debe hacer click en el Tab Bar Controller mientras se mantiene presionado el botón control del teclado y arrastrar hacia la pantalla que queremos agregar a la barra de navegación, al liberar el click debemos seleccionar la opción view controllers. Con esto hemos logrado agregar al Tab Bar Controller un nuevo tab que llevará a la pantalla que seleccionamos.



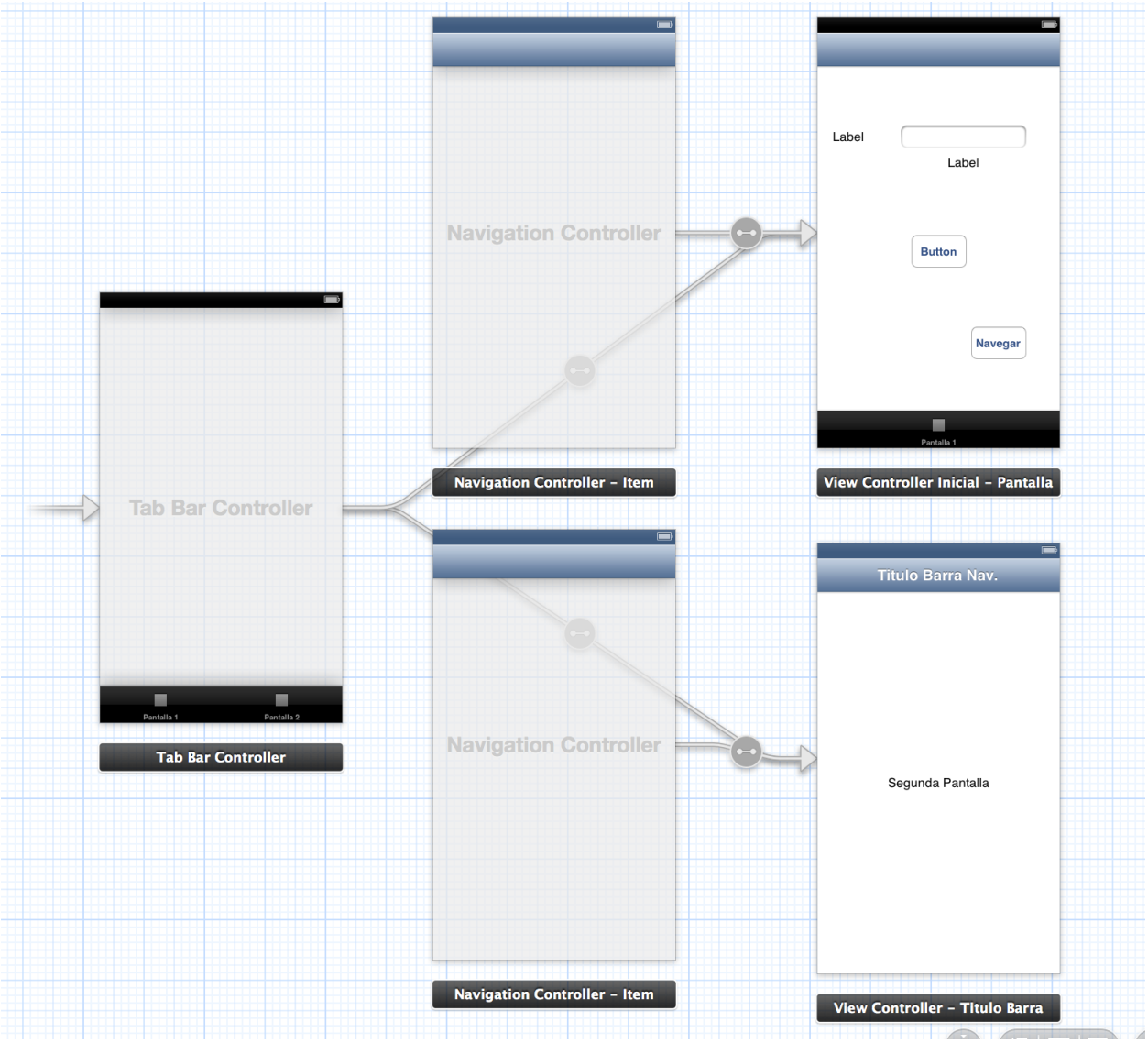
En la pantalla que hemos seleccionado, ahora podemos encontrar una barra inferior en la que podemos ver lo que será el icono y nombre del tab en la barra de navegación, si la seleccionamos podemos modificar estos campos.



Vista de un StoryBoard que utiliza Tab Bar



Vista de un StoryBoard que utiliza ambos modos de navegación



Tips adicionales

En esta ultima sección estarán pequeños segmentos de código que ayudan en funciones muy especificas

Expresiones regulares

Las expresiones regulares nos ayudan principalmente a validar el contenido de un Text Box dentro de formularios, es recomendable tener un archivo de recursos con las expresiones regulares que utilizaremos en toda la aplicación y utilizar el siguiente código para evaluarlo.

```
- (BOOL) texto: (NSString*) texto CumpleConExpresionRegular: (NSString *) expresion
{
   NSPredicate *predicado = [NSPredicate predicateWithFormat:@"SELF MATCHES %@", expresion];
   return [predicado evaluateWithObject:texto];
}
```

Expresiones regulares comunes:

```
static NSString *const RegExSoloNumeros = @"[0-9]+";
static NSString *const RegExNumerosConDosDecimales = @"^[0-9]+([.]{1}[0-9]{1,2})?$";
static NSString *const RegExCedula = @"[VEJGPvejgp]{1}\\d{6,8}";

static NSString *const RegExTarjetaCreditoVisa = @"^4[0-9]{12}([0-9]{3})?$";
static NSString *const RegExTarjetaCreditoMasterCard = @"^5[1-5][0-9]{14}$";
static NSString *const RegExTarjetaCreditoAmericanExpress = @“^3{47}[0-9]{13}$";
```

TextField personalizado

Esta clase de texField agrega varias funciones adicionales, como un color de borde diferente al seleccionarlo, limitar el numero máximo de caracteres y marcar el campo como un campo con error. Si se desea utilizar este TextField, es necesario modificar su delegate en la clase ViewController que lo implemente y colocar algo como esto:

```
self.txtNumeroCedula.delegate = self.txtNumeroCedula;
```

De esta manera cada textField es responsable de su propia funcionalidad ampliada y el código es autocontenido.

```
//
//  TextFieldPersonalizado.h
//  Recetario para iOS
//
//  Created by Gabriel Cruz on 11/1/13.
//  Copyright Gabriel Cruz. All rights reserved.
//

#import <UIKit/UIKit.h>

@interface TextFieldTrascend : UITextField <UITextFieldDelegate>

@property (nonatomic) int maxLength;

- (void) marcarCampoConError : (BOOL) tieneError;

@end
```

```
//
//  TextFieldPersonalizado.m
//  Recetario para iOS
//
//  Created by Gabriel Cruz on 11/1/13.
//  Copyright Gabriel Cruz. All rights reserved.
//

#import "TextFieldPersonalizado.h"
#import <QuartzCore/QuartzCore.h>

@interface TextFieldPersonalizado ()

@property (nonatomic) BOOL campoTieneError;

@end


@implementation TextFieldPersonalizado

static int const cornerRadius = 10;
static float const borderWidth = 1;

@synthesize maxLength = _maxLength;
@synthesize campoTieneError = _campoTieneError;

- (id)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];

    if (self) {
        _campoTieneError = NO;

    }
    return self;
}
```

```
- (void) marcarCampoConError : (BOOL) tieneError
{
    self.campoTieneError = tieneError;
    UIColor *colorBorde;

    if(tieneError)
        colorBorde = [UIColor redColor];
    else
        colorBorde = [UIColor clearColor];

    [[self layer] setBorderColor:[colorBorde CGColor]];
    [[self layer] setBorderWidth: borderWidth];
    [[self layer] setCornerRadius: cornerRadius];
}


- (BOOL)textField:(UITextField *)textField shouldChangeCharactersInRange:(NSRange)range replacementString:(NSString
*)string
{
    if(self.maxLength != 0)
    {
        if ([textField.text length] >= self.maxLength && ![string isEqualToString:@""]) {
            textField.text = [textField.text substringToIndex: self.maxLength];
            return NO;
        }
    }
    return YES;
}


- (void)textFieldDidBeginEditing:(UITextField *)textField
{
    [[self layer] setBorderColor:[[UIColor  colorWithRed:255.0f/255.0f green:153.0/255.0f blue:0.0/255.0f alpha:1.0]
CGColor]]; // HEX FF9900
    [[self layer] setBorderWidth: borderWidth];
    [[self layer] setCornerRadius: cornerRadius];
}


- (void)textFieldDidEndEditing:(UITextField *)textField
{
    UIColor *colorBorde;

    if(self.campoTieneError)
        colorBorde = [UIColor redColor];
    else
        colorBorde = [UIColor clearColor];

    [[self layer] setBorderColor:[colorBorde CGColor]];
    [[self layer] setBorderWidth: borderWidth];
    [[self layer] setCornerRadius: cornerRadius];
}

@end
```