

UNIT – 4 – Introduction to iOS

1. Introduction to IOS
2. History, Versions and Features
3. MVC Controller
4. View Controller
5. Building UI
6. Event Handling
7. Application Life Cycle
8. Tab Bars
9. Storey Boards
10. Navigation Controller
11. Push Notification
12. Database Handling
13. Debugging and Deployment
14. Publishing app in Appstore

1. Introduction to IOS

Operating system is a set of programs that manage computer hardware resources and provide common services for application software important system software in computer system. User cannot run an application program on computer without OS. Ie. Android, Mac OS X, Microsoft Windows.

Apples mobile operating system considered the foundation of the iPhone Originally designed for the iPhone but now supports iPod touch, iPad, and Apple TV It is updated just like Itune for iPods As of Oct 2011 Apple contains over 500,000 iOS applications.

2. History, Vesion and Features

2.1 History

iPhone OS was first unveiled in Jan 2007 at the Macworld Conference and Expo. Released June 2007. In June 2010, it licensed the trademark iOS (from Cisco IOS). Now it goes all the way up to iOS 5. Originally, it did not allow third-party applications, but after Feb 2008, this changed. With either 30% profit to Apple, or free with membership fee. The following figure 1 shows the features of iOS.

2.2 Features



Figure 1 Features of IOS

The power of iOS can be felt with some of the following features provided as a part of the device.

- ▶ Maps
- ▶ Siri
- ▶ Facebook and Twitter

- ▶ Multi-Touch
- ▶ Accelerometer
- ▶ GPS
- ▶ High end processor
- ▶ Camera
- ▶ Safari
- ▶ Powerful APIs
- ▶ Game center
- ▶ In-App Purchase
- ▶ Reminders

The primary applications consists of

- ▶ Safari
- ▶ Music
- ▶ Mail
- ▶ Phone, Face time video calling

The secondary applications consists of

- ▶ Camera, Camcorder
- ▶ Photos
- ▶ Calendar
- ▶ Messaging
- ▶ WeTube
- ▶ Stocks
- ▶ Map
- ▶ Clock

New Features such as

- ▶ Over 200 new features including better security
- ▶ Full Twitter integration
- ▶ New Apple iCloud enable users to synch data automatically wirelessly with no USB connection to iTunes

2.3 Environmental Setup

iOS - Xcode Installation

Step 1 – Download the latest version of Xcode from(<https://developer.apple.com/downloads/>)

Step 2 – Double click the Xcode dmg file.

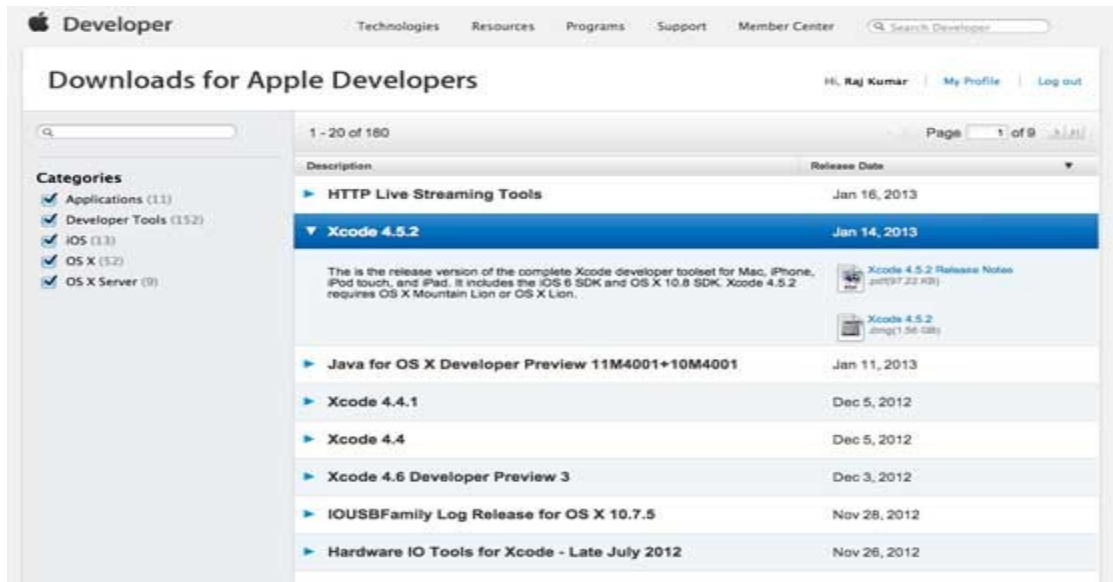
Step 3 – We will find a device mounted and opened.

Step 4 – There will be two items in the window that's displayed namely, Xcode application and the Application folder's shortcut.

Step 5 – Drag the Xcode to application and it will be copied to our applications.

Step 6 – Now Xcode will be available as a part of other applications from which we can select and run.

We also have another option of downloading Xcode from the Mac App store and then install following the step-by-step procedure given on the screen.



Interface Builder

Interface builder is the tool that enables easy creation of UI interface. We have a rich set of UI elements that is developed for use. We just have to drag and drop into our UI view. We'll learn about adding UI elements, creating outlets and actions for the UI elements in the upcoming pages.

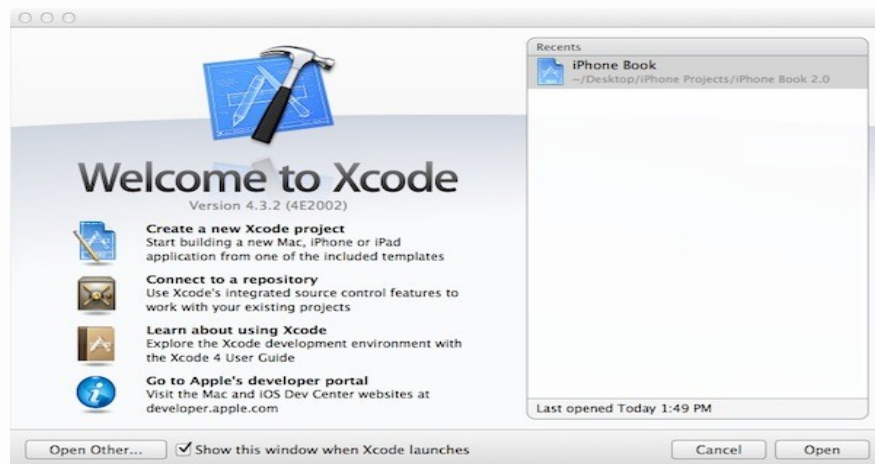
Step 1 : First, launch Xcode. If we've installed Xcode via Mac App Store, we should be able to

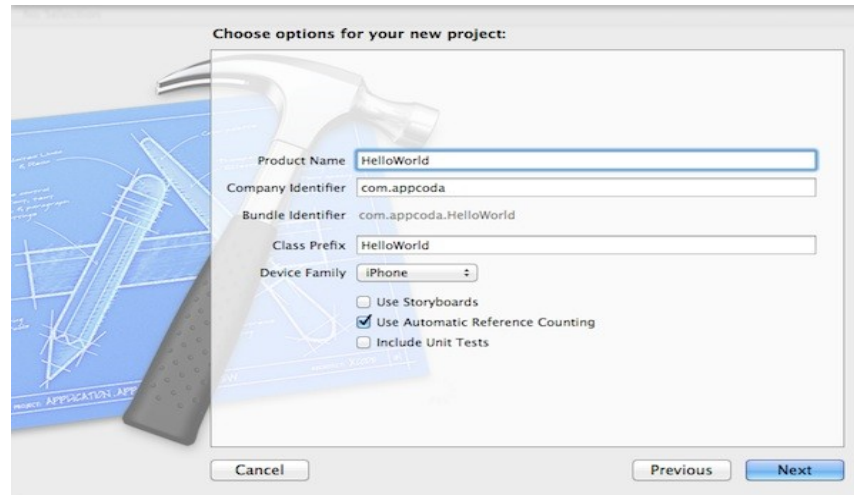
locate Xcode in the LaunchPad. Just click on the Xcode icon to start it up.

Step 2: Once launched, Xcode displays a welcome dialog. From here, choose "Create a new Xcode project" to start a new project:



Step 3: Xcode shows we various project template for selection. For our first app, choose “Single View Application” and click “Next”.

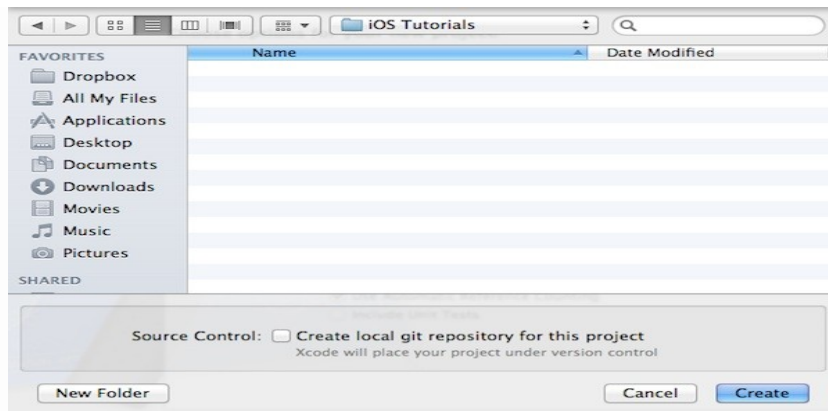




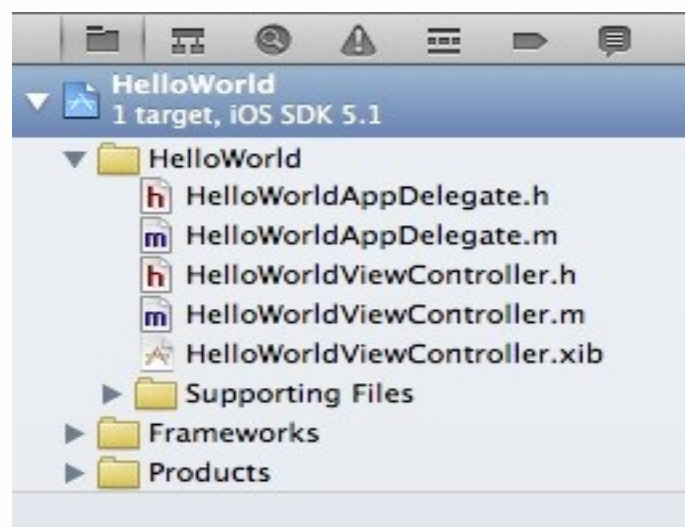
We can simply fill in the options as follows:

- **Product Name: *HelloWorld*** – This is the name of our app.
- **Company Identifier: *com.appcoda*** – It's actually the domain name written the other way round. If we have a domain, we can use our own domain name. Otherwise, we may use mine or just fill in "edu.self".
- **Class Prefix: *HelloWorld*** – Xcode uses the class prefix to name the class automatically. In future, we may choose our own prefix or even leave it blank. But for this tutorial, let's keep it simple and use "HelloWorld".
- **Device Family: *iPhone*** – Just use "iPhone" for this project.
- **Use Storyboards: *[unchecked]*** – Do not select this option. We do not need Storyboards for this simple project.
- **Use Automatic Reference Counting: *[checked]*** – By default, this should be enabled. Just leave it as it is.
- **Include Unit Tests: *[unchecked]*** – Leave this box unchecked. For now, we do not need the unit test class.

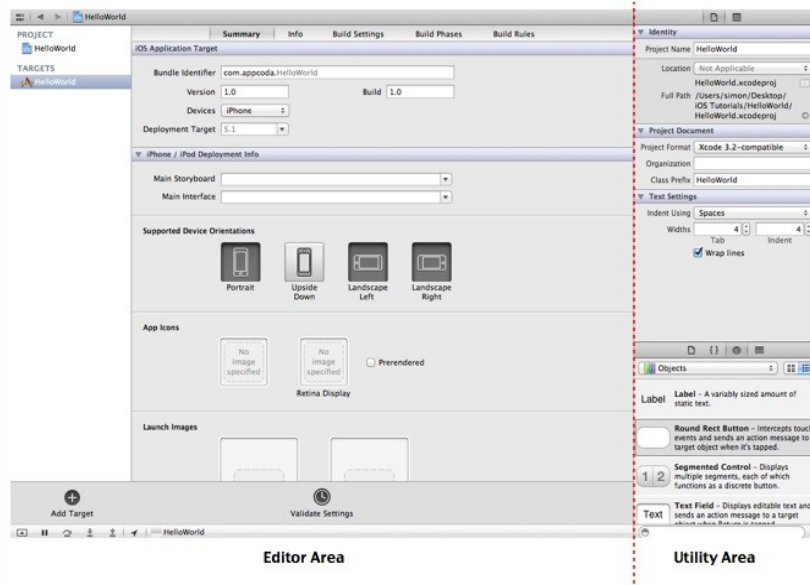
Step 4: Click "Next" to continue. Xcode then asks we where we saves the "Hello.

**Step 5: Xcode Workspace :**

On the left pane, it's the project navigator. We can find all our files under this area.

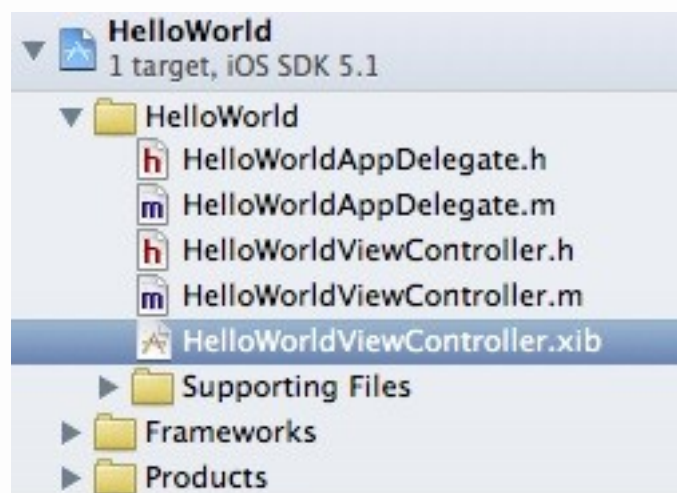


Step 6: The center part of the workspace is the editor area. We do all the editing stuffs (such as edit project setting, class file, user interface, etc) in this area depending on the type of file selected.

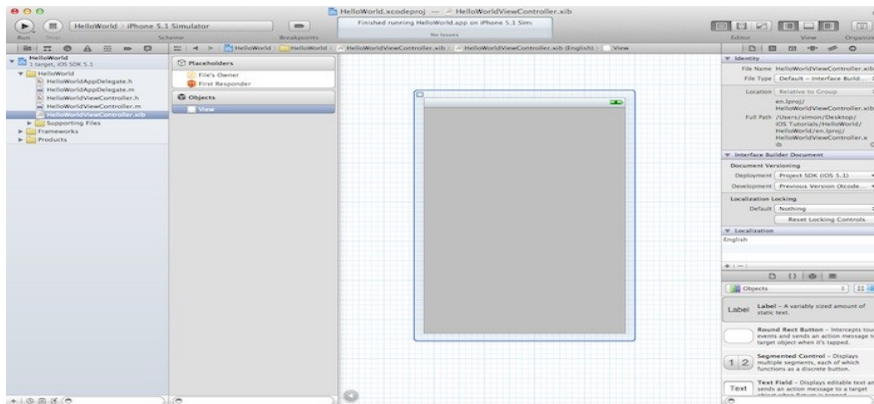


The rightmost pane is the utility area. This area displays the properties of the file and allows we to access Quick Help. If Xcode doesn't show this area, we can select the rightmost view button in the toolbar to enable it.

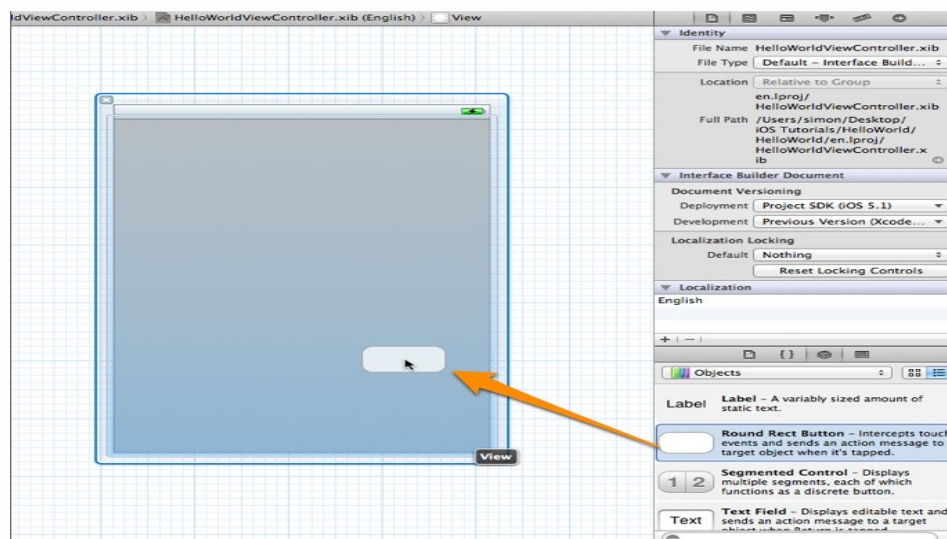
Step 7: Add the Hello World button to our app. Go back to the Project Navigator and select “HelloWorldViewController.xib”.



The editor changes to an Interface Builder and displays an empty view of our app like below.



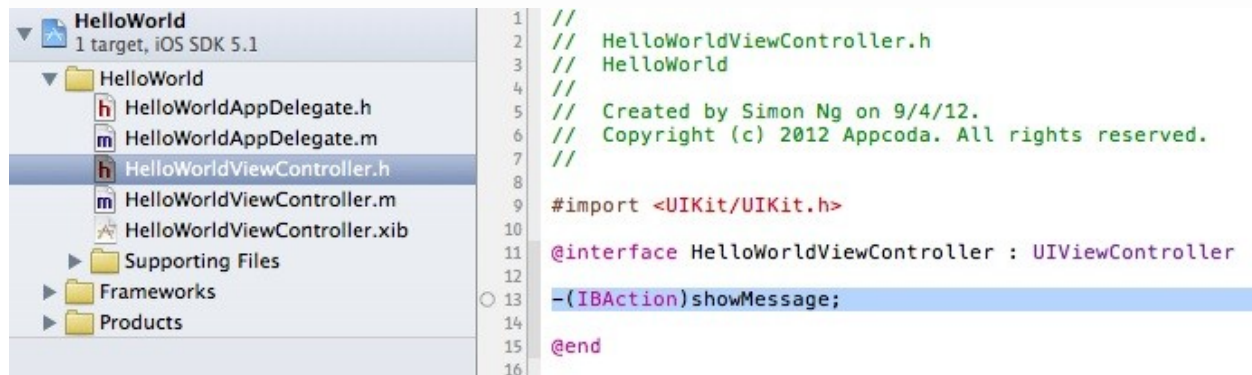
Step 8: In the lower part of the utility area, it shows the Object library. From here, we can choose any of the UI Controls, drag-and-drop it into the view. For the Hello World app, let's pick the “Round Rect Button” and drag it into the view. Try to place the button at the center of the view. To edit the label of the button, double-click it and name it “Hello World”.



Step 9: Coding the Hello World Button

In the Project Navigator, select the “HelloWorldViewController.h”. The editor area now displays the source code of the selected file. Add the following line of code before the “@endline”.

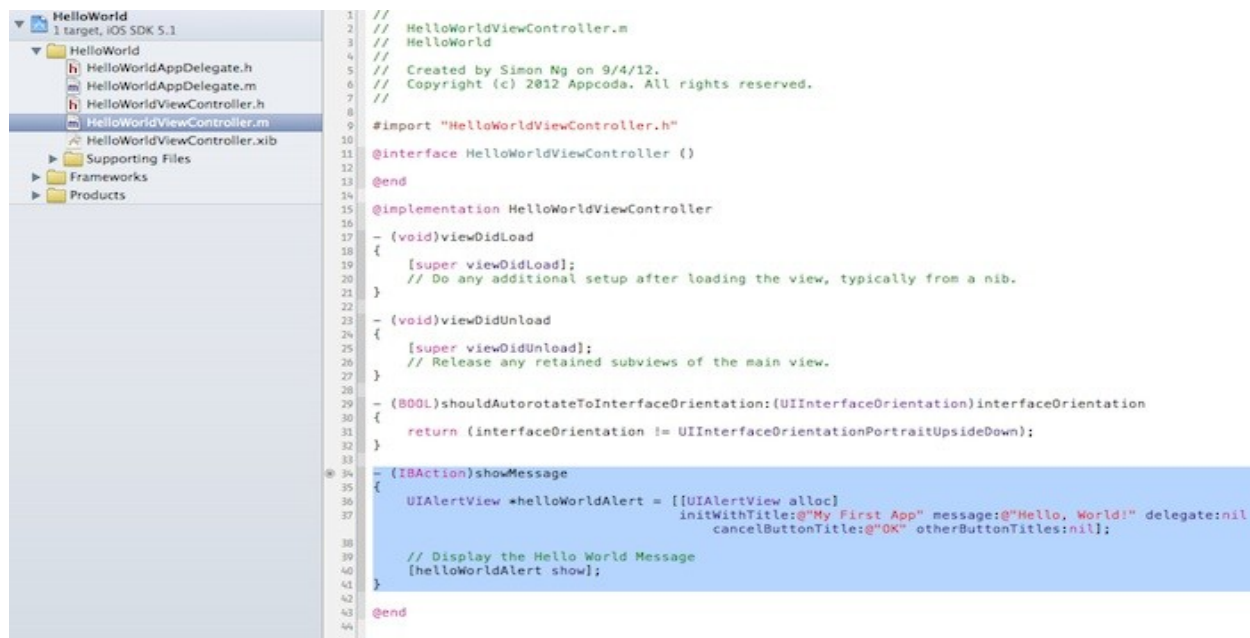
-(IBAction)showMessage;



Step 10: Next, select the “HelloWordViewController.m” and insert the following code before the “@endline”.

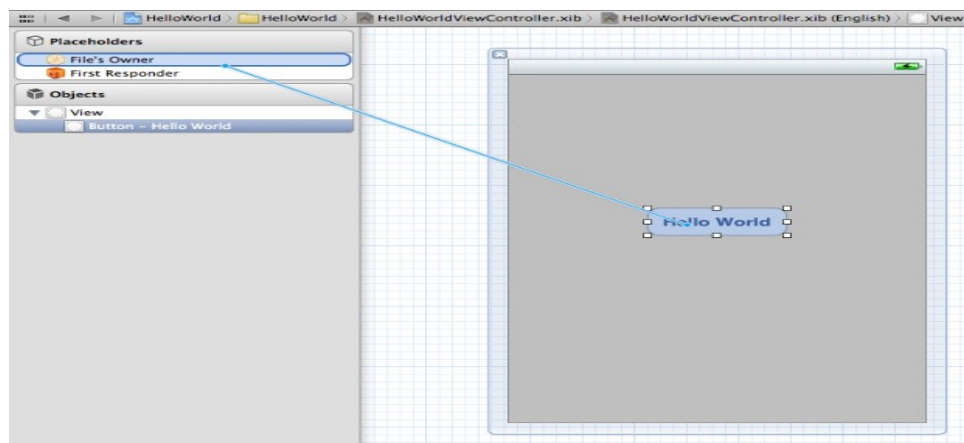
```
- (IBAction)showMessage
{
    UIAlertView *helloWorldAlert = [[UIAlertView alloc]
                                   initWithTitle:@"My First App" message:@"Hello, World!" delegate:nil
                                   cancelButtonTitle:@"OK" otherButtonTitles:nil];

    // Display the Hello World Message
    [helloWorldAlert show];
}
```



Step 11: Connecting Hello World Button with the Action

we'll need to establish a connection between the "Hello World" button and the "showMessage" action we've just added. Select the "HelloWorldViewController.xib" file to go back to the Interface Builder. Press and hold the *Control* key on our keyboard, click the "Hello World" button and drag to the "File's Owner". Our screen should look like this:



Step 12: Test Our App

Just hit the “Run” button. If everything is correct, our app should run properly in the Simulator. An iOS simulator actually consists of two types of devices, namely iPhone and iPad with their different versions. iPhone versions include iPhone (normal), iPhone Retina, iPhone 5. iPad has iPad and iPad Retina. We can simulate location in an iOS simulator for playing around with latitude and longitude effects of the app. We can also simulate memory warning and in-call status in the simulator. We can use the simulator for most purposes, however we cannot test device features like accelerometer.

3 Model-View-Controller

The Model-View-Controller design pattern (MVC) is quite old. Variations of it have been around at least since the early days of Smalltalk. It is a high-level pattern in that it concerns itself with the global architecture of an application and classifies objects according to the general roles they play in an application. It is also a compound pattern in that it comprises several, more elemental patterns. The MVC design pattern considers there to be three types of objects: model objects, view objects, and controller objects. The MVC pattern defines the roles that these types of objects play in the application and their lines of communication.

3.1 Model Object

Model objects represent special knowledge and expertise. They hold an application’s data and define the logic that manipulates that data. A well-designed MVC application has all its important data encapsulated in model objects. Any data that is part of the persistent state of the application (whether that persistent state is stored in files or databases) should reside in the model objects once the data is loaded into the application. Because they represent knowledge and expertise related to a specific problem domain, they tend to be reusable.

3.2 View Objects

A view object knows how to display, and might allow users to edit, the data from the application’s model. The view should not be responsible for storing the data it is displaying. A view object can be in charge of displaying just one part of a model object, or a whole model

object, or even many different model objects. Views come in many different varieties. View objects tend to be reusable and configurable, and they provide consistency between applications. A view should ensure it is displaying the model correctly.

3.3 Controller Objects

A controller object acts as the intermediary between the application's view objects and its model objects. Controllers are often in charge of making sure the views have access to the model objects they need to display and act as the conduit through which views learn about changes to the model. Controller objects can also perform set-up and coordinating tasks for an application and manage the life cycles of other objects.

Model-View-Controller is a design pattern that is composed of several more basic design patterns. These basic patterns work together to define the functional separation and paths of communication that are characteristic of an MVC application. MVC is made up of the Composite, Strategy, and Observer patterns.

- Composite—The view objects in an application are actually a composite of nested views that work together in a coordinated fashion (that is, the view hierarchy). These display components range from a window to compound views, such as a table view, to individual views, such as buttons. User input and display can take place at any level of the composite structure.
- Strategy—A controller object implements the strategy for one or more view objects. The view object confines itself to maintaining its visual aspects, and it delegates to the controller all decisions about the application-specific meaning of the interface behavior.
- Observer—A model object keeps interested objects in an application—usually view objects—advised of changes in its state.

A controller object receives the event and interprets it in an application-specific way—that is, it applies a strategy. This strategy can be to request (via message) a model object to change its state or to request a view object (at some level of the composite structure) to change its behavior or appearance. The model object, in turn, notifies all objects who have registered as

observers when its state changes; if the observer is a view object, it may update its appearance accordingly. The following figure 2 shows the MVC design patterns.

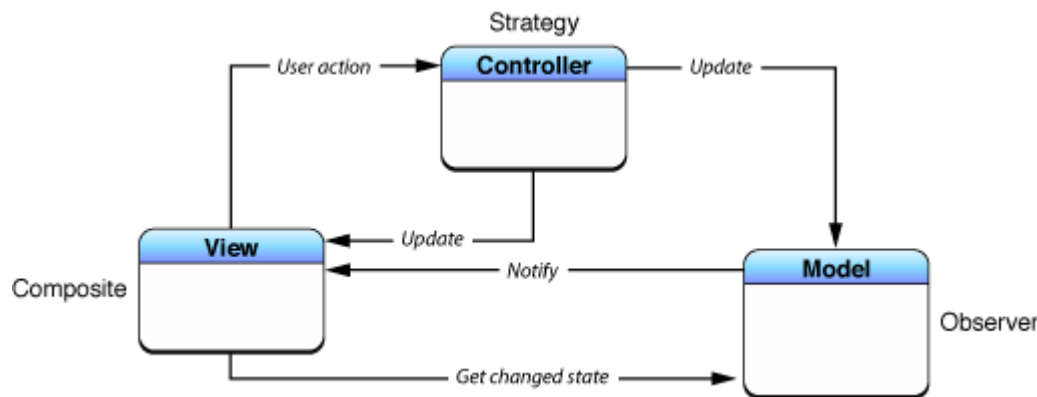


Figure 2 MVC Design Patterns

A user needs to interact with an app interface in the simplest way possible. Design the interface with the user in mind, and make it efficient, clear, and straightforward. Storyboards let we design and implement our interface in a graphical environment. We see exactly what we're building while we're building it, get immediate feedback about what's working and what's not, and make instantly visible changes to our interface.

They are the building blocks for constructing our user interface and presenting our content in a clear, elegant, and useful way. As we develop more complex apps, we'll create interfaces with more scenes and more views. The following figure 3 shows the MVC architecture.

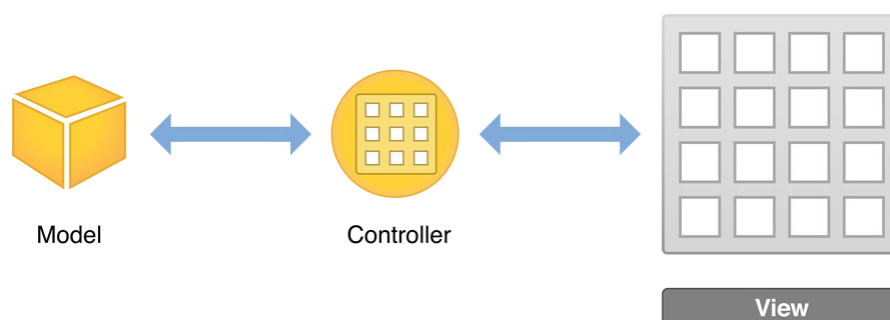


Figure 3. MVC Architecture

One can merge the MVC roles played by an object, making an object, for example, fulfill both the controller and view roles—in which case, it would be called a *view controller*.

A *model controller* is a controller that concerns itself mostly with the model layer. It “owns” the model; its primary responsibilities are to manage the model and communicate with view objects. Action methods that apply to the model as a whole are typically implemented in a model controller. The document architecture provides a number of these methods for we; for example, an `NSDocument` object (which is a central part of the document architecture) automatically handles action methods related to saving files.

A *view controller* is a controller that concerns itself mostly with the view layer. It “owns” the interface (the views); its primary responsibilities are to manage the interface and communicate with the model. Action methods concerned with data displayed in a view are typically implemented in a view controller. An `NSWindowController` object (also part of the document architecture) is an example of a view controller.

A *coordinating controller* is typically an `NSWindowController` or `NSDocumentController` object (available only in AppKit), or an instance of a custom subclass of `NSObject`. Its role in an application is to oversee—or coordinate—the functioning of the entire application or of part of the application, such as the objects unarchived from a nib file. A coordinating controller provides services such as:

- Responding to delegation messages and observing notifications
- Responding to action messages
- Managing the life cycle of owned objects (for example, releasing them at the proper time)
- Establishing connections between objects and performing other set-up tasks

4 View Controller

A *view controller* is a controller that concerns itself mostly with the view layer. It “owns” the interface (the views); its primary responsibilities are to manage the interface and communicate with the model. Action methods concerned with data displayed in a view are typically

implemented in a view controller. An `NSWindowController` object (also part of the document architecture) is an example of a view controller.


Views not only display themselves onscreen and react to user input, they can serve as containers for other views. As a result, views in an app are arranged in a hierarchical structure called the view hierarchy. The view hierarchy defines the layout of views relative to other views. Within that hierarchy, views enclosed within a view are called sub views, and the parent view that encloses a view is referred to as its super view. Even though a view can have multiple sub views, it can have only one super view.






At the top of the view hierarchy is the window object. Represented by an instance of the `UIWindow` class, a window object is the basic container into which we add our view objects for display onscreen. By itself, a window doesn't display any content.

To display content, we add a content view object (with its hierarchy of sub views) to the window. For a content view and its sub views to be visible to the user, the content view must be inserted into a window's view hierarchy. When we use a storyboard, this placement is configured automatically for us. When an app launches, the application object loads the storyboard, creates instances of the relevant view controller classes, unarchives the content view hierarchies for each view controller, and then adds the content view of the initial view controller into the window.

4.1 Types of Views

A UIKit view object is an instance of the `UIView` class or one of its subclasses. The UIKit framework provides many types of views to help present and organize data. Although each view has its own specific function, UIKit views can be grouped into these general categories.

View category	Purpose	Examples of views
 Content	Display a particular type of content, such as an image or text.	Image view, label

View category	Purpose	Examples of views
 Collections	Display collections or groups of views.	Collection view, table view
 Controls	Perform actions or display information.	Button, slider, switch
 Bars	Navigate, or perform actions.	Toolbar, navigation bar, tab bar
 Input	Receive user input text.	Search bar, text view
 Containers	Serve as containers for other views.	View, scroll view

4.2 Use Storyboards to Lay Out Views

Storyboards provide a direct, visual way to work with views and build our interface and composed of scenes, and each scene has an associated view hierarchy. We drag a view out of the object library and place it in a storyboard scene to add it automatically to that scene's view hierarchy. The view's location within that hierarchy is determined by where we place it. After we add a view to our scene, we can resize, manipulate, configure, and move it on the canvas.

The canvas also shows an outline view of the objects in our interface. The outline view which appears on the left side of the canvas—lets we see a hierarchical representation of the objects in our storyboard. The following figure 4 shows the view controller.

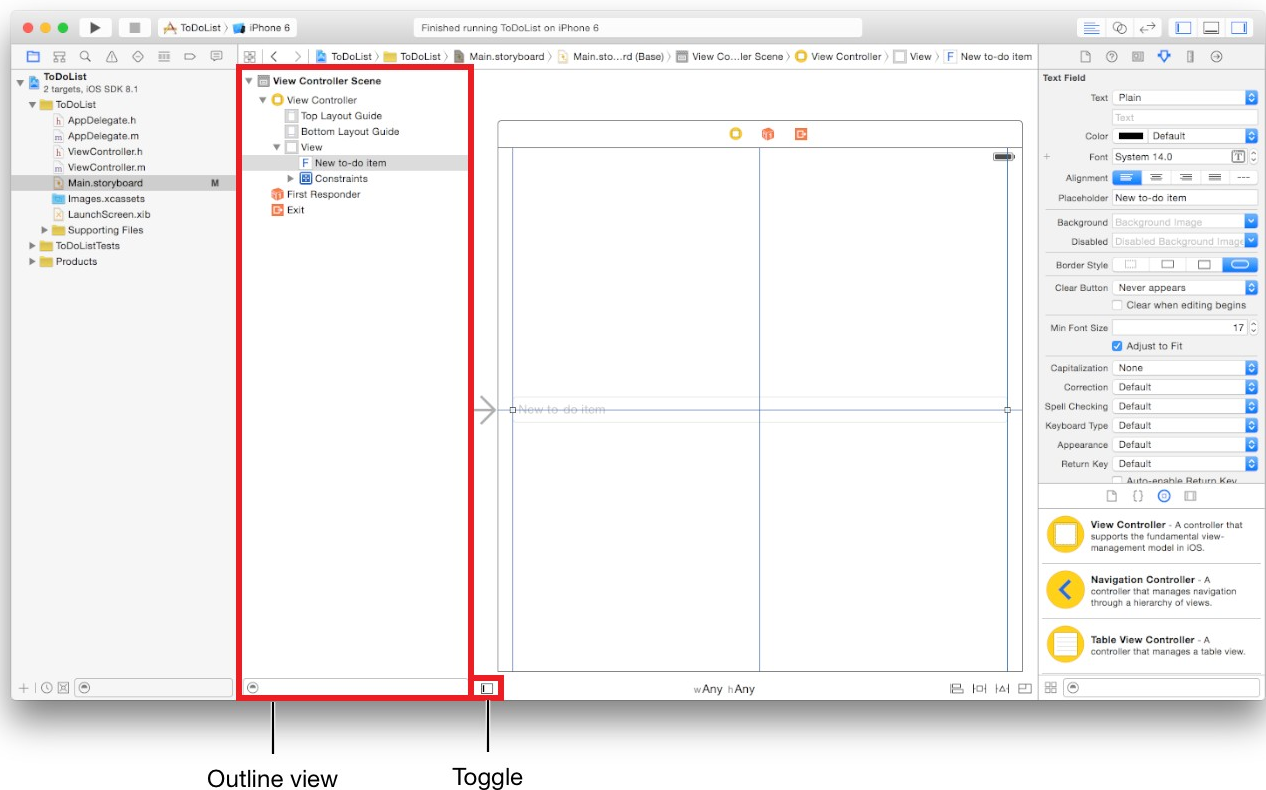


Figure 4 View Controller

The view hierarchy that we create graphically in a storyboard scene is effectively a set of archived Objective-C objects. At runtime, these objects are unarchived. The result is a hierarchy of instances of the relevant classes configured with the properties we've set visually using the various inspectors in the utility area.

When we need to adjust our interface for specific device sizes or orientations, we make the changes to specific size classes. A size class is a high-level way to describe the horizontal or vertical space that's available in a display environment, such as iPhone in portrait or iPad in landscape. There are two types of size classes: regular and compact. A display environment is characterized by a pair of size classes, one that describes the horizontal space and one that describes the vertical space. We can view and edit our interface for different combinations of regular and compact size classes using the size class control on the canvas. The following figure 5 and 6 show the inspector pane and auto layout icons.

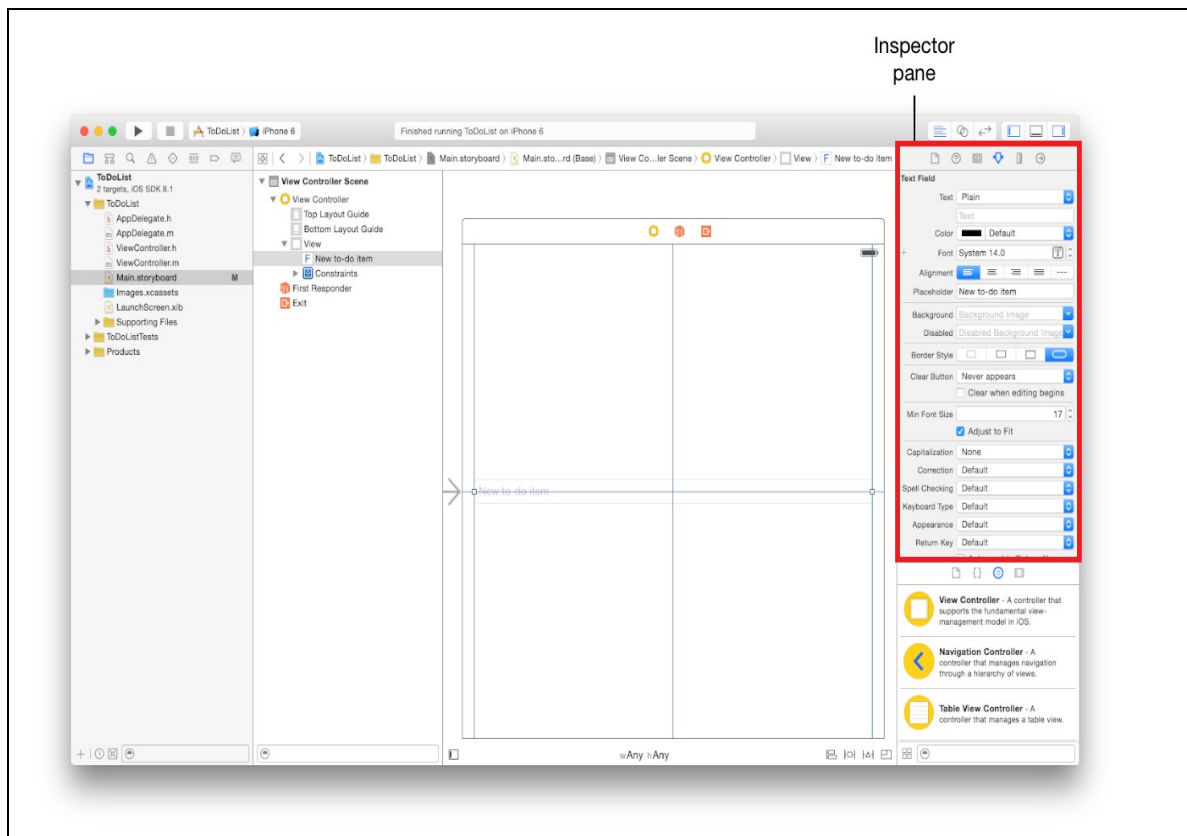
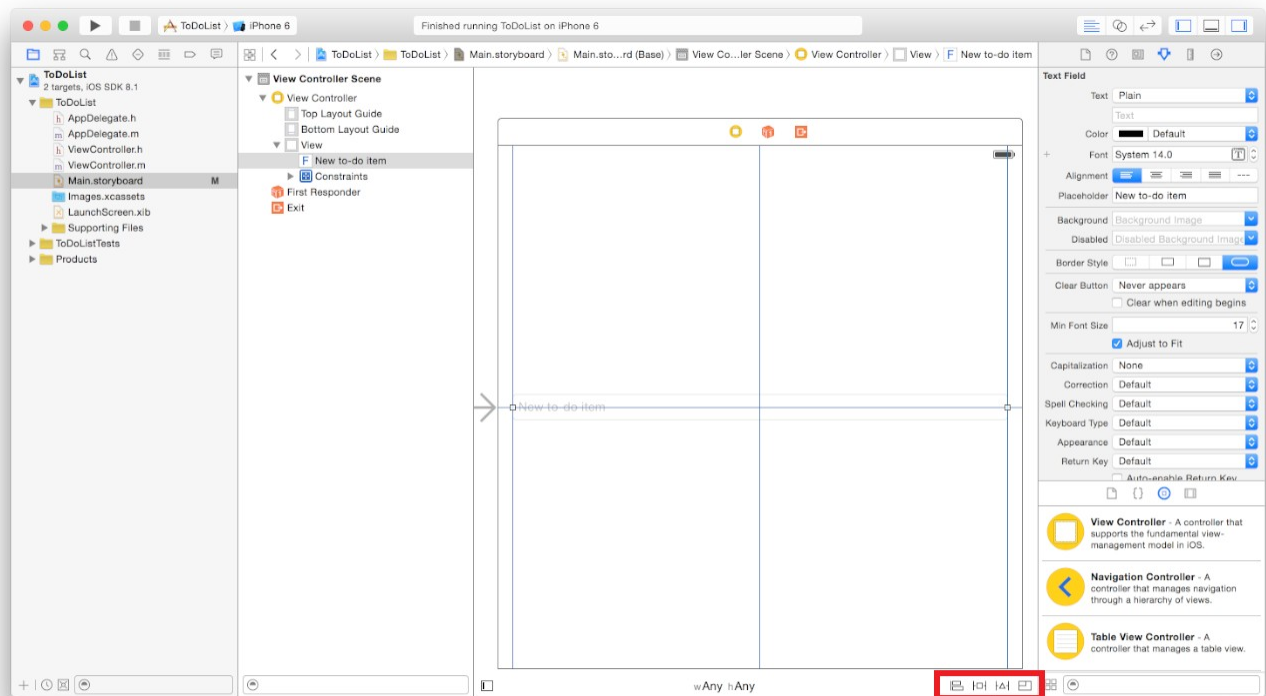


Figure 5 Inspection Pane

Use the Auto Layout icons in the bottom-right area of our canvas to add various types of constraints to views on our canvas, resolve layout issues, and determine constraint resizing behavior.

- **Align.** Create alignment constraints, such as centering a view in its container, or aligning the left edges of two views.
- **Pin.** Create spacing constraints, such as defining the height of a view, or specifying its horizontal distance from another view.
- **Resolve Auto Layout Issues.** Resolve layout issues by adding or resetting constraints based on suggestions.
- **Resizing Behavior.** Specify how resizing affects constraints.



Auto Layout icons

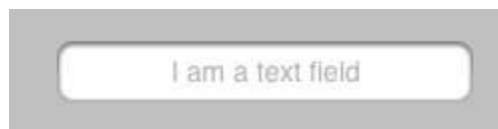
Figure 6 Auto Layout Icons

5 Building UI

UI elements are the visual elements that we can see in our applications. Some of these elements respond to user interactions such as buttons, text fields and others are informative such as images, labels.

5.1 Use of Text Field

A text field is a UI element that enables the app to get user input. A UITextField is shown below.



Important Properties of Text Field

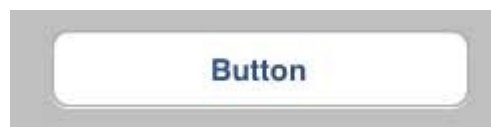
- Placeholder text which is shown when there is no user input

- ▶ Normal text
- ▶ Auto correction type
- ▶ Key board type
- ▶ Return key type
- ▶ Clear button mode
- ▶ Alignment
- ▶ Delegate

Input Type	Description
UIKeyboardTypeASCIICapable	Keyboard includes all standard ASCII characters.
UIKeyboardTypeNumbersAndPunctuation	Keyboard display numbers and punctuations once it's shown.
UIKeyboardTypeURL	Keyboard is optimized for URL entry.
UIKeyboardTypeNumberPad	Keyboard is used for PIN input and shows a numeric keyboard.
UIKeyboardTypePhonePad	Keyboard is optimized for entering phone numbers.
UIKeyboardTypeNamePhonePad	Keyboard is used for entering name or phone number.
UIKeyboardTypeEmailAddress	Keyboard is optimized for entering email address.
UIKeyboardTypeDecimalPad	Keyboard is used for entering decimal numbers.

5.2 Buttons

Buttons are used for handling user actions. It intercepts the touch events and sends message to the target object.



5.2.1 Buttons Types

- ▶ UIButtonTypeCustom
- ▶ UIButtonTypeRoundedRect
- ▶ UIButtonTypeDetailDisclosure
- ▶ UIButtonTypeInfoLight
- ▶ UIButtonTypeInfoDark
- ▶ UIButtonTypeContactAdd

5.2.2 Code

```
-(void)addDifferentTypesOfButton  
  
{  
  
    // A rounded Rect button created by using class method  
  
    UIButton *roundRectButton = [UIButton buttonWithType: UIButtonTypeRoundedRect];  
  
    [roundRectButton setFrame:CGRectMake(60, 50, 200, 40)]; // sets title for the button  
  
    [roundRectButton setTitle:@"Rounded Rect Button" forState: UIControlStateNormal];  
  
    [self.view addSubview:roundRectButton];  
}
```

5.3 Labels

Labels are used for displaying static content, which consists of a single line or multiple lines.

5.3.1 Important Properties

- ▶ textAlignment
- ▶ textColor

- ▶ text
- ▶ numberOfLines
- ▶ lineBreakMode

5.3.2 - (void)addLabel

```
{  
  
    UILabel *aLabel = [[UILabel alloc] initWithFrame: CGRectMake(20, 200, 280, 80)];  
  
    aLabel.numberOfLines = 0;  
  
    aLabel.textColor = [UIColor blueColor];  
  
    aLabel.backgroundColor = [UIColor clearColor];    aLabel.textAlignment =  
    NSTextAlignmentCenter;  
  
    aLabel.text = @"This is a sample text\n of multiple lines. here number of lines is not  
    limited.";  
  
    [self.view addSubview:aLabel];  
  
}  
  
- (void)viewDidLoad  
  
{  
  
    [super viewDidLoad];  
  
    [self addLabel];  
  
}
```




5.4 Toolbar

If we want to manipulate something based on our current view we can use toolbar. Example would be the email app with an inbox item having options to delete, make favourite, reply and so on. It is shown below.



5.5 Status Bar

- ▶ Status bar displays the key information of device like –
- ▶ Device model or network provider
- ▶ Network strength
- ▶ Battery information
- ▶ Time

Status bar is shown below.



5.5.1 Add a Custom Method `hideStatusBar` to our Class

- It hides the status bar animated and also resize our view to occupy the statusbar space.

```
-(void)hideStatusBar  
{  
    [[UIApplication sharedApplication] setStatusBarHidden:YES  
withAnimation:UIStatusBarAnimationFade];  
  
    [UIView beginAnimations:@"Statusbar hide" context:nil];  
  
    [UIView setAnimationDuration:0.5];  
  
    [self.view setFrame:CGRectMake(0, 0, 320, 480)];  
  
    [UIView commitAnimations];  
}
```

5.6 Tab Bar

- It's generally used to switch between various subtasks, views or models within the same view.
- Example for tab bar is shown below.

Important Properties

- `backgroundImage`
- `items`
- `selectedItem`

5.7 Image View

- ▶ Image view is used for displaying a single image or animated sequence of images.

Important Properties

- ▶ image
- ▶ highlightedImage
- ▶ userInteractionEnabled
- ▶ animationImages
- ▶ animationRepeatCount

Important Methods

- (id)initWithImage:(UIImage *)image - (id)initWithImage:(UIImage *)image
highlightedImage: (UIImage *)highlightedImage
- (void)startAnimating
- (void)stopAnimating

Add a Custom Method addImageView

```
-(void)addImageView  
  
{  
  
    UIImageView *imgview = [[UIImageView alloc] initWithFrame:CGRectMake(10, 10, 300,  
400)];  
  
    [imgview setImage:[UIImage imageNamed:@"AppleUSA1.jpg"]];  
  
    [imgview setContentMode:UIViewContentModeScaleAspectFit];  
}
```

```
[self.view addSubview:imgview];  
  
}
```

5.8 Scroll View

Scroll View is used for displaying content more than the size of the screen. It can contain all of the other UI elements like image views, labels, text views and even another scroll view itself.

Important Properties

- ▶ `contentSize`
- ▶ `contentInset`
- ▶ `contentOffset`
- ▶ `delegate`

5.8.1 Code

```
-(void)addScrollView  
{  
  
    myScrollView = [[UIScrollView alloc] initWithFrame: CGRectMake(20, 20, 280, 420)];  
  
    myScrollView.accessibilityActivationPoint = CGPointMake(100, 100);  
  
    imgView = [[UIImageView alloc] initWithImage: [UIImage  
    imageNamed:@"AppleUSA.jpg"]];  
  
    [myScrollView addSubview:imgView];  
  
    myScrollView.minimumZoomScale = 0.5; myScrollView.maximumZoomScale = 3;  
  
    myScrollView.contentSize = CGSizeMake(imgView.frame.size.width,  
    imgView.frame.size.height);
```

```
myScrollView.delegate = self; [self.view addSubview:myScrollView];  
  
}
```



5.9 Table View

It is used for displaying a vertically scrollable view which consists of a number of cells (generally reusable cells). It has special features like headers, footers, rows, and section.

Important Properties

- ▶ Delegate
- ▶ Data source
- ▶ Row height
- ▶ Section footer height
- ▶ Section header height
- ▶ Separator color
- ▶ Table header view
- ▶ Table footer view

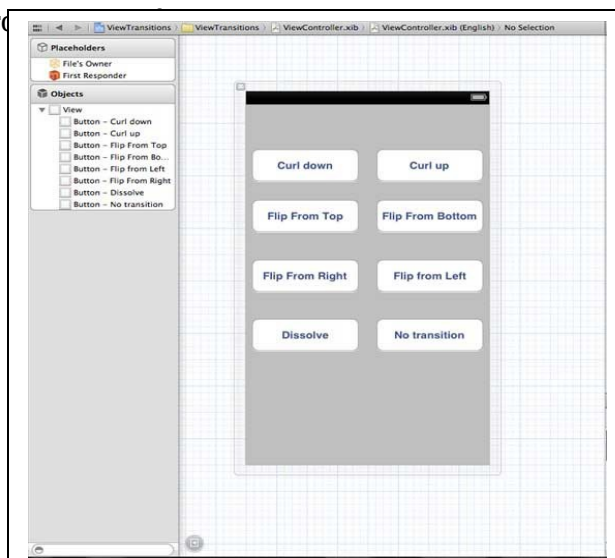
5.9.1 Code

- @interface ViewController ()
- @end
- @implementation ViewController
- (void)viewDidLoad
- {
- [super viewDidLoad]; // table view data is being set here
- myData = [[NSMutableArray alloc] initWithObjects:
- @"Data 1 in array",
- @"Data 2 in array",
- @"Data 3 in array",
- @"Data 4 in array",
- @"Data 5 in array",
- @"Data 5 in array",
- @"Data 6 in array",
- @"Data 7 in array",
- @"Data 8 in array",
- @"Data 9 in array", nil];

5.10 View Transitions

View Transitions are effective ways of adding one view on another view with a proper transition animation effect.

Update ViewController



5.10 Pickers

Pickers consist of a rotating scrollable view, which is used for picking a value from the list of items.

Important Properties

- ▶ delegate
- ▶ dataSource

Important Methods

- ▶ - (void)reloadAllComponents:(NSInteger)component - Reloads all components of the picker.
- ▶ - (void)selectRow:(NSInteger)row inComponent:(NSInteger)component animated:(BOOL)animated - Selects the row in the component.



5.12 Switches

- ▶ Switches are used to toggle between on and off states.

Important Properties

- ▶ `onImage`
- ▶ `offImage`
- ▶ `on`

Important Method

- `(void)setOn:(BOOL)on animated:(BOOL)animated`

- `(IBAction)switched:(id)sender`

{

`NSLog(@"Switch current state %@", mySwitch.on ? @"On" : @"Off");`

}

- `(void)addSwitch`

{

`mySwitch = [[UISwitch alloc] init];`

`[self.view addSubview:mySwitch];`

`mySwitch.center = CGPointMake(150, 200);`


```
[mySwitch addTarget:self action:@selector(switched:)
forControlEvents:UIControlEventValueChanged];

}
```

5.13 Sliders

- ▶ Sliders are used to choose a single value from a range of values.

Important Properties

- ▶ Continuous
- ▶ Maximum Value
- ▶ Minimum Value
- ▶ Value

Important Method

- ▶ - (void)setValue:(float)value animated:(BOOL)animated

5.13.1 Code

```
-(IBAction)sliderChanged:
(id)sender
{
    NSLog(@"SliderValue %f",mySlider.value);
}

-(void)addSlider
{
    mySlider = [[UISlider alloc] initWithFrame:CGRectMake(50, 200, 200, 23)]; [self.view
addSubview:mySlider];

    mySlider.minimumValue = 10.0;

    mySlider.maximumValue = 99.0;

    mySlider.continuous = NO;
```

```
[mySlider addTarget:self action:@selector(sliderChanged:)
forControlEvents:UIControlEventValueChanged];

}
```



5.14 Alerts

- ▶ Alerts are used to give important information to user. Only after selecting the option in the alert view, we can proceed further using the app.

Important Properties

- ▶ Alert View Style
- ▶ Cancel Button Index
- ▶ Delegate message
- ▶ Number Of Buttons
- ▶ Title

5.14

.1 Code

```
(NSInteger)addButtonWithTitle:(NSString *)title
```

```
- (NSString *)buttonTitleAtIndex:(NSInteger)buttonIndex
```

```
- (void)dismissWithClickedButtonIndex: (NSInteger)buttonIndex animated:
(BOOL)animated
```

- (id)initWithTitle:(NSString *)title message: (NSString *)message delegate:(id)delegate cancelButtonTitle:(NSString *)cancelButtonTitle otherButtonTitles: (NSString*)otherButtonTitles, ...

- - (void)show



6 Event Handling

users manipulate their iOS devices in a number of ways, such as touching the screen or shaking the device. iOS interprets when and how a user is manipulating the hardware and passes this information to our app. The more our app responds to actions in natural and intuitive ways, the more compelling the experience is for the user.

Events are objects sent to an app to inform it of user actions. In iOS, events can take many forms: Multi-Touch events, motion events, and events for controlling multimedia. This last type of event is known as a *remote control event* because it can originate from an external accessory.

iOS apps recognize combinations of touches and respond to them in ways that are intuitive to users, such as zooming in on content in response to a pinching gesture and scrolling through content in response to a flicking gesture. In fact, some gestures are so common that they are built in to UIKit. For example, [UIControl](#) subclasses, such as [UIButton](#) and [UISlider](#), respond to specific gestures—a tap for a button and a drag for a slider. When we configure these controls, they send an action message to a target object when that touch occurs. We can also employ the target-action mechanism on views by using gesture recognizers. When we attach a gesture

recognizer to a view, the entire view acts like a control—responding to whatever gesture we specify.

Gesture recognizers provide a higher-level abstraction for complex event handling logic. Gesture recognizers are the preferred way to implement touch-event handling in our app because gesture recognizers are powerful, reusable, and adaptable. We can use one of the built-in gesture recognizers and customize its behavior. Or we can create our own gesture recognizer to recognize a new gesture.

6.1 Gesture Recognizers

When iOS recognizes an event, it passes the event to the initial object that seems most relevant for handling that event, such as the view where a touch occurred. If the initial object cannot handle the event, iOS continues to pass the event to objects with greater scope until it finds an object with enough context to handle the event. This sequence of objects is known as a *responder chain*, and as iOS passes events along the chain, it also transfers the responsibility of responding to the event. This design pattern makes event handling cooperative and dynamic.

6.2 Multitouch Events

Depending on our app, UIKit controls and gesture recognizers might be sufficient for all of our app's touch event handling. Even if our app has custom views, we can use gesture recognizers. As a rule of thumb, we write our own custom touch-event handling when our app's response to touch is tightly coupled with the view itself, such as drawing under a touch. In these cases, we are responsible for the low-level event handling. We implement the touch methods, and within these methods, we analyze raw touch events and respond appropriately.

6.3 Motion Events

Motion events provide information about the device's location, orientation, and movement. By reacting to motion events, we can add subtle, yet powerful features to our app. Accelerometer and gyroscope data allow us to detect tilting, rotating, and shaking.

Motion events come in different forms, and we can handle them using different frameworks. When users shake the device, UIKit delivers a `UIEvent` object to an app. If we want our app to

receive high-rate, continuous accelerometer and gyroscope data, use the Core Motion framework.

6.4 Remote Control Events

IOS controls and external accessories send remote control events to an app. These events allow users to control audio and video, such as adjusting the volume through a headset. Handle multimedia remote control events to make our app responsive to these types of commands.

The following figure shows the architecture of the main run loop and how user events result in actions taken by our app. As the user interacts with a device, events related to those interactions are generated by the system and delivered to the app via a special port set up by UIKit. Events are queued internally by the app and dispatched one-by-one to the main run loop for execution. The UIApplication object is the first object to receive the event and make the decision about what needs to be done. A touch event is usually dispatched to the main window object, which in turn dispatches it to the view in which the touch occurred. Other events might take slightly different paths through various app objects.

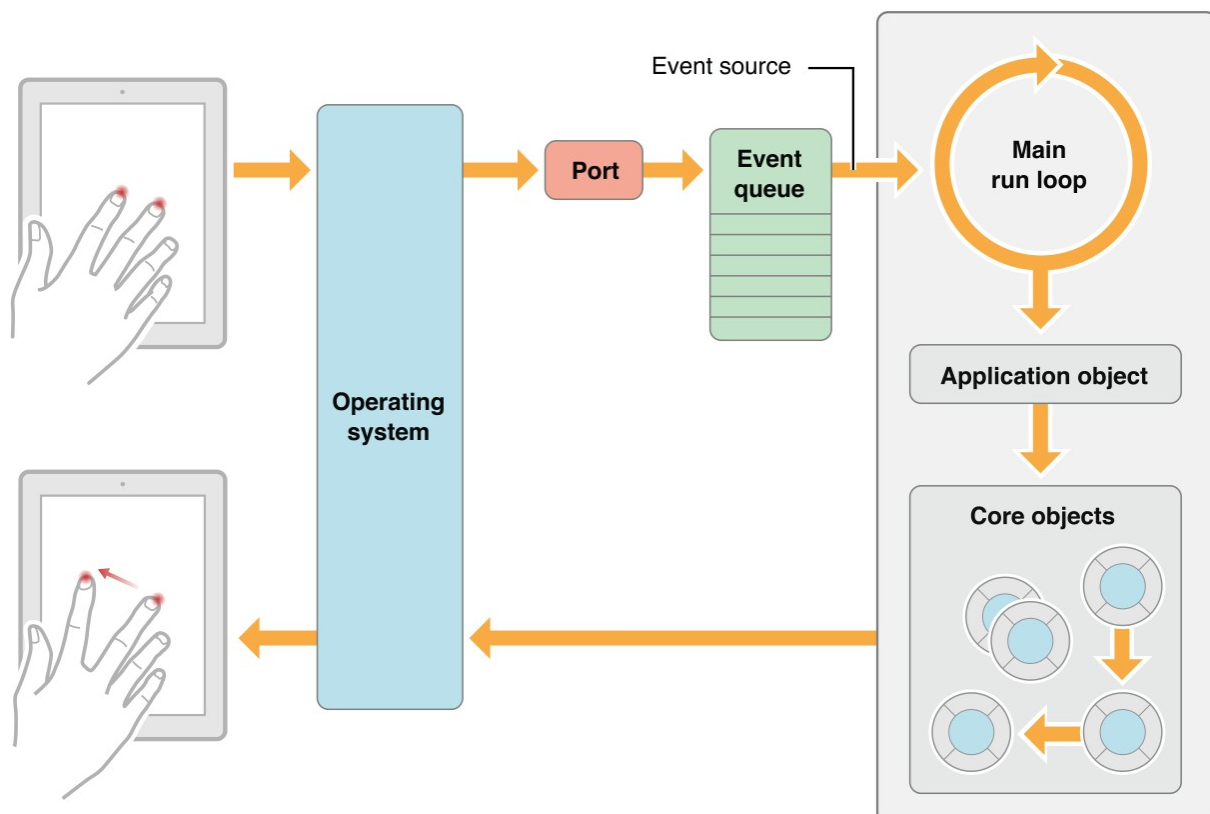


Figure 7 Event handling

Event	Delivered to	Description
Touch	The view object in which the event occurred	Views are responder objects. Any touch events not handled by the view are forwarded down the responder chain for processing.
Remote control Shake motion events	First responder object	Remote control events are for controlling media playback and are generated by headphones and other accessories.
Accelerometer Magnetometer Gyroscope	The object designated	Events related to the accelerometer, magnetometer, and gyroscope hardware are delivered to the object designated.
Location	The object designated	Register to receive location events using the Core Location framework.
Redraw	The view that needs the update	Redraw events do not involve an event object but are simply calls to the view to draw itself.
Touch	The view object in which the event occurred	Views are responder objects. Any touch events not handled by the view are forwarded down the responder chain for processing.

Some events, such as touch and remote control events, are handled by our app's responder objects. Responder objects are everywhere in our app. Most events target a

specific responder object but can be passed to other responder objects (via the responder chain) if needed to handle an event. For example, a view that does not handle an event can pass the event to its superview or to a view controller.

Touch events occurring in controls (such as buttons) are handled differently than touch events occurring in many other types of views. There are typically only a limited number of interactions possible with a control, and so those interactions are repackaged into action messages and delivered to an appropriate target object. This target-action design pattern makes it easy to use controls to trigger the execution of custom code in our app.

7 App Life Cycle

Apps are a sophisticated interplay between our custom code and the system frameworks. The system frameworks provide the basic infrastructure that all apps need to run, and we provide the code required to customize that infrastructure and give the app the look and feel we want. To do that effectively, it helps to understand a little bit about the iOS infrastructure and how it works.

The system moves our app from state to state in response to actions happening throughout the system. For example, when the user presses the Home button, a phone call comes in, or any of several other interruptions occurs, the currently running apps change state in response.

State	Description
Not running	The app has not been launched or was running but was terminated by the system.
Inactive	The app is running in the foreground but is currently not receiving events. (It may be executing other code though.) An app usually stays in this state only briefly as it transitions to a different state.
Active	The app is running in the foreground and is receiving events. This is the normal mode for foreground apps.

Background	The app is in the background and executing code. Most apps enter this state briefly on their way to being suspended. However, an app that requests extra execution time may remain in this state for a period of time. In addition, an app being launched directly into the background enters this state instead of the inactive state.
Suspended	The app is in the background but is not executing code. The system moves apps to this state automatically and does not notify them before doing so. While suspended, an app remains in memory but does not execute any code. When a low-memory condition occurs, the system may purge suspended apps without notice to make more space for the foreground app.

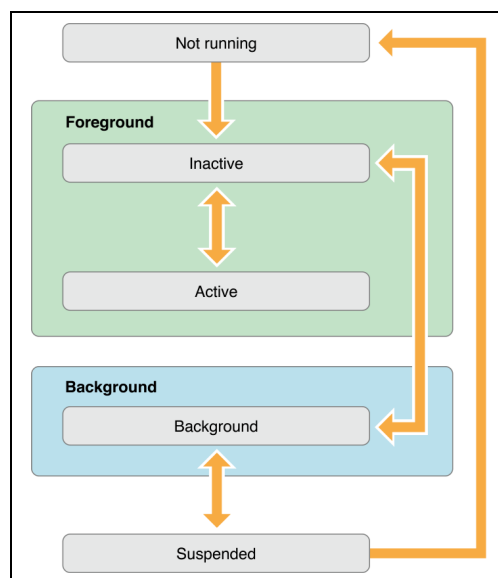


Figure 8 App Life Cycle

Tab bar Controllers

A *tab bar controller* is a container view controller that we use to divide our app into two or more distinct modes of operation. A tab bar controller is an instance of the `UITabBarController` class. The tab bar has multiple tabs, each represented by a child view controller. Selecting a tab causes the tab bar controller to display the associated view controller's view on the screen. The following figure shows several modes of the Clock app along with the relationships between the corresponding view controllers. Each mode has a content view

controller to manage the main content area. In the case of the Clock app, the Clock and Alarm view controllers both display a navigation-style interface to accommodate some additional controls along the top of the screen. The other modes use content view controllers to present a single screen.

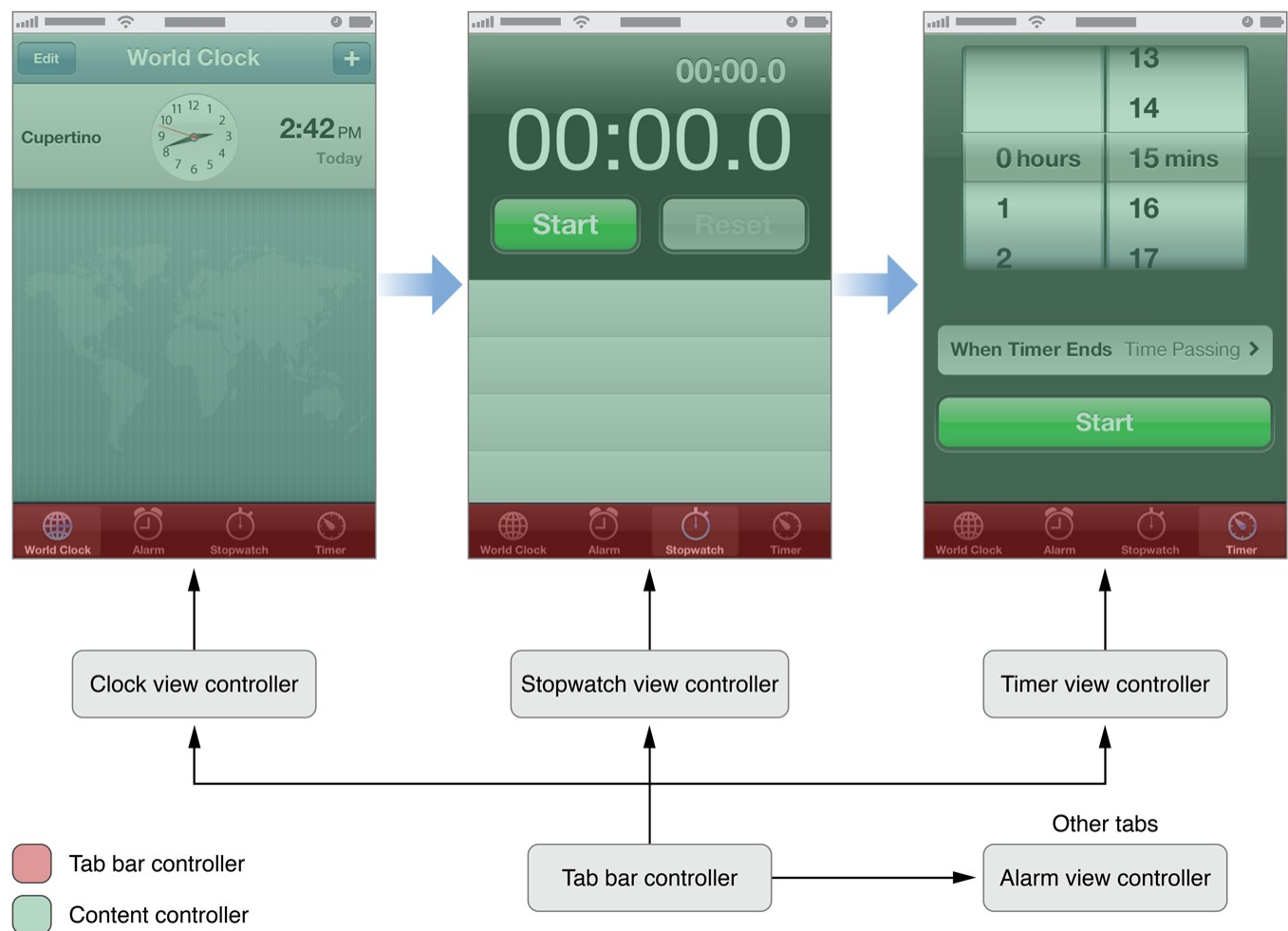


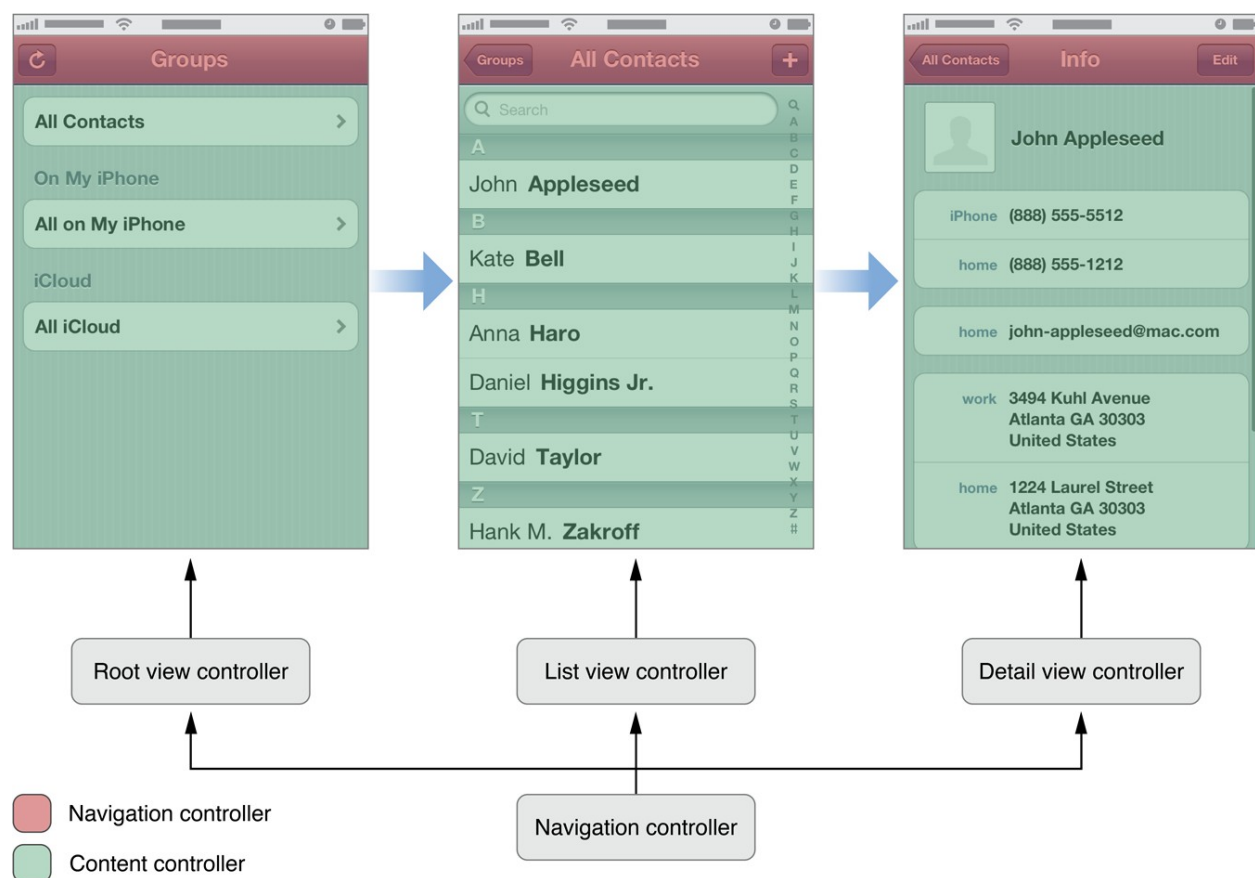
Figure 9 Tab bar Controller

Navigation Controllers

A navigation controller presents data that is organized hierarchically and is an instance of the UINavigationController class. The methods of this class provide support for managing a

stack-based collection of content view controllers. This stack represents the path taken by the user through the hierarchical data, with the bottom of the stack reflecting the starting point and the top of the stack reflecting the user's current position in the data.

The figure 10 shows screens from the Contacts app, which uses a navigation controller to present contact information to the user. The navigation bar at the top of each page is owned by the navigation controller. The rest of each screen displayed to the user is managed by a content view controller that presents the information at that specific level of the data hierarchy. As the user interacts with controls in the interface, those controls tell the navigation controller to display the next view controller in the sequence or dismiss the current view controller.



8 Story Board

A **Storyboard** is a visual representation of the appearance and flow of our application. When we implement our app using storyboards, we use Interface Builder to organize our app's view controllers and any associated views. The following figure shows an example interface layout from Interface Builder. The visual layout of Interface Builder allows us to understand the flow through an app at a glance. The resulting storyboard is stored as a file in the project. When we build our project, the storyboards in our project are processed and copied into the app bundle, where they are loaded by our app at runtime. The figure 11 shows the details of the storyboard.

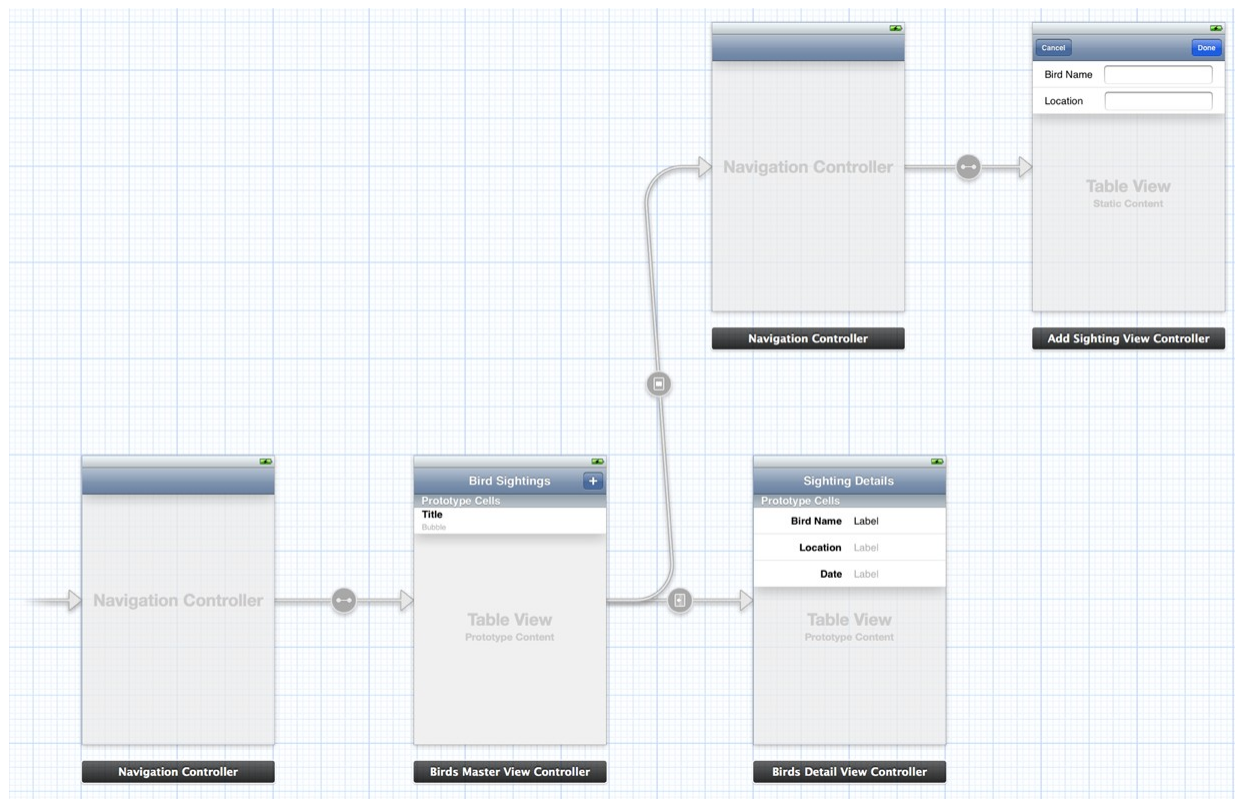


Figure 11 Story Board

Often, iOS can automatically instantiate the view controllers in our storyboard at the moment they are needed. Similarly, the view hierarchy associated with each controller is automatically loaded when it needs to be displayed. Both view controllers and views are instantiated with the same attributes we configured in Interface Builder. Because most of this

behavior is automated for us, it greatly simplifies the work required to use view controllers in our app.

A *scene* represents an onscreen content area that is managed by a view controller. We can think of a scene as a view controller and its associated view hierarchy. We create **relationships** between scenes in the same storyboard. Relationships are expressed visually in a storyboard as a connection arrow from one scene to another. Interface Builder usually infers the details of a new relationship automatically when we make a connection between two objects. Two important kinds of relationships exist:

- **Containment** represents a parent-child relationship between two scenes. View controllers contained in other view controllers are instantiated when their parent controller is instantiated. For example, the first connection from a navigation controller to another scene defines the first view controller pushed onto the navigation stack. This controller is automatically instantiated when the navigation controller is instantiated.

An advantage to using containment relationships in a storyboard is that Interface Builder can adjust the appearance of the child view controller to reflect the presence of its ancestors. This allows Interface Builder to display the content view controller as it appears in our final app.

- A *segue* represents a visual transition from one scene to another. At runtime, segues can be triggered by various actions. When a segue is triggered, it causes a new view controller to be instantiated and transitioned onscreen.

Although a segue is always from one view controller to another, sometimes a third object can be involved in the process. This object actually triggers the segue. For example, if we make a connection from a button in the source view controller's view hierarchy to the destination view controller, when the user taps the button, the segue is triggered. When a segue is made directly from the source view controller to the destination view controller, it usually represents a segue we intend to trigger programmatically.

Different kinds of segues provide the common transitions needed between two different view controllers:

- A *push segue* pushes the destination view controller onto a navigation controller's stack.
- A *modal segue* presents the destination view controller.
- A *popover segue* displays the destination view controller in a popover.
- A *custom segue* allows weto design our own transition to display the destination view controller.

9 Push Notification

Apple Push Notification service (APNs) is the centerpiece of the remote notifications feature. It is a robust and highly efficient service for propagating information to iOS (and, indirectly, watchOS), tvOS, and OS X devices. Each device establishes an accredited and encrypted IP connection with APNs and receives notifications over this persistent connection. If a notification for an app arrives when that app is not running, the device alerts the user that the app has data waiting for it.

We provide our own server to generate the remote notifications for our users. This server, known as the *provider*, gathers data for our users and decides when a notification needs to be sent. For each notification, the provider generates the notification payload and attaches that payload to an HTTP/2 request, which it then sends to APNs using a persistent and secure channel using the HTTP/2 multiplex protocol. Upon receipt of our request, APNs handles the delivery of our notification payload to our app on the user's device.

11.1 The Path of a Remote Notification

Apple Push Notification service transports and routes remote notifications for our apps from our provider to each user's device. The following figure shows the path each notification takes. When our provider determines that a notification is needed, we send the notification and a device token to the APNs servers. The APNs servers handle the routing of that notification to the correct user device, and the operating system handles the deliver of the notification to our client app.

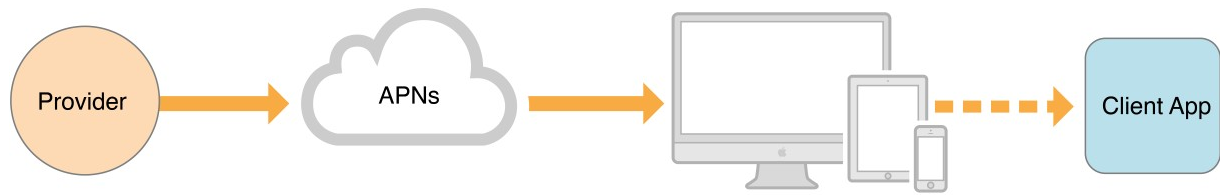


Figure 12 Pushing a remote notification from a provider to a client app

The device token we provide to the server is analogous to a phone number; it contains information that enables APNs to locate the device on which our client app is installed. APNs also uses it to authenticate the routing of a notification. The device token is provided to our client app, which receives the token after registering itself with the remote notification service.

The notification payload is a JSON dictionary containing the data we want sent to the device. The payload contains information about how we want to notify the user, such as using an alert, badge or sound. It can also contain custom data that we define.

The following figure 13 shows a more realistic depiction of the virtual network APNs makes possible among providers and devices. The device-facing and provider-facing sides of APNs both have multiple points of connection; on the provider-facing side, these are called gateways. There are typically multiple providers, each making one or more persistent and secure connections with APNs through these gateways. And these providers are sending notifications through APNs to many devices on which their client apps are installed.

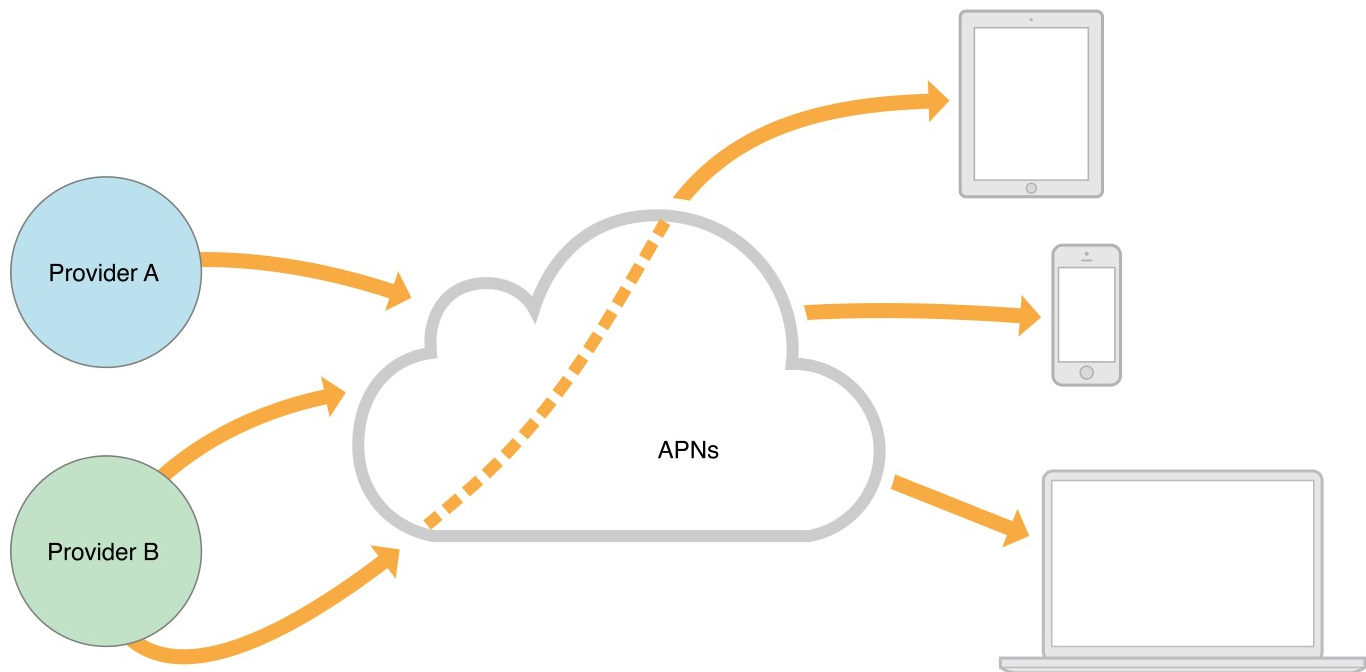


Figure 13 Pushing remote notifications from multiple providers to multiple devices

11.2 Quality of Service

Apple Push Notification service includes a default Quality of Service (QoS) component that performs a store-and-forward function. If APNs attempts to deliver a notification but the device is offline, the notification is stored for a limited period of time, and delivered to the device when it becomes available. Only one recent notification for a particular app is stored. If multiple notifications are sent while the device is offline, the new notification causes the prior notification to be discarded. This behavior of keeping only the newest notification is referred to as *coalescing* notifications. If the device remains offline for a long time, any notifications that were being stored for it are discarded.

11.3 Security Architecture

To ensure secure communication, APNs regulates the entry points between providers and devices using two different levels of trust: connection trust and token trust.

Connection trust establishes certainty that APNs is connected to an authorized provider for whom Apple has agreed to deliver notifications. APNs also uses connection trust with the

device to ensure the legitimacy of that device. Connection trust with the device is handled automatically by APNs but we must take steps to ensure connection trust exists between our provider and APNs.

Token trust ensures that notifications are routed only between legitimate start and end points. Token trust involves the use of a device token, which is an opaque identifier assigned to a specific app on a specific device. Each app instance receives its unique token when it registers with APNs and must share this token with its provider. Thereafter, the token must accompany each notification sent by our provider. Providing the token ensures that the notification is delivered only to the app/device combination for which it is intended.

11.4 Provider-to-APNs Connection Trust

Each provider must have a unique provider certificate and private cryptographic key, which are used to validate the provider's connection with APNs. The provider certificate (which is provisioned by Apple) identifies the topics supported by the provider. (A topic is the bundle ID associated with one of our apps.)

Our provider establishes connection trust with APNs through TLS peer-to-peer authentication. After the TLS connection is initiated, we get the server certificate from APNs and validate that certificate on our end. Then we send our provider certificate to APNs, which validates that certificate on its end. After this procedure is complete, a secure TLS connection is established; APNs is now satisfied that the connection has been made by a legitimate provider.

12. Data Base

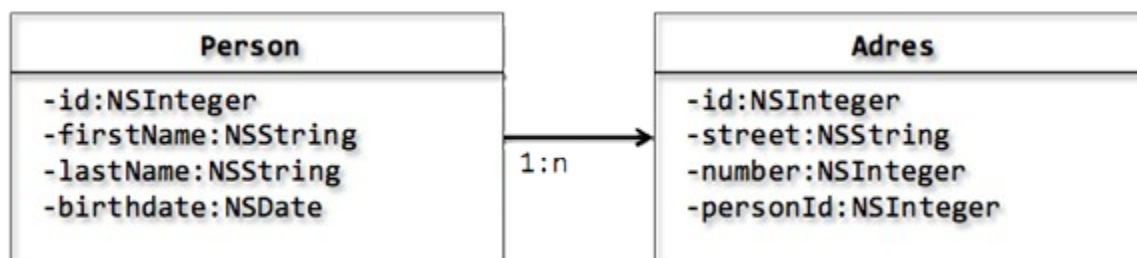
The database that can be used by apps in iOS (and also used by iOS) is called **SQLite**, and it's a *relational database*. It is contained in a C-library that is embedded to the app that is about to use it. Note that it does not consist of a separate service or daemon running on the background and attached to the app. On the contrary, the app runs it as an integral part of it. Nowadays, SQLite lives its third version, so it's also commonly referred as *SQLite 3*.

SQLite is not as powerful as other DBMSs, such as MySQL or SQL Server, as it does not include all of their features. However, its greatness lies mostly to these factors:

- It's lightweight.
- It contains an embedded SQL engine, so almost all of our SQL knowledge can be applied.
- It works as part of the app itself, and it doesn't require extra active services.
- It's very reliable.
- It's fast.
- It's fully supported by Apple, as it's used in both iOS and Mac OS.
- It has continuous support by developers in the whole world and new features are always added to it.

SQLite is an **embedded implementation of SQL**. SQL stands for Structured Query Language and is a standard language to work with relational databases. SQLite can be embedded inside any application, so there is no need for a separate process running the database instance. It follows the principals of a **Relational Database Management System (RDBMS)**. Inside a RDBMS data is stored inside tables and the relationship between this data is also stored inside tables.

A good example for this is the relationship between a person and his address. A person has typically some properties like first name, last name, birthdate and much more. An address has properties like street name, street number, etc... But there is also a relationship between them, a person can have several addresses. In the database this is achieved by adding a foreign key to the address object. This foreign key points to the primary key of the person it belongs to. This has also as advantage that when a person is deleted a warning is given about an associated address. So it becomes possible to also delete the address if needed.



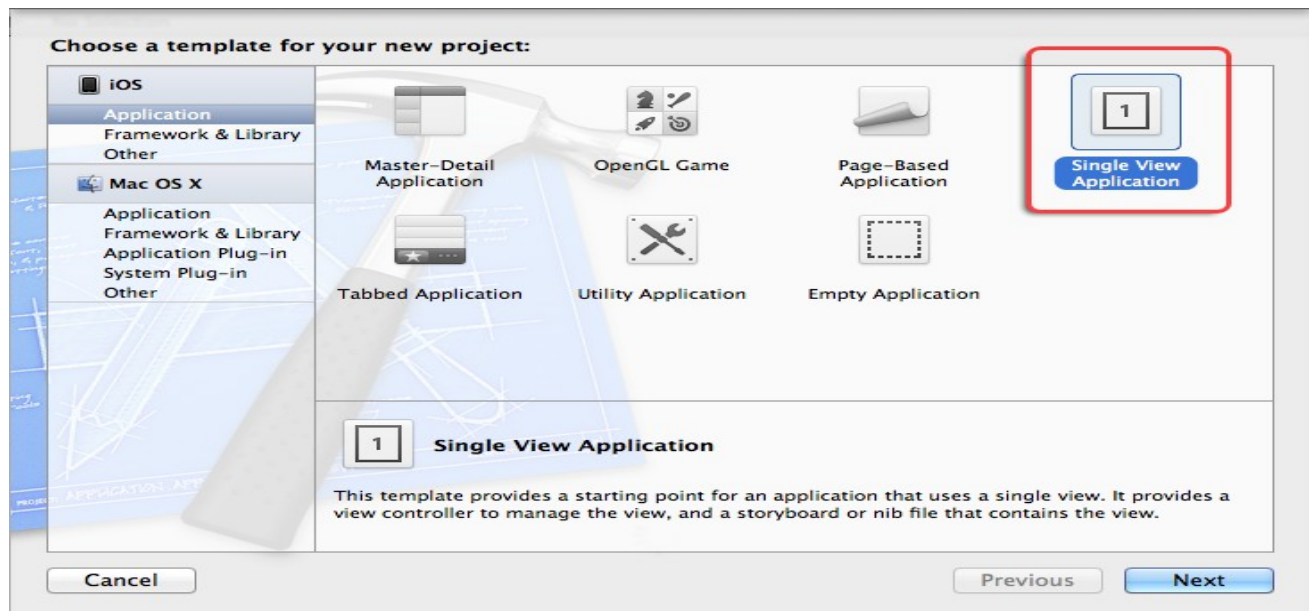
The Core SQLite functions

Lets first start with a list of the most used SQLite functions and describe their purpose:

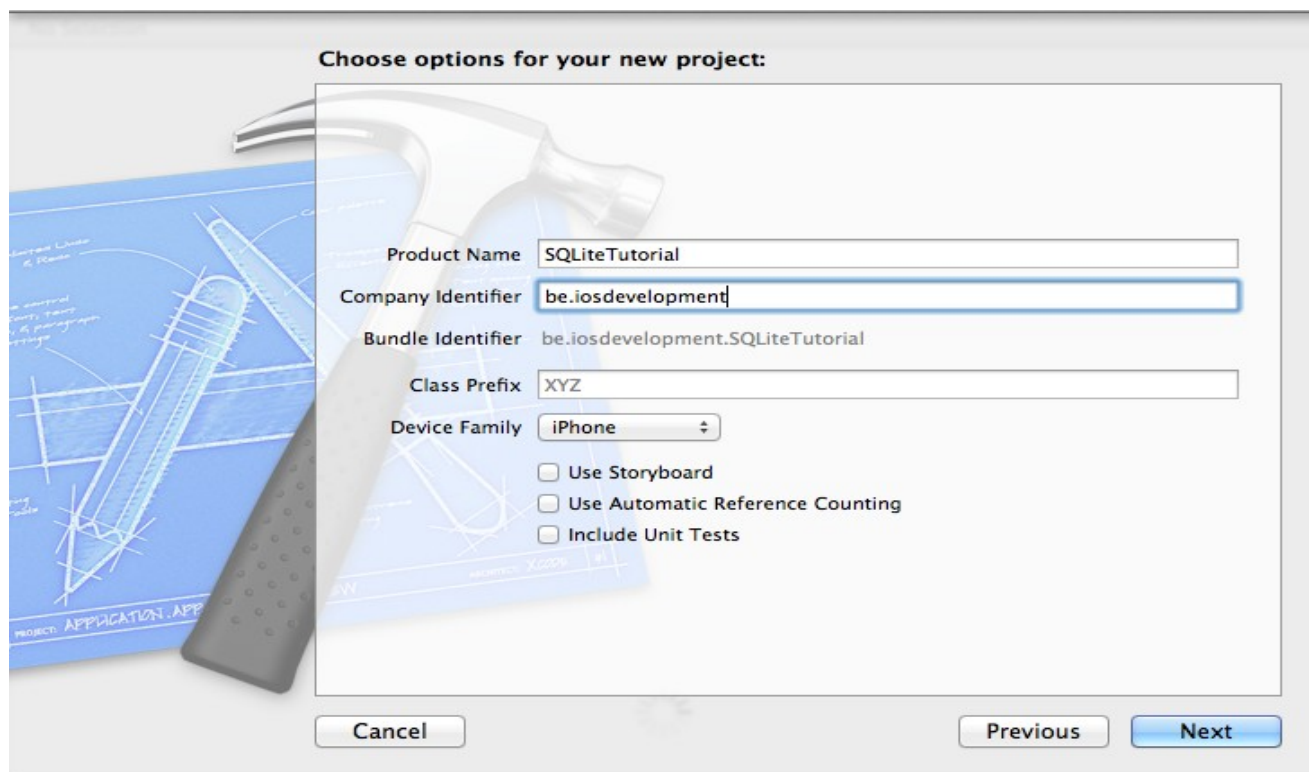
- **sqlite3_open():** This function **creates** and **opens** an empty database with the specified filename argument. If the database already exists it will only open the database. Upon return the second argument will contain a handle to the database instance.
- **sqlite3_close():** This function should be used to **close** a previously opened SQLite database connection. It will free all system resources associated with the database connection.
- **sqlite3_prepare_v2():** To execute an SQL statement it first needs to be **compiled into byte-code** and that is exactly what this function is doing. It basically transforms an SQL statement written in a string to an executable piece of code.
- **sqlite3_step():** Calling this function will **execute** a previously prepared SQL statement.
- **sqlite3_finalize():** This function **deletes** a previously prepared SQL statement from memory.
- **sqlite3_exec():** Combines the functionality of sqlite3_prepare_v2(), sqlite3_step() and sqlite3_finalize() into a single function call.
- **sqlite3_column_<type>():** This routine returns **information about a single column** of the current result row of a query. Typical values for <type> are text and int. It is important to note that the column indexes are zero based.

Setting up the project

1. Create a new project and choose Single View Application.

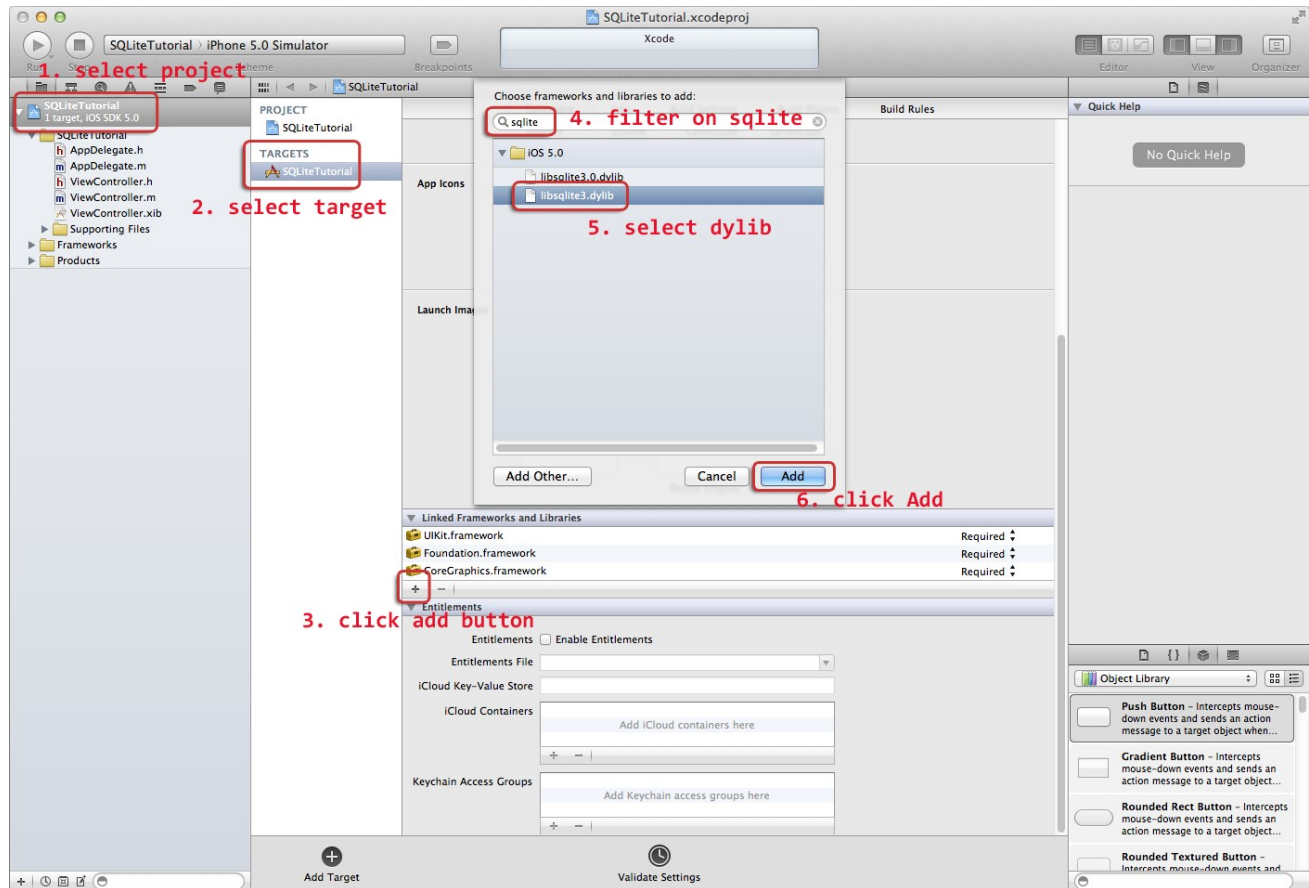


2. Name the application "SQLiteTutorial" and make sure to uncheck all options.



3. Now add the SQLite framework called "libsqlite3.dylib". To do so select the SQLiteTutorial project inside the navigation area and then select the SQLiteTutorial target inside the editor area. Scroll to the section called "Linked Frameworks and Libraries" and click the add button. Filter

the frameworks by typing "sqlite". Select "libsqlite3.dylib" and press add. We will also notice a framework called "libsqlite3.0.dylib" this is the physical library, "libsqlite3.dylib" is just a symbolic link to the latest version.



4. Add a new file to the project. Choose the Cocoa Touch Objective-C template and call this new file "DataController".

5. Open the header file "DataController.h" and add an import for "sqlite3.h" and a data member to store a handle to the database:

```
1. #import <Foundation/Foundation.h>
2. #import <sqlite3.h>
3.
4. @interface DataController : NSObject
5. {
```

```
6.    sqlite3 *databaseHandle;
7. }
8.
9. -(void)initDatabase;
10.
11. @end
```

6. Now it is time to start adding some entities. Again choose for the Cocoa Touch Objective-C template and call the first entity Address. The Address entity will be holding a street name and a street number.

```
1. #import <Foundation/Foundation.h>
2. @interface Address : NSObject
3. {
4.     NSString *streetName;
5.     NSNumber *streetNumber;
6. }
7.
8. @property (nonatomic,
9.     retain) NSString* streetName;
9. @property (nonatomic,
10.     retain) NSNumber* streetNumber;
10.
11. -(id)initWithStreetName:(NSString*)aStreetName
12.     andStreetNumber:(NSNumber*)streetNumber;
13.
14. @end
```

```
1. #import "Address.h"
```

```
2.
3. @implementation Address
4.
5. @synthesize streetName;
6. @synthesize streetNumber;
7.
8. // Custom initializer
9. -(id)initWithStreetName:(NSString*)aStreetName
10.    andStreetNumber:(NSNumber*)aStreetNumber
11. {
12.     self = [super init];
13.     if(self) {
14.         self.streetName = aStreetName;
15.         self.streetNumber = aStreetNumber;
16.     }
17.     return self;
18. }
19.
20. // Cleanup all contained properties
21. -(void)dealloc {
22.     [self.streetName release];
23.     [self.streetNumber release];
24.     [super dealloc];
25. }
26.
27. @end
```

7. The next entity to add will be the Person entity. It will contain a first name, last name and birthday. The Person class will also contain an Address object, this will be reflected in the SQLite database by using a foreign key inside the address table, but that will become more clear when creating the database. Again a custom initializer was added for convenience and a dealloc method will clean up the object:

```
1. #import <Foundation/Foundation.h>
2. #import "Address.h"
3. @interface Person : NSObject
4. {
5.     NSString *firstName;
6.     NSString *lastName;
7.     NSDate *birthday;
8.     Address *address;
9. }
10.
11. @property (nonatomic, retain) NSString* firstName;
12. @property (nonatomic, retain) NSString* lastName;
13. @property (nonatomic, retain) NSDate* birthday;
14. @property (nonatomic, retain) Address* address;
15.
16. -(id)initWithFirstName:(NSString*)aFirstName
17.     andLastName:(NSString*)aLastName
18.     andBirthday:(NSDate*)aBirthday
19.     andAddress:(Address*)anAddress;
20.
21. @end
```

```
1. #import "Person.h"
2.
3. @implementation Person
4.
5. @synthesize firstName;
6. @synthesize lastName;
7. @synthesize birthday;
8. @synthesize address;
9.
10. // Custom initializer
11. -(id)initWithFirstName:(NSString*)aFirstName
12.     andLastName:(NSString*)aLastName
13.     andBirthday:(NSDate*)aBirthday
14.     andAddress:(Address*)anAddress
15. {
16.     self = [super init];
17.     if(self) {
18.         self.firstName = aFirstName;
19.         self.lastName = aLastName;
20.         self.birthday = aBirthday;
21.         self.address = anAddress;
22.     }
23.     return self;
24. }
25.
26. // Cleanup all contained objects
27. - (void)dealloc {
```



```
28. [self.firstName release];
29. [self.lastName release];
30. [self.birthday release];
31. [self.address release];
32. [super dealloc];
33. }
34.
35. @end
```

Now that are basic building blocks are in-place it is time to start working with the SQLite database.

Creating an SQLite database

The SQLite database for this sample application will be stored inside the Documents folder of the application sandbox and will be called "sqlite.db". To do this add the method "initDatabase" to the DataController.

```
1. // Method to open a database, the database will be created if it doesn't exist
2. -(void)initDatabase
3. {
4.     // Create a string containing the full path to the sqlite.db inside the documents folder
5.     NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
6.     NSString *documentsDirectory = [paths objectAtIndex:0];
7.     NSString *databasePath = [documentsDirectory
8.     stringByAppendingPathComponent:@"sqlite.db"];
9.     // Check to see if the database file already exists
```

```
10.  bool
    databaseAlreadyExists = [[NSFileManager defaultManager] fileExistsAtPath:databasePath];
11.
12.  // Open the database and store the handle as a data member
13.  if (sqlite3_open([databasePath UTF8String], &databaseHandle) == SQLITE_OK)
14.  {
15.      // Create the database if it doesn't yet exists in the file system
16.      if (!databaseAlreadyExists)
17.      {
18.          // Create the PERSON table
19.          const char *sqlStatement = "CREATE TABLE IF NOT EXISTS PERSON (ID
    INTEGER PRIMARY KEY AUTOINCREMENT, FIRSTNAME TEXT, LASTNAME
    TEXT, BIRTHDAY DATE)";
20.          char *error;
21.          if (sqlite3_exec(databaseHandle,
    sqlStatement, NULL, NULL, &error) == SQLITE_OK)
22.          {
23.              // Create the ADDRESS table with foreign key to the PERSON table
24.              sqlStatement = "CREATE TABLE IF NOT EXISTS ADDRESS (ID INTEGER
    PRIMARY KEY AUTOINCREMENT, STREETNAME TEXT, STREETNUMBER
    INT, PERSONID INT, FOREIGN KEY(PERSONID) REFERENCES PERSON(ID))";
25.          if (sqlite3_exec(databaseHandle,
    sqlStatement, NULL, NULL, &error) == SQLITE_OK)
26.          {
27.              NSLog(@"Database and tables created.");
28.          }
29.          else
```

```
30.      {
31.          NSLog(@"Error: %s", error);
32.      }
33.  }
34.  else
35.  {
36.      NSLog(@"Error: %s", error);
37.  }
38.  }
39.  }
40. }
```

Lets highlight some points inside this method:

1. A full path is created that points to sqlite.db inside the documents folder of the application. In case when running inside the simulator this will be inside the *folder* ~Library/Application Support/iPhone Simulator
2. Check if the database file already exists inside the file system.
3. Open a connection to the database and store the databaseHandle for later use.
4. If the database did not exist inside the file system then the tables will be created.
5. The table PERSON is created with a auto-incrementing primary key.
6. The table ADDRESS is also created with an auto-incrementing primary key and a foreign key constraint set to the ID of the PERSON table and will be called PERSONID.

It is also important to close the database connection once the DataController gets released. To do this simply override the "dealloc" method of the class DataController:

```
1. // Close the database connection when the DataController is disposed
2. - (void)dealloc {
3.     sqlite3_close(databaseHandle);
```

```
4.    [super dealloc];  
5. }
```

To verify this piece of code update the method "viewDidLoad" from the file "ViewController.m" so that it looks like:

```
1. - (void)viewDidLoad  
2. {  
3.    [super viewDidLoad];  
4.  
5.    // Create datacontroller and initialize database  
6.    DataController *dataController = [[DataController alloc] init];  
7.    [dataController initDatabase];  
8.    [dataController release];  
9. }
```

There is also an easy trick to verify if the database was correctly created. Open the terminal and launch sqlite3 with the full path to the "sqlite.db". Once sqlite3 is started execute the command .schema and see the tables and their columns:

```
Last login: Fri Feb 10 19:34:52 on ttys001
```

```
Blackwing:~ lucwollants$ sqlite3 /Users/lucwollants/Library/Application\ Support/iPhone\ Simulator/5.0/Applications/D2BC14F2-F260-40C8-A57D-D6A7F337B612/Documents/sqlite.db
```

```
SQLite version 3.7.5
```

```
Enter ".help" for instructions
```

```
Enter SQL statements terminated with a ";"
```

```
sqlite> .schema
```

```
CREATE TABLE ADDRESS (ID INTEGER PRIMARY KEY AUTOINCREMENT,  
STREETNAME TEXT, STREETNUMBER INT, PERSONID INT, FOREIGN  
KEY(PERSONID) REFERENCES PERSON(ID));
```

```
CREATE TABLE PERSON (ID INTEGER PRIMARY KEY AUTOINCREMENT,  
FIRSTNAME TEXT, LASTNAME TEXT, BIRTHDAY DATE);
```

```
sqlite>
```

Storing values inside the SQLite database

Next part to implement is a method to insert a Person and his associated Address inside the database. To do so a new method called "insertPerson" needs to be created inside the DataController:

```
1. // Method to store a person and his associated address  
2. -(void)insertPerson:(Person*)person  
3. {  
4.     // Create insert statement for the person  
5.     NSString *insertStatement = [NSString stringWithFormat:@"INSERT INTO  
        PERSON (FIRSTNAME, LASTNAME, BIRTHDAY) VALUES  
        (\"%@\", \"%@\", \"%@\")", person.firstName, person.lastName, person.birthday];  
6.  
7.     char *error;  
8.     if ( sqlite3_exec(databaseHandle, [insertStatement  
        UTF8String], NULL, NULL, &error) ==SQLITE_OK)  
9.     {  
10.        int personID = sqlite3_last_insert_rowid(databaseHandle);  
11.  
12.        // Create insert statement for the address  
13.        insertStatement = [NSString stringWithFormat:@"INSERT INTO ADDRESS  
        (STREETNAME, STREETNUMBER, PERSONID) VALUES
```

```
(\\"%@\\", \\"%@\\", \\"%d\\")", person.address.streetName, person.address.streetNumber,
personID];
14.     if ( sqlite3_exec(databaseHandle, [insertStatement
        UTF8String], NULL, NULL, &error) ==SQLITE_OK)
15.     {
16.         NSLog(@"Person inserted.");
17.     }
18.     else
19.     {
20.         NSLog(@"Error: %s", error);
21.     }
22. }
23. else
24. {
25.     NSLog(@"Error: %s", error);
26. }
27. }
```

Lets discuss the previous code snippet:

1. Create an insert statement for the person object.
2. Execute the insert statement for the person by calling "sqlite3_exec".
3. Get the ID of the last inserted row by calling "sqlite3_last_insert_rowid". This ID needs to be pasted as the foreign key for the address object.
4. Create the insert statement for the address object. Note that the foreign key is also passed in.
5. Execute the insert statement for the address by calling "sqlite3_exec".

Again it is possible to test the new code by updating the method "viewDidLoad":

```
1. // Create address and person objects
2. Address *address = [[Address alloc] initWithStreetName:@"Infinite
   Loop" andStreetNumber:[NSNumber numberWithInt:1]];
3. NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
4. [dateFormatter setDateFormat:@"%yyyy-MM-dd"];
5. NSDate *birthday = [dateFormatter dateFromString: @"1955-02-24"];
6. Person *person = [[Person
   alloc] initWithFirstName:@"Steve" andLastName:@"Jobs"andBirthday:birthday
   andAddress:address];
7.
8. // Insert the person
9. [dataController insertPerson:person];
10.
11. // Cleanup
12. [dateFormatter release];
13. [address release];
14. [person release];
15. [DataController release];
```

Testing the result of this action can be done again from the command line with sqlite3 Terminal command:

Enter SQL statements terminated with a ";"

```
sqlite> SELECT * FROM ADDRESS;
```

```
1|Infinite Loop|1|
```

```
sqlite> SELECT * FROM PERSON;
```

```
1|Steve|Jobs|1955-02-23 23:00:00 +0000
```

Retrieving values from the SQLite database

Now it is time to programmatically retrieve values from the database. This can be done by using the "sqlite3_step" function. The DataController implementation file needs to be updated with a method called "getAddressByPersonID" and "getPersons". The method "getAddressByPersonID" is a helper method to get an address associated with a person. The method "getPersons" returns an array of all persons inside the database.

```
1. // Get an array of all persons stored inside the database
2. -(NSArray*)getPersons
3. {
4.     // Allocate a persons array
5.     NSMutableArray *persons = [[NSMutableArray alloc] init];
6.
7.     // Create the query statement to get all persons
8.     NSString *queryString = [NSString stringWithFormat:@"SELECT ID,
FIRSTNAME, LASTNAME, BIRTHDAY FROM PERSON"];
9.
10.    // Prepare the query for execution
11.    sqlite3_stmt *statement;
12.    if (sqlite3_prepare_v2(databaseHandle, [queryString UTF8String], -
1, &statement, NULL) == SQLITE_OK)
13.    {
14.        // Iterate over all returned rows
15.        while (sqlite3_step(statement) == SQLITE_ROW) {
16.
17.            // Get associated address of the current person row
18.            int personID = sqlite3_column_int(statement, 0);
19.            Address *address = [self getAddressByPersonID:personID];
```



```
20.
21.     // Convert the birthday column to an NSDate
22.     NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
23.     dateFormatter.dateFormat = @"yyyy-MM-dd HH:mm:ss Z";
24.     NSString *birthdayAsString = [NSString stringWithUTF8String:
(char*)sqlite3_column_text(statement, 3)];
25.     NSDate *birthday = [dateFormatter dateFromString: birthdayAsString];
26.     [dateFormatter release];
27.
28.     // Create a new person and add it to the array
29.     Person *person = [[Person alloc] initWithFirstName:
[NSString stringWithUTF8String:(char*)sqlite3_column_text(statement, 1)]
andLastName:[NSString stringWithUTF8String:
(char*)sqlite3_column_text(statement, 2)]
andBirthday:birthday
andAddress:address];
30.
31.     [persons addObject:person];
32.
33.
34.
35.     // Release the person because the array takes ownership
36.     [person release];
37. }
38.     sqlite3_finalize(statement);
39. }
40. // Return the persons array and mark for autorelease
41.     return [persons autorelease];
42. }
```

12. Debug and deploy application

1. Join the Apple iOS Developer Program.

We can log in using our existing Apple ID or create an Apple ID. The Apple Developer Registration guides we through the necessary steps.

2. Register the Unique Device Identifier (UDID) of the device.

This step is applicable only if we are deploying our application to an iOS device and not the Apple App Store. If we want to deploy our application on several iOS devices, register the UDID of each device.

3. Obtain the UDID of our iOS device

- Connect the iOS device to our development computer and launch iTunes. The connected iOS device appears under the Devices section in iTunes.
- Click the device name to display a summary of the iOS device.
- In the Summary tab, click Serial Number to display the 40-character UDID of the iOS device.

4. Register the UDID of our device

- Log in to the iOS Provisioning Portal using our Apple ID and register the device's UDID.
- Generate a Certificate Signing Request (CSR) file (*.certSigningRequest). We generate a CSR to obtain a iOS developer/distribution certificate. We can generate a CSR by using Keychain Access on Mac or Open SSL on Windows. When we generate a CSR we only provide our user name and email address; we don't provide any information about our application or device.
- Generating a CSR creates a public key and a private key as well as a *.cert Signing Request file. The public key is included in the CSR, and the private key is used to sign the request. For more information on generating a CSR, see Generating a certificate signing request. Generate an iOS developer certificate or an iOS distribution certificate (*.cer), as required.

5. Generate an iOS developer certificate

Log in to the iOS Provisioning Portal using our Apple ID, and select the Development tab.

- Click Request Certificate and browse to the CSR file that we generated and saved on our computer (step 3).
- Select the CSR file and click Submit.
- On the Certificates page, click Download.
- Save the downloaded file (*.developer_identity.cer).

6. Generate an iOS distribution certificate

- Log in to the iOS Provisioning Portal using our Apple ID, and select the Distribution tab
- Click Request Certificate and browse to the CSR file that we generated and saved on our computer (step 3).
- Select the CSR file and click Submit.
- On the Certificates page, click Download.
- Save the downloaded file (*.distribution_identity.cer).

1 Convert the iOS developer certificate or the iOS distribution certificate to a P12 file format (*.p12).

6. Generate the Application ID by following these steps:

- Log in to the iOS Provisioning Portal using our Apple ID.
- Go to the App IDs page, and click New App ID.
- In the Manage tab, enter a description for our application, generate a new Bundle Seed ID, and enter a Bundle Identifier.
- Every application has a unique Application ID, which we specify in the application descriptor XML file. An Application ID consists of a ten-character "Bundle Seed ID" that Apple provides and a "Bundle Identifier" suffix that we specify. The Bundle Identifier we specify must match the application ID in the application descriptor file. For example, if our

Application ID is com.myDomain.*, the ID in the application descriptor file must start with com.myDomain.

- Generate a Developer Provisioning Profile file or a Distribution Provisioning Profile File (*.mobileprovision).

7. **Generate a Developer Provisioning Profile**

- Log in to the iOS Provisioning Portal using our Apple ID.
- Go to Certificate > Provisioning, and click New Profile.
- Enter a profile name, select the iOS developer certificate, the App ID, and the UDIDs on which we want to install the application.
- Click Submit.
- Download the generated Developer Provisioning Profile file (*.mobileprovision) and save it on our computer.

8. **Generate a Distribution Provisioning Profile**

- Log in to the iOS Provisioning Portal using our Apple ID.
- Go to Certificate > Provisioning, and click New Profile.
- Enter a profile name, select the iOS distribution certificate and the App ID. If we want to test the application before deployment, specify the UDIDs of the devices on which we want to test.
- Click Submit.
- Download the generated Provisioning Profile file (*.mobileprovision) and save it on our computer.

Files to select when we test, debug, or install an iOS application

To run, debug, or install an application for testing on an iOS device, we select the following files in the Run/Debug Configurations dialog box:

- iOS developer certificate in P12 format (step 5)
- Application descriptor XML file that contains the Application ID (step 6)

- Developer Provisioning Profile (step 7)

For more information, see Debug an application on an Apple iOS device and Install an application on an Apple iOS device.

Files to select when we deploy an application to the Apple App Store

To deploy an application to the Apple App Store, select the Package Type in the Export Release Build dialog box as Final Release Package For Apple App Store, and select the following files:

- iOS distribution certificate in P12 format (step 5)
- Application descriptor XML file that contains the Application ID (step 6).

Note: We can't use a wildcard Application ID while submitting an application to the Apple App Store.

- Distribution Provisioning Profile

14 Publishing App in App Store

The App Store review process is a black box for the most part, that doesn't mean that we can't prepare ourselves and our application for Apple's review process. Apple provides guidelines to help us stay within the sometimes invisible boundaries of what is and isn't allowed in the App Store.

14.1 Testing

An application isn't necessarily ready when we've written the last line of code or implemented the final feature of the application's specification. The family of iOS devices has grown substantially over the past years and it is important to test our application on as many iOS devices as we can lay our hands on. The iOS Simulator is a great tool, but it runs on our Mac, which has more memory and processing power than the phone in our pocket. Apple's Review Process isn't airtight, but it is very capable of identifying problems that might affect our application's user experience. If our application crashes from time to time or it becomes slow after ten minutes of use, then we have some work to do before submitting it to the App Store. Even if Apple's review team doesn't spot the problem, our users will. If the people using our

application are not pleased, they will leave bad reviews on the App Store, which may harm sales or inhibit downloads.

14.2 Rules and Guidelines

Our application ...

- doesn't crash.
- shouldn't use private API's.
- shouldn't replicate the functionality of native applications.
- should use In App Purchase for in-app (financial) transactions.
- shouldn't use the camera or microphone without the user's knowledge.
- only uses artwork that we have the copyright of or we have permission to use.

14.2.1 App ID

Every application needs an App ID or application identifier. There are two types of application identifiers, (1) an **explicit App ID** and (2) a **wildcard App ID**. A wildcard App ID can be used for building and installing multiple applications. Despite the convenience of a wildcard App ID, an explicit App ID is **required** if our application uses iCloud or makes use of other iOS features, such as Game Center, Apple Push Notifications, or In App Purchase.

14.2.2 Distribution Certificate

To submit an application to the App Store, we need to create an iOS provisioning profile for distribution. To create such a provisioning profile, we first need to create a distribution certificate. The process for creating a distribution certificate is very similar to creating a development certificate. If we have tested our application on a physical device, then we are probably already familiar with the creation of a development certificate.

14.2.3 Provisioning Profile

Once we've created an App ID and a distribution certificate, we can create an iOS provisioning profile for distributing our application through the App Store. Keep in mind that we cannot use the same provisioning profile that we use for ad hoc distribution. We need to create a

separate provisioning profile for App Store distribution. If we use a wildcard App ID for our project, then we can use the same provisioning profile for multiple applications.

14.2.4 Build Settings

With the App ID, distribution certificate, and provisioning profile in place, it is time to configure our target's build settings in Xcode. This means selecting the target from the list of targets in Xcode's **Project Navigator**, opening the **Build Settings** tab at the top, and updating the settings in the **Code Signing** section to match the distribution provisioning profile we created earlier. Newly added provisioning profiles are sometimes not immediately visible in the **Code Signing** section of the build settings. Quitting and relaunching Xcode remedies this issue.

14.2.5 Deployment Target

Each target in an Xcode project, has a deployment target, which indicates the minimum version of the operating system that the application can run on. It is up to we to set the deployment target, but keep in mind that modifying the deployment target is not something we can do without consequences once our application is in the App Store. If we increase the deployment target for an update of our application, then users who already purchased our application but don't meet the new deployment target, cannot run the update.

14.3 Assets

14.3.1 Icons

We need to make sure that our application ships with the correct sizes of the artwork.

- iTunes Artwork: 1024px x 1024px (required)
- iPad/iPad Mini: 72px x 72px **and** 114px x 114px (required)
- iPhone/iPod Touch: 57px x 57px **and** 114px x 114px (required)
- Search Icon: 29px x 29px **and** 58px x 58px (optional)
- Settings Application: 50px x 50px **and** 100px x 100px (optional)

14.3.2 Screenshots

Each application can have up to five screenshots and we must provide at least one. If we are developing a universal application, then we need to provide separate screenshots for iPhone/iPod Touch and iPad/iPad Mini. In addition, we can optionally include separate screenshots for the 3.5" and the 4" screen sizes of the iPhone/iPod Touch. This is quite a bit of work and we want to make sure that the screenshots show our application from its best side.

14.3.3 Metadata

Before we submit our application, it is a good idea to have our application's metadata at hand. This includes (1) our application's name, (2) the version number, (3) the primary (and an optional secondary) category, (4) a concise description, (5) keywords, and (6) a support URL.

14.4. Submission Preparation

1. The submission process has become much easier since the release of Xcode 4. We can now validate and submit an application using Xcode, for example. First, however, we need to create our application in iTunes Connect.
2. The **App Name**, which needs to be unique, is the name of our application as it will appear in the App Store. This can be different than the name that is displayed below our application icon on the home screen, but it is recommended to choose the same name.
3. The **SKU Number** is a unique string that identifies our application. I usually use the application's bundle identifier. The last piece of information is the **Bundle ID** of our application. This means selecting the (wildcard or explicit) App ID that we created earlier from the drop down menu.
4. Specifying Price and Availability

5. Once our application's metadata is submitted, we will be presented with a summary of our application. Under **Versions**, we should see the version that we submitted a moment ago.
6. To submit our application, we need to create an **archive** of our application. We can only create an archive by building our application on a **physical device**.. Select the archive from the list and click the **Distribute** button on the right. From the options we are presented with, select **Submit to the iOS App Store**. After entering our iOS developer account credentials and selecting the **Application** and **Code Signing Identity**, the application binary is uploaded to Apple's servers. During this process, our application is also validated. If an error occurs during the validation, the submission process will fail. The validation process is very useful as it will tell us if there is something wrong with our application binary that would otherwise result in a rejection by the App Store review team.