

INTRODUCTION TO

Sensible TableView 3.0



FIRST EDITION

© 2012, Sensible Cocoa. Edition 1.1

Sensible Cocoa®, Sensible TableView®, and all other trademarks used in this book are the properties of their respective owners.
www.sensiblecocoa.com

Getting Started

Congratulations on choosing Sensible TableView for your next project, you are definitely in for a treat!

In this chapter, you will be taken through a quick journey of what the Sensible TableView framework has to offer. You will also be given a tutorial on how to set up STV and get your applications up and running in a matter minutes.

What is Sensible TableView?

Sensible TableView (STV) is a framework that drastically simplifies iOS table view development, while focusing on delivering a simple and enjoyable developer experience. Focusing on such a great developer experience is perhaps one of the key reasons why STV has become so popular in such a short period of time.

Another key reason behind STV's popularity has been undoubtedly the unprecedented amount of time it saves its users. Based on countless user testimonials and feedback, we have found that developers save on average about 70% of their development time when they start using STV in their applications. What this means for you is that an app that would normally take weeks or even months for you to develop alone, can now be conceived in just a matter of days when you use STV.

And this is only half the story. As any experienced developer would definitely agree, the real devil is in the application's

maintenance. It is here where STV becomes really indispensable. Since STV keeps your code very short, simple and straight forward, maintenance suddenly becomes an enjoyable trivial task.

So what's the catch, you may ask? Fortunately, there is no catch. Since day one, it has been integral to STV's architecture to fully expose everything the iOS framework has to offer. This means that you do not lose any flexibility while using STV. Every single thing that can be done with normal table views can still be done with STV, only a lot easier. Have a custom UITableView subclass? No problem, STV flawlessly integrates with that. A custom cell? Even better! Custom view controllers? STV's bread and butter!

All this probably sounds too good to be true, and you should definitely not take our word for it! In the next few chapters, you will get to experience first hand what it means to have STV be part of your project. Whether you are developing a simple Core Data application, dealing with complex web services or even iCloud, you will get to see how STV elegantly reduces the amount of work you have to do to an absolute bare minimum!

What's new in STV 3.0?

STV 3.0 has been by far the largest upgrade STV has had to date, perhaps with more new features than all the previous upgrades combined! One serious challenge with adding all these new features was maintaining STV's most important (and much beloved) core value: *simplicity*. We understood how integral this core value is to STV, and we had zero tolerance for any solution that undermined this value even by the slightest amount.

After a lot of research, we ended up virtually reimplementing STV from the ground up. We used everything that we had learned from our users since STV's inception, and we made sure to stick to all our initial core values that had made STV popular in the first place. We also did one more thing that was not available in any previous STV version: *framework extensions*. With framework extensions, STV users can now easily extend STV itself to provide any functionality that is not currently available out of the box. As a matter of fact, we used

STV 3.0's framework extensions ourselves to add features such as Core Data, web services, and iCloud integration! You can read more on this exciting new functionality in the chapter titled: *'Extending STV'*.

Fortunately, all this hard work really paid off at the end. The feedback we got from everyone that had the chance to experience STV 3.0 was simply extraordinary! Not only were they blown away by the new features, but they almost unanimously raved about the fact that STV 3.0 is now even easier to use, sometimes by several orders of magnitude!

The following is a non-comprehensive list of STV 3.0's most notable new features (all new features will be discussed in great detail throughout the book):

- **Web Service Integration.** STV now binds and consumes web services exactly as it used to bind to classes and Core Data entities. It now even fully integrates with parse.com to deliver the ultimate ease of use in web service application development.
- **iCloud Integration.** STV 3.0 supports key-value binding to iCloud out of the box. Deploy the same app on several devices and watch as the data automatically synchronize between them as soon as it's changed!
- **User-Defaults Integration.** Using the new `SCUserDefaultsDefinition` and a single line of code, you can now use STV to read and write data to the application's user defaults.
- **Asynchronous Data Loading.** STV 3.0 now supports batched loading from arrays, Core Data, web services, and even your own custom framework extensions!
- **Themes.** As demonstrated with our bundled sample application, you can now style your application using CSS-like styling via theme files. To make things even easier for you, we've made an agreement with AppDesignVault.com to create STV-ready theme files for their designs. Now all you need is place a single line

of code and have your whole application styled! (we've also bundled several free themes with the STV package).

- **Actions.** Actions will simply revolutionize everything all over again. By virtually eliminating the need for using delegates, customizing STV using actions delivers unprecedented clarity and ease of use.
- **Dynamic Expand-Collapse Sections.** Sections can now dynamically expand and collapse.
- **Framework extensions.** Framework extensions open the door wide open for you to extend STV's core functionality. Think of framework extensions as STV plugins.
- **Data Fetch Options.** You can now fully control how data is fetched, no matter where it's being fetched from. This includes sorting, filtering, and the use of `NSPredicate` where applicable.
- **Enhanced `SCViewController` and `SCTableViewController`.** By popular demand, `SCViewController` and `SCTableViewController` have been redesigned for your own use (as opposed to earlier STV versions where they were only intended to be used internally by STV). As you will see in the following chapters (and in the bundled samples), these view controllers are extremely convenient when used with STV.

- **Additional Special Views.** Special views such as Pull-to-Refresh and InputAccessoryView now come out-of-the-box.
- **Additional Special Cells.** We've added new special cell such as the Expand-Collapse cell, and the Load-More cell.
- **Much, much more!**

Setting up STV

As you'll see in this section, STV 3.0 is fairly simple and straight forward to set up. Once you follow the next few steps, you should have your STV app up and running in virtually no time.

STV 3.0 minimum requirements

- Xcode 4.3 or later. It is always recommended to use the latest Xcode version, so make sure you always get the latest updates from the Mac App Store.
- iOS 4.0 or later deployment target. Please note that applications developed with STV 3.0 will not work on devices using iOS versions prior to 4.0. Latest surveys show that more than 70% of all devices have upgraded to iOS 5.0, and about 18% are using iOS 4.0.

STV 3.0 formats

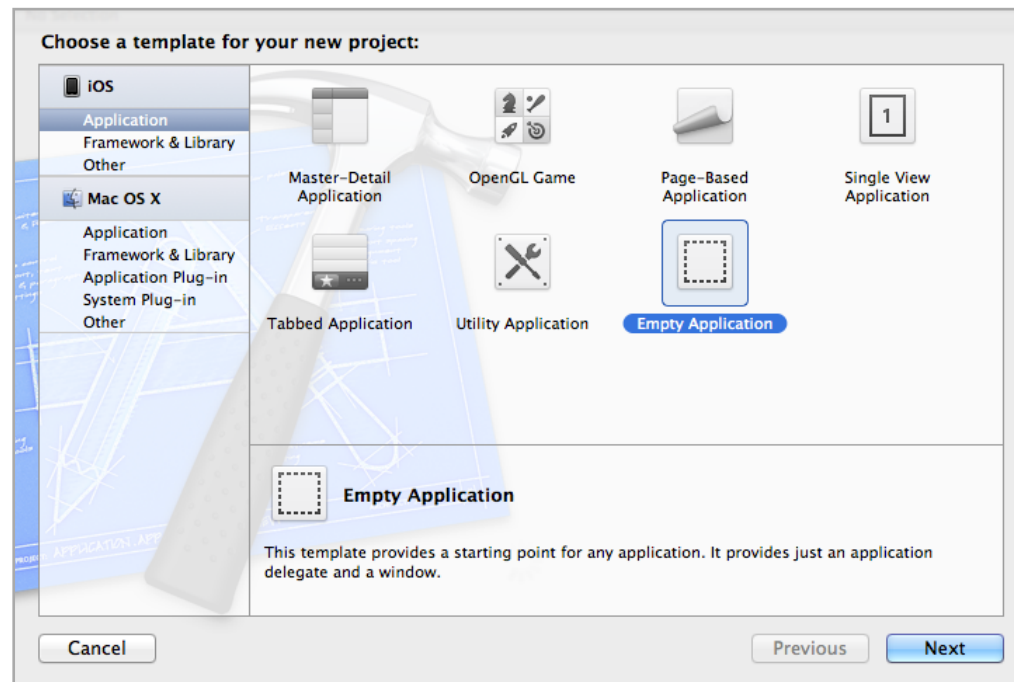
STV 3.0 can be used in your project in two basic formats:

- **As a static framework.** Using STV as a static framework is very simple, and should generally be your format of choice.
- **As source code.** Using STV in source code format requires a few more steps than when using a static framework, but may prove really useful if you need to trace into STV's code. It is also recommended (but not required) to use this format when developing custom STV framework extensions.

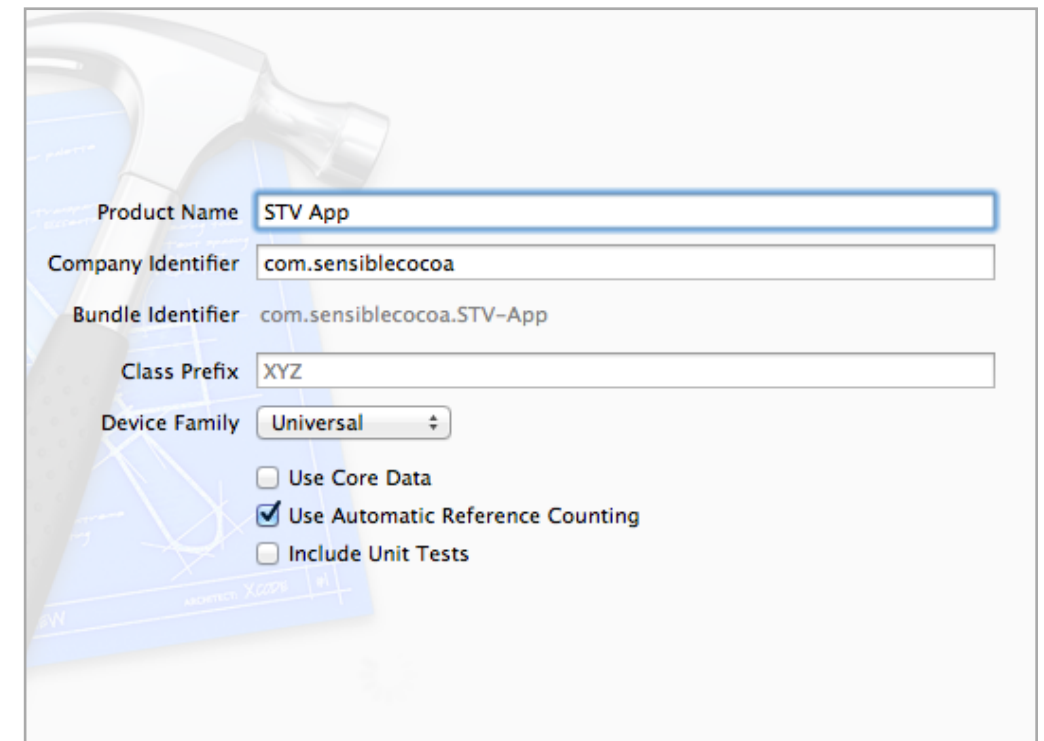
STV's source code is only available with the STV Pro version.

Using STV as a static framework

1. Launch Xcode, then create a new project using File->New->Project...
2. You can now choose any project template you wish from the iOS application tab. Since STV generally doesn't need most of the additional code provided by these templates, using the "Empty Application" template is often the most convenient option to use.



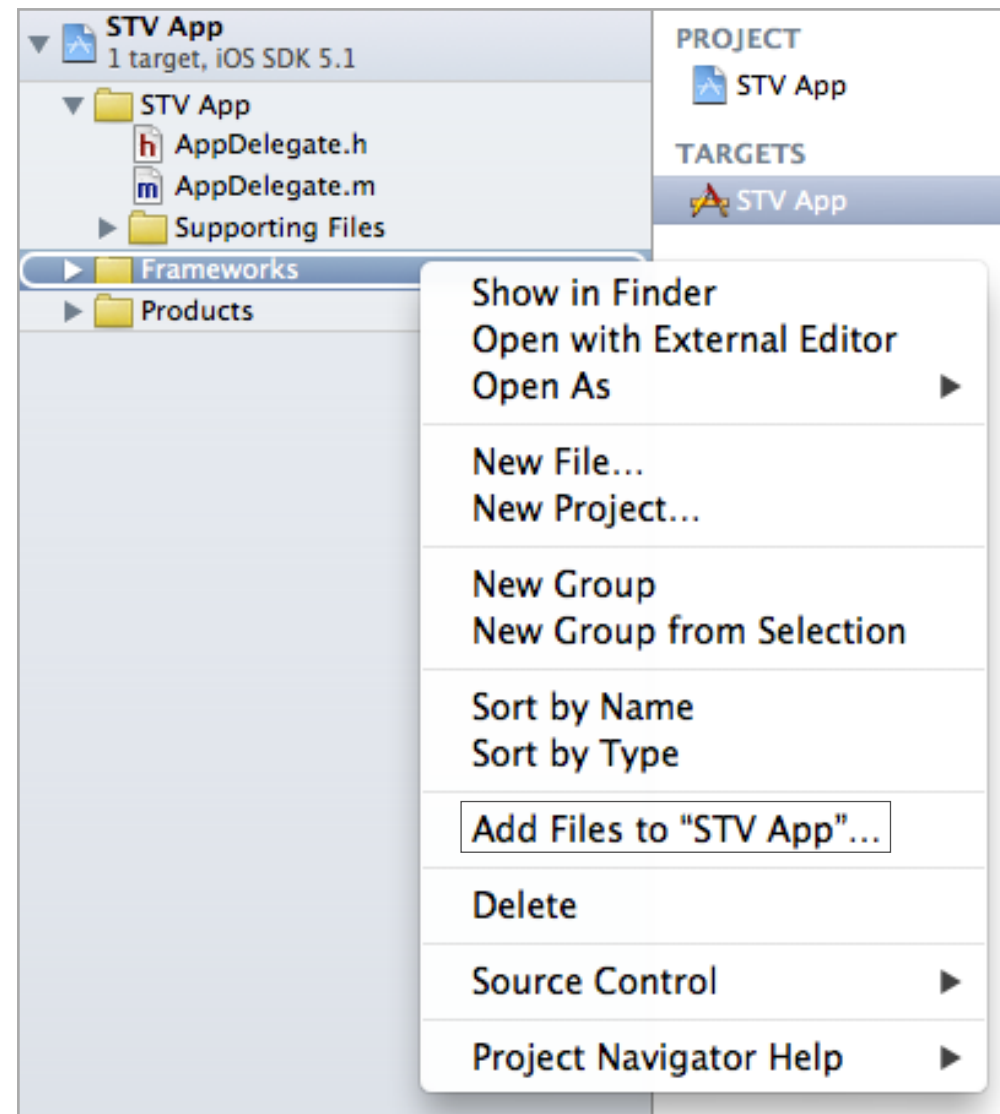
3. In the next form, enter "STV App" as the project name, then set the other options as shown below. Make sure to check "Use Core Data" if your project is a Core Data application.



It is worth noting here that STV fully supports using Storyboards whenever you wish to do so. You must be careful however because by using storyboards, you can only deploy your application on iOS 5.0 devices. This is why we will usually be omitting storyboards in our examples as long as we keep supporting iOS 4.0 as a minimum deployment target. Since STV typically auto-generates all detail views for you, you should have very little need for storyboards anyways.

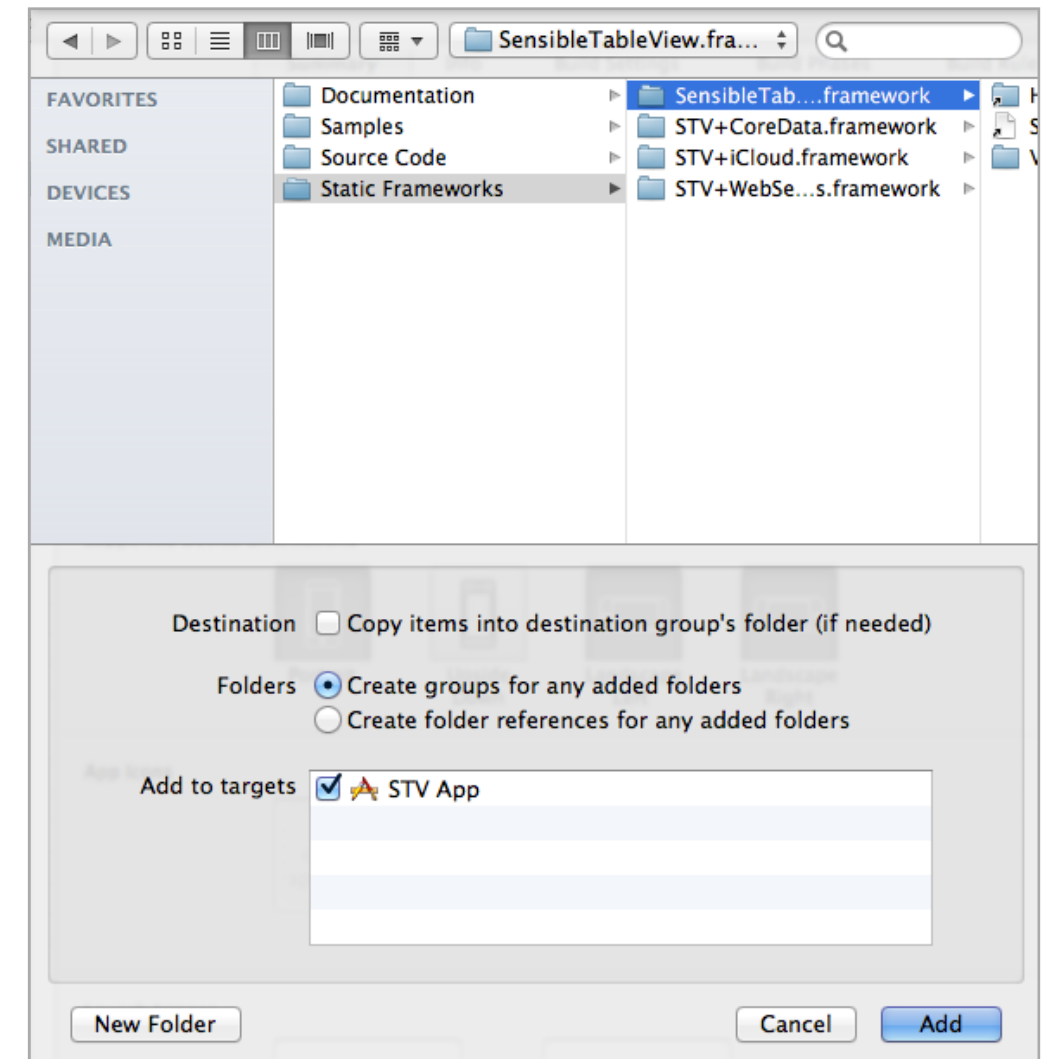
Also, STV fully supports working with non Automatic Reference Counting (ARC) projects, but there is generally no reason for you to not use ARC unless you're migrating an old project.

4. Choose a place to save your project, then click “Create”.
5. Right-click on the Frameworks group, then select Add files to “STV App”...

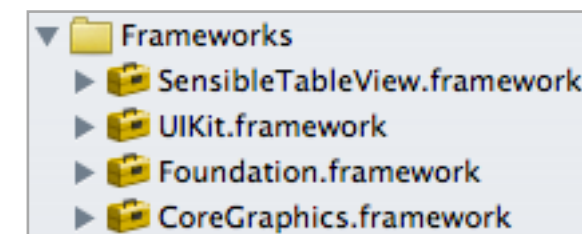


6. Now choose the ‘SensibleTableView.framework’ file from your STV package (should be under the ‘Static Frameworks’ folder), then click on the “Add” button. It is generally recommended to leave the “Copy items into destination group’s folder” option unchecked for

easier upgrades to future STV versions. Once you



click the Add button, you should now see the Sensible TableView framework under the frameworks group.



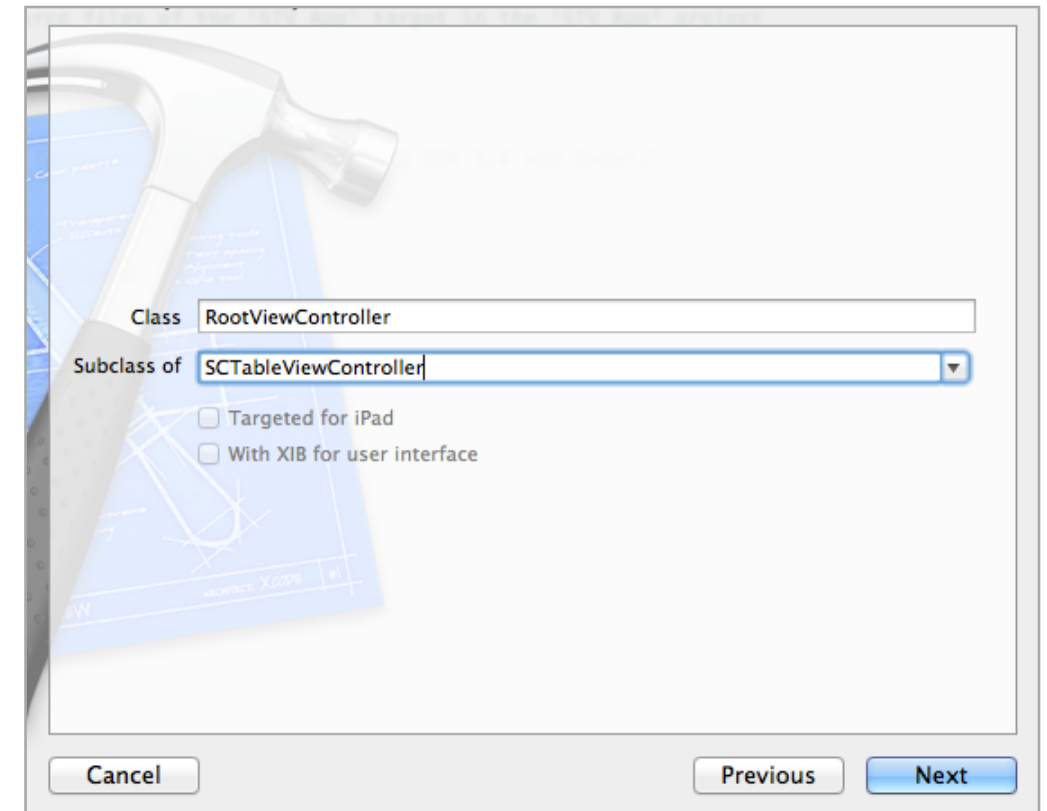
7. For your convenience, you should now import the <SensibleTableView/SensibleTableView.h> file to

your prefix headers file. This will save you the need to keep importing it in every file that needs to use STV. Xcode usually keeps the prefix headers file under the “Supporting Files” group. Navigate to this group, select the “STV App-prefix.pch” file, then add the following statement to the file:

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



8. Now let's add a root view controller to our project. Right-click on the “STV App” group, then select “New file...”. From the file templates, choose “Objective-C class” then click ‘Next’. Enter ‘RootViewController’ for the class name field, then enter ‘SCTableViewController’ in the Subclass of field. There is usually no need to select an XIB file when working with SCTableViewController subclasses.



It is perfectly fine to use a normal UITableViewController here instead of STV's own SCTableViewController. However, SCTableViewController makes your life much easier by automatically creating an STV model and connecting it to the view controller's table view. More on STV's view controllers in the next chapter.

9. Click “Next”, then click the “Create” button to save the file.
10. Repeat steps 8 & 9 to create an iPad detail view controller. Call it ‘iPadDetailViewController’.
11. Finally, navigate to AppDelegate.m and modify it to use the newly created view controllers and you should be all done! Your AppDelegate.m should look like the code below.

```

#import "AppDelegate.h"

#import "RootViewController.h"
#import "iPadDetailViewController.h"

@implementation AppDelegate

@synthesize window = _window;

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen]
                                                    bounds]];

    RootViewController *rootViewController = [[RootViewController alloc]
                                              initWithStyle:UITableViewStyleGrouped];
    UINavigationController *rootNavController =
        [[UINavigationController alloc]
         initWithRootViewController:rootViewController];

    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPhone)
    {
        self.window.rootViewController = rootNavController;
    }
    else
    {
        iPadDetailViewController *detailViewController =
            [[iPadDetailViewController alloc]
             initWithStyle:UITableViewStyleGrouped];
        UINavigationController *detailNavController =
            [[UINavigationController alloc]
             initWithRootViewController:detailViewController];

        // Connect the master model to the detail model
        rootViewController.tableViewModel.detailViewController =
            detailViewController;

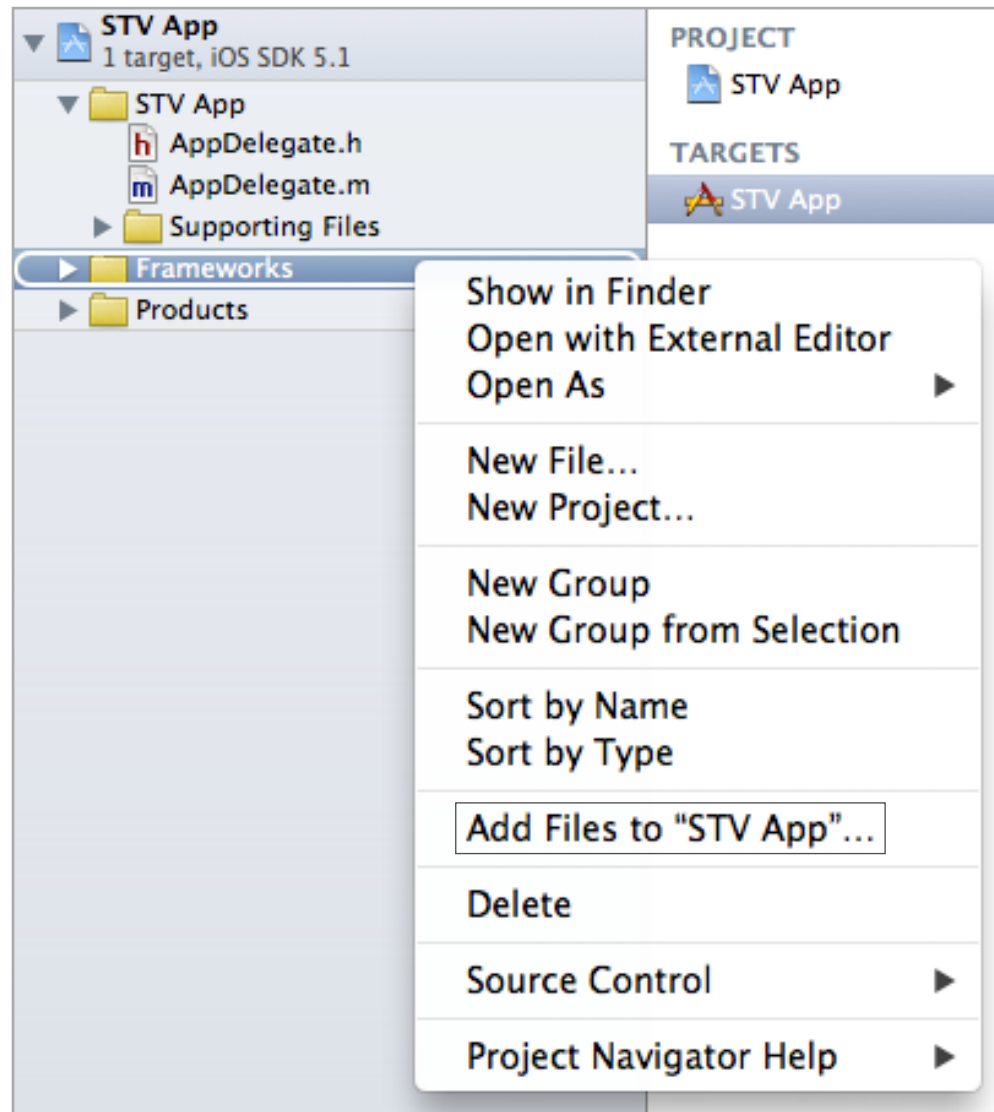
        UISplitViewController *splitViewController =
            [[UISplitViewController alloc] init];
        splitViewController.viewControllers =
            [NSArray arrayWithObjects:rootNavController,
            detailNavController, nil];
        self.window.rootViewController = splitViewController;
    }
    [self.window makeKeyAndVisible];
    return YES;
}

```

Congratulations! Your project is now set up and you're ready to start unleashing STV's magic!

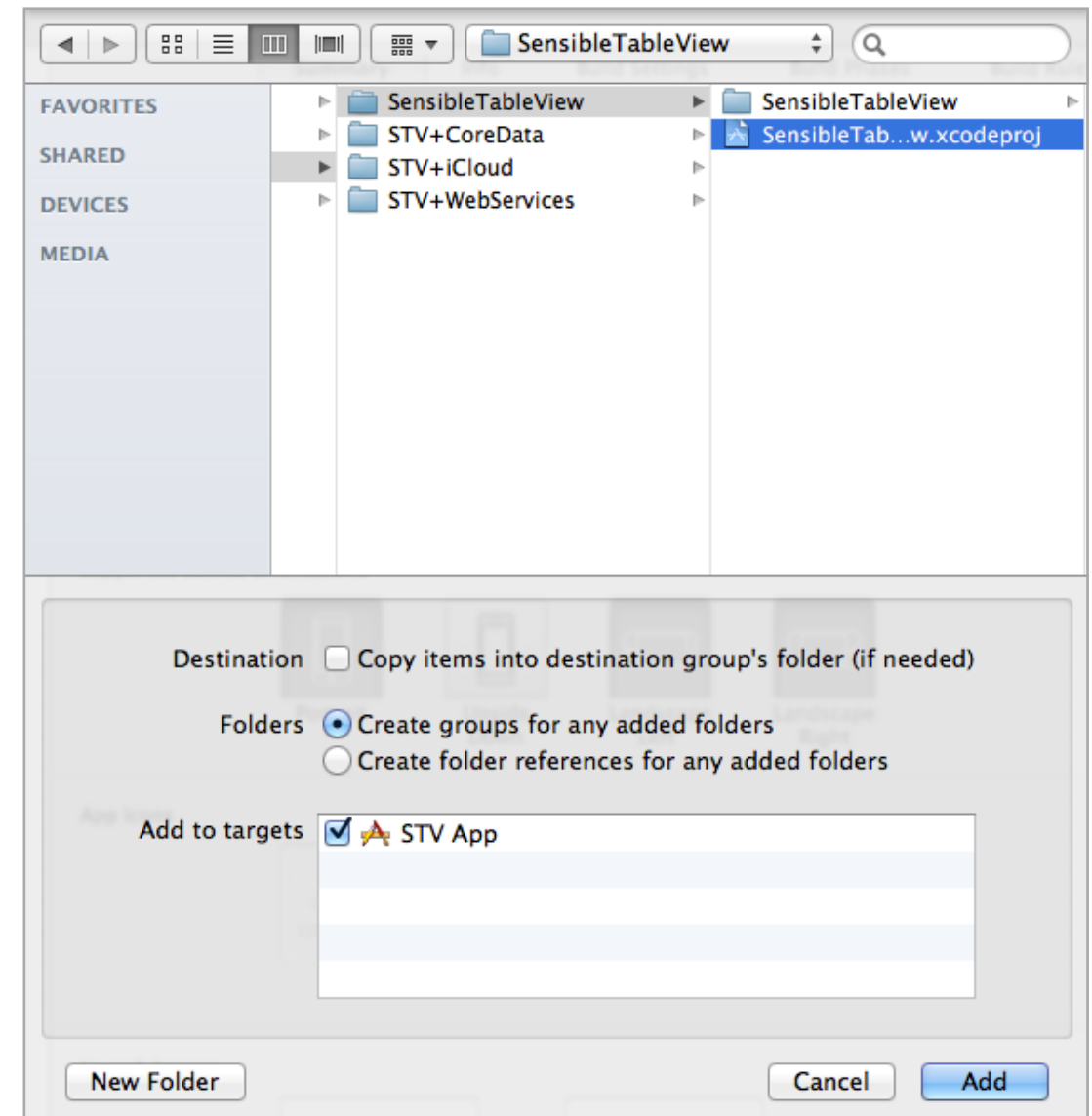
Using STV in source code format

1. Follow steps 1 to 4 from ‘Using STV as a static framework’.
2. Right-click on the STV App project, then select Add files to “STV App”...

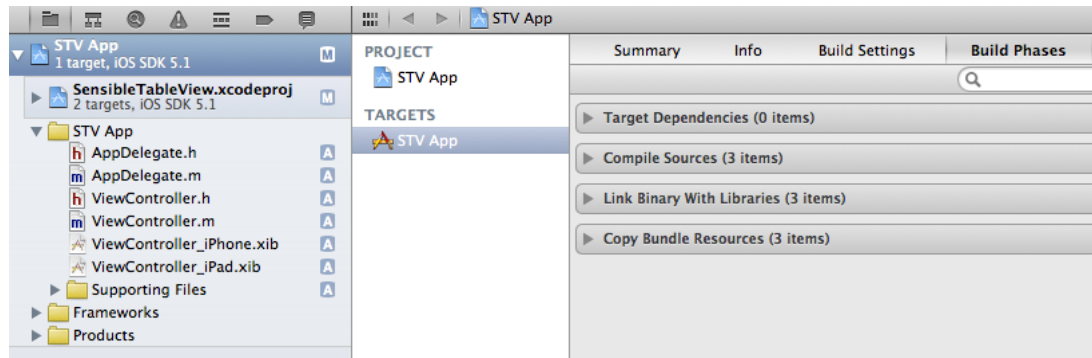


3. Now choose the ‘SensibleTableView.xcodeproj’ file from your STV package (should be under the ‘Source

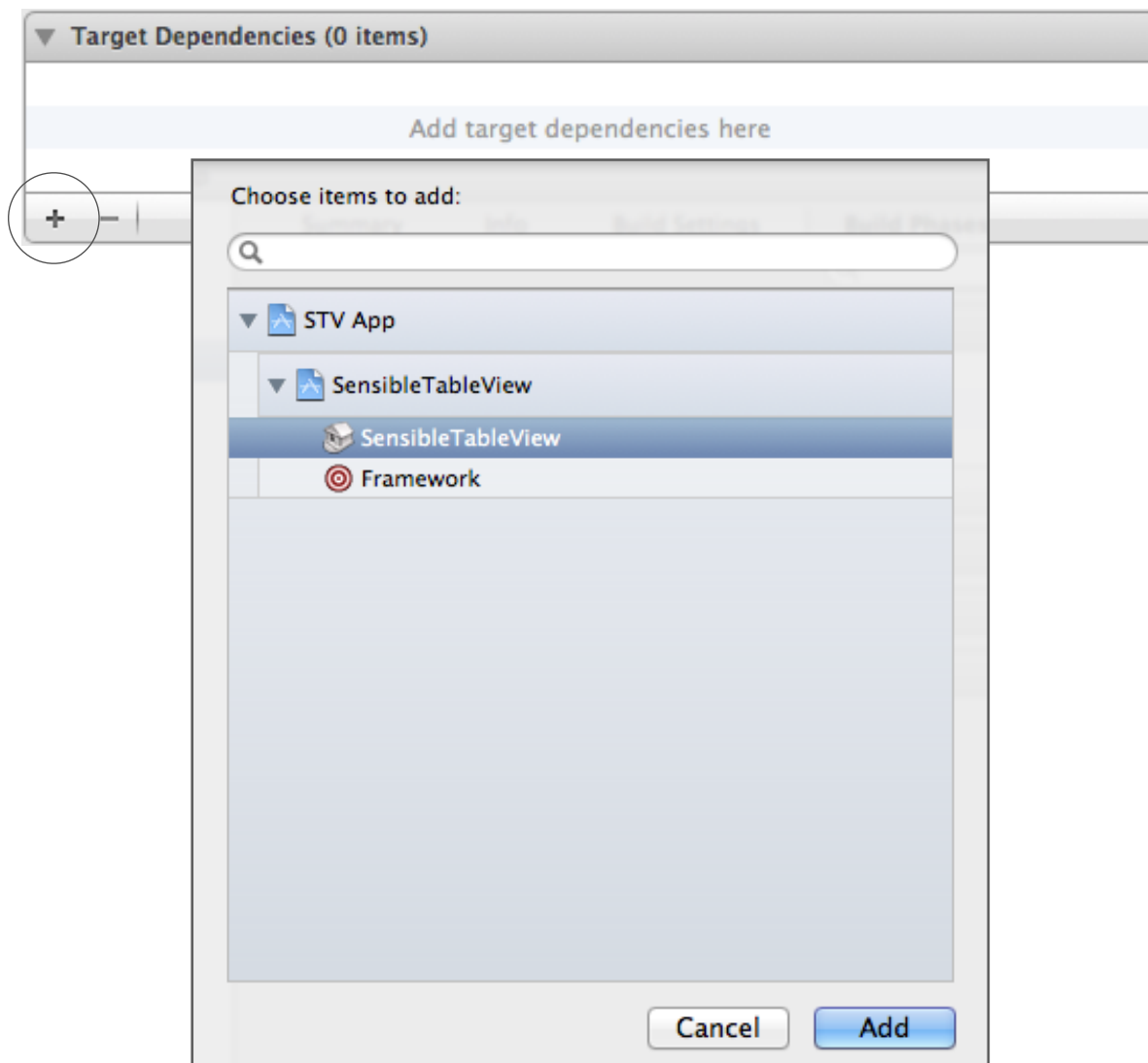
Code’ folder), then click on the “Add” button. If you wish for your project to have its own separate copy of the framework, select the “Copy items into destination group’s folder” option.



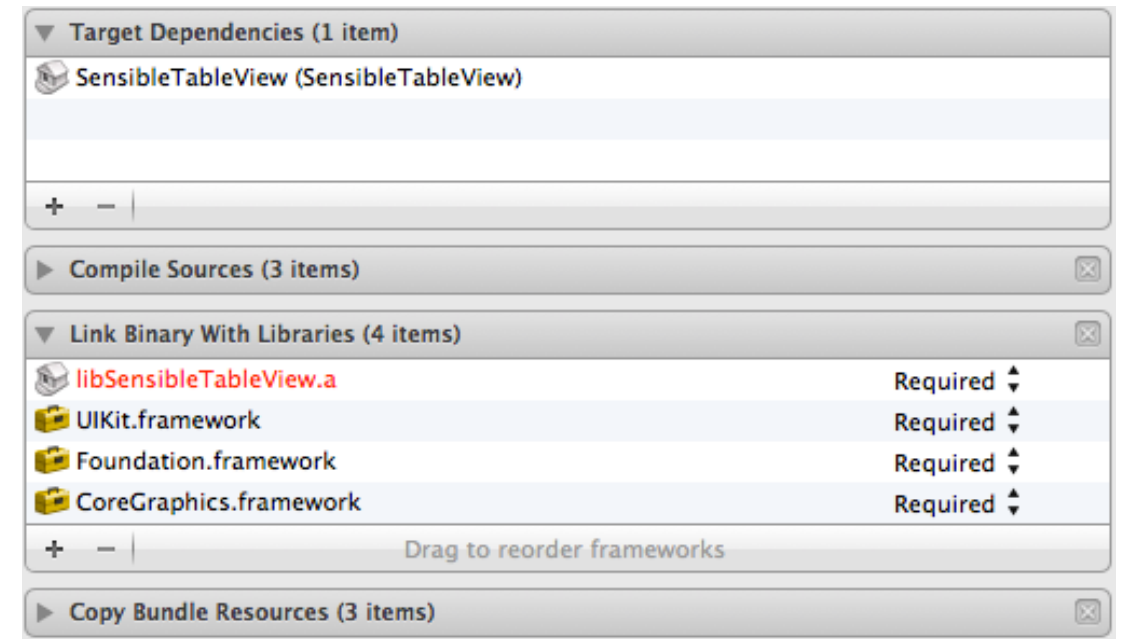
4. Select the “STV App” target, then select “Build Phases”.



5. Expand the “Target Dependencies” tab, then click the + button to add the ‘SensibleTableView’ target.



6. Similarly, expand the “Link Binary With Libraries” tab, then click the + button to add the ‘libSensibleTableView.a’ static library.



7. Follow steps 7 to 11 from ‘Using STV as a static framework’, then you should be all done.

Exploring the possibilities

Having got the set up out of the way, we will now start exploring what STV has to offer. Even though we'll be just scratching the surface here, this should give you a very good start.

Let's first start by verifying that you've got STV set up correctly as per the steps given in the *Setting up STV* section. If you don't have the "STV App" project already open, go ahead and open it in Xcode. Make sure that the selected target device is the iPhone simulator.

STV App > iPhone 5.1 Simulator

Now navigate to the RootViewController.m file, then add the following code:

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

    NSMutableArray *items = [NSMutableArray arrayWithObjects:@"One",
    @"Two", @"Three!", nil];
    SCArrayOfStringsSection *stringsSection = [SCArrayOfStringsSection
    sectionWithTitle:@"Strings Section" items:items];

    [self.tableViewModel addSection:stringsSection];
}
```

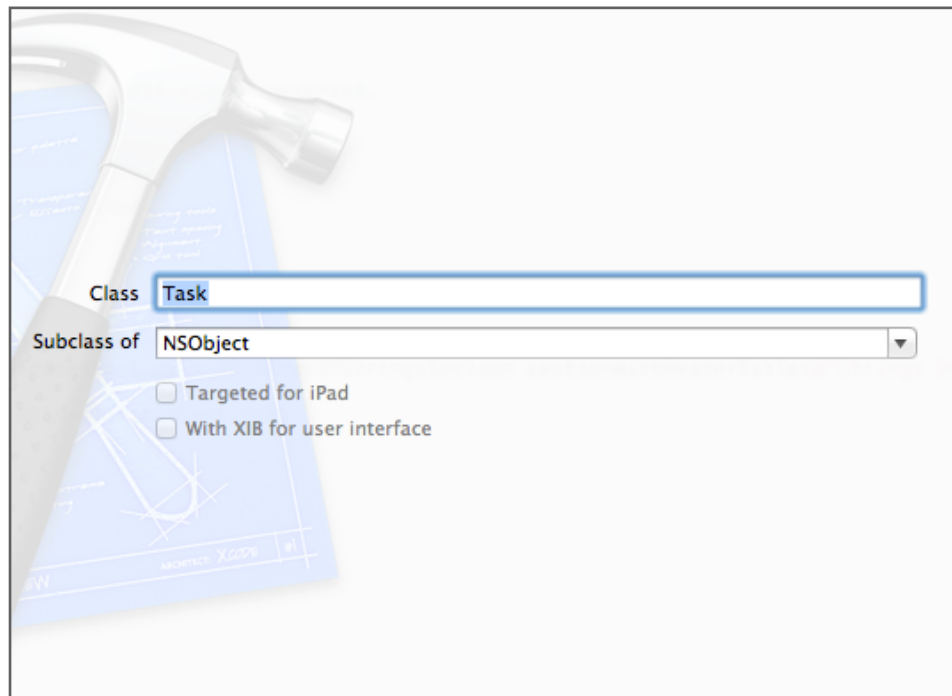
From Xcode's menu, choose Product->Run. If everything has been set up correctly, you should now see the following:



Exploring Object Binding

One of the most powerful features of STV is its ability to “understand” your objects, then create a user interface that resembles their property structure. Not only will STV create cells that resemble your object’s properties, but it will also automatically create detail views for any detail objects owned by the main object. Furthermore, STV will also monitor any input the user provides and will sync all that back to its respective properties.

To see all this in action, we’ll be creating a simple task management application. Let’s start by adding a Task object to the STV App project. From Xcode, choose File->New->Objective-C Class. Name the class ‘Task’, and make sure it descends from NSObject.



After saving the file, navigate to Task.h and add the following properties:

```
@interface Task : NSObject

@property (nonatomic, strong) NSString *name;
@property (nonatomic, strong) NSString *description;
@property (nonatomic, strong) NSString *category;
@property (nonatomic, strong) NSDate *dueDate;
@property (nonatomic, assign) BOOL completed;

@end
```

Then complete the implementation in Task.m

```
@implementation Task

@synthesize name, description, category, dueDate, completed;

- (id)init
{
    self = [super init];
    if(self)
    {
        name = nil;
        description = nil;
        category = nil;
        dueDate = nil;
        completed = FALSE;
    }
    return self;
}

@end
```

Now it’s time to tell STV about our new class. Using a class called `SCClassDefinition`, we will be able to fully describe the Task class to STV. We will be studying class definitions later on in great detail, but for the time being, just navigate to `iPhoneRootViewController.m` and add the following code:


```

#import "Task.h"

@implementation iPhoneRootViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    // Create the Task definition
    SCClassDefinition *taskDef = [SCClassDefinition
definitionWithClass:[Task class]
propertyNameString:@"name;description;category;dueDate;completed"]
;

    // Create an instance of the Task object
    Task *myTask = [[Task alloc] init];

    // Create the section(s) for the task object
    [self.tableViewModel generateSectionsForObject:myTask
withDefinition:taskDef];
}

@end

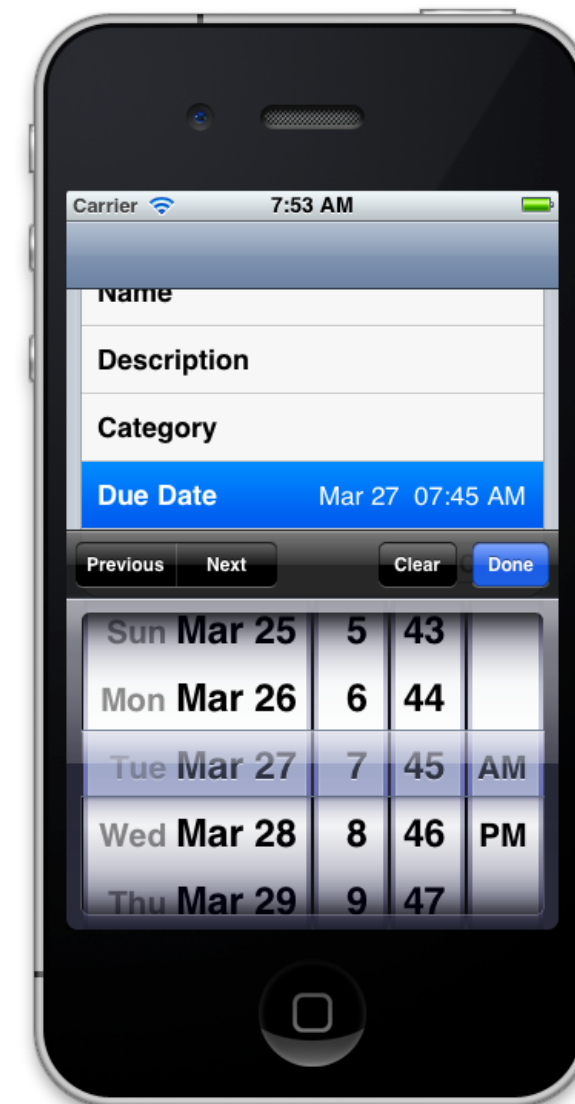
```

Now run the app. If you've correctly followed the earlier steps, the simulator should look like this:



Feel free to play around with the app. As you start interacting with the UI, you will quickly notice the following:

- All properties have been automatically renamed in a user friendly manner. We will see later how to fully customize this behavior.
- While STV displays a normal text keyboard for the name, description and category properties, it automatically detects that dueDate is a date property and displays a date picker instead.



- STV detects that ‘completed’ is a BOOL property and automatically generates a switch cell to match it.
- STV automatically (and optionally) provides a keyboard accessory view with a Previous and Next buttons, enabling the user to easily move between the different text fields.

Now even though STV did the best it could to detect what type of cells to generate for each property, sometimes it is necessary to further extend all this to satisfy our application’s requirements. For example, we want the description to become a resizable text view instead of a simple text field cell. Also, we want to select a category from a predefined list, instead of typing it each time.

Fortunately, STV makes it really easy to perform these customizations by using *Property Definitions*. As soon as we created the Task class definition, STV automatically created a property definition for each of the provided properties. All we need is simply retrieve the property definition and change its type and/or attributes. Navigate to iPhoneRootViewController.m and modify the code as follows:

```
#import "Task.h"

@implementation iPhoneRootViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    // Create the Task definition
    SCClassDefinition *taskDef = [SCClassDefinition
definitionWithClass:[Task class]
propertyNamesString:@"name;description;category;dueDate;completed"];
    SCPropertyDefinition *descPropertyDef = [taskDef
propertyDefinitionWithName:@"description"];
    descPropertyDef.type = SCPropertyTypeTextView;
    SCPropertyDefinition *categoryPropertyDef = [taskDef
propertyDefinitionWithName:@"category"];
    categoryPropertyDef.type = SCPropertyTypeSelection;
    NSArray *categoryItems = [NSArray arrayWithObjects:@"Home",
@"Work", @"Other", nil];
    categoryPropertyDef.attributes = [SCSelectionAttributes
attributesWithItems:categoryItems allowMultipleSelection:NO
allowNoSelection:NO];

    // Create an instance of the Task object
    Task *myTask = [[Task alloc] init];

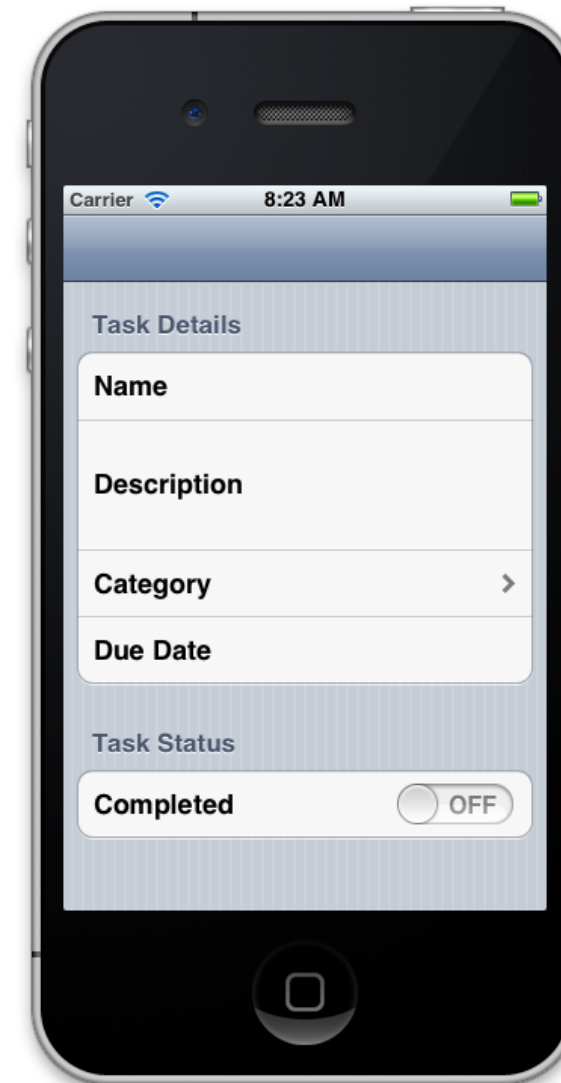
    // Create the section(s) for the task object
    [self.tableViewModel generateSectionsForObject:myTask
withDefinition:taskDef];
}

@end
```

Now run the app and try playing around with the new controls added. Note that the category property now automatically generates a selection detail view for you with items to choose from.



Now run the app and you should get the following setup.



Finally, let's organize the UI into separate categories. STV easily allows us to do so right from within the property names string in the class definition. Just change the names string to look like this:

```
SCClassDefinition *taskDef = [SCClassDefinition
definitionWithClass:[Task class] propertyNameString:@"Task
Details:(name,description,category,dueDate);Task
Status:(completed)"];
```

As useful as the current app is, it's rarely the case where a task management app has only one task! Amazingly enough, adding multiple task functionality to our application actually requires very little work on our part, full with the ability to add/edit/delete/rearrange tasks! Using a class called `SCArrayOfObjectsSection`, we'll have STV take care of everything for us one more time. Modify `iPhoneRootViewController.m` as follows:



```
#import "Task.h"

@implementation iPhoneRootViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.title = @"Tasks";
    self.navigationController = UINavigationControllerAddLeftEditRight;

    // Create the Task definition
    SCClassDefinition *taskDef = [SCClassDefinition
definitionWithClass:[Task class] propertyNameString:@"Task
Details:(name,description,category,dueDate);Task
Status:(completed)"];
    SCPropertyDefinition *namePropertyDef = [taskDef
propertyDefinitionWithName:@"name"];
    namePropertyDef.required = TRUE;
    SCPropertyDefinition *descPropertyDef = [taskDef
propertyDefinitionWithName:@"description"];
    descPropertyDef.type = SCPropertyTypeTextView;
    SCPropertyDefinition *categoryPropertyDef = [taskDef
propertyDefinitionWithName:@"category"];
    categoryPropertyDef.type = SCPropertyTypeSelection;
    NSArray *categoryItems = [NSArray arrayWithObjects:@"Home",
@"Work", @"Other", nil];
    categoryPropertyDef.attributes = [SCSelectionAttributes
attributesWithItems:categoryItems allowMultipleSelection:NO
allowNoSelection:NO];

    // Create the Tasks array
    NSMutableArray *tasksArray = [NSMutableArray array];

    // Create the section for the tasks array
    SCArrayOfObjectsSection *tasksSection = [SCArrayOfObjectsSec-
tion sectionWithTitle:nil items:tasksArray
itemsDefinition:taskDef];
    tasksSection.addButtonItem = self.addButton;
    tasksSection.placeholderCell = [SCTableViewCell
cellWithText:@"No tasks yet!"
textAlignment: NSTextAlignmentCenter];

    [self.tableViewModel addSection:tasksSection];
}
}
```

Finally, we'd like STV to take care of a relationship between our Task class and another detail class, like a list of task completion steps for instance. Let's first create a new class called TaskStep and implement it as follows:

TaskStep.h

```
@interface TaskStep : NSObject

@property (nonatomic, strong) NSString *name;
@property (nonatomic, strong) NSString *description;

@end
```

TaskStep.m

```
@implementation TaskStep

@synthesize name, description;

- (id)init
{
    self = [super init];
    if(self)
    {
        name = nil;
        description = nil;
    }
    return self;
}

@end
```

Now modify the Task class as follows:

```
@interface Task : NSObject

@property (nonatomic, strong) NSString *name;
@property (nonatomic, strong) NSString *description;
@property (nonatomic, strong) NSString *category;
@property (nonatomic, strong) NSDate *dueDate;
@property (nonatomic, assign) BOOL completed;
@property (nonatomic, readonly) NSMutableArray *taskSteps;

@end

@implementation Task

@synthesize name, description, category, dueDate, completed, taskSteps;

- (id)init
{
    self = [super init];
    if(self)
    {
        name = nil;
        description = nil;
        category = nil;
        dueDate = nil;
        completed = FALSE;
        taskSteps = [NSMutableArray array];
    }
    return self;
}

@end
```

Important: the taskSteps array must be declared as an *NSMutableArray* in order for STV to be able to add and remove task steps.

Moving back to *iPhoneRootViewController.m*, we will now create a new definition for the TaskStep class, then tell our taskDef about it using property *attributes*:

```
#import "Task.h"
#import "TaskStep.h"

@implementation iPhoneRootViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.title = @"Tasks";
    self.navigationBarType = SCNavigationBarTypeAddLeftEditRight;

    // Create the TaskStep definition
    SCClassDefinition *taskStepDef = [SCClassDefinition
definitionWithClass:[TaskStep class]
autoGeneratePropertyDefinitions:YES];
    [taskStepDef propertyDefinitionWithName:@"description"].type =
SCPropertyTypeTextView;

    // Create the Task definition
    SCClassDefinition *taskDef = [SCClassDefinition
definitionWithClass:[Task class] propertyNameString:@"Task
Details:(name,description,category,dueDate, taskSteps);Task
Status:(completed)"];
    SCPropertyDefinition *namePropertyDef = [taskDef
propertyDefinitionWithName:@"name"];
    namePropertyDef.required = TRUE;
    SCPropertyDefinition *descPropertyDef = [taskDef
propertyDefinitionWithName:@"description"];
    descPropertyDef.type = SCPropertyTypeTextView;
    SCPropertyDefinition *categoryPropertyDef = [taskDef
propertyDefinitionWithName:@"category"];
    categoryPropertyDef.type = SCPropertyTypeSelection;
    NSArray *categoryItems = [NSArray arrayWithObjects:@"Home",
@"Work", @"Other", nil];
    categoryPropertyDef.attributes = [SCSelectionAttributes
attributesWithItems:categoryItems allowMultipleSelection:NO
allowNoSelection:NO];
    SCPropertyDefinition *taskStepsRelDef = [taskDef
propertyDefinitionWithName:@"taskSteps"];
    taskStepsRelDef.title = @"Steps";
    taskStepsRelDef.type = SCPropertyTypeArrayOfObjects;
```

```
taskStepsRelDef.attributes = [SCArrayOfObjectsAttributes
attributesWithObjectDefinition:taskStepDef allowAddingItems:YES
allowDeletingItems:YES allowMovingItems:YES];

// Create the Tasks array
NSMutableArray *tasksArray = [NSMutableArray array];

// Create the section for the tasks array
SCArrayOfObjectsSection *tasksSection = [SCArrayOfObjectsSec-
tion sectionWithTitle:nil items:tasksArray
itemsDefinition:taskDef];
tasksSection.addButtonItem = self.addButton;
tasksSection.placeholderCell = [SCTableViewCell
cellWithText:@"No tasks yet!" textAlignment: NSTextAlignmentCen-
ter];

[self.tableViewModel addSection:tasksSection];
}
```

Running the app, you should now find that STV has automati-
cally handled all the UI related with the relationship:



So far so good, but how would we implement all this on the iPad? Since there is more screen real estate there, it makes sense for us to flatten out the UI a bit by displaying the task details using a detail view controller, instead of pushing a new view controller as we do on the iPhone. Fortunately again, STV comes to the rescue by automatically handling the whole process for us. All we need is place a *UISplitViewController* (or any other custom container) as the root view controller, then set its viewControllers property to a master and a detail view controllers. We'll then use the tableViewModel in the master view controller and assign the detail view controller to its detailViewController property. If you recall, this is exactly what we did in the “*Setting up STV*” section earlier:

AppDelegate.m

```
#import "AppDelegate.h"

#import "RootViewController.h"
#import "iPadDetailViewController.h"

@implementation AppDelegate

@synthesize window = _window;

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen]
                                                    bounds]];

    RootViewController *rootViewController = [[RootViewController alloc]
                                              initWithStyle:UITableViewStyleGrouped];
    UINavigationController *rootNavController =
        [[UINavigationController alloc]
         initWithRootViewController:rootViewController];

    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPhone)
    {
        self.window.rootViewController = rootNavController;
    }
    else
    {
        iPadDetailViewController *detailViewController =
            [[iPadDetailViewController alloc]
             initWithStyle:UITableViewStyleGrouped];
        UINavigationController *detailNavController =
            [[UINavigationController alloc]
             initWithRootViewController:detailViewController];

        // Connect the master model to the detail model
        rootViewController.tableViewModel.detailViewController =
            detailViewController;

        UISplitViewController *splitViewController =
            [[UISplitViewController alloc] init];
        splitViewController.viewControllers =
            [NSArray arrayWithObjects:rootNavController,
            detailNavController, nil];
        self.window.rootViewController = splitViewController;
    }

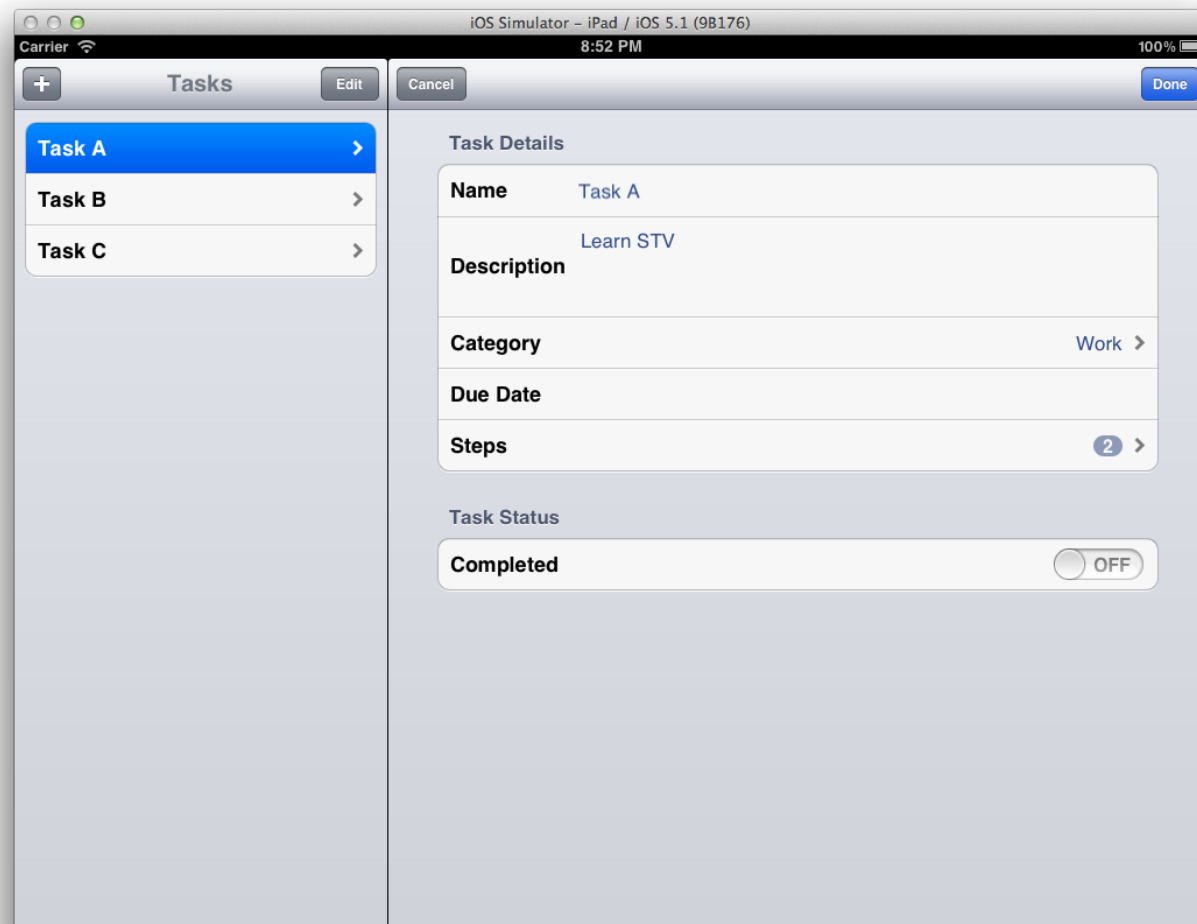
    [self.window makeKeyAndVisible];
    return YES;
}
```

Important: Since we're now using the same RootViewController class for both the iPhone and iPad, it is important that we provide the correct interface orientation in the 'shouldAutorotateToInterfaceOrientation' method. Fortunately, SCViewController/SCTableViewController will automatically take care of this as long as you don't have this method implemented. To make sure SCTableViewController is automatically handling this, make sure to search RootViewController.m for the 'shouldAutorotateToInterfaceOrientation' method and remove it if it has been placed there by Xcode's template.

If you've set up everything correctly, all you need is select the iPad simulator as the iOS device and run the app.

STV App > iPad 5.1 Simulator

Viola!



Exploring Core Data Binding

It is no secret that STV's *Core Data Binding* is one of the most sought-after features of the whole STV framework. While using *Object Binding* gets you to have STV automatically handle all the UI generation, you're still very much responsible for persisting all the objects yourself. With Core Data however, the whole object graph is automatically persisted on your behalf (typically to an SQLite database). Combine this functionality with STV and you can have a full fledged application up and running in literally a matter of minutes!

In this section, we'll be recreating the same sample we created in the Object Binding section, but using Core Data. We'll start by creating a project based on the template created in *Section 3: Setting up STV*. Since we'll be using Core Data, you should be checking the 'Use Core Data' option in step number 3. Call the project 'STV Core Data' and save it to the location of your choice.

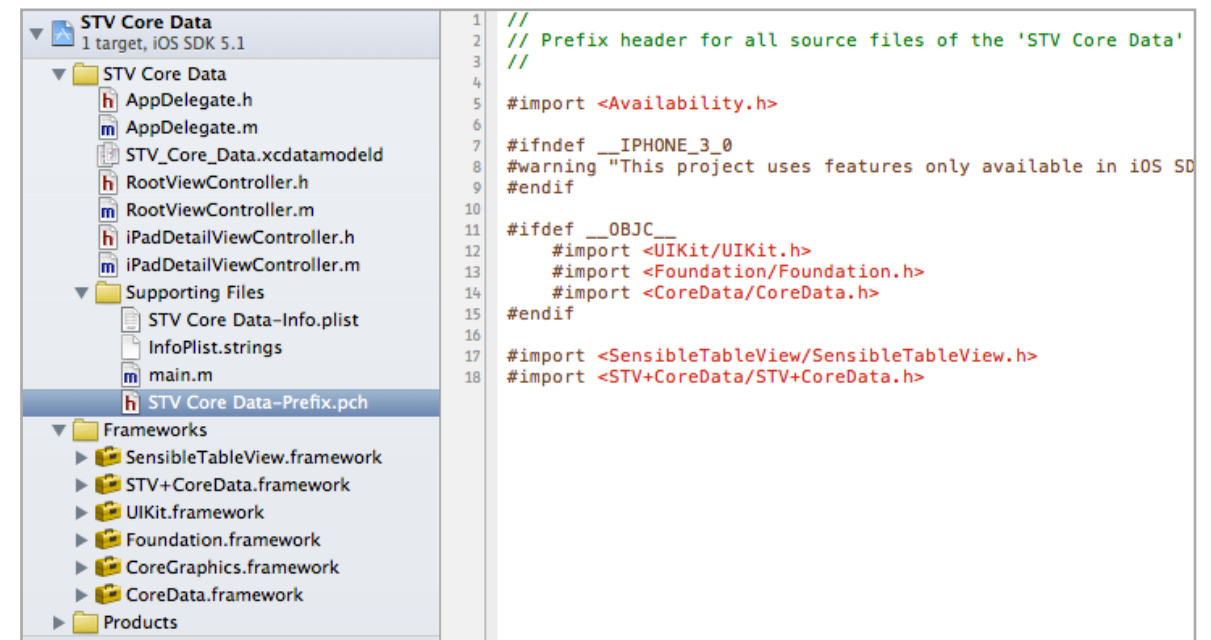
Next, we'll need to add the STV+CoreData framework extension to our project.

In STV 3.0, the main SensibleTableView framework only has the core STV functionality, and all other features have been implemented as 'framework extensions'. This keeps your project's size minimal, with only the features you need added. In addition, this lays out the ground for STV to be extremely extensible, either by you or by other third parties providing their own framework extensions for STV.

To add STV+CoreData, right-click on the Frameworks group, then select 'Add files to STV CoreData...'. From your package files, select the 'STV+CoreData.framework' file and click Add. This adds STV+CoreData as a static framework. If you need to add it in source code format, please follow the same steps laid out earlier in Section 3 for setting up SensibleTableView in source code format. Finally, add the following statement to the 'STV Core Data-Prefix.pch' prefix header file:

```
#import <STV+CoreData/STV+CoreData.h>
```

Your project should now look like this:



Next, let's add some entities to our Core Data model.

1. Select the 'STV_Core_Data.xcdatamodeld' file to have the model edit appear. Make sure the Data Model inspector is visible by selecting View->Utilities->Show Data Model Inspector from Xcode's menu.
2. Click on the Add Entity button to add a new entity. Name the new entity 'TaskEntity'.
3. Click on the '+' button in the Attributes section to add a new attribute to TaskEntity. Name the new attribute 'name', and set its type to String. To have 'name' become a required attribute, uncheck the *Optional* checkbox in the Data Model inspector.

The screenshot shows the 'Attribute' inspector for the 'name' attribute. The 'Name' field is set to 'name'. Under 'Properties', the 'Optional' checkbox is unchecked and highlighted with a red box. The 'Attribute Type' is set to 'String'. The 'Validation' section shows 'No Value' for both 'Min Length' and 'Max Length'. The 'Default Value' is 'Default Value' and the 'Reg. Ex.' is 'Regular Expression'.

4. Add the following attributes to TaskEntity (no need to uncheck *Optional* for any other attribute):

ENTITIES	
TaskEntity	
FETCH REQUESTS	
CONFIGURATIONS	
Default	
Attributes	
Attribute	Type
completed	Boolean
category	String
desc	String
dueDate	Date
name	String

5. Similarly, add a new Entity called 'TaskStepEntity' with the following attributes:

ENTITIES	
TaskEntity	
TaskStepEntity	
FETCH REQUESTS	
CONFIGURATIONS	
Attributes	
Attribute	Type
details	String
name	String

6. We'll now establish a 1-to-many relationship between TaskEntity and TaskStepEntity. With the TaskStepEntity still selected, click the '+' button in the *Relationships* section to create a new relationship. Name the relationship 'task', and select TaskEntity as its *Destination* entity. Next, select the TaskStep entity and click the '+' button to create a new relationship. Name the relationship 'taskSteps', select TaskStepEntity as its destination, and select task as its inverse relationship. Finally, while taskSteps is still selected, check the 'To-Many Relationship' checkbox.

▼ Relationship

Name

Destination

Inverse

Properties ☐ Transient ☒ Optional

Arranged ☐ Ordered

Plural ☒ To-Many Relationship


Count ☐ Minimum

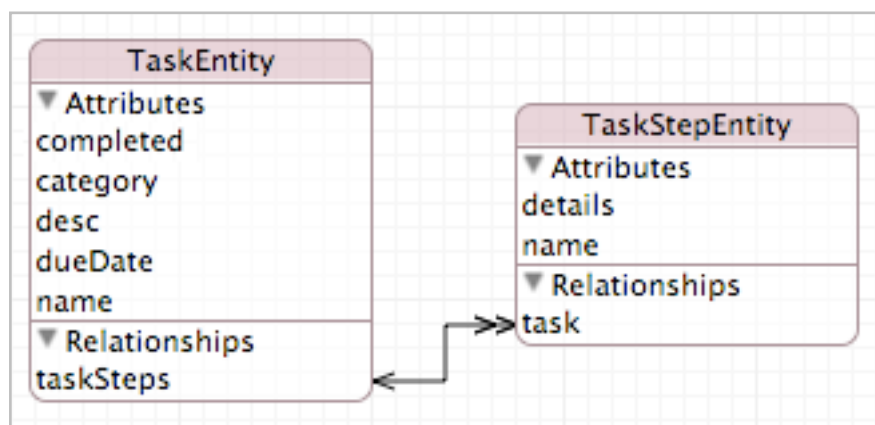
☐ Maximum

Delete Rule

Advanced ☐ Index in Spotlight

☐ Store in External Record File

This should be all we need. To make sure you did everything correctly, click on the graph editor style button  and check that the entities have been correctly set up. Your graph should look like the following:



Now that we're done with the Core Data model, the rest should be even simpler than what we did in *Object Binding*!

Similarly to our Object Binding sample, we now need to describe our entities to STV. If you recall, we earlier used a class called `SCClassDefinition` to describe object classes. To describe Core Data entities, we'll be using another class called *SCEntityDefinition*.

Both `SCClassDefinition` and `SCEntityDefinition` (in addition to many other similar classes) are subclasses of the *SCDataDefinition* abstract base class. Much more details on that are provided in the next chapter and throughout the rest of the book.

Let's select `RootViewController.m` and insert the following code there:

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.title = @"Tasks";
    self.navigationController = UINavigationController
        sharedApplication].delegate managedObjectContext];

    // Create the TaskStep definition
    SCEntityDefinition *taskStepDef =
        [SCEntityDefinition definitionWithEntityName:@"TaskStepEntity"
        managedObjectContext:context
        propertyNamesString:@"name;details"];
    [taskStepDef propertyDefinitionWithName:@"details"].type =
        SCPropertyTypeTextView;

    // Create the Task definition
    SCEntityDefinition *taskDef =
        [SCEntityDefinition definitionWithEntityName:@"TaskEntity"
        managedObjectContext:context
        propertyNamesString:@"Task Details:(name,desc,category,dueDate,
taskSteps);Task Status:(completed)"];
    SCPropertyDefinition *namePropertyDef = [taskDef
propertyDefinitionWithName:@"name"];
    namePropertyDef.required = TRUE;
    SCPropertyDefinition *descPropertyDef = [taskDef
propertyDefinitionWithName:@"desc"];
    descPropertyDef.title = @"description";
    descPropertyDef.type = SCPropertyTypeTextView;
    SCPropertyDefinition *categoryPropertyDef = [taskDef
propertyDefinitionWithName:@"category"];
    categoryPropertyDef.type = SCPropertyTypeSelection;
    NSArray *categoryItems = [NSArray arrayWithObjects:@"Home", @"Work",
@"Other", nil];
    categoryPropertyDef.attributes = [SCSelectionAttributes
attributesWithItems:categoryItems allowMultipleSelection:NO
allowNoSelection:NO];
    SCPropertyDefinition *taskStepsRelDef = [taskDef
propertyDefinitionWithName:@"taskSteps"];
    taskStepsRelDef.title = @"Steps";
    taskStepsRelDef.type = SCPropertyTypeArrayOfObjects;
    taskStepsRelDef.attributes = [SCArrayOfObjectsAttributes
attributesWithObjectDefinition:taskStepDef allowAddingItems:YES
allowDeletingItems:YES allowMovingItems:YES];

```

```

// Create the tasks section
SCArrayOfObjectsSection *tasksSection =
    [SCArrayOfObjectsSection sectionWithHeaderTitle:nil
    entityDefinition:taskDef];
tasksSection.addButtonItem = self.addButton;
tasksSection.placeholderCell =
    [SCTableViewCell cellWithText:@"No tasks yet!"
    textAlignment: NSTextAlignmentCenter];

[self.tableViewModel addSection:tasksSection];
}

```

That it, all we need is a single page of code! Note that the code is almost identical to the one we used in Object Binding, with the exception of using `SCEntityDefinition` instead of `SCClassDefinition`. Now compile and run the project, and you should get an identical app to the one we had in Object Binding, only this time when you exit the app and return back the data is actually persisted!

Note: When you're working on the simulator, you should always click the Home button to properly exit an application. Quitting the simulator without clicking the Home button doesn't give the app the chance to persist any data.

Exploring Dictionary Binding

Using Dictionary Binding, STV can read and write values directly from an NSMutableDictionary. This usually proves to be very useful in some applications where having to create an object class in order to use STV is simply an overkill.

As always, we'll start by creating a new project based on the template created in *Section 3: Setting up STV*. Now select RootViewController.m and insert the following code:

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

    self.title = @"Dictionary Editor";

    // Define the dictionary using SCDictionaryDefinition
    SCDictionaryDefinition *dictionaryDef =
    [SCDictionaryDefinition
     definitionWithDictionaryKeyNamesString:@"key1;key2;key3"];
    [dictionaryDef propertyDefinitionWithName:@"key2"].type =
    SCPropertyTypeSwitch;
    [dictionaryDef propertyDefinitionWithName:@"key3"].type =
    SCPropertyTypeSlider;

    // Create some sample data
    NSMutableDictionary *dictionary =
    [NSMutableDictionary dictionary];
    [dictionary setValue:@"Text" forKey:@"key1"];
    [dictionary setValue:[NSNumber numberWithInt:YES]
    forKey:@"key2"];
    [dictionary setValue:[NSNumber numberWithFloat:0.7f]
    forKey:@"key3"];

    // Generate sections for the dictionary
    [self.tableViewModel generateSectionsForObject:dictionary
    withDefinition:dictionaryDef];
}
```

Now run the app and you should get the following:



Again, everything goes along the same concepts discussed in the previous chapters. If you're already feeling that you got the hang of STV, then this is a great time to start trying out some projects on your own!

Exploring Web Service Binding

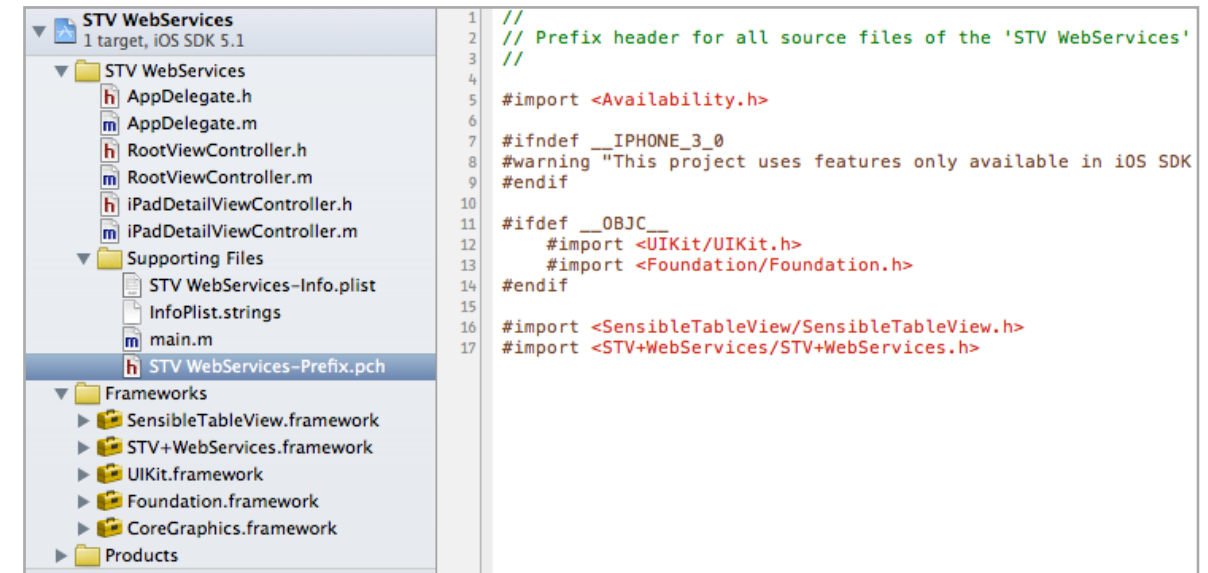
Web Service Binding was one of the most anticipated new features of STV 3.0. The vision was to give STV the ability to magically bind to web services in very much the same way it used to bind to objects and Core Data managed objects. In the following sample, we'll demonstrate how STV is able to consume a Twitter search web service to retrieve all the tweets mentioning the #iosdev hashtag. Please note that the sample natively talks to the web service, and does not use any of the new iOS 5.0 Twitter APIs to fetch the data.

As always, we'll start by creating a new project based on the template created in *Section 3: Setting up STV*, naming our new project: "STV WebServices". Similar to CoreData Binding, STV's Web Service Binding functionality comes in a separate framework extension called STV+WebServices.

To add STV+WebServices, right-click on the Frameworks group, then select 'Add files to STV WebServices...'. From your package files, select the 'STV+WebServices.framework' file and click Add. This adds STV+CoreWebServices as a static framework. If you need to add it in source code format, please follow the same steps laid out earlier in Section 3 for setting up SensibleTableView in source code format. Finally, add the following statement to the 'STV WebServices-Prefix.pch' prefix header file:

```
#import <STV+WebServices/STV+WebServices.h>
```

At the end of your initial set up, the project should look like this:



If you had gone through the previous samples or had previously been using STV, odds are you're pretty much expecting what to come next. Along the same lines of all we did before, we'll start out by defining Twitter's 'search' web service to STV. And yes, you guessed it correctly, the definition class is called *SCWebServiceDefinition*! Insert the following code into RootViewController.m:

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.title = @"#iosdev Search";

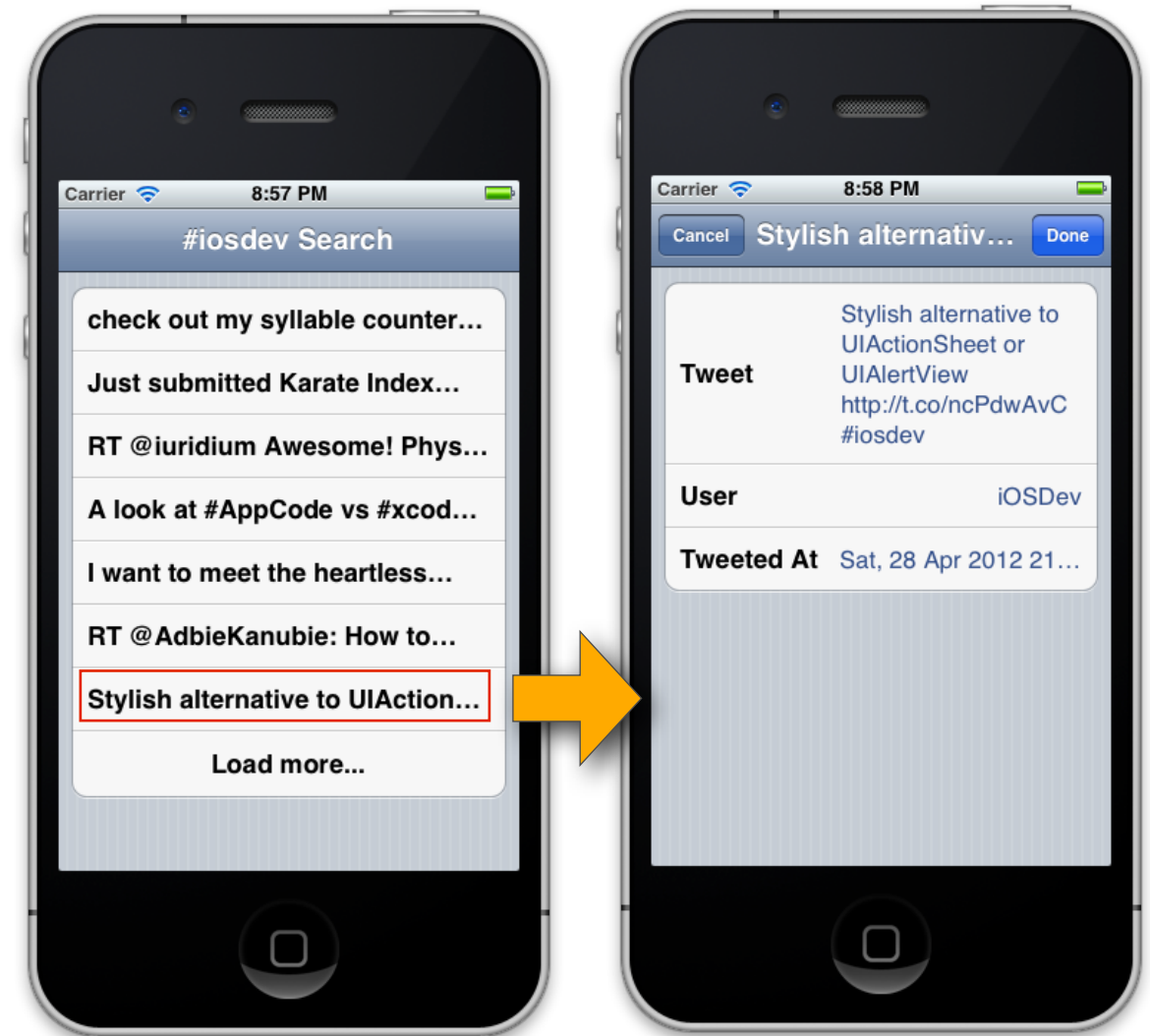
    // Create the web service definition for a 'tweet'
    SCWebServiceDefinition *tweetDef = [SCWebServiceDefinition
        definitionWithBaseURL:@"http://search.twitter.com/"
        fetchObjectsAPI:@"search.json" resultsKeyName:@"results"
        resultsDictionaryKeyNamesString:@"text;from_user_name;created_at"];
    [tweetDef.fetchObjectsParameters setValue:@"#iosdev" forKey:@"q"];
    [tweetDef.fetchObjectsParameters setValue:@"recent"
        forKey:@"result_type"];
    tweetDef.batchSizeParameterName = @"rpp";
    tweetDef.nextBatchURLKeyName = @"next_page";
    SCPropertyDefinition *textPropertyDef = [tweetDef
        propertyDefinitionWithName:@"text"];
    textPropertyDef.title = @"Tweet";
    textPropertyDef.type = SCPropertyTypeTextView;
    SCPropertyDefinition *userPropertyDef = [tweetDef
        propertyDefinitionWithName:@"from_user_name"];
    userPropertyDef.title = @"User";
    userPropertyDef.type = SCPropertyTypeLabel;
    SCPropertyDefinition *tweetedAtPropertyDef = [tweetDef
        propertyDefinitionWithName:@"created_at"];
    tweetedAtPropertyDef.title = @"Tweeted At";
    tweetedAtPropertyDef.type = SCPropertyTypeLabel;

    // Enable pull-to-refresh
    self.tableViewModel.enablePullToRefresh = TRUE;

    // Add tweets section
    SCArrayOfObjectsSection *tweetsSection = [SCArrayOfObjectsSection
        sectionWithHeaderTitle:nil webServiceDefinition:tweetDef batchSize:7];
    tweetsSection.itemsAccessoryType = UITableViewCellStyleNone;
    [self.tableViewModel addSection:tweetsSection];
}

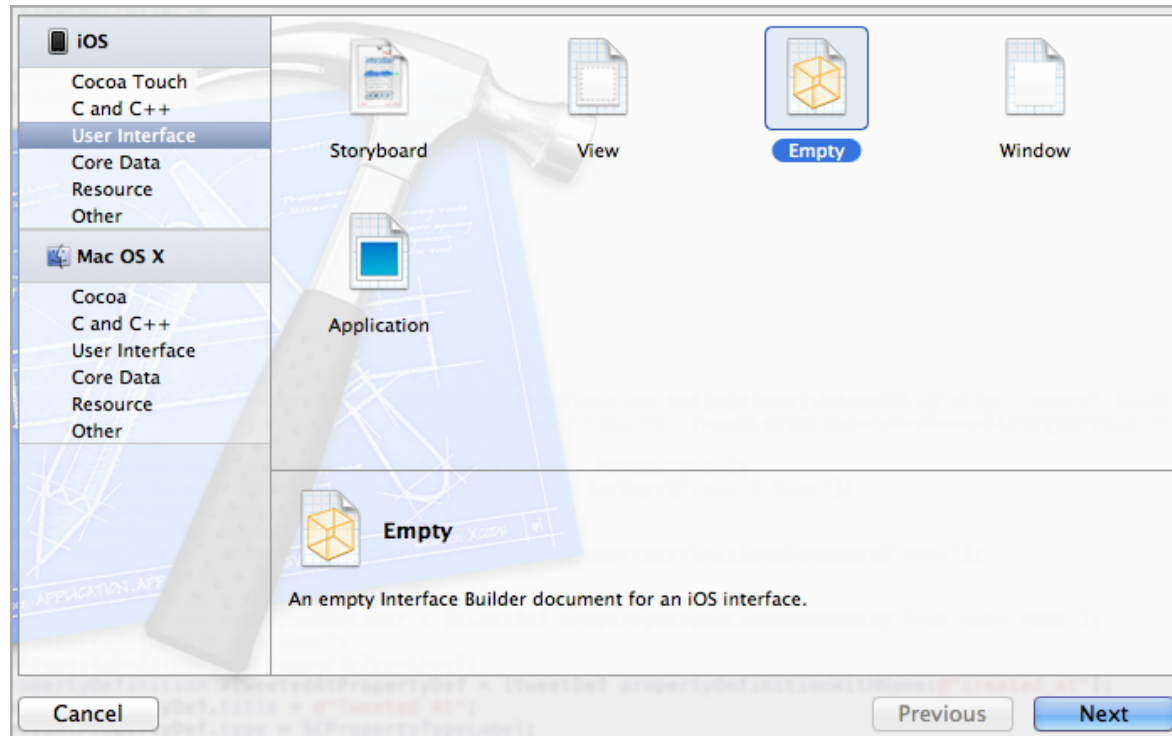
```

Running the app should give us the following:

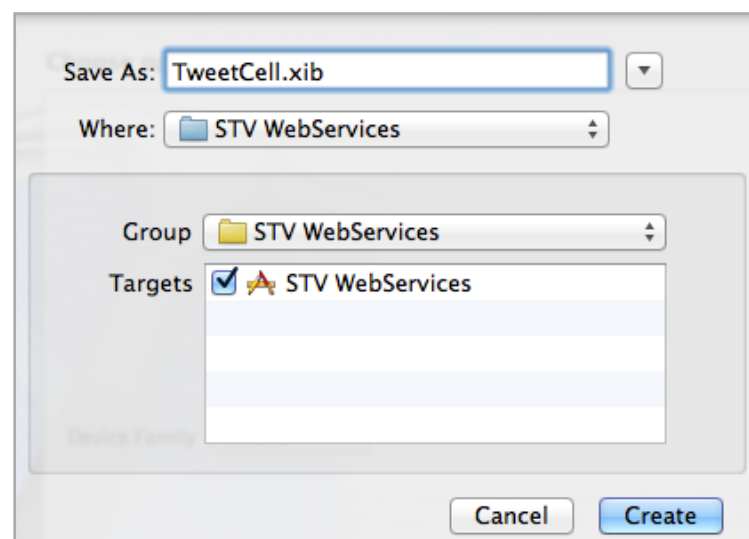


While not being too bad for a half page of code, STV can actually do a much better job than that. For starters, we'd like to see the whole tweet in the first screen, as it's really inconvenient to have to tap on each tweet to see the full text. Also, we'd like to have the user's avatar image displayed, just like professional Twitter clients do. Fortunately, both of these features (and a lot more!) can be achieved using STV's custom cells feature.

In this sample, we'll be creating the custom cell using Interface Builder. From Xcode's menu, select File->New->File... Now select the Empty XIB document template:

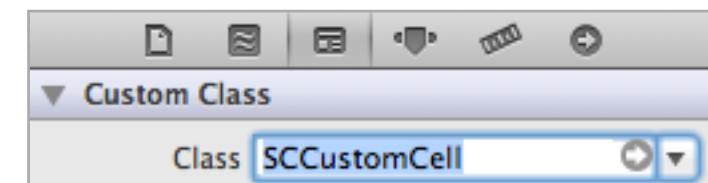


Press Next, then press Next again selecting iPhone as the device family. Finally, name the XIB file as TweetCell.xib and click Create.

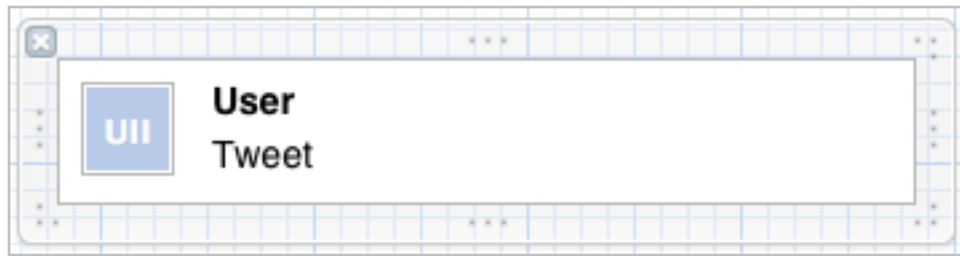


Xcode should now have the new TweetCell.xib file selected and an empty Interface Builder designer should be loaded. Now add the cell by following the next steps:

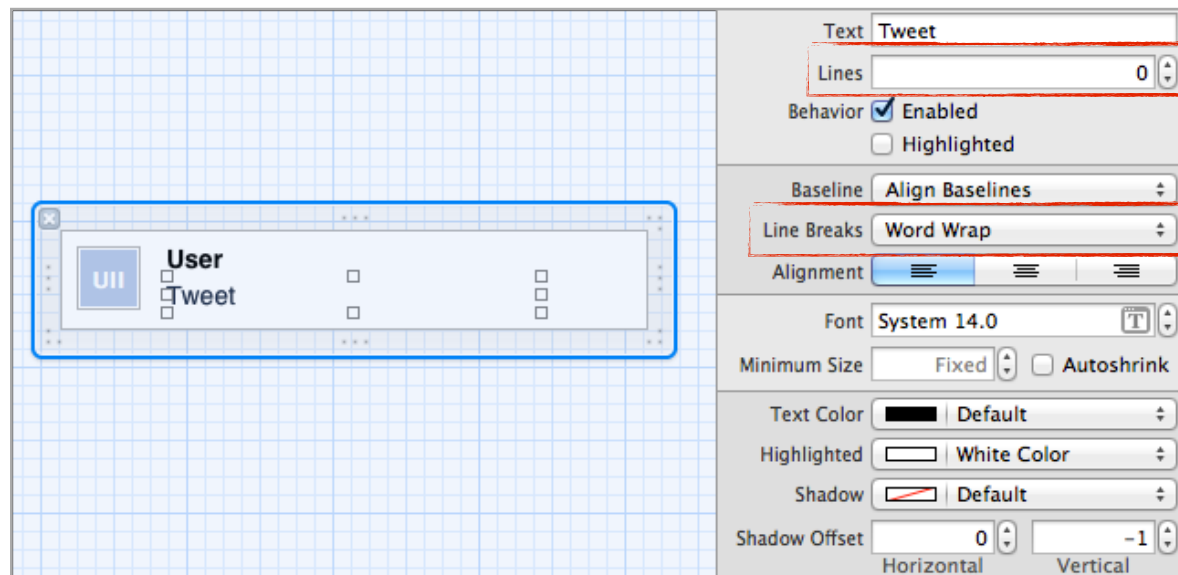
1. Drag a Table View Cell from Object Library into the empty canvas (make sure the Object Library is displayed by clicking View->Utilities->Show Object Library).
2. Set the cell's class to SCCustomCell from the Identity Inspector.



3. From the Object Library, drag an Image View and two Labels into the cell. Resize the cell to an initial height of 53 (cell will later auto resize to fit contents) to make it easier to layout the controls. Set the text of one of the labels to User, and the other to Tweet. These are just placeholders to help us identify the labels and have no other use. Now set the Tag value (from Attributes Inspector) of the Image View, User Label, and Tweet Label to 1, 2, and 3 respectively. Later on, we'll be using these tags to have STV bind the data to these controls.
4. Rearrange the controls to look like the following:



Make sure the both labels' width takes the whole width of the cells. Since we want STV to autoresize the Tweet Label depending on the text it holds, we should set its Line Breaks to 'Word Wrap' and Lines to zero (indicating that it should resize indefinitely).



All we need now is to tell the tweetsSection to use our new custom cell instead of its default cell. Change the code in RootViewController.m to the following:

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

    self.title = @"#iosdev Search";

    // Create the web service definition for a 'tweet'
    SCWebServiceDefinition *tweetDef = [SCWebServiceDefinition
        definitionWithBaseURL:@"http://search.twitter.com/"
        fetchObjectsAPI:@"search.json" resultsKeyName:@"results"
        resultsDictionaryKeyNamesString:@"text;from_user_name;created_at"];
    [tweetDef.fetchObjectsParameters setValue:@"#iosdev" forKey:@"q"];
    [tweetDef.fetchObjectsParameters setValue:@"recent"
        forKey:@"result_type"];
    tweetDef.batchSizeParameterName = @"rpp";
    tweetDef.nextBatchURLKeyName = @"next_page";
    SCPropertyDefinition *textPropertyDef = [tweetDef
        propertyDefinitionWithName:@"text"];
    textPropertyDef.title = @"Tweet";
    textPropertyDef.type = SCPropertyTypeTextView;
    SCPropertyDefinition *userPropertyDef = [tweetDef
        propertyDefinitionWithName:@"from_user_name"];
    userPropertyDef.title = @"User";
    userPropertyDef.type = SCPropertyTypeLabel;
    SCPropertyDefinition *tweetedAtPropertyDef = [tweetDef
        propertyDefinitionWithName:@"created_at"];
    tweetedAtPropertyDef.title = @"Tweeted At";
    tweetedAtPropertyDef.type = SCPropertyTypeLabel;

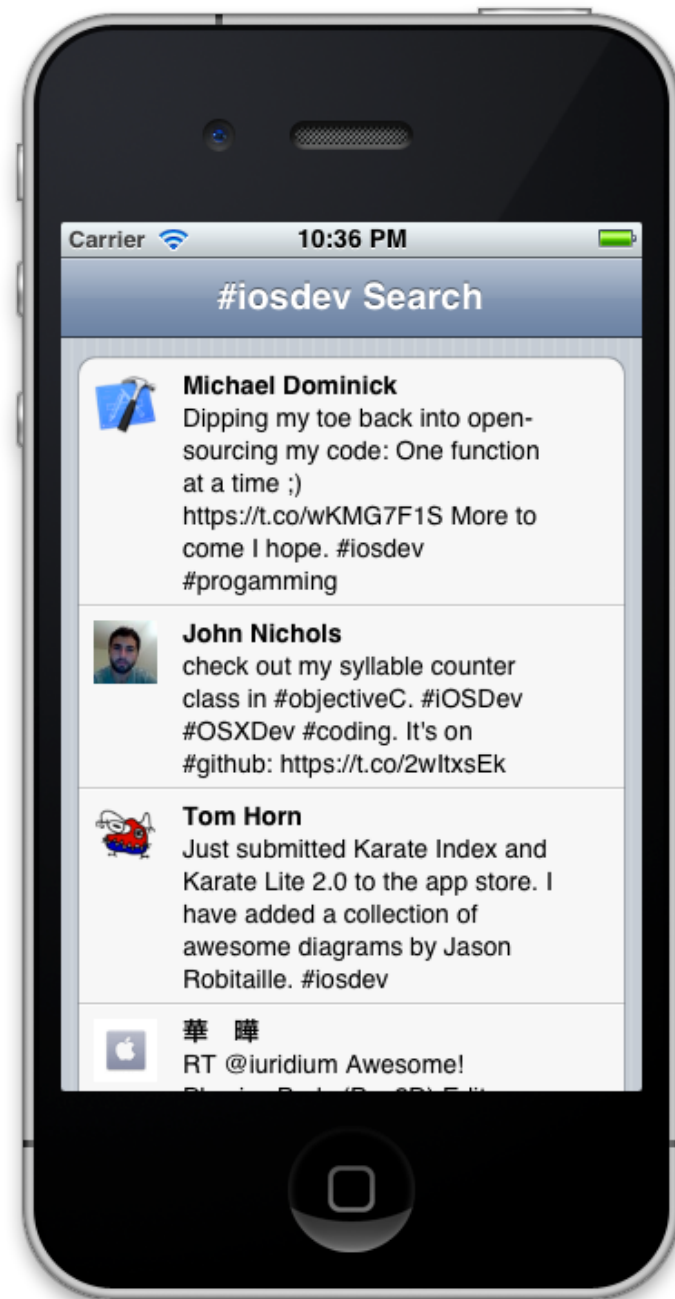
    // Enable pull-to-refresh
    self.tableViewModel.enablePullToRefresh = TRUE;

    // Add tweets section
    SCArrayOfObjectsSection *tweetsSection = [SCArrayOfObjectsSection
        sectionWithTitle:nil webServiceDefinition:tweetDef batchSize:7];
    tweetsSection.itemsAccessoryType = UITableViewCellStyleAccessoryNone;
    tweetsSection.sectionActions.cellForRowAtIndexPath =
    ^SCCustomCell*(SCArrayOfItemsSection *itemsSection, NSIndexPath *indexPath)
    {
        SCCustomCell *customCell = [SCCustomCell cellWithText:nil
            objectBindingsString:@"1:profile_image_url;2:from_user_name;3:text"
            nibName:@"TweetCell"];

        return customCell;
    };

    [self.tableViewModel addSection:tweetsSection];
}
```

Now run the app and it should look like the following:

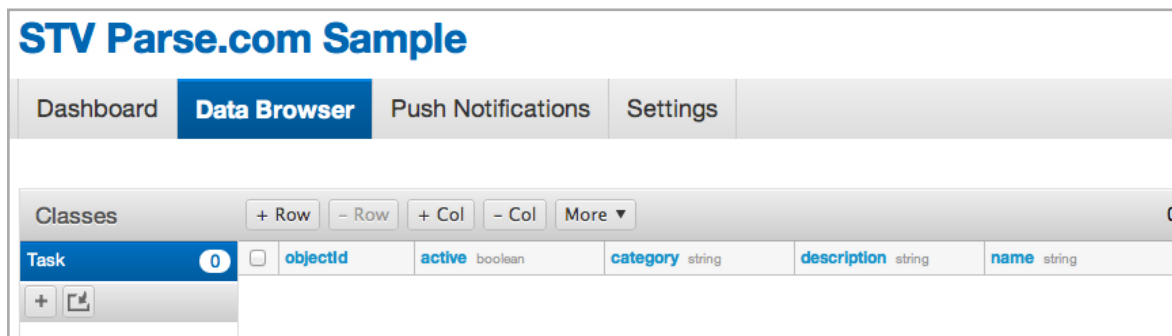


Much better! It's amazing how ridiculously simple STV makes developing such an app look like!

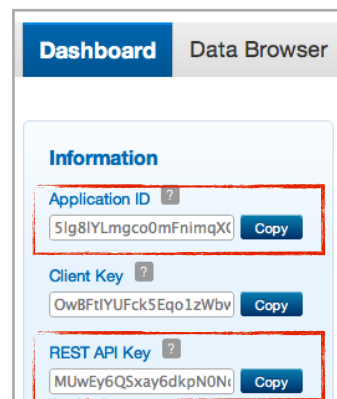
Exploring Parse.com Binding

To extend *Web Service Binding* even further, STV 3.0 fully integrates with Parse.com, which is an amazing service that enables you to easily create server-side web services. Registering is free, so if you haven't already registered, go to <http://www.parse.com> and create an account.

Once you've done that, we'll be exploring this service by creating a version of our beloved Tasks sample. Start by logging into your parse.com account, then create a new class called Task. Once you've done that, add the columns called name, description, category and active to your class as follows:



Now make sure you get back to the Dashboard and get a copy of your Application ID and REST API Key, as you'll be needing those while setting up STV.



Your Task web service should now be ready for STV to communicate with. Start by creating a new STV project based on the *Exploring Web Service Binding* section. Now select RootViewController.m and insert the following code:

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

    self.navigationController = UINavigationControllerAddLeftEditRight;

    SCParseComDefinition *taskDef = [SCParseComDefinition
        definitionWithClassName:@"Task"
        columnNamesString:@"name;description;category;active"
        applicationId:@"your_application_id"
        restAPIKey:@"your_rest_api_key"];
    SCPropertyDefinition *descPDef = [taskDef
        propertyDefinitionWithName:@"description"];
    descPDef.type = SCPropertyTypeTextView;
    SCPropertyDefinition *categoryPDef = [taskDef
        propertyDefinitionWithName:@"category"];
    categoryPDef.type = SCPropertyTypeSelection;
    categoryPDef.attributes = [SCSelectionAttributes
        attributesWithItems:[NSArray arrayWithObjects:@"Home", @"Work",
        @"Other", nil] allowMultipleSelection:NO allowNoSelection:NO];
    SCPropertyDefinition *activePDef = [taskDef
        propertyDefinitionWithName:@"active"];
    activePDef.type = SCPropertyTypeSwitch;

    SCArrayOfObjectsSection *tasksSection = [SCArrayOfObjectsSection
        sectionWithTitle:nil webServiceDefinition:taskDef batchSize:0];
    tasksSection.dataFetchOptions.sort = TRUE;
    tasksSection.addButtonItem = self.addButton;
    [self.tableViewModel addSection: tasksSection];
}
```

Compile and run the app, which should look very similar to the ones we created in the Object Binding and Core Data Binding sections. Now try adding some tasks and return back to your parse.com Data Browser. If everything has been set up correctly, you should find all the added tasks there!

Exploring User Defaults Binding

STV 3.0 has made the very common task of saving the user defaults to NSUserDefaults an extremely trivial task. Using an SCUUserDefaultsDefinition and a single line of code, you can have your user defaults UI up and running in literally one or two minutes!

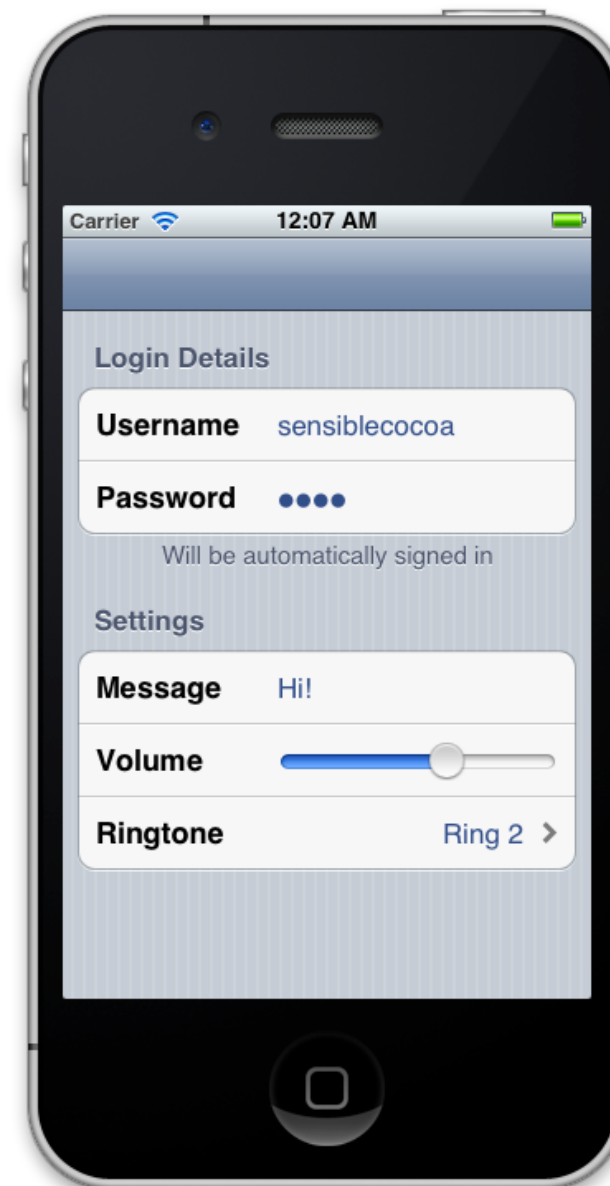
As always, we'll start by creating a new project based on the template created in *Section 3: Setting up STV*. Now select RootViewController.m and insert the following code:

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

    SCUUserDefaultsDefinition *userDefaultsDef = [SCUserDefaultsDefinition
definitionWithUserDefaultsNamesString:@"Login
Details:(username,password):Will be automatically signed in;
Settings:(message, volume, ringtone)"];
    SCPropertyDefinition *passwordDef = [userDefaultsDef
propertyDefinitionWithName:@"password"];
    passwordDef.attributes = [SCTextFieldAttributes
attributesWithPlaceholder:nil secureTextEntry:YES
autocorrectionType:UITextAutocorrectionTypeNo
autocapitalizationType:UITextAutocapitalizationTypeNone];
    SCPropertyDefinition *volumeDef = [userDefaultsDef
propertyDefinitionWithName:@"volume"];
    volumeDef.type = SCPropertyTypeSlider;
    volumeDef.attributes = [SCSliderAttributes
attributesWithMinimumValue:0 maximumValue:100];
    SCPropertyDefinition *ringtoneDef = [userDefaultsDef
propertyDefinitionWithName:@"ringtone"];
    ringtoneDef.type = SCPropertyTypeSelection;
    ringtoneDef.attributes = [SCSelectionAttributes
attributesWithItems:[NSArray arrayWithObjects:@"Ring 1", @"Ring 2",
@"Ring 3", nil] allowMultipleSelection:NO allowNoSelection:YES];

    [self.tableViewModel
generateSectionsForUserDefaultsDefinition:userDefaultsDef];
}
```

Now compile and run the app:



Exiting and reopening the app, you'll discover that all values are being automatically saved in NSUserDefaults. It really can't get any easier than this!

Understanding STV's Core Concepts

In this chapter you'll be introduced to all the basic core concepts behind STV. While an understanding of these concepts is **not** absolutely essential for you to be able to use the framework, you'll only be able to utilize STV to its fullest by having a solid understanding of all the basics discussed here.

Data Definitions

If you’ve followed the earlier ‘*Getting Started*’ chapter and utilized `SCClassDefinition` or `SCEntityDefinition` in your projects, odds are you already have an idea of what data definitions are and how essential are they to the STV framework.

Simply put, a data definition is how STV gets to “understand” what your own data model looks like. Once STV understands your data model, it will be able to generate all the required user interface elements that represent this model. Furthermore, it will understand the relationships between your different data models, and hence will accordingly reflect these relationships in the generated UI.

The following is a list of all data model types that STV supports out-of-the-box.

DATA MODEL	DATA DEFINITION CLASS
Normal Objective-C classes	<code>SCClassDefinition</code>
Core Data	<code>SCEntityDefinition</code>
Web services	<code>SCWebServiceDefinition</code>
Parse.com classes	<code>SCParseComDefinition</code>
iCloud key-value store	<code>SCiCloudKeyValueDefinition</code>
NSUserDefaults store	<code>SCUserDefaultsDefinition</code>
NSDictionary	<code>SCDictionaryDefinition</code>

While most Objective-C classes are fully definable using `SCClassDefinition`, the following special classes are an exception, and must be defined using their own data definition classes.

OBJECTIVE-C CLASS	DATA DEFINITION CLASS
<code>NSString</code>	<code>SCStringDefinition</code>
<code>NSNumber</code>	<code>SCNumberDefinition</code>
<code>NSDate</code>	<code>SCDateDefinition</code>

The previous exceptions actually allow for easily generating tables from basic data types. For example, here is the code used to display the contents of an array of NSDate values:

```
NSMutableArray *datesArray = [NSMutableArray
 arrayWithObjects:date1, date2, date3, nil];

SCArray0f0bjectsSection *datesSection = [SCArray0f0bjectsSection
 sectionWithTitle:nil items:datesArray
 itemsDefinition:[SCDateDefinition definition]];
[self.tableViewModel addSection:datesSection];
```

In addition to all the predefined data model definition classes listed above, STV also allows you to create your own custom ones, just in case you're using a special unsupported data model (a binary flat file database for example). For more on that, please refer to the chapter titled '*Extending STV*'.

SCDataDefinition

All STV data definition classes must descend from the abstract base class called `SCDataDefinition`. `SCDataDefinition` represents a generic class that is able to define the structure of any given data model. At its core, it consists of a list of *property definitions* of type `SCPropertyDefinition`. Each property definition represents an element of the underlying data model. For example, in an `SCClassDefinition`, each property definition resembles a class property. In an `SCEntityDefinition`, each property definition resembles an entity attribute. In an `SCDictionaryDefinition`, each property definition resembles a dictionary key, and so on.

Most of the property definitions are typically created automatically at the data definition's initialization for you from the information provided to the constructor method. You are still however able to manually add any custom property definitions you may have. There will be examples throughout the whole book that illustrate this.

SCPropertyDefinition

As stated above, property definitions are the building blocks of SCDataDefinition. The purpose of a property definition is to fully describe how the UI is generated for the data model element it resembles. For example, in the ‘Exploring Object Binding’ section of the previous chapter, the property definition was used to have STV generate a text view for the Task’s ‘description’ property.

The type of the generated UI element is specified in SCPropertyDefinition’s ‘type’ property. The following is a list of all the supported type values:

TYPE VALUE	GENERATED UI
SCPropertyTypeAutoDetect	STV automatically detects what UI to generate. This is the default type value.
SCPropertyTypeLabel	A cell with UILabel (SCLabelCell)
SCPropertyTypeTextView	A cell with UITextView (SCTextViewCell)
SCPropertyTypeTextField	A cell with UITextField (SCTextFieldCell)
SCPropertyTypeNumericTextField	A cell with UITextField that only accepts numeric values (SCNumericTextFieldCell)
SCPropertyTypeSlider	A cell with UISlider (SCSliderCell)
SCPropertyTypeSegmented	A cell with UISegmentedControl (SCSegmentedCell)

TYPE VALUE	GENERATED UI
SCPropertyTypeSwitch	A cell with UISwitch (SCSwitchCell)
SCPropertyTypeDate	A cell that provides a date picker (SCDateCell)
SCPropertyTypeImagePicker	A cell that provides an image picker (SCImagePickerCell)
SCPropertyTypeSelection	A cell that automatically generates a detail view of selection strings (SCSelectionCell)
SCPropertyTypeObjectSelection	A cell that automatically generates a detail view of selection objects (SCObjectSelectionCell)
SCPropertyTypeObject	A cell that automatically generates a detail view displaying all the associated object’s properties (SCObjectCell)
SCPropertyTypeArrayOfObjects	A cell that automatically generates a detail view displaying an array of objects (SCArrayOfObjectsCell)
SCPropertyTypeCustom	Generates a custom user-defined cell. Set the property definition’s ‘uiElementClassName’ or ‘uiElementNibName’ properties to the custom cell.
SCPropertyTypeNone	Does not generate any UI
SCPropertyTypeUndefined	Reserved for internal framework use

For each property definition type, only certain property data types are allowed. For example, a property definition of type `SCPropertyTypeTextField` only supports properties of data type `NSString`. Following is a list of all supported data types for each property type:

TYPE VALUE	SUPPORTED DATA TYPE
<code>SCPropertyTypeAutoDetect</code>	Any
<code>SCPropertyTypeLabel</code>	Any
<code>SCPropertyTypeTextView</code>	<code>NSString</code>
<code>SCPropertyTypeTextField</code>	<code>NSString</code>
<code>SCPropertyTypeNumericTextField</code>	<code>NSNumber</code> , <code>int</code> , <code>float</code> , <code>double</code>
<code>SCPropertyTypeSlider</code>	<code>NSNumber</code> , <code>int</code> , <code>float</code> , <code>double</code>
<code>SCPropertyTypeSegmented</code>	<code>NSNumber</code> , <code>int</code>
<code>SCPropertyTypeSwitch</code>	<code>NSNumber</code> , <code>BOOL</code>
<code>SCPropertyTypeDate</code>	<code>NSDate</code>
<code>SCPropertyTypeImagePicker</code>	<code>NSString</code> containing the image path
<code>SCPropertyTypeSelection</code>	<code>NSString</code> , <code>NSNumber</code> , <code>NSMutableSet</code>
<code>SCPropertyTypeObjectSelection</code>	<code>NSObject</code>
<code>SCPropertyTypeObject</code>	<code>NSObject</code>
<code>SCPropertyTypeArrayOfObjects</code>	<code>NSMutableArray</code>
<code>SCPropertyTypeCustom</code>	N/A
<code>SCPropertyTypeNone</code>	N/A
<code>SCPropertyTypeUndefined</code>	N/A

To set the type of an automatically generated property definition, it's very common to retrieve it using the owner data definition's `propertyDefinitionWithName:` method.

```
SCPropertyDefinition *descPropertyDef =
    [taskDef propertyDefinitionWithName:@"description"];
descPropertyDef.type = SCPropertyTypeTextView;
```

Finally, most property types can have the attributes of their generated UI elements further configured by setting the property definition's `attributes` property to a compatible class (they usually have the same name, e.g.: `SCPropertyTypeTextField` and `SCTextFieldAttributes`). For example, to set a placeholder for the generated `UITextField` control, the following code can be used:

```
SCPropertyDefinition *namePropertyDef =
    [taskDef propertyDefinitionWithName:@"name"];
namePropertyDef.type = SCPropertyTypeTextField;
namePropertyDef.attributes = [SCTextFieldAttributes
    attributesWithPlaceholder:@"Enter task name"];
```

It's worth noting here that the above could have also been achieved by using the 'Actions' feature. Here is an example:

```
namePropertyDef.type = SCPropertyTypeTextField;
namePropertyDef.cellActions.willConfigure = ^(SCTableViewCell
    *cell, NSIndexPath *indexPath)
{
    SCTextFieldCell *textFieldCell = (SCTextFieldCell *)cell;
    textFieldCell.textField.placeholder = @"Enter task name";
};
```

For more on actions, please refer to this chapter's 'Actions' section.

Automatic property definitions

As stated earlier, most of your property definitions will be automatically generated at initialization time to save you the hassle of having to add them manually yourself. To be able to do this, data definition initializer methods typically have a parameter called ‘propertyNamesString’ (also sometimes called ‘keyNamesString’ in some SCDataDefinition subclasses), which is a string containing all property names separated by semi colons. For example, this is how TaskDef from our ‘*Getting Started*’ chapter is initialized:

```
SCClassDefinition *taskDef = [SCClassDefinition
definitionWithClass:[Task class]
propertyNamesString:@"name;description;category;dueDate;completed"];
```

The above will place all the generated cells in the same group (section). To have the definition generate more than one group, include the property names between parenthesis, preceded by the section’s header title and a colon. The property names inside the parenthesis must be separated by commas. Here is an example:

```
@"Task Details:(name,description,category,dueDate);Task
Status:(completed)";
```

To create a section without a header title, simply remove all text before the colon:

```
@":(name,description,category,dueDate);Task Status:(completed)";
```

Similarly, to add a section footer, add a colon after the parenthesis and type the footer’s name.

```
@":(name,description,category,dueDate):Task Details;Task
Status:(completed)";
```

Finally, to automatically generate a custom property definition, place a tilde ‘~’ character before the custom property’s name.

```
@"Task Details:(name,description,category,dueDate);Task
Status:(completed);Task Actions:(~Delete Task)";
```

Custom property definitions are property definitions for properties that do not actually exist in the data model. Custom property definitions will be discussed in great detail later on in this section.

If for any reason you need to add a property definition manually, just create a new SCPropertyDefinition instance and use SCDataDefinition’s ‘addPropertyDefinition:’ method property to add it. Also, to add or manage property groups, use SCDataDefinition’s ‘propertyGroups’ property.

Custom generated UI elements

STV gives you the flexibility to have the property definition generate your own custom cell, instead of the framework's standard predefined cells. For example: if the property definition's type is `SCPropertyTypeTextView`, instead of generating the standard `SCTextViewCell`, you can have the framework generate any other cell that you have defined.

To specify what type of cell the framework should create, simply set the property definition's `uiElementClass` property to the class of your custom cell.

```
descPropertyDef.uiElementClass = [MyCustomTextViewCell class];
```

Alternatively, if you've created your custom cell as a Nib file, just set the property definition's `uiElementNibName` property to the name of this file.

```
descPropertyDef.uiElementNibName = @"MyCustomTextViewCell";
```

In the above example, if your cell is an `SCTextViewCell` subclass, then you're all done. If it's based on `SCCustomCell` however, you have to tell STV which of your cell's controls to bind the description property to.

Important: all custom cells **must** subclass `SCCustomCell` or any of its subclasses.

Setting up how STV binds to your custom cell's controls is really simple. First, you give each control in the custom cell a tag value that is **greater than zero**. Next, you set the prop-

erty definition's `objectBindingsString` property to a string containing the control tag and the property name it binds to, separated by a colon.

For example, if your custom cell has a `UITextView` control with a tag value of 1 that needs to bind to the `'description'` property:

```
descPropertyDef.objectBindingsString = @"1:description";
```

If your custom cell has more than one control that bind to more than one property (like the `TweetCell` from the *Exploring Web Service Binding* section), you can still specify more than one binding by separating them by semi colons:

```
namePropertyDef.objectBindingsString = @"1:firstName;2:lastName";
```

Custom property definitions

With STV's custom property definitions feature, you can add property definitions that, unlike normal property definitions, do not have to directly correspond to an existing element of the data model. For instance, if you're working with an `SCEntityDefinition`, a custom property definition does not have to correspond to an already existing entity attribute.

This feature provides you with maximum flexibility, and has many useful applications. Some typical applications of custom property definitions are:

- **Complex custom cells.** Sometimes you need the generated cell to have the data for more than one property. For example, you might have two properties called `firstName` and `lastName`, and you'd like to display both in a single custom cell. In this case, you can create a custom cell with two `UILabel`s, with tags 1 and 2 respectively. Next, you would add the custom cell to your data definition via a custom property definition as follows:

```
SCCustomPropertyDefinition *namePropertyDef =
    [SCCustomPropertyDefinition definitionWithName:@"full name"
    uiElementNibName:@"NameCell"
    objectBindingsString:@"1:firstName;2:lastName"];
[taskDef addPropertyDefinition:namePropertyDef];
```

- **Button Cells.** It's a relatively common task to want to add additional cells to the automatically generated ones. One typical use of this is to add cells that act like buttons, where a specific action would occur when they're tapped. Using cus-

tom property definitions makes adding such special cells possible.

```
SCCustomPropertyDefinition *buttonPropertyDef =
    [SCCustomPropertyDefinition definitionWithName:@"button"
    uiElementClass:[SCTableViewCell class]
    objectBindingsString:nil];
buttonPropertyDef.cellActions.willConfigure = ^(SCTableViewCell
*cell, NSIndexPath *indexPath)
{
    cell.textLabel.textAlignment = NSTextAlignmentCenter;
};
buttonPropertyDef.cellActions.didSelect = ^(SCTableViewCell *cell,
NSIndexPath *indexPath)
{
    // place button action here
};
[taskDef addPropertyDefinition:buttonPropertyDef];
```

If you want to place the button on its own separate section, it's usually much easier to have the framework automatically create the section and the custom property definition in the data definition's `propertyNamesString`. To specify a custom property (as opposed to a normal property) in the `propertyNamesString`, just place a tilde '~' character before the property's name.

```
SCClassDefinition *taskDef = [SCClassDefinition
    definitionWithClass:[TestObj class]
    propertyNamesString:@"Task Details:(name, description);Task
    Actions:(~button)"];
SCPropertyDefinition *buttonPropertyDef = [taskDef
    propertyDefinitionWithName:@"~button"];
buttonPropertyDef.cellActions.willConfigure = ^(SCTableViewCell
*cell, NSIndexPath *indexPath)
{
    cell.textLabel.textAlignment = NSTextAlignmentCenter;
};
buttonPropertyDef.cellActions.didSelect = ^(SCTableViewCell *cell,
NSIndexPath *indexPath)
{
    // place button action here
};
```

Data Stores

Data stores provide the STV framework with a generic interface that enables it to access any kind of data storage. Using data stores, the same STV element can fetch, create, update and delete data from any number of different sources. For instance, the same `SCArrayOfObjectsSection` object can access data from an in-memory array, Core Data, or even a remote web service.

STV provides the following data stores out-of-the-box:

DATA STORE	DATA STORE CLASS
Heap memory storage	<code>SCMemoryStore</code>
Core Data	<code>SCCoreDataStore</code>
Web services	<code>SCWebServiceStore</code>
iCloud key-value storage	<code>SCiCloudKeyValueStore</code>
<code>NSUserDefaults</code> storage	<code>SCUserDefaultsStore</code>

In addition to all the built in data stores, you can also create your own custom stores (typically with your own custom data definitions too). This is however rarely needed, as the aforementioned STV classes already cover the vast majority of all your data access needs. For more on custom data stores, please refer to the chapter titled ‘*Extending STV*’.

SCDataStore

All STV data store classes must descend from the abstract base class called `SCDataStore`. `SCDataStore` represents a generic class that is able to represent any kind of data storage. It also exposes common CRUD (create, retrieve, update, delete) methods that allows full manipulation of the data in the store.

Every data store requires at least one default *data definition* that defines the structure of the data stored in the store. Each data store also supports two modes for data access: *synchronous* and *asynchronous*. The data store mode is usually set by the `SCDataStore` subclass. For example, `SCMemoryStore` is synchronous, while `SCWebServiceStore` is asynchronous.

Fortunately, you almost don't need to worry about data stores at all. For your convenience, STV usually creates the data stores automatically for you depending on your data definition. As a matter of fact, every data definition is able to generate a compatible data store using its 'generateCompatibleDataStore' method. You are still however able to instantiate STV elements directly from data stores if you wish. For example:

```
SCArrayOfObjectsSection *objectsSection = [SCArrayOfObjectsSection
    sectionWithTitle:nil datastore:myDataStore];
```

SCMemoryStore

`SCMemoryStore` is a synchronous store that represents the heap memory storage. This store is typically used to hold your own custom Objective C classes. The following are all the data definitions that are compatible with `SCMemoryStore`:

- `SCClassDefinition`
- `SCDictionaryDefinition`
- `SCStringDefinition`
- `SCNumberDefinition`
- `SCDateDefinition`

An `SCMemoryStore` can be created in the following manner:

```
NSMutableArray *objectsArray = [NSMutableArray
    arrayWithObjects:object1,object2,object3, nil];
SCClassDefinition *objectDef = [SCClassDefinition
    definitionWithClass:[MyCustomObject class]
    autoGeneratePropertyDefinitions:YES];

SCMemoryStore *objectsStore = [SCMemoryStore
    storeWithObjectsArray:objectsArray defaultDefiniton:objectDef];
```


SCCoreDataStore

SCCoreDataStore is a synchronous store that represents Core Data storage. Using an `SCEntityDefinition` as its default data definition (the store's only compatible data definition), an `SCCoreDataStore` is typically created as follows:

```
SCEntityDefinition *entityDef = [SCEntityDefinition
    definitionWithEntityName:@"MyEntity" managedObjectContext:context
    autoGeneratePropertyDefinitions:YES];

SCCoreDataStore *entityStore = [SCCoreDataStore
    storeWithDefaultDataDefinition:entityDef];
```

SCWebServiceStore

SCWebServiceStore is an asynchronous store that represents any remote REST web service. The store has two compatible data definitions: `SCWebServiceDefinition` and `SCParseComDe-
finition`. An `SCWebServiceStore` can be created in the following manner:

```
SCWebServiceDefinition *tweetDef = [SCWebServiceDefinition
    definitionWithBaseURL:@"http://search.twitter.com/"
    fetchObjectsAPI:@"search.json" resultsKeyName:@"results"
    resultsDictionaryKeyNamesString:@"text;from_user_name"];

SCWebServiceStore *tweetsStore = [SCWebServiceStore
    storeWithDefaultWebServiceDefinition:tweetDef];
```

SCiCloudKeyValueStore

SCiCloudKeyValueStore represents the `NSUbiquitousKeyValueStore` iCloud storage. This store is typically used to store user preferences to iCloud and have it available on every iDevice they have.

Using an `SCiCloudKeyValueDefinition` as its default data definition, an `SCiCloudKeyValueStore` can be created in the following manner:

```
SCiCloudKeyValueDefinition *iCloudDef = [SCiCloudKeyValueDefinition
    definitionWithiCloudKeyNamesString:@"username;password"];
SCiCloudKeyValueStore *iCloudStore = [SCiCloudKeyValueStore
    storeWithDefaultDataDefinition:iCloudDef];
```

SCUserDefaultsStore

SCUserDefaultsStore represents the `NSUserDefaults` local storage. This store is typically used to easily store user preferences to their device.

Using an `SCUserDefaultsDefinition` as its default data definition, an `SCUserDefaultsStore` can be created in the following manner:

```
SCUserDefaultsDefinition *userDefaultsDef = [SCUserDefaultsDefinition
    definitionWithUserDefaultsKeyNamesString:@"username;password"];
SCUserDefaultsStore *userDefaultsStore = [SCUserDefaultsStore
    storeWithDefaultDataDefinition:userDefaultsDef];
```

Direct data store access

As was stated earlier in this section, you almost never have the need to create data stores yourself. Furthermore, data stores are usually transparent to you and can go completely unnoticed. Having said that, there are certain cases where it's very convenient to have direct access your application's data stores:

- **Manual data manipulation.** Even though STV usually handles all the data manipulation tasks automatically, you may often need to add or remove objects manually from the data store. Let's say you have an `SCArrayOfObjectsSection` displaying several Core Data managed objects, and you need to add an extra object manually. In that case, the object can be easily added using your section's 'dataStore' property:

```
NSObject *myObject = [objectsSection.dataStore createNewObject];
[myObject setValue:@"My new object" forKey:@"title"];
[objectsSection.dataStore insertObject:myObject];

[objectsSection reloadBoundValues];
[objectsSection.ownerTableViewModel.tableView reloadData];
```

- **Single object views.** Sometimes your entire table view is based on the presence of a single object, which is created when the application is first launched and reused afterwards (very common in Core Data based applications). In that case, directly using data stores can greatly simplify your task:

```
SCEntityDefinition *entityDef = [SCEntityDefinition
    definitionWithEntityName:@"MyEntity" managedObjectContext:context
    autoGeneratePropertyDefinitions:YES];
SCCoreDataStore *entityStore = [SCCoreDataStore
    storeWithDefaultDataDefinition:entityDef];
NSArray *fetchedObjects = [entityStore fetchObjectsWithOptions:nil];

NSObject *myObject = nil;
if(fetchedObjects.count)
    myObject = [fetchedObjects objectAtIndex:0];
if(!myObject)
{
    // Create the object for the first time
    myObject = [entityStore createNewObject];
    [entityStore insertObject:myObject];
}

[self.tableViewModel generateSectionsForObject:myObject
    withDefinition:entityDef];
```

It is needless to say that both of the above operations could've been equally done using Core Data's own native API. Using STV's data stores however provided a much more convenient alternative, and a much better and unified user experience. Furthermore, with some other data stores (such as web services) , the above simple operations are much more complicated and might take pages of code if done using native API.

Table View Models

Table view models are the master mind behind all of STV's magic. A table view model brings everything to life by acting as the datasource and delegate for a UITableView control, providing it with all its sections, cells, and a lot of other functions.

A model also monitors all events that occur on the UITableView and reports them back to the framework. Most events are automatically handled by the framework on your behalf, such as when a cell is moved or deleted. You are still however able to intercept most events and include your own custom behavior by using STV's *Actions* feature (for more on actions, please refer to the section titled *Actions*). For example, you can implement the `willDisplay` cell action to get notified when the cell is displayed, or the `didSelect` to take action when the cell is selected.

SCTableViewModel

In its simplest form, an SCTableViewModel is a collection of SCTableViewSection(s), which is in turn a collection of SCTableViewCell(s) (much more on these two classes in the next two sections). Each model models a single UITableView that is given at creation time:

```
self.tableViewModel = [SCTableViewModel  
    modelWithTableView:self.tableView];
```

If you're subclassing your view controller from SCViewController or SCTableViewController (like we did in the *Getting Started* chapter), it's very rare when you'll actually need to create the model yourself. Both of these view controllers automatically create the model for you and associate it with the view controller's `self.tableView` property. For more information on the many benefits of subclassing from STV's view controllers, please refer to the upcoming *View Controllers* section.

As discussed earlier, SCTableViewModel acts as the data-Source and delegate for its modeled UITableView (also exposed via the model's `tableView` property), implementing all the UITableViewDataSource and UITableViewDelegate methods on your behalf.

Once the model is ready, adding the different kinds of sections becomes very straight forward:

```
// Add an array of objects section
SCArrayOfObjectsSection *objectsSection = [SCArrayOfObjectsSection
sectionWithTitle:nil entityDefinition:taskDef];
objectsSection.addButtonItem = self.addButton;
[self.tableViewModel addSection:objectsSection];

// Add a basic section with a single cell that acts as a refresh but-
ton
SCTableViewSection *section = [SCTableViewSection section];
SCTableViewCell *refreshCell = [SCTableViewCell
    cellWithText:@"Refresh Tasks"
   .textAlignment:NSTextAlignmentCenter];
refreshCell.cellActions.didSelect = ^(SCTableViewCell *cell, NSIndexPath
*indexPath)
{
    [cell.ownerTableViewModel reloadBoundValues];
    [cell.ownerTableViewModel.tableView reloadData];
};
[section addCell:refreshCell];
[self.tableViewModel addSection:section];
```



SCArrayOfObjectsModel

Very similar to its `SCArrayOfObjectsSection` counterpart, `SCArrayOfObjectsModel` serves as a model that can display a list of any kind of objects. There are two major differences between the two classes however:

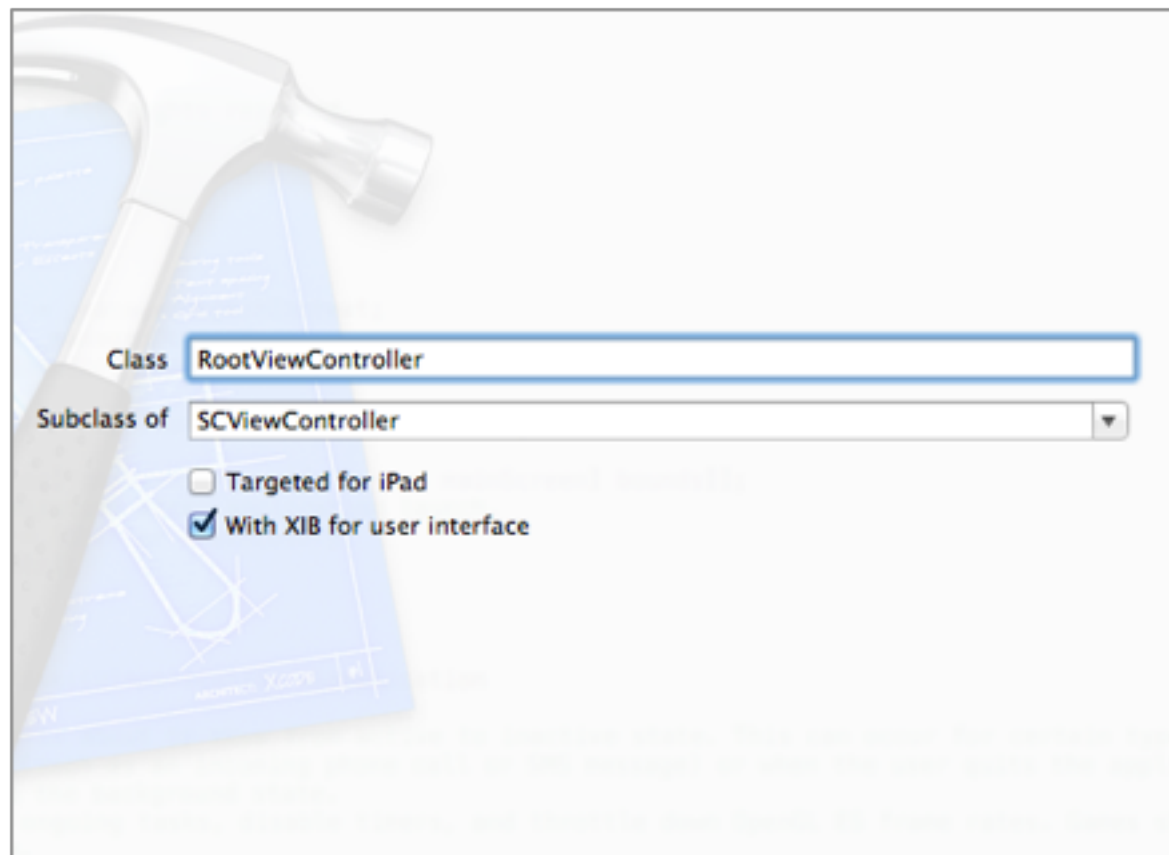
- While `SCArrayOfObjectsSection` is only a single section, `SCArrayOfObjectsModel` can display many automatically generated sections that are generated based on the data it holds. This is very useful for applications similar to Apple's 'Contacts App', where each group of contacts are grouped under a separate section based on the first letter of their name.
- `SCArrayOfObjectsModel` provides automatic searching functionality, where you could easily attach a `UISearchBar` to the model to enable the user of your application to search the contents of the model.

It's worth noting here that both `SCArrayOfObjectsModel` and `SCArrayOfObjectsSection` directly descend from the abstract base classes `SCArrayOfItemsModel` and `SCArrayOfItemsSection`, respectively. `SCArrayOfItemsModel` and `SCArrayOfItemsSection` provide all the internal plumbing for these classes, and should never be directly instantiated.

Let's create a small sample application that illustrates both these features. The sample app will be a mini version of the iPhone's Contacts App, and will be based on Core Data.

Start by creating a new project based on the project we created in the section titled *‘Exploring Core Data Binding’*, but keep in mind the following differences:

- Since we’ll be adding a UISearchBar control to our root view controller, we need to subclass ‘RootViewController’ from SCViewController, instead of SCTableViewController as we did earlier. The reason being that SCTableViewController (like its UITableViewController superclass) only supports a single view of type UITableView. Also, since we’ll be adding the UISearchBar in Interface Builder, make sure to select the ‘With XIB for user interface’ option.



- Since ‘RootViewController’ is now loaded from a XIB file, make sure you update the code in AppDelegate.m file to reflect that:

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

Once how have the project ready, please follow these steps:

1. Add an entity called ‘ContactEntity’ to your project with the following attributes:

ENTITIES	
E ContactEntity	
FETCH REQUESTS	
CONFIGURATIONS	
C Default	
Attributes	
Attribute	Type
S email	String
S firstName	String
S lastName	String
+ -	

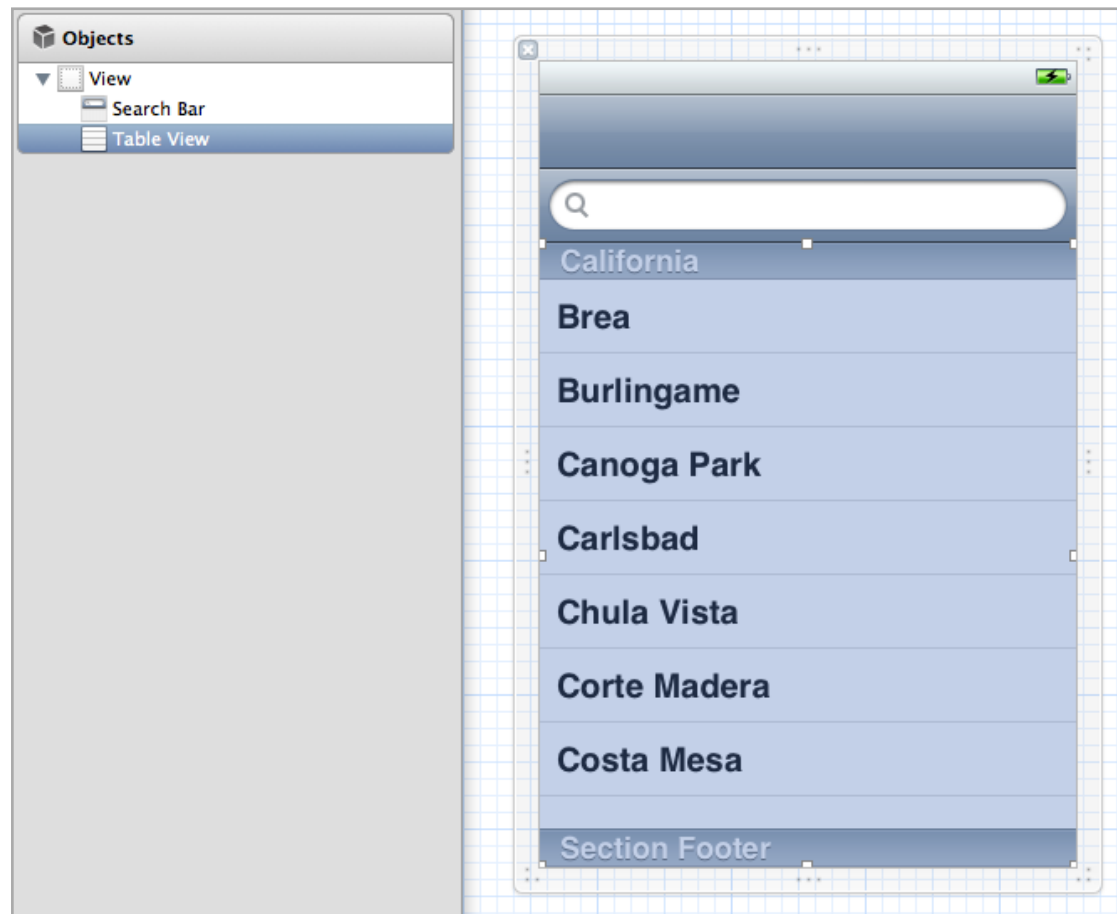
2. Navigate to ‘RootViewController.h’ and add a new IBOutlet property for our UISearchBar.

```
@interface RootViewController : SCViewController
@property (nonatomic, strong) IBOutlet UISearchBar *searchBar;
@end
```

Make sure you also synthesize the property in RootViewController’s implementation file (.m).

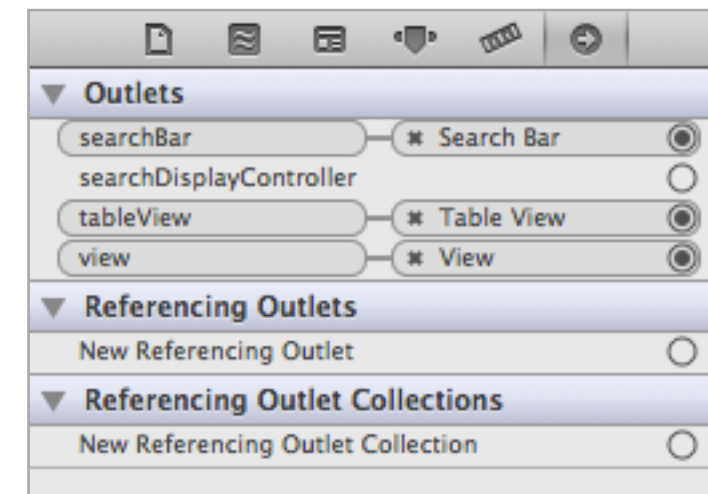
3. Navigate to ‘RootViewController.xib’ and make sure the empty view is selected. From the ‘Attributes inspector’,

set the simulated ‘Top Bar’ attribute to ‘Navigation Bar’, which will better help you visualize the generated view as you lay out the controls. From the ‘Object Library’ add a UISearchBar and a UITableView control, then lay them out similar to the following:



4. Now we need to connect the UISearchBar and UITableView controls to their respective outlets in RootViewController (SCViewController automatically provides a UITableView outlet called ‘tableView’). To do that, Control-click the ‘File’s Owner’ object and drag to the UISearchBar, then select ‘searchBar’. Similarly, Control-click the

‘File’s Owner’ object and drag to the UITableView, then select ‘tableView’. To make sure everything is properly connected, just bring up the ‘Connections inspector’ and you should see something identical to the following:



5. Finally, update RootViewController.m with the following code:

```

@implementation RootViewController

@synthesize searchBar;

- (void)viewDidLoad
{
    [super viewDidLoad];

    // Add an 'Add' and 'Edit' buttons to the navigation bar
    self.navigationController.navigationBarType = UINavigationControllerTypeAddLeftEditRight;

    NSManagedObjectContext *context = [(id)[UIApplication
sharedApplication].delegate managedObjectContext];

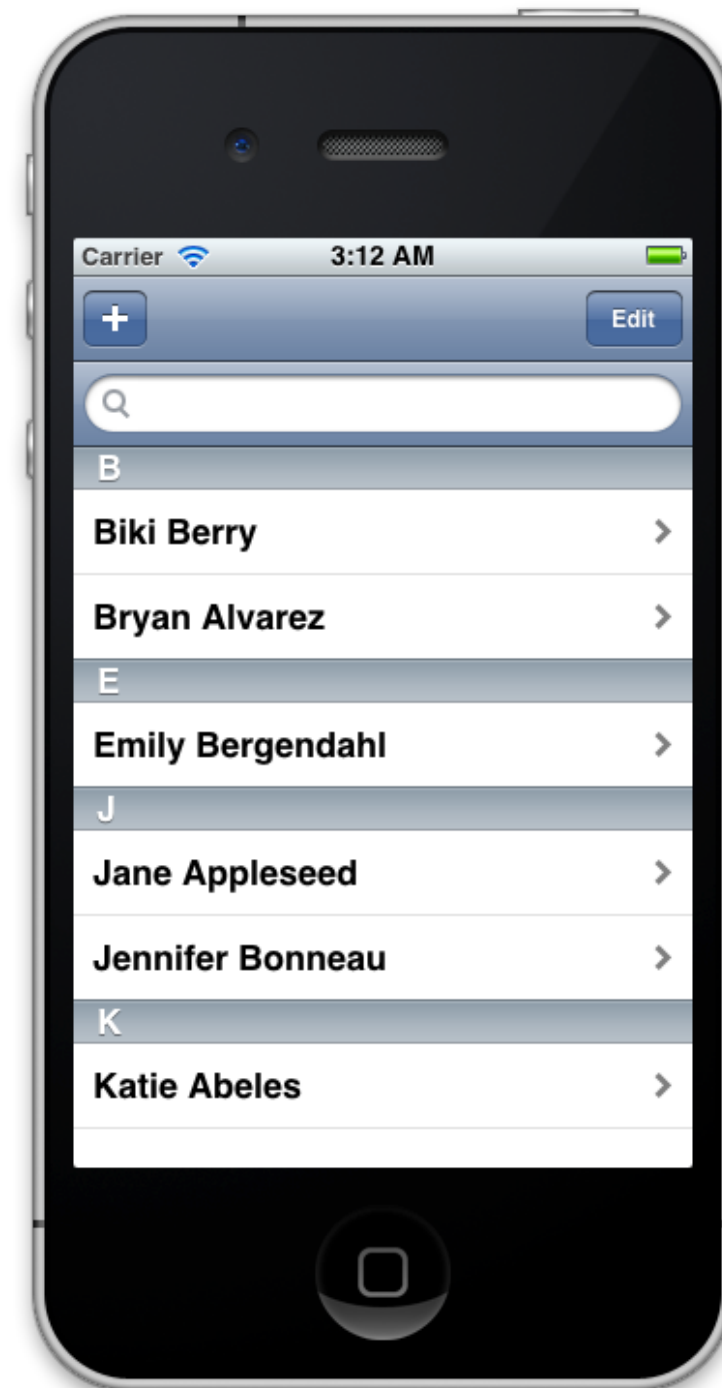
    SCEntityDefinition *contactDef = [SCEntityDefinition
definitionWithEntityName:@"ContactEntity"
managedObjectContext:context
propertyNameString:@"firstName;lastName;email"];
    // Set the generated cell's title
    contactDef.titlePropertyName = @"firstName;lastName";

    // We need to set self.tableViewModel to an
    // SCArrayOfObjectsModel instance, since SCViewController
    // creates it as SCTableViewModel by default
    SCArrayOfObjectsModel *contactsModel = [SCArrayOfObjectsModel
modelWithTableView:self.tableView entityDefinition:contactDef];
    contactsModel.addButtonItem = self.addButton;
    contactsModel.searchBar = self.searchBar;
    //search by both first and last names
    contactsModel.searchPropertyName = @"firstName;lastName";
    contactsModel.autoSortSections = YES;
    contactsModel.modelActions.sectionHeaderTitleForItem =
^NSString*(SCArrayOfItemsModel *itemsModel, NSObject *item,
NSUInteger itemIndex)
    {
        NSString *objectName = (NSString *) [item
valueForKey:@"firstName"];

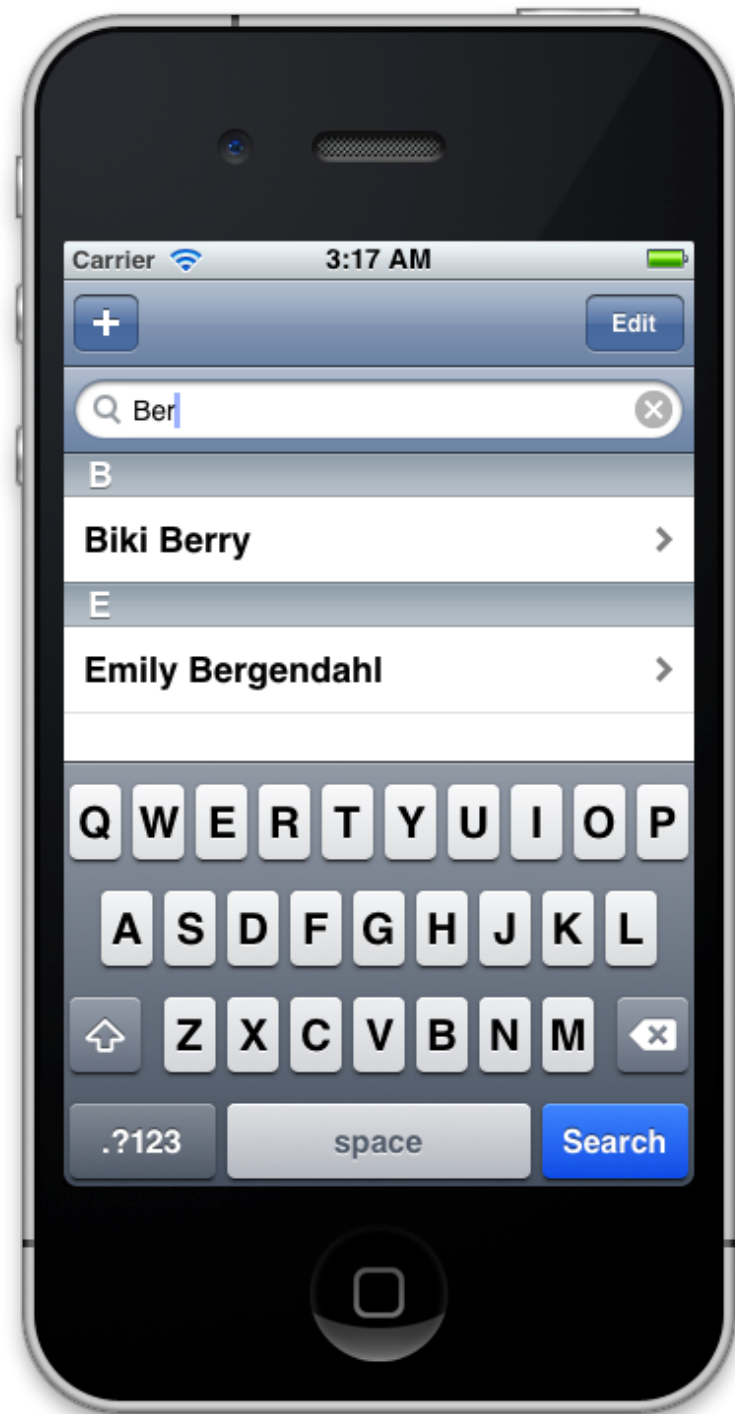
        // Return first character of objectName
        return [[objectName substringToIndex:1] uppercaseString];
    };
    contactsModel.dataFetchOptions.sort = TRUE; // Sort names
    self.tableViewModel = contactsModel;
}

```

Running the app and entering a few names should create something like this:



Notice all the sections that have been automatically generated. Furthermore, notice how entering characters into the search box automatically filters the table view:



Sections

STV’s sections correspond to regular UITableView sections, and are essentially a collection of SCTableViewCell(s). STV has many types of sections, starting from regular sections where you add the cells manually, and all the way up to sections that automatically generate their cells depending on how your data is structured.

Note: Using STV, you never need to worry about handling cell reuse, no matter what type of section you’re using or how you’re adding the cells to it. The STV framework automatically handles all cell reuse chores, in addition to other optimization functions that make your table view as responsive and as memory efficient as possible.

The following is a list of the different types of sections available:

SECTION CLASS	
SCTableViewSection	A basic section where cells are added manually using its ‘addCell’ method. This section is also serves as the superclass of all other sections.
SXObjectSection	A section that automatically generates its cells from the properties/attributes of a given object.
SCArrayOfItemsSection	An abstract base class that servers as the superclass of several classes such as SCArrayOfObjectsSection and SCSelectionSection
SCArrayOfObjectsSection	A section that automatically generates its cells from an list of any type of objects.
SCArrayOfStringsSection	A section that automatically generates its cells from a list of NSStrings.
SCSelectionSection	A section that provides selection functionality from a list of strings.
SXObjectSelectionSection	A section that provides selection functionality from a list of objects.

SCTableViewSection

This type of basic section is used internally as a superclass to all the other types of sections. You can also use it when you want to add sections manually to your table view (versus adding them automatically using data definitions).

Adding a new SCTableViewSection all with a header and footer to your model is really simple:

```
SCTableViewSection *section = [SCTableViewSection
    sectionWithTitle:@"Header" footerTitle:@"Footer"];
[self.tableViewModel addSection:section];
```

You can also specify header/footer views instead of regular text if you need custom headers/footers.

```
SCTableViewSection *section = [SCTableViewSection section];
UILabel *headerLabel = [[UILabel alloc] init];
headerLabel.text = @"Header";
headerLabel.textAlignment = NSTextAlignmentCenter;
headerLabel.font = [UIFont fontWithName:@"Zapfino" size:14];
headerLabel.backgroundColor = [UIColor clearColor];
[headerLabel sizeToFit];
section.headerView = headerLabel;
[self.tableViewModel addSection:section];
[section addCell:[SCTableViewCell cell]];
```



Notice that STV automatically calculates the header's height based on the height of the header view. If you still need to fine tune it or even set the header height yourself, just set the section's 'headerHeight' property to the desired height. The same functionality is available for footers as well.

SCObjectSection

Given any object and its data definition, SCObjectSection automatically creates all its cells based on the object's properties and their types.

You rarely ever need to create this section manually, as its usually created automatically by the framework. If you have an object that you want to create the cells for, it's always recommended to use your model's generateSectionsForObject method instead of creating an SCObjectSection directly. The reason for this is that the object's definition may have more than one section defined in its propertyNameString (as we've seen in many earlier examples).

```
[self.tableViewModel generateSectionsForObject:myObject
    withDefinition:myObjectDef];
```

Once the object section is created, you can always get the respective cell generated for each property using the section's 'cellForPropertyName:' method.

```
SCTableViewCell *firstNameCell = [myObjectSection
    cellForPropertyName:@"firstName"];
```

SCArrayOfObjectsSection

This is one of STV's most popular sections, and it's usually the starting point for many STV based applications. As you've probably noticed in most of the samples created earlier, an SCArrayOfObjectsSection takes a list of objects from a data store and displays them each in its own cell. When each cell is tapped, the section automatically creates a detail view with one or more SCObjectSection(s) that give the user access to the object's properties. SCArrayOfObjectsSection also conveniently handles adding new objects, rearranging objects order (if supported by the data store), and deleting objects. The functionality provided by this section can literally save you days of work for each single time you use it!

An SCArrayOfObjectsSection can be created using different initializer methods. You should always use the correct initializer for your data store type:

- Regular NSMutableArray of objects (SCMemoryStore):

```
SCArrayOfObjectsSection *objectsSection = [SCArrayOfObjectsSection
sectionWithHeaderTitle:nil items:objectsArray
itemsDefinition:objectDef];
```

Note: objectsArray **must** be of type NSMutableArray since the section needs to support operations such as adding, removing, and rearranging objects.

Note: if the elements of objectsArray are of basic data types such as NSString, NSNumber, or NSDate, you should use the SCStringDefinition, SCNumberDefinition, or SCDateDefinition respectively for the section's definition parameter.

- Core Data objects (SCCoreDataStore):

```
SCArrayOfObjectsSection *objectsSection = [SCArrayOfObjectsSection
sectionWithHeaderTitle:nil entityDefinition:entityDef];
```

The above initializer fetches all the objects of the given entity-Def. To fetch only a subset of the objects, just provide an NSPredicate using the following initializer:

```
SCArrayOfObjectsSection *objectsSection = [SCArrayOfObjectsSection
sectionWithHeaderTitle:nil entityDefinition:entityDef
filterPredicate:predicate];
```

Note: The above initializers are only available when the *STV+CoreData* framework extension is added to your project.

- Web Service dictionary objects (SCWebServiceStore):

```
SCArrayOfObjectsSection *objectsSection = [SCArrayOfObjectsSection
sectionWithHeaderTitle:nil webServiceDefinition:webServiceDef];
```

To retrieve the data in small batches, pass in the batch size using the following initializer:

```
SCArrayOfObjectsSection *objectsSection = [SCArrayOfObjectsSection
sectionWithHeaderTitle:nil webServiceDefinition:webServiceDef
batchSize:50];
```

Note: The above initializers are only available when the *STV+WebServices* framework extension is added to your project.

Once an `SCArrayOfObjectsSection` is initialized, it fetches all its data from the provided data store, then generates a cell per each fetched object. Each generated cell is a basic `SCTableViewCell`, and is automatically setup as follows:

- The cell's `titleLabel.text` property is set to the value of the property name(s) provided in the object definition's `titlePropertyName` property (a property of `SCDataDefinition`).
- The cell's `detailTextLabel.text` property is set to the value of the property name(s) provided in the object definition's `descriptionPropertyName` property (a property of `SCDataDefinition`).
- The cell's `boundObject` property is set to the object it corresponds to.

As with any other cell, you can override this setup in the `SCCellActions` action called 'willConfigure'.

```
objectsSection.cellActions.willConfigure = ^(SCTableViewCell *cell,
NSIndexPath *indexPath)
{
    // Add the cell's index before its title
    NSString *title = [NSString stringWithFormat:@"%i- %@",
        indexPath.row, cell.titleLabel.text];
    cell.titleLabel.text = title;
};
```

To enable the section to add new objects, you need to do any of the following:

- Create an 'Add' `UIBarButtonItem` and attach the section to it. If you're using an `SCTableViewController` or an `SCViewController`, you can have it create the 'Add' button automatically by setting its 'navigationBarType' property (for much more details on this, please check out the upcoming *View Controllers* section):

```
self.navigationBarType = SCNavigationBarTypeAddLeftEditRight;
objectsSection.addButtonItem = self.addButton;
```

- Have `SCArrayOfObjectsSection` generate an extra 'Add' cell that adds a new object when it's tapped. You can do that by setting the section's 'addItemCell' property to any cell you want:

```
objectsSection.addItemCell = [SCTableViewCell
    cellWithText:@"Add new object"
    textAlign: NSTextAlignmentCenter];
```

- Dispatch the add new item event yourself in response to any of your custom actions:

```
-(void)myCustomButtonAction {
    SCArrayOfObjectsSection *objectsSection = (SCArrayOfObjectsSection *)
        [self.tableViewModel sectionAtIndex:0];
    [objectsSection dispatchEventAddNewItem];
}
```

In addition to the automatically generated `SCTableViewCell`(s), you can also specify your own custom cells that will appear instead of the standard ones. To do that, just return your custom cell in the `SCSectionActions` action called `'cellForRowAtIndexPath'` (we already saw that in the *Getting Started* chapter, under the *Exploring Web Service Binding* section).

```
objectsSection.sectionActions.cellForRowAtIndexPath =
    ^SCCustomCell*(SCArrayOfItemsSection *itemsSection, NSIndexPath *indexPath)
{
    // '1' and '2' are the tags of the labels corresponding to
    // the firstName and lastName object properties
    NSString *bindingsString = @"1:firstName;2:lastName";

    SCCustomCell *customCell = [SCCustomCell cellWithText:nil
        objectBindingsString:bindingsString nibName:@"MyCustomCell"];

    return customCell;
};
```

When a custom cell is returned, STV automatically calculates its height depending on the size of the cell's controls. For example, you might have a multi-line UILabel with text that exceeds the height you set for it in Interface Builder, and STV will automatically resize the cell to fit its new size (we've seen an identical case in the *Getting Started* chapter, under *Exploring Web Service Binding*, where each cell automatically resized to fit a tweet). If you don't want STV to automatically resize the cells, just set the section's `'enableCellAutoResizing'` property to `FALSE`.

It is also possible to return more than one custom cell, depending on the index of the cell requested. It is also possible to have STV use its standard default cell for a specific index just by returning `nil`. In the case where more than one custom cell is returned, the `'reuseIdentifierForRowAtIndexPath'` action

must also be implemented, returning a unique string for each of the custom cells. Here is some sample code illustrating this:

```
objectsSection.sectionActions.cellForRowAtIndexPath = ^SCCustom-
Cell*(SCArrayOfItemsSection *itemsSection, NSIndexPath *indexPath)
{
    SCCustomCell *customCell;
    if(indexPath.row % 2)
        customCell = [[MyCustomEvenCell alloc] init];
    else
        customCell = [[MyCustomOddCell alloc] init];

    return customCell;
};

objectsSection.sectionActions.reuseIdentifierForRowAtIndexPath =
    ^NSString*(SCArrayOfItemsSection *itemsSection, NSIndexPath *index-
    Path)
{
    NSString *reuseId;
    if(indexPath.row % 2)
        reuseId = @"EvenCell";
    else
        reuseId = @"OddCell";

    return reuseId;
};
```

SCArrayOfStringsSection

This section is a direct descendant of `SCArrayOfObjectsSection` that enables you to easily display the contents of an `NSMutableArray` of `NSStrings`. This section is typically used to easily create a menu popover with several choices to choose from. The following is an example of a typical use of `SCArrayOfStringsSection`:

```
NSMutableArray *menuArray = [NSMutableArray arrayWithObjects:@"Menu
Item 1", @"Menu Item 2", @"Menu Item 3", nil];
SCArrayOfStringsSection *menuSection = [SCArrayOfStringsSection
    sectionWithTitle:@"Select Item" items:menuArray];
menuSection.cellActions.didSelect = ^(SCTableViewCell *cell, NSIndexPath
*indexPath)
{
    switch (indexPath.row)
    {
        case 0:
            // Item 1 selected
            break;
        case 1:
            // Item 2 selected
            break;
        case 2:
            // Item 3 selected
            break;;
    }
};
[self.tableViewModel addSection:menuSection];
```

It's worth noting here that we could still have used a regular `SCArrayOfObjectsSection` to display the strings array. This is made possible by the `SCStringDefinition` class, since `SCArrayOfObjectsSection` always requires an object definition at initialization:

```
SCArrayOfObjectsSection *menuSection = [SCArrayOfObjectsSection
    sectionWithTitle:@"Select Item" items:menuArray
    itemsDefinition:[SCStringDefinition definition]];
```

SCSelectionSection

This section is a direct descendant of `SCArrayOfStringsSection` that provides selection functionality out of a given array of strings. Once the user does their selection, the index of the selected item is stored in the section's 'selectedItemIndex' property. Furthermore, the index is also assigned to the section's `boundObject`'s `boundPropertyName` (if set).

It is very seldom when you actually need to create this section yourself, as it's usually created automatically by your data definitions.

```
NSMutableArray *items = [NSMutableArray
    arrayWithObjects:@"Choice 1", @"Choice 2", @"Choice 3", nil];
SCSelectionSection *selectionSection = [SCSelectionSection
    sectionWithTitle:nil items:items];
[self.tableViewModel addSection:selectionSection];
```



SCObjectSelectionSection

This section is a direct descendant of `SCArrayOfObjectsSection` that provides selection functionality out of a given array of objects. Once the user does their selection, the index of the selected object is stored in the section's `selectedIndex` property. Furthermore, the object itself is assigned to the section's `boundObject`'s `boundPropertyName` (if set).

It is very seldom when you actually need to create this section yourself, as it's usually created automatically by your data definitions.

Cells

STV’s cells correspond to regular UITableView cells. As a matter of fact, STV’s SCTableViewCell directly descends from UITableViewCell. To provide a significantly better and enjoyable developer’s experience, the STV framework defines several types of other specialized cells. The following is a list of the different types of cells available:

CELL CLASS	
SCTableViewCell	A basic cell that serves as a superclass for all the other cell types. Any cell used with the STV framework must descend from this cell.

CELL CLASS	
SCControlCell	Serves as the superclass for any cell with a UIControl control.
SCLabelCell	A cell with a UILabel control that is used to display static text.
SCTextViewCell	A cell with a UITextView control. This cell automatically resizes to fit the contents of its text view.
SCTextFieldCell	A cell with a UITextField control.
SCNumericTextFieldCell	A cell with a UITextField control that only accepts numeric data.
SCSliderCell	A cell with a UISlider control.
SCSegmentedCell	A cell with a UISegmentedControl.
SCSwitchCell	A cell with a UISwitch control.
SCDateCell	A cell that provides a UIDatePicker to choose a date with.
SCImagePickerCell	A cell that provides a UIImagePickerController to pick an image with.
SCSelectionCell	A cell that automatically generates a detail view with an SCSelectionSection.
SCObjectSelectionCell	A cell that automatically generates a detail view with an SCObjectSelectionSection.

CELL CLASS	
<code>SCObjectCell</code>	A cell that automatically generates a detail view with an <code>SCObjectSection</code> .
<code>SCArrayOfObjectsCell</code>	A cell that automatically generates a detail view with an <code>SCArrayOfObjectsSection</code> .
<code>SCCustomCell</code>	This cell enables you to define your own custom cells. All custom cells must be an instance or a subclass of this cell.

We will now explore each kind of cell in more detail.

SCTableViewCell

This cell is the most basic of all STV cells. Directly descending from `UITableViewCell`, `SCTableViewCell` gives you full access to all the properties and methods of a regular cell. In addition to that, `SCTableViewCell` provides many other convenience properties not available in `UITableViewCell`, such as the ‘height’ and ‘movable’ properties.

In addition to that, `SCTableViewCell` provides many of the plumbing required by all the other STV cell types. For instance, it provides properties such as ‘boundObject’ and ‘boundValue’, which are essential for data manipulation.

What is *boundObject*, *boundPropertyName*, and *boundValue*?

As you probably know by now, STV cells are able to automatically retrieve data from your objects’ properties, edit it, and set it back. To be able to do that, each cell needs to get hold of the object it’s retrieving the data from, and the specific property name that the cell is working with. If the cell is automatically created, the STV framework sets its ‘boundObject’ property to this object. Similarly, it sets the ‘boundPropertyName’ to the object’s property name that the cell represents. Once these two properties are set, the ‘boundValue’ property returns the value of the given property. If you decide to create any of the cell types manually and would like the cell to be fetching its data automatically, then you should always provide the `boundObject` and `boundPropertyName` values in the cell’s initializer.

SCControlCell

SCControlCell serves as the superclass for all the popular STV control cells, such as SCTextFieldCell and SCSwitchCell.

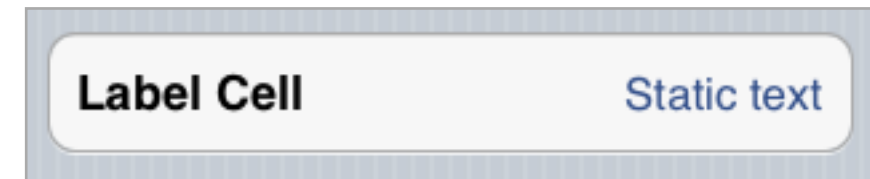
One of the most convenient functionality that SCControlCell provides is a set of properties that control the layout of the text and the UIControl of its subclasses. Here is an overview of these properties:

- `controlMargin` : The margin between the cell's textLabel and the control. Since the cell's control usually sticks to the textLabel and moves along with it as it gets smaller or bigger, it's usually visually appealing to leave a margin between the textLabel and the control. This property defaults to 10 points.
- `controlIndentation` : This property controls the space between the control and the left edge of the cell. Although the default behavior is for the control to stick to the textLabel, you might want to omit this behavior if you have several cells under each other and want all the controls aligned. Setting this value makes the control respect a minimum distance between it and the cell's edge, thus ignoring the textLabel unless it grows past this distance. This property defaults to 120 points.
- `maxTextLabelWidth` : This property controls the maximum width of the cell's text label and defaults to 200 points.

SCLabelCell

This is a cell with a right-aligned UILabel control that is used to hold static text. The cell is usually automatically generated for property definitions of type SCPropertyTypeLabel. If you wish, you could also create the cell manually and directly assign text to its label control:

```
SCLabelCell *labelCell = [SCLabelCell cellWithText:@"Label Cell"];  
labelCell.label.text = @"Static text";  
[section addCell:labelCell];
```

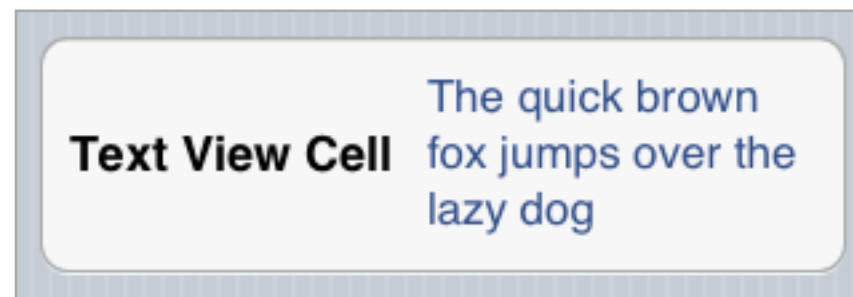


The bound property for this cell can be of any valid Objective C type.

SCTableViewCell

This is a cell with an auto-resizing UITextView control that is used to hold any variable amount of text. The cell is usually automatically generated for property definitions of type SCPropertyTypeTextView. If you wish, you could also create the cell manually and directly assign text to its textView control:

```
SCTableViewCell *textViewCell = [SCTableViewCell  
    cellWithText:@"Text View Cell"];  
textViewCell.textView.text =  
    @"The quick brown fox jumps over the lazy dog";  
[section addCell:textViewCell];
```



If you wish, you can always set a minimum height and a maximum height for the cell, further controlling how the cell resizes:

```
textViewCell.minimumHeight = 100;  
textViewCell.maximumHeight = 200;
```

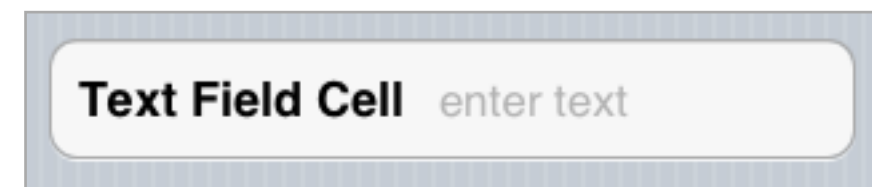
Furthermore, you can disable the auto-resizing functionality altogether by setting the cell's 'autoResize' property (inherited from SCTableViewCell) to FALSE.

The bound property for this cell must be of type NSString.

SCTextFieldCell

This is a cell with a UITextField control that is used to hold a small amount of text. The cell is usually automatically generated for property definitions of type SCPropertyTypeTextField. If you wish, you could also create the cell manually and directly assign text to its textField control:

```
SCTextFieldCell *textFieldCell = [SCTextFieldCell  
    cellWithText:@"Text Field Cell"];  
textFieldCell.textField.placeholder = @"enter text";  
[section addCell:textFieldCell];
```

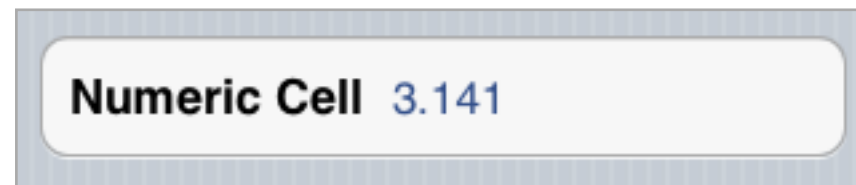


The bound property for this cell must be of type NSString.

SCNumericTextFieldCell

This is a cell with a UITextField control that is used to hold numeric data. The cell is usually automatically generated for property definitions of type SCPropertyTypeNumericTextField. If you wish, you could also create the cell manually and directly assign text to its textField control:

```
SCNumericTextFieldCell *numericFieldCell = [SCNumericTextFieldCell
    cellWithText:@"Numeric Cell"];
numericFieldCell.textField.text = @"3.141";
[section addCell:numericFieldCell];
```



The main advantage of using SCNumericTextFieldCell over a normal SCTextFieldCell is that the earlier automatically validates user input to make sure that only numbers are entered. Furthermore, you can further control the validation by specifying a minimum, maximum, and whether the number can be a float value:

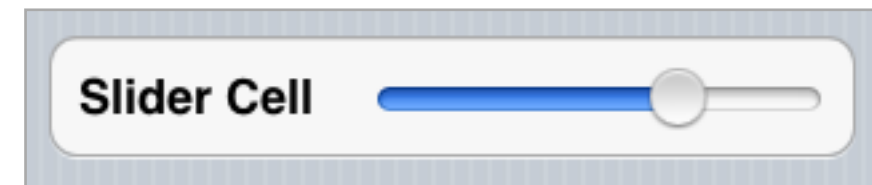
```
numericFieldCell.minimumValue = [NSNumber numberWithInt:0];
numericFieldCell.maximumValue = [NSNumber numberWithInt:100];
numericFieldCell.allowFloatValue = NO;
```

The bound property for this cell must be of types: NSNumber, int, float, or double.

SCSliderCell

This is a cell with a UISlider control that is used to represent a certain numeric value. The cell is usually automatically generated for property definitions of type SCPropertyTypeSlider. If you wish, you could also create the cell manually and directly assign a value to the cell's slider control:

```
SCSliderCell *sliderCell = [SCSliderCell
    cellWithText:@"Slider Cell"];
sliderCell.slider.value = 0.7;
[section addCell:sliderCell];
```

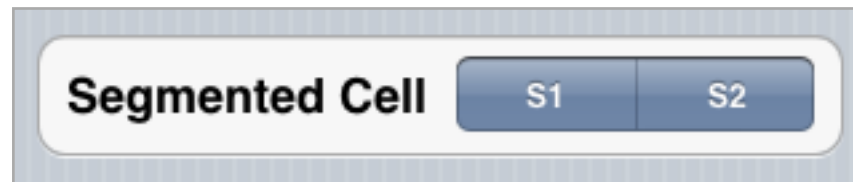


The bound property for this cell must be of types: NSNumber, int, float, or double.

SCSegmentedCell

This is a cell with a UISegmentedControl control that is used to store the index of the selected segment. The cell is usually automatically generated for property definitions of type SCPropertyTypeSegmented. If you wish, you could also create the cell manually:

```
SCSegmentedCell *segmentedCell = [SCSegmentedCell  
    cellWithText:@"Segmented Cell"];  
[segmentedCell.segmentedControl insertSegmentWithTitle:@"S1"  
    atIndex:0 animated:NO];  
[segmentedCell.segmentedControl insertSegmentWithTitle:@"S2"  
    atIndex:1 animated:NO];  
[section addCell:segmentedCell];
```

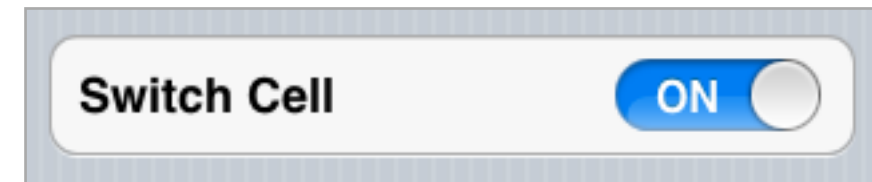


The bound property for this cell must be of types: NSNumber or int.

SCSwitchCell

This is a cell with a UISwitch control that is used to represent a boolean value. The cell is usually automatically generated for property definitions of type SCPropertyTypeSwitch. If you wish, you could also create the cell manually and directly assign an 'on' value to the cell's switch control:

```
SCSwitchCell *switchCell = [SCSwitchCell  
    cellWithText:@"Switch Cell"];  
switchCell.switchControl.on = TRUE;  
[section addCell:switchCell];
```

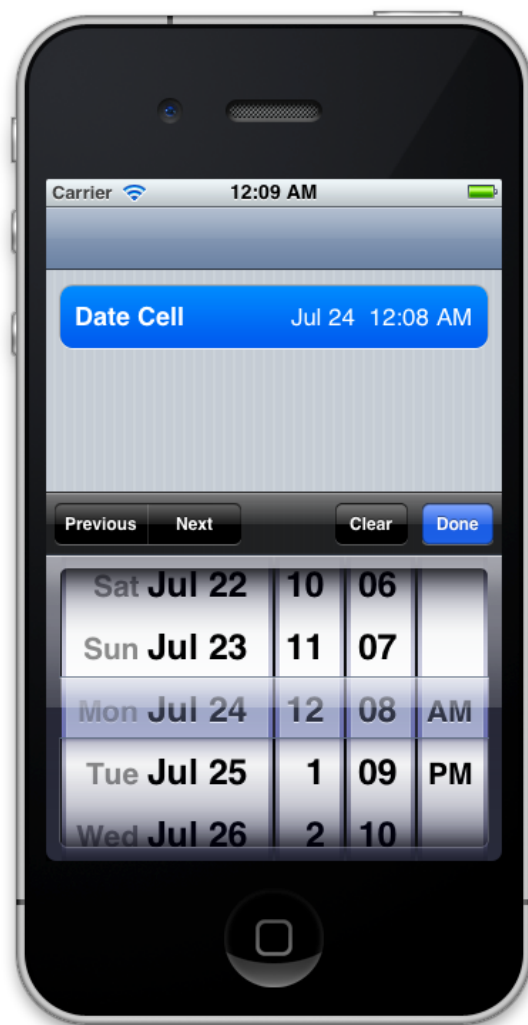


The bound property for this cell must be of types: NSNumber or BOOL.

NSDateCell

This is a cell with a UIDatePicker control that is used to represent a date value. The cell is usually automatically generated for property definitions of type `SCPropertyTypeDate`. If you wish, you could also create the cell manually and directly assign a date to the cell's `datePicker` property:

```
NSDateCell *dateCell = [NSDateCell cellWithText:@"Date Cell"];
dateCell.datePicker.date = [NSDate date];
[section addCell:dateCell];
```



The bound property for this cell must be of type `NSDate`.

View Controllers

The STV framework defines two custom view controller classes that are analogous to the iOS SDK's `UITableViewController` and `UIViewController` classes: `SCTableViewController` and `SCViewController`.

In previous versions of STV, these two classes were reserved for STV's internal use only. Due to popular demand by our users, starting STV 3.0 these classes have been completely rewritten and made publicly available for everyone to use. As a matter of fact, we now strongly recommend that you base all your view controllers on one of these two classes. Here is a list of why we recommend this:

- `SCView/TableViewController` automatically creates a table view model for you and sets it to its `'tableViewModel'` property, thus allowing you to immediately start using STV once the view controller is created. `SCViewController` also adds an `IBOutlet` property called `'tableView'` that you can assign

a `UITableView` control to either in Interface Builder or in code. Once you assign a value to `tableView`, `SCViewController` automatically pairs the table view to the model.

- `SCView/TableViewController` automatically handles all memory related issues during low memory warnings. Once a low memory warning is issued, the view controller's `tableView` is automatically released. Once the view controller is loaded, it automatically pairs the newly created `tableView` with the `tableViewModel`.
- You can automatically create all the common navigation bar buttons just by setting `SCView/TableViewController`'s `'navigationBarType'` property. A list of all the possible `navigationBarType` values will be provided in the next section.
- `SCView/TableViewController` provides several actions that let you easily determine when it has been presented or dismissed. This makes it really trivial to implement a certain behavior once the action occurs.
- Only view controllers of type `SCViewController` or `SCTableViewController` are allowed when providing your own custom detail views (via the `'detailViewControllerForRowAtIndexPath'` section action).
- Best of all, you can get all the above functionality just by renaming your `UITableViewController` and `UIViewController` superclasses to `SCTableViewController` and `SCViewController`, respectively. There isn't any development overhead whatsoever!

Navigation bar types

Both `SCTableViewController` and `SCViewController` provide a property called `'navigationBarType'`. Setting this property automatically creates a set of commonly used navigation bar controls. Here is a list of all the available values `navigationBarType`:

BAR TYPE	
<code>SCNavigationBarTypeAuto</code>	Have the framework automatically determine what navigation bar type is needed. Only applicable when the view controller is passed as a custom detail view controller.
<code>SCNavigationBarTypeNone</code>	Creates an empty navigation bar with no buttons. If the view controller is pushed into a navigation controller however, a back button is added anyways.
<code>SCNavigationBarTypeAddLeft</code>	Creates a navigation bar with an 'Add' button to the left.
<code>SCNavigationBarTypeAddRight</code>	Creates an 'Add' button to the right.
<code>SCNavigationBarTypeEditLeft</code>	Creates an 'Edit' button to the left.
<code>SCNavigationBarTypeEditRight</code>	Creates an 'Edit' button to the right.

BAR TYPE	
<code>SCNavigationBarTypeAddRightEditLeft</code>	Creates an 'Add' button to the right and an 'Edit' button to the left.
<code>SCNavigationBarTypeAddLeftEditRight</code>	Creates an 'Add' button to the left and an 'Edit' button to the right.
<code>SCNavigationBarTypeDoneLeft</code>	Creates a 'Done' button to the left.
<code>SCNavigationBarTypeDoneRight</code>	Creates a 'Done' button to the right.
<code>SCNavigationBarTypeDoneLeftCancelRight</code>	Creates a 'Done' button to the left and a 'Cancel' button to the right.
<code>SCNavigationBarTypeDoneRightCancelLeft</code>	Creates a 'Done' button to the right and a 'Cancel' button to the left.
<code>SCNavigationBarTypeAddEditRight</code>	Creates both an 'Add' button and an 'Edit' button to the right.

Once you've set the bar type, you can easily access the created buttons using the view controller's `'addButton'`, `'editButton'`, `'cancelButton'`, and `'doneButton'` properties.

SCTableViewController

SCTableViewController directly descends from UITableViewController, functioning as a view controller with a single UITableView control as its view. Since the table view control is already there, SCTableViewController is usually very convenient to use without creating an XIB file, and sometimes even without subclassing it!

You create an SCTableViewController exactly as you'd create a regular UITableViewController:

```
SCTableViewController *viewController = [[SCTableViewController
alloc] initWithStyle:UITableViewStyleGrouped];
```

If you're not subclassing SCTableViewController (just like we did above), you can start configuring the model right after initialization:

```
NSMutableArray *stringsArray = [NSMutableArray
    arrayWithObjects:@"String1", @"String2", @"String3", nil];
SCArrayOfStringsSection *stringsSection = [SCArrayOfStringsSection
    sectionWithTitle:nil items:stringsArray];

[viewController.tableViewModel addSection:stringsSection];
```

You can also start implementing actions such as the 'willPresent' to set custom behavior for the different view controller actions:

```
viewController.actions.willPresent = ^(SCTableViewController *vc)
{
    NSLog(@"viewController is about to be presented.");
};
```

If you've subclassed SCTableViewController however, the 'viewDidLoad' method is the recommended place to configure your model:

```
@interface MyViewController : SCTableViewController

@end

@implementation MyViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    NSMutableArray *stringsArray = [NSMutableArray
        arrayWithObjects:@"String1", @"String2", @"String3", nil];
    SCArrayOfStringsSection *stringsSection =
        [SCArrayOfStringsSection sectionWithTitle:nil
            items:stringsArray];
    [self.tableViewModel addSection:stringsSection];
}

@end
```

Generally speaking, 'viewDidLoad' is the only method you need to implement when subclassing SCTableViewController.

SCViewController

As convenient as SCTableViewController may be, it is sometimes required to place more than one control on the view controller's view. Since SCTableViewController (as its UITableViewController parent) is configured to only have a single UITableView control as its view, it will not be suitable for this requirement.

SCViewController on the other hand, directly subclasses UIViewController, and thus fully supports as many controls as you wish on its main view (including more than one UITableView).

Unlike SCTableViewController where you often don't need a XIB file, with SCViewController it is usually very convenient to have one. Since SCViewController already provides an IBOutlet property called 'tableView', it's really simple to connect your XIB UITableView control to it. For a complete step by step example on how to do this, please refer to the section titled '*Table View Models*', under SCArrayOfObjectsModel.

Similar to an SCTableViewController, an SCViewController can either be directly instantiated or subclassed. However, since an SCViewController usually has many controls with potentially several IBOutlets, it is generally recommended to subclass it:

```
@interface MyViewController : SCViewController

@end

@implementation MyViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    NSMutableArray *stringsArray = [NSMutableArray
        arrayWithObjects:@"String1", @"String2", @"String3", nil];
    SCArrayOfStringsSection *stringsSection =
        [SCArrayOfStringsSection sectionWithTitle:nil
        items:stringsArray];
    [self.tableViewModel addSection:stringsSection];
}

@end
```

Generally speaking, 'viewDidLoad' is the only method you need to implement when subclassing SCViewController.

Actions

Actions is one of the highlights of the STV 3.0 release. It is one of those features that after you have used, you will wonder how you ever survived without!

As you’ve probably noticed throughout the rest of this book, actions enables you to execute custom code whenever a specific action occurs. While you can do the same job using delegates, actions are much simpler to use since they can be set to fire only for the specific element you’re interested in providing the custom behavior for.

STV 3.0 heavily relies on actions, and has defined several action classes that cover a wide range of the framework’s elements. The following is the list of available action classes:

ACTION CLASS	
SCCellActions	Actions relating to SCellTableViewCell and its subclasses
SCSectionActions	Actions relating to SCTableViewSection and its subclasses
SCModelActions	Actions relating to SCTableViewModel and its subclasses
SCTableViewControllerActions	Actions relating to SCTableViewController
SCViewControllerActions	Actions relating to SCViewController

We will now explore each action class in more detail.

SCCellActions

This is perhaps the most popular of all the action classes. Using SCCellActions, you are able to provide custom behavior in response to many of the cell's different events. For example, to provide a custom behavior when a cell is tapped, you simply set the 'didSelect' SCCellActions action:

```
myCell.cellActions.didSelect = ^(SCTableViewCell *cell, NSIndexPath *indexPath)
{
    NSLog(@"Cell at indexPath:%@ has been selected.", indexPath);
};
```

For a list of all the actions available for SCCellActions, please refer to SCCellAction's documentation (also available online at: <http://www.sensiblecocoa.com/documentation/STV30/Classes/SCCellActions.html>)

You will find that cell actions can be set at three different levels from within the STV framework: the model level, the section level, and finally the cell level. When you set an action at the model level, it fires for every single cell in the model. For example, the following code sets the background color of all cells in the model to yellow:

```
self.tableViewModel.cellActions.willDisplay = ^(SCTableViewCell *cell, NSIndexPath *indexPath)
{
    cell.backgroundColor = [UIColor yellowColor];
};
```

Similarly, setting a cell action at the section's level fires for every single cell in this section. It also overrides any identical action defined at the model's level. For example, while all the model cells background color is now yellow, the following code excludes the cells of 'mySection' and sets their background color to blue:

```
mySection.cellActions.willDisplay = ^(SCTableViewCell *cell, NSIndexPath *indexPath)
{
    cell.backgroundColor = [UIColor blueColor];
};
```

Finally, setting an action at the cell's level fires for only this specific cell. As you may have guessed, it will also override any identical action defined at either the section or the model levels. The following code sets the background color of the very first cell in 'mySection' to green:

```
SCTableViewCell *firstCell = [mySection cellAtIndex:0];
firstCell.cellActions.willDisplay = ^(SCTableViewCell *cell, NSIndexPath *indexPath)
{
    cell.backgroundColor = [UIColor greenColor];
};
```

One of the very convenient features of STV 3.0 is that actions at the cell level can also be set by the data definition that automatically generates the cells, way before the cell actually exists. For example, the following code sets the height of all the cells generated for taskDef to 60 points:

```
taskDef.cellActions.willConfigure = ^(STableViewCell *cell, NSIndexPath *indexPath)
{
    cell.height = 60;
};
```

You can also set an action for a specific generated cell at the property definition level. For instance, the following code sets the height of the description cell to 120 points:

```
SCPropertyDefinition *descPropertyDef = [taskDef
    propertyDefinitionWithName:@"description"];
descPropertyDef.cellActions.willConfigure = ^(STableViewCell *cell,
NSIndexPath *indexPath)
{
    cell.height = 60;
};
```

Once the respective cells are generated, all the actions defined in the data definitions will be copied to these cells. The only exception would be to the cells who have an identical action defined in their respective property definition, at which case the property definition action will override the global data definition action.

SCSectionActions

SCSectionActions provides actions for section related events. For example, the following action fires before the section's automatically generated detail view is presented:

```
objectsSection.sectionActions.detailModelWillPresent = ^(STableViewSection *section, STableViewCell *detailModel, NSIndexPath *indexPath)
{
    // Set a custom title for the generated detail view
    detailModel.viewController.title = @"My custom title";
};
```

For a list of all the actions available for SCSectionActions, please refer to SCSectionAction's documentation (also available online at: <http://www.sensiblecocoa.com/documentation/STV30/Classes/SCSectionActions.html>)

Section actions can be set at two levels: the model level and the section level. Setting the action at the model level fires for every single section in the model. Setting it at the section level fires only for this specific section. Similarly to cell actions, setting an action at the section level overrides any identical action set at the model level.

SCModelActions

SCModelActions provides actions for model related events. For example, the following action fires after `SCArrayOfObjectsModel` has fetched all its objects from their data store:

```
objectsModel.modelActions.didFetchItemsFromStore = ^(SCArrayOfItems-
Model *itemsModel, NSMutableArray *items)
{
    // Add a button cell at the end of the fetched items list
    SCTableViewCell *buttonCell = [SCTableViewCell
cellWithText:@"Tap me!" textAlign:NSTextAlignmentCenter];
    buttonCell.cellActions.didSelect = ^(SCTableViewCell *cell,
NSIndexPath *indexPath)
    {
        NSLog(@"buttonCell tapped!");
    };

    [items addObject:buttonCell];
};
```

For a list of all the actions available for SCModelActions, please refer to SCModelAction's documentation (also available online at: <http://www.sensiblecocoa.com/documentation/STV30/Classes/SCModelActions.html>)

Model actions can only be set at a single level, the model level.

SCTableViewControllerActions

SCTableViewControllerActions provides actions for SCTableViewController related events. For example, the following action fires right before the view controller is dismissed:

```
viewController.actions.willDismiss = ^(SCTableViewController *viewCon-
troller)
{
    NSLog(@"viewController is about to be dismissed.");
};
```

For a list of all the actions available for SCTableViewControllerActions, please refer to SCTableViewControllerAction's documentation (also available online at: <http://www.sensiblecocoa.com/documentation/STV30/Classes/SCTableViewControllerActions.html>)

View controller actions can only be set at a single level, the view controller level.

SCViewControllerActions

SCViewControllerActions provides actions for SCViewController related events. It's almost identical to SCTableViewControllerActions, except that the action parameters declare an SCViewController instead of an SCTableViewController. For example, the same action that fires right before the view controller is dismissed is declared as follows:

```
viewController.actions.willDismiss = ^(SCViewController *viewController)
{
    NSLog(@"viewController is about to be dismissed.");
};
```

For a list of all the actions available for SCViewControllerActions, please refer to SCViewControllerAction's documentation (also available online at:

<http://www.sensiblecocoa.com/documentation/STV30/Classes/SCViewControllerActions.html>)

View controller actions can only be set at a single level, the view controller level.

Themes

Themes is an amazing new feature of STV 3.0 that automatically styles your entire application using CSS like theme files. This enables you to completely change the look and feel of your application without touching the code at all!

Exploring SCTheme

SCTheme is the class responsible for all the framework's theme styling magic. Once SCTheme is initialized from a theme file and assigned to a table view model, everything associated with that model is automatically styled, including all detail models and view controllers. It is worth noting here that STV's theme styling is not restricted to the framework's own elements, but can also style any kind of UI element in your entire application!

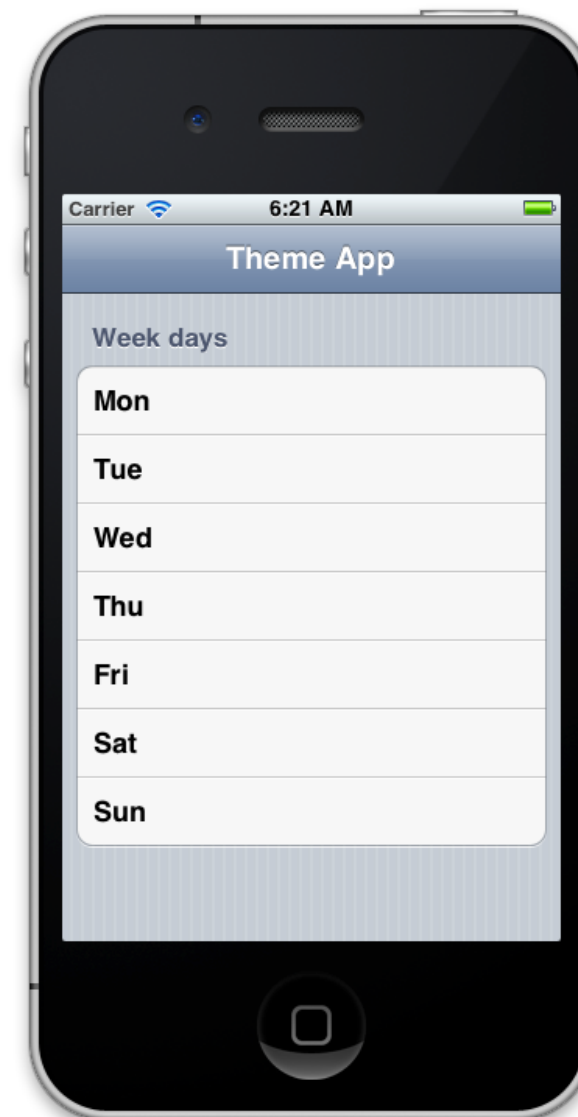
To explore themes, let's create a simple application with a theme file and see how you're able to instantly style it by changing the contents of this file. First, start by creating a new project based on the template we created in the *'Getting Started'* chapter. Once you do that, add the following code to RootViewController.m:

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

    self.title = @"Theme App";

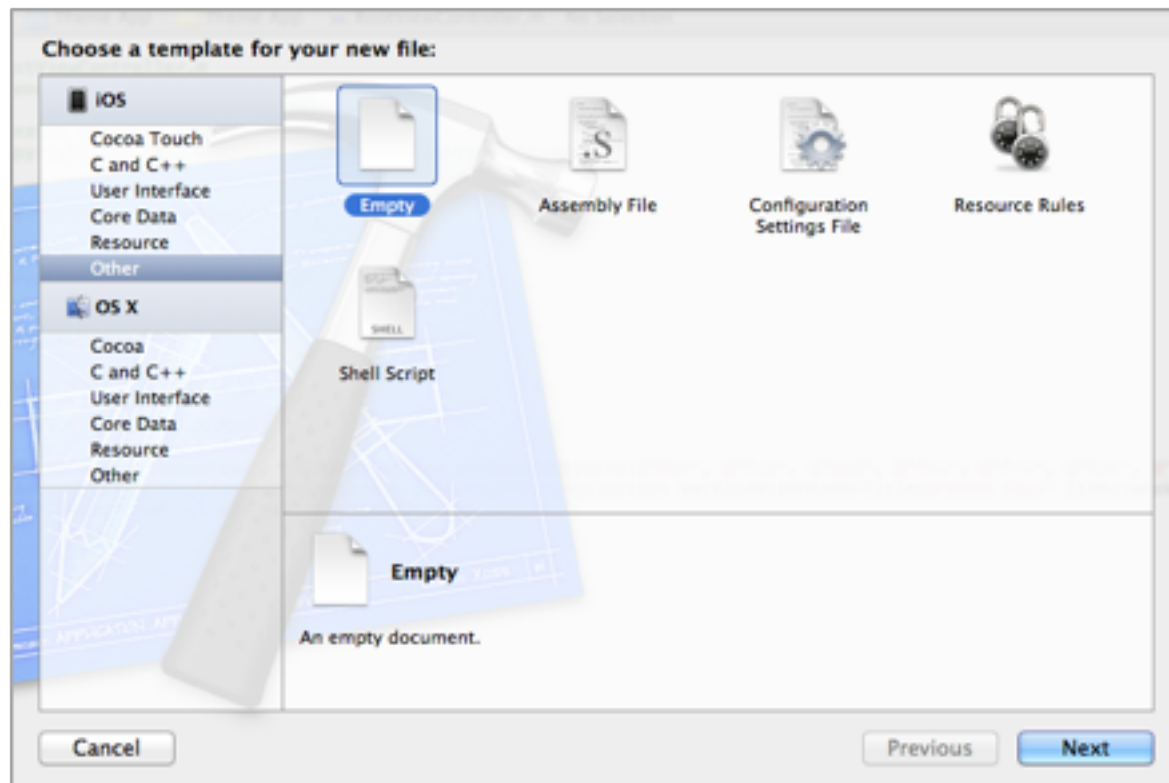
    NSMutableArray *weekDays = [NSMutableArray
arrayWithObjects:@"Mon", @"Tue", @"Wed", @"Thu", @"Fri", @"Sat",
@"Sun", nil];
    SCArrayOfStringsSection *daysSection = [SCArrayOfStringsSection
sectionWithTitle:@"Week days" items:weekDays];
    [self.tableViewModel addSection:daysSection];
}
```

Now run the app and you should get something like this:

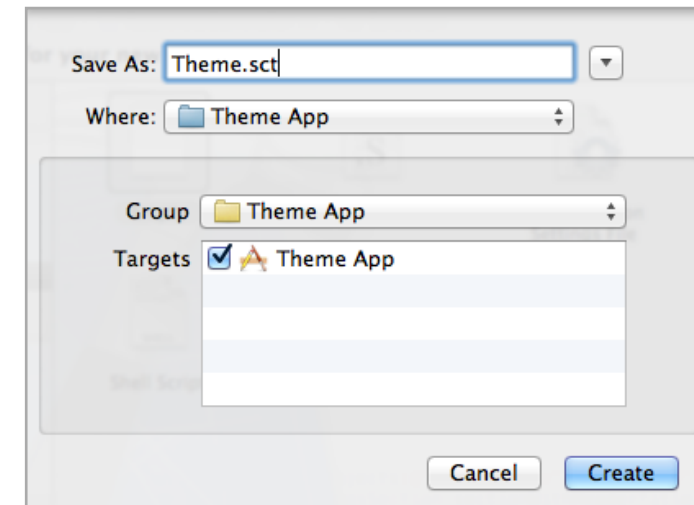


Now let's create the theme file by following the next steps:

1. From Xcode, navigate to File->New->File... and select the 'Empty' document template under the 'Other' iOS templates group, then click next.



2. Name the file Theme.sct then click on the *Create* button (you can name the file anything you want, but make sure to use the same name later on when we initialize SCTheme).



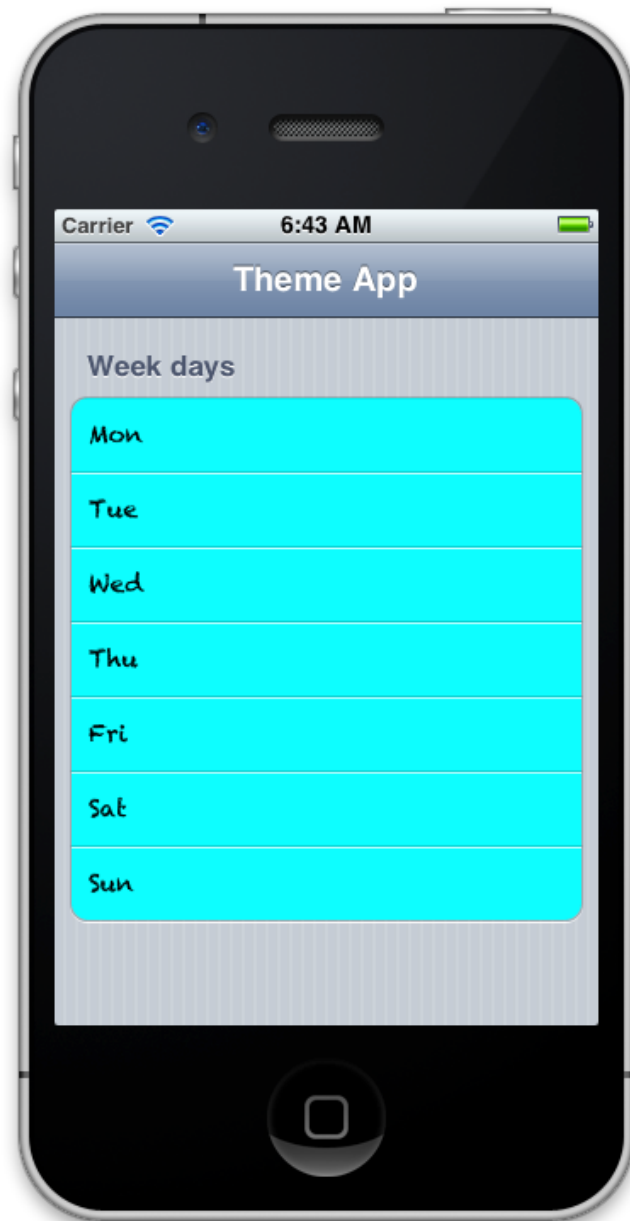
3. Once the file is created, insert the following text into it:

```
SCTableViewCell
{
    backgroundColor: cyanColor;
    titleLabel.font: Chalkduster 14;
}
```

4. Finally, initialize and add an SCTheme instance to your model by adding the following line to viewDidLoad:

```
self.tableViewModel.theme = [SCTheme themeWithPath:@"Theme.sct"];
```

Now running the app again, you should find that it got styled as follows:



That was easy! However, let's try and understand what just happened.

First, we started by adding what we call a *theme style* to the Theme.sct file. A theme style is very similar in syntax to how CSS classes are defined, having a *style name* that is super-

seded by brackets containing several style attributes. The style name can be anything you want, but choosing the name of an existing class automatically styles all elements of this class. For example, choosing the name of our theme style as 'SCTableViewController' automatically styled all elements of type SCTableViewController in our application.

Next, we placed several attributes for the SCTableViewController style, each superseded by a colon and an attribute value:

```
SCTableViewController
{
    backgroundColor: cyanColor;
   .textLabel.font: Chalkduster 14;
}
```

Notice that each attribute correspond to an existing property of the theme style (SCTableViewController in this case). You can actually style any property you want provided that its type is supported by SCTheme (we will discuss all the supported types in the next section). Even key-path properties are fully supported! For example, we were able to set 'textLabel.font', which is not a direct property of SCTableViewController.

So that's about it! As you can see, this makes styling your applications extremely trivial. For example, now lets try and style the table view itself. Since it's of type UITableView, just create a theme style with the same name and it should get automatically styled. Similarly, we'll do the same thing for the UINavigationController control.

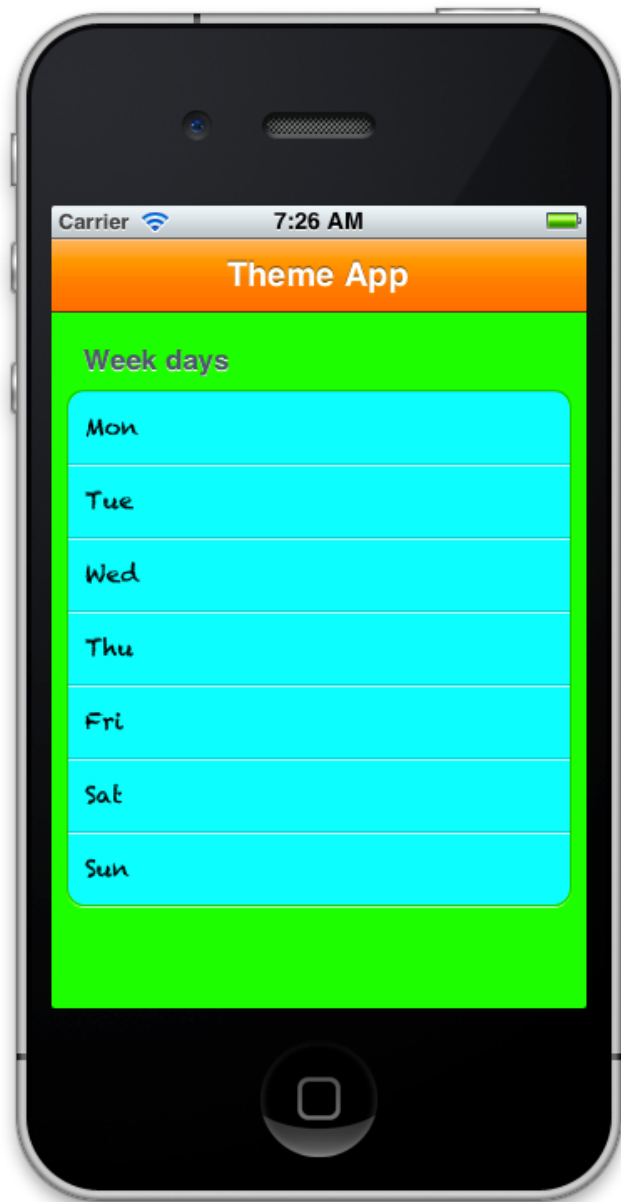

```

UINavigationController
{
    tint_color: orange_color;
}

UITableView
{
    background_color: green_color;
}

```

Running the app now gives us the following:



As you recall, we earlier said you could name the theme style any name you wish. Doing so enables us to select specific UI elements and apply the theme style to, instead of having it applied automatically. For example, let's create a theme style called 'RedCell' as follows:

```

RedCell
{
    background_color: red_color;
    text_label.font: Chalkduster 14;
}

```

Now let's assign the theme style of the second cell in the stringsSection to this style. Setting theme styles is usually done in the 'willStyle' cell action:

```

- (void)viewDidLoad
{
    [super viewDidLoad];

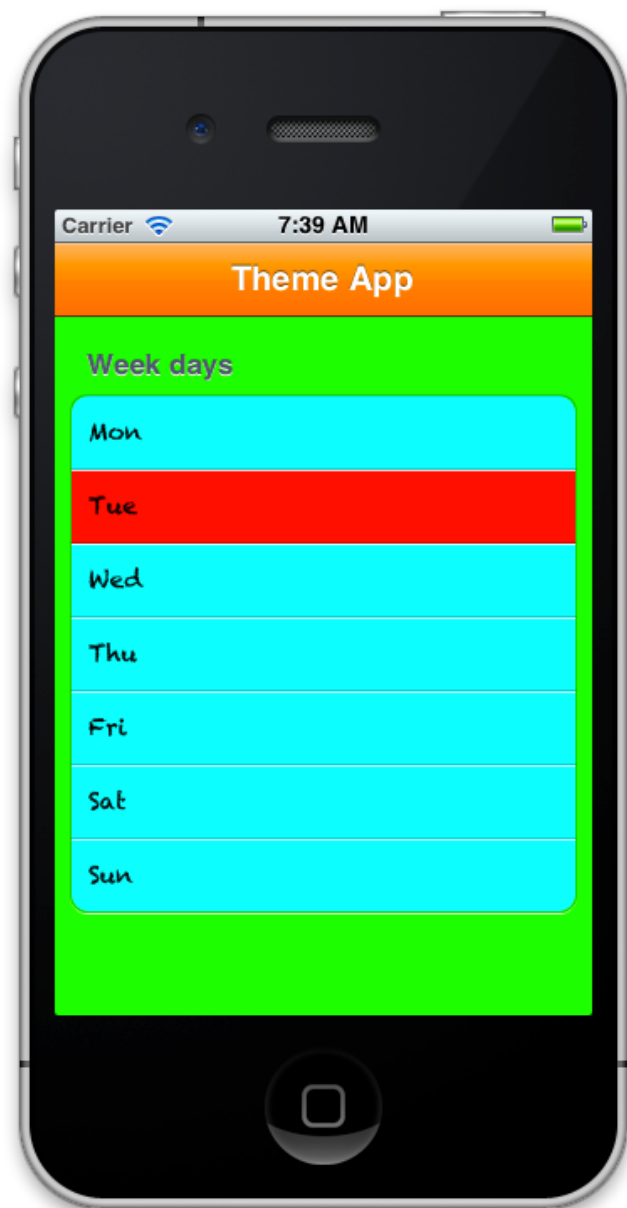
    self.title = @"Theme App";

    self.tableViewModel.theme = [SCTheme themeWithPath:@"Theme.sct"];

    NSMutableArray *weekDays = [NSMutableArray
    arrayWithObjects:@"Mon", @"Tue", @"Wed", @"Thu", @"Fri", @"Sat",
    @"Sun", nil];
    SCArrayOfStringsSection *daysSection = [SCArrayOfStringsSection
    sectionWithTitle:@"Week days" items:weekDays];
    daysSection.cellActions.willStyle = ^(SCTableViewCell *cell, NSIndexPath *indexPath)
    {
        if(indexPath.row == 1) // 2nd cell
            cell.themeStyle = @"RedCell";
    };
    [self.tableViewModel addSection:daysSection];
}

```

And this is what we get now:



Taking all this a step further, we can start assigning alternating cell styles, which is a very common theme in many applications. Instead of doing this via the 'willStyle' cell action this time however, we'll be doing it entirely from the theme file by setting the `SCTableViewSection` properties called 'odd-

`CellThemeStyle`' and 'evenCellThemeStyle'. Here is how the full `Theme.sct` file looks like now:

```
UINavigationController
{
  tint_color: orange_color;
}

UITableView
{
  background_color: green_color;
}

SCTableViewSection
{
  odd_cells_theme_style: CyanCell;
  even_cells_theme_style: OrangeCell;
}

CyanCell
{
  background_color: cyan_color;
  text_label.font: Chalkduster 14;
}

OrangeCell
{
  background_color: orange_color;
  text_label.font: Chalkduster 14;
}
```

And this is the updated code where no actions are needed anymore:

```

- (void)viewDidLoad
{
    [super viewDidLoad];

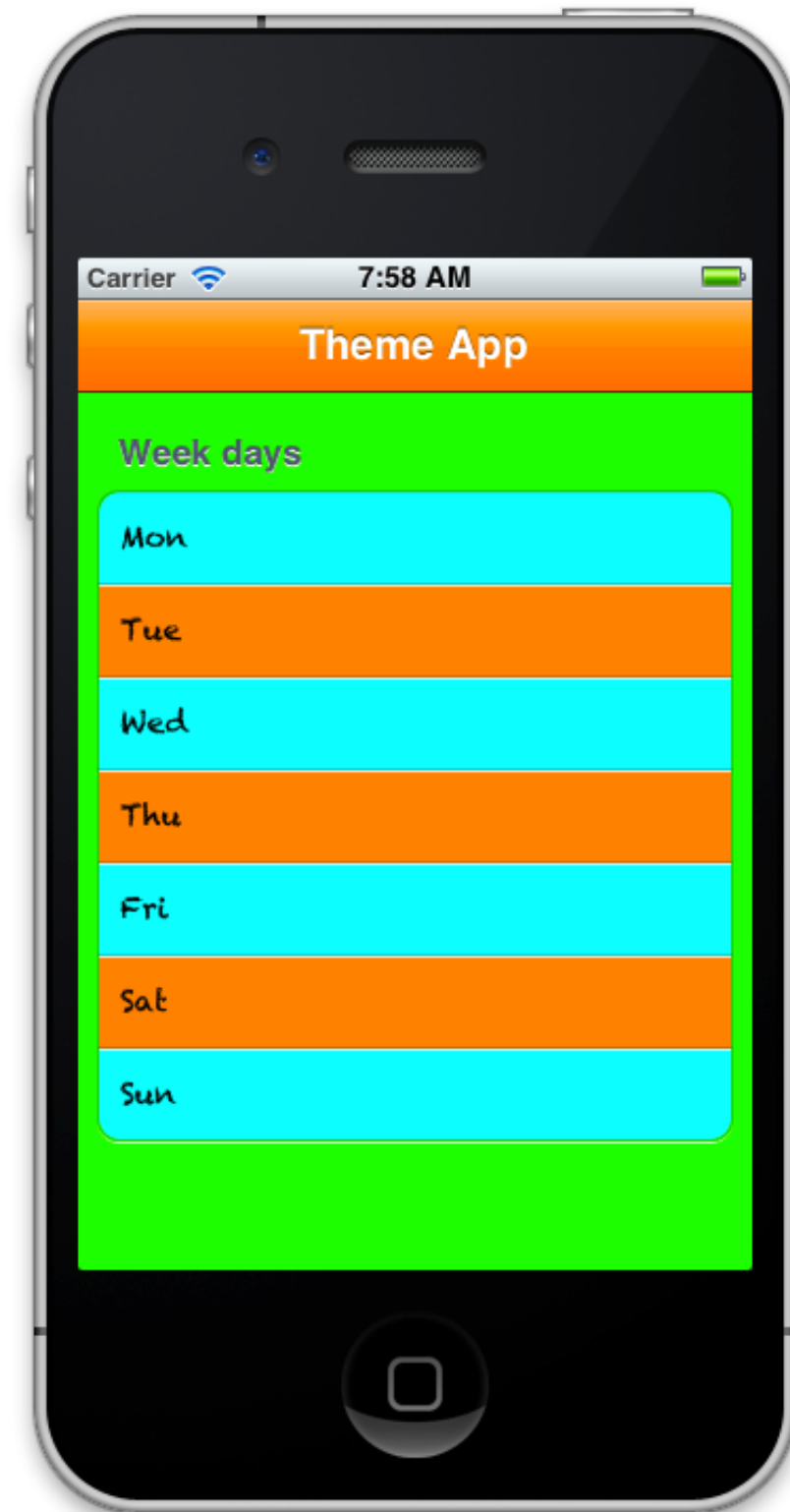
    self.title = @"Theme App";

    self.tableViewModel.theme = [SCTheme themeWithPath:@"Theme.sct"];

    NSMutableArray *weekDays = [NSMutableArray
arrayWithObjects:@"Mon", @"Tue", @"Wed", @"Thu", @"Fri", @"Sat",
@"Sun", nil];
    SCArrayOfStringsSection *daysSection = [SCArrayOfStringsSection
sectionWithTitle:@"Week days" items:weekDays];
    [self.tableViewModel addSection:daysSection];
}

```

Running now gives us this beautiful table view:



SCTheme file structure

As we've seen in the previous section, an SCTheme file is simply a collection of theme styles. The syntax of a theme style is as follows:

```
/* comment */
ThemeStyleName
{
    // comment
    attribute1: value1;
    attribute2: value2;
    ...
}
```

Once the theme style is defined, it can be applied to any UI element in your application. Furthermore, if the name of the theme style corresponds to an existing class name, all the elements of this class will be automatically styled.

There are two restrictions however to styling UI elements:

- a. The attribute specified in the theme style must actually exist in the UI element. This is logical enough, as you cannot set a non-existing attribute.
- b. The attribute's type must be a type supported by SCTheme. The following is a list of all the supported attribute types:
 - NSString
 - CGFloat
 - BOOL
 - UIColor
 - CGColorRef
 - UIImage
 - UIFont
 - UIView
 - UITableViewCellSeparatorStyle

The value of each of the above types must be provided in one or more standard formats. The following sections will discuss how the values should be formatted to satisfy each type's syntax.

NSString type

The value for an NSString attribute is provided as a simple string between one or two quotes.

Example:

```
detailTextLabel.text: "Hello World!";
```

CGFloat type

The value for a CGFloat attribute is provided as a regular number and can include decimal places.

Example:

```
height: 60;
```

BOOL type

The value for a BOOL attribute is provided as any of these constants: TRUE, FALSE, YES, NO (all case insensitive).

Example:

```
clipsToBounds: NO;
```

UIColor type

The value for a UIColor attribute can be provided in any of the following formats:

- Any UIColor color name constructor.

```
backgroundColor: blueColor;
```

- rgb(redValue, greenValue, blueValue, optionalAlphaValue)

```
backgroundColor: rgb(100, 0, 255);
```

- #hexValue

```
backgroundColor: #CC33FF
```

- A string containing an image resource.

```
backgroundColor: "background.png";
```

CGColorRef type

Format is identical to that of the UIColor type.

```
layer.borderColor: redColor;
```

UIImage type

The value for a UIImage attribute can be provided in any of the following formats:

- A string containing the image resource.

```
backgroundImage: "background.png"
```

- A string containing the image resource and capInsets(top, left, bottom, right).

```
backgroundImage: "background.png" capInsets(0,0,0,0)
```

UIFont type

The value for a UIFont attribute is provided as the font name and the font size separated by a space.

```
textLabel.font: Courier-Bold 12;
```

For a good resource of iOS font names, visit: <http://iosfonts.com>

UIView type

The value for a UIView attribute is provided as a string containing an image resource that will later be loaded into a UIImageView.

```
backgroundView: "background.png";
```

UITableViewCellSeparatorStyle type

The value for a UITableViewCellSeparatorStyle attribute is provided as any valid UITableViewCellSeparatorStyle constant.

```
separatorStyle: UITableViewCellSeparatorStyleNone;
```

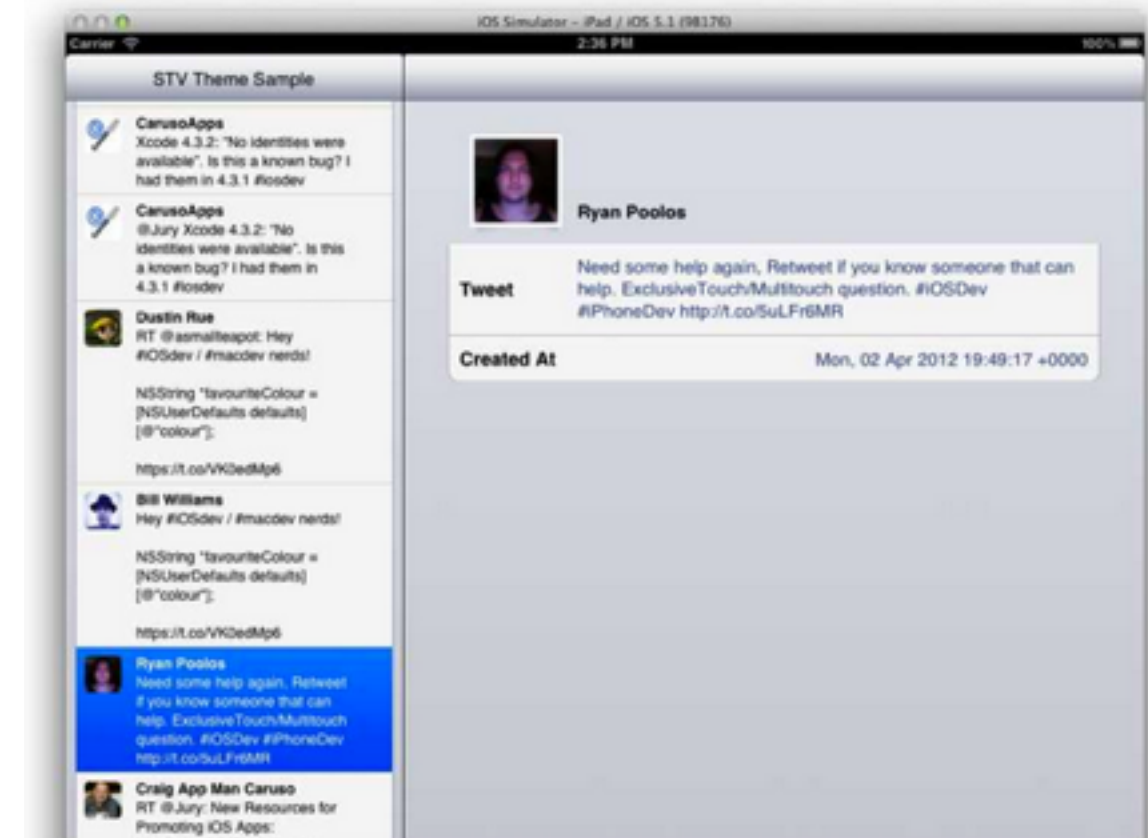

Third party themes

One of the most fascinating things about STV's theme files is that you don't even need to be the one who developed them! What this means is that third parties could start developing their own STV theme files, and using a single line of code, you could have your application fully styled professionally.

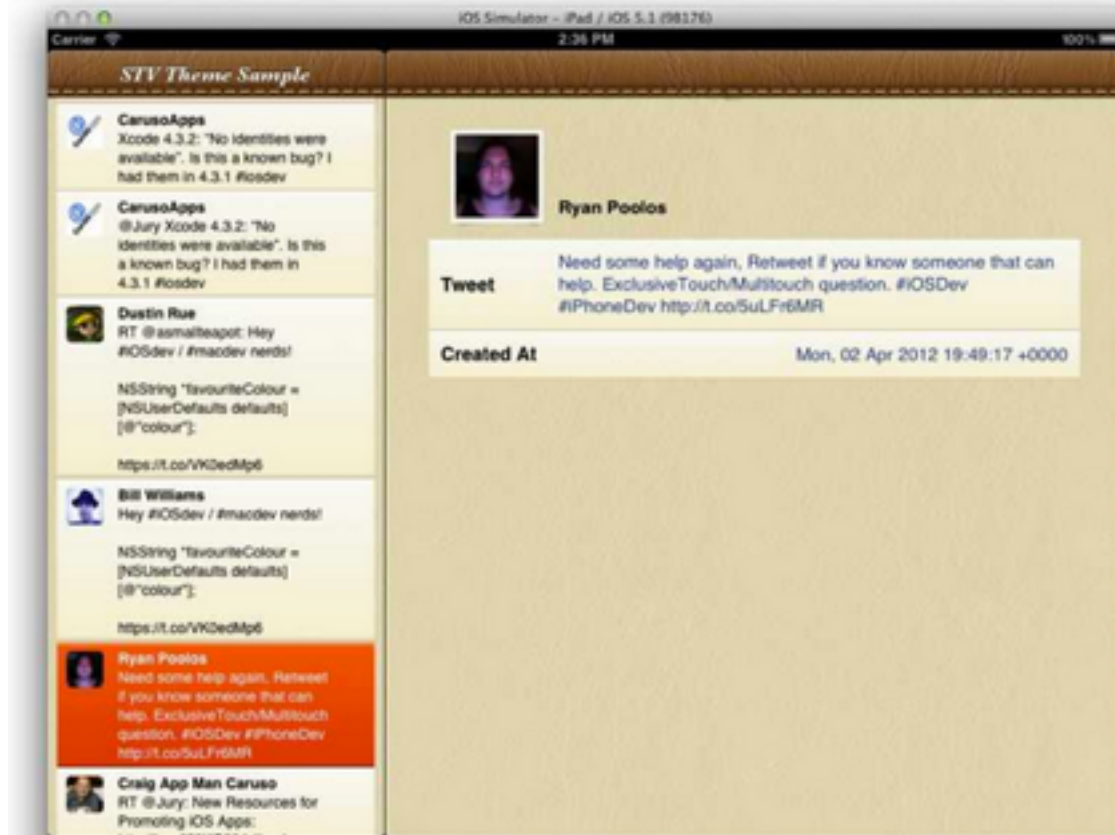
One company that started doing that is called AppDesignVault.com (<http://www.appdesignvault.com>). For each of their standard iOS themes, they've created a .sct theme file, ready to be passed to SCTheme and have your application immediately styled.

AppDesignVault were kind enough to provide all STV Pro customers with three full complementary themes, and all STV Std customers with two full complementary themes. For more information on how to download your complementary themes, please refer to the 'Themes' folder inside your purchased STV package.

To give you an idea of the power of these theme files, here is our twitter sample application before applying the theme:



And here it is after applying AppDesignVault's 'Foody Theme' file:



Feel free to play around with the other complementary theme files and see for yourself how each completely transforms your application, all with only a single line of code!

Extending STV

Work in progress.

Appendix A

Migrating your projects to STV 3.0

Up to STV 3.0, all previous STV releases have been fully backward compatible all the way to our very first version. For reasons discussed in ‘*Chapter 1: What’s new in STV 3.0*’, we took the decision to take a completely fresh approach to STV’s internal architecture. For someone who’s looking forward to migrate their project to STV 3.0, this might seem very worrying.

Fortunately, even though STV’s architecture got a complete make over, its developer interface remained more or less the same, and should still be very familiar to anyone who has used any previous STV version. Having said that, your previous STV projects still do need to get migrated to STV 3.0, and will not work out of the box.

To help you get started, we wrote this appendix to serve as a detailed migration guide. In addition, we also migrated all the old STV samples and bundled them with the new STV package (we left the samples as non-ARC projects to demonstrate how STV can seamlessly work with non-ARC applications).

A general overview of what has changed

1. In general, tasks are now easier to achieve. This means that you’ll be removing several pieces of unnecessary code that you usually had to include in your older STV projects.
2. The key-binding method have been completely replaced with the much more straight forward and robust *Dictionary Binding* method. You can see a full example of this in our migrated “Overview App”.
3. Since STV’s new architecture adopts the new concept of a generic *Data Definition* (as opposed to the older *Class Definition*), most methods that have the word “classDefinition” have been renamed to just “definition”. In addition, several method names have changed to become more consistent with the iOS framework nomenclature (e.g. removing all unnecessary ‘with’ words). To give you an example, the `SCArrayOfObjectsSection` method called :

```
sectionWithHeaderTitle:withItems:withClassDefintion:
```

has been renamed to:

```
sectionWithHeaderTitle:items:itemsDefintion:
```

4. The `SCTableViewCellDelegate` protocol has been completely replaced by STV 3.0’s new *Actions* feature.

General project migration steps

- Completely remove any old STV files.
- Completely remove any import statements that import STV header files (e.g. `#import "SCTableViewModel.h"` statements in view controllers).
- Add STV 3.0 to your project, either as a static framework or in source code format. For a detailed discussion of this, please refer to *'Chapter 1: Setting up STV'*.
- `SCTableViewModel` now doesn't take the view controller as a parameter during initialization.

```
tableModel = [[SCTableViewModel alloc]
initWithTableView:self.tableView withViewController:self];
```

becomes

```
tableModel = [[SCTableViewModel alloc]
initWithTableView:self.tableView];
```

- Remove any extra 'with' from method names (as discussed earlier in the general overview). Similarly, refactor any methods with 'classDefinition' in their name to become only 'definition'. If you're not sure what methods to change, just try compiling the project and Xcode should point them out.
- Rename any 'detailViewWillAppear' methods to 'detailViewWillAppear'. The name change was significant since the method gets called only the first time the detail view is pre-

sented. Similarly, and 'detailViewWillDisappear' methods should be renamed to 'detailViewWillDismiss'.

- All custom cells now should descend from `SCCustomCell` instead of `SCControlCell`. Descending your cells from `SCControlCell` is still technically correct since `SCControlCell` is itself a subclass of `SCCustomCell`. You should only do that however when you have a valid reason for using `SCControlCell`.
- Core Data is now a separate STV framework extension (called `STV+CoreData`), and should be added very similarly to how the main STV framework is added. Furthermore, you should now be using the new '`SCEntityDefinition`' instead of the older `SCClassDefinition` class to define your Core Data entities. Referring to our migrated STV 2.0 Core Data sample should give you a very good starting point.
- When developing for the iPad, a master-detail relationship should now be established by simply assigning the detail view to the 'detailViewController' property of the master model. You'll find a thorough example of this in the migrated STV 2.0 iPad app.
- Finally, STV now has a console warning system, and will give you warnings regarding any STV syntax inconsistencies that were not detected by the compiler. To view any potential STV warnings, please make sure you have Xcode console window open while running your app.

Further recommendations

- Consider replacing all your `UITableViewController` superclasses with `SCTableViewController`. While not being strictly required to use STV, doing this change has many advantages and will keep your code very simple (as discussed earlier in the book). If you decide to do so, it should as simple renaming `UITableViewController` to `SCTableViewController` in your view controller's header file. You should also remove any `SCTableViewModel` ivars that you're added manually to your view controller and use `'self.tableViewModel'` instead. Similarly, consider replacing `UIViewController` with `SCViewController`.
- Whenever possible, consider using STV's new Actions features instead of your implemented `SCTableViewModelDelegates`. Your existing delegates should still work fine, but actions are much simpler and a lot easier to maintain.