Salesforce.com: Winter '14

Creating a Native iOS Warehouse Application

Last updated: December 13, 2013

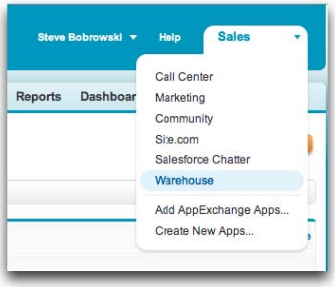
Table of Contents

Tutorial: Creating a Native iOS Warehouse App

Prerequisites

- This tutorial uses a Warehouse app that contains a basic inventory database. You'll need to install this app in a DE org. If you install it in an existing DE org, be sure to delete any existing Warehouse components you've made before you install.

  1. Click the installation URL link: https://login.salesforce.com/packaging/installPackage.apexp?p0=04ti0000000MMMT
  2. If you aren't logged in already, enter the username and password of your DE org.
  3. On the Package Installation Details page, click **Continue**.
  4. Click **Next**, and on the Security Level page click **Next**.
  5. Click **Install**.
  6. Click **Deploy Now** and then **Deploy**.
  7. Once the installation completes, you can select the **Warehouse** app from the app picker in the upper right corner.



  8. To create data, click the **Data** tab.
  9. Click the **Create Data** button.

- Install the latest versions of Xcode and the iOS SDK.
- Install the Salesforce Mobile SDK using npm:

  1. If you've already successfully installed Node.js and npm, skip to step 4.
  2. Install Node.js on your system. The Node.js installer automatically installs npm.

     i. Download Node.js from www.nodejs.org/download.

     ii. Run the downloaded installer to install Node.js and npm. Accept all prompts asking for permission to install.

  3. At the Terminal window, type `npm` and press `Return` to make sure your installation was successful. If you don't see a page of usage information, revisit Step 2 to find out what's missing.
  4. At the Terminal window, type `sudo npm install forceios -g`

     This command uses the forceios package to install the Mobile SDK globally. With the `-g` option, you can run `npm install` from any directory. The npm utility installs the package under `/usr/local/lib/node_modules`, and links binary modules in `/usr/local/bin`. Most users need the `sudo` option because they lack read-write permissions in `/usr/local`.

Create a Native iOS App

In this tutorial, you learn how to get started with the Salesforce Mobile SDK, including how to install the SDK and a quick tour of the native project template using your DE org. Subsequent tutorials show you how to modify the template app and make it work with the Warehouse schema.

Step 1: Create a Connected App

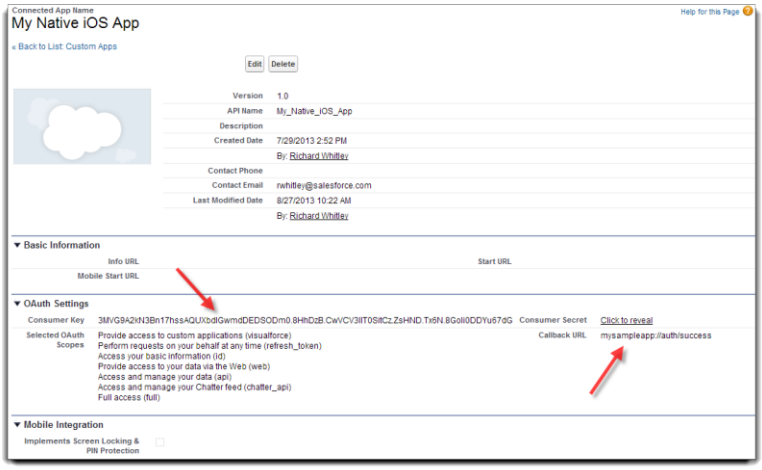
Step 1: Create a Connected App

In this step, you learn how to configure a Connected App in Force.com. Doing so authorizes the mobile app you will soon build to communicate securely with Force.com and access Force.com APIs on behalf of users via the industry-standard OAuth 2.0 protocol.

1. In your DE org, click ***Your Name*** > **Setup** and under App Setup, click **Create** > **Apps**.
2. Under **Connected Apps**, click **New** to bring up the **New Connected App** page.
3. Under **Basic Information**, fill out the form as follows:
   - **Connected App Name**: `My Native iOS App`
   - **API Name**: accept the suggested value
   - **Contact Email**: enter your email address
4. Under OAuth Settings, check the **Enable OAuth Settings** checkbox.
5. Set **Callback URL** to `mysampleapp://auth/success`.
6. Under **Available OAuth Scopes**, check "Access and manage your data (api)" and "Perform requests on your behalf at any time (refresh_token)", then click **Add**.
7. Click **Save**.

After you save the configuration, notice the details of the Connected App you just created.

- Note the Callback URL and Consumer Key; you will use these when you set up your native app in the next step.
- Mobile apps do not use the Consumer Secret, so you can ignore this value.



Step 2: Create a Native iOS Project

To create a new Mobile SDK project, use the forceios utility again in the Terminal window.

1. Change to the directory in which you want to create your project.
2. To create an iOS project, type `forceios create`.

   The forceios utility prompts you for each configuration value.

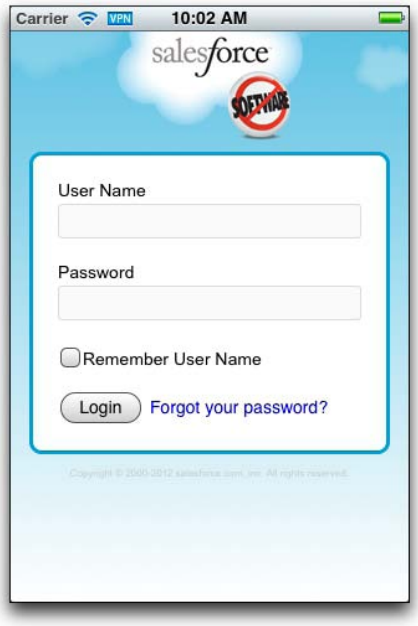
3. For application type, enter `native`.
4. For application name, enter `MyNativeiOSApp`.
5. For company identifier, enter `com.acme.goodapps`.
6. For organization name, enter `GoodApps, Inc.`.
7. For output directory, enter `tutorial/iOSNative`.
8. For the Connected App ID, copy and paste the Consumer Key from your Connected App definition.
9. For the Connected App Callback URI, copy and paste the Callback URL from your Connected App definition.

The input screen should look similar to this:

```
rwhitley-ltm1:Downloads rwhitley$ forceios create
Enter your application type (native, hybrid_remote, or hybrid_local): native
Enter your application name: MyNativeiOSApp
Enter your company identifier (com.mycompany): com.acme.goodapps
Enter your organization name (Acme, Inc.): GoodApps, Inc.
Enter the output directory for your app (defaults to the current directory):
Enter your Connected App ID (defaults to the sample app's ID):
Enter your Connected App Callback URI (defaults to the sample app's URI):
Creating app in /Users/rwhitley/Downloads/MyNativeiOSApp
Successfully created native app 'MyNativeiOSApp'.
```

Step 3: Run the New iOS App

1. In Xcode, select **File** > **Open**.
2. Navigate to the output folder you specified.
3. Open your app's `xcodeproj` file.
4. Click the **Run** button in the upper left corner to see your new app in the iOS simulator. Make sure that you've selected **Product** > **Destination** > **iPhone 6.0 Simulator** in the Xcode menu.
5. When you start the app, after showing an initial splash screen, you should see the Salesforce login screen. Login with your DE username and password.



6. When prompted, click **Allow** to let the app access your data in Salesforce. You should see a table listing the names of users defined in your DE org.

Step 4: Explore How the iOS App Works

The native iOS app uses a straightforward Model View Controller (MVC) architecture.

- The model is the Force.com database schema
- The views come from the nib and implementation files in your project
- The controller functionality represents a joint effort between the iOS SDK classes, the Salesforce Mobile SDK, and your app

AppDelegate Class and the Root View Controller

When the app is launched, the `AppDelegate` class initially controls the execution flow. After the login process completes, the `AppDelegate` instance passes control to the root view. In the template app, the root view controller class is named `RootViewController`. This class becomes the root view for the app in the `AppDelegate.m` file, where it's subsumed by a `UINavigationController` instance that controls navigation between views:

```
- (void)setupRootViewController
{
    RootViewController *rootVC = [[RootViewController alloc] initWithNibName:nil bundle:nil];

    UINavigationController *navVC = [[UINavigationController alloc]
initWithRootViewController:rootVC];
    self.window.rootViewController = navVC;
}
```

Before it's customized, though, the app doesn't include other views or touch event handlers. It simply logs into Salesforce, issues a request using Salesforce Mobile SDK REST APIs, and displays the response in the root view.

UITableViewController Class

`RootViewController` inherits the `UITableViewController` class. Because it doesn't customize the table in its inherited view, there's no need for a nib or xib file. The controller class simply loads data into the `tableView` property and lets the super class handle most of the display tasks. However, `RootViewController` does add some basic cell formatting by calling the `tableView:cellForRowAtIndexPath:` method. It creates a new cell, assigns it a generic ID (`@"CellIdentifier"`), puts an icon at the left side of the cell, and adds an arrow to the right side. Most importantly, it sets the cell's label to assume the Name value of the current row from the REST response object. Here's the code:

```
// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView_ cellForRowAtIndexPath:(NSIndexPath
 *)indexPath {
   static NSString *CellIdentifier = @"CellIdentifier";

   // Dequeue or create a cell of the appropriate type.
    UITableViewCell *cell = [tableView_ dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleValue1
reuseIdentifier:CellIdentifier] autorelease];

    }
 //if you want to add an image to your cell, here's how
 UIImage *image = [UIImage imageNamed:@"icon.png"];
 cell.imageView.image = image;

 // Configure the cell to show the data.
 NSDictionary *obj = [self.dataRows objectAtIndex:indexPath.row];
 cell.textLabel.text =  [obj objectForKey:@"Name"];

 //this adds the arrow to the right hand side.
 cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;

 return cell;

}
```

SFRestAPI Shared Object and SFRestRequest Class

You can learn how the app creates and sends the REST request by browsing the `RootViewController.viewDidLoad` method. The app defines a literal SOQL query string and passes it to the `SFRestAPI:requestForQuery:` instance method. To call this method, the app sends a message to the shared singleton `SFRestAPI` instance. The method creates and returns an appropriate, pre-formatted `SFRestRequest` object that wraps the SOQL query. The app then forwards this object to the server by sending the `send:delegate:` message to the shared `SFRestAPI` object:

```
    //Here we use a query that should work on either Force.com or Database.com
   SFRestRequest *request = [[SFRestAPI sharedInstance] requestForQuery:@"SELECT Name FROM
 User LIMIT 10"];
    [[SFRestAPI sharedInstance] send:request delegate:self];
```

The `SFRestAPI` class serves as a factory for `SFRestRequest` instances. It defines a series of request methods that you can call to easily create request objects. If you want, you can also build `SFRestRequest` instances directly, but, for most cases, manual construction isn't necessary.

Notice that the app specifies `self` for the `delegate` argument. This tells the server to send the response to a delegate method implemented in the `RootViewController` class.

SFRestDelegate Interface

To be able to accept REST responses, `RootViewController` implements the `SFRestDelegate` interface. This interface declares four methods—one for each possible response type. The `request:didLoadResponse:` delegate method executes when the request succeeds. When `RootViewController` receives a `request:didLoadResponse:` callback, it copies the returned records into its data rows and reloads the data displayed in the Warehouse view. Here's the code that implements the `SFRestDelegate` interface in the `RootViewController` class:

```
#pragma mark - SFRestAPIDelegate

- (void)request:(SFRestRequest *)request didLoadResponse:(id)jsonResponse {
    NSArray *records = [jsonResponse objectForKey:@"records"];
    NSLog(@"request:didLoadResponse: #records: %d", records.count);
    self.dataRows = records;
    [self.tableView reloadData];
}


- (void)request:(SFRestRequest*)request didFailLoadWithError:(NSError*)error {
    NSLog(@"request:didFailLoadWithError: %@", error);
    //add your failed error handling here
}

- (void)requestDidCancelLoad:(SFRestRequest *)request {
    NSLog(@"requestDidCancelLoad: %@", request);
    //add your failed error handling here
}

- (void)requestDidTimeout:(SFRestRequest *)request {
    NSLog(@"requestDidTimeout: %@", request);
    //add your failed error handling here
}
```

As the comments indicate, this code fully implements only the `request:didLoadResponse:` success delegate method. For responses other than success, this template app simply logs a message.

Customize the List Screen

In this tutorial, you modify the root view controller to make the app specific to the Warehouse schema. You also adapt the existing SOQL query to obtain all the information we need from the Merchandise custom object.

Step 1: Modify the Root View Controller

To adapt the template project to our Warehouse design, let's rename the `RootViewController` class.

1. In the Project Navigator, choose the `RootViewController.h` file.
2. In the Editor, click the name "RootViewController" on this line:

```
@interface RootViewController : UITableViewController <SFRestDelegate>{
```

3. Using the Control-Click menu, choose **Refactor** > **Rename**. Be sure that **Rename Related Files** is checked.
4. Change "RootViewController" to "WarehouseViewController". Click **Preview**.

Xcode presents a new window that lists all project files that contain the name "RootViewController" on the left. The central pane shows a diff between the existing version and the proposed new version of each changed file.

5. Click **Save**.
6. Click **Enable** when Xcode asks you if you'd like it to take automatic snapshots before refactoring.

After the snapshot is complete, the Refactoring window goes away, and you're back in your refactored project. Notice that the file names `RootViewController.h` and `RootViewController.m` are now `WarehouseViewController.h` and `WarehouseViewController.m`. Every instance of `RootViewController` in your project code has also been changed to `WarehouseViewController`.

Step 2: Create the App's Root View

The native iOS template app creates a SOQL query that extracts Name fields from the standard User object. For this tutorial, though, you use records from a custom object. Later, you create a detail screen that displays Name, Quantity, and Price fields. You also need the record ID.

Let's update the SOQL query to operate on the custom Merchandise__c object and to retrieve the fields needed by the detail screen.

1. In the Project Navigator, select `WarehouseViewController.m`.
2. Scroll to the `viewDidLoad` method.
3. Update the view's display name to "Warehouse App". Change:

```
self.title = @"Mobile SDK Sample App"
```

to

```
self.title = @"Warehouse App"
```

4. Change the SOQL query in the following line:

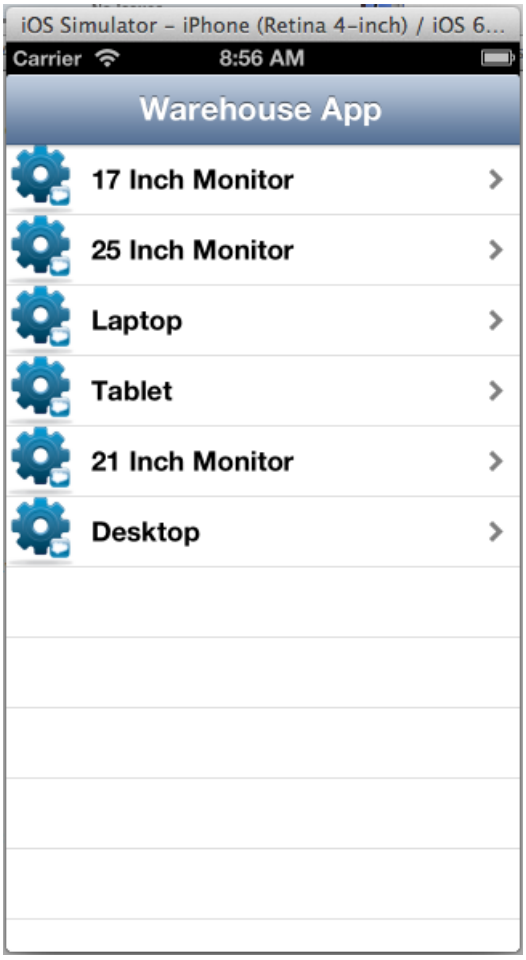
```
 SFRestRequest *request = [[SFRestAPI sharedInstance] requestForQuery:@"SELECT Name
FROM User LIMIT 10"];
```

to:

```
SELECT Name, Id, Quantity__c, Price__c FROM Merchandise__c LIMIT 10
```

Step 3:Try Out the App

Build and run the app. When prompted, log into your DE org. The initial page should look similar to the following screen.



At this point, if you click a Merchandise record, nothing happens. You'll fix that in the next tutorial.

Create the Detail Screen

In the previous tutorial, you modified the template app so that, after it starts, it lists up to ten Merchandise records. In this tutorial, you finish the job by creating a detail view and controller. You also establish communication between list view and detail view controllers.

Step 1: Create the App's Detail View Controller

When a user taps a Merchandise record in the Warehouse view, an `IBAction` generates record-specific information and then loads a view from `DetailViewController` that displays this information. However, this view doesn't yet exist, so let's create it.

1. Click **File** > **New** > **File...** > **Cocoa Touch** > **Objective-C class**.

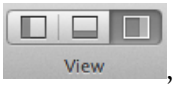
2. Create a new Objective-C class named `DetailViewController` that subclasses `UIViewController`. Make sure that **With XIB for user interface** is checked.
3. Click **Next**.
4. Place the new class in the Classes group under Mobile Warehouse App in the **Groups** drop-down menu.

   Xcode creates three new files in the `Classes` folder: `DetailViewController.h`, `DetailViewController.m`, and `DetailViewController.xib`.

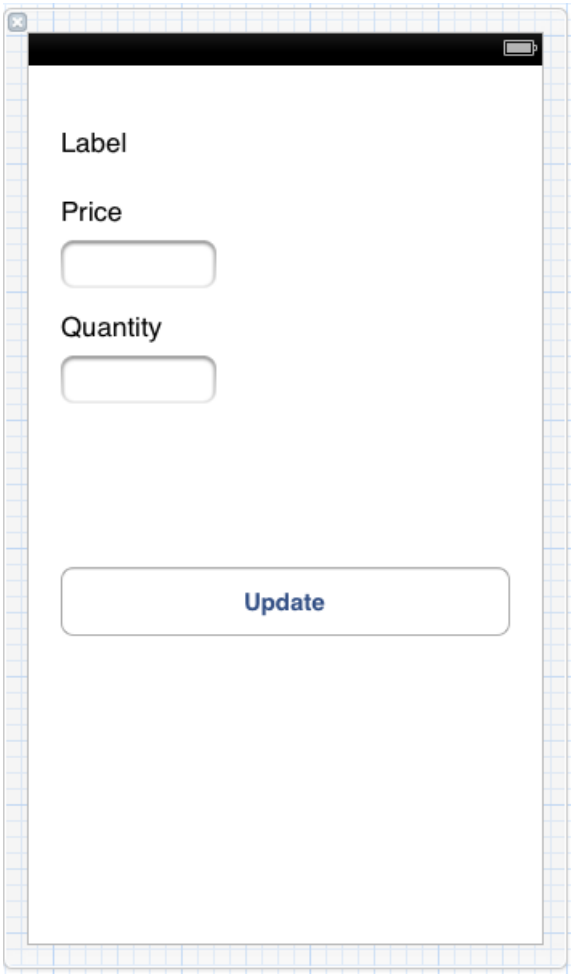
5. Select `DetailViewController.m` in the Project Navigator, and delete the following method declaration:

```
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        // Custom initialization
    }
    return self;
}
```

   In this app, you don't need this initialization method because you're not specifying a NIB file or bundle.

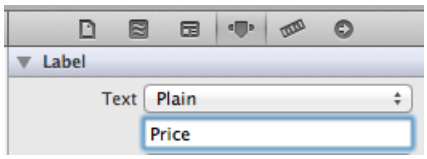
6. Select `DetailViewController.xib` in the Project Navigator to open the Interface Builder.
7. From the Utilities view 

, drag three labels, two text fields, and one button onto the view layout. Arrange and resize the controls so that the screen looks like this:



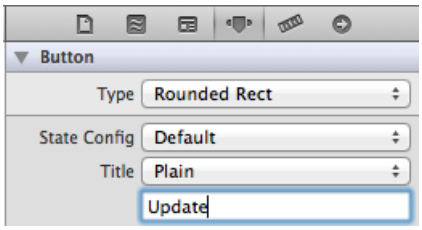
We'll refer to topmost label as the Name label. This label is dynamic. In the next tutorial, you'll add controller code that resets it at runtime to a meaningful value.

8. In the Attributes inspector, set the display text for the static Price and Quantity labels to the values shown. Select each label individually in the Interface Builder and specify display text in the unnamed entry field below the Text drop-down menu.



Note: Adjust the width of the labels as necessary to see the full display text.

9. In the Attributes inspector, set the display text for the Update button to the value shown. Select the button in the Interface Builder and specify its display text in the unnamed entry field below the Title drop-down menu.



10. Build and run to check for errors. You won't yet see your changes.

The detail view design shows Price and Quantity fields, and provides a button for updating the record's Quantity. However, nothing currently works. In the next step, you learn how to connect this design to Warehouse records.

Step 2: Set Up DetailViewController

To establish connections between view elements and their view controller, you can use the Xcode Interface Builder to connect UI elements with code elements.

Add Instance Properties

1. Create properties in `DetailViewController.h` to contain the values passed in by the `WarehouseViewController`: Name, Quantity, Price, and Id. Place these properties within the `@interface` block. Declare each `nonatomic` and `strong`, using these names:

```
@interface DetailViewController : UIViewController

@property (nonatomic, strong) NSNumber *quantityData;
@property (nonatomic, strong) NSNumber *priceData;
@property (nonatomic, strong) NSString *nameData;
@property (nonatomic, strong) NSString *idData;


@end
```

2. In `DetailViewController.m`, just after the `@implementation` tag, synthesize each of the properties.

```
@implementation DetailViewController

@synthesize nameData;
```

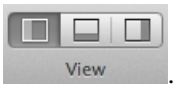
```
@synthesize quantityData;
@synthesize priceData;
@synthesize idData;
```

Add IBOutlet Variables

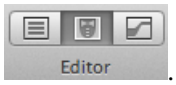
`IBOutlet` member variables let the controller manage each non-static control. Instead of coding these manually, you can use the Interface Builder to create them. Interface Builder provides an Assistant Editor that gives you the convenience of side-by-side editing windows. To make room for the Assistant Editor, you'll usually want to reclaim screen area by hiding unused controls.

1. In the Project Navigator, click the `DetailViewController.xib` file.

   The `DetailViewController.xib` file opens in the Standard Editor.

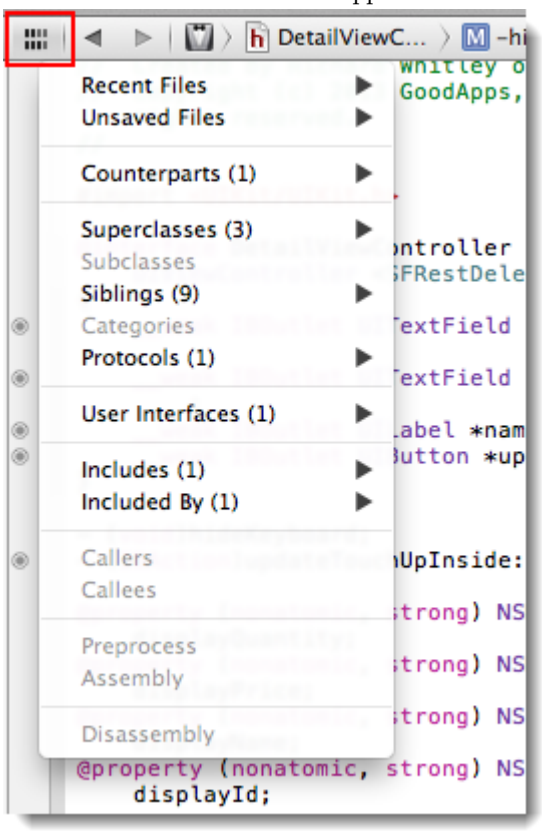
2. Hide the Navigator by clicking Hide or Show Navigator on the View toolbar 

. Alternatively, you can choose **View** > **Navigators** > **Hide Navigators** in the Xcode menu.

3. Open the Assistant Editor by clicking Show the Assistant editor in the Editor toolbar 

. Alternatively, you can choose **View** > **Assistant Editor** > **Show Assistant Editor** in the Xcode menu.

   Because you opened `DetailViewController.xib` in the Standard Editor, the Assistant Editor shows the `DetailViewController.h` file. The Assistant Editor guesses which files are most likely to be used together. If you need to open a different file, click the Related Files control in the upper left hand corner of the Assistant Editor.



.

4. At the top of the interface block in `DetailViewController.h`, add a pair of empty curly braces:

```
@interface DetailViewController : UiViewController <SFRestDelegate>
{
```

```
}
```

5. In the Standard Editor, control-click the Price text field control and drag it into the new curly brace block in the `DetailViewController.h` file.
6. In the popup dialog box, name the new outlet `_priceField`, and click **Connect**.
7. Repeat steps 2 and 3 for the Quantity text field, naming its outlet `_quantityField`.
8. Repeat steps 2 and 3 for the Name label, naming its outlet `_nameLabel`.

Your interface code now includes this block:

```
@interface DetailViewController : UIViewController <SFRestDelegate>
{
    __weak IBOutlet UITextField *_priceField;
    __weak IBOutlet UITextField *_quantityField;
    __weak IBOutlet UILabel *_nameLabel;
}
```

Add an Update Button Event

1. In the Interface Builder, select the **Update** button and open the Connections Inspector

2. In the Connections Inspector, select the circle next to **Touch Up Inside** and drag it into the `DetailViewController.h` file. Be sure to drop it below the closing curly brace. Name it `updateTouchUpInside`, and click **Connect**.

   The Touch Up Inside event tells you that the user raised the finger touching the Update button without first leaving the button. You'll perform a record update every time this notification arrives.

Step 3: Create the Designated Initializer

Now, let's get down to some coding. Start by adding a new initializer method to `DetailViewController` that takes the name, ID, quantity, and price. The method name, by convention, must begin with "init".

1. Click **Show the Standard Editor** and open the Navigator.
2. Add this declaration to the `DetailViewController.h` file just above the `@end` marker:

```
- (id) initWithName:(NSString *)recordName
          sobjectId:(NSString *)salesforceId
           quantity:(NSNumber *)recordQuantity
              price:(NSNumber *)recordPrice;
```

   Later, we'll code `WarehouseViewController` to use this method for passing data to the `DetailViewController`.

3. Open the `DetailViewController.m` file, and copy the signature you created in the previous step to the end of the file, just above the `@end` marker.
4. Replace the terminating semi-colon with a pair of curly braces for your implementation block.

```
- (id) initWithName:(NSString *)recordName
          sobjectId:(NSString *)salesforceId
           quantity:(NSNumber *)recordQuantity
              price:(NSNumber *)recordPrice {
}
```

5. In the method body, send an `init` message to the super class. Assign the return value to `self`:

```
self = [super init];
```

This `init` message gives you a functional object with base implementation which will serve as your return value.

6. Add code to verify that the super class initialization succeeded, and, if so, assign the method arguments to the corresponding instance variables. Finally, return `self`.

```
if (self) {
    self.nameData = recordName;
    self.idData = salesforceId;
    self.quantityData = recordQuantity;
    self.priceData = recordPrice;
}
return self;
```

Here's the completed method:

```
- (id) initWithName:(NSString *)recordName
                          sobjectId:(NSString *)salesforceId
                          quantity:(NSNumber *)recordQuantity
                                price:(NSNumber *)recordPrice {
    self = [super init];
    if (self) {
        self.nameData = recordName;
        self.idData = salesforceId;
        self.quantityData = recordQuantity;
        self.priceData = recordPrice;
    }
    return self;
}
```

7. To make sure the controls are updated each time the view appears, add a new `viewWillAppear:` event handler after the `viewDidLoad` method implementation. Begin by calling the super class method.

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

}
```

8. Copy the values of the property variables to the corresponding dynamic controls.

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    [_nameLabel setText:self.nameData];
    [_quantityField setText:[self.quantityData stringValue]];
    [_priceField setText:[self.priceData stringValue]];
}
```

9. Build and run your project to make sure you've coded everything without compilation errors. The app will look the same as it did at first, because you haven't yet added the code to launch the Detail view.

Note: The `[super init]` message used in the `initWithName:` method calls `[super initWithNibName:bundle:]` internally. We use `[super init]` here because we're not passing a NIB name or a bundle. If you are specifying these resources in your own projects, you'll need to call `[super initWithNibName:bundle:]` explicitly.

Step 4: Establish Communication Between the View Controllers

Any view that consumes Salesforce content relies on a `SFRestAPI` delegate to obtain that content. You can designate a single view to be the central delegate for all views in the app, which requires precise communication between the view controllers. For this exercise, let's take a slightly simpler route: Make `WarehouseViewController` and `DetailViewController` each serve as its own `SFRestAPI` delegate.

Update WarehouseViewController

First, let's equip `WarehouseViewController` to pass the quantity and price values for the selected record to the detail view, and then display that view.

1. In `WarehouseViewController.m`, above the `@implementation` block, add the following line:

```
#import "DetailViewController.h"
```

2. On a new line after the `#pragma mark - Table view data source` marker, type the following starter text to bring up a list of `UITableView` delegate methods:

```
- (void)tableView
```

3. From the list, select the `tableView:didSelectRowAtIndexPath:` method.
4. Change the `tableView` parameter name to `itemTableView`.

```
- (void)tableView:(UITableView *)itemTableView didSelectRowAtIndexPath:
(NSIndexPath *)indexPath
```

5. At the end of the signature, type an opening curly brace ({) and press return to stub in the method implementation block.
6. At the top of the method body, per standard iOS coding practices, add the following call to deselect the row.

```
    [itemTableView deselectRowAtIndexPath:indexPath animated:NO];
```

7. Next, retrieve a pointer to the `NSDictionary` object associated with the selected data row.

```
NSDictionary *obj = [self.dataRows objectAtIndex:indexPath.row];
```

8. At the end of the method body, create a local instance of `DetailViewController` by calling the `DetailViewController.initWithName:salesforceId:quantity:price:` method. Use the data stored in the `NSDictionary` object to set the name, Salesforce ID, quantity, and price arguments. The finished call looks like this:

```
DetailViewController *detailController =
    [[DetailViewController alloc] initWithName:[obj objectForKey:@"Name"]
                                  salesforceId:[obj objectForKey:@"Id"]
                                      quantity:[obj objectForKey:@"Quantity__c"]
                                         price:[obj objectForKey:@"Price__c"]];
```

9. To display the Detail view, add code that pushes the initialized `DetailViewController` onto the `UINavigationController` stack:

```
[[self navigationController]
pushViewController:detailController animated:YES];
```

Great! Now you're using a `UINavigationController` stack to handle a set of two views. The root view controller is always at the bottom of the stack. To activate any other view, you just push its controller onto the stack. When the view is dismissed, you pop its controller, which brings the view below it back into the display.

10. Build and run your app. Click on any Warehouse item to display its details.

Add Update Functionality

Now that the `WarehouseViewController` is set up, we need to modify the `DetailViewController` class to send the user's updates to Salesforce via a REST request.

1. In the `DetailViewController.h` file, add the following line at the top of the file.

```
#import "SFRestAPI.h"
```

2. Add an instance method to `DetailViewController` that lets a user update the price and quantity fields. This method needs to send a record ID, the names of the fields to be updated, the new quantity and price values, and the name of the object to be updated. Add this declaration after the interface block and just above the `@end` marker.

```
- (void)updateWithObjectType:(NSString *)objectType
              objectId:(NSString *)objectId
               quantity:(NSString *)quantity
               price:(NSString *)price;
```

To implement the method, you create an `SFRestRequest` object using the input values, then send the request object to the shared instance of the `SFRestAPI`.

3. In the `DetailViewController.m` file, just before the final `@end` marker, copy the `updateWithObjectType:objectId:quantity:price:` signature followed by a pair of curly braces.

```
- (void)updateWithObjectType:(NSString *)objectType
                    objectId:(NSString *)objectId
                    quantity:(NSString *)quantity
                       price:(NSString *)price {

}
```

4. In the implementation block, create a new `NSDictionary` object to contain the Quantity and Price fields. To allocate this object, use the `dictionaryWithObjectsAndKeys: ...` `NSDictionary` class method with the desired list of fields.

```
- (void)updateWithObjectType:(NSString *)objectType
                    objectId:(NSString *)objectId
                    quantity:(NSString *)quantity
                       price:(NSString *)price {

    NSDictionary *fields = [NSDictionary dictionaryWithObjectsAndKeys:
                                            quantity, @"Quantity__c",
                                            price, @"Price__c",
                                            nil];

}
```

5. Create a `SFRestRequest` object. To allocate this object, use the `requestForUpdateWithObjectType:objectId:fields:` instance method on the `SFRestAPI` shared instance.

```
- (void)updateWithObjectType:(NSString *)objectType
                    objectId:(NSString *)objectId
                    quantity:(NSString *)quantity
                       price:(NSString *)price {
```

```
    NSDictionary *fields = [NSDictionary dictionaryWithObjectsAndKeys:
                                            quantity, @"Quantity__c",
                                            price, @"Price__c",
                                            nil];
    SFRestRequest *request =
        [[SFRestAPI sharedInstance]
            requestForUpdateWithObjectType:objectType
                                  objectId:objectId
                                    fields:fields];

}
```

6. Finally, send the new `SFRestRequest` object to the service by calling `send:delegate:` on the `SFRestAPI` shared instance. For the `delegate` argument, be sure to specify `self`, since `DetailViewController` is the `SFRestDelegate` in this case.

```
- (void)updateWithObjectType:(NSString *)objectType
                    objectId:(NSString *)objectId
                    quantity:(NSString *)quantity
                       price:(NSString *)price {

    NSDictionary *fields = [NSDictionary dictionaryWithObjectsAndKeys:
                                            quantity, @"Quantity__c",
                                            price, @"Price__c",
                                            nil];
    SFRestRequest *request =
        [[SFRestAPI sharedInstance]
            requestForUpdateWithObjectType:objectType
                                  objectId:objectId
                                    fields:fields];

   [[SFRestAPI sharedInstance] send:request delegate:self];
}
```

7. Edit the `updateTouchUpInside:` action method to call the `updateWithObjectType:objectId:quantity:price:` method when the user taps the **Update** button.

```
- (IBAction)updateTouchUpInside:(id)sender {
    // For Update button
    [self updateWithObjectType:@"Merchandise__c"
                      objectId:self.idData
                      quantity:[_quantityField text]
                         price:[_priceField text]];}
```

Note:

- **Extra credit:** Improve your app's efficiency by performing updates only when the user has actually changed the quantity value.

Add SFRestDelegate to DetailViewController

We're almost there! We've issued the REST request, but still need to provide code to handle the response.

1. Open the `DetailViewController.h` file and change the `DetailViewController` interface declaration to include `<SFRestDelegate>`.

```
@interface DetailViewController : UIViewController <SFRestDelegate>
```

2. Open the `WarehouseViewController.m` file.

3. Find the `pragma` that marks the `SFRestAPIDelegate` section.

```
#pragma mark - SFRestAPIDelegate
```

4. Copy the four methods under this pragma into the `DetailViewController.m` file.

```
- (void)request:(SFRestRequest *)request didLoadResponse:(id)jsonResponse {
    NSArray *records = [jsonResponse objectForKey:@"records"];
    NSLog(@"request:didLoadResponse: #records: %d", records.count);
    self.dataRows = records;
    [self.tableView reloadData];
}

- (void)request:(SFRestRequest*)request didFailLoadWithError:(NSError*)error {
    NSLog(@"request:didFailLoadWithError: %@", error);
    //add your failed error handling here
}

- (void)requestDidCancelLoad:(SFRestRequest *)request {
    NSLog(@"requestDidCancelLoad: %@", request);
    //add your failed error handling here
}

- (void)requestDidTimeout:(SFRestRequest *)request {
    NSLog(@"requestDidTimeout: %@", request);
    //add your failed error handling here
}
```

These methods are all we need to implement the `SFRestAPI` interface. For this tutorial, we can retain the simplistic handling of error, cancel, and timeout conditions. However, we need to change the `request:didLoadResponse:` method to suit the detail view purposes. Let's use the `UINavigationController` stack to return to the list view after an update occurs.

5. In the `DetailViewController.m` file, delete the existing code in the `request:didLoadResponse:` delegate method. In its place, add code that logs a success message and then pops back to the root view controller. The revised method looks like this.

```
- (void)request:(SFRestRequest *)request didLoadResponse:(id)jsonResponse {
    NSLog(@"1 record updated");
    [self.navigationController popViewControllerAnimated:YES];
}
```

6. (Mobile SDK 2.1 and later only) At the top of the `request:didLoadResponse:` method, you also need code that guarantees you're on the main thread before calling UI methods. Wrap the existing code in a dispatch call, as highlighted:

```
- (void)request:(SFRestRequest *)request didLoadResponse:(id)jsonResponse {
    dispatch_async(dispatch_get_main_queue(), ^{
        NSLog(@"1 record updated");
        [self.navigationController popViewControllerAnimated:YES];
    });
}
```

7. Build and run your app. In the Warehouse view, click one of the items. You're now able to access the Detail view and edit its quantity, but there's a problem: the keyboard won't go away when you want it to. You need to add a little finesse to make the app truly functional.

Hide the Keyboard

The iOS keyboard remains visible as long as any text input control on the screen is responding to touch events. This is where the "First Responder" setting, which you might have noticed in the Interface Builder, comes into play. We didn't set a first

responder because our simple app just uses the default UIKit behavior. As a result, iOS can consider any input control in the view to be the first responder. If none of the controls explicity tell iOS to hide the keyboard, it remains active.

You can resolve this issue by making every touch-enabled edit control resign as first responder.

1. In `DetailViewController.h`, below the curly brace block, add a new instance method named `hideKeyboard` that takes no arguments and returns `void`.

```
- (void)hideKeyboard;
```

2. In the implementation file, implement this method to send a `resignFirstResponder` message to each touch-enabled edit control in the view.

```
- (void)hideKeyboard {
    [_quantityField resignFirstResponder];
    [_priceField resignFirstResponder];
}
```

The only remaining question is where to call the `hideKeyboard` method. We want the keyboard to go away when the user taps outside of the text input controls. There are many likely events that we could try, but the only one that is sure to catch the background touch under all circumstances is `[UIResponder touchesEnded:withEvent:]`.

3. Since the event is already declared in a class that `DetailViewController` inherits, there's no need to re-declare it in the `DetailViewController.h` file. Rather, in the `DetailViewController.m` file, type the following incomplete code on a new line outside of any method body:

```
- (void)t
```

A popup menu displays with a list of matching instance methods from the `DetailViewController` class hierarchy.

Note: If the popup menu doesn't appear, just type the code described next.

4. In the popup menu, highlight the `touchesEnded:withEvent:` method and press `Return`. The editor types the full method signature into your file for you. Just type an opening brace, press `Return`, and your stub method is completed by the editor. Within this stub, send a `hideKeyboard` message to `self`.

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event{
    [self hideKeyboard];
}
```

Normally, in an event handler, you'd be expected to call the super class version before adding your own code. As documented in the iOS Developer Library, however, leaving out the super call in this case is a common usage pattern. The only "gotcha" is that you also have to implement the other touches event handlers, which include:

```
- touchesBegan:withEvent:
- touchesMoved:withEvent:
- touchesCancelled:withEvent:
```

The good news is that you only need to provide empty stub implementations.

5. Use the Xcode editor to add these stubs the same way you added the `touchesEnded:` stub. Make sure your final code looks like this:

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event{
    [self hideKeyboard];
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event{
```

```
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event{

}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event{

}
```

Refreshing the Query with viewWillAppear

The `viewDidLoad` method lets you configure the view when it first loads. In the `WarehouseViewController` implementation, this method contains the REST API query that populates both the list view and the detail view. However, since `WarehouseViewController` represents the root view, the `viewDidLoad` notification is called only once—when the view is initialized. What does this mean? When a user updates a quantity in the detail view and returns to the list view, the query is not refreshed. Thus, if the user returns to the same record in the detail view, the updated value does not display, and the user is not happy.

You need a different method to handle the query. The `viewWillAppear` method is called each time its view is displayed. Let's add this method to `WarehouseViewController` and move the SOQL query into it.

1. In the `WarehouseViewController.m` file, add the following code after the `viewDidLoad` implementation.

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

}
```

2. Cut the following lines from the `viewDidLoad` method and paste them into the `viewWillAppear:` method, after the call to `super`:

```
    SFRestRequest *request = [[SFRestAPI sharedInstance] requestForQuery:@"SELECT Name,
        ID, Price__c, Quantity__c FROM Merchandise__c LIMIT 10"];
    [[SFRestAPI sharedInstance] send:request delegate:self];
```

   The final `viewDidLoad` and `viewWillAppear:` methods look like this.

```
- (void)viewDidLoad{
    [super viewDidLoad];
    self.title = @"Warehouse App";
}

- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    //Here we use a query that should work on either Force.com or Database.com
    SFRestRequest *request = [[SFRestAPI sharedInstance] requestForQuery:@"SELECT Name,
        ID, Price__c, Quantity__c FROM Merchandise__c LIMIT 10"];
    [[SFRestAPI sharedInstance] send:request delegate:self];
}
```

The `viewWillAppear:` method refreshes the query each time the user navigates back to the list view. Later, when the user revisits the detail view, the list view controller updates the detail view with the refreshed data.

Step 5: Try Out the App

1. Build your app and run it in the iPhone emulator. If you did everything correctly, a detail page appears when you click a Merchandise record in the Warehouse screen.

2. Update a record's quantity and price. Be sure to click the **Update** button in the detail view after you edit the values. When you navigate back to the detail view, the updated values display.
3. Log into your DE org and view the record using the browser UI to see the updated values.