Dynamic Table Views

Dynamic table views differ from Static table views in that they generate their display at runtime. This is done using a *datasource* and *delegate*. The datasource mediates between the table view and the data or model, consistent with the MVC pattern. The delegate manages the appearance and behavior of the table view.

Recall the delegate pattern we've used before, where the class that implements the delegate protocol must implement certain methods in order to respond accordingly. Datasource works similarly to delegates.

iOS provides the UITableViewController class, which is a subclass of UIViewController for handling table views. It implements two protocols, the UITableViewDatasource and UITableViewDelegate.

As a datasource, there are several methods that are required at a minimum:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:
(NSInteger)section
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
```

The first two tell the tableview how many sections and rows the data has. The third method tells the tableview how to display the cell in a particular index path.

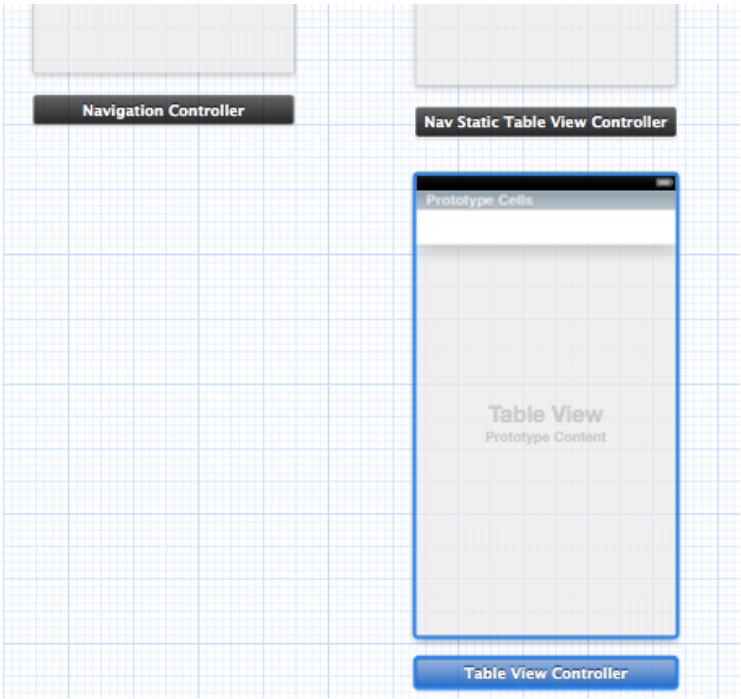
NOTE: The NSIndexPath in this case is simplistically used to refer to sections and rows. In reality, it can be used for locating objects in nested structures. For purposes of table view, the NSIndexPath instances will have .section and .row properties.

Let's start modifying our project to use dynamic table views instead of static.

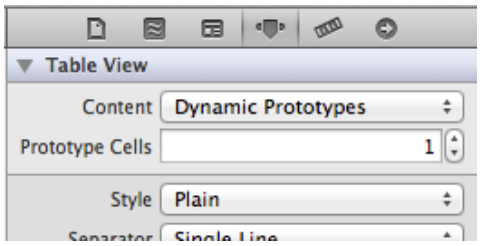
First, let's drag over a new Table View Controller onto the storyboard.

Note that we haven't removed the other view controllers yet... we'll use them later for our tab bar.

Let's make this new table view the root view of our navigation controller by control-dragging to it.



Notice how, by default, the table view is dynamic. You can verify this clicking the table view and going into the attributes inspector.



By default, it provides us with one prototype cell.

A prototype cell is used as a template for displaying the content cells later on. Each content cell can use a prototype cell so that the layout elements can be used. You can also programmatically create cells at runtime, but prototype cells provide an easy way of predefining the layouts.

Each prototype cell is a UITableViewCell object. Note that UITableViewCells supports four standard layouts, Basic, RIght Detail, Left Detail, and Subtitle:

The four standard layouts are pre-wired to the UITableViewCell's properties called "textLabel" (where it says "Title" in the layout) and "detailTextLabel" (where it says "Subtitle" in the layout).

There is also a custom layout wherein you can create your own layouts, but this tends to be slightly more finicky. We'll discuss several ways to do this when we discuss customizing table views.

In the meantime, let's set our prototype cell to Basic by clicking on the cell and selecting Basic in the attributes inspector. Another important thing we need to do is to give this prototype a "Reuse identifier". For now, type "Cell" (without the quotes) into the Identifier field.

Unlike a static table view, which can work even without a custom view controller class, a dynamic table view requires one because of its datasource requirement. Thus, we need to create a new class by going to File -> New -> File..., select Objective-C class, and let's call it NavDynamicTableViewController, and make a sub-class of UITableViewController. Then let's go to storyboard and set the custom class of our new table view contorller to NavDynamicTableViewController in the identity inspector.

If you run it now, it should give no errors but it will not show any cells. Notice that it will give (at least) two warnings. These warnings are purposely placed by the UITableViewController subclass template to remind us to do certain things. In this case, it's reminding us about the number of sections and number of rows methods.

Let's go ahead and look at NavDynamicTableViewController.m and notice that under the #pragma mark Table View Data source it has created for us placeholders for the three required methods.

Let's change the number of sections to return 1, and the number of rows to return 5. If we run it at this point, we'll see Title repeated five times. If you change the number of sections to 2, you'll see Title repeated five times, twice.

To change the content of each cell, we need to edit cellForRowAtIndexPath: method.

Notice how this method returns a UITableViewCell instance. Also notice how it begins with:

```
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier
forIndexPath:indexPath];
```

The CellIdentifier is really treated as a constant string (the static modifier means the same instance of the variable CellIdentifier is used across all instances of this class).

The more important line here is the second line. The dequeueReusable... method basically tells the tableview to return a prototype with the identifier "Cell" to be used in indexPath. This is the reason why we had to give our prototype a reuse identifier... so that we can identify it by name in calling dequeueReusable... If we had more than one prototype, we could do an if statement to determine which prototype is to be used, and pass on the appropriate identifier. We'll see this later on.

The reason why we say "reuse" is because the table view will only create as many instances of UITableViewCell as it needs to to fill the screen (and a couple more to handle scrolling). When you start scrolling, it will start reusing these instances.

In any case, whether it's a new instance or an actual reuse, we will get a table view cell instance in the variable cell which we can then use to set up the display.

Recall what we said earlier about UITableViewCells and the standard layouts providing two properties textLabel and detailTextLabel. Our variable cell, therefore, will have these properties. By changing these properties, we can therefore change the display. Note, however, that each of these properties are *UILabels*. Therefore, to set the display, we need to use the following format:

```
    cell.textLabel.text = @"some text...";
```

Let's make it display the row number:

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier
forIndexPath:indexPath];

    // Configure the cell...
    cell.textLabel.text = [NSString stringWithFormat:@"Row %d",indexPath.row];

    return cell;
}
```

If you run it, you'll find it displays the row number.

Let's play around with it and set the number of rows to 99, and change it to say:

```
    // Configure the cell...
    if (indexPath.row < 10) {
        cell.textLabel.text = [NSString stringWithFormat:@"Row %d",indexPath.row];
    }
```

If you run it with the changes, you'll get some weirdness after Row 9, because the table view starts to reuse the cells but we're not assigning display values for rows after that, so it shows whatever the old cell had before.

Let's try doing multiple prototypes. (You can create a snapshot at this point as we will be going back to the previous state later on.)

Go back to the storyboard, click on the table view, and increase the number of prototypes to 2 in the attributes inspector. You should get a clone of your original prototype.

Click on this second cell, change it to Subtitle (instead of Basic) and change its reuse identifier to Cell2.

Go back to the NavDynamicTVC.m and change the cellForRow... method to:

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    static NSString *CellIdentifier2 = @"Cell2";
    UITableViewCell *cell;

    if (indexPath.row < 10) {
        cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier
forIndexPath:indexPath];
    } else {
        cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier2
forIndexPath:indexPath];

    }
    // Configure the cell...
    cell.textLabel.text = [NSString stringWithFormat:@"Row %d",indexPath.row];
    cell.detailTextLabel.text = [NSString stringWithFormat:@"Section
%d",indexPath.section];

    return cell;
}
```

We've created a second static NSString containing Cell2, and did an if statement to determine which cell identifier to reuse. Notice we've declared our UITableViewCell outside the if statement. Then we set the property detailTextLabel to display the section number.

Notice that although we set the detailTextLabel for all cells (it's outside the if, after all), the information will only show up in the rows where we use Cell2 (i.e., the 11th row onwards).

Let's bring everything back to our single prototype and display something more useful.

Dynamic table views are useful for displaying data coming from a structured source, such as a database. We will discuss Core Data later on, but in some cases, it is better to use simpler structures such as arrays and dictionaries to store our data.

For this example, we'll start with a simple array.

Go to NavDynamicTVC.h and declare a property called languages.

```
@property (strong, nonatomic) NSArray *languages;
```

Then go to the .m, remember to @synthesize if you are using Xcode 4.4 and earlier.

In the viewDidLoad (if you don't have it, type it in), we will initialize the array to contain the languages that we want.

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

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

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

    self.languages = @[@"French", @"Spanish", @"German"];

    self.navigationItem.title = @"Hello World";

}
```

Note that the self.languages line above is equivalent to the old syntax:

```
    self.languages = [[NSArray alloc] initWithObjects:@"French", @"Spanish",
@"German",nil];
```

Next, we'll modify the numberOfRows to make sure it returns the number of objects int he array. If we don't do this, we will likely get an index out of bounds error.

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    // Return the number of rows in the section.
    return [self.languages count];
}
```

cellForRow method to display the languages:

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier
forIndexPath:indexPath];

    // Configure the cell...
    cell.textLabel.text = [self.languages objectAtIndex:indexPath.row];

    return cell;
}
```

Now let's replicate our previous functionality...

Go to storyboard and create a push segue from your prototype cell to the NavTranslateVC. We'll reuse this same view controller and set up our NavDynamicTVC to pass data. Note that for each prototype, you need to create a segue if you wish to use the storyboard functionality for transitioning to another view. You can make each prototype segue to the same or a different view.

You might also notice in the NavDynamicTVC.m that there is a method called didSelectRowAtIndexPath:. This method gets called when the user taps a row, and was the pre-storyboard way of transitioning to another view. With storyboards, it still works, but it gets called after the segue. Thus, the sequence of events is: user taps a row, prepareForSegue gets called, segue occurs, didSelectRow gets called.

In previous incarnations of this translation app, we used prepareForSegue to determine the what to display in the translation view. In this case, let's take advantage of the dynamic data. Rather than doing a bunch of if statements and hardcoding the translation, let's put the translation in the array.

To do this, instead of an array of string, we'll change it to an array of dictionaries. Change the array initialization in viewDidLoad to:

```
    self.languages = @[
        @{@"Language" : @"French", @"Translation" : @"Bonjour, Monde!"},
        @{@"Language" : @"Spanish", @"Translation" : @"Hola, Mundo!"},
        @{@"Language" : @"German", @"Translation" : @"Hallo, Welt!"}
    ];
```

Our languages array now contains dictionaries with two keys each, Language and Translation. We then need to change our cellForRow to say:

```
    // Configure the cell...
    NSDictionary *oneLanguage = [self.languages objectAtIndex:indexPath.row];
    cell.textLabel.text = [oneLanguage objectForKey:@"Language"];
```

We use the oneLanguage variable to make things easier to read by using it to temporarily store the NSDictionary retrieved from the array. Then, we need to make sure our segue has an identifier (let's call it Translate), and add a method prepareForSegue before @end:

```
- (void) prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if ([[segue identifier] isEqualToString:@"Translate"]) {

        NSIndexPath *indexPath = [self.tableView indexPathForSelectedRow];
        NSDictionary *oneLanguage = [self.languages objectAtIndex:indexPath.row];

        NavTranslateViewController *vc = [segue destinationViewController];
        vc.language = [oneLanguage objectForKey:@"Language"];
        vc.translatedText = [oneLanguage objectForKey:@"Translation"];

    }
}
```

It starts of similarly to our previous incarnations, however, notice the line with NSIndexPath. This line queries the table view for the current index path. This information is not part of the segue, so we need to get it directly from the table view. Next, as we did with cellForRow, we use an NSDictionary temporary variable. Then we proceed to set the properties of the destinationViewController. Remember to #import "NavTranslateViewController.h"

Let's add flag images to our app!

In order to do that, first you need to create artwork for each flag. Once you have these artwork in jpg or png or gif, you can go to File -> Add Files to "projectname". Make sure to check the "Copy items into destination group's folder", otherwise your project will be referencing files all over the place and it will become difficult to manage.

Once you've selected your files, and click Open, you'll find that the image files in your project. At this point, you can try organizing your project files by creating groups. You can quickly group files together by highlighting them, option clicking (or right clicking), and new group from selection.

Next, we need to add the image file name to our array of languages:

```
    self.languages = @[
        @{@"Language" : @"French", @"Translation" : @"Bonjour, Monde!", @"Flag" :
@"france.gif"},
        @{@"Language" : @"Spanish", @"Translation" : @"Hola, Mundo!", @"Flag" :
@"spain.gif"},
        @{@"Language" : @"German", @"Translation" : @"Hallo, Welt!", @"Flag" :
@"germany.gif"}
    ];
```

Make sure the file name matches what you've imported into the project.

Next we go to storyboard and drag over an image view onto the NavTranslateVC scene. Size the image view appropriately. Create an IBOutlet for this called flagImageView.

It is best to not rely on the scaling as much as possible and match the size of the content with the size of the view. Also you can create two versions of your image asset, say france.jpg and france@2x.jpg. You can simply use france.jpg in your code, but the system will automatically seek out and use the @2x version for retina devices. You can also just have the @2x version if you wish and let the system scale for you, but that might not be optimal in all cases. Remember the size of your view is measured in points, but your image pixels remain as pixels so a 240 x 180 image view can host a 480x360pixel image on retina devices.

Next go to NavTranslateVC.h and add a property for flag:

```
@property (strong, nonatomic) NSString *flag;
```

Now add the following to the NavTranslateVC.m viewDidLoad:

```
    self.flagImageView.image = [UIImage imageNamed:self.flag];
```

This creates a new instance of UIImage from the file named by self.flag, and assigns it to the image view.

Then we add the following to prepareForSegue.

```
        vc.flag = [oneLanguage objectForKey:@"Flag"];
```

That's it!