User interface design

for iOS, using Objective C

# Lab 1 – Creating a basic application

## Functional Goals

Create a functional iOS app that contains the necessary controls to display bandwidth usage data from Rose-Hulman’s network usage tool.

## Learning Goals

* Understand iOS UI paradigms, including the tab bar control and the use of UIKit to create a straightforward, attractive iOS application
* Understand the basics of Interface Builder and Objective-C

## Prerequisites

* You’ll need to install Xcode (Version 4.5.2 was used to create this lab) from the App Store
* A basic understanding of Xcode and Objective C development

## Submission Instructions

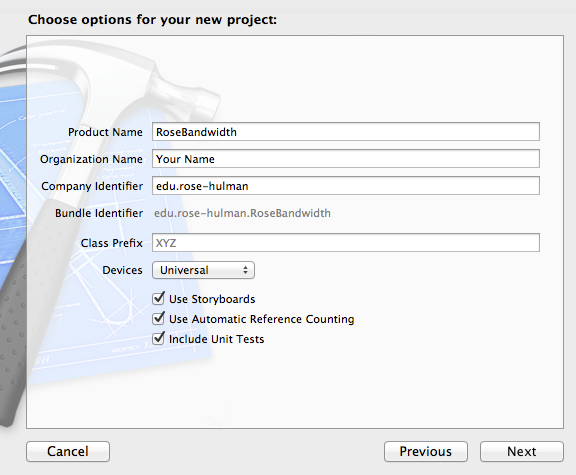
Submit answers to the **3** (or **4**, with extra credit)questions in this lab as a .pdf to the appropriate Moodle submission form.

## Get started: Create a project

Open Xcode and create a new iOS Tabbed Application from template.

Xcode comes with lots of built in templates. Choosing the correct template is important for starting your project. Tabbed applications are useful for easily swapping between more than one distinct view.

Under Product Name, enter RoseBandwidth. Set the organization name to your own name, and the company identifier to edu.rose-hulman. Leave “Use Storyboards”, “Use Automatic Reference Counting”, and “Include Unit Tests” checked.

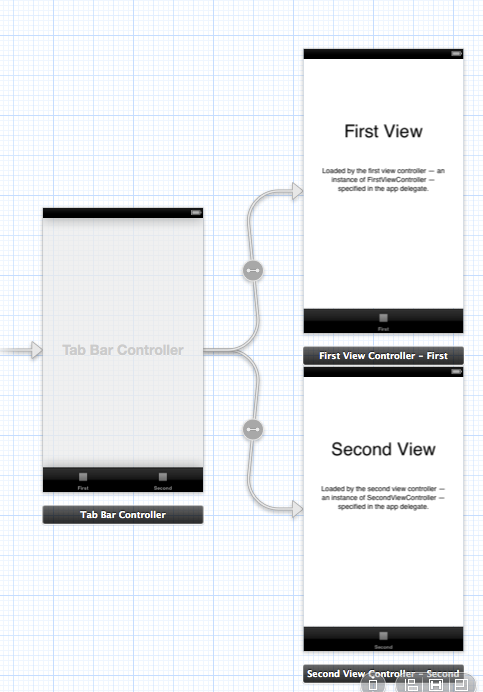


Choose a location to save your project and it should be created! Go ahead and run it by clicking the Run button in the top left corner of Xcode. You should see the simulator come up with something just like this by default.

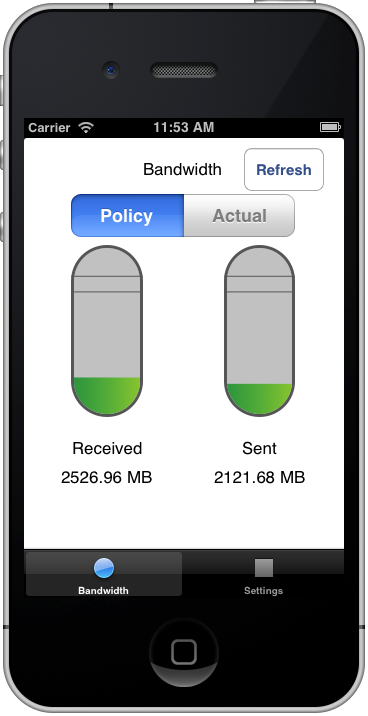


## Using Interface Builder

Go to the *MainStoryboard\_iPhone.storyboard* file and click open it. You should see 2 views contained in a single tab bar controller, like so:



We are first going to add our UIKit elements, or built-in control items, to the first view. Our goal is for our first view to look like this.



What all do we have on this page? We have:

1. a text label (UILabel) at the top
2. a button (UIButton) for refreshing the bandwidth
3. a segmented control which acts like a radio selection (UISegmentedControl) to select whether we want to see the Policy or Actual bandwidth
4. 2 custom views that look like test tubes to display the bandwidth information graphically
5. text labels to label the left set of information as Received and the right set of information as Sent
6. text labels below each to give the exact bandwidth usage in megabytes

Lets focus on the UIKit items right now (ignore the custom views). Double click on the “First View” in interface builder to begin editing that view. Start by deleting both UILabels currently existing by selecting them and pressing delete.

In the bottom right corner of Xcode you should see the Object Library. If you don’t, select the cube button above the bottom right library control to switch to the object library. Either by searching manually or using the search bar below the library, find the UILabel object and drag 5 out into your first view, arrange them as you see them in the screenshot above. Double click them and edit the text for them to what you see above, with the exception of the labels displaying exact megabytes of bandwidth used. We will need to set those dynamically using code.

Go ahead and add the segmented control (UISegmentedControl) and the button (UIButton) as well and change the text to match the screenshot above.

Lastly, before we get into the code, change the title of the tab bar item in the first view from First to Bandwidth, by double clicking on the text at the bottom of the first view. We are now ready to code.

## Connecting InterfaceBuilder elements to code

Select the file FirstViewController.h. We are going to make it look like this:

#import <UIKit/UIKit.h>

@interface FirstViewController : UIViewController

{

}

@property (nonatomic, strong) IBOutlet UISegmentedControl \*segmentedControl;

@property (nonatomic, strong) IBOutlet UILabel \*leftLabel;

@property (nonatomic, strong) IBOutlet UILabel \*rightLabel;

@end

What we are doing here is making properties for each of the UI elements we need to interact with. The IBOutlet keyword indicates to Interface Builder that this is something we want to eventually connect to a UI element in Interface Builder. A property defines a member variable of your class and, once synthesized, auto generates getters and setters for itself.

Question 1: A strong property has its value retained for the lifetime of the object, while a weak property does not gurantee that. What does nonatomic specify? (Feel free to use the Apple Developer Center) (6 points)

Next, lets synthesize them. Go to the FirstViewController.m file. Make it look like this. All we are changing is that we are taking each property created previously and synthesizing it.

#import "FirstViewController.h"

@interface FirstViewController ()

@end

@implementation FirstViewController

@synthesize segmentedControl;

@synthesize leftLabel;

@synthesize rightLabel;

- (void)viewDidLoad

{

[super viewDidLoad];

// Do any additional setup after loading the view, typically from a nib.

}

- (void)didReceiveMemoryWarning

{

[super didReceiveMemoryWarning];

// Dispose of any resources that can be recreated.

}

@end

This lab is going to focus on just getting the UI elements, no actual data calls, so lets start with adding some dummy methods that we will use later in our code, but that will let us test our UI right now.

Add this code somewhere in the implementation of FirstViewController.m:

/\* DUMMY METHODS - we will make these return real values later! \*/

- (CGFloat) getMinValue {

return 0;

}

- (CGFloat) getMaxValue {

return 5500;

}

- (CGFloat) getCurrentPolicyDownloadValue {

return 3200;

}

- (CGFloat) getCurrentActualDownloadValue {

return 3900;

}

- (CGFloat) getCurrentPolicyUploadValue {

return 600;

}

- (CGFloat) getCurrentActualUploadValue {

return 1000;

}

- (CGFloat) getWarningLevel {

return 4000;

}

- (CGFloat) getAlertLevel {

return 4500;

}

Now that we have all of these methods returning the information we need to update our views, we need to actually update our views! Lets write an update views method.

- (IBAction) updateViews:(id)sender {

NSLog(@"Updating Views");

if(segmentedControl.selectedSegmentIndex == BANDWIDTH\_POLICY){

self.leftLabel.text = [NSString stringWithFormat:@"%.2f MB", [self getCurrentPolicyDownloadValue]];

self.rightLabel.text = [NSString stringWithFormat:@"%.2f MB", [self getCurrentPolicyUploadValue]];

} else if(segmentedControl.selectedSegmentIndex == BANDWIDTH\_ACTUAL){

self.leftLabel.text = [NSString stringWithFormat:@"%.2f MB", [self getCurrentActualDownloadValue]];

self.rightLabel.text = [NSString stringWithFormat:@"%.2f MB", [self getCurrentActualUploadValue]];

}

}

We should now probably define the macros we used, BANDWIDTH\_POLICY and BANDWIDTH\_ACTUAL, with the following code just below the imports:

#define BANDWIDTH\_POLICY 0

#define BANDWIDTH\_ACTUAL 1

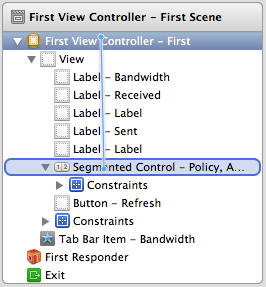
Finally, don’t forget to put the signature for updateViews into your .h file below the properties! This allows the interface builder to see this method for buttons to connect to.

- (IBAction) updateViews:(id)sender;

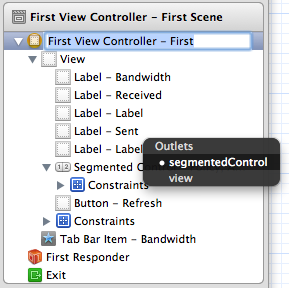
## Connecting the Interface Builder elements to Code

We have set up several hooks to connect the items we placed earlier in Interface Builder to the code. Now we actually have to do that! Go to your iPhone storyboard. For each item placed with an associated property (note that this does not include your refresh button) do the following:

1. Hold control as you click and drag from your View Controller to your UI element for which you have created an IBOutlet in code.

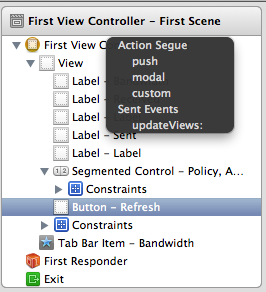


2. Select the appropriate outlet (in this case, segmentedControl).



Make sure to repeat this process for each of your IBOutlets.

For your UI Button, you will follow a similar process, but will hold control as you click on the button and drag up to the View Controller. Select your IBAction, updateViews. This will call the method updateViews every time that your button is pressed. The (id)sender that we take as an argument in each IBAction will have a reference to the button that was pressed to initiate the call, although in this case we will not use it.



Congrats! All of your UI elements should now be hooked up to code! Try running your app again. Nothing will appear to be different until you press the Refresh button. Go ahead and change the segmented control from Policy to Actual, and pressing refresh again. You will notice the numbers change. This is good, our controls are all working properly. However, we want 2 more things to happen. Firstly, the UI should update automatically when we launch. Secondly, we want changing the segmented control to have immediate consequences, not to require a refresh after making the change. So we go into the viewDidLoad method in our FirstViewController.m and add 2 lines of code, after the super call.

[segmentedControl addTarget:self action:@selector(updateViews:)

forControlEvents:UIControlEventValueChanged];

[self updateViews:nil];

The first line registers self as a listener for whenever the segmented control value changes. When we receive this, we perform the action of calling the selector updateViews. The second line does the initial update of the view with the initial information.

Question 2: We are listening on our segmented control for a change in the value. What are some other UIControlEvents that you can listen for? List 3. What do each of the listed control events do? (Hint: Autocomplete is your friend here) (6 points)

## Adding a Custom View for Visually Displaying Bandwidth

You should have been provided with 2 files – VerticalProgressView.h and VerticalProgressView.m. Import them into your project now (drag and drop works just fine). Make sure you add the files to the RoseBandwidth target, and copy the items into your destination group’s folder.

Place two generic UIViews into your bandwidth view. You may find it useful to go to the Attributes inspector in the top right corner of Xcode and make the background some color other than white to be able to see the views better. Switch to the Identity inspector and under the Custom Class field enter VerticalProgressView. Now we need to reference these custom views in code. Switch back to the FirstViewController and add two new properties.

@property (nonatomic, strong) IBOutlet VerticalProgressView \*leftUsageView;

@property (nonatomic, strong) IBOutlet VerticalProgressView \*rightUsageView;

Make sure to first import the VerticalProgressView.h with this line.

#import "VerticalProgressView.h"

Go ahead and synthesize the two usage views the way you learned earlier. Now we need to update the custom views when we update the rest of the views. Add this code to updateViews to set up the usage views.

[self.leftUsageView setMinValue:[self getMinValue]];

[self.leftUsageView setMaxValue:[self getMaxValue]];

[self.leftUsageView addLabelAt:[self getWarningLevel]];

[self.leftUsageView addLabelAt:[self getAlertLevel]];

[self.rightUsageView setMinValue:[self getMinValue]];

[self.rightUsageView setMaxValue:[self getMaxValue]];

[self.rightUsageView addLabelAt:[self getWarningLevel]];

[self.rightUsageView addLabelAt:[self getAlertLevel]];

When the segmented control has selected Policy, update the views in the following way.

[self.leftUsageView setCurrentValue:[self getCurrentPolicyDownloadValue]];

[self.rightUsageView setCurrentValue:[self getCurrentPolicyUploadValue]];

And similarly, for the actual bandwidth, add the following.

[self.leftUsageView setCurrentValue:[self getCurrentActualDownloadValue]];

[self.rightUsageView setCurrentValue:[self getCurrentActualUploadValue]];

Go back into the storyboard and connect the custom views in the storyboard to the code. Now run it again!

## Changing the bar color for warnings

Add the following method that just checks where our value is compared to the warning and alert levels and returns green if you’re good, yellow if you’re warned, and red if you’re over the alert value.

- (UIColor \*)barColorForUsageValue:(CGFloat)valf {

if(valf > [self getAlertLevel]) {

return [UIColor redColor];

} else if(valf > [self getWarningLevel]) {

return [UIColor yellowColor];

} else {

return [UIColor greenColor];

}

}

We now need to update the usageviews barColor with the UIColor returned by this method. Add these two lines to your updateViews method to ensure that happens.

self.leftUsageView.barColor = [self barColorForUsageValue:[self.leftUsageView

currentValue]];

self.rightUsageView.barColor = [self barColorForUsageValue:[self.rightUsageView

currentValue]];

Question 3: We are using built in colors to create our colors. How would you make a custom color? What alternative method would you use if you wanted that custom color to be slightly transparent? (6 points)

## Congratulations

You’re done; run your app and check it out! It doesn’t do anything just yet – you’ll take care of that in the next lab. Don’t forget: submit answers to the **3** questions in this lab as a .pdf to the appropriate Moodle submission form.