Lab 01: Backgrounding in iOS

Prerequisites

You will need a development environment for iOS setup; either a Mac with XCode and Xamarin Studio or a Windows PC with Visual Studio and the Xamarin tools installed. Remember that if you are using Windows, you will still need an accompanying Mac on your network with XCode and the Xamarin tools to build and run the code.

See the Xamarin.iOS setup documentation if you need help getting your environment setup:

http://docs.xamarin.com/guides/ios/getting_started/installation/

Downloads

There are three projects in a single solution that we will use to explore the capabilities of iOS to run background tasks. You can download the projects from the Xamarin University website:

http://university.xamarin.com

Lab Goals

The goal of this lab will be to demonstrate the application lifecycle of an iOS application and then to use the built-in iOS features to run background tasks when the application is suspended. By completing this lab you will learn about:

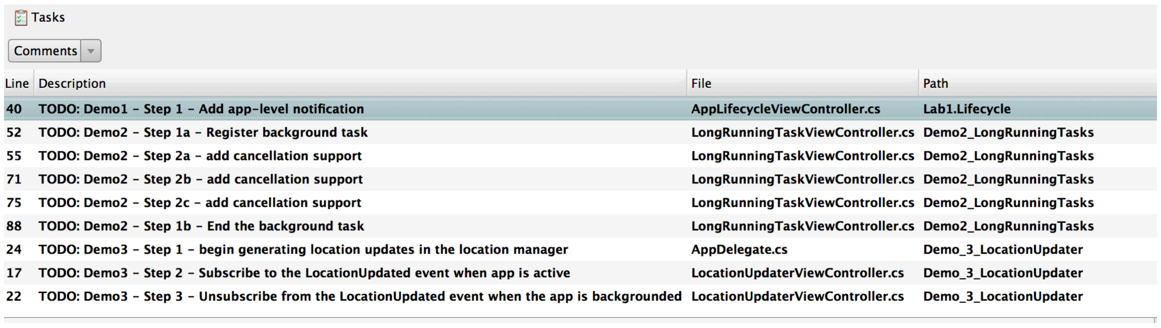
- The iOS Application lifecycle – how the system loads and unloads your application and what notifications you receive when this happens.
- Backgrounding capabilities – what types of work your code can do when it's not the currently running application. First, we'll take a look at registering a long-running task to run in the background. Then, we will see how to register an entire application for backgrounding privileges. Finally, we will walk through building and registering a Location application that gets continuous location updates in the background.

This will be accomplished through three pre-built projects that are provided in a single Backgrounding.iOS solution in the above GitHub repository. These projects are:

- Demo1_AppLifecycle – this will show you the various notifications that iOS sends your application as the user launches it, switches away and then switches back.
- Demo2_LongRunningTasks – this will demonstrate how to schedule tasks which continue to run both while your application is active, but also when the application moves to the background.

- Demo3_LocationUpdater – this final project will monitor the current location of the device while it runs and then report location changes when the application is moved to the background.

The lab has been provided as a starter solution with most of the code already filled in for you – as you following along with the instructor you will make small changes for each step, either writing a little code or uncommenting a block of code. Most of these steps are clearly marked in the supplied solution with `// TODO:` comments. These comments are picked up by Xamarin Studio and shown in the Task Pad, which you can make visible either by clicking the Tasks button in the status bar of the application, or through the **View > Pads > Tasks** menu item. When the Tasks Pad is open, it will look like this:



Line	Description	File	Path
40	TODO: Demo1 – Step 1 – Add app-level notification	AppLifecycleViewController.cs	Lab1.Lifecycle
52	TODO: Demo2 – Step 1a – Register background task	LongRunningTaskViewController.cs	Demo2_LongRunningTasks
55	TODO: Demo2 – Step 2a – add cancellation support	LongRunningTaskViewController.cs	Demo2_LongRunningTasks
71	TODO: Demo2 – Step 2b – add cancellation support	LongRunningTaskViewController.cs	Demo2_LongRunningTasks
75	TODO: Demo2 – Step 2c – add cancellation support	LongRunningTaskViewController.cs	Demo2_LongRunningTasks
88	TODO: Demo2 – Step 1b – End the background task	LongRunningTaskViewController.cs	Demo2_LongRunningTasks
24	TODO: Demo3 – Step 1 – begin generating location updates in the location manager	AppDelegate.cs	Demo_3_LocationUpdater
17	TODO: Demo3 – Step 2 – Subscribe to the LocationUpdated event when app is active	LocationUpdaterViewController.cs	Demo_3_LocationUpdater
22	TODO: Demo3 – Step 3 – Unsubscribe from the LocationUpdated event when the app is backgrounded	LocationUpdaterViewController.cs	Demo_3_LocationUpdater

You can quickly jump to the code by clicking on the task itself to keep up with the lecture as the instructor runs through this lab. If you need additional time to complete a task or need some help please let the instructor know – the goal is for you to work through this lab in the class itself.

Part 1 – App Lifecycle Steps

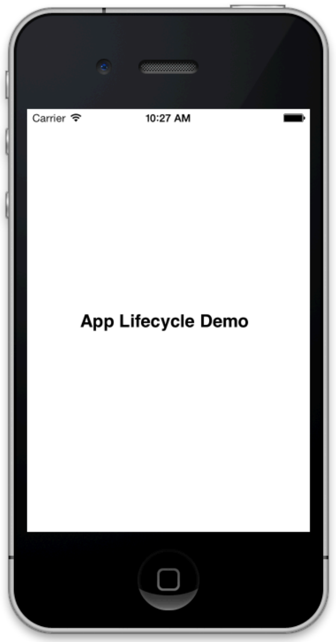
Open and Examine the Solution

1. Launch Xamarin Studio and open the **Backgrounding.iOS** solution file included with your lab resources.
2. Make sure the first project – **Demo1_AppLifecycle** is the active project, this is where we will start.
3. The code has `Console.WriteLine` statements sprinkled through it to demonstrate the launch sequence and activity notifications sent from iOS to your applications. Open the `AppDelegate` class and look for the logging methods that have been added into the lifecycle notifications from iOS:

```
public AppDelegate()
{
    Console.WriteLine("AppDelegate Constructor called.");
}

public override bool FinishedLaunching(UIApplication app, NSDictionary
 options)
{
    Console.WriteLine("FinishedLaunching called, creating screen");
    ...

    return true;
}

public override void OnActivated(UIApplication application)
{
    Console.WriteLine("OnActivated called, App is active.");
}

public override void WillEnterForeground(UIApplication application)
{
    Console.WriteLine("App will enter foreground");
}

public override void OnResignActivation(UIApplication application)
{
    Console.WriteLine("OnResignActivation called, App moving to inacti
ve state.");
}

public override void DidEnterBackground(UIApplication application)
{
    Console.WriteLine("App entering background state.");
}
```

Run the Application and See the State Transitions

4. Go ahead and run the application in the simulator. Make sure your Application Output window is open in Xamarin Studio (**View > Pads > Application Output**). The app will present a minimal UI with a single `UILabel`:



5. All the actual output is done by the `Console.WriteLine` statements – check the Output Window to see the sequence to bring the app to the Active state:

```
FinishedLaunching called, App is creating screen
AppLifecycleViewController constructed
ViewDidLoad called, View is ready to be shown.
OnActivated called, App is active.
```

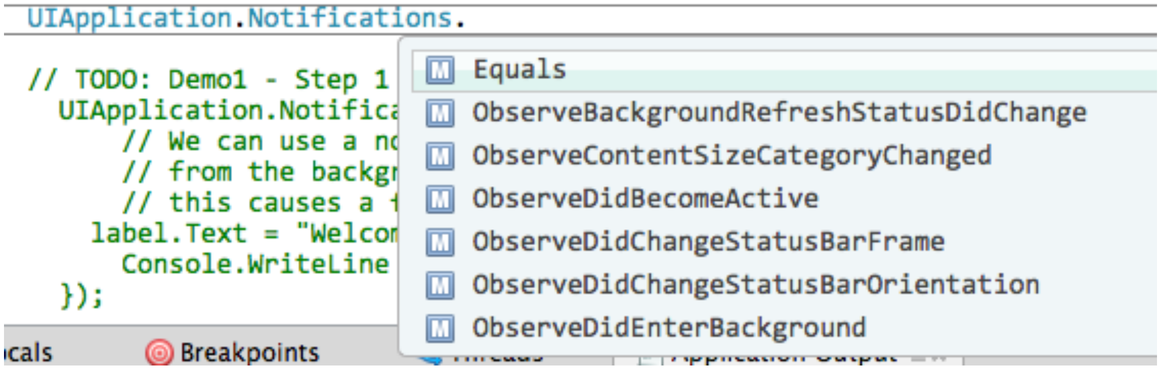
6. Hit the **Home** button on the simulator or device to move the application to the background – check the Output window to see the notifications you received as the app transitions from the Active state to the Inactive and then Background state.
7. Re-activate the application (you can either tap the icon or use the app switcher in iOS) to bring it back to the foreground and again watch the notifications to understand what notifications your app receives.

```
OnActivated called, App is active.
OnResignActivation called, App moving to inactive state.
App entering background state.
App will enter foreground
OnActivated called, App is active.
```

8. These are the notifications your `AppDelegate` will use to know what state the application is in – sometimes it's also convenient to receive these notifications in the active `ViewController`. You can do that by subscribing to application events in your View Controller. Locate the `TODO: Demo1 – Step 1` comment and uncomment the code that follows it:

```
// TODO: Demo1 - Step 1 - Add app-level notification
UIApplication.Notifications.ObserveWillEnterForeground ((sender, args)
 => {
    // We can use a notification to let us know when the app has
    // entered the foreground from the background, and update the
    // text in the View.  This causes a flicker, but we will use it
    // for demo purposes
    label.Text = "Welcome back!";
    Console.WriteLine ("ObserveWillEnterForeground called.");
});
```

9. Run the application again and go back through the steps to move it to the background and back to the foreground – notice you now see the `ObserveWillEnterForeground` notification just before the second `OnActivate` is received. You can use this same technique to subscribe to the other notifications as well:



```
UIApplication.Notifications.

// TODO: Demo1 - Step 1
  UIApplication.Notifica
      // We can use a no
      // from the backgr
      // this causes a t
    label.Text = "Welcom
      Console.WriteLine
  });
```

Equals
ObserveBackgroundRefreshStatusDidChange
ObserveContentSizeCategoryChanged
ObserveDidBecomeActive
ObserveDidChangeStatusBarFrame
ObserveDidChangeStatusBarOrientation
ObserveDidEnterBackground
cals
Breakpoints

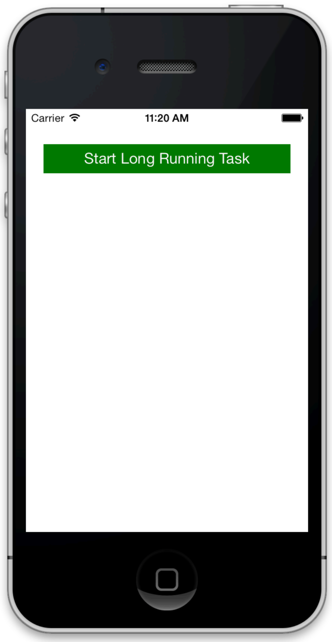
You can also add notifications into the `Main` entry point, `AppDelegate` constructor and `FinishedLaunching` method if you'd like to see the entire launch sequence played out. Just add the appropriate `Console.WriteLine` statements and run the application to see the flow.

Now that we've seen the iOS application states and transitions, let's look at the different options available for running background tasks in iOS.

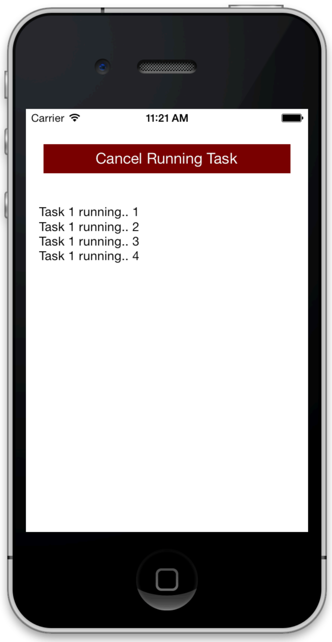
Part 2 - iOS Backgrounding Techniques

Registering a Long-Running Task

1. Switch to active project to Demo2_LongRunningTask.
2. Build & Run the application – it will display a simple UI with a button:



3. Click the button to start a long-running task – this is performed using the .NET Task Parallel Library (TPL). You should see output both on the screen and in the Output Window in Xamarin Studio.



```
Task 1 running.. 1
Task 1 running.. 2
Task 1 running.. 3
Task 1 running.. 4
```

4. Hit the **Home** button on the simulator or device to move the app to the background. Notice that the Output Window no longer displays the running task – this is because the app is now backgrounded, and by default receives no CPU resources in order to preserve the battery life of the device.

5. We can fix this behavior by registering a background task with iOS. Stop the app in Xamarin Studio and open **LongRunningTaskViewController.cs**.

   a. Examine the code which starts the task – there are a couple of interesting things here, in particular note the use of the new C# `async` / `await` keywords and the use of the `BeginInvokeOnMainThread` method which ensures we only update UI elements when on the main UI thread.

6. To register a background task, locate the comment:

   `// TODO: Demo2 - Step 1a - Register background task`

   And uncomment the line that directly follows it.

```
// TODO: Demo2 - Step 1a - Register background task
taskId = UIApplication.SharedApplication.BeginBackgroundTask(() =>{});
```

7. Next, scroll down a bit further in the code and locate the comment

   `// TODO: Demo2 - Step 1b - End the background task`

   Uncomment the line that directly follows it which will properly unregister the background task from iOS.

```
// TODO: Demo2 - Step 1b - End the background task
UIApplication.SharedApplication.EndBackgroundTask(taskId);
```

8. Run the application again and move the app to the background – this time you should see the output window continue to run the task, and if you activate the application again it should have continued running the task.

   When your app is running a task in the backgrounded state (i.e. when the app is not visible on the screen), the task is given a limited amount of time to complete (about 10 minutes, or 600 seconds in the current release of iOS). You can supply a delegate method to the `BeginBackgroundTask` function which will be called when the task times-out.

9. Locate the comment

   `// TODO: Demo2 – Step 2a – add cancellation support`

   Uncomment the lines directly underneath, and go ahead and comment out the original task registration (shown below with strikeout):

```
// TODO: Demo2 - Step 1a - Register background task
~~taskId = UIApplication.SharedApplication.BeginBackgroundTask(() =>{});~~

// TODO: Demo2 - Step 2a - add cancellation support
var cts = new CancellationTokenSource();
taskId = UIApplication.SharedApplication.BeginBackgroundTask("Long-
Running Task", () => {
    LogText("Task {0} timeout occurred, canceled.", taskId);
    cts.Cancel();
});
```

10. Next, add support to the task for cancellation – this is done in two parts, locate the comments for Step 2b and 2c and uncomment the code shown below:

```
try {
    await Task.Run(() => {
        for (long count = 1; running == true ; count++) {

            this.BeginInvokeOnMainThread(() => {
                LogText("Task {0} running.. {1}", taskId, count);
            });

            // TODO: Demo2 - Step 2b - add cancellation support
            cts.Token.ThrowIfCancellationRequested();
            Thread.Sleep(1000);
        }
    // TODO: Demo2 - Step 2c - add cancellation support
    }, cts.Token);
```

11. If you have time, try running the app on a device or in the simulator and let it run in the background for about 10 minutes to see the cancellation code take effect – you should see a "Timeout" message in the output window.

In the next section, we will look at an example that registers a Location application, and runs location updates in the background by hooking into a system location service.

Registering as a Background Necessary Application

1. Switch the active project to the final project in the solution – Demo3_LocationUpdater.
2. Open up the **LocationManager.cs** source file and examine the usage of the iOS CoreLocation `CLLocationManager` class. This project has a wrapper around it to make it a little easier to use. It also abstracts away the difference between iOS5 and iOS6 by checking to see what version of iOS we are running on and wiring up to the proper event. This is an example of allowing your app to use the newest features, while still running on older devices.
   a. The key points here are that the class raises a `LocationUpdated` event when it detects a location change and passes you a `LocationUpdatedEventArgs` object (defined in the source file of the same name).
3. As a first step, let's create and activate the location manager class. Open the `AppDelegate` class and locate the comment in the `FinishedLaunching` method

   `// TODO: Demo3 - Step 1 - begin generating location updates`

   Uncomment the two lines that follow it.

```
public override void FinishedLaunching(UIApplication application)
{
    // TODO: Demo3 - Step 1 - begin generating location updates
    Manager = new LocationManager();
    Manager.StartLocationUpdates();
}
```

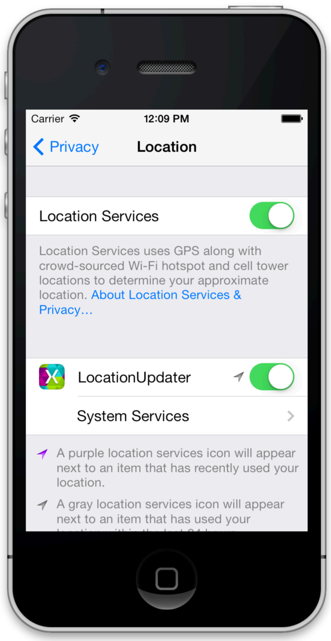
4. Next, open the `LocationUpdaterViewController` class and find the comment

   `// TODO: Demo3 - Step 2 - Subscribe to the LocationUpdated event when app is active`

   Uncomment the block that follow it to hook the ObservedDidBecomActive event from the application and hook into the LocationUpdated event.

```
// TODO: Demo3 - Step 2 -
 Subscribe to the LocationUpdated event when app is active
AppDelegate.Manager.LocationUpdated += HandleLocationChanged;
```

5. Run the application – it should ask if it's ok to use your location, make sure to allow access or the app won't be able to read the location data. If you accidentally cancel this you can always turn on location services for the lab application in **Settings > Privacy > Location**:



Location Services

Location Services uses GPS along with crowd-sourced Wi-Fi hotspot and cell tower locations to determine your approximate location. About Location Services & Privacy…

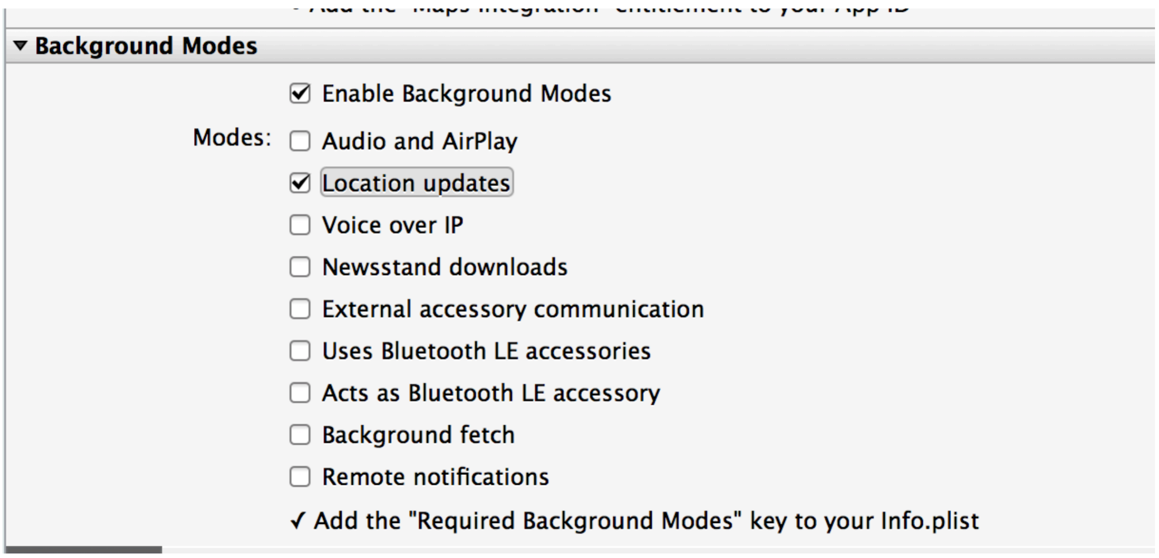
LocationUpdater

System Services

A purple location services icon will appear next to an item that has recently used your location.

A gray location services icon will appear next to an item that has used your

6. When the app executes, it should display a location – even in the simulator. You can change the location through the **Debug > Location** menu on the simulator. If you choose one of the active locations (driving, running, biking) then the application will print location updates while it is Active. You should see the same updates occurring in the output window – our `LocationManager` class is printing these.
7. Click the **Home** button to move the app to the background and notice the updates stop – just as our background task did in the last project.
8. We don't need to alter any code to fix this – instead we just need to tell iOS that we want to receive location updates in the background. We do this through the **info.plist**. Open the **info.plist** file and scroll towards the bottom in the GUI editor. You should see a section labeled **Background Modes**. Check the **Enable Background Modes** box and then the **Location Updates** box.



Background Modes

- [✓] Enable Background Modes
- Modes:
  - [ ] Audio and AirPlay
  - [✓] Location updates
  - [ ] Voice over IP
  - [ ] Newsstand downloads
  - [ ] External accessory communication
  - [ ] Uses Bluetooth LE accessories
  - [ ] Acts as Bluetooth LE accessory
  - [ ] Background fetch
  - [ ] Remote notifications

✓ Add the "Required Background Modes" key to your Info.plist

9. Run the application and verify that it is indeed continuing to run in the background. However, notice that the UI is still being updated (at least the console says it is).

   While this works on current versions of iOS, it used to fail and at a minimum is inefficient because we are trying to update objects that aren't visually present on the screen. To fix this, let's subscribe to some application-level events to let us know when the app is moving to the background and when it's moving back to the foreground.

10. Open the `LocationIUpdaterViewController` again and find the `TODO Step 3a` and `3b`. Uncomment the lines associated with them, and go ahead and comment out the original subscription to the `LocationUpdated` event just above this. The code should look something like this:

```
public override void ViewDidLoad()
{
    base.ViewDidLoad();

    // TODO: Demo 3 - Step 2 - Subscribe to the LocationUpdated event
    // AppDelegate.Manager.LocationUpdated += HandleLocationChanged;

    // TODO: Demo3 - Step 3a -
    // Subscribe to the LocationUpdated event when app is active
    UIApplication.Notifications.ObserveDidBecomeActive ((sender, e) =>
    {
        AppDelegate.Manager.LocationUpdated += HandleLocationChanged;
    });

    // TODO: Demo3 - Step 3b -
    // Unsubscribe from the LocationUpdated event when the app
    // is backgrounded
    UIApplication.Notifications.ObserveDidEnterBackground ((sender, e)
 => {
        AppDelegate.Manager.LocationUpdated -= HandleLocationChanged;
    });
}
```

11. Run the app a final time and make sure the UI is no longer updated when the app is backgrounded.

Summary

In this lab, we have explored the application lifecycle of an iOS application and built an application, which continues to execute code when the app is backgrounded, and not the active application on the screen.