Lab 02: Backgrounding in Android

Prerequisites

You will need a development environment for Android setup; either a Mac or PC with Xamarin Studio or a Windows PC with Visual Studio and the Xamarin tools installed.

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

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

Downloads

There are three projects in a single solution that we will use to explore the capabilities of Android to run background tasks. All of these can be found in the included folder:

Content/Exercises/BackgroundAndroid/Lab Resources

Lab Goals

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

- The Android 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.Android** solution. The three 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_HelloService – 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.Android** 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 `Log.Debug` statements sprinkled through it to demonstrate the launch sequence and activity notifications sent from Android to your applications. Open the `MainActivity` class and look for the logging methods that have been added into the lifecycle notifications from Android:

```
protected override void OnCreate(Bundle bundle)
{
    Log.Debug(logTag, "OnCreate called, View being created");
    base.OnCreate(bundle);

    // Set our view from the "main" layout resource
    SetContentView(Resource.Layout.Main);
}

protected override void OnStart()
{
    Log.Debug (logTag, "OnStart called, App is Active");
```

```
        base.OnStart();
    }

    protected override void OnResume()
    {
        Log.Debug (logTag, "OnResume called, app is ready to interact with
     the user");
        base.OnResume();
    }

    protected override void OnPause()
    {
        Log.Debug (logTag, "OnPause called, App is moving to background");
        base.OnPause();
    }

    protected override void OnStop()
    {
        Log.Debug (logTag, "OnStop called, App is in the background");
        base.OnStop();
    }

    protected override void OnDestroy ()
    {
        base.OnDestroy ();
        Log.Debug (logTag, "OnDestroy called, App is Terminating");
    }
```
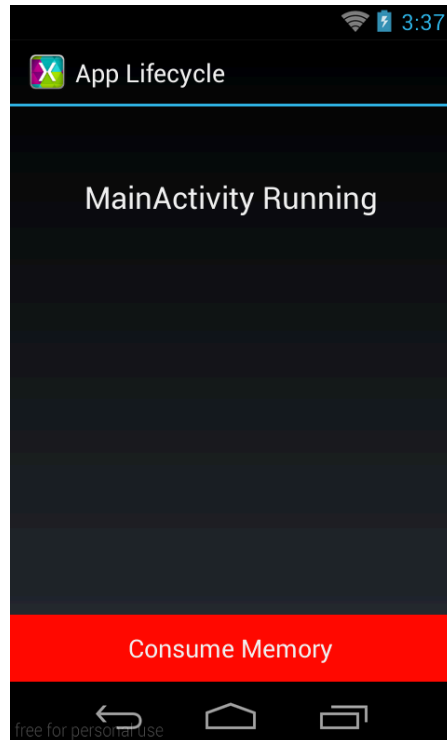
Run the Application and See the State Transitions

4. Go ahead and run the application in the emulator. 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 `Button` and
    `TextView`:

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

```
[MainActivity] OnCreate called, View being created

[MainActivity] OnStart called, App is Active

[MainActivity] OnResume called, app is ready to interact with the user
```

> **Note**: you will see a lot more information in the Output Window – look for the prefix **[MainActivity]** to narrow down the output to what is being shown here in the lab.

6. Hit the **Home** button on the emulator 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 Background state.

7. Re-activate the application (you can either tap the icon or use the app switcher) to bring it back to the foreground and again watch the notifications to understand what notifications your app receives.

```
[MainActivity] OnResume called, app is ready to interact with the user

[MainActivity] OnPause called, App is moving to background

[MainActivity] OnStop called, App is in the background

[MainActivity] OnStart called, App is Active

[MainActivity] OnResume called, app is ready to interact with the user
```

8. These are the notifications your `Activity` will use to know what state the application is in.

Low Memory Conditions

Next, We're going to explore a scenario where `OnPause` is called without being followed by `OnStop`. This can happen in low-memory situations, when Android needs to destroy the Activity quickly.

1. Open up the Activity called **MemoryEaterActivity**. Here, we have added a memory-consuming operation to `OnStart`:

```csharp
protected override void OnStart()
{
    Log.Debug("MemoryEater", "OnStart called - allocating memory.");

    base.OnStart();
    var ii = new int[1000000000];
}
```

2. Run the application in the emulator or on a device, and hit the "Consume Memory" button. You should see `OnPause` getting called in the **MainActivity**, and `OnCreate` getting called in the **MemoryEaterActivity** before the application starts experiencing memory problems. Notice that `OnStop` is never called:

```
[MainActivity] OnPause called, App is moving to background

[MemoryEater] OnCreate called

[MemoryEater] OnStart called - allocating memory.

[MonoDroid] UNHANDLED EXCEPTION: System.OutOfMemoryException: Out of
memory
```

In addition, if you continue the exception, the Activity will now be terminated abruptly – Android might even report that the activity has stopped responding.

Starting Background threads in Android

1. Let's add some background work using a .NET Task – locate the comment

```
// TODO: Demo1 – Step 1 – Start a background task
```

And uncomment the code that follows it:

```csharp
// TODO: Demo1 – Step 1 – Start a background task
Task.Run(() => {

    for ( ;; )    {
        Log.Debug(logTag, "Bad thread running..");
        Thread.Sleep(500);
    }

});
```

2. This is a bad piece of code, for a variety of reasons, but go ahead and build and run the application.

3. Hit the **Home** button to background the activity and watch the output window. Notice that the task continues to run – even though the activity is backgrounded!

   This is actually bad behavior – when Android calls the `OnStop` method, it expects that your activity will stop running; the View is removed and the activity will not receive any more notifications unless it moves back to the foreground.

   While this does work, it goes against the design of the Android Activity model. Your code should be stopping the task when the Activity is backgrounded – feel free to add in cancellation just like we used in the iOS lab if you'd like.

   Next, let's look at how we *should* do long-running work which outlives the Activity – with *Services*.

Part 2 – Android Services

Starting a Service

1. Make the project **to Demo2_HelloService** active.

2. Our first step is to add a new service into the project. This is done by creating a class derived from `Service` and decorating it with the `[Service]` attribute. This code has been written for you: open the file **TheService.cs** and uncomment the class contained within it. There is a comment marker for this `// TODO: – Demo2 – Step 1.`

3. Next, we need to start the service. Locate the comment

   `// TODO: Demo2 – Step 2 – Start the service`

4. Uncomment the code that follows it that adds a button to the UI:

```
// TODO: Demo2 - Step 2 - Start the service
Button startButton = new Button(this) {
    Text = "Start the Service",
    LayoutParameters = new LinearLayout.LayoutParams(
        ViewGroup.LayoutParams.MatchParent,
        ViewGroup.LayoutParams.WrapContent),
};
rootLayout.AddView(startButton);

startButton.Click += delegate {
    StartService(new Intent(this, typeof(TheService)));
};
```

5. Build & Run the application – it will display a simple UI with a button:
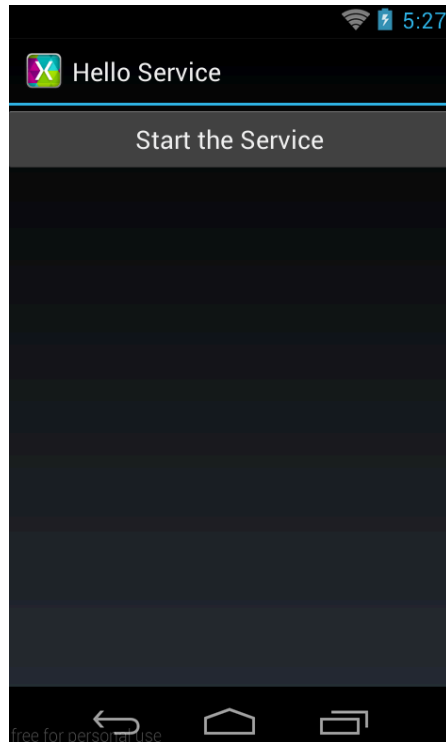
6. Click the button to start the service – it will show a brief Toast notification and then you will begin to see output in the Output window of Xamarin Studio indicating the service is running.

7. Click the **Home** button to background the. Notice in the Output window that the service continues running independent of the activity – in this case, it will continue running forever (or at least until Android kills the process).

8. Go ahead and stop the debugger.

9. Most of the time you won't have services that execute forever – instead, they will perform some unit of work and then stop on their own. Let's add that support to this service. Locate the comments:

    ```
    // TODO Demo2 – Step 3a – Stop the service when work is complete
    // TODO Demo2 – Step 3b – Tell Android we are done.
    ```

10. And uncomment the line that follows each one, also make sure to comment out the original **for** instruction which we are replacing (indicated in strikeout below) – your code should look like this:

```
Task.Run(() => {
    //for (long index = 1; ; index++) {

    // TODO: Demo2 - Step 3a - Stop the service when work is complete
    for (long index = 1; index < 15; index++) {

    // TODO: Demo2 - Step 4a - Stop the service when requested
// for (long index = 1; isRunning && index < 15; index++) {

        Thread.Sleep(1000);
```

```
        Log.Debug(logTag, "[{0}] Service running - {1}",
                    startId, index);
    }
    Log.Debug(logTag, "Service {0} stopping - {1}", startId,
            isRunning ? "Work complete" : "OnDestroy");

    if (isRunning) {
        // TODO: Demo2 - Step 3b - Tell Android we are done.
        StopSelf();
    }
```

11. Build and run the activity again, this time, after 15 seconds, it will stop on it's own and you should see the service get destroyed.

```
[TheService] Service constructed

[TheService] Service OnCreate

[TheService] Service OnStartCommand - 1

[TheService] [1] Service running - 1

[TheService] [1] Service running - 2

[TheService] [1] Service running - 3

[TheService] [1] Service running - 4

[TheService] [1] Service running - 5

[TheService] [1] Service running - 6

[TheService] [1] Service running - 7

[TheService] [1] Service running - 8

[TheService] [1] Service running - 9

[TheService] [1] Service running - 10

[TheService] [1] Service running - 11

[TheService] [1] Service running - 12

[TheService] [1] Service running – 13

[TheService] [1] Service running - 14

[TheService] Service 1 stopping – Work complete

[TheService] Service Destroyed.
```

12. Stop the app in the debugger and let's add some code to stop the service programmatically. We can do that by calling the StopService method. Locate the comment

    ```
    // TODO: Demo3 – Step 4a – Request the service stop
    ```

    And uncomment the block of code to create a second button.

13. Build and run the app again – start the service and then click the new button to ask it to stop. Notice that the service ignores the request and continues to the end of the 15 seconds. That's because the service is not currently honoring the stop notification.

```
[TheService] [1] Service running - 1
[TheService] Service Destroyed.
[TheService] [1] Service running - 2
[TheService] [1] Service running - 3
[TheService] [1] Service running - 4
...
[TheService] [1] Service running - 14
[TheService] Service 1 stopping - OnDestroy
```

14. Stop the app and let's fix that.

15. Locate the comment:

    ```
    // TODO: Demo2 – Step 4b – Stop the service when requested
    ```

    And uncomment the line that follows – make sure to comment out the prior version of the **for** statement (shown in strikeout below):

```
Task.Run(() => {
   //for (long index = 1; ; index++) {

   // TODO: Demo2 - Step 3a - Stop the service when work is complete
//    for (long index = 1; index < 15; index++) {

   // TODO: Demo2 - Step 4a - Stop the service when requested
   for (long index = 1; isRunning && index < 15; index++) {
```

16. Build and run the application a final time – start and stop the service and note that we now see the service properly stops when requested:

```
 [TheService] Service constructed
[TheService] Service OnCreate
[TheService] Service OnStartCommand - 1
[TheService] [1] Service running - 1
[TheService] [1] Service running - 2
[TheService] [1] Service running - 3
[TheService] Service Destroyed.
[TheService] [1] Service running - 4
[TheService] Service 1 stopped – OnDestroy
```

Binding to a Service

In this final project we will be using a location service to track our current location. In this instance, we will be binding to a service that the application itself is also starting, but the same concepts would apply if we were binding to a service exposed by another application in a different process. The process of connecting to a Bound Service is as follows:

1. Create an object that implements `IServiceConnection` and wire up the `ServiceConnected` event on it.

2. Create an `Intent` that contains the `Context` of the client (such as an Activity), and the `Type` of the Service to be connected to.

3. Call `BindService` on a `Context` and pass the Intent. Android will then locate the Service, create it, and call `OnBind` on it.

The following diagram illustrates how the caller uses a `Binder` and `ServiceConnection` to communicate with a Service running locally:
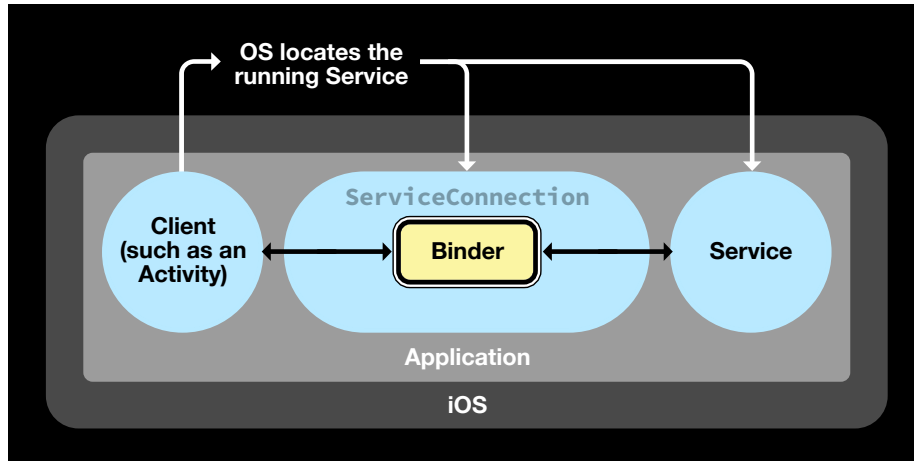
The Service must override the *OnBind* method and return an `IBinder` object that contains a reference to the Service. Once the `OnBind` method returns, Android will raise the `ServiceConnected` event on the `IServiceConnection` object and the client can then reference the Service via the `Binder` that is owned by the `IServiceConnnection`.

Let's see how this all works by building the location service.

1. Switch to the Demo3_LocationUpdater project in the solution.

2. The service has already been created – it's located in the **LocationService.cs** source file, go ahead and open the file and look through the code – it should look fairly familiar.

    There's some new code in this service that uses the built-in Android `LocationManager` class and also implements the `ILocationListener`, which is used in tandem with the manager to report location changes. This is used in the `OnStartCommand` and `OnDestroy` notifications.

3. Typically, the service will create an IBinder interface to be able to expose itself to the system. This needs to be returned by the OnBind method override. Let's create the binder first – open the **LocationServiceBinder.cs** file (or locate the comment `TODO: Demo3 – Step 1`) and uncomment the class it contains.

4. Next, locate the comment `// TODO: Demo3 – Step 2 – Return a binder implementation` and uncomment the creation of the binder:

```
public override IBinder OnBind(Intent intent)
{
    // TODO: Demo3 - Step 2 - Return a binder implementation
    binder = new LocationServiceBinder(this);
    return binder;
}
```

5. Now, we need to create an `IServiceConnection` for our local application to bind to this new service. This will be used by Android to locate the running service and attach the binder to it. Locate the comment `// TODO: Demo3 – Step 3 – Create an IServiceConnection to locate and bind to the service` and uncomment the class that follows.

6. Next, we need to provide some access to this service connection for our application to use it. A common approach is to use a *singleton* to access the global data. Open the **App.cs** file – in here you will find two TODO elements:

   `// TODO: Demo3 – Step 4 – Provide access to bound service`

   `// TODO: Demo3 – Step 5 – Bind to the service`

   Go ahead and locate both and uncomment the associated code. The code should look something like this when you are finished:

```
public class App
{
    static Lazy<App> app = new Lazy<App>(() => new App());

    public static App Current
    {
        get { return app.Value; }
    }

    // TODO: Demo3 - Step 4 - Provide access to bound service
    LocationServiceConnection lsConnection;
    public LocationService LocationService
    {
        get { return lsConnection.Service;       }
    }

    public event EventHandler<ServiceConnectionEventArgs>
                                ConnectionChanged = delegate{};

    private App()
    {
        Task.Run(() => {
            var context = Application.Context;

            // Start the service - since it's in our process.
            var intent = new Intent(context, typeof(LocationService));
            context.StartService(intent);

            // TODO: Demo3 - Step 5 - Bind to the service
            lsConnection = new LocationServiceConnection();
            lsConnection.ConnectionChanged += (s,e) =>
                        ConnectionChanged(this, e);
```

```
                context.BindService(intent, lsConnection, Bind.AutoCreate);
        });
    }
}
```

The code to use the service is already in **MainActivity.cs** – it will hook the **ConnectionChanged** event exposed by our global App object and when we are connected to the service, it will be notified about location changes and update the UI accordingly. When the activity is backgrounded, it will unsubscribe from the notification – but the **LocationService** will continue to output location changes in the Output Window.

7. Build and run this project – use the Genymotion emulator so you can control the location right from the UI (otherwise you need to use the Android Device Monitor or a telnet session to change the location).

8. Once it's running, click the GPS button on the toolbar and activate location changes.
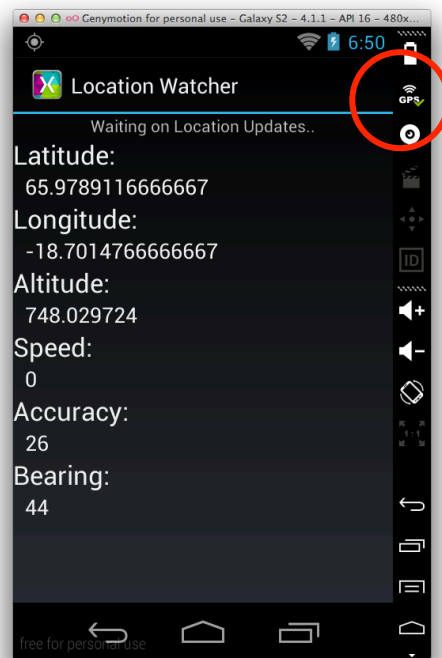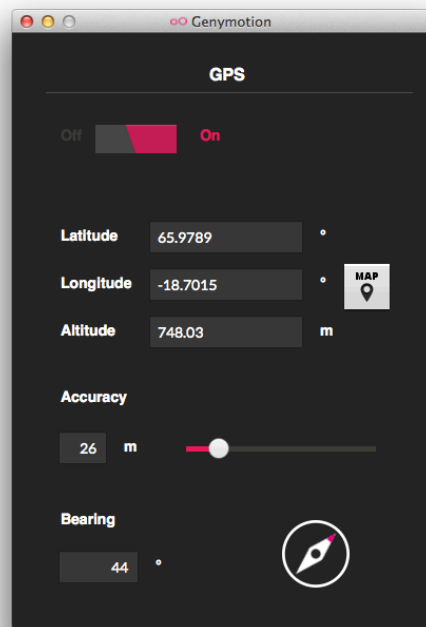
9. In the output window you should see the UI being updated continuously (even if the values don't change)

```
[Location Watcher] OnLocationChanged - UI being updated.
[LocationService] OnLocationChanged – Service updated.
[Location Watcher] OnLocationChanged - UI being updated.
[LocationService] OnLocationChanged – Service updated.
[Location Watcher] OnLocationChanged - UI being updated.
```

10. Hit the **Home** button to background the activity – the UI updates should stop, but the location service should continue updating in the Output window.

11. Reactivate the app – the UI updates should resume.

Summary

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