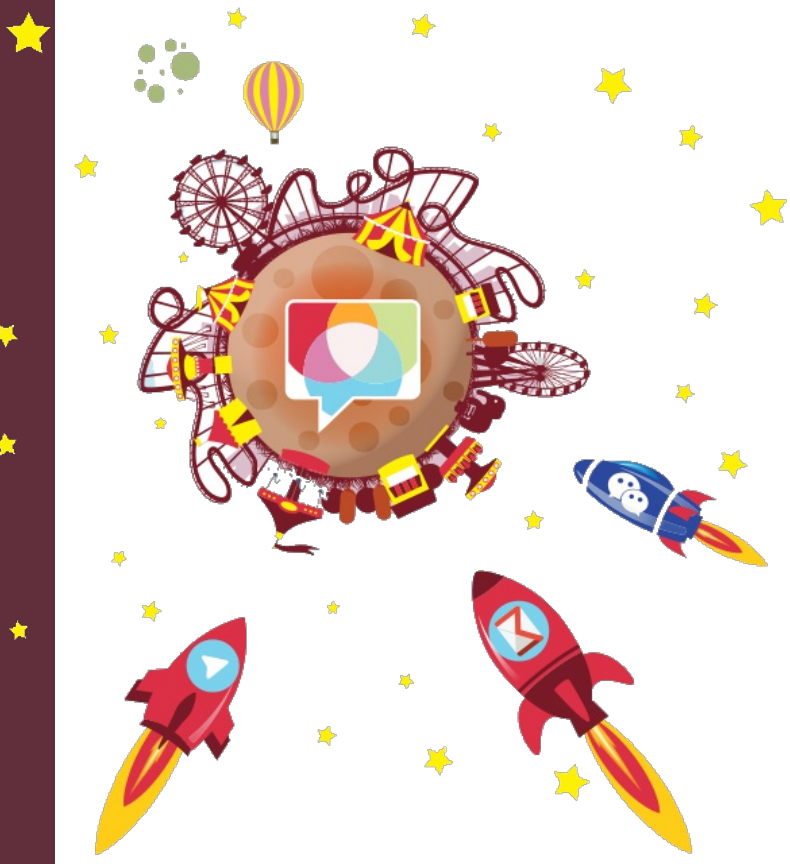
Disa

DEVELOPER GUIDE



contents

Disa Framework Developer Guide

Add your introductions here!

3 Your First Plugin

Learn how to set up your IDE for Disa plugins.

Also get to know the basic structures of a Disa plugin and how to use them yourself.

The Most Basic Plugin

Abstract

If you ventured into here by now, you're probably wanting to know how to start building a plugin.

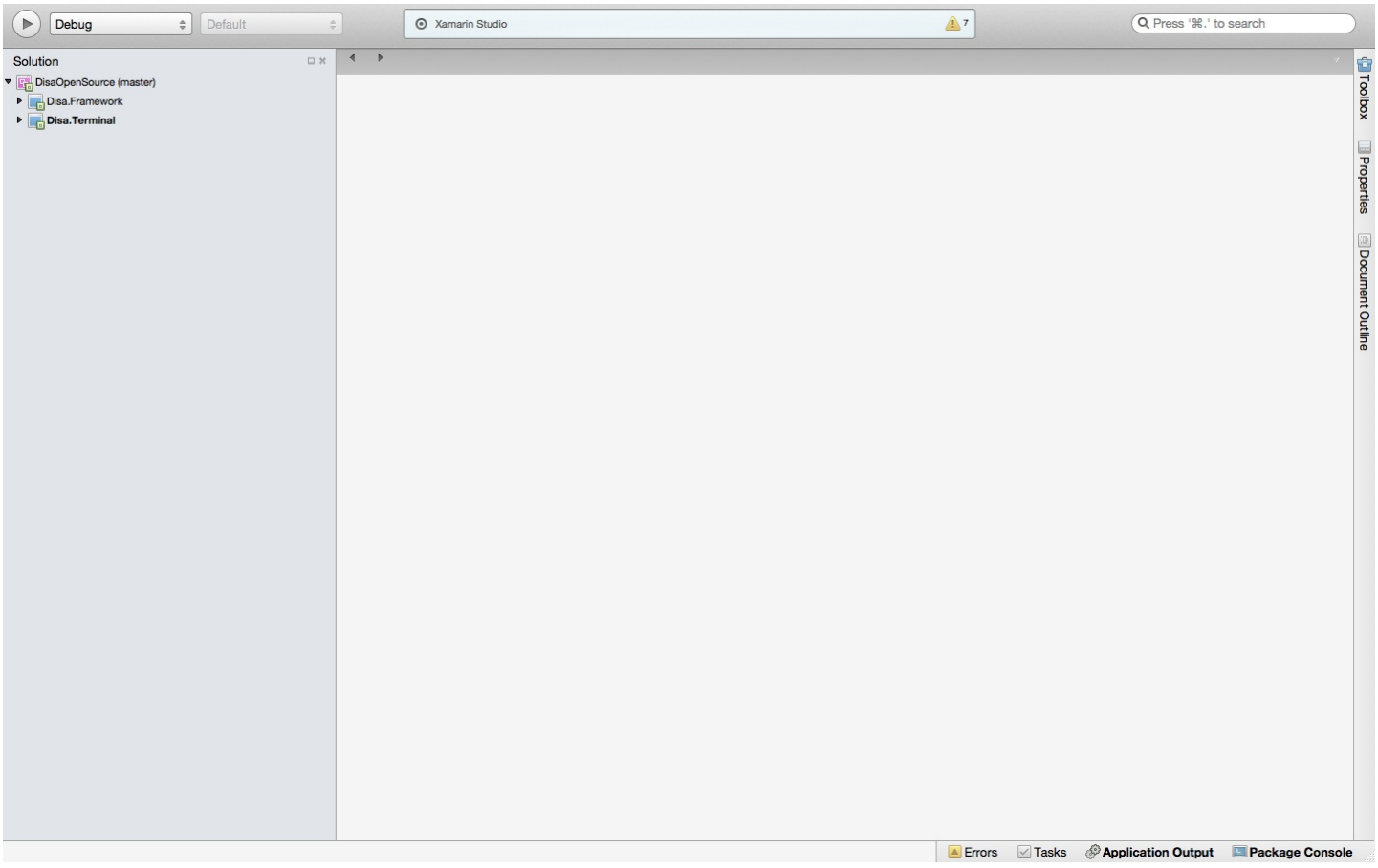
This tutorial will guide you into building a very basic plugin for WackyMessenger (a made-up service). WackyMessenger is incredibly basic - it only supports text messages. When a user sends a text message, it waits a few seconds, and then responds to the user with his or her message reversed.

The source to this plugin can be found in full under the Examples/TheMostBasicPlugin folder in the main directory.

Setting Up Your IDE

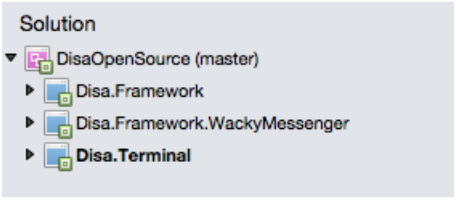
Alright, to begin! You need to setup your IDE. We'll be using Xamarin Studio (as I am on a Mac). However, Visual Studio works just fine too and you should be able to follow along easily. If not, send us an email at opensource@disa.im and we'll figure it out.

First, clone or download this repo. Then, open it. You should be presented with a screen similar to this:

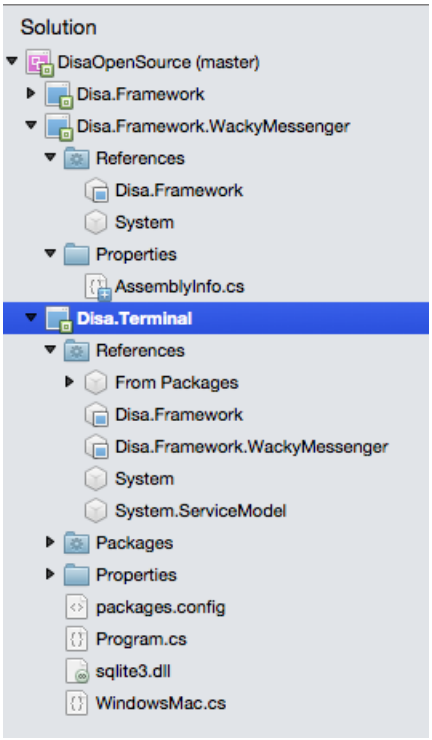


Now, lets add our WackyMessenger project. Add a new Library project (not the PCL one!) by choosing .NET from "Other", then choosing "Library", and calling it Disa.Framework.WackyMessenger.

Aside: for plugins to properly work, they need to be labelled with the Disa.Framework prefix. If for example, I was writing a Telegram plugin, I'd label the plugin project Disa.Framework.Telegram. There will now be three projects in your solution:



Go ahead, and add Disa.Framework as a reference to Disa.Framework.WackyMessenger. Then, add Disa.Framework.WackyMessenger as a reference to Disa.Terminal. You'll now be left with the following:



Wonderful! Now we're ready to add the service skeleton.

Add the Service Skeleton

Create a new file in Disa.Framework.WackyMessenger, calling it WackyMessenger.cs.

Paste the following into the file:

```
using System;
using System.Threading.Tasks;
using System.Collections.Generic;
using Disa.Framework.Bubbles;

namespace Disa.Framework.WackyMessenger
{
    [ServiceInfo("WackyMessenger", true, false, false, false, false, typeof(WackyMessengerSettings),
        ServiceInfo.ProcedureType.ConnectAuthenticate, typeof(TextBubble))]
    public class WackyMessenger : Service
    {
        public override bool Initialize(DisaSettings settings)
        {
            throw new NotImplementedException();
        }
```

```
        public override bool InitializeDefault()
        {
            throw new NotImplementedException();
        }

        public override bool Authenticate(WakeLock wakeLock)
        {
            throw new NotImplementedException();
        }

        public override void Deauthenticate()
        {
            throw new NotImplementedException();
        }

        public override void Connect(WakeLock wakeLock)
        {
            throw new NotImplementedException();
        }

        public override void Disconnect()
        {
            throw new NotImplementedException();
        }

        public override string GetIcon(bool large)
        {
            throw new NotImplementedException();
        }

        public override IEnumerable<Bubble> ProcessBubbles()
        {
            throw new NotImplementedException();
        }

        public override void SendBubble(Bubble b)
        {
            throw new NotImplementedException();
        }

        public override bool BubbleGroupComparer(string first, string second)
        {
            throw new NotImplementedException();
        }

        public override Task GetBubbleGroupLegibleId(BubbleGroup group, Action<string> result)
        {
            throw new NotImplementedException();
        }

        public override Task GetBubbleGroupName(BubbleGroup group, Action<string> result)
        {
            throw new NotImplementedException();
        }

        public override Task GetBubbleGroupPhoto(BubbleGroup group, Action<DisaThumbnail> result)
        {
            throw new NotImplementedException();
        }

        public override Task GetBubbleGroupPartyParticipants(BubbleGroup group, Action<DisaParticipant[]>
result)
        {
            throw new NotImplementedException();
        }
```

```
        public override Task GetBubbleGroupUnknownPartyParticipant(BubbleGroup group, string
unknownPartyParticipant, Action<DisaParticipant> result)
        {
            throw new NotImplementedException();
        }

        public override Task GetBubbleGroupPartyParticipantPhoto(DisaParticipant participant,
Action<DisaThumbnail> result)
        {
            throw new NotImplementedException();
        }

        public override Task GetBubbleGroupLastOnline(BubbleGroup group, Action<long> result)
        {
            throw new NotImplementedException();
        }
    }

    public class WackyMessengerSettings : DisaSettings
    {
        // store settings in here:
        // e.g: public string Username { get; set; }
    }
}
```

At the very top of the Service class, we specify its information (i.e: how the framework must manage it).

```
[ServiceInfo("WackyMessenger", true, false, false, false, false, typeof(WackyMessengerSettings),
ServiceInfo.ProcedureType.ConnectAuthenticate, typeof(TextBubble))]
```

We will be using event driven bubbles. What does this mean? Some services require a dedicated thread to be infinitely polling against a keep-alive connection. By setting event driven bubbles to false, the ProcessBubbles iterator block is called in an infinite threaded loop while the service is running. Thus, the Framework completely manages this aspect of keeping the poller constantly alive. By setting event driven bubbles to true, you are effectively telling the Framework: "I want to manage all the polling myself, and invoke off the EventBubble method whenever I a new bubble comes in."

We will not be using media progress. We don't support anything but text bubbles. If your service can support giving feedback back to the client on the upload process of media bubbles (images, videos, etc), you'll set this flag to true and then use the Transfer.Progress callback in the associated media bubble you're uploading.

This service does not use internet. If it we set this to true, then the Framework will ensure that the service is stopped if there is no internet connection.

This service does not support battery savings mode.

This service does not use delayed notifications. Delayed notifications will delay notification dispatches by 1 (one) second. Setting this to true and using NotificationManager.Remove allows you to have multiple clients working together without notifications going off while chatting on another client.

The Framework manages a settings store of your service. You can use this to store any information. WackyMessenger doesn't need to store anything, so we don't need it. Additionally, you can use MutableSettings and MutableSettingsManager to save information you find yourself frequently saving (such as a timestamp you need to keep updated everytime the service is started).

The procedure type is set to ConnectAuthenticate. This means that the service scheduler will call Connect before calling Authenticate. The other option is AuthenticateConnect - which called Authenticate before Connect. Once again, the option is given here because some services require Authenticating before connecting, and vice-versa.

Finally, the last argument is a params[] of all the supported bubble types. Since WackyMessenger only supports Text bubbles, that's the only bubble we list there.

It also should be mentioned that if you use files, audio, or video bubbles in your plugin, you need to add the associated attribute. These can be found in ServiceInfo.cs.

Great! Now let's talk a bit about the starting of your newly implemented service.

Service Start Process

The first thing that happens is that InitializeDefault() is called. It attempts to try and start the service without any settings. If this method returns true, it is assumed your service doesn't need any settings. Authenticate and Connect is then directly called afterwards (the order of which one is first depends on your set procedure type, as mentioned above). If this method returns false, then InitializeDefault(DisaSettings) is called - the framework provided you with your stored settings. Whenever you want to save your settings, you can call SettingsManager.Save.

So, for WackyMessenger, all we need to do is initialize the default - we are not using Settings.

```
public override bool InitializeDefault()
{
     return true;
}
```

For connect, WackyMessenger doesn't really connect to anything. So, we can just leave it blank:

```
public override void Connect(WakeLock wakeLock)
{
   // do nothing
}
```

As with Authenticate:

```
public override bool Authenticate(WakeLock wakeLock)
{
   return true;
}
```

What exactly is that WakeLock? Whenever Disa executes one of these methods, it holds a wake lock on the method's duration so the phone doesn't fall asleep and stop your service's starting process. However, wake locks are expensive to battery life. If you know that you can temporarily free the wakelock (such as when you're awaiting for a response from a server), you can use WakeLock.TemporaryFree disposable (wrap it in a using statement) to do so.

Aside: Whenever you are awaiting data from a socket connection (including HTTP connections) in Android, you can allow the device to fall asleep. When there's a response from the socket, your device will be woken back up, allowing the newly presented data to be processed. This is the motivation behind the temporary free optimization. In the event that Authenticate or Connect doesn't succeed, and can just pass up the exception to the Framework. It will deal with a connection failure or authentication failure accordingly. However, there are some exceptions to let the Framework know that there is something particularly wrong. For example, passing a ServiceExpiredException in one of these methods will let the Framework know that the service has expired (subscription needs to be repaid for example). For more exceptions, take a look at all the defined *Exceptions in the framework, and their mappings into the service scheduler.

Service Stop Process

When a service is stopped, two methods are called: Deauthenticate and Disconnect. The order once again depends on the procedure type. Use these methods to teardown your service.

In WackyMessenger, we can simply ignore them, as there's nothing to teardown:

```
public override void Deauthenticate()
{
    // do nothing
}

public override void Disconnect()
{
    // do nothing
}
```

Interim Summary

Your code should now look like this:

```
.
.
.
public override bool InitializeDefault()
{
    return true;
}

public override bool Authenticate(WakeLock wakeLock)
{
    return true;
}

public override void Deauthenticate()
{
    // do nothing
}

public override void Connect(WakeLock wakeLock)
{
    // do nothing
}

public override void Disconnect()
{
    // do nothing
}
.
.
.
```

Implementing bubble sending

Great. So now, the service will both start and stop. Its pretty damn useless though. Lets add some sending:

```
public override void SendBubble(Bubble b)
{
    var textBubble = b as TextBubble;
    if (textBubble != null)
    {
     Utils.Delay(2000).Wait();
     Platform.ScheduleAction(1, new WakeLockBalancer.ActionObject(() = >
        {
            EventBubble(new TextBubble(Time.GetNowUnixTimestamp(), Bubble.BubbleDirection.Incoming,
        textBubble.Address, null, false, this, Reverse(textBubble.Message)));
        }, WakeLockBalancer.ActionObject.ExecuteType.TaskWithWakeLock));
    }
}

private static string Reverse(string s)
{
    char[] charArray = s.ToCharArray();
    Array.Reverse(charArray);
    return new string(charArray);
}
```

Alright. So what's happening here? We wait 2 seconds (a poor simulation of how long it takes to send a message to a server), and then we schedule a wake-locked action to occur in 1 second. That action is an incoming message of the message we sent, reversed - exactly as we set out to do. The latter is accomplished by the `EventBubble` method call to which we got access to via the aforementioned `eventDrivenBubbles` flag.

After that, the `SendBubble` method ends - no exception has been encountered. That means that the Framework will mark the message successfully as sent. Fantastic! If any any exception is thrown up the stack in `SendBubble` (that is, to the Framework) then the Framework will mark the bubble as failed and alert the user via a notification if necessary. Moreover, if you catch that exception in the `SendBubble` method, and consequently throw a `ServiceQueuedBubbleException` back up the stack to the Framework, then you'll ask the Framework to resend to bubble. The Framework will then deal with sending the bubble a later time.

In addition to scheduling a once-off action, we can ask the Framework to schedule a reoccurring action (perfect for keep-alive heartbeats).

ProcessBubbles

Even though WackyMessenger does not use the `ProcessBubbles` method, its important to explain a bit about it. Firstly, it's an iterator block. In a trivial messenger setup, you'll call upon your `Socket.Receive` blocking method in `ProcessBubbles`. When data comes in, it'll be processed by your serializer (XML, JSON, custom, etc.), and then objectified into a Disa Framework bubble. A mere `yield return bubble` will then catapult the bubble back to the Framework, in the exact same way that EventBubble behaves.

Interim Summary 2

Alright, now you'll be left with something like this:

```
.
.
.
private static string Reverse( string s )
{
    char[] charArray = s.ToCharArray();
    Array.Reverse( charArray );
    return new string( charArray );
}

public override void SendBubble(Bubble b)
{
    var textBubble = b as TextBubble;
    if (textBubble != null)
    {
        Utils.Delay(2000).Wait();
        Platform.ScheduleAction(1, new WakeLockBalancer.ActionObject(() =>
        {
            EventBubble(new TextBubble(Time.GetNowUnixTimestamp(), Bubble.BubbleDirection.Incoming,
                textBubble.Address, null, false, this, Reverse(textBubble.Message)));
        }, WakeLockBalancer.ActionObject.ExecuteType.TaskWithWakeLock));
    }
}
.
.
.
```

The Final Tid-bits

We'll need to implement the `BubbleGroupComparer` method. This method is basically how Disa deals with grouping Bubbles into `BubbleGroups`. A BubbleGroup is synonymous with a thread or conversation.

In most cases, you'll just do an ordinal string comparison (the '==' in C#). However, in some cases you may need to use an algorithm - such as a `PhoneNumberComparer` that will yield equality to the number tuple +1 604 393 2838 and 604-393-2838. You can find this comparer in PhoneBook.cs.

In WackyMessenger's case, ordinal string comparison will suffice:

```
public override bool BubbleGroupComparer(string first, string second)
{
    return first == second;
}
```

Next off, `GetBubbleGroupLegibleId`. Some conversations need an additional mark on them in the conversation list (in a SMS/MMS plugin, we need to label the conversation in the conversation list with a Mobile, Work, Home, tag).

In WackyMessenger, we don't need such a thing! Leave this method be - we don't need to touch it.

Next up, GetBubbleGroupName. This should be pretty self explanatory. How do I relate the address "604 232 9830" to "Meghan"? This is that method.

In WackyMessenger's case, its so simple, that we don't really have any way to relate an address back to who it is. After all, it is just repeating what we say. Therefore, the name of the BubbleGroup will be the address of it. Simple, right?

```
public override Task GetBubbleGroupName(BubbleGroup group, Action<string> result)
{
    return Task.Factory.StartNew(() =>
    {
        result(group.Address);
    });
}
```

It should be noted that this is a common pattern that you'll be seeing in a lot of Disa's interface code. We try to enforce the plugin developer to use the Task Programming Library as tasks are cheap, and the Framework is incredibly asynchronous. When the result is found, you call the provided callback, result, with the, well, result.

Next, `GetBubbleGroupPhoto`. This gets the photo of the conversation. Once again, WackyMessenger is simple and dumb. Lets just tell the framework to generate the default thumbnail.

```
public override Task GetBubbleGroupPhoto(BubbleGroup group, Action<DisaThumbnail> result)
{
    return Task.Factory.StartNew(() =>
    {
        result(null);
    });
}
```

The next three methods, `GetBubbleGroupPartyParticipants`, `GetBubbleGroupUnknownPartyParticipant`, `GetBubbleGroupPartyParticipantPhoto` are all Party orientated. WackyMessenger doesn't support parties yet. We'll ignore these for now - we'll come back to it in the later tutorials when we actually give WackyMessenger Party support.

And finally, `GetBubbleGroupLastOnline`, fetches the last seen time of the specified conversation. The BubbleGroup passed into here will always be a solo (i.e: not a Party/GroupChat). WackyMessenger doesn't support last seen times. Therefore, we just ignore it.

Interim Summary 3

At this point, you should have the following code:

```
using System;
using System.Threading.Tasks;
using System.Collections.Generic;
using Disa.Framework.Bubbles;

namespace Disa.Framework.WackyMessenger
{
    [ServiceInfo("WackyMessenger", true, false, false, false, false, typeof(WackyMessengerSettings),
        ServiceInfo.ProcedureType.ConnectAuthenticate, typeof(TextBubble))]
    public class WackyMessenger : Service
    {
        public override bool Initialize(DisaSettings settings)
        {
            throw new NotImplementedException();
        }

        public override bool InitializeDefault()
        {
            return true;
        }

        public override bool Authenticate(WakeLock wakeLock)
        {
            return true;
        }

        public override void Deauthenticate()
        {
            // do nothing
        }

        public override void Connect(WakeLock wakeLock)
        {
            // do nothing
        }
```

```
        public override void Disconnect()
        {
            // do nothing
        }

        public override string GetIcon(bool large)
        {
            throw new NotImplementedException();
        }

        public override IEnumerable<Bubble> ProcessBubbles()
        {
            throw new NotImplementedException();
        }

        private static string Reverse( string s )
        {
            char[] charArray = s.ToCharArray();
            Array.Reverse( charArray );
            return new string( charArray );
        }

        public override void SendBubble(Bubble b)
        {
            var textBubble = b as TextBubble;
            if (textBubble != null)
            {
                Utils.Delay(2000).Wait();
                Platform.ScheduleAction(1, new WakeLockBalancer.ActionObject(() =>
                {
                    EventBubble(new TextBubble(Time.GetNowUnixTimestamp(), Bubble.BubbleDirection.Incoming,
                        textBubble.Address, null, false, this, Reverse(textBubble.Message)));
                }, WakeLockBalancer.ActionObject.ExecuteType.TaskWithWakeLock));
            }
        }

        public override bool BubbleGroupComparer(string first, string second)
        {
            return first == second;
        }

        public override Task GetBubbleGroupLegibleId(BubbleGroup group, Action<string> result)
        {
            throw new NotImplementedException();
        }

        public override Task GetBubbleGroupName(BubbleGroup group, Action<string> result)
        {
            return Task.Factory.StartNew(() =>
            {
                result(group.Address);
            });
        }

        public override Task GetBubbleGroupPhoto(BubbleGroup group, Action<DisaThumbnail> result)
        {
            return Task.Factory.StartNew(() =>
            {
                result(null);
            });
        }

        public override Task GetBubbleGroupPartyParticipants(BubbleGroup group, Action<DisaParticipant[]>
result)
        {
            throw new NotImplementedException();
```

```
            throw new NotImplementedException();
        }

        public override Task GetBubbleGroupUnknownPartyParticipant(BubbleGroup group, string
unknownPartyParticipant, Action<DisaParticipant> result)
        {
            throw new NotImplementedException();
        }

        public override Task GetBubbleGroupPartyParticipantPhoto(DisaParticipant participant,
Action<DisaThumbnail> result)
        {
            throw new NotImplementedException();
        }

        public override Task GetBubbleGroupLastOnline(BubbleGroup group, Action<long> result)
        {
            throw new NotImplementedException();
        }
    }

    public class WackyMessengerSettings : DisaSettings
    {
        // store settings in here:
        // e.g: public string Username { get; set; }
    }
}
```

Wonderful! We actually now have enough code to run out first plugin! ^^ exciting ^^

Let's Run It!

Add Disa.Framework.WackyMessenger as a reference in Disa.Terminal.

Go into Program.cs in Disa.Terminal, and change:

```
Initialize(new Service[] { });
```

into

```
Initialize(new [] { new WackyMessenger() });
```

Note: you may have to add a using statement to the top of Program.cs here.

```
using Disa.Framework.WackyMessenger;
```

Now, launch Disa.Terminal.

You have to register a service before you can use it. So, Type:

```
register WackyMessenger
```

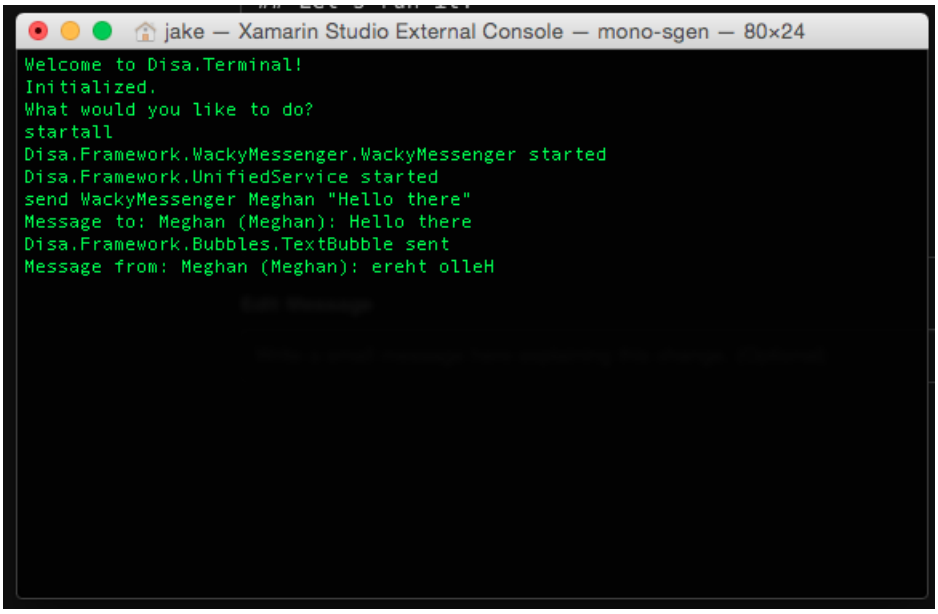
You'll only have to do this once. Disa has now saved this setting onto your disk. Then type,

```
startall
```

All the services will now start. We can now test our plugin:

```
send WackyMessenger Meghan "Hello there"
```

Now, wait a bit.. and then you should get a response!



Wahoo, we did it!

A Big Problem | Duplicate Bubbles

It's possible that Disa may send a bubbles twice (e.g: lost acknowledgment but server actually received the message which results in Disa resending the bubble again) - which is quite common when you have flaky internet connections. This isn't a flaw accustomed to just Disa - many other clients have the exact same issue.

To address this issue, bubbles typically get unique IDs associated with them. The bubble is then sent to the server along with an ID. If the server gets a duplicate ID, then it merely discards of it.

To implement this in Disa, implement the `IVisualBubbleServiceId` interface:

```
public void AddVisualBubbleIdServices(VisualBubble bubble)
{
    throw new NotImplementedException();
}

public bool DisctinctIncomingVisualBubbleIdServices()
{
    throw new NotImplementedException();
}
```

You are free to set the `VisualBubble.IdService` and `VisualBubble.IdService2` properties in `AddVisualBubbleIdServices`. `AddVisualBubbleIdServices` is called just before the bubble reached SendBubble in your service code.

`DisctinctIncomingVisualBubbleIdServices` asks if you want to filter any duplicate incoming bubbles. There's also the possibility that the server may send you multiple messages of the same ID. If this method returns true, it will filter those. If it returns false, it will not filter them.

In WackyMessenger's case, it's not necessary to include this. But for the sake of an example, here's the full-code with it implemented:

```
using System;
using System.Threading.Tasks;
using System.Collections.Generic;
using Disa.Framework.Bubbles;

namespace Disa.Framework.WackyMessenger
{
    [ServiceInfo("WackyMessenger", true, false, false, false, false, typeof(WackyMessengerSettings),
        ServiceInfo.ProcedureType.ConnectAuthenticate, typeof(TextBubble))]
    public class WackyMessenger : Service, IVisualBubbleServiceId
    {
        private string _deviceId;
        private int _bubbleSendCount;
        .
        .
        .
        public void AddVisualBubbleIdServices(VisualBubble bubble)
        {
            bubble.IdService = _deviceId + ++_bubbleSendCount;
        }

        public bool DisctinctIncomingVisualBubbleIdServices()
        {
            return true;
        }
        .
        .
        .
```

Notice that we get the Unix timestamp here, and then append a message counter to it. This ensures that Message Ids are always unique.

In Summary

In summary you have learned how to make your first plugin, and also quite a bit about the Disa Framework.

Disa Bubble Group Manager and Bubble Groups

Disa Bubble Manager and Bubbles

Disa Contacts

`Disa.Framework.Contact` is an abstract class that Plugins will derive from for their own Plugin specific needs (e.g., TelegramContact).

`Contact` has the following fields:

FIELD	DESCRIPTION
FirstName	
LastName	
Status	
Ids	
LastSeen	
Available	
FullName	

And `Contact.ID` has the following fields:

FIELD	DESCRIPTION
Service	
Id	
LegibleId	
Name	
Tag	

Deriving from `Contact` are `PartyContact` and `BotContact`. Plugins will derive from these classes as necessary to indicate that a Contact is participating in a Party or that a Contact is a Bot.

Disa Deploying

2 The Disa Framework

The Disa Framework provides a set of logical building blocks to provide developers everything they need to build plugins for their favorite instant messaging platforms (or anything that they can really *make* out of it). The building blocks cover the following categories:

A set of platform abstractions that allow you to interact with a particual platform (e.g., Android) in a platform agnostic manner.
PlatformManager, PlatformImplementation

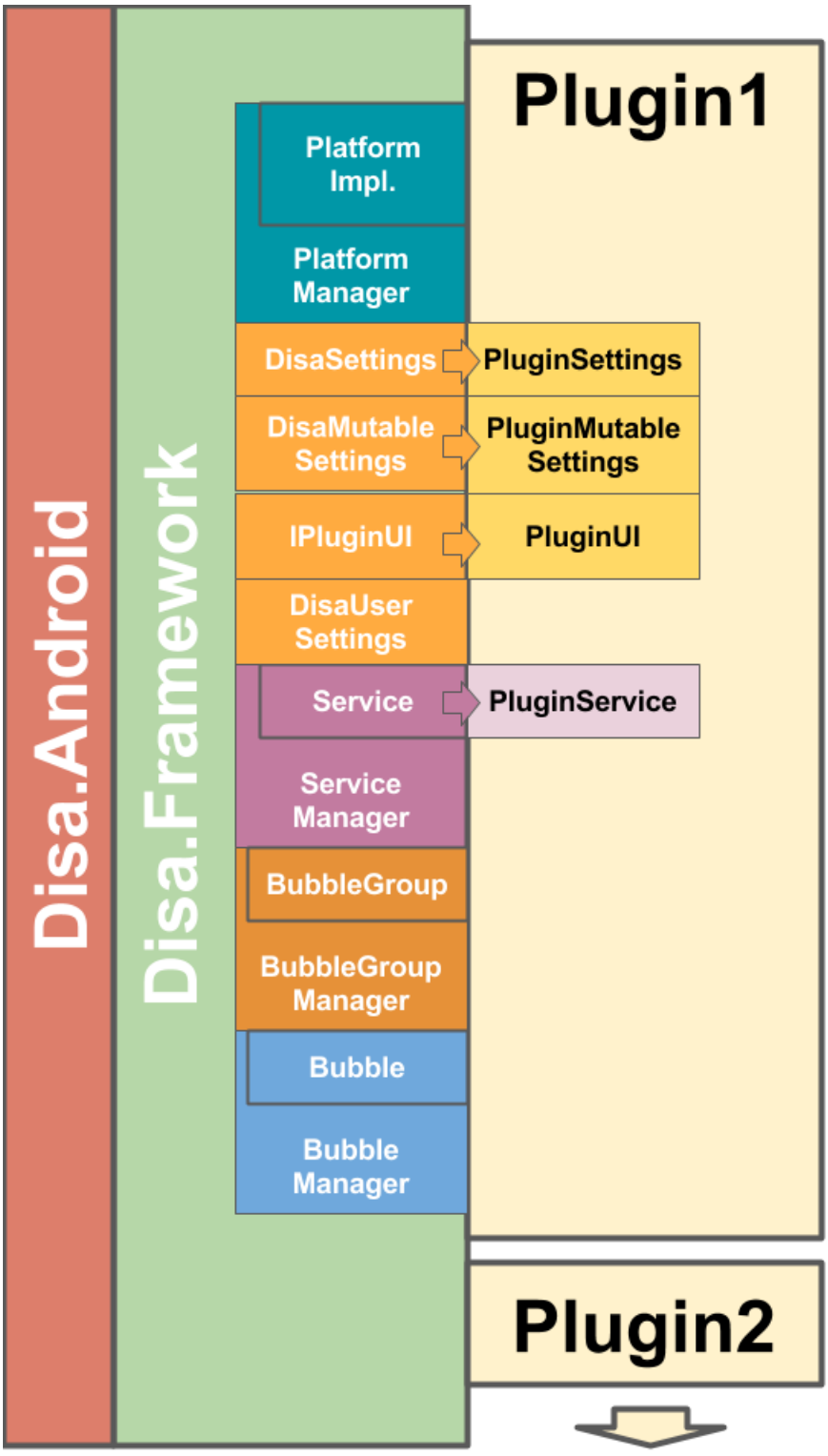
A categorized set of setting implementations for you to derive from to support your plugin's setting needs
DisaSettings, DisaMutableSettings, IPluginUI, DisaUserSettings

A Service Manager and defined Service lifecycle to allow your plugin to register and expose its functionality in a Disa front-end (e.g., Disa.Android).
Service Manager, Service

A conversation thread manager and a representation for a conversation thread
BubbleGroupManager, BubbleGroup

A conversation element manager and representation for various conversation elements
BubbleManager, Bubble

The following diagram gives a high-level overview of the Disa.Framework components.


Disa.Android
Disa.Framework
Platform Impl.
Platform Manager
DisaSettings
PluginSettings
DisaMutable Settings
PluginMutable Settings
IPluginUI
PluginUI
DisaUser Settings
Service
PluginService
Service Manager
BubbleGroup
BubbleGroup Manager
Bubble
Bubble Manager
Plugin1
Plugin2

New Message and New Message Extended

A plugin will will implement the `Disa.Framework.INewMessage` interface to participate in new message creation. `INewMessage` defines the following API:

API	DESCRIPTION
GetContacts	
GetContactsFavorites	
GetContactPhoto	
FetchBubbleGroup	
FetchBubbleGroupAddress	
GetContactFromAddress	
MaximumParticipants	
FastSearch	
CanAddContact	

In addition, a Plugin can implement the `Disa.Framework.INewMessageExtended` interface to provide additional capabilities in new message creation. `INewMessageExtended` defines the following API:

API	DESCRIPTION
FetchBubbleGroupAddressFromLink	
SupportsShareLinks	

Disa Platform Manager

The `PlatformManager` is responsible for exposing appropriate platform abstractions that a plugin can take advantage of. Take, for example, the common requirement for mobile developers to abstract the location for database files. Most mobile developers are familiar with writing the following platform specific logic for Android and iOS platforms:

Android

```
string databasePath = System.Environment.GetFolderPath(
    System.Environment.SpecialFolder.Personal);
```

iOS

```
string documentsPath = Environment.GetFolderPath (
    Environment.SpecialFolder.Personal);
string databasePath = Path.Combine (
    documentsPath, "..", "Library");
```

While this is certainly good knowledge to have, a platform can have many more platform specific scenarios such as this one that would benefit from a common abstraction - allowing you to focus on your plugin logic instead. The `PlatformManager` and its supporting classes provide this benefit and in some cases enforce a certain behavior for a plugin to follow.

PlatformImplementation

The `PlatformManager` manages classes derived from `PlatformImplementation`. It is in the derived class implementation that you'll find support for various platform specific scenarios. `PlatformImplementation` is an abstract class with numerous abstract methods to build out a rich support for particular platform. Currently, Disa provides an Android and Desktop implementations. By familiarizing yourselft with `PlatformImplementation`'s API surface, you can take advantage of pre-built support for platform specific scenarios. Also, several of the APIs provide messaging and other plugin specific functionality that you will need to know to implement various plugin features. Here is a summary of the current API surface for `PlatformImplementation`:

API	DESCRIPTION
MarkTemporaryFileForDeletion	
UnmarkTemporaryFileForDeletion	
GetIcon	
GetCurrentLocale	
GetFilesPath	

API	DESCRIPTION
GetPicturesPath	
GetVideosPath	
GetAudioPath	
GetLogsPath	
GetSettingsPath	
GetDatabasePath	
GetDeviceId	
GetPhoneBookContacts	
ScheduleAction	
RemoveAction	
ScheduleAction	
WakeLock	
AquireWakeLock	
OpenContact	
DialContact	
LaunchViewIntent	
DeviceHasApp	
HasInternetConnection	
ShouldAttemptInternetConnection	
GetMimeTypeFromPath	
GetExtensionFromMimeType	
GenerateJpegBytes	
GenerateVideoThumbnail	
GenerateBytesFromContactCard	
GenerateContactCardFromBytes	

API	DESCRIPTION
GenerateLocationThumbnail	
CreatePartyBitmap	
GetCurrentBubbleGroupOnUI	
SwitchCurrentBubbleGroupOnUI	
DeleteBubbleGroup	
ExecuteAllOldWakeLocksAndAllGracefulWakeLocksImmediately	

Platform Initialization

Platform initialization is handled by the Disa client. `PlatformManager` exposes the following API to inject a particular `PlatformImplementation`:

```
public static void PreInitialize(PlatformImplementation platform,
                                  AxolotlImplementation axolotl)
```

`AxolotlImplementation` provides support for the Axolotl messaging protocol which provides perfect forward secrecy. We will discuss this in another section of the Guide. `PreInitialize` simply assigns the injected `PlatformImplementation` to:

```
PlatformManager.PlatformImplementation
```

and

```
Platform.PlatformImplementation
```

TODO: Should we DRY this up?

The assignment to `Platform.PlatformImplementation` is for convenience as `Platform` exposes a simpler API to call such as:

```
Platform.GetDatabasePath();
```

With `PlatformManager.PreInitialize` completed, the Disa client will then call `PlatformManager.InitializeMain`.

```
public static void InitializeMain(Service[] allServices)
```

An array of `Service` instances is passed into `InitializeMain`. How this list is built up by the Disa client will be discussed in the Deploying section of this Guide. In `InitializeMain`, the `Service` instances are passed in to `ServiceManager.Initialize`.

```
ServiceManager.Initialize(allServices.ToList());
```

See the section on `ServiceManager` in this Guide for further details here. Then, a call is made to `ServiceUserSettingsManager.LoadAll`.

```
ServiceUserSettingsManager.LoadAll();
```

See the section Settings in the Guide for further details. Finally, a call is made to `BubbleGroupFactory.LoadAllPartiallyIfPossible`.

```
BubbleGroupFactory.LoadAllPartiallyIfPossible();
```

See the section on Bubbles and BubbleGroups for further details. OK, we punted on most of the description that is going on here, but it will

make more sense when we pick up the thread in each of the sections devoted to each particular piece of functionality.

Versioning

```
PlatformManager' is the offical location to determine the version of the Disa.Framework you are coding against.
The APIs
```

FrameworkVersion `and 'FrameworkVersionInt` are used for this.

Assemblies

`PlatformManager` maintains an official list of core assemblies needed when deploying your Plugin. We will see how this list is used in the Deploying section.

Disa Service

Disa Service Manager

`Disa.Framework.ServiceManager` controls the registration and lifecycle of your Plugin.

Supporting Classes

`ServiceManager` depends on several supporting classes that you'll want to have a good understanding of before we dig deeper into the core of `ServiceManager`.

ServiceBinding

`ServiceManager` maintains a static `List` of `ServiceBinding` instances. This allows ServiceManager to track lifecycle state for a particular `Service`.

```
{
    public Service Service { get; private set; }
    public ServiceFlags Flags { get; private set; }

    public ServiceBinding(Service service, ServiceFlags flags)
    {
        Service = service;
        Flags = flags;
    }
}
```

ServiceFlags

The lifecycle state of a particular `Service` is represented by an instance of `ServiceFlags`.

```
private class ServiceFlags
{
    public bool Running { get; set; }
    public bool Starting { get; set; }
    public bool ManualSettingsNeeded { get; set; }
    public bool ConnectionFailed { get; set; }
    public bool AuthenticationFailed { get; set; }
    public bool DisconnectionFailed { get; set; }
    public bool DeauthenticationFailed { get; set; }
    public bool Aborted { get; set; }
    public bool AbortedSpecial { get; set; }
    public bool Expired { get; set; }
}
```

This allows `ServiceManager` to expose methods for querying the state of a particular `Service` such as:

```
private static IEnumerable<Service> BindingQuery(Func<ServiceBinding, bool> predicate)
{
    return from serviceBinding in ServicesBindings
            where predicate(serviceBinding)
            select serviceBinding.Service;
}

public static IEnumerable<Service> Running
{
    get { return BindingQuery(binding => binding.Flags.Running); }
}

public static bool IsRunning(Service service)
{
    return Running.FirstOrDefault(s => s == service) != null;
}
```

Querying for Services

Here is a summary of the `ServiceManager` APIs you can use to query the services based on their lifecycle state.

API	DESCRIPTION
Registered	
RegisteredNoUnified	
Starting	
Running	
RunningNoUnified	
ManualSettingsNeeded	
ConnectionFailed	
AuthenticationFailed	
DisconnectionFailed	
DeauthenticationFailed	
Expired	
Aborted	
AbortedSpecial	
GetNonRegistered	
GetRegistered	
IsRegistered	

API	DESCRIPTION
IsStarting	
IsRunning	
IsExpired	
IsAborted	
IsAbortedSpecial	
IsManualSettingsNeeded	
IsConnectionFailed	
IsAuthenticationFailed	
IsDisconnectionFailed	
IsDeauthenticationFailed	
GetFlags	

And here are some additional `ServiceManager` APIs for querying for `Service`s and `Service` state.

API	DESCRIPTION
Get(BubbleGroup group)	
Has(BubbleGroup group, Service service)	
Get(string guid)	

Registering and Unregistering Services

Now we can return to `PlatformManager.InitializeMain` and the call to:

```
ServiceManager.Initialize(allServices.ToList());
```

Recall that the collection of `Service` instance we have here was passed in to `InitializeMain`. Typically, this will be called by the Disa front-end. We will see how this list of `Service` instances is created in the section on Deploying. For now, let's pickup in `ServiceManager.Initialize` where see that we assign the passed in `Service` instances to the collection `AllInternal`.

```
AllInternal = allServices;
```

While `AllInternal` is private, it is exposed via the following APIs:

API	DESCRIPTION
All	

API	DESCRIPTION
AllNoUnified	
Get(Type serviceType)	
GetByName(string serviceName)	
GetUnified()	

We then get our first exposure to `ServiceManager.RegisteredServicesDatabase` via this call:

```
RegisteredServicesDatabase.RegisterAllRegistered();
```

ServiceManager.RegisteredServicesDatabase

`RegisteredServicesDatabase` maintains an XML listing of all registered services. The XML backing file is named `RegisteredServicesList.xml` and will be located at the location pointed to by `Platform.GetSettingsPath()`. Besides `RegisterAllRegistered`, which we will explore in a sec, `RegisteredServicesDatabase also exposes:

API	DESCRIPTION
SaveAllRegistered	Saves the name of each `Service` in `RegisterNoUnified` into `RegisteredServicesList.xml`
FetchAllRegistered(string settingsPath)	Given a settings path, will return a `List<string>` of all registered service names.
AddToRegisteredAndSaveAllForImminentRestart(string settingsPath, string additionalService)	Given a settings path and the name of a new `Service`, will write out a new `RegisteredServicesList.xml` with new `Service` name included.

OK, let's get back to our `RegisteredServicesDatabase.RegisterAllRegistered` call. In this method, we read in all the `Service` names in `RegisteredServicesList.xml`. We then loop over all the names and verify that it is contained in `AllInternal`. With this check in place we proceed to call:

```
Register(service);
```

Following into this function, we see that a `Service` is considered registered once a `ServiceBinding` instance has been added into the `ServiceManager.ServiceBinding` collection for it. Also, the `ServiceBinding` will have freshly initialized `ServiceFlags` instance at this point in time.

```
lock (ServicesBindings) ServicesBindings.Add(
    new ServiceBinding(service, new ServiceFlags()));
```

Unregistering

A `Service` can be unregistered by calling `Unregister`. This will have the effect of removing the `ServiceBinding` instance for the `Service`. This will also cause an event to be raised that you can listen for:

```
ServiceEvents.RaiseServiceUnRegistered(service);
```

TODO SettingsChangedManager.SetNeedsContactSync(service, true);

Managing Service Lifecycle

Starting

`ServiceManager.Start` is used to start a `Service`. We start by wrapping the entire start of the `Service` in a background thread:

```
return Task.Factory.StartNew(() =>
{
```

We then further wrap the entire start of the `Service` in the DisaStart `WakeLock`.

```
using (var wakeLock = Platform.AquireWakeLock("DisaStart"))
{
```

Simplistic checks are then performed to make sure the `Service` is not already running or starting. We simply return if so. We then add one more additional wrapping of the start of the `Service` by locking on the service instance passed in:

```
lock (service)
{
```

With this setup in place, we now set our current state for this `Service`:

```
GetFlags(service).Aborted = false;
GetFlags(service).AbortedSpecial = false;
GetFlags(service).Starting = true;
GetFlags(service).ManualSettingsNeeded = false;
```

We then attempt to load our `DisaSettings` derived settings class for this `Service`.

```
var settings = SettingsManager.Load(service);
```

If this returns null, it means we have not specified a `DisaSettings` derived class for this `Service` - see the section on Disa Settings in this Guide for further details on why this could occur. If the settings are null, then we call our `Service` lifecycle method `InitializeDefault`.

```
if (!service.InitializeDefault())
{
    GetFlags(service).ManualSettingsNeeded = true;
    ServiceEvents.RaiseServiceManualSettingsNeeded(service);
}
else
{
    Utils.DebugPrint("Service initialized under no settings.");
}
```

Note here how if our implementation for `Service.InitializeDefault` returns false, then we set our `ManaulSettingsNeeded` flag and raise the event `ManualSettingsNeeded`.

Now let's take the opposite path where are our call to SettingsManager.Load returns our DisaSettings derived settings class instance. In this case, we call the `Service` lifecycle method `Initialize` passing in our settings instance.

```
if (service.Initialize(settings))
{
    Utils.DebugPrint("Successfully initialized service!");
}
else
{
    GetFlags(service).ManualSettingsNeeded = true;
    ServiceEvents.RaiseServiceManualSettingsNeeded(service);
}
```

Similar to our implementation for `InitializeDefault`, if our implementation for `Initialize` returns false, then we set our `ManaulSettingsNeeded` flag and raise the event `ManualSettingsNeeded`.

Next, if our `Service` has specified that it `UsesInternet` we perform our platform specific checks to see if we have an Internet connection and if we should attempt an Internet connection. The `ServiceInfo` attribute on your Plugin's `Service` derived class will be where you specify your `Service` uses Internet.

With these initial checks and initialization lifecycle calls out of the way we now call `StartInternal` which will handle our connect and authenticate lifecycle method calls.

```
StartInternal(service, wakeLock);
```

Picking up in `StartInternal`, we make our checks to see if manual settings are needed, the service is registered and the service is not already running. An appropriate `ServiceSchedulerException` is thrown if necessary. We then update our state for the `Service` with a call to `ClearFailures` which will update the state as follows:

```
flags.ConnectionFailed = false;
flags.AuthenticationFailed = false;
flags.DeauthenticationFailed = false;
flags.DisconnectionFailed = false;
```

Following this, we raise the `SettingsLoaded` event for any interested listeners.

With all of this behind us, we are now ready to call our lifecycle methods for connect and authenticate. A `Service` can specify via the `ServiceInfo` attribute the order in which these lifecycle methods are called as we can see here:

```
if (registeredService.Information.Procedure
    == ServiceInfo.ProcedureType.AuthenticateConnect)
{
    authenticate();
    connect();
}
else if (registeredService.Information.Procedure
            == ServiceInfo.ProcedureType.ConnectAuthenticate)
{
    connect();
    authenticate();
}
```

Note that the `connect` and `authenticate` calls here are actually `Action` delegates defined above this code snippet. The delegates handle catching and rethrowing exceptions as appropriate. Also, the delegates will handle setting the `ConnectionFailed` and `AuthenticationFailed` states for the `Service` if necessary.

If the calls to `connect` and `authenticate` succeed, then we set the state of the `Service` to `Running` and raise the `Started` event.

OK, back in our `ServiceManager.Start` method, we pickup by handling any exceptions coming out of our most recent steps. Exception handlers here will set the `Starting` state of the `Service` to false. In addition the `ServiceSpecialRestartException` handler will attempt another start of the `Service` with an appropriate delay:

```
catch (ServiceSpecialRestartException ex)
{
    Utils.DebugPrint("Service " + service.Information.ServiceName +
                        " is asking to be restarted on connect/authenticate. This should be called sparingly,
Disa can easily " +
        "break under these circumstances. Reason: " + ex + ". Restarting...");
    StopInternal(service);
    epilogue();
    Start(service, smartStart, smartStartSeconds);
    return;
}
```

Also, the `ServiceExpiredException` handler will set the `Aborted` and `Expired` state for the `Service` as well as raise the `ServiceExpired` event.

```
catch (ServiceExpiredException ex)
{
    Utils.DebugPrint("The service " + service.Information.ServiceName +
                        " has expired: " + ex);
    GetFlags(service).Aborted = true;
    GetFlags(service).Expired = true;
    ServiceEvents.RaiseServiceExpired(service);
    StopInternal(service);
    epilogue();
    return;
}
```

A this point we hook into the `BubbleManager` and start our Bubble receiving thread.

```
BubbleManager.SendSubscribe(service, true);
BubbleManager.SendLastPresence(service);

service.ReceivingBubblesThread = new Thread(() =>
{
    StartReceiveBubbles(service);
});
service.ReceivingBubblesThread.Start();
```

We'll hold off on describing these for the section on the Bubble Manager and Bubbles later in this Guide.

Finally, we set the `Starting` state for our `Service` to false.

```
GetFlags(service).Starting = false;
```

TODO

```
BubbleQueueManager.SetNotQueuedToFailures(service);

Utils.Delay(1000).ContinueWith(x =>
{
    BubbleGroupSync.ResetSyncsIfHasAgent(service);
    BubbleGroupUpdater.Update(service);
    BubbleQueueManager.Send(new[] {service.Information.ServiceName});
    BubbleGroupManager.ProcessUpdateLastOnlineQueue(service);
    SettingsChangedManager.SyncContactsIfNeeded(service);
});
```

Stopping

Restarting

Aborting

Disa Settings

The Disa Framework provides four categories of settings that you should be familiar with to support your Plugin's needs.

CATEGORY	DESCRIPTION
`DisaSettings`	A settings store of your service. You can use this to store any information.
`DisaMutableSettings`	Use `DisaMutableSettings` and `MutableSettingsManager` to save information you find yourself frequently saving (such as a timestamp you need to keep updated everytime the service is started).
PluginSettingsUI/IPluginPage	Plugin defined Xamarin.Forms page or pages to present a UI to capture settings from the user
`DisaServiceUserSettings`	User settings such as `Ringtone` and `VibrateOption`

DisaSettings

Setup and Initialization

Typically you will derive a class from `DisaSettings` to hold your Plugin specific settings. While we haven't discussed implementing your Plugin `Service` class yet, we can briefly mention the touch-points for your `DisaSettings`. The `ServiceInfo` attribute on your `Service` class takes a `Type` parameter of your `DisaSettings` derived class as can be seen here:

```
[ServiceInfo("WackyMessenger", true, false, false, false, false, typeof(WackyMessengerSettings),
        ServiceInfo.ProcedureType.ConnectAuthenticate, typeof(TextBubble))]
```

This will have the effect of storing away a `ServiceName` ("WackyMessenger") and a `Type` (typeof(WackyMessengerSettings)) for your `DisaSettings` derived class at:

```
Service.Information.ServiceName
```

and

```
Service.Information.Settings
```

We'll see how this is used shortly. Now when your Plugin's `Service` derived class instance is started, the first thing that happens is that your override of `InitializeDefault()` is called. It attempts to try and start the service without any settings. If this method returns true, it is assumed your service doesn't need any settings. If this method returns false, then `InitializeDefault(DisaSettings)` is called - the framework will provide you with your stored settings at this point.

Managing

`Disa.Framework.SettingsManager` manages saving, loading and deleting your `DisaSettings` derived class instance. Whenever you want to save your settings, you can call `SettingsManager.Save`.

```
public static void Save(Service service, DisaSettings settings)
```

This method will first determine a path for your settings based on the `Service.Information.ServiceName`:

```
Path.Combine(Platform.GetSettingsPath(), service.Information.ServiceName + ".xml");
```

It will then hand-off completion of saving to:

```
public static void Save(Stream fs, Type settingsType, DisaSettings settings)
```

And here we see how the `Type` you originally specified for your settings is used to serialize an XML representation of your settings:

```
var serializerObj = new XmlSerializer(settingsType);
serializerObj.Serialize(fs, settings);
```

Since your settings are based off of your `ServiceName`, loading and deleting your settings have simple APIs:

```
public static DisaSettings Load(Service ds)
public static void Delete(Service service)
```

Note that if your settings have not been saved out yet, `Load` will return `null`.

Additional public API's are availalbe in SettingsManager for loading and saving that allow you to specify stream, settings path, etc.

DisaMutableSettings

Setup

Typically you will derive a class from `DisaMutableSettings` to hold frequently changing settings.

Managing

`Disa.Framework.MutableSettingsManager` manages saving, loading and deleting your `DisaMutableSettings` derived class instance. Whenever you want to save your settings, you can call `MutableSettingsManager.Save`.

```
public static void Save<T>(T settings) where T : DisaMutableSettings
```

This method will call out to

```
Save(typeof(T).Name, settings);
```

Following this through we see that your DisaMutableSettings derived class will be serialized out to XML at:

```
Path.Combine(Platform.GetSettingsPath(), name + ".xml");
```

Where `name` here is the class name of your `DisaMutableSettings` derived class.

Since your settings are based off of your class name, loading and deleting your settings have simple APIs:

```
public static T Load<T>() where T : DisaMutableSettings
public static void Delete<T>() where T : DisaMutableSettings
```

Note, that in this case, `Load` will create an appropriate instance of your settings if a file backing has not been saved out yet:

```
private static DisaMutableSettings Load(string name,
                                        Type settings)
{
    .
    .
    .
    return Activator.CreateInstance(settings)
        as DisaMutableSettings;
}
```

PluginSettingsUI/IPluginPage

Disa Plugins can present a Xamarin.Forms based UI to expose settings necessary for initializing and modifying attributes for their needs. An example would be to allow a user to input credentials to identify the user to a particular messaging back-end.

To indicate to the Disa Framework a designated class for this, you annotate a class with the `Disa.Framework.PluginSettingsUI` attribute. You pass in the `Type` of your Plugin's `Service` derived class. Once you have identified a class with the `PluginSettingsUI` attribute, you can implement the `Disa.Framework.Mobile.IPluginPage` interface's `Fetch` method to return a Xamarin.Forms `Page` to present a UI to capture/modify settings particular to your Plugin.

DisaServiceUserSettings

Setup

Recall in `PlatformManager.InitializeMain` the call to `ServiceUserSettingsManager.LoadAll`. If we dig into this we see that we loop over `ServiceManager`'s `AllNoUnified` collection of `Service`s. For each Plugin `Service` we call `ServiceUserSettingsManager.Load`. While this is a similar XML serialization/deserialization setup that we have seen above for the other settings, the important detail to note here is the determination of the settings path. The critical function to examine is listed here:

```
private static string GetBaseLocation()
{
    var databasePath = Platform.GetDatabasePath();
    if (!Directory.Exists(databasePath))
    {
        Utils.DebugPrint("Creating database directory.");
        Directory.CreateDirectory(databasePath);
    }

    var bubbleGroupsSettingsBasePath = Path.Combine(databasePath,
        "serviceusersettings");
    if (!Directory.Exists(bubbleGroupsSettingsBasePath))
    {
        Utils.DebugPrint(
            "Creating bubble service user settings base directory.");
        Directory.CreateDirectory(bubbleGroupsSettingsBasePath);
    }

    return bubbleGroupsSettingsBasePath;
}
```

First, we determine, and create if necessary, the platform specific directory for our database location. Then, we append "serviceusersettings" and create, if necessary, this subdirectory. OK, with this location in hand, let's see how it is used. The critical function here is listed below:

```
private static string GetPath(Service service)
{
    return Path.Combine(GetBaseLocation(),
        service.Information.ServiceName + ".xml");
}
```

OK, we are familiar with this now. The `DisaUserSettings` for a particular Plugin is stored using the Plugin's `ServiceName`. For example, for the Telegram plugin, the path would be something like:

```
<database path>\serviceusersettings\Telegram.xml
```

Managing

Currently, `DisaUserSettings` are typically managed by the Disa front-end. Here is a listing of the current settings maintained:

SETTING	DESCRIPTION
NotificationLed	
Ringtone	
BlockNotifications	
ServiceColor	
VibrateOption	
VibrateOptionCustomPattern	

Disa Unified Service

Disa User Information

Represents the info that is presented in a Contact card.

Telegram Contacts

TelegramBotContact in Telegram contacts.

Telegram.FetchContacts has call

```
using (var client = new FullClientDisposable(this))
{
    var response = (ContactsContacts)await client.Client.Methods.ContactsGetContactsAsync(
        new ContactsGetContactsArgs
        {
            Hash = string.Empty
        });
    contactsCache.AddRange(response.Users.OfType<User>().ToList());
    _dialogs.AddUsers(response.Users);
    return contactsCache;
}
```

Telegram.GetContacts (INewMessage)

```
using (var client = new FullClientDisposable(this))
{
    var searchResult =
        TelegramUtils.RunSynchronously(
            client.Client.Methods.ContactsSearchAsync(new ContactsSearchArgs
            {
                Q = query,
                Limit = 50 //like the official client
            }));
    var contactsFound = searchResult as ContactsFound;
    var globalContacts = GetGlobalContacts(contactsFound);
    localContacts.AddRange(globalContacts);
}
```

NewMessage.GetGlobalContacts had the filter for Bot removal

Telegram Messaging

4 The Telegram Plugin

Telegram Service

Telegram Settings

Recall that the Disa Framework provides three categories of settings:

CATEGORY	DESCRIPTION
DisaSettings	Settings tied to a particular Plugin
DisaMutableSettings	Miscellaneous settings based on your Plugin's needs
PluginSettingsUI/IPluginPage	Plugin defined Xamarin.Forms page or pages to present a UI to capture settings from the user

`Disa.Framework.Telegram.Mobile.Settings` class handles settings for Telegram. To indicate to the Disa Framework we would like this class to present a Xamarin.Forms UI for settings, we annotate the class with the `PluginSettingsUI` attribute. We also implement the `Disa.Framework.Mobile.IPluginPage` interface's `Fetch` method to return a Xamarin.Forms `Page` to capture/modify settings for the Telegram plugin. In the `Fetch` implementation we see that we use the `ServiceManager.IsManualSettingsNeeded` to determine if we should display the UI for an initial setup or the UI for modifying already existing settings.

```
if (ServiceManager.IsManualSettingsNeeded(service))
{
    navigationPage = new NavigationPage(Setup.Fetch(service));
}
else
{
    navigationPage = new NavigationPage(new Main(service));
}
return navigationPage;
```

`ManualSettingsNeeded` is a boolean flag on the `ServiceBindings` class which we will explore in more depth later.

Let's take the path for an initial setup first. Picking up in `Setup.Fetch` we see a setup for a Xamarin.Forms `TabbedPage` which will act as our setup wizard. At it's most simplest, the wizard will collect a phone number and then send an SMS code to the device to be sent back to Telegram. But let's get a more detailed overview in place just to provide proper context.

PAGE	DESCRIPTION
Info	Collects phone number from user and displays a switch to allow user to load conversations. When Next is tapped, calls `Telegram.GenerateAuthentication` to authenticate the phone number and if the authentication is successful, moves to the next wizard page - `Code`.
Code	Coming from the `Info` page, we have a valid phone number to use. The Verify button will request a code be sent via SMS via a call to `Telegram.RequestCode`. The user will input the SMS code and then tap Submit. If the user has not input user info yet, the wizard will go to `UserInformation`, otherwise the wizard will call `Telegram.RegisterCode` and then either navigate to `Password` or call `Setup.Save` to save the settings and end the wizard.

PAGE	DESCRIPTION
Password	If the wizard has determined we need to enter a password then we are directed here to allow the user to enter a password which is verified by `Telegram.VerifyPassword`. This is followed by a call to `Setup.Save` and the wizard finishes. The "forgot password" button will take the user to the `PasswordCode` page.
PasswordCode	We get here if the user has specified they forgot their password. A call to `Telegram.RequestAccountReset` is made from the constructor to setup the page. This will send an SMS code which can be entered on the page and a subsequent `Telegram.VerifyCode` will do the verification for us. If everything is ok then a call to `Setup.Save` is made and the wizard finishes.
UserInformation	For a first time registration we need to also collect first and last name. This page will collect that info and then call `Telegram.RegisterCode` followed by a call to `Setup.Save` and then the wizard will finish.

Notice how there was a call to `Setup.Save` at all exit points of the wizard. Let's take a closer look at what is going on here. The signature for this function is as follows:

```
private static void Save(Service service, uint accountId, TelegramSettings settings)
```

Since the Telegram plugin can represent different Telegram accounts keyed off of an appropriate Telegram phone number, the `TelegramSetupSettings` class that is derived from `DisaMutableSettings` records a collection of `TelegramSettings` keyed off of the Telegram phone number. The `TelegramSettings` class which is derived off of `DisaSettings` records the details for an individual Telegram phone number.

Now back to `Setup.Save`. We see that we first save the current `TelegramSettings`:

```
SettingsManager.Save(service, settings);
```

Then we start the actual plugin service:

```
ServiceManager.Start(service, true);
```