# Introduction

The *ConsoleClient* sample demonstrates a way to use the IRC.NET library.

It is composite by two projects:

* **ConsoleClient**

This project is a simple console UI that allow you to connect to one server, and then to one channel. Once it is connected to a channel, you can listen and speak to it.

You can also list the users present on the server or the channel, and send a private message to them.

* **IrcDotNet.Service**

This project is wrapper to the IRC.NET API, with the objective of making it easier to use in a UI.

It provides synchronous methods and exposes only what is needed for this sample (for example, a server cannot have multiple channels using this wrapper).

# Code Walkthrough

We will start our explanation of the code by the Console project, and then go deeper in the API by reviewing the wrapper.

1. Open the IrcDotNet solution.
2. Expand the ConsoleClient project in the Sample folder.
3. Open the Program.cs file.
4. The **Main** method is the entry point of our application, so the first thing we do is to initialize a Server object.

static IrcDotNet.Service.Server myServer { get; set; }

static void Main(string[] args)

{

myServer = new IrcDotNet.Service.Server();

These objects come from the Service wrapper and will be examined later. The rest of the **Main** method is classical Console UI logic and will not be explain in this documentation.

1. The first command available is **Connect**.

It takes the address (IP or hostname) and sends it to the Server.Connected method, with some predefined test credentials.

if (myServer.Connect(serverAddress, "TestUser", "TestUser", "User for test purpose"))

If this method return true, then we are connected, if not, then it is because the connection have timeout.

This method can throw exceptions from the underling API, like NullArgumentException if it misses a parameter. That’s why we have a Try-Catch block for this method.

1. **Disconnect** is even simpler to use, it calls the Server.Disconnect with the comment to use.

if (myServer.Disconnect(comment))

If it times out you are still disconnected, it is just so that you can be sure that your comment has been received by the server.

1. **Join** involves two tasks in code.
   1. Call the **Join** method of the channel of the server and pass it the channel name.

if (myServer.Channel.Join(channelName))

* 1. If the joining to the channel is successful, we subscribe to the CollectionChanged event of the **Messages** object of the channel.

myServer.Channel.Messages.CollectionChanged += messageReceived;

This event is raise each time the channel receives one or more messages and adds it to the message list. That’s because this list is an ObservableCollection, so it tells when it is changed, for binding purpose.

* 1. As a Console UI doesn't have a textbox to be bound with this list, we use a method that will display any new messages received by our Channel object.

foreach (Tuple<string, string> newItem in e.NewItems)

{

Console.Write("{ [{0:t}] <{1}> {2} }", DateTime.Now, newItem.Item1, newItem.Item2);

}

In this method we display any new items added to the list, each item composed by 2 strings, the source and the message. For storing it inside the list (which can handle only one object at a time) we use a Tuple, new class introduced with .NET 4.0, which can encapsulate as many other objects we want.

1. **Leave**, **Say** and **Whisp** have nothing fundamentally new so we'll skip them.
2. For the **ListUser** method we can use the Server.ListUser as an IEnumerable for displaying any users in the server or the channel with a foreach statement.

Now we will explore the Server class of the wrapper.

1. Expand the IrcDotNet.Service project.
2. Open the Server.cs file.
3. The Server class is mainly a wrapper for IrcClient so we have a private member **client** that contains an IrcClient instance.
4. This class have three public properties:
   1. **Name** will simply get the server name of the client.

public string Name { get { return client.ServerName; } }

* 1. **IsConnected** will get the client.IsRegistered and not the client.isConnected because the IsConnected of the client only say that a server have been contacted not that the server have acknowledge the client, so we will consider that the client is connected to a server only when the server have registered the client and allow it to send command.

public bool IsConnected { get { return client.IsRegistered; } }

* 1. A **Channel** property will expose the channel connected to this server.

public Channel Channel { get; private set; }

1. The **Constructor** will initialize the client and its **FloodPreventer** (which is a mechanism of auto-moderation) and the Channel we will use too.

public Server()

{

this.client = new IrcClient();

this.client.FloodPreventer = new IrcStandardFloodPreventer(4, 2000);

Channel = new Channel(client);

}

In this example we will be able to send only 4 messages for every 2 seconds.

1. The first method of the Server will be **Connect.**
   1. In order to make it synchronous we will use a lock from System.Threading: ManualResetEvent.

using (var connexionEvent = new ManualResetEvent(false))

This lock will be opened when the connection with the IRC server is established.

EventHandler<EventArgs> connected = (s, e) =>

{ connectionEvent.Set(); };

Here we use an EventHandler as we want to react to an event of our **client**.

In this EventHandler we put a delegate that will open the lock, with a lambda expression.

We call this EventHandler when the **client** raises the **Registered** event (and not the **Connected** event, as this event is raise when the client have contacted the server, but before the server accept the client).

client.Registered += connected;

* 1. Now we can call the **Connect** method for start the connection.

client.Connect(serverAddress, new IrcUserRegistrationInfo()

{

NickName = nickName,

UserName = userName,

RealName = realName

});

Note that we use an IrcRegistrationInfo child object to store our credentials, in this case for a User.

* 1. Now we will wait for the connection to be accepted by the IRC server (or for the timeout).

if (!connectionEvent.WaitOne(10000))

**{**

client.Registered -= connected;

return false;

}

else

{

client.Registered -= connected;

return true;

}

If the lock **connectionEvent** isn’t raise before 10000ms we return *false* (timeout), otherwise we return *true* (connection successful).

Note that in each case we unsubscribe the EventHandler from the **Registered** event, because if we don’t do that, the lock can’t be recycled in the end of the using, and the system will continue to keep it alive (this lock isn’t a full managed object, it comes from the OS). Keeping that lock alive will result in a DisposedObjectException the next time we try to use it.

1. **Disconnect** method is the same but the **Quit** method of the API take a timeout for forcefully disconnect the client if it not receive an acknowledgment of the disconnection.

client.Quit(10000, comment);

1. The **Whisp** method will use the IrcLocalClient instance of the client to send a message to a specified user.

public void Whisp(string user, string message)  
{  
 client.LocalUser.SendMessage(user, message);  
}

1. **ListUser** use yield return to return any user found in the Users list of the client.

public IEnumerable ListUser()  
{  
 foreach (IrcUser user in client.Users)  
           {  
            yield return user.NickName;  
           }  
}

Let’s take a look at the Channel class now.

1. Open the Channel.cs file.
2. As see with the **Messages** collection in the Program file, the Channel class is design to be bindable, especially with a WPF UI, so it implements the INotifyPropertyChanged interface.

public class Channel : INotifyPropertyChanged

public event PropertyChangedEventHandler PropertyChanged;

void NotifyPropertyChanged(String info)  
{  
 if (PropertyChanged != null)  
 {

PropertyChanged(this, new PropertyChangedEventArgs(info));  
 }  
}

And for **Name** and **IsConnected** properties, it calls the **NotifyPropertyChanged** method for tell that it has been set to a new value.

public string Name  
{  
 get { return name; }  
 private set  
 {  
 if (value != this.name)  
 {  
 this.name = value;  
 NotifyPropertyChanged("Name");  
 }  
 }  
}

1. The Channel class needs the IrcClient to join a channel of the server or to send messages so it gets it in the **Constructor**.

public Channel(IrcClient server)  
{  
 this.client = server;  
 this.isConnected = false;  
}

1. To Join a channel, we start by adding a ‘#’ if the channel name received don’t have one (since it won’t work otherwise).

if (!name.StartsWith("#"))  
{  
 name = '#' + name;  
}

* 1. Now we use the same trick as to connect to a server with the ManualResetEvent but in the EventHandler, we will check if the **Joined** event concerns our channel and get the channel objet returned by the API in addition to open the lock.

EventHandler<IrcChannelEventArgs> connected = (s, e) =>  
{  
 if (e.Channel.Name == name)  
 {  
 this.channel = e.Channel;  
 connectionEvent.Set();  
 }  
};

* 1. If we succeed to join the channel, we set our properties and subscribe to the channel.MessageReceived event to get new messages from this channel.

client.LocalUser.JoinedChannel -= connected;  
 IsConnected = true;  
 Name = this.channel.Name;  
          Messages = new ObservableCollection<Tuple<string, string>>();  
 this.channel.MessageReceived += messageReceived;

* 1. The **messageReceived** method will add the Tuple see in the Program.cs file in the **Messages** collection.

void messageReceived(object sender, IrcMessageEventArgs e)  
 { this.Messages.Add(new Tuple<string, string>(e.Source.Name, e.Text));  
 }

1. For write a message on the channel, we use the same method as for sending a message to a user because a channel is in fact a **target** as another (it can also be a server, a service, etc.).

public void Say(string message)  
{

client.LocalUser.SendMessage(channel, message);  
}

That’s all!