DarkRift

Server Plugin Tutorial

Introduction

This tutorial will guide you through the process of writing server plugins. It will introduce you to the server's inner architecture and will give you a good understanding about how to go about writing your own.

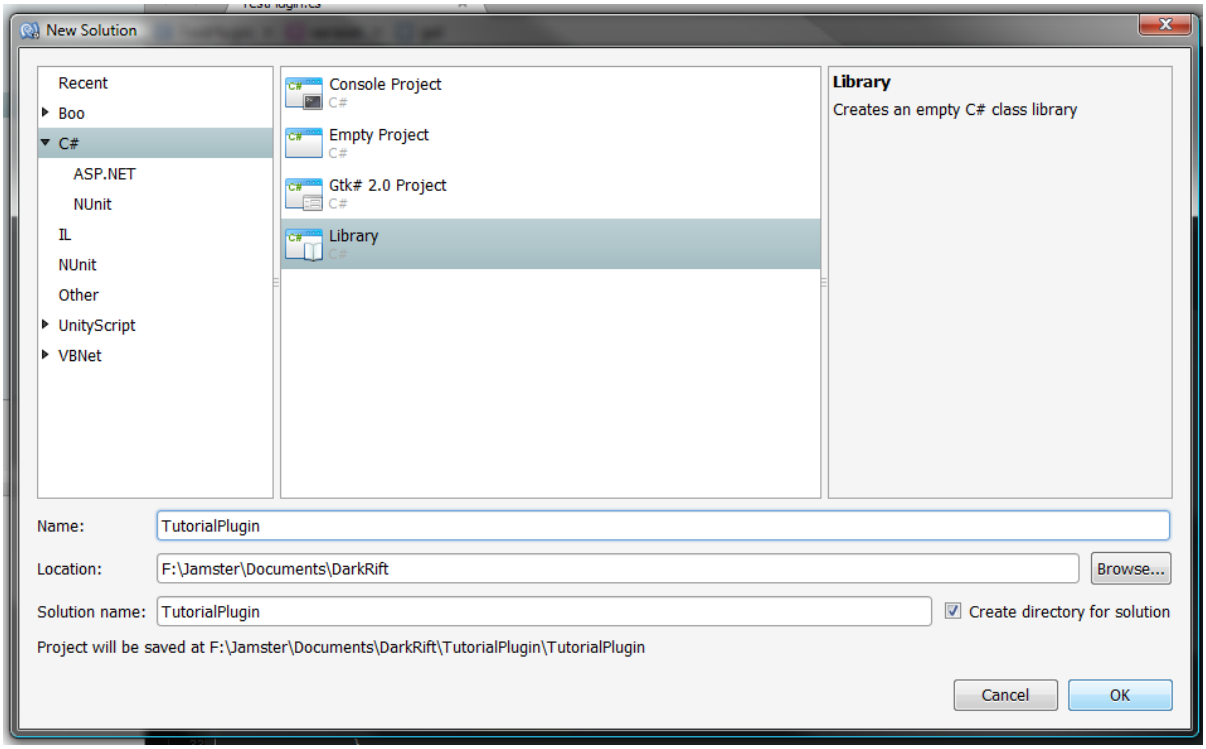
By the end of this tutorial you will have a plugin written that will say "Hello World" (As always!) and will log all the messages coming through the server but only after you've issued a command!

Remember if you're unsure after this tutorial you can always find out more about the APIs in the references!

Setup

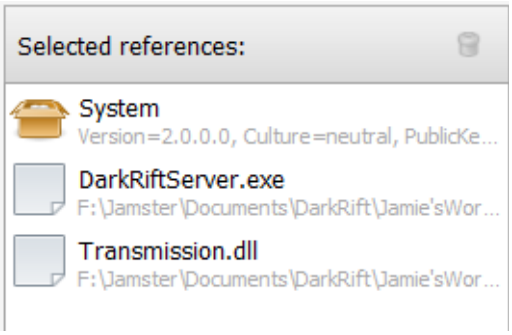
Creating a new project

1. Create a new solution in Monodevelop by going to File > New > Solution
2. Select C# in the left column and then Library in the right.
3. Give your project a name and press ok.



Setting up references

4. In the 'Solution' sub-window in Monodevelop (left hand side for me) right click references and press 'Edit References...'
5. Select the .NET Assembly tab and browse for the server executable and Transmission.dll and add them using the 'Add' button near the bottom.
6. Check they have been added to the right hand column and press ok.



Hello World!

Making us a plugin

1. Firstly, our class needs to be able to access the classes in DarkRift so add the following line near the top.

```
using DarkRift;
```

2. To be seen by the server as a plugin we need a class inheriting from DarkRift.Plugin so replace the class declaration with:

```
public class TutorialPlugin : Plugin
```

3. Finally we need to have certain fields defined for our plugin to work properly. Add the following inside the class.

```
public override string name{              get{ return "Tutorial plugin";              }}
public override string version{           get{ return "1.0";                          }}
public override Command[] commands{get{ return new Command[0];                        }}
public override string author{            get{ return "<Your name>";                  }}
public override string supportEmail{      get{ return "example@example.com";          }}
```

Most of the fields should be fairly self explanatory, we'll come to commands later.

4. You should have something similar to this:

```
using System;

using DarkRift;

namespace TutorialPlugin
{
    public class TutorialPlugin : Plugin
    {
        public override string name{        get{ return "Tutorial plugin";      }}
        public override string version{     get{ return "1.0";                  }}
        public override Command[] commands{ get{ return new Command[0];         }}
        public override string author{      get{ return "<Your name>";          }}
        public override string supportEmail{get{ return "example@example.com";  }}

        public MyClass ()
        {
        }
    }
}
```

Saying Hello!

When our plugin is loaded a new instance of it is created, this means any code we want to run at startup needs to be in the constructor.

5. Update the constructor to have the same name as the class.

To say something on the console you need to access DarkRift.Interface. All user output should go through this rather than System.Console etc so that it is correctly displayed.

6. Add the following inside the constructor:

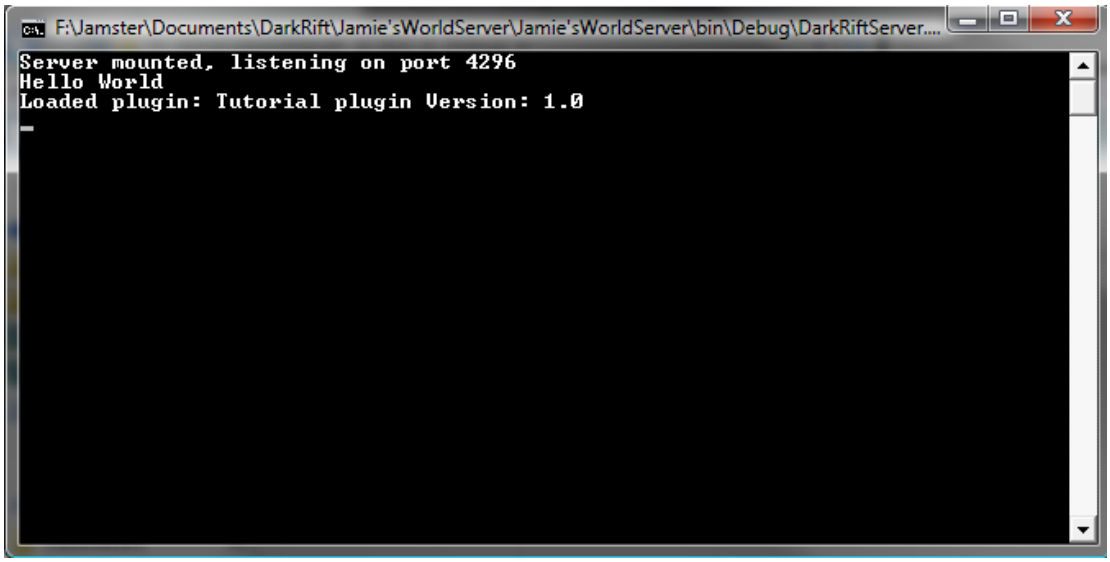
```
Interface.Log("Hello World");
```

7. You should have something along these lines:

```
public TutorialPlugin ()
{
    Interface.Log ("Hello World");
}
```

Testing

8. Build the plugin by going to Build > Build All or by pressing F8.
9. Open the directory it built to (most likely where you saved > the solution name > bin > Debug) and copy the .dll file you created (mine being TutorialPlugin.dll) to the Plugins folder of your server.



Congratulations you've made a plugin! Let's move on to something trickier...!

A Simple Logger

Getting data

Your plugin can attach onto events, this allows it to be told when data is received, when players connect/disconnect, at specific intervals, etc. This is the main way you're plugin is going to process data so it's a key skill to learn.

1. Create a new method called OnDataReceived with parameters ConnectionService con and a reference to NeworkMessage msg that returns void.
2. Inside add the line:

```
Interface.Log("Received data from " + msg.senderID.ToString() );
```

ConnectionServices hold the connection to a client; you can send and receive messages through these and performs various other operations – have a look in the Server Reference for more details.

NetworkMessages represent a message, they have a sender, a tag, a subject, data, a distribution type (who the server should pass it on to) and, if the distribution type is set to ID, and ID to pass it to. As we are passed a reference you can edit this message if you want and it will be re-encoded and sent.

3. In the constructor add the line:

```
ConnectionService.onData += OnDataReceived;
```

This means that whenever data is received the method you just wrote will be called and passed the ConnectionService the message originated from and the message itself. The onData event doesn't have the data decoded (use onDataDecoded if you need it) - only the header is available (tag, subject, distribution type, sender, destination) so the server doesn't have to waste time re-encoding lengthy data.

```
using System;

using DarkRift;

namespace TutorialPlugin
{
    public class TutorialPlugin : Plugin
    {
        public override string name{        get{ return "Tutorial plugin";      }}
        public override string version{     get{ return "1.0";                  }}
        public override Command[] commands{ get{ return new Command[0];         }}
        public override string author{      get{ return "<Your name>";          }}
        public override string supportEmail{get{ return "example@example.com";  }}

        public TutorialPlugin ()
        {
            Interface.Log ("Hello World");
            ConnectionService.onData += OnDataRecieved;
        }

        public void OnDataRecieved(ConnectionService con, ref NetworkMessage msg){
            Interface.Log("Recieved data from " + msg.senderID.ToString() );
        }
    }
}
```

Heads up!
When you start using the events you should make sure all your code is thread safe; this example isn't going to cause problems if 2 threads call it at the same time but keep it in mind when you write your own and remember to lock{} things!

4. Build your plugin, move the file and run the server. Connect to the server using one of the examples provided and everytime data is received you should see a message appear on screen.

Commands & Data Alterations

Getting Commands

In the same way you tell the server to “LogPerformance” or “Stop” you can tell plugins to do certain thing via command. This is how you should be managing your servers when you don’t want to completely take them offline and change config settings so: here’s how.

1. Add a new method called SetLog_Command that takes a string array called parts.
2. Validate the array to ensure it has a length of 1.
3. Create a Boolean variable called log and if parts[0] is “on” set it true, if it is “off” set it false.
4. You should have something like:

```
bool log;

public void SetLog_Command(string[] parts){
    //Validate
    if (parts.Length != 1) {
        Interface.LogError ("SetLog Command should only have 1 argument: 'on' or 'off'!");
        return;
    }

    if (parts [0] != "on" && parts [0] != "off") {
        Interface.LogError ("SetLog Command's argument should only ever be: 'on' or 'off'!");
        return;
    }

    log = (parts [0] == "on") ? true : false;
}
```

5. Change the implementation of the commands variable we did earlier to call our method when the user types “SetLog” by changing it to:

```
public override Command[] commands{
  get{
     return new Command[]{
        new Command("SetLog", "Turns data logging on or off.", SetLog_Command)
     };
  }
}
```

This registers a command with the Interface when the plugin is loaded meaning you can now call “SetLog” from the console and it will call the SetLog_Command function!

6. Finally change the OnDataReceived method so it only logs when log is true:

```
if( log )
    Interface.Log("Received data from " + msg.senderID.ToString() );
```

7. Now you should be able to run your server and a client of whatever you like and when you type “SetLog on” you should see all the data passing through!

Standards

DarkRift plugins should be redistributable, I should be able to write a plugin that I can send to any one of you and it'll work just as it did for me. What this means is that we should have a standard way of writing plugins and interfacing with them. In simpler words: Stick to the following rules and everyone will be happy!

Write both server and client APIs

If you distribute a plugin people are probably going to send it messages from their clients therefore you should try to simplify their life by providing a client API.

In a room plugin example there may be a tag/subject combination that caused the plugin to assign them to a room and send them back the room's ID. Hence you should provide a method that does this (something like *public static void RoomPlugin.JoinRoom()*) and then also handle the return so that the user can access the room's id through your API (something like *public static ushort roomID*).

Now a fellow plugin may also want to query which room a specific player is in and so therefore you should provide some functionality for this. Generally you should avoid storing data about a player in your plugin and instead should use ConnectionService.SetData and ConnectionService.GetData to store the data. This allows other plugins to function even if yours isn't there because they don't need to reference your plugin and (in the future) may allow plugins to update without having to restart the server and disconnect all the players in the process.

Needless to say please document both your APIs...

Let the user choose the tags and subjects

If a user installs 2 plugins that both use tag 1, bad stuff will happen. Therefore all tags and subjects should be configurable. I don't care how you do it, just do it.

The best way on the server would be to have a config file that they can set them all in (on that note, please use camel caps in config files!) but on the client side it's completely up to your digression. If you're going open source it can be as simple as a set of constants at the top.