Discord.Net Documentation

Release 0.9.4

RogueException

Sep 23, 2017

Documentation

Discord.Net is an unofficial C# wrapper around the *Discord Chat Service*. It offers several methods to create automated operations, bots, or even custom clients.

Feel free to join us in the Discord API chat.

> **Warning:** This is a beta!
>
> This library has been built thanks to a community effort reverse engineering the Discord client. As the API is still unofficial, it may change at any time without notice, breaking this library as well. Discord.Net itself is still in development (and is currently undergoing a rewrite) and you may encounter breaking changes throughout development until the official Discord API is released.

It is highly recommended that you always use the latest version and please report any bugs you find to our Discord chat.

This Documentation is **currently undergoing a rewrite**. Some pages (marked with a wrench) are not updated, or are not completed yet. Please submit any feedback to `foxbot#0282` on Discord.

CHAPTER 1

Getting Started

Requirements

Discord.Net currently requires logging in with a user account, however Discord will soon require the use of Bot Accounts. You can register a bot account here.

Bot Accounts must be added to a server, you must use the OAuth 2 Flow to add them to servers.

Installation

You can get Discord.Net from NuGet:

- Discord.Net
- Discord.Net.Commands
- **'Discord.Net.Modules'_**
- **'Discord.Net.Audio'_**

If you have trouble installing from NuGet, try installing dependencies manually.

You can also pull the latest source from GitHub

As an alternative, precompiled binaries are available on our Continuous Integration server.

Async

Discord.Net uses C# tasks extensively - nearly all operations return one. It is highly recommended that these tasks be awaited whenever possible. To do so requires the calling method be marked as async, which can be problematic in a console application. An example of how to get around this is provided below.

For more information, go to MSDN's Await-Async section.

First Steps

```
using Discord;

class Program
{
  static void Main(string[] args) => new Program().Start();

  private DiscordClient _client;

  public void Start()
  {
    _client = new DiscordClient();

    _client.MessageReceived += async (s, e) =>
    {
      if (!e.Message.IsAuthor)
        await e.Channel.SendMessage(e.Message.Text);
    };

    _client.ExecuteAndWait(async () => {
      await _client.Connect("aaaaabbbbbbccccddddeeeeeffffffgggggg", TokenType.Bot)
    });
  }
}
```

First Steps Annotated

The above example will be enough for you to create a basic Echo Bot. Keep in mind, echo bots are discouraged in public servers, so make sure your bot is only in your testing server.

That might have been a lot, so let's go through each line.

`using Discord;` - This is the first line, and it declares that we will be using the Discord.Net API in our class.

Next, we create the Program class, as you would generally do in C#.

`static void Main(string[] args) => new Program().Start();` - It's not good practice to run all of your code in one static method, so we create a new non-static instance of `Program` and run the `Start` method.

`private DiscordClient _client;` - Here we define the main `DiscordClient` that we will be using in our project. It's standard convention to name the private DiscordClient `_client`, and I encourage that you do so also.

`public void Start()` - As explained above, it's not good practice to run everything out of Main, so here is our new Main method.

`_client = new DiscordClient();` - Here, we define `_client` as a new `DiscordClient`, so we can begin to use it.

`_client.MessageReceived += async (s, e) => {` - This is a lambda, a feature which allows us to define functions or handlers inline, without creating a new method. Here, we are hooking into the `MessageReceived` event on the DiscordClient. The `async (s, e)` indicates that the lambda will be an async function, and we are passing two parameters, s(ender) and e(vent args) into it.

`if (!e.Message.IsAuthor)` - This ensures that we did not create the message that was received. This helps to keep us from creating an infinite echo bot.

`await e.Channel.SendMessage(e.Message.Text)` - Here, we are sending a message to the channel the message was received in. The contents of the message we are sending is identical to that of the message we received.

`};` - Close up the lambda

`_client.ExecuteAndWait(async () => {` - This invokes the `ExecuteAndWait` function of the DiscordClient, which allows us to run async code in a non-async method, and block the Console app until the Discord Client disconnects. Inside this function, we are creating an async lambda with no parameters (ExecuteAndWait takes an `Action`, so we cannot use and parameters).

`await _client.Connect("aaabbbccc");` - Next, we are going to connect our client, using the bot token that Discord provides us. If you are unsure of how to access your token, see this image.

Finally, we close up our lambdas and program.

CHAPTER 2

Logging

Discord.Net will log all of its events/exceptions using a built-in LogManager. This LogManager can be accessed through DiscordClient.Log

Usage

To handle Log Messages through Discord.Net's Logger, you must hook into the Log.Message<LogMessageEventArgs> Event.

The LogManager does not provide a string-based result for the message, you must put your own message format together using the data provided through LogMessageEventArgs See the Example for a snippet of logging.

Logging Your Own Data

The LogManager included in Discord.Net can also be used to log your own messages.

You can use DiscordClient.Log.Log(LogSeverity, Source, Message, Exception), or one of the shortcut helpers, to log data.

Example: .. code-block:: c#

_client.MessageReceived += async (s, e) { // Log a new Message with Severity Info, Sourced from 'MessageReceived', with the Message Contents. _client.Log.Info("MessageReceived", e.Message.Text, null);

};

Example

```
class Program
{
    private static DiscordBotClient _client;
    static void Main(string[] args)
    {
        var client = new DiscordClient(x =>
        {
      LogLevel = LogSeverity.Info
    });

        _client.Log.Message += (s, e) => Console.WriteLine($"[{e.Severity}] {e.Source}
↪: {e.Message}");

        client.ExecuteAndWait(async () =>
        {
            await client.Connect("discordtest@email.com", "Password123");
            if (!client.Servers.Any())
                await client.AcceptInvite("aaabbbcccdddeee");
        });
    }
}
```

CHAPTER 3

Server Management

Discord.Net will allow you to manage most settings of a Discord server.

Usage

You can create Channels, Invites, and Roles on a server using the CreateChannel, CreateInvite, and CreateRole function of a Server, respectively.

You may also edit a server's name, icon, and region.

```
// Create a Channel and retrieve the Channel object
var _channel = await _server.CreateChannel("announcements", ChannelType.Text);

// Create an Invite and retrieve the Invite object
var _invite = await _server.CreateInvite(maxAge: null, maxUses: 25, tempMembership:␣
↪false, withXkcd: false);

// Create a Role and retrieve the Role object
var _role = await _server.CreateRole(name: "Bots", permissions: null, color: Color.
↪DarkMagenta, isHoisted: false);

// Edit a server
var _ioStream = new System.IO.StreamReader("clock-0500-1952.png").BaseStream
_server.Edit(name: "19:52 | UTC-05:00", region: "east", icon: _ioStream, iconType:␣
↪ImageType.Png);

// Prune Users
var _pruneCount = await _server.PruneUsers(30, true);
```

Invite Parameters

maxAge: The time (in seconds) until the invite expires. Use null for infinite. maxUses: The maximum amount of uses the invite has before it expires. tempMembership: Whether or not to kick a user when they disconnect. withXkcd: Generate the invite with an XKCD 936 style URL

Role Parameters

name: The name of the role permissions: A set of ServerPermissions for the role to use by default color: The color of the role, recommended to use Discord.Color isHoisted: Whether a role's users should be displayed separately from other users in the user list.

Edit Parameters

name: The server's name region: The region the voice server is hosted in icon: A System.IO.Stream that will read an image file iconType: The type of image being sent (png/jpeg).

CHAPTER 4

User Management

Banning

To ban a user, invoke the Ban function on a Server object.

```
_server.Ban(_user, 30);
```

The pruneDays parameter, which defaults to 0, will remove all messages from a user dating back to the specified amount of days.

Kicking

To kick a user, invoke the Kick function on the User.

```
_user.Kick();
```

CHAPTER 5

Permissions

There are two types of permissions: *Channel Permissions* and *Server Permissions*.

Permission Overrides

Channel Permissions are expressed using an enum, `PermValue`.

The three states are fairly straightforward -

`PermValue.Allow`: Allow the user to perform a permission. `PermValue.Deny`: Deny the user to perform a permission. `PermValue.Inherit`: The user will inherit the permission from its role.

Channel Permissions

Channel Permissions are controlled using a set of flags:

Flag	Type	Description
AttachFiles	Text	Send files to a channel.
Connect	Voice	Connect to a voice channel.
CreateInstantInvite	General	Create an invite to the channel.
DeafenMembers	Voice	Prevent users of a voice channel from hearing other users (server-wide).
EmbedLinks	Text	Create embedded links.
ManageChannel	General	Manage a channel.
ManageMessages	Text	Remove messages in a channel.
ManagePermissions	General	Manage the permissions of a channel.
MentionEveryone	Text	Use @everyone in a channel.
MoveMembers	Voice	Move members around in voice channels.
MuteMembers	Voice	Mute users of a voice channel (server-wide).
ReadMessageHistory	Text	Read the chat history of a voice channel.
ReadMessages	Text	Read any messages in a text channel; exposes the text channel to users.
SendMessages	Text	Send messages in a text channel.
SendTTSMessages	Text	Send TTS messages in a text channel.
Speak	Voice	Speak in a voice channel.
UseVoiceActivation	Voice	Use Voice Activation in a text channel (for large channels where PTT is preferred)

Each flag is a PermValue; see the section above.

Setting Channel Permissions

To set channel permissions, create a new `ChannelPermissionOverrides`, and specify the flags/values that you want to override.

Then, update the user, by doing `Channel.AddPermissionsRule(_user, _overwrites);`

Roles

Accessing/modifying permissions for roles is done the same way as user permissions, just using the overload for a Role. See above sections.

Server Permissions

Server Permissions can be viewed with `User.ServerPermissions`, but **at the time of this writing** cannot be set.

A user's server permissions also contain the default values for it's channel permissions, so the channel permissions listed above are also valid flags for Server Permissions. There are also a few extra Server Permissions:

Flag	Type	Description
BanMembers	Server	Ban users from the server.
KickMembers	Server	Kick users from the server. They can still rejoin.
ManageRoles	Server	Manage roles on the server, and their permissions.
ManageChannels	Server	Manage channels that exist on the server (add, remove them)
ManageServer	Server	Manage the server settings.

Example

```
// Find a User's Channel Permissions
var UserPerms = _channel.GetPermissionsRule(_user);

// Set a User's Channel Permissions

var NewOverwrites = new ChannelPermissionOverrides(sendMessages: PermValue.Deny);
await channel.AddPermissionsRule(_user, NewOverwrites);
```

CHAPTER 6

Commands

This page is incomplete. While the information below is accurate, it should be noted that it is not thorough.

The Discord.Net.Commands package DiscordBotClient extends DiscordClient with support for commands.

Setup

To use Commands, you must first install the CommandService to your DiscordClient.

```
_client.UsingCommands(x => {
        x.PrefixChar = '$';
        x.HelpMode = HelpMode.Public;
});
```

By default, your bot will also respond to @mentions. It is reccomended you leave this feature enabled, and in crowded servers, require a mention to trigger your bot.

Example (Simple)

```
//Since we have setup our CommandChar to be '~', we will run this command by typing ~
↪greet
_client.GetService<CommandService>().CreateCommand("greet") //create command greet
        .Alias(new string[] { "gr", "hi" }) //add 2 aliases, so it can be run with ~
↪gr and ~hi
        .Description("Greets a person.") //add description, it will be shown when ~
↪help is used
        .Parameter("GreetedPerson", ParameterType.Required) //as an argument, we have␣
↪a person we want to greet
        .Do(async e =>
        {
            await e.Channel.SendMessage($"{e.User.Name} greets {e.GetArg(
↪"GreetedPerson")}");
```

```
            //sends a message to channel with the given text
        });
```

Example (Groups)

```
//we would run our commands with ~do greet X and ~do bye X
_client.GetService<CommandService>().CreateGroup("do", cgb =>
        {
            cgb.CreateCommand("greet")
                    .Alias(new string[] { "gr", "hi" })
                    .Description("Greets a person.")
                    .Parameter("GreetedPerson", ParameterType.Required)
                    .Do(async e =>
                    {
                        await e.Channel.SendMessage($"{e.User.Name} greets {e.GetArg(
↪"GreetedPerson")}");
                    });

            cgb.CreateCommand("bye")
                    .Alias(new string[] { "bb", "gb" })
                    .Description("Greets a person.")
                    .Parameter("GreetedPerson", ParameterType.Required)
                    .Do(async e =>
                    {
                        await e.Channel.SendMessage($"{e.User.Name} says goodbye to
↪{e.GetArg("GreetedPerson")}");
                    });
        });
```

CHAPTER 7

Voice

Installation

Before setting up the AudioService, you must first install the package from NuGet or GitHub.

Add the package to your solution, and then import the namespace `Discord.Audio`.

Discord.Audio also relies on the Opus Audio Library for encoding audio. You must acquire a compiled binary of Opus, and place it in the directory your project runs from (/ on dnx, /bin/debug on net45). A 32-bit binary is available for your convienence.

You will also need libsodium for voice encryption. You must acquire a compiled binary of libsodium, and place it in the directory your project runs from. A precompiled binary is also available here. You may also find 64-bit/linux releases from here.

Setup

To use audio, you must install the AudioService to your DiscordClient.

```
var _client = new DiscordClient();

_client.UsingAudio(x => // Opens an AudioConfigBuilder so we can configure our␣
↪AudioService
{
        x.Mode = AudioMode.Outgoing; // Tells the AudioService that we will only be␣
↪sending audio
});
```

Joining a Channel

Joining Voice Channels is pretty straight-forward, and is required to send Audio. This will also allow us to get an IAudioClient, which we will later use to send Audio.

```
var voiceChannel = _client.FindServers("Music Bot Server").FirstOrDefault().
↪VoiceChannels.FirstOrDefault(); // Finds the first VoiceChannel on the server
↪'Music Bot Server'

var _vClient = await _client.GetService<AudioService>() // We use GetService to find␣
↪the AudioService that we installed earlier. In previous versions, this was␣
↪equivelent to _client.Audio()
        .Join(voiceChannel); // Join the Voice Channel, and return the IAudioClient.
```

The client will sustain a connection to this channel until it is kicked, disconnected from Discord, or told to Disconnect.

The IAudioClient

The IAudioClient is used to connect/disconnect to/from a Voice Channel, and to send audio to that Voice Channel.

`IAudioClient.Disconnect();`
: Disconnects the IAudioClient from the Voice Server.

`IAudioClient.Join(Channel);`
: Moves the IAudioClient to another channel on the Voice Server, or starts a connection if one has already been terminated.

Note: Because versions previous to 0.9 do not discretely differentiate between Text and Voice Channels, you may want to ensure that users cannot request the audio client to join a text channel, as this will throw an exception, leading to potentially unexpected behavior

---

`IAudioClient.Wait();`
: Blocks the current thread until the sending audio buffer has cleared out.

`IAudioClient.Clear();`
: Clears the sending audio buffer.

`IAudioClient.Send(byte[] data, int offset, int count);`
: Adds a stream of data to the Audio Client's internal buffer, to be sent to Discord. Follows the standard c# Stream.Send() format.

Broadcasting

There are multiple approaches to broadcasting audio. Discord.Net will convert your audio packets into Opus format, so the only work you need to do is converting your audio into a format that Discord will accept. The format Discord takes is 16-bit 48000Hz PCM.

Broadcasting with NAudio

NAudio is one of the easiest approaches to sending audio, although it is not multi-platform compatible. The following example will show you how to read an mp3 file, and send it to Discord. You can download NAudio from NuGet.

```
using NAudio;
using NAudio.Wave;
using NAudio.CoreAudioApi;

public void SendAudio(string filePath)
{
        var channelCount = _client.GetService<AudioService>().Config.Channels; // Get␣
↪the number of AudioChannels our AudioService has been configured to use.
        var OutFormat = new WaveFormat(48000, 16, channelCount); // Create a new␣
↪Output Format, using the spec that Discord will accept, and with the number of␣
↪channels that our client supports.
        using (var MP3Reader = new Mp3FileReader(filePath)) // Create a new␣
↪Disposable MP3FileReader, to read audio from the filePath parameter
        using (var resampler = new MediaFoundationResampler(MP3Reader, OutFormat)) //␣
↪Create a Disposable Resampler, which will convert the read MP3 data to PCM, using␣
↪our Output Format
        {
                resampler.ResamplerQuality = 60; // Set the quality of the resampler␣
↪to 60, the highest quality
                int blockSize = OutFormat.AverageBytesPerSecond / 50; // Establish␣
↪the size of our AudioBuffer
                byte[] buffer = new byte[blockSize];
                int byteCount;

                while((byteCount = resampler.Read(buffer, 0, blockSize)) > 0) // Read␣
↪audio into our buffer, and keep a loop open while data is present
                {
                        if (byteCount < blockSize)
                        {
                                // Incomplete Frame
                                for (int i = byteCount; i < blockSize; i++)
                                        buffer[i] = 0;
                        }
                        _vClient.Send(buffer, 0, blockSize); // Send the buffer to␣
↪Discord
                }
        }

}
```

Broadcasting with FFmpeg

FFmpeg allows for a more advanced approach to sending audio, although it is multiplatform safe. The following example will show you how to stream a file to Discord.

```
public void SendAudio(string pathOrUrl)
{
        var process = Process.Start(new ProcessStartInfo { // FFmpeg requires us to␣
↪spawn a process and hook into its stdout, so we will create a Process
                FileName = "ffmpeg",
```

```
                Arguments = $"-i {pathOrUrl} " + // Here we provide a list of␣
↪arguments to feed into FFmpeg. -i means the location of the file/URL it will read␣
↪from
                        "-f s16le -ar 48000 -ac 2 pipe:1", // Next, we tell it to␣
↪output 16-bit 48000Hz PCM, over 2 channels, to stdout.
                UseShellExecute = false,
                RedirectStandardOutput = true // Capture the stdout of the process
        });
        Thread.Sleep(2000); // Sleep for a few seconds to FFmpeg can start processing␣
↪data.

        int blockSize = 3840; // The size of bytes to read per frame; 1920 for mono
        byte[] buffer = new byte[blockSize];
        int byteCount;

        while (true) // Loop forever, so data will always be read
        {
                byteCount = process.StandardOutput.BaseStream // Access the␣
↪underlying MemoryStream from the stdout of FFmpeg
                        .Read(buffer, 0, blockSize); // Read stdout into the buffer

                if (byteCount == 0) // FFmpeg did not output anything
                        break; // Break out of the while(true) loop, since there was␣
↪nothing to read.

                _vClient.Send(buffer, 0, byteCount); // Send our data to Discord
        }
        _vClient.Wait(); // Wait for the Voice Client to finish sending data, as␣
↪ffMPEG may have already finished buffering out a song, and it is unsafe to return␣
↪now.
}
```

Note: The code-block above assumes that your client is configured to stream 2-channel audio. It also may prematurely end a song. FFmpeg can — especially when streaming from a URL — stop to buffer data from a source, and cause your output stream to read empty data. Because the snippet above does not safely track for failed attempts, or buffers, an empty buffer will cause playback to stop. This is also not 'memory-friendly'.

Multi-Server Broadcasting

Warning: Multi-Server broadcasting is not supported by Discord, will cause performance issues for you, and is not encouraged. Proceed with caution.

To prepare for Multi-Server Broadcasting, you must first enable it in your config.

From here on, it is as easy as creating an IAudioClient for each server you want to join. See the sections on broadcasting to proceed.

Receiving

Receiving is not implemented in the latest version of Discord.Net

CHAPTER 8

Events

Usage

Messages from the Discord server are exposed via events on the DiscordClient class and follow the standard EventHandler<EventArgs> C# pattern.

> **Warning:** Note that all synchronous code in an event handler will run on the gateway socket's thread and should be handled as quickly as possible. Using the async-await pattern to let the thread continue immediately is recommended and is demonstrated in the examples below.

Connection State

Connection Events will be raised when the Connection State of your client changes.

> **Warning:** You should not use DiscordClient.Connected to run code when your client first connects to Discord. If you lose connection and automatically reconnect, this code will be ran again, which may lead to unexpected behavior.

Messages

- MessageReceived, MessageUpdated and MessageDeleted are raised when a new message arrives, an existing one has been updated (by the user, or by Discord itself), or deleted.
- MessageAcknowledged is only triggered in client mode, and occurs when a message is read on another device logged-in with your account.

Example of MessageReceived:

```
// (Preface: Echo Bots are discouraged, make sure your bot is not running in a public␣
↪server if you use them)

// Hook into the MessageReceived event using a Lambda
_client.MessageReceived += async (s, e) => {
        // Check to make sure that the bot is not the author
        if (!e.Message.IsAuthor)
            // Echo the message back to the channel
            await e.Channel.SendMessage(e.Message);
};
```

Users

There are several user events:

- UserBanned: A user has been banned from a server.
- UserUnbanned: A user was unbanned.
- UserJoined: A user joins a server.
- UserLeft: A user left (or was kicked from) a server.
- UserIsTyping: A user in a channel starts typing.
- UserUpdated: A user object was updated (presence update, role/permission change, or a voice state update).

Note: UserUpdated Events include a `User` object for Before and After the change. When accessing the User, you should only use `e.Before` if comparing changes, otherwise use `e.After`

Examples:

```
// Register a Hook into the UserBanned event using a Lambda
_client.UserBanned += async (s, e) => {
    // Create a Channel object by searching for a channel named '#logs' on the server␣
↪the ban occurred in.
    var logChannel = e.Server.FindChannels("logs").FirstOrDefault();
    // Send a message to the server's log channel, stating that a user was banned.
    await logChannel.SendMessage($"User Banned: {e.User.Name}");
};

// Register a Hook into the UserUpdated event using a Lambda
_client.UserUpdated += async (s, e) => {
    // Check that the user is in a Voice channel
    if (e.After.VoiceChannel == null) return;

    // See if they changed Voice channels
    if (e.Before.VoiceChannel == e.After.VoiceChannel) return;

    await logChannel.SendMessage($"User {e.After.Name} changed voice channels!");
};
```