RITlug TeleIRC Documentation

Release 2.0.0-pre2

TeleIRC community

Mar 05, 2023

ABOUT TELEIRC:

Note: This documentation is for v2.x.x releases of TeleIRC. See the v1.3.4 documentation for support with older releases.

RITlug TeleIRC is a Go implementation of a Telegram <=> IRC bridge. TeleIRC works with any IRC channel and Telegram group. It bridges messages and images between a Telegram group and an IRC channel.

This bot was originally written for RITlug. Today, it is used by various communities.

CHAPTER

ONE

WHO USES TELEIRC?

1.1 v2.x.x

The following projects and communities use RITlug TeleIRC v2.0.0 or later:

- Arch Linux Mexico (@archlinuxmx | #archlinux-mx)
- Chaskis IRC Bot Framework (@ChaskisIrc | #chaskis)
- mIRC Italia Crew (@mircitaliacrew | #mircitaliacrew)

1.2 v1.x.x

The following projects and communities use a v1.x.x release of RITlug TeleIRC:

- BSD Argentina (@bsdar | #bsdar)
- CityZen app (@CityZenApp | #cityzen)
- Fedora Project (@fedora | #fedora-telegram)
  - Fedora Albania (@FedoraAlbania | #fedora-sq)
  - Fedora CommOps (@fedoracommops | #fedora-commops)
  - Fedora Diversity and Inclusion Team (@fedoradiversity | #fedora-diversity)
  - Fedora LATAM (@fedoralat | #fedora-latam)
  - Fedora Marketing Team (@fedoramktg | #fedora-mktg)
  - Flock to Fedora (@flocktofedora | #fedora-flock)
- FOSS@MAGIC at RIT (@fossrit | #rit-foss)
- Hamara Linux (@hamaraLinux | #hamara)
- LibreOffice Community (@libreofficecommunity | #libreoffice-telegram)
  - LibreLadies (*Telegram invite-only* | #libreladies)
  - LibreOffice AppImage (*no public invite link* | #libreoffice-appimage)
  - LibreOffice Design Team (*no public invite link* | #libreoffice-design)
  - LibreOffice QA Team (*no public invite link* | #libreoffice-qa)
- Linux Users Massa Carrara (Telegram | IRC)
- MINECON Agents (*no public invite link* | #MineconAgents)

- MusicBrainz (@musicbrainz | #musicbrainz-telegram)
- Pure Data (@puredata | #dataflow)
- RITlug (@ritlug | #rit-lug)
  - RITlug teleirc (@teleirc | #rit-lug-teleirc)

1.3 How to get on this list

Want to have your community added to this page? Let us know you are using TeleIRC too! Open an issue with the following info:

- Organization / group name and website
- Telegram group URL
- Your IRC channel

To be added, your group must not discuss illegal, illicit, or sexually explicit content.

CHAPTER

TWO

CONFIG FILE GLOSSARY

This page is a glossary of different settings in the `env.example` configuration file. All values shown are the default settings. This glossary is intended for advanced users.

2.1 IRC settings

2.1.1 Host connection settings

`IRC_HOST_IP=""` Specify IP address to use for IRC connection. Useful for hosts with multiple IP addresses.

2.1.2 Server connection settings

`IRC_SERVER=chat.freenode.net` IRC server to connect to

`IRC_SERVER_PASSWORD=""` IRC server password

`IRC_PORT=6697` IRC server port

Encryption options

`IRC_USE_SSL=true` Connect to the IRC server with SSL

`IRC_CERT_ALLOW_EXPIRED=false` Allow connecting to IRC server with an expired TLS/SSL certificate

`IRC_CERT_ALLOW_SELFSIGNED=false` Allows TeleIRC to accept TLS/SSL certificates from non-trusted/unknown Certificate Authorities (CA)

2.1.3 Channel settings

`IRC_CHANNEL="#channel"` IRC channel for bot to join

`IRC_CHANNEL_KEY=""` IRC channel key, for password-protected channels

`IRC_BLACKLIST=""` Comma-separated list of IRC nicks to ignore

2.1.4 Bot settings

IRC_BOT_NAME=teleirc IRC nickname for bot. Most IRC clients and bridges show this nickname.

IRC_BOT_REALNAME="Powered by TeleIRC <github.com/RITlug/teleirc>" IRC `REALNAME` for bot. Often visible in IRC `/whois` reports, but most clients do not show this information by default.

IRC_BOT_IDENT=teleirc Identification metadata for bot connection. This is rarely used or shown in most IRC clients and bridges, so only change this if you know what you are doing. Often visible in IRC `/whois` reports, but most clients do not show this information by default.

NickServ options

IRC_NICKSERV_SERVICE=NickServ IRC service used for authentication.

IRC_NICKSERV_USER="" IRC NickServ username.

IRC_NICKSERV_PASS="" IRC NickServ password.

2.1.5 Message settings

IRC_PREFIX="<" Text displayed before Telegram name in IRC

IRC_SUFFIX=">" Text displayed after Telegram name in IRC

IRC_SEND_STICKER_EMOJI=true Send emojis associated with a sticker to IRC (when a Telegram user sends a sticker)

IRC_SEND_DOCUMENT=false Send documents and files from Telegram to IRC (why is this false by default?)

IRC_EDITED_PREFIX="(edited) " Prefix to prepend to messages when a user edits a Telegram message and it is resent to IRC

IRC_MAX_MESSAGE_LENGTH=400 Maximum length of the message that can be sent to IRC. Longer messages are split into multiple messages.

IRC_SHOW_ZWSP=true Prevents users with the same Telegram and IRC username from pinging themselves across platforms.

`IRC_SHOW_LOCATION_MESSAGE=false` If a user shares their location on Telegram, this will forward the GPS coordinates of their location to IRC if set to true.

IRC_NO_FORWARD_PREFIX="[off]" A string users can prefix their message with to prevent it from being relayed across the bridge. Removing this option or setting it to "" disables it.

IRC_QUIT_MESSAGE="TeleIRC bridge stopped." A string that TeleIRC sends to the IRC channel when the application exits using IRC's "QUIT" command. If not specified, it simply closes the connection without providing a reason. The bot must be connected to a server for a certain amount of time for the server to send the quit message to the channel.

2.2 Telegram settings

TELEGRAM_CHAT_ID=-0000000000000 Telegram chat ID of bridged group (*how do I get this?*).

TELEIRC_TOKEN=000000000:AAAAAAaAAa2AaAAaoAAAA-a_aaAAaAaaaAA Private API token for Telegram bot.

MAX_MESSAGES_PER_MINUTE=20 Maximum number of messages sent to Telegram from IRC per minute.

``**TELEGRAM_MESSAGE_REPLY_PREFIX="["** Prefix separator for Telegram reply

``**TELEGRAM_MESSAGE_REPLY_SUFFIX="]"** Suffix separator for Telegram reply

``**TELEGRAM_MESSAGE_REPLY_LENGTH=15** Length of quoted reply message before truncation

SHOW_TOPIC_MESSAGE=true Send Telegram message when the topic in the IRC channel is changed.

SHOW_ACTION_MESSAGE=true Relay action messages (e.g. `/me thinks TeleIRC is cool!`).

SHOW_JOIN_MESSAGE=false Send Telegram message when someone joins IRC channel.

JOIN_MESSAGE_ALLOW_LIST="" List of users (separated by a space character) whose IRC leave messages will be sent to Telegram, even if SHOW_JOIN_MESSAGE is false. This is ignored if SHOW_JOIN_MESSAGE is set to true.

SHOW_KICK_MESSAGE=true Send Telegram message when someone is kicked from IRC channel.

SHOW_NICK_MESSAGE=false Send Telegram message when someone changes their nickname in the IRC channel.

SHOW_LEAVE_MESSAGE=false Send Telegram message when someone leaves IRC channel either by quitting or parting.

LEAVE_MESSAGE_ALLOW_LIST="" List of users (separated by a space character) whose IRC leave messages will be sent to Telegram, even if SHOW_LEAVE_MESSAGE is false. This is ignored if SHOW_LEAVE_MESSAGE is set to true.

SHOW_DISCONNECT_MESSAGE=true Sends a message to Telegram when the bot disconnects from the IRC side.

2.3 Imgur settings

IMGUR_CLIENT_ID=7d6b00b87043f58 Imgur API client ID value to access Imgur API. Uses a default client ID. If you are bridging to a very active Telegram group, *please :ref:`register your own application <imgur-setup>`*.

IMGUR_CLIENT_SECRET="" Imgur API client secret. Only needed when *uploading to an account*.

IMGUR_REFRESH_TOKEN="" Imgur API refresh_token for the account where images should be uploaded. Only needed when *uploading to an account*.

IMGUR_ALBUM_HASH="" The album hash for the Imgur album uploaded images should belong to. *How to find this*

CHAPTER

THREE

FREQUENTLY ASKED QUESTIONS

This page collects frequently asked scenarios or problems with TeleIRC. Did you find something confusing? Please let us know in our developer chat or open a new pull request with a suggestion!

3.1 IRC

3.1.1 Messages do not appear in the IRC channel. Why?

There are a lot of things that *could* be the cause. However, make sure the **IRC channel is surrounded by quotes in the `.env` file**:

```
` IRC_CHANNEL="#my-cool-channel" `
```

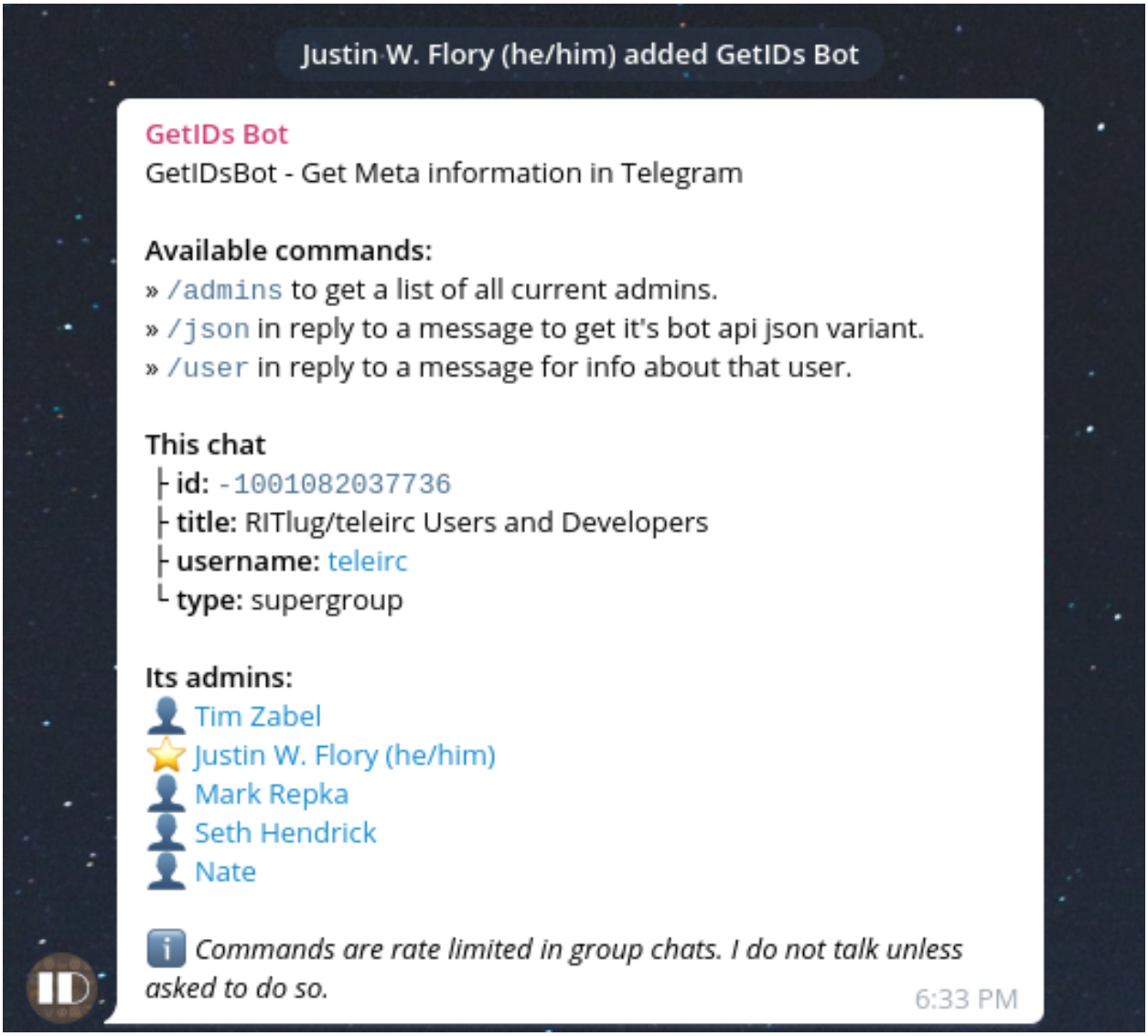
If there are no quotes, the value is interpreted as a comment, or the same thing as if it were an empty string.

3.2 Telegram

3.2.1 How do I find a chat ID for a Telegram group?

There are two ways we suggest finding the chat ID of a Telegram group.

The easiest way is to add the @getidsbot to the group. As soon as the bot joins the group, it will print a message with the group chat ID. You can remove the bot once you get the chat ID.



Another way to get the chat ID is from the Telegram API via a web browser. First, add your bot to the group. Then, open a browser and enter the Telegram API URL with your API token, as explained in this post. Next, send a message in the group that @tags the bot username, and refresh the browser window. You will see the chat ID for the Telegram group along with other information.

3.2.2 I reinstalled TeleIRC after it was inactive for a while. But the bot doesn't work. Why?

If a Telegram bot is not used for a while, it "goes to sleep". Even if TeleIRC is configured and installed correctly, you need to "wake up" the bot. To fix this, *remove the bot from the group and add it again*. Restart TeleIRC and it should work again.

3.2.3 Why do I have to disable privacy on the Telegram bot during setup?

The privacy setting must be disabled for TeleIRC bot to "see" messages in the Telegram group. By default, bots cannot see messages unless a person uses a command to interact directly with a bot. Since TeleIRC forwards all sent messages from Telegram to IRC, it must see all messages to work.

Messages are not stored or tracked by TeleIRC (but may optionally be logged by an administrator).

3.3 Imgur

3.3.1 How do I get an Imgur API Client ID?

If you are bridging to a busy telegram group, please register your own application.

Visit Imgur's *application registration page <https://api.imgur.com/oauth2/addclient>* to create a new application. Choose a name, set the type to "OAuth 2 authorization without a callback URL", and enter your email address and a description for the app. From the next page, copy the Client ID into your configuration file. If you intend to upload to an account (rather than anonymously), also copy your client secret.

These can be retrieved later from your *applications page <https://imgur.com/account/settings/apps>*

3.3.2 How do I upload to my Imgur account, instead of anonymously?

If you only configure a Client ID, uploaded images will not be linked with any account. If you want them to belong to an account, in addition to the Client ID above you will need to put your app's **client secret** into the configuration file and connect TeleIRC to your Imgur account using OAuth2. Replace the *client_id* parameter in this link with yours and then visit it to authorize the app.

```
` https://api.imgur.com/oauth2/authorize?response_type=token&client_id=YOUR_CLIENT_ID
`
```

You will then be redirected to the imgur homepage, with some additional parameters in the URL:

```
` https://imgur.com/#access_token=e3b0...c442&expires_in=315360000&token_type=bearer&refres
..1c14&account_username=... `
```

From these parameters copy the string of characters after *refresh_token*=, in this case *98fc...1c14*, into the *IMGUR_REFRESH_TOKEN* variable in your configuration file, then restart the bridge.

3.3.3 How do I find the album hash?

The easiest way is by signing in to Imgur, clicking your name at the top right, clicking "Images", opening your browser's developer tools, switching to the "Network" tab, and then choosing the album you want from the dropdown on the page.

In the network inspector will be a request for a URL like the following:

```
` https://yourname.imgur.com/ajax/images?sort=3&order=0&album=r7bbAyI&page=1&perPage=60
`
```

In this example, the album hash is *r7bbAyI*.

CHAPTER

FOUR

QUICK START GUIDE

This page is a quick start guide to using TeleIRC. It is an overview of how to install, set up, and deploy TeleIRC v2.x.x releases. Note this does not apply to v1.x.x releases; see the v1.3.4 documentation.

4.1 Contents

4.2 Overview

This section is a written, high-level overview of how to configure and deploy a TeleIRC bot. The Quick Start Guide will cover these topics:

1. *Create a Telegram bot to obtain a Telegram API token*
2. *Set up an IRC channel for best user experience*
3. *Deploy TeleIRC to your system*

It is important each step is followed exactly and in order. Missing a step or skipping a section often results in common frustrations, such as one-way relay of chat messages.

4.2.1 Telegram Channels

TeleIRC **DOES NOT** support Telegram Channels, only groups. Read more on the differences between channels and groups in the Telegram FAQ.

4.3 Create a Telegram bot

TeleIRC requires a Telegram API token in order to access messages in a Telegram group. To obtain a token, someone must register a new Telegram bot. The Telegram bot will appear as the Sending User in Telegram for all IRC messages.

4.3.1 Create bot with BotFather

BotFather is the Telegram bot for creating Telegram bots. **See the official Telegram documentation for how to create a new bot.**

Once you create a new bot, you *must* follow these additional steps (**IN EXACT ORDER**) for TeleIRC:

1. Send `/setprivacy` to @BotFather, change to **Disable** *why?*
2. Add bot to Telegram group to be bridged
3. Send `/setjoingroups` to @BotFather, change to **Disable** *why?*

4.3.2 Optional BotFather tweaks

1. Set a description or add profile picture for your bot
2. Block your bot from being added to more groups (`/setjoingroups`)

4.4 Configure IRC channel

This section explains best practices for configuring IRC channels for TeleIRC. Because IRC networks can run different software, exact instructions may differ depending on your IRC network. So, this section is divided in two ways:

1. High-level overview of how to set up your IRC channel
2. How to actually do it on Freenode IRC network

4.4.1 IRC channel overview

No matter what IRC network you use, TeleIRC developers recommend this IRC channel configuration:

- Register your IRC channel with the IRC network (i.e. `ChanServ`).
- Unauthenticated users may join the channel.
- Only authenticated users may write in the channel (i.e. `NickServ`).
- Any user connecting from a network-recognized gateway (e.g. web chat) with an assigned hostmask automatically receives voice on join (and thus, does not need to authenticate to write in channel).
- TeleIRC bot hostmask automatically receives voice on join (and thus, does not need to authenticate to write in channel).
- IRC channel operators automatically receive operator privilege on join.

4.4.2 Configure a Freenode IRC channel

If your IRC channel is on the Freenode IRC network, use these exact commands to create a channel policy as described above:

1. `/join #channel`
2. `/query ChanServ REGISTER #channel`
3. `/query ChanServ SET #channel GUARD on`
4. `/query ChanServ ACCESS #channel ADD <NickServ account> +AORfiorstv` (*repeat for each IRC user who needs admin access*) (*what do these mean?*)
5. `/query ChanServ SET mlock #channel +Ccnt`
6. `/mode #channel +q $~a`
7. `/query ChanServ ACCESS #channel ADD *!*@gateway/* +V`
8. `/query ChanServ ACCESS #channel ADD *!*@freenode/staff/* +Aiotv`
9. `/query ChanServ ACCESS #channel ADD <bot NickServ account or hostmask> +V`

4.4.3 Configure Imgur Image Upload (IIU)

By default, TeleIRC uploads images sent to the Telegram group to Imgur. Since IRC does not support images, Imgur is an intermediary approach to sending pictures sent on Telegram over to IRC. Note that images will be publicly visible on the Internet if the URL is known. See context for why Imgur is enabled by default.

By default, TeleIRC uses the TeleIRC-registered Imgur API key. We highly recommend registering your own API key in high-traffic channels.

To register your own Imgur API key, follow these steps:

1. Create an Imgur account
2. Register your bot with Imgur API (select *OAuth2 without callback* option)
3. Add provided Imgur client ID to `.env` file

4.5 Deployment Guide

There are two ways to deploy TeleIRC persistently:

1. Run Go binary
2. Run TeleIRC in a container

4.5.1 Run binary

This section explains how to configure and install TeleIRC as a simple executable binary.

NOTE: **This assumes you are building from source.** If you use a pre-built binary from a GitHub Release, skip to Configuration.

Pre-requirements

- git
- go (v1.14 and v1.15 supported)

Packages for these pre-requirements are available on most `*NIX` distributions. Check your distribution documentation for more info on how to install these packages.

Build TeleIRC

This section is only required if you are building a binary from source:

1. Clone repository (`git clone https://github.com/RITlug/teleirc.git`)
2. Enter repository (`cd teleirc/`)
3. Build binary (`./build_binary.sh`)

Configuration

TeleIRC uses godotenv to manage API keys and settings. The config file is a `.env` file. Copy the example file to a production file to get started (`cp env.example .env`). Edit the `.env` file with your API keys and settings.

See *Config file glossary* for detailed information.

Start bot

NOTE: This section is one opinionated way to start and configure TeleIRC. Experienced system administrators may have other preferences and slight deviation is permittable. However *upstream only offers free support for installations that follow our documentation*.

To start the bot, you need to consider the following factors:

1. Where will the binary go?
2. Where is your config file on the system?
3. How will you automate the bot to start-up automatically after a system reboot?

Example Linux setup

NOTE: *Looking for an easier way? Check out the TeleIRC Ansible Role for an automated installation of the following steps.*

Upstream offers a systemd unit file to automate TeleIRC on a Linux system that uses systemd. This example uses the upstream systemd unit file to automatically run TeleIRC on a Linux system.

This example was tested on a CentOS 8 system and is easily adaptable for other `*NIX` distributions. It uses `v2.2.1` as a default:

```
# Download TeleIRC deployment assets from GitHub.
$ curl --location --output ~/teleirc https://github.com/RITlug/teleirc/releases/
↪download/v2.2.1/teleirc-2.2.1-linux-x86_64
$ curl --location --output ~/teleirc.sysusers https://raw.githubusercontent.com/
↪RITlug/teleirc/v2.2.1/deployments/systemd/teleirc.sysusers
$ curl --location --output ~/teleirc.tmpfiles https://raw.githubusercontent.com/
↪RITlug/teleirc/v2.2.1/deployments/systemd/teleirc.tmpfiles
$ curl --location --output ~/teleirc@.service https://raw.githubusercontent.com/
↪RITlug/teleirc/v2.2.1/deployments/systemd/teleirc@.service
$ curl --location --output ~/teleirc.env https://raw.githubusercontent.com/RITlug/
↪teleirc/v2.2.1/env.example

# Install TeleIRC files and user
$ sudo install -Dm755 -o root -g root ~/teleirc /usr/local/bin/teleirc
$ sudo install -Dm644 -o root -g root ~/teleirc.sysusers /etc/sysusers.d/teleirc.conf
$ sudo install -Dm644 -o root -g root ~/teleirc.tmpfiles /etc/tmpfiles.d/teleirc.conf
$ sudo install -Dm644 -o root -g root ~/teleirc@.service /etc/systemd/system/teleirc@.
↪service
$ sudo systemd-sysusers /etc/sysusers.d/teleirc.conf
$ sudo systemd-tmpfiles --create /etc/tmpfiles.d/teleirc.conf
$ sudo install -Dm644 -o root -g root ~/teleirc.env /etc/teleirc/example
$ rm ~/teleirc ~/teleirc.sysusers ~/teleirc.tmpfiles ~/teleirc@.service ~/teleirc.env

# Systems with SELinux ONLY.
$ sudo chcon --type bin_t --user system_u /usr/local/bin/teleirc
$ sudo chcon --type etc_t --user system_u -R /etc/teleirc/
$ sudo chcon --type etc_t --user system_u /etc/sysusers.d/teleirc.conf
$ sudo chcon --type etc_t --user system_u /etc/tmpfiles.d/teleirc.conf
$ sudo chcon --type systemd_unit_file_t --user system_u /etc/systemd/system/teleirc@.
↪service

# Start and enable TeleIRC.
$ sudo systemctl enable --now teleirc@example.service
```

To run multiple instances, create other files in /etc/teleirc/ and enable the service named teleirc@FILENAME.service.

4.6 Run container

Containers are another way to deploy TeleIRC. Dockerfiles and other deployment resources are available in `deployments/`.

NOTE: At time of v2.0.0 release, a container image is available, but mostly untested. Feeling bold and adventurous? Take our `Dockerfile` for a spin and let us know on GitHub how it works for you.

Download Dockerfile (beta)

CHAPTER

FIVE

TELEIRC CODE CONVENTIONS

This page explains TeleIRC coding best practices. This was originally written by Tim Zabel (@Tjzabel).

5.1 Naming Conventions

TeleIRC constitutes two halves: IRC and Telegram. Function names should be agnostic to each platform. Lastly, function names should be consistent across IRC and Telegram where possible.

```
func (*tg Client) SendMessage(msg string) {
    ...
}
```

5.2 Handlers

Handlers are blocks of code responsible for "handling" specific message types. Such handlers should be named appropriately in camelCase.

```
func joinHandler(...)
```

CHAPTER

SIX

CONTRIBUTING GUIDELINES

This guide explains how to contribute to the TeleIRC project. It defines working practices of the development team. This document helps new contributors start working on the project. It is a living document and will change. If you think something could be better, please open an issue with your feedback.

6.1 Table of contents

6.2 Set up a development environment

Contents:

6.2.1 Requirements

To set up a TeleIRC development environment, you need the following:

- Go (v1.13.x or later)
- Telegram account
- IRC client (HexChat recommended)
- For docs: Python 3 (3.6 or later)

6.2.2 Create Telegram bot

Create a Telegram bot using the Telegram BotFather. See the TeleIRC Quick Start for more instructions on how to do this.

6.2.3 Create Telegram group

Create a new Telegram group for testing. Invite the bot user as another member to the group. Configure the Telegram bot to TeleIRC specifications before adding it to the group.

6.2.4 Register IRC channel

Registering an IRC channel is encouraged, but optional. At minimum, you need an unused IRC channel to use for testing. Registering the channel gives you additional privileges as a channel operator (e.g. testing NickServ authentication to join private IRC channels). See your IRC network's documentation on registering a channel.

6.2.5 Configure and run TeleIRC

Change the `env.example` file to `.env`. Change the configuration values to the Telegram bot's tokens. For more help with configuration, see the *Config file glossary*.

6.3 Open a pull request

These guidelines help maintainers review new pull requests. Stick to the guidelines for quicker and easier pull request reviews.

1. We prefer gradual, small changes over sudden, big changes
2. Write a helpful title for your pull request (if someone reads only one sentence, will they understand your change?)
3. Address the following questions in your pull request:
    1. What is a short summary of your change?
    2. Why is this change helpful?
    3. Any specific details to consider?
    4. What is the desired outcome of your change?

6.4 Maintainer response time

Project maintainers make a best effort to respond in **10 days or less** to new issues. Current maintainers are volunteers working on the project, so we try to keep up as best we can. If more than 10 days passed and you have not received a reply, follow up in Telegram or IRC (`#rit-lug-teleirc` on irc.freenode.net). Someone may have missed your comment – we are not intentionally ignoring anyone.

Remember, using issue templates and answering the above questions in new pull requests reduces the response time from a maintainer to your issue or pull request.

CHAPTER

SEVEN

MESSAGE STATE DIAGRAM

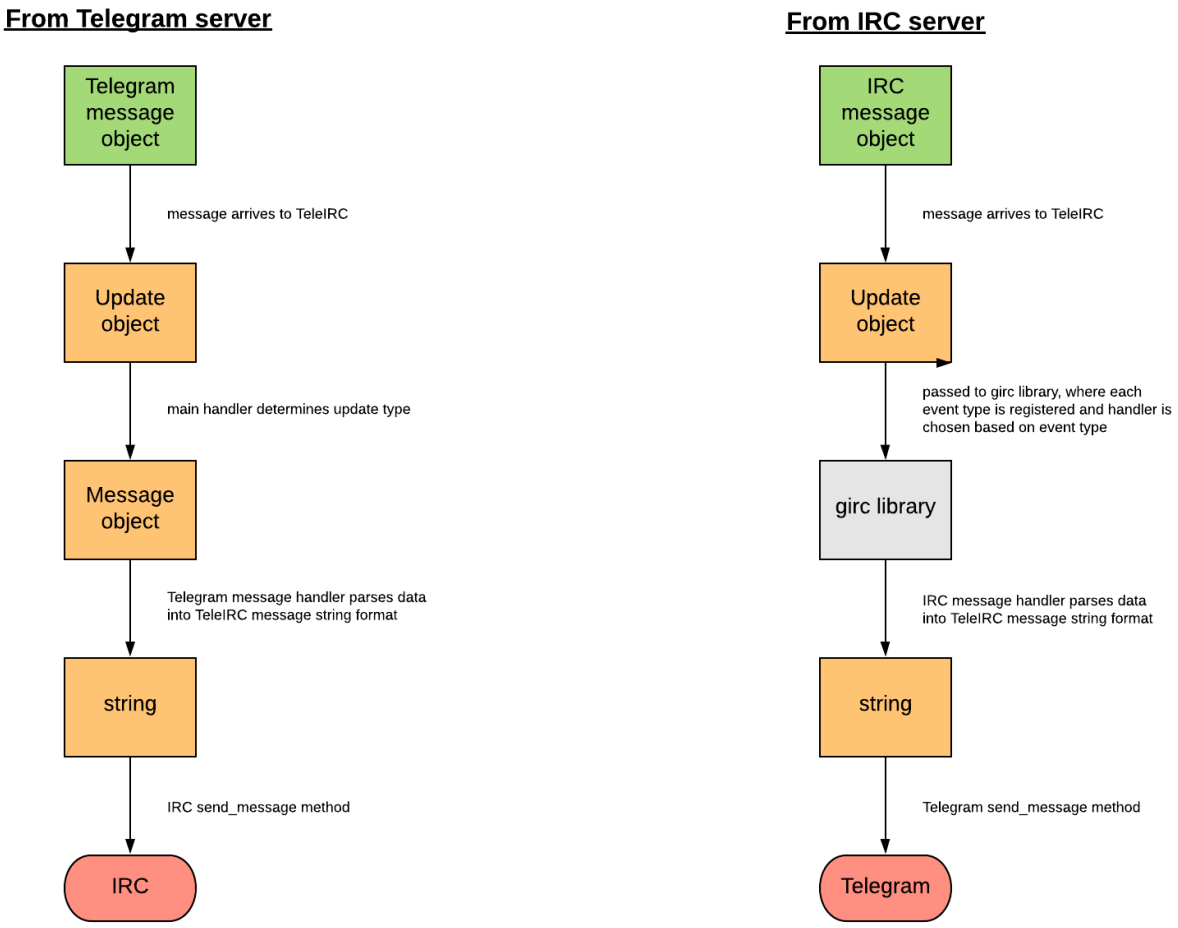
New in version v2.0.0.

Note: Only applicable to v2.0+ Golang port.

This document explains possible states of a Message as it travels through TeleIRC. There are two possible pathways, depending if a message is sourced from Telegram or IRC:

TeleIRC Message State Diagram

Prepared by Justin W. Flory
Last updated: 2020 February 15



CHAPTER

EIGHT

GO TESTING: BEST PRACTICES

This page explains our best practices for writing unit tests in Go. This was originally written by contributor Nicholas Jones (@Zedjones).

8.1 What is unit testing?

Unit testing is a type of testing where, as the name implies, we want to test an individual *unit* of code. In any given programming language, a *unit* is the smallest piece of grouped code, usually a function.

In a unit test, we only want to test that this piece of code does its job. We don't care if any other piece of code does its job, and optimally a unit test should pass even if every other unit of code that it calls fails.

To give an example from our current code base:

```
func (c Client) StartBot(errChan chan<- error, sendMessage func(string)) {
        fmt.Println("Starting up IRC bot...")
        c.sendToTg = sendMessage
        c.addHandlers()
        if err := c.Connect(); err != nil {
                errChan <- err
        } else {
                errChan <- nil
        }
}
```

In this example, we care that our function does the following things:

- Passes "Starting up IRC bot..." to `fmt.Println`.
- Sets `c.sendToTg` to the `sendMessage` passed in.
- Calls `c.addHandlers`.
- Calls `c.Connect`
  - If an error is returned from this function, passed the error over the `errChan` channel passed in.
  - Otherwise, passes `nil` over the `errChan` channel.

Here are some things that we don't care about:

- What `fmt.Println` does with our string, that's its responsibility.
- What `c.addHandlers` does
- What `c.Connect` does, we only want its return values

In fact, we have an issue: if `c.Connect` *actually* gets called, we can't guarantee that it will ever return what we want it to so we can actually test all of our cases. Not to mention, we want our unit tests to pass even if there is no network connection available.

The solution to this: **mocking**. Due to static typing and the lack of inheritance in Go, the only real way to do proper mocking is to use interfaces.

8.2 Interfaces

An interface in Go is similar to most other languages. It defines a contract that a concrete implementation must follow.

For example:

```
type Shape interface {
    Area() float64
}
```

This interface defines what a shape is: anything that has an area. A concrete implementation would be:

```
type Rectangle struct {
    Width  float64
    Height float64
}

func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}
```

Another implementation might be:

```
type Circle struct {
    Radius float64
}

func (c Circle) Area() float64 {
    return math.Pi * math.Pow(c.Radius, 2)
}
```

Now, we can create a function that accepts our interface:

```
func PrintArea (s Shape) {
    fmt.Println(s.Area())
}

func main() {
        PrintArea(Rectangle{Width: 10, Height: 15}) // prints 150
        PrintArea(Circle{Radius: 15}) // prints 706.8583
}
```

8.2.1 Why do we *need* interfaces for testing?

Because we cannot modify the implementation of functions on structs directly, we must change our functions to accept *interfaces* instead.

So, looking at one of our handlers:

```
func connectHandler(c Client) func(*girc.Client, girc.Event) {
        return func(gc *girc.Client, e girc.Event) {
                c.Cmd.Join(c.Settings.Channel)
        }
}
```

In this case, there is a small problem: we called `c.Cmd.Join`. So how does that work? How can we mock an inner struct inside of a struct? Well, we define an interface as such:

```
type IRCClient interface {
    Join(string)
}
```

Then, we modify our Client struct and add the following method:

```
func (c Client) Join(channel string) {
        c.Cmd.Join(channel)
}
```

Now, our `Client` struct implements our interface and we can change the original function to accept our interface instead and change from `c.Cmd.Join` to `c.Join`:

```
func connectHandler(c IRCClient) func(*girc.Client, girc.Event) {
        return func(gc *girc.Client, e girc.Event) {
                c.Join(c.Settings.Channel)
        }
}
```

While this does add complexity to our code, it also decouples the process of joining a channel from our exact library implementation. Now, if we want to change our IRC library down the line, we just need to write a `Join` method that fulfills this contract.

As an added bonus, we can now write our own implementation of the `IRCClient` to use as a mock. Something that fulfills the contract of the interface, but where the `Join` method doesn't actually do any work, just verifies that the string passed in is correct. **Or**, even better, let's use a library to generate this for us.

8.3 Generating mocks from interfaces with libraries

The most popular mock generation library is `mockgen`.

So, to generate a mock from the previous interface, we would run the following commands:

```
$ GO111MODULE=on go get github.com/golang/mock/mockgen@latest
$ mockgen -source=./internal/handlers/irc.go
# OR
$ mockgen github.com/ritlug/teleirc/internal/handlers/irc IRCClient
```

Now, we could write the following code in a unit test:

```
func TestConnectHandler(t *testing.T) {
  ctrl := gomock.NewController(t)

  // Assert that Join() is invoked.
  defer ctrl.Finish()

  m := NewMockIRCClient(ctrl)
  m.Settings = IRCSettings {Channel: "some channel"}

  // Asserts that the first and only call to Join() is passed "some channel".
  // Anything else will fail.
  m.
    EXPECT().
    Join(gomock.Eq("some channel"))

  connectHandler(m)(nil, nil) // Disclaimer: I didn't actually test this
}
```

CHAPTER

NINE

LIVE DEMO

A public Telegram group and IRC channel on Freenode are available for testing. Our developer community is also found in these channels.

- Telegram: @teleirc
- IRC: #rit-lug-teleirc (chat.freenode.net)

CHAPTER

TEN

LEGAL

All TeleIRC documentation is licensed under the Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0).

CHAPTER

ELEVEN

INDICES AND TABLES

- github.com/RITlug/teleirc
- genindex
- modindex
- search