Telegram Bots Book

downloads 366k Stars 854

Telegram.Bot is the most popular .NET Client for Telegram Bot API.

The Bot API is an HTTP-based interface created for developers keen on building bots for Telegram. Check *Bots: An introduction for developers* to understand what a Telegram bot is and what it can do.

We, Telegram Bots team, mainly focus on developing multiple NuGet packages for creating chatbots.

Packages	Team	News Channel	Group Chat
nuget		.NET	.NET
Packages we release on NuGet	The team contributing to this work	Subscribe to `@tgbots_dotnet` channel to get our latest news	Join our chat to talk about bots and ask questions

What Is This Book For

All Bot API methods are already documented by Telegram[1] but this book covers all you need to know to create a chatbot in .NET. There are also many concrete examples written in C#. The guides here can even be useful to bot developers using other languages/platforms as it shows best practices in developing Telegram chatbots with examples.

Correctness & Testing

This project is fully tested using Unit tests and Systems Integration tests before each release. In fact, our test cases are self-documenting and serve as examples for Bot API methods. Once you learn the basics of Telegram chatbots, you will be able to easily understand the code in examples and use it in your own bot program.

Get Started

Quickstart

Bot Father

Before you start developing a bot, you need to talk to `@BotFather` on Telegram. Register a bot with him and get an access token.

Access token is a key used to identify and authorize your bot in API requests so keep it with yourself as a secret. It looks like this:

```
1234567:4TT8bAc8GHUspu3ERYn-KGcvsvGB9u_n4ddy
```

Hello World

Now you have a bot, it's time to bring it to life! Create a new console project for your bot. It could be a .NET Core project or a .NET project targeting versions 4.5+.

> This guide uses .NET Core examples but full .NET framework projects work as well.

```
dotnet new console
```

Add a reference to `Telegram.Bot` package.

```
dotnet add package Telegram.Bot
```

Open `Program.cs` file and use the following content. This code fetches Bot information based on its access token by calling `getMe` method on the Bot API.

> Replace `YOUR_ACCESS_TOKEN_HERE` with the access token from Bot Father.

```
using System;
using Telegram.Bot;

namespace Awesome {
  class Program {
    static void Main() {
      var botClient = new TelegramBotClient("YOUR_ACCESS_TOKEN_HERE");
      var me = botClient.GetMeAsync().Result;
      Console.WriteLine(
        $"Hello, World! I am user {me.Id} and my name is {me.FirstName}."
      );
    }
  }
}
```

Running the program gives you the following output:

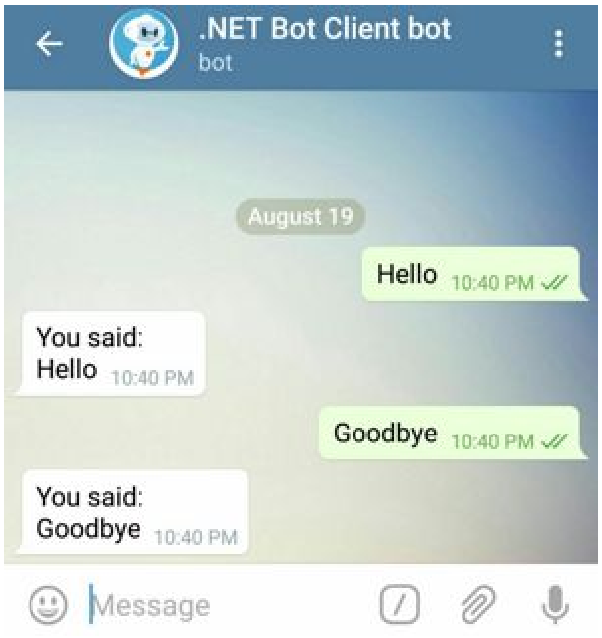
```
dotnet run
```

```
Hello, World! I am user 1234567 and my name is Awesome Bot.
```

Great! This bot is self-aware. To make the bot interact with a user, head to the next page.

Example - First Chat Bot

On the previous page, we got an access token and used the `getMe` method to check our setup. Now, it is time to make an *interactive* bot that gets users' messages and replies to them like in this screenshot:


.NET Bot Client bot
bot
August 19
Hello 10:40 PM
You said:
Hello 10:40 PM
Goodbye 10:40 PM
You said:
Goodbye 10:40 PM

Copy the following code to `Program.cs`.

Replace `YOUR_ACCESS_TOKEN_HERE` with the access token from Bot Father.

```
using System;
using System.Threading;
using Telegram.Bot;
using Telegram.Bot.Args;

namespace Awesome {
  class Program {
    static ITelegramBotClient botClient;

    static void Main() {
      botClient = new TelegramBotClient("YOUR_ACCESS_TOKEN_HERE");

      var me = botClient.GetMeAsync().Result;
      Console.WriteLine(
        $"Hello, World! I am user {me.Id} and my name is {me.FirstName}."
      );

      botClient.OnMessage += Bot_OnMessage;
      botClient.StartReceiving();
      Thread.Sleep(int.MaxValue);
    }

    static async void Bot_OnMessage(object sender, MessageEventArgs e) {
      if (e.Message.Text != null)
      {
        Console.WriteLine($"Received a text message in chat
{e.Message.Chat.Id}.");

        await botClient.SendTextMessageAsync(
          chatId: e.Message.Chat,
          text:   "You said:\n" + e.Message.Text
        );
      }
    }
  }
}
```

Run the program.

```
dotnet run
```

It runs for a long time waiting for text messages unless forcefully stopped. Open a private chat with your bot in Telegram and send a text message to it. Bot should reply in no time.

In the code above, we subscribe to `OnMessage` event on bot client to take action on messages that users send to bot.

By invoking `StartReceiving()`, bot client starts fetching updates using `getUpdates` method for the bot from Telegram Servers. This is an asynchronous operation so `Thread.Sleep()` is used right after that to keep the app running for a while in this demo.

When a user sends a message, `Bot_OnMessage()` gets invoked with the message object passed via event arguments. We are expecting a text message so we check for `Message.Text` value. Finally, we send a text message back to the same chat we got the message from.

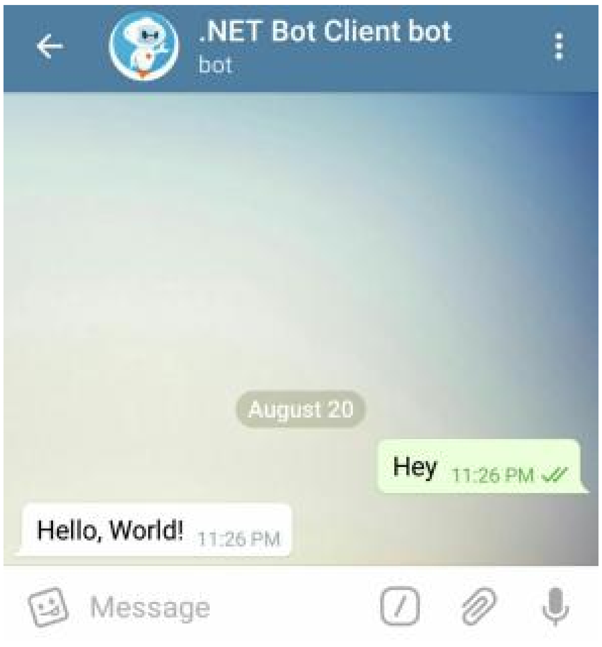
If you take a look at the console, the program outputs the `chatId` value. **Copy the chat id number** to make testing easier for yourself in the next pages.

```
Received a text message in chat 123456789.
```

Sending Messages

There are many different types of message that a bot can send. Fortunately, methods for sending such messages are similar. Take a look at these examples:

Sending text message:

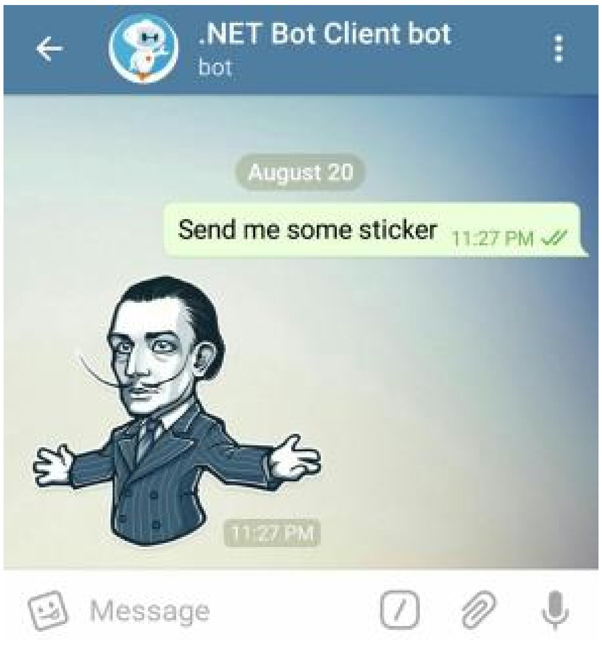


Hey 11:26 PM

Hello, World! 11:26 PM

```
await botClient.SendTextMessageAsync(
  chatId: e.Message.Chat,
  text:   "Hello, World!"
);
```

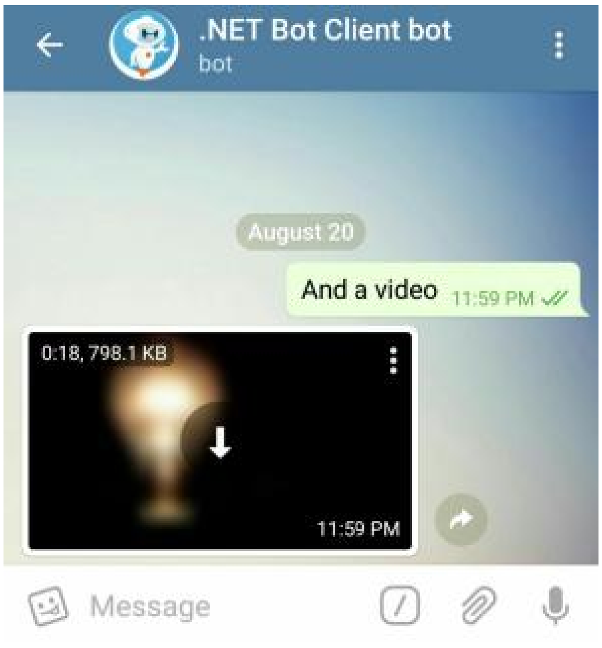
Sending sticker message:



Send me some sticker 11:27 PM

```
await botClient.SendStickerAsync(
  chatId:  e.Message.Chat,
  sticker: "https://github.com/TelegramBots/book/raw/master/src/docs/sticker-
dali.webp"
);
```

Sending video message:



And a video 11:59 PM

0:18, 798.1 KB

11:59 PM

```
await botClient.SendVideoAsync(
  chatId:  e.Message.Chat,
  video: "https://github.com/TelegramBots/book/raw/master/src/docs/video-
bulb.mp4"
);
```

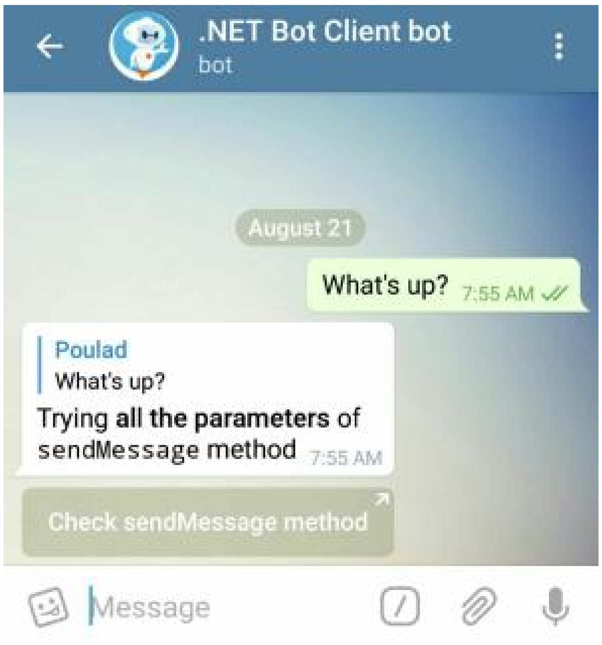
Text Messages and More

Bot API method send message Examples Text Messages

Text is a powerful interface for your bot and `sendMessage` probably is the most used method of the Telegram Bot API. Text messages are easy to send and fast to display on devices with slower networking. *Don't send boring plain text to users all the time*. Telegram allows you to format the text using Markdown or HTML.

Send Text Message

The code snippet below sends a message with multiple parameters that looks like this:



What's up? 7:55 AM

Poulad
What's up?
Trying **all the parameters** of `sendMessage` method 7:55 AM

You can use this code snippet in the event handler from Example Bot page and use `e.Message.Chat` or put the `chatId` value if you know it.

```
// using Telegram.Bot.Types;
// using Telegram.Bot.Types.Enums;
// using Telegram.Bot.Types.ReplyMarkups;

Message message = await botClient.SendTextMessageAsync(
  chatId: e.Message.Chat, // or a chat id: 123456789
  text: "Trying *all the parameters* of `sendMessage` method",
  parseMode: ParseMode.Markdown,
  disableNotification: true,
  replyToMessageId: e.Message.MessageId,
  replyMarkup: new InlineKeyboardMarkup(InlineKeyboardButton.WithUrl(
    "Check sendMessage method",
    "https://core.telegram.org/bots/api#sendmessage"
  ))
);
```

The method `SendTextMessageAsync` of .NET Bot Client maps to `sendMessage` on Telegram's Bot API. This method sends a text message and returns the message object sent.

`text` is written in MarkDown format and `parseMode` indicates that. You can also write in HTML or plain text.

By passing `disableNotification` we tell Telegram client on user's device not to show/sound a notification.

It's a good idea to make it clear to a user the reason why the bot is sending this message and that's why we pass the user's message id for `replyToMessageId`.

You have the option of specifying a `replyMarkup` when sending messages. Reply markups are explained in details later in this book. Here we used an *Inline Keyboard Markup* with a button that attaches to the message itself. Clicking that opens `sendMessage` method documentation in the browser.

The Message Sent

Almost all of the methods for sending messages return you the message you just sent. Let's have a look at this message object. Add this statement after the previous code.

```
Console.WriteLine(
  $"{message.From.FirstName} sent message {message.MessageId} " +
  $"to chat {message.Chat.Id} at {message.Date}. " +
  $"It is a reply to message {message.ReplyToMessage.MessageId} " +
  $"and has {message.Entities.Length} message entities."
);
```

Output should look similar to this:

```
Awesome bot sent message 123 to chat 123456789 at 8/21/18 11:25:09 AM. It is a
reply to message 122 and has 2 message entities.
```

There are a few things to note.

Date and time is in UTC format and not your local timezone. Convert it to local time by calling `message.Date.ToLocalTime()` method.

Message Entity refers to those formatted parts of the text: *all the parameters* in bold and *sendMessage* in mono-width font. Property `message.Entities` holds the formatting information and `message.EntityValues` gives you the actual value. For example, in the message we just sent:

```
message.Entities.First().Type == MessageEntityType.Bold
message.EntityValues.First()  == "all the parameters"
```

Currently, message object doesn't contain information about its reply markup.

Try putting a breakpoint in the code to examine all the properties on a message objects you get.

Photo and Sticker Messages

You can provide the source file for almost all multimedia messages (e.g. photo, video) in 3 ways:

- Uploading a file with the HTTP request
- HTTP URL for Telegram to get a file from the internet
- `file_id` of an existing file on Telegram servers (*recommended*)

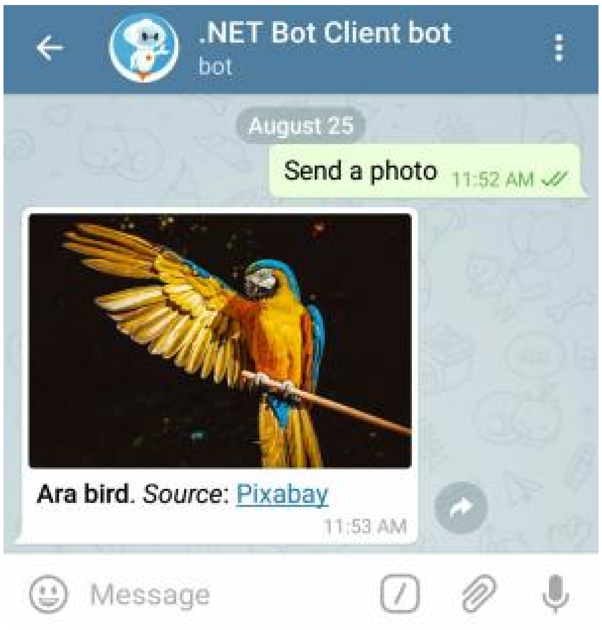
Examples in this section show all three. You will learn more about them later on when we discuss file upload and download.

Photo

Bot API method send photo Examples Photo Messages

Sending a photo is simple. Here is an example:

```
Message message = await botClient.SendPhotoAsync(
  chatId: e.Message.Chat,
  photo: "https://github.com/TelegramBots/book/raw/master/src/docs/photo-
ara.jpg",
  caption: "<b>Ara bird</b>. <i>Source</i>: <a
href=\"https://pixabay.com\">Pixabay</a>",
  parseMode: ParseMode.Html
);
```



Caption

Multimedia messages can *optionally* have a caption attached to them. Here we sent a caption in HTML format. A user can click on *Pixabay* in the caption to open its URL in the browser.

Similar to message entities discussed before, caption entities on `Message` object are the result of parsing formatted(Markdown or HTML) caption text. Try inspecting these properties in debug mode:

- `message.Caption`: caption in plain text without formatting
- `message.CaptionEntities`: info about special entities in the caption
- `message.CaptionEntityValues`: text values of mentioned entities

Photo Message

The `message` returned from this method represents a *photo message* because `message.Photo` has a value. Its value is a `PhotoSize` array with each element representing the same photo in different dimensions. If your bot needs to send this photo again at some point, it is recommended to store this array so you can reuse the `file_id` value.

Here is how `message.Photo` array looks like in JSON:

```
[
  {
    "file_id": "AgADBAADDqgxG-QDDVCm5JVvld7MN0z6kBkABCQawlb-dBXqBZUEAAEC",
    "file_size": 1254,
    "width": 90,
    "height": 60
  },
  {
    "file_id": "AgADBAADDqgxG-QDDVCm5JVvld7MN0z6kBkABAKByRnc22RmBpUEAAEC",
    "file_size": 16419,
    "width": 320,
    "height": 213
  },
  {
    "file_id": "AgADBAADDqgxG-QDDVCm5JVvld7MN0z6kBkABHezqGiNOz9yB5UEAAEC",
    "file_size": 57865,
    "width": 640,
    "height": 426
  }
]
```

Sticker

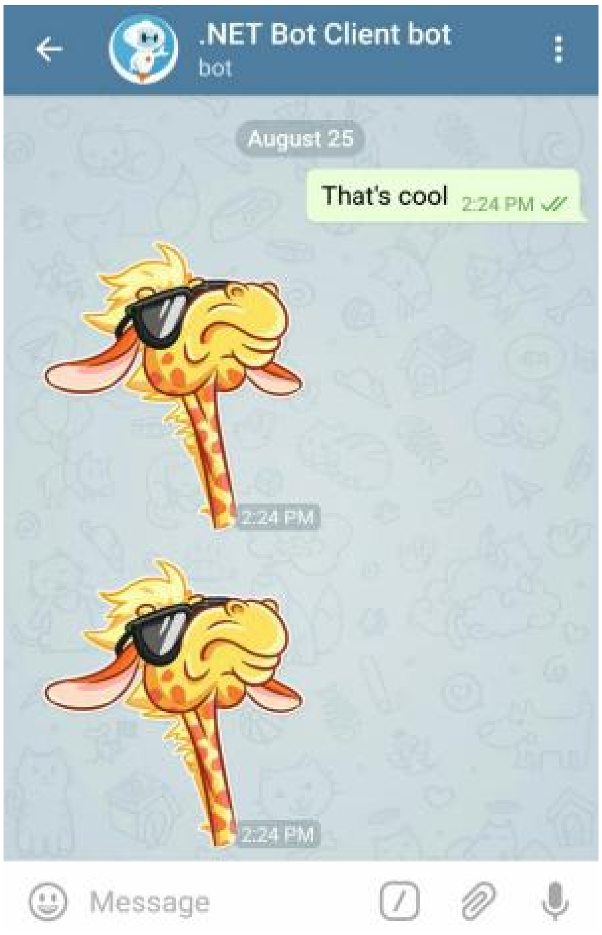
Bot API method send sticker Examples Sticker Messages

Telegram stickers are fun and our bot is about to send its very first sticker. Sticker files should be in WebP format.

This code sends the same sticker twice. First by passing HTTP URL to a WebP sticker file and second by reusing `file_id` of the same sticker on Telegram servers.

```
Message msg1 = await botClient.SendStickerAsync(
  chatId: e.Message.Chat,
  sticker: "https://github.com/TelegramBots/book/raw/master/src/docs/sticker-
fred.webp"
);

Message msg2 = await botClient.SendStickerAsync(
  chatId: e.Message.Chat,
  sticker: msg1.Sticker.FileId
);
```



Try inspecting the `msg1.Sticker` property. It is of type `Sticker` and its schema looks similar to a photo.

> There is more to stickers and we will talk about them in greater details later.

Audio and Voice Messages

Examples Audio Messages

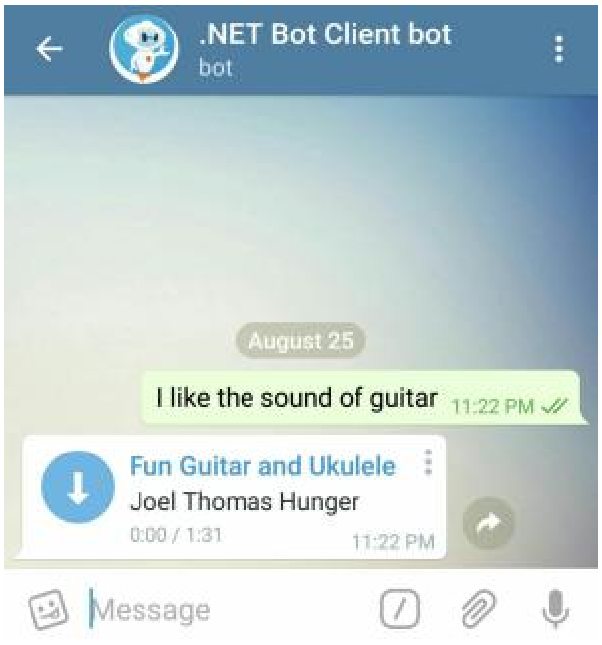
These two types of messages are pretty similar. An audio is in MP3 format and gets displayed in music player. A voice file has OGG format and is not shown in music player.

Audio

Bot API method send audio

This is the code to send an MP3 soundtrack. You might be wondering why some parameters are commented out. That's because this MP3 file has metadata on it and Telegram does a good job at reading them.

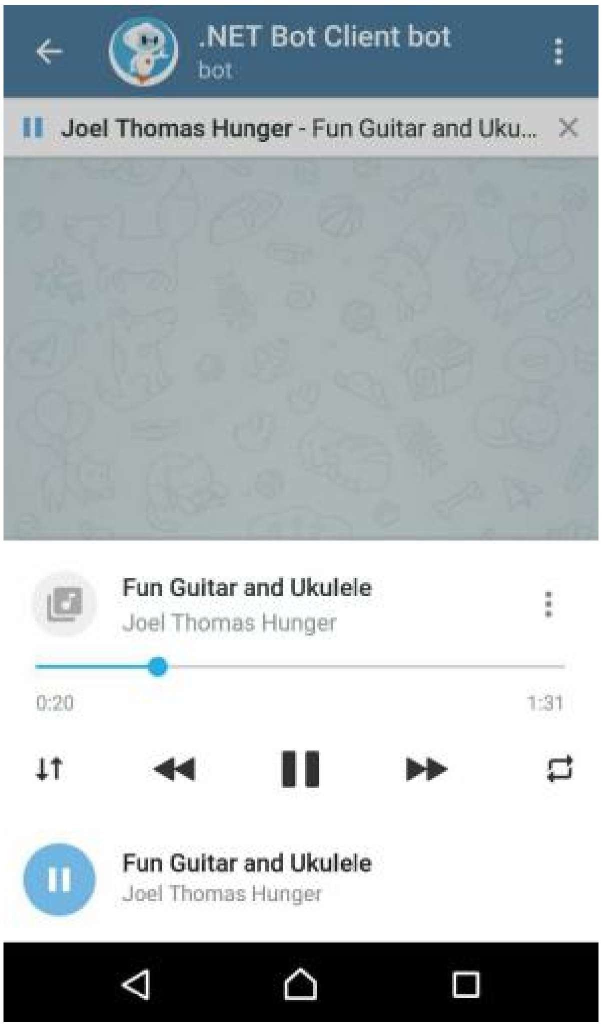
```
Message msg = await botClient.SendAudioAsync(
  e.Message.Chat,
  "https://github.com/TelegramBots/book/raw/master/src/docs/audio-guitar.mp3"
  /* ,
  performer: "Joel Thomas Hunger",
  title: "Fun Guitar and Ukulele",
  duration: 91 // in seconds
  */
);
```



I like the sound of guitar 11:22 PM

Fun Guitar and Ukulele
Joel Thomas Hunger
0:00 / 1:31 11:22 PM

And a user can see the audio in Music Player:



Method returns an audio message. Let's take a look at the value of `msg.Audio` property in JSON format:

```
{
  "duration": 91,
  "mime_type": "audio/mpeg",
  "title": "Fun Guitar and Ukulele",
  "performer": "Joel Thomas Hunger",
  "file_id": "CQADBAADKQADA3oUUKalqDOOcqesAg",
  "file_size": 1102154
}
```

Voice

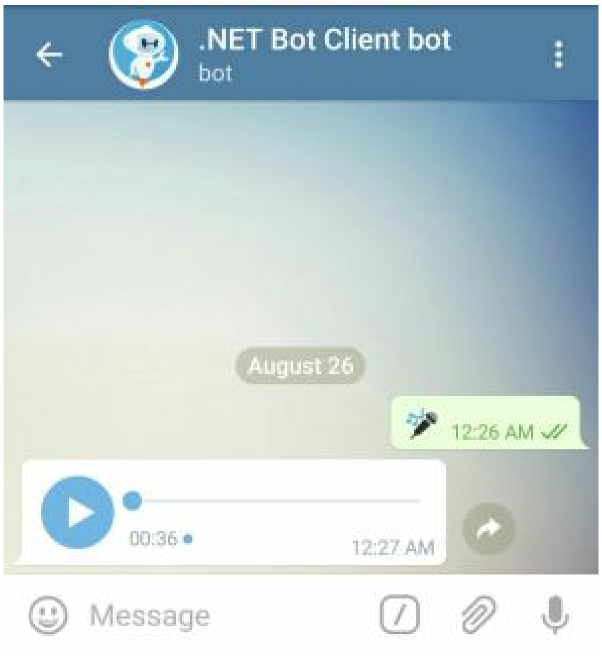
Bot API method send voice

A voice message is just an OGG audio file. Let's send it differently this time by uploading the file from disk alongside with an HTTP request.

To run this example, download the NFL Commentary voice file to your disk.

A value is passed for `duration` because Telegram can't figure that out from a file's metadata.

```
Message msg;
using (var stream = System.IO.File.OpenRead("/path/to/voice-nfl_commentary.ogg"))
{
  msg = await botClient.SendVoiceAsync(
    chatId: e.Message.Chat,
    voice: stream,
    duration: 36
  );
}
```



A voice message is returned from the method. Inspect the `msg.Voice` property to learn more.

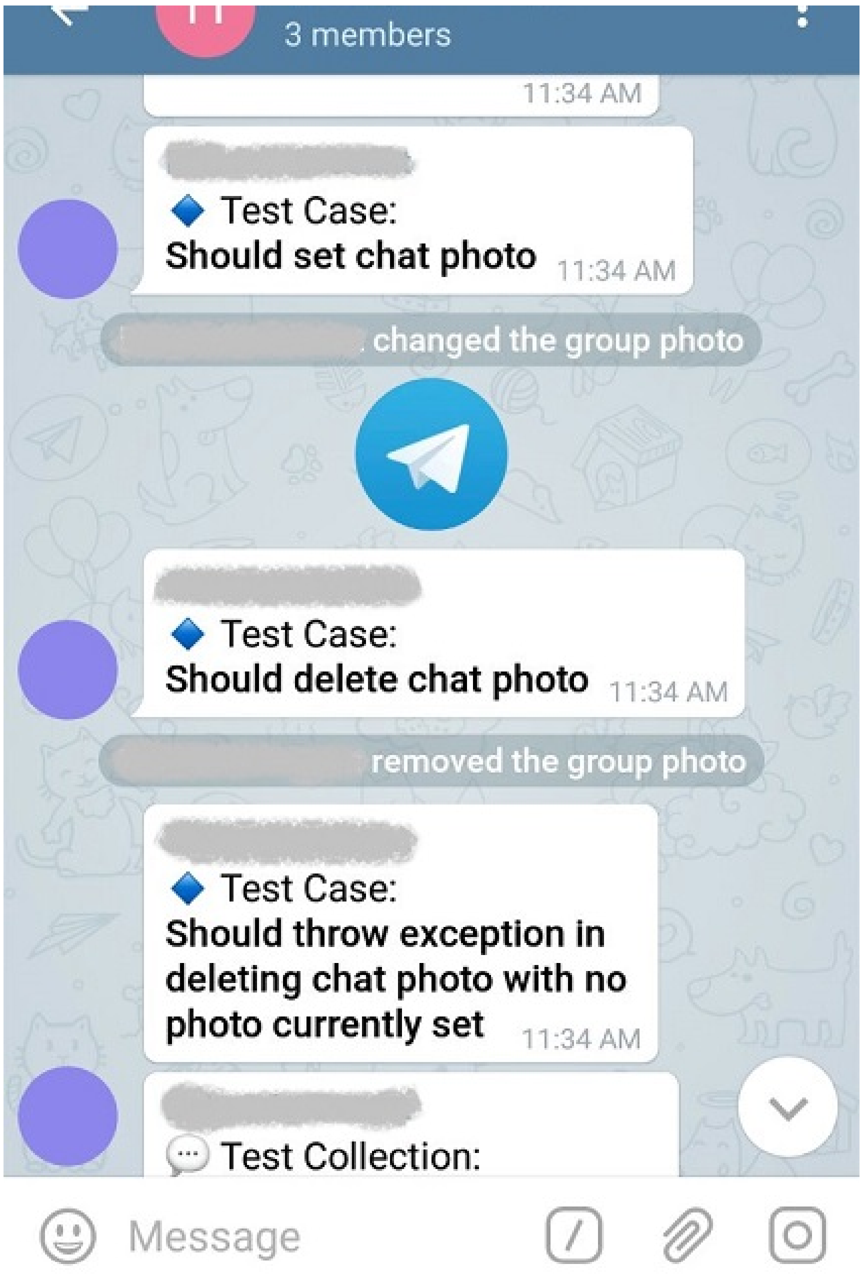
Systems Integration Tests

Integration tests are meant to test the project with real data from Telegram. They are semi-automated tests and tester(s) need to interact with bot for some cases during the test execution. Tests could be used as a playground for exploring Bot API methods.

Sample Test Diagnostics Output

All the test output goes into the supergroup/private chats specified in configurations or interactively during test execution. You can see some samples of test output below.

Admin bots can change chat photo.



11:34 AM

[redacted]
🔷 Test Case:
Should set chat photo 11:34 AM

[redacted] changed the group photo

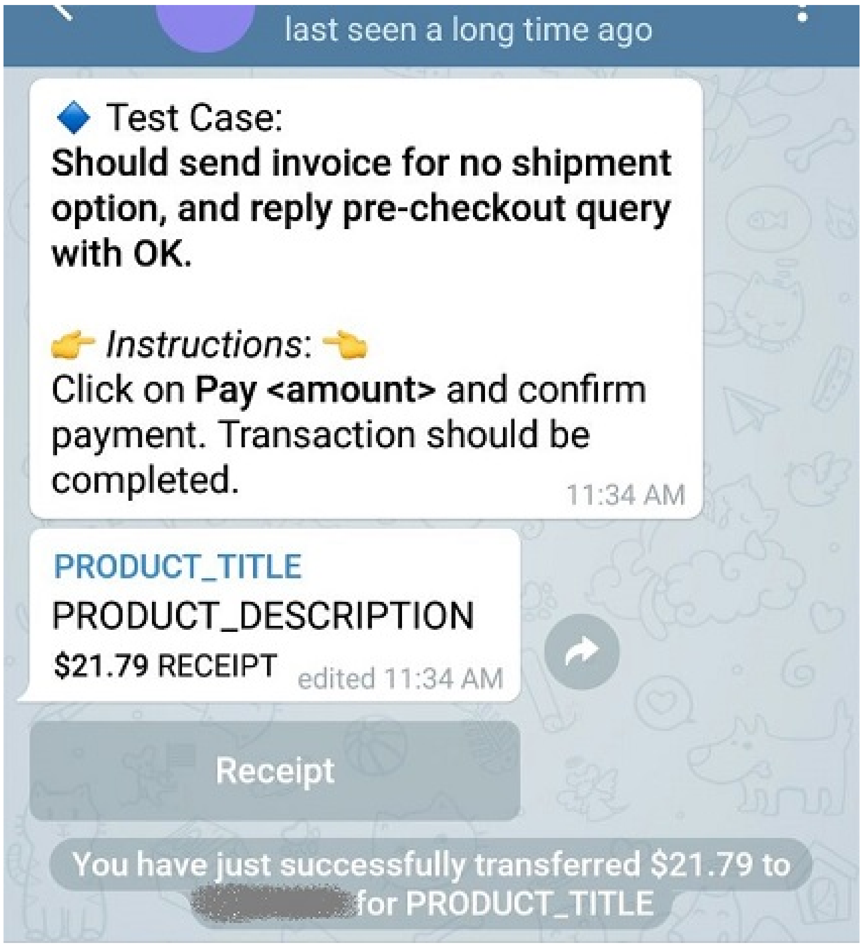
[redacted]
🔷 Test Case:
Should delete chat photo 11:34 AM

[redacted] removed the group photo

[redacted]
🔷 Test Case:
Should throw exception in deleting chat photo with no photo currently set 11:34 AM

[redacted]
💬 Test Collection:

Invoices could be paid in private chats.


last seen a long time ago

🔷 Test Case:
Should send invoice for no shipment option, and reply pre-checkout query with OK.

👉 *Instructions:* 👈
Click on **Pay <amount>** and confirm payment. Transaction should be completed.
11:34 AM

PRODUCT_TITLE
PRODUCT_DESCRIPTION
$21.79 RECEIPT edited 11:34 AM

Receipt

You have just successfully transferred $21.79 to [redacted] for PRODUCT_TITLE

How Tests Works

These integration tests are written just like regular unit tests with xUnit framework so they seem to be unit tests. When you run test(s), bot makes a request to Bot API and you should see results(message or service notification) in the chat with bot.

When you build the solution, you will see them in Test Explorer window. Tests could be run through .NET Core's CLI as well and that's how this project's CI is set up.

A bot, of course, is needed to test Bot API. This document refers to its user name as *MyTestBot*.

Tester refers to user name of person testing the bot. Multiple testers could interact with bot during test execution. If super group chat has other members that are not listed as testers, bot ignores their messages during test execution. Testers must have user names assigned and their user names should be set in test configurations before hand.

All the tests happen in two chats. A Super Group chat and a Private chat with one of the testers.

Test cases that need tester's interaction to continue, have a limit of usually 2 minues to wait for receving an expected update from API.

Tests could be run individually, in collections, or all at once. All the test collection and test cases whithin them are ordered and tests will not run in parallel.

Test Environment Setup

Create a Super Group and add bot to it. Promote bot to admin and make sure it has all the permissions. This group needs to have another regular(non-admin) member to be used in tests for chat administration methods(such as Kick, Restrict, Unban). A super group with 2 testers in it, one admin and the other non-admin member, is enough.

Bot should have some features enabled, usually through BotFather, in order to pass tests. These features are listed below:

- Inline Queries
- Payment Provider

For making testing process more convenient, set the following commands for MyTestBot as well. The purpose for these commands is explained in the sections below.

```
test - Start test execution
  me - Select me for testing admin methods
```

Test Configurations

You can see a list of configuration keys in `appsettings.json` file. Make a copy of this file and store your configurations there. In addition to `appsettings.json` and `appsettings.Development.json`, environment variables prefixed by `TelegramBot_` are also read into program.

```
cp appsettings.json appsettings.Development.json
```

Required Settings

Only 2 values must be provided before test execution.

API Token

This is required for executing any test case.

```
{
    "ApiToken": "MyTestBot-API-TOKEN"
    /* ... */
}
```

Allowed Users

A comma separated list indicating user name(s) of tester(s). Any update coming from users other than the ones listed here are discarded during test execution.

```
{
    /* ... */
    "AllowedUserNames": "tester1, Tester2, TESTER3"
    /* ... */
}
```

Optional Settings

The following settings are not required for two reasons. Either bot can ask for them during test execution or it is not a required setting for all test cases.

Bot will ask testers in supergroup/private chat for the necessary information. It would be faster to set all the optional settings as well because it makes testing process faster and less manual.

> For obtaining values of necessary settings, you can set breakpoints in some test methods and extract values such as chat id or user id.

Supergroup Chat Id

Bot send messages to this chat in almost all test cases except cases like sending payments that must be to a private chat.

If not set, before starting any test method, bot waits for a tester to send it a `/test` command in a super group chat (that bot is also a member of).

```
{
    /* ... */
    "SuperGroupChatId": -1234567890
    /* ... */
}
```

Payment Settings

[Required] Payment Provider Token

This token is **required** for any test case regarding payments and must be provided before starting tests.

Consult Telegram API documentations and talk to BotFather to get a test token from a payment provider.

```
{
    /* ... */
    "PaymentProviderToken": "MY-PAYMENT-PROVIDER-TOKEN"
    /* ... */
}
```

TesterPrivateChatId

Invoices could only be sent to private chats. If not set, bot will wait for a tester to send it `/test` command in a private chat.

```
{
    /* ... */
    "TesterPrivateChatId": 1234567890
    /* ... */
}
```

Chat Administration

For this type of tests, bot should be a priviledged admin of that super group. Methods such as kick or unban will be performed on a regular (non-admin) tester in that chat.

If the following 3 settings are not set, bot will ask a tester to send it `/me` command in a private chat with bot.

- Regular Member's User Id
- Regular Member's User Name
- Regular Member's Private Chat Id

```
{
    /* ... */
    "RegularMemberUserId": 1234567890,
    "RegularMemberUserName": "tester3",
    "RegularMemberPrivateChatId": 1234567890
    /* ... */
}
```

First, read the documentation on sending files.

Uploading the actual file

```
using (FileStream fs = System.IO.File.OpenRead("Local file.pdf"))
{
    InputOnlineFile inputOnlineFile = new InputOnlineFile(fs, "Name for the
user.pdf");
    await Bot.SendDocumentAsync(message.Chat, inputOnlineFile);
}
```

Uploading by file id

This is the method that you will want to use the most, as it is the most efficient. After sending a file to the user, or receiving a file, you can send the file again using its ID.

This saves you from uploading the entire file.

```
InputOnlineFile inputOnlineFile = new InputOnlineFile("file id");
await Bot.SendDocumentAsync(message.Chat, inputOnlineFile);
```

Uploading by URL

```
InputOnlineFile inputOnlineFile = new
InputOnlineFile("telegram.org/img/t_logo.png");
await Bot.SendDocumentAsync(message.Chat, inputOnlineFile);
```

Telegram Passport

nuget v1.0.0 downloads 105 Stars 3

Telegram Passport is a unified authorization method for services that require personal identification. As a bot developer, you can use it to receive confidential user data in an end-to-end encrypted fashion. There are several Know Your Customer(KYC) solutions that have already added support for Telegram Passport.

This guide is targeted at bot developers and assumes the audience is already familiar with:

- Telegram Passport
- Telegram Passport Example
- Telegram Passport Manual

Telegram Passport - Quickstart

Examples Passport Quickstart

This guide teaches the basics of working with Telegram Passport. See the complete version of the code at Quickstart project. Code snippets on this page are in the context of that project.

Package

nuget v1.0.0

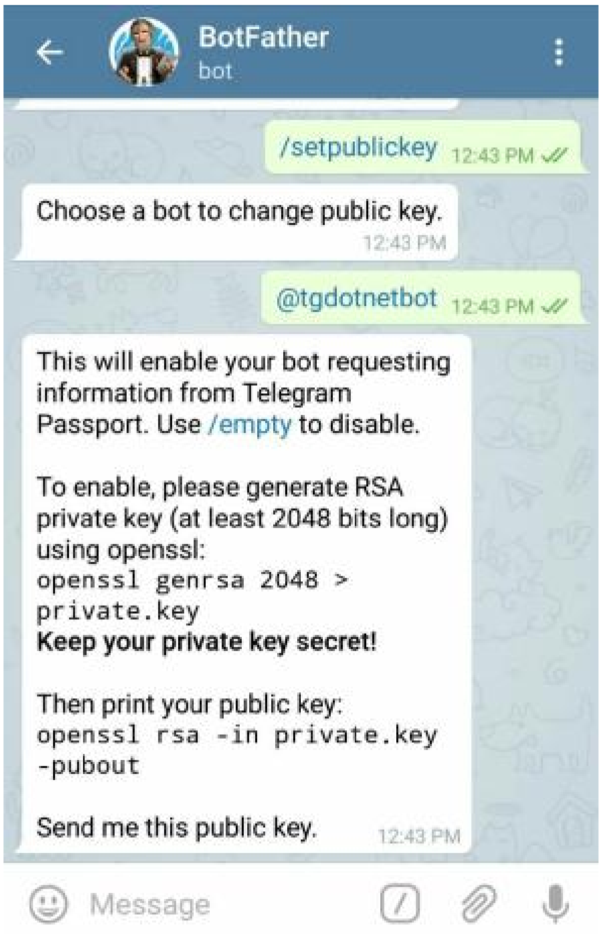
You need to add `Telegram.Bot.Extensions.Passport` extension package to your project in addition to the core package (`Telegram.Bot`).

☆ Star the Telegram.Bot.Extensions.Passport project on GitHub 👍

```
dotnet add package Telegram.Bot.Extensions.Passport
```

Encryption Keys

You don't really need to generate any RSA key. Use our sample keys for this demo. Send the public key to @BotFather using `/setpublickey` command:


BotFather
bot

/setpublickey 12:43 PM

Choose a bot to change public key. 12:43 PM

@tgdotnetbot 12:43 PM

This will enable your bot requesting information from Telegram Passport. Use /empty to disable.

To enable, please generate RSA private key (at least 2048 bits long) using openssl:
```
openssl genrsa 2048 >
private.key
```
Keep your private key secret!

Then print your public key:
```
openssl rsa -in private.key
-pubout
```

Send me this public key. 12:43 PM

Copy this public key and send it to BotFather.

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA0VElWoQA2SK1csG2/sY/
wlssO1bjXRx+t+JlIgS6jLPCefyCAcZBv7ElcSPJQIPEXNwN2XdnTc2wEIjZ8bTg
BlBqXppj471bJeX8Mi2uAxAqOUDuvGuqth+mq7DMqol3MNH5P9FO6li7nZxI1FX3
9u2r/4H4PXRiWx13gsVQRL6Clq2jcXFHc9CvNaCQEJX95jgQFAybal216EwlnnVV
giT/TNsfFjW41XJZsHUny9k+dAfyPzqAk54cgrvjgAHJayDWjapq90Fm/+e/DVQ6
BHGkV0POQMkkBrvvhAIQu222j+03frm9b2yZrhX/qS01lyjW4VaQytGV0wlewV6B
FwIDAQAB
-----END PUBLIC KEY-----
```

Now Telegram client app can encrypt the data for your bot using this key.

Request Information

Bot waits for a text message from user. Once it receives a text message, it generates an authorization request link and sends that to the user.

Authorization Request

Passport API type Request Parameters

A passport authorization request means that the bot should ask the user to open a `tg://resolve` URI in the browser with specific parameters in its query string. You can alternatively have a button in an HTML page on your website for that.

Type AuthorizationRequestParameters helps you in creating such an URI.

```
AuthorizationRequestParameters authReq = new AuthorizationRequestParameters(
  botId: 123456, // bot user ID
  publicKey: "...", // public key in PEM format. same as the key above.
  nonce: "unique nonce for this request",
  scope: new PassportScope(new[] { // a PassportScope object
    new PassportScopeElementOne("address"),
    new PassportScopeElementOne("phone_number")
  })
);
```

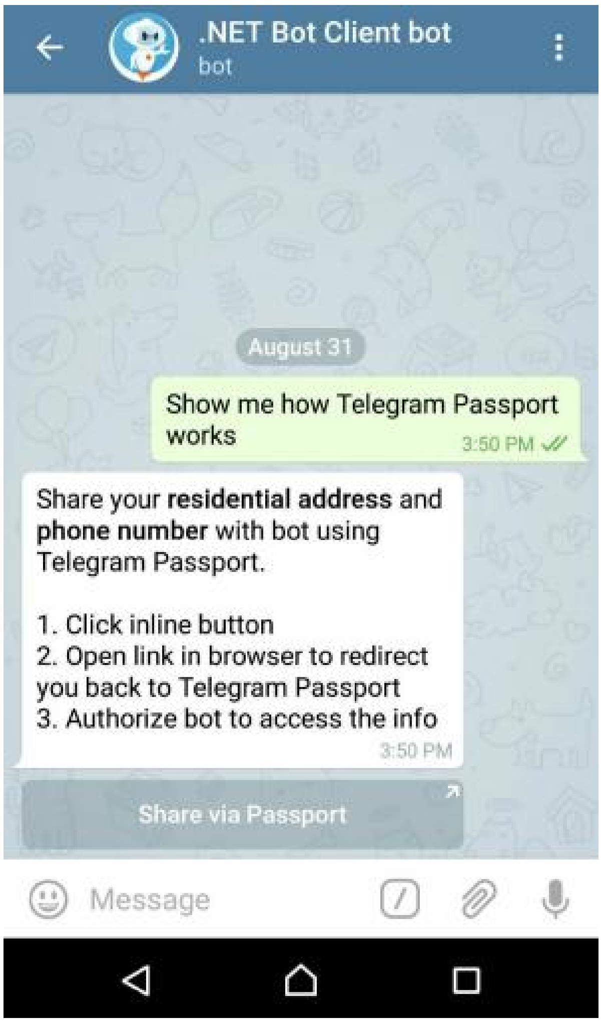
In SendAuthorizationRequestAsync method, we ask for `address` and `phone_number` scopes. Then, we generate the query string and ask user to open the link.

You might be wondering what is the magic in here?

https://telegrambots.github.io/Telegram.Bot.Extensions.Passport/redirect.html

This web page redirects user to `tg://resolve` URI, appending whatever query string was passed to it.

> If a user is using an Android device, the URI will start with `tg:` instead of the default `tg://`.

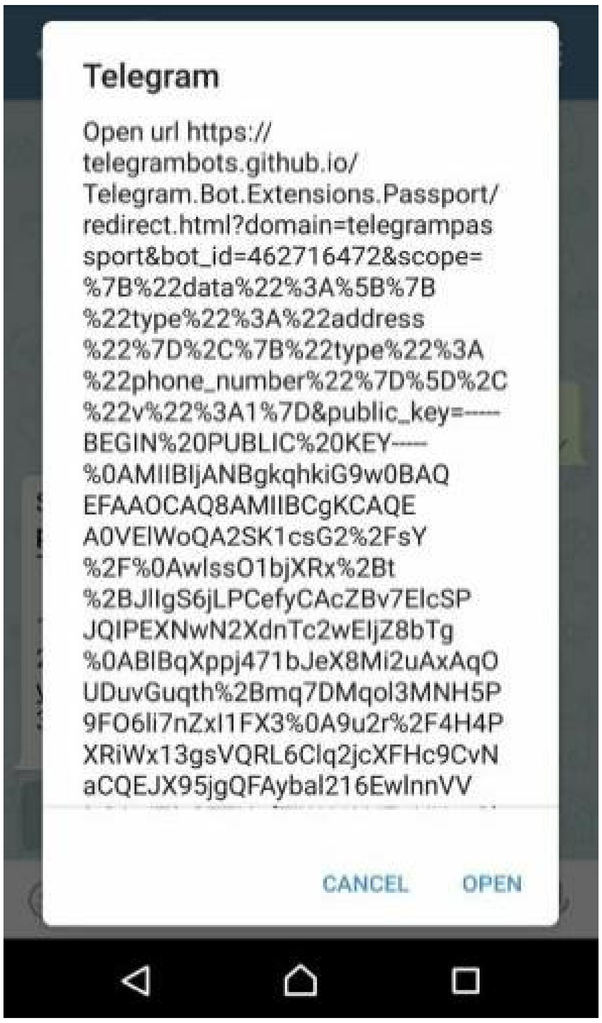


Show me how Telegram Passport works
3:50 PM

Share your **residential address** and **phone number** with bot using Telegram Passport.

1. Click inline button
2. Open link in browser to redirect you back to Telegram Passport
3. Authorize bot to access the info

3:50 PM



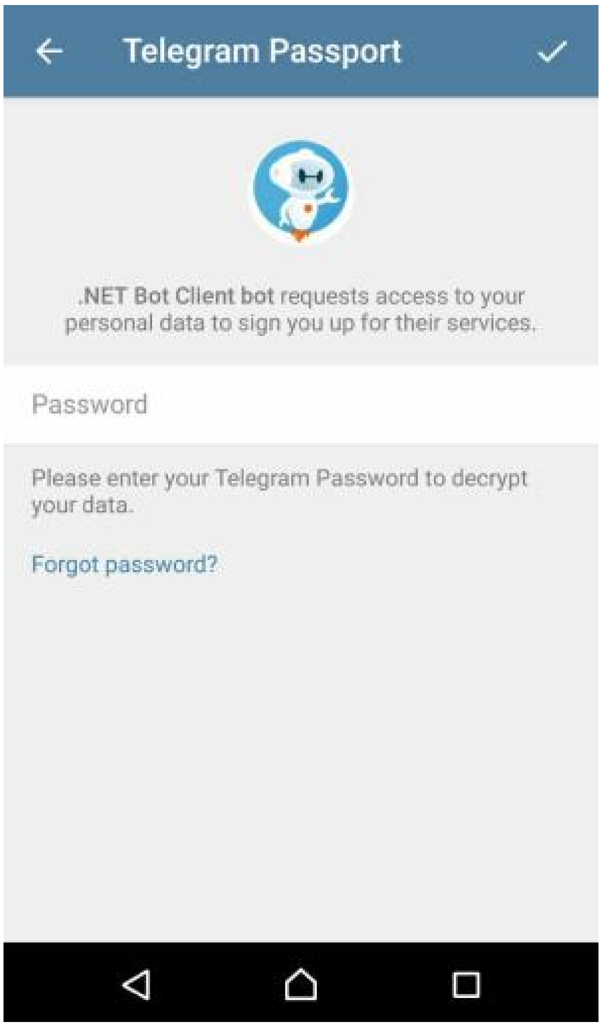
Telegram

```
Open url https://
telegrambots.github.io/
Telegram.Bot.Extensions.Passport/
redirect.html?domain=telegrampas
sport&bot_id=462716472&scope=
%7B%22data%22%3A%5B%7B
%22type%22%3A%22address
%22%7D%2C%7B%22type%22%3A
%22phone_number%22%7D%5D%2C
%22v%22%3A1%7D&public_key=-----
BEGIN%20PUBLIC%20KEY-----
%0AMIIBIjANBgkqhkiG9w0BAQ
EFAAOCAQ8AMIIBCgKCAQE
A0VElWoQA2SK1csG2%2FsY
%2F%0AwlssO1bjXRx%2Bt
%2BJIlgS6jLPCefyCAcZBv7ElcSP
JQIPEXNwN2XdnTc2wEljZ8bTg
%0ABlBqXppj471bJeX8Mi2uAxAqO
UDuvGuqth%2Bmq7DMqol3MNH5P
9FO6li7nZxl1FX3%0A9u2r%2F4H4P
XRiWx13gsVQRL6Clq2jcXFHc9CvN
aCQEJX95jgQFAybal216EwlnnVV
```

Passport Data

You, the user, should now be redirected to the Telegram Passport screen in your Telegram client app. Enter your password and log in.

> Note that the app will ask you to register if this is the first time you are using Telegram Passport.


Telegram Passport

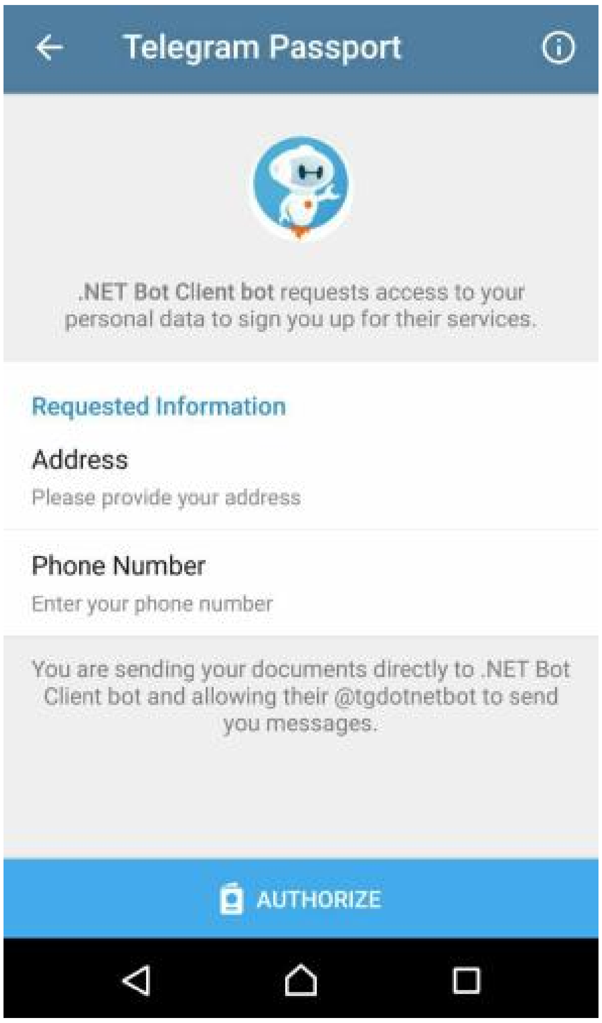
.NET Bot Client bot requests access to your personal data to sign you up for their services.

Password

Please enter your Telegram Password to decrypt your data.

Forgot password?

Fill in the *address* and *phone number* data. Click on the *Authorize* button at the end.



Telegram Passport

.NET Bot Client bot requests access to your personal data to sign you up for their services.

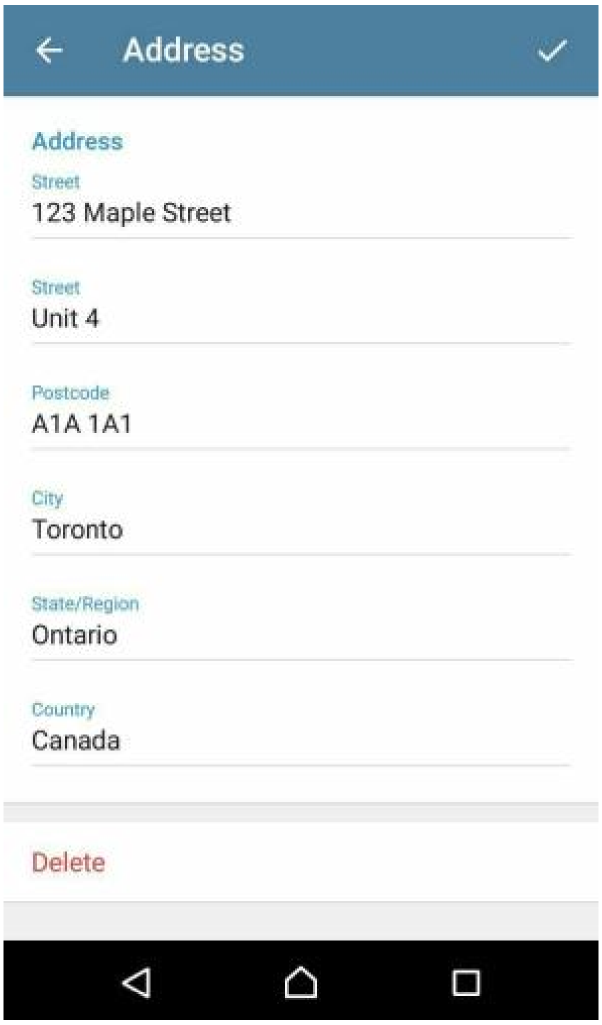
Requested Information

Address
Please provide your address

Phone Number
Enter your phone number

You are sending your documents directly to .NET Bot Client bot and allowing their @tgdotnetbot to send you messages.

AUTHORIZE

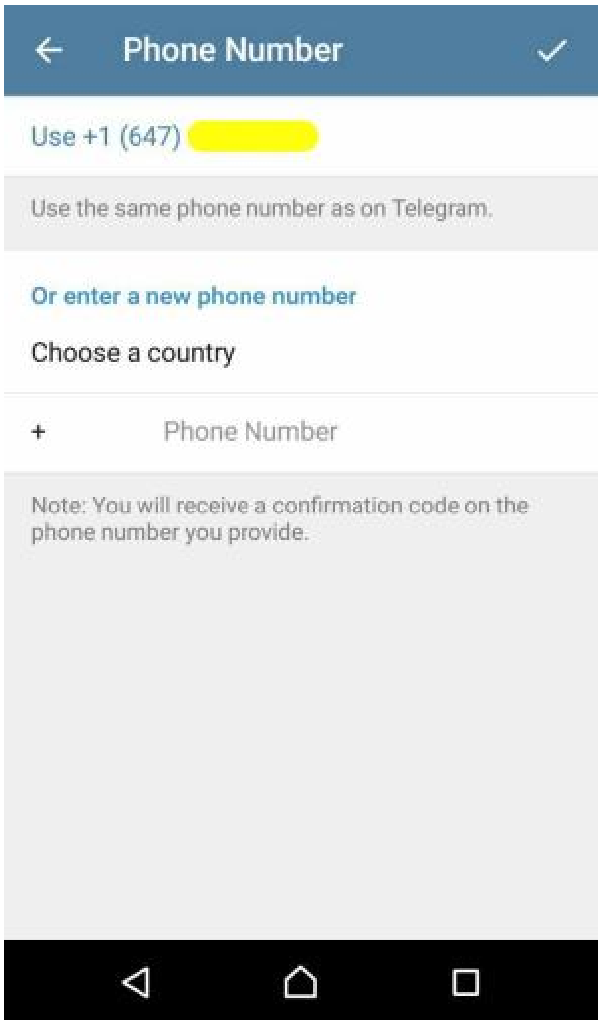


Address

Address

Street	123 Maple Street
Street	Unit 4
Postcode	A1A 1A1
City	Toronto
State/Region	Ontario
Country	Canada

Delete



Phone Number

Use +1 (647) [redacted]

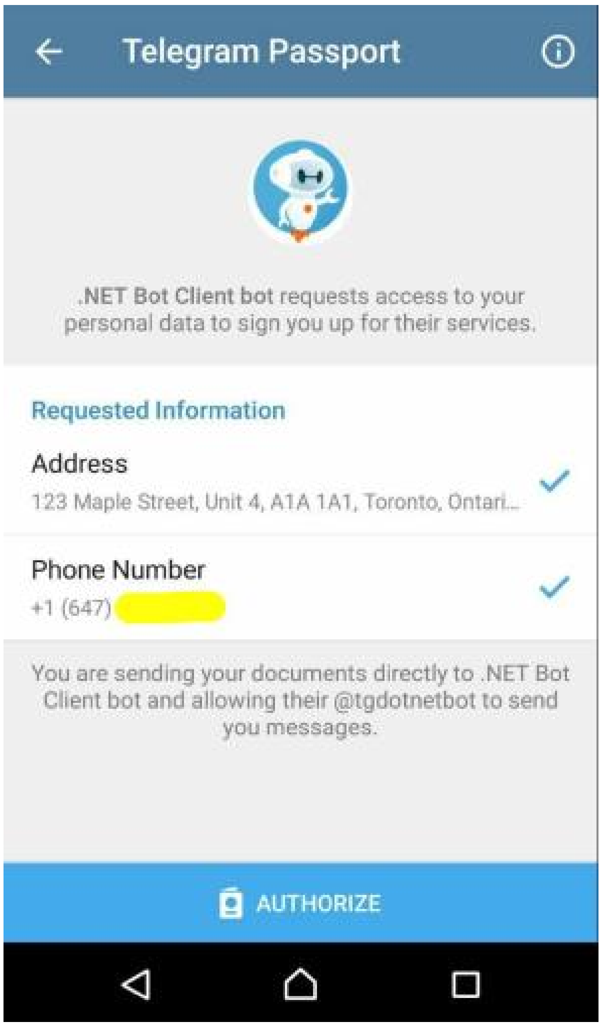
Use the same phone number as on Telegram.

Or enter a new phone number

Choose a country

+ Phone Number

Note: You will receive a confirmation code on the phone number you provide.



Telegram Passport

.NET Bot Client bot requests access to your personal data to sign you up for their services.

Requested Information

Address ✓
123 Maple Street, Unit 4, A1A 1A1, Toronto, Ontari...

Phone Number ✓
+1 (647) [redacted]

You are sending your documents directly to .NET Bot Client bot and allowing their @tgdotnetbot to send you messages.

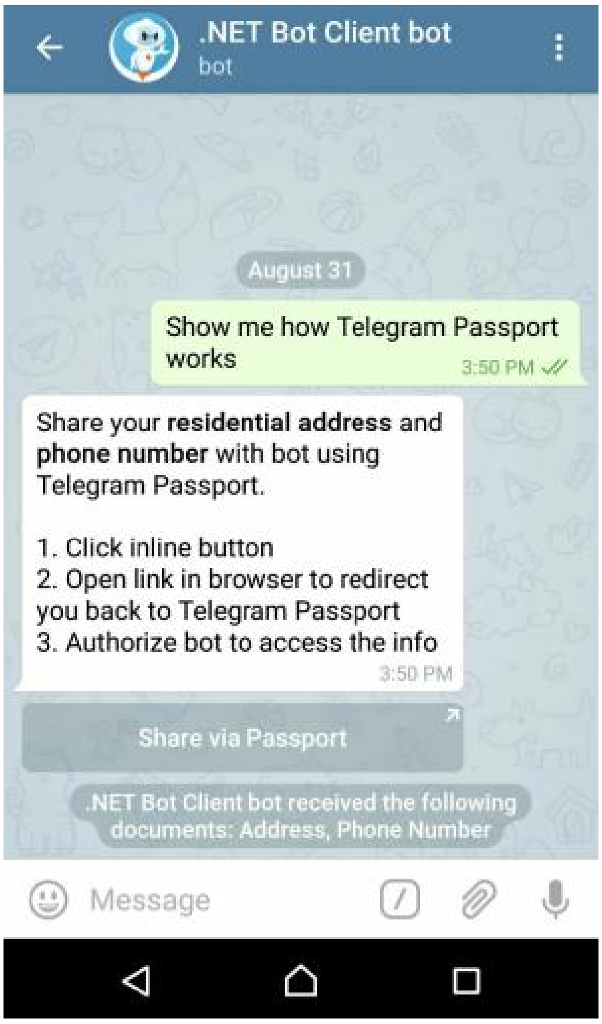
AUTHORIZE

At this point, your Telegram client app encrypts the actual Telegram Passport data (e.g. address) using the AES algorithm, and then encrypts the info required for decryption using your bot's public RSA key. Finally, it sends the result of both encryptions to Telegram servers.

Data Decryption

Bot API type Passport Data

Your bot now receives a new message update with the encrypted Passport data. The user is also notified in the chat:



.NET Bot Client bot
bot

August 31

Show me how Telegram Passport works
3:50 PM

Share your **residential address** and **phone number** with bot using Telegram Passport.

1. Click inline button
2. Open link in browser to redirect you back to Telegram Passport
3. Authorize bot to access the info
3:50 PM

.NET Bot Client bot received the following documents: Address, Phone Number

Let's decrypt that gibberish to get the information. That's what DecryptPassportDataAsync method does.

Step 1: Credentials

Bot API type EncryptedCredentials Passport API type Credentials

You can't just access the encrypted data in the `message.passport_data.data` array. Required parameters for their decryption are in the `message.passport_data.credentials` object. But that credentials object is encrypted using bot's public key!

We first take the bot's *private key* this time and decrypt the credentials.

There are more details about importing a key in PEM format on the RSA Key page.

```
IDecrypter decrypter = new Decrypter();
Credentials credentials = decrypter.DecryptCredentials(
  message.PassportData.Credentials, // EncryptedCredentials object
  GetRsaPrivateKey() // private key as an RSA object
);
```

Step 2: Nonce

Passport API type Credentials Passport API type Request Parameters

There is a `nonce` property on the credentials (now decrypted) object. In order to prevent certain attacks, ensure its value is exactly the same as the nonce you set in the authorization request. Read more about nonce on Wikipedia.

Step 3: Residential Address

Bot API type EncryptedPassportElement Passport API type ResidentialAddress

Passport API type SecureData Passport API type SecureValue Passport API type DataCredentials

It's finally time to see the user's address. We are looking for an encrypted element with type of *address* in `message.passport_data.data` array. Also, decryption parameters for that are in `credentials.secure_data.address.data`. Here is how the decryption magic happens:

```
EncryptedPassportElement addressElement = message.PassportData.Data.Single(
  el => el.Type == PassportEnums.Scope.Address
);
ResidentialAddress address = decrypter.DecryptData<ResidentialAddress>(
  encryptedData: addressElement.Data,
  dataCredentials: credentials.SecureData.Address.Data
);
```

DecryptData method does 3 tasks here:

1. Decrypts the data into a JSON-serialized string
2. Verifies that the data hashes match
3. Converts from JSON to a .NET object

Step 4: Phone Number

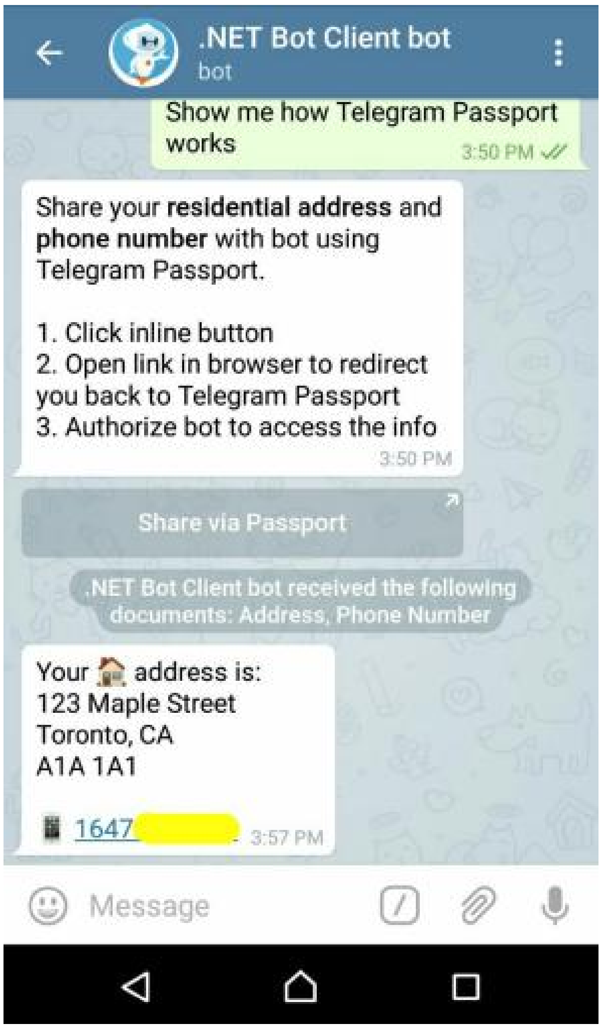
Bot API type EncryptedPassportElement

Values for phone number and email address are not end-to-end encrypted in Telegram Passport and Telegram stores these values after being verified.

There is no need for decryption at this point. Just find the element with the type of *phone_number* in the `message.passport_data.data` array.

Information Demo

At the end, bot sends some of the information received to the user for demo purposes.



Show me how Telegram Passport works 3:50 PM

Share your **residential address** and **phone number** with bot using Telegram Passport.

1. Click inline button
2. Open link in browser to redirect you back to Telegram Passport
3. Authorize bot to access the info

3:50 PM

.NET Bot Client bot received the following documents: Address, Phone Number

Your 🏠 address is:
123 Maple Street
Toronto, CA
A1A 1A1

📱 1647[redacted] 3:57 PM

Passport Files and Documents

Examples Driver License Scope

We use the driver's license scope here to show decryption of *ID document data* and *passport files* for front side scan, reverse side scan, selfie photo, and translation scan. That should cover most of the field types in Telegram Passport.

Sections below are referring to the test methods in Driver's License Scope Tests collection. Here are the steps:

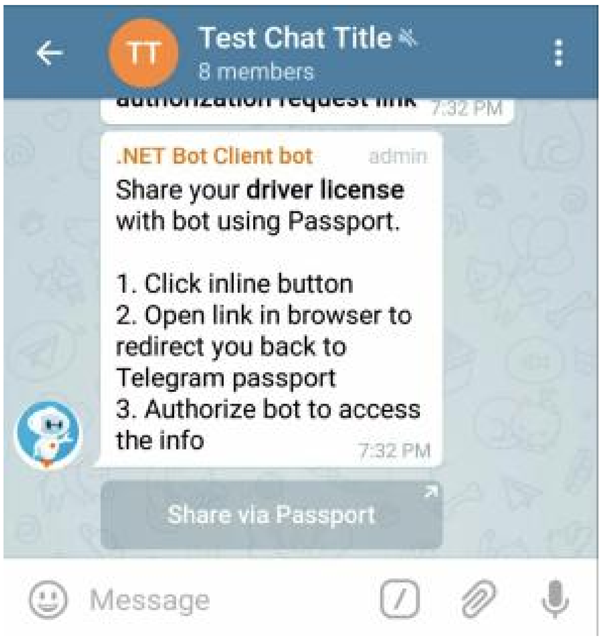
1. Authorization Request
2. Driver's License Info
3. Passport Message
4. Credentials
5. ID Document Data
6. Passport File
   - Front Side File
   - Reverse Side File
   - Selfie File
   - Translation File

Authorization Request

Passport API type PassportScope Passport API type PassportScopeElementOne

Test Method Generate Auth Link

We start by generating an authorization URI. Since a driver's license is considered as a proof of identity, we ask for optional data *selfie with document* and *translation document scan* as well.


Test Chat Title

8 members

authorization request link 7:32 PM

.NET Bot Client bot admin

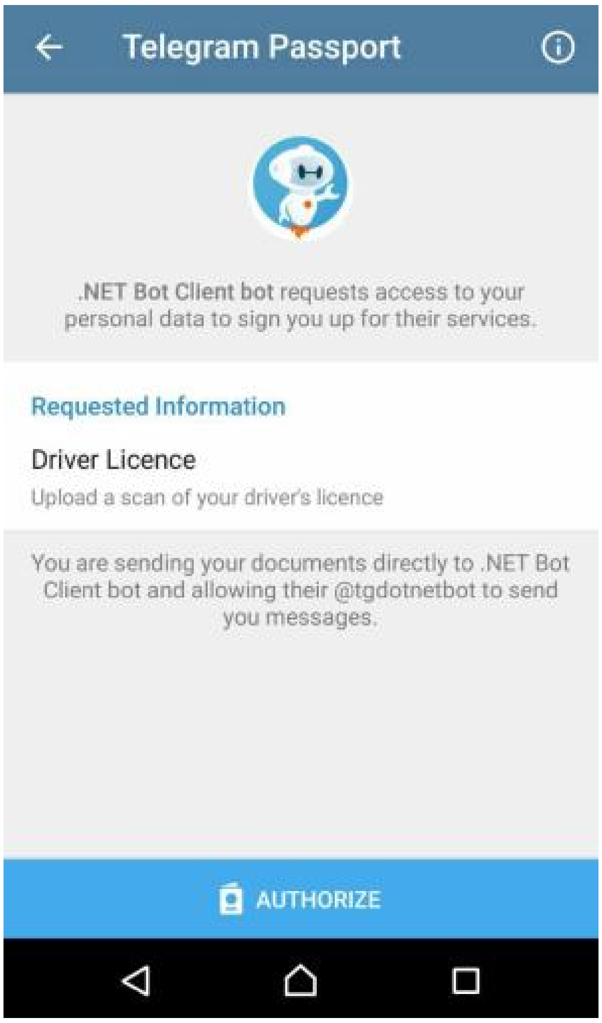
Share your **driver license** with bot using Passport.

1. Click inline button
2. Open link in browser to redirect you back to Telegram passport
3. Authorize bot to access the info

7:32 PM

Driver's License Info

As a user, provide information for the required fields: front side, reverse side, and document number. Also, test methods here expect a selfie photo and a file for translation scan.



Telegram Passport

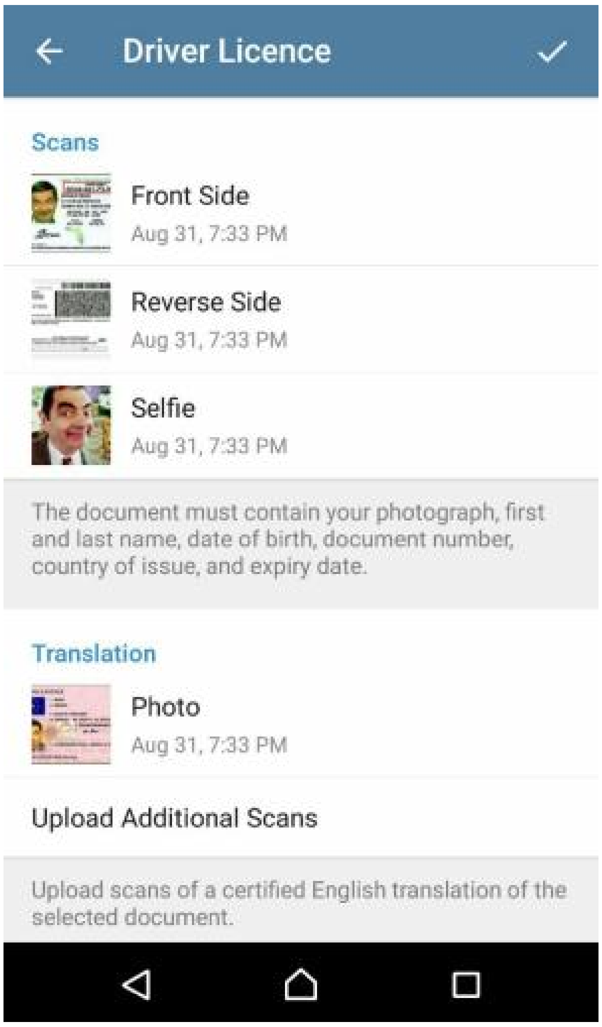
.NET Bot Client bot requests access to your personal data to sign you up for their services.

Requested Information

Driver Licence

Upload a scan of your driver's licence

You are sending your documents directly to .NET Bot Client bot and allowing their @tgdotnetbot to send you messages.



Driver Licence

Scans

Front Side
Aug 31, 7:33 PM

Reverse Side
Aug 31, 7:33 PM

Selfie
Aug 31, 7:33 PM

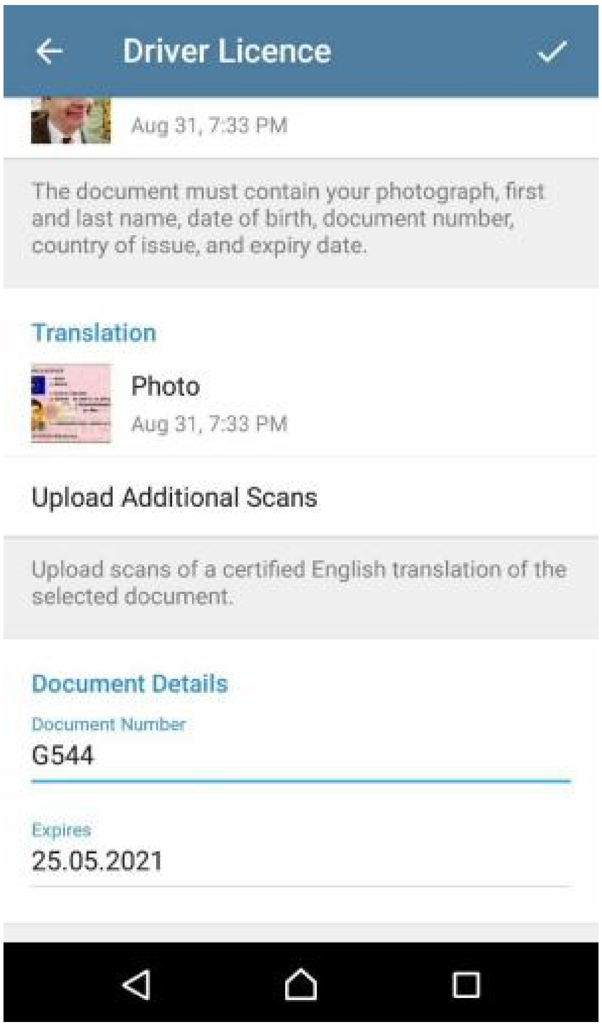
The document must contain your photograph, first and last name, date of birth, document number, country of issue, and expiry date.

Translation

Photo
Aug 31, 7:33 PM

Upload Additional Scans

Upload scans of a certified English translation of the selected document.



Driver Licence

Aug 31, 7:33 PM

The document must contain your photograph, first and last name, date of birth, document number, country of issue, and expiry date.

Translation

Photo
Aug 31, 7:33 PM

Upload Additional Scans

Upload scans of a certified English translation of the selected document.

Document Details

Document Number
G544

Expires
25.05.2021

Click the *Authorize* button at the end.

Passport Message

Bot API type Passport Data

Test Method Validate Passport Update

This test method checks for a Passport message with a driver's license element on it.

Credentials

Bot API type EncryptedCredentials Passport API type Credentials

Test Method Decrypt Credentials

We decrypt credentials using the RSA private key and verify that the same nonce is used.

```
RSA key = EncryptionKey.ReadAsRsa();
IDecrypter decrypter = new Decrypter();
Credentials credentials = decrypter.DecryptCredentials(
  passportData.Credentials,
  key
);
bool isSameNonce = credentials.Nonce == "Test nonce for driver's license";
```

ID Document Data

Passport API type IdDocumentData

Test Method Decreypt Document Data

In our test case, there is only 1 item in the `message.passport_data.data` array and that's the encrypted element for the driver's license scope. We can get information such as document number and expiry date for the license from that element:

```
IdDocumentData licenseDoc = decrypter.DecryptData<IdDocumentData>(
  encryptedData: element.Data,
  dataCredentials: credentials.SecureData.DriverLicense.Data
);
```

Passport File

Bot API type PassportFile Passport API type FileCredentials

Passport file is an encrypted JPEG file on Telegram servers. You need to download the passport file and decrypt it using its accompanying *file credentials* to see the actual JPEG file content. In this section we try to demonstrate different use cases that you might have for such files.

No matter the method used, the underlying decryption logic is the same. It really comes down to your decision on working with *streams* vs. *byte arrays*. IDecrypter gives you both options.

Front Side File

Test Method Decreypt Front Side File

A pretty handy extension method is used here to stream writing the front side file to disk. Method DownloadAndDecryptPassportFileAsync does a few things:

1. Makes an HTTP request to fetch the encrypted file's info using its *passport file_id*
2. Makes an HTTP request to download the encrypted file using its *file_path*
3. Decrypts the encrypted file
4. Writes the actual content to the destination stream

```
File encryptedFileInfo;
using (System.IO.Stream stream = System.IO.File.OpenWrite("/path/to/front-side.jpg"))
{
  encryptedFileInfo = await BotClient.DownloadAndDecryptPassportFileAsync(
    element.FrontSide, // PassportFile object for front side
    credentials.SecureData.DriverLicense.FrontSide, // front side FileCredentials
    stream // destination stream for writing the JPEG content to
  );
}
```

Warning: This method is convenient to use but gives you the least amount of control over the operations.

Reverse Side File

Test Method Decreypt Reverse Side File

Previous method call is divided into two operations here for reverse side of the license. Streams are used here as well.

```
File encryptedFileInfo;
using (System.IO.Stream
  encryptedContent = new System.IO.MemoryStream(element.ReverseSide.FileSize),
  decryptedFile = System.IO.File.OpenWrite("/path/to/reverse-side.jpg")
) {
  // fetch the encrypted file info and download it to memory
  encryptedFileInfo = await BotClient.GetInfoAndDownloadFileAsync(
    element.ReverseSide.FileId, // file_id of passport file for reverse side
    encryptedContent // stream to copy the encrypted file into
  );
  // ensure memory stream is at the beginning before reading from it
  encryptedContent.Position = 0;

  // decrypt the file and write it to disk
  await decrypter.DecryptFileAsync(
    encryptedContent,
    credentials.SecureData.DriverLicense.ReverseSide, // reverse side
FileCredentials
    decryptedFile // destination stream for writing the JPEG content to
  );
}
```

Selfie File

Test Method Decreypt Selfie File

We deal with selfie photo as a byte array. This is essentially the same operation as done above via streams. We also post the selfie photo to a chat.

```
// fetch the info of the passport file(selfie) residing on Telegram servers
File encryptedFileInfo = await BotClient.GetFileAsync(element.Selfie.FileId);

// download the encrypted file and get its bytes
byte[] encryptedContent;
using (System.IO.MemoryStream
  stream = new System.IO.MemoryStream(encryptedFileInfo.FileSize)
)
{
  await BotClient.DownloadFileAsync(encryptedFileInfo.FilePath, stream);
  encryptedContent = stream.ToArray();
}

// decrypt the content and get bytes of the actual selfie photo
byte[] selfieContent = decrypter.DecryptFile(
  encryptedContent,
  credentials.SecureData.DriverLicense.Selfie
);

// send the photo to a chat
using (System.IO.Stream stream = new System.IO.MemoryStream(selfieContent)) {
  await BotClient.SendPhotoAsync(
    123456,
    stream,
    "selfie with driver's license"
  );
}
```

Translation File

Test Method Decreypt Translation File

A bot can request certified English translations of a document. Translations are also encrypted passport files so their decryption is no different from others passport files.

Assuming that the user sends one translation scan only for the license, we receive the translation passport file object in `message.passport_data.data[0].translation[0]` and its accompanying file credentials object in `credentials.secure_data.driver_license.translation[0]`.

File gets written to disk as a byte array.

```csharp
PassportFile passportFile = element.Translation[0];
FileCredentials fileCreds = credentials.SecureData.DriverLicense.Translation[0];

// fetch passport file info
File encryptedFileInfo = await BotClient.GetFileAsync(passportFile.FileId);

// download encrypted file and get its bytes
byte[] encryptedContent;
using (System.IO.MemoryStream
  stream = new System.IO.MemoryStream(encryptedFileInfo.FileSize)
)
{
  await BotClient.DownloadFileAsync(encryptedFileInfo.FilePath, stream);
  encryptedContent = stream.ToArray();
}

// decrypt the content and get bytes of the actual selfie photo
byte[] content = decrypter.DecryptFile(
  encryptedContent,
  fileCreds
);

// write the file to disk
await System.IO.File.WriteAllBytesAsync("/path/to/translation.jpg", content);
```

Import RSA Key

In order to decrypt the credentials you need to provide the private RSA key to DecryptCredentials method. If you have the RSA key in PEM format, you cannot simply instantiate an RSA .NET object from it. Here we discuss two ways of importing your PEM private key.

From PEM Format

This is the easier option and recommended **for development time only**. We can generate an RSA .NET object from an RSA Key in PEM format using the BouncyCastle package.

```
dotnet add package BouncyCastle
```

Code snippet here shows the conversion from a PEM file to the needed RSA object.

```
// using System.IO;
// using System.Security.Cryptography;
// using Org.BouncyCastle.Crypto;
// using Org.BouncyCastle.Crypto.Parameters;
// using Org.BouncyCastle.OpenSsl;
// using Org.BouncyCastle.Security;

static RSA GetPrivateKey() {
  string privateKeyPem = File.ReadAllText("/path/to/private-key.pem");
  PemReader pemReader = new PemReader(new StringReader(privateKeyPem));
  AsymmetricCipherKeyPair keyPair = (AsymmetricCipherKeyPair)
pemReader.ReadObject();
  RSAParameters rsaParameters = DotNetUtilities
    .ToRSAParameters(keyPair.Private as RsaPrivateCrtKeyParameters);
  RSA rsa = RSA.Create(rsaParameters);
  return rsa;
}
```

Note: You don't necessarily need to have a dependency on the BouncyCastle package in your bot project. The section below offers a better alternative.

From RSA Parameters

We recommend to JSON-serialize RSAParameters of your key and create an RSA object using its values without any dependency on the BouncyCastle package in production deployment.

Copy EncryptionKeyUtility and EncryptionKeyParameters files from our Quickstart project. Those help with serialization.

You still need to **use BouncyCastle only once** to read the RSA key in PEM format and serialize its parameters:

```
// ONLY ONCE: read the RSA private key and serialize its parameters to JSON
static void WriteRsaParametersToJson() {
  string privateKeyPem = System.IO.File.ReadAllText("/path/to/private-key.pem");
  string json = EncryptionKeyUtility.SerializeRsaParameters(privateKeyPem);
  System.IO.File.WriteAllText("/path/to/private-key-params.json", json);
}

// Now, read the JSON file and create an RSA instance
static RSA GetRsaKey() {
  string json = System.IO.File.ReadAllText("/path/to/private-key-params.json");
  return EncryptionKeyUtility.GetRsaKeyFromJson(json);
}
```

Content of `private-key-params.json` will look similar to this:

```
{
  "E": "AQAB",
  "M": "0VElW...Fw==",
  "P": "56Mdiw...i7FSwDaM=",
  "Q": "51UN2sd...J44NTf0=",
  "D": "nrXEeOl2Ky...JIQ==",
  "DP": "KZYZWbsy.../lk60=",
  "DQ": "Y25KgzPj...AdBd0=",
  "IQ": "0153...N6Y="
}
```

It's worth mentioning that EncryptionKeyParameters is just a copy of RSAParameters struct. There are inconsistencies in serialization of RSAParameters type on different .NET platforms and that's why we use our own EncryptionKeyParameters type for serialization.

For instance, compare `RSAParameters` implementations on .NET Framework and .NET Core.

Telegram Passport Data Decryption - FAQ

What is `PassportDataDecryptionException`

Methods on `IDecrypter` might throw `PassportDataDecryptionException` exception if an error happens during decryption. The exception message tells you what went wrong but there is not much you can do to resolve it. Maybe let your user know the issue and ask for Passport data again.

It is important to pass each piece of encrypted data, e.g. Id Document, Passport File, etc., with the right accompanying credentials to decryption methods.

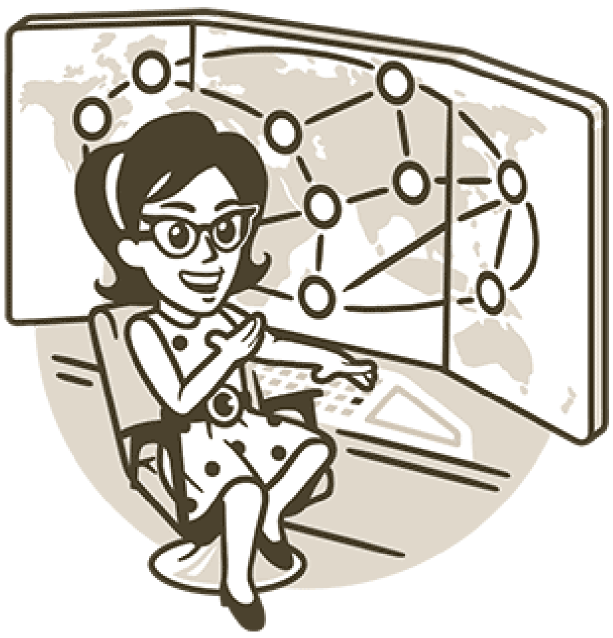
Spot the *problem in this code* decrypting driver's license files:

```
byte[] selfieContent = decrypter.DecryptFile(
  encSelfieContent, // byte array of encrypted selfie file
  credentials.SecureData.DriverLicense.FrontSide // WRONG! use selfie file
credentials
);
// throws PassportDataDecryptionException: "Data hash mismatch at position 123."
```

Working Behind a Proxy

`TelegramBotClient` allows you to use a proxy for Bot API connections. This guide covers using three different proxy solutions.

- HTTP Proxy
- SOCKS5 Proxy
- SOCKS5 Proxy over Tor (Testing Only)



If you are in a country, such as Iran, where HTTP and SOCKS proxy connections to Telegram servers are blocked, consider using a VPN, using Tor Network, or hosting your bot in other jurisdictions.

HTTP Proxy

You can pass an `IWebProxy` to bot client for HTTP Proxies.

```
// using System.Net;

var httpProxy = new WebProxy(address: "https://example.org", port: 8080) {
  // Credentials if needed:
  Credentials = new NetworkCredential("USERNMAE", "PASSWORD")
};
var botClient = new TelegramBotClient("YOUR_API_TOKEN", httpProxy);
```

SOCKS5 Proxy

Unfortunately, there is no built-in support for socks proxies in the .NET Standard libraries. You can use an external NuGet package: `HttpToSocks5Proxy` provided by one of our team members.

```
// using MihaZupan;

var proxy = new HttpToSocks5Proxy(Socks5ServerAddress, Socks5ServerPort);

// Or if you need credentials for your proxy server:
var proxy = new HttpToSocks5Proxy(
  Socks5ServerAddress, Socks5ServerPort, "USERNAME", "PASSWORD"
);

// Allows you to use proxies that are only allowing connections to Telegram
// Needed for some proxies
proxy.ResolveHostnamesLocally = true;

var botClient = new TelegramBotClient("YOUR_API_TOKEN", proxy);
```

SOCKS5 Proxy over Tor

Warning: Use for Testing only!

> Do not use this method in a production environment as it has high network latency and poor bandwidth.

Using Tor, a developer can avoid network restrictions while debugging and testing the code before a production release.

1. Install Tor Browser
2. Open the `torcc` file with a text editor (Found in `Tor Browser\Browser\TorBrowser\Data\Tor`)
3. Add the following lines: (configurations are described below)

   ```
   EntryNodes {NL}
   ExitNodes {NL}
   StrictNodes 1
   SocksPort 127.0.0.1:9050
   ```

4. Look at the Socks5 proxy example above.
5. Start the Tor Browser

Usage:

```
// using MihaZupan;

var botClient = new TelegramBotClient(
  "YOUR_API_TOKEN",
  new HttpToSocks5Proxy("127.0.0.1", 9050)
);
```

> Note that Tor has to be active at all times for the bot to work.

Configurations in `torcc`

```
EntryNodes {NL}
ExitNodes {NL}
StrictNodes 1
```

These three lines make sure you use nodes from the Netherlands as much as possible to reduce latency.

`SocksPort 127.0.0.1:9050`

This line tells tor to listen on port 9050 for any socks connections. You can change the port to anything you want (9050 is just the default), only make sure to use the same port in your code.

Date and Time

All `DateTime` values are now in UTC format. Here are some examples of usage:

```
// Use UTC time when making a request
await BotClient.KickChatMemberAsync(
  chatId: -9876,
  userId: 1234,
  untilDate: DateTime.UtcNow.AddDays(2)
);
```

```
// Convert to local time (not recommended though)
DateTime localTime = update.Message.Date.ToLocalTime();
```

Keyboard Buttons

Many keyboard button types are removed from project. It is more convenient to use factory methods on `KeyboardButton` and `InlineKeyboardButton` classes.

Here are some examples:

```
// Message having an inline keyboard button with URL that redirects to a page
await BotClient.SendTextMessageAsync(
  chatId: -9876,
  text: "Check out the source code",
  replyMarkup: new InlineKeyboardMarkup(
    InlineKeyboardButton.WithUrl("Repository",
"https://github.com/TelegramBots/Telegram.Bot")
  )
);
```

```
// Message to a private chat having a 2-row reply keyboard
await BotClient.SendTextMessageAsync(
  chatId: 1234,
  text: "Share your contact & location",
  replyMarkup: new ReplyKeyboardMarkup(
    new [] { KeyboardButton.WithRequestContact("Share Contact") },
    new [] { KeyboardButton.WithRequestLocation("Share Location") },
  )
);
```

`GetFileAsync()`

Downloading a file from Telegram Bot API has 2 steps (see docs here):

1. Get file info by calling `getFile`
2. Download file from `https://api.telegram.org/file/bot<token>/<file_path>`

`GetFileAsync()` is replaced by 3 methods. Method `GetInfoAndDownloadFileAsync()` looks very similar to old `GetFileAsync()`:

```
// Gets file info and saves it to "path/to/file.pdf"
using (var fileStream = System.IO.File.OpenWrite("path/to/file.pdf"))
{
  File fileInfo = await BotClient.GetInfoAndDownloadFileAsync(
    fileId: "BsdfgLg4Khdlsn-bldBD",
    destination: fileStream
  );
}
```

> Note that calling the method `GetInfoAndDownloadFileAsync()` results in 2 HTTP requests (steps 1 and 2 above) being sent to the Bot API.

There are two more methods that assist you with downloading files:

```
// New version of GetFileAsync() only gets the file info (step 1)
File fileInfo = await BotClient.GetFileAsync("BsdfgLg4Khdlsn-bldBD");

// Download file from server (step 2)
using (var fileStream = System.IO.File.OpenWrite("path/to/file.pdf")) {
  await BotClient.DownloadFileAsync(
    filePath: fileInfo.FilePath,
    destination: fileStream
  );
}
```

`GetUpdatesAsync()`, `SetWebhookAsync()`

Value `All` is removed from enum `Telegram.Bot.Types.Enums.UpdateType`. In order to get all kind of updates, pass an empty list such as `new UpdateType[0]` for `allowedUpdates` argument.

`SetWebhookAsync()`

Parameter `url` is required. If you intend to remove the webhook, it is recommended to use `DeleteWebhookAsync()` instead. However, you could achieve the same result by passing `string.Empty` value to `url` argument.

`AnswerInlineQueryAsync()` and `InlineQueryResult`

Classes `InlineQueryResultNew` and `InlineQueryResultCache` are removed. `InlineQueryResult` has become the only shared base type for all inline query result classes.

Many shared and redundant properties are removed. This might require significant changes to your `.cs` files if your bot is in *inline mode*. Fortunately, all input query results now have constructors with only the required properties as their parameters. This is the preferred way to instantiate input query result instances e.g.:

Instead of:

```
// bad way. easy to get exceptions
documentResult = new InlineQueryResultDocument
{
  Id = "some-id",
  Url = "https://example.com/document.pdf",
  Title = "Some title",
  MimeType = "application/pdf"
};
```

You should use:

```
// good way
documentResult = new InlineQueryResultDocument(
  id: "some-id",
  documentUrl: "https://example.com/document.pdf",
  title: "Some title",
  mimeType: "application/pdf"
);
```

`SendMediaGroupAsync()`

`InputMediaType` is renamed to `InputMedia`.

> *ToDo*

Inline Message Overloads

Many inline message methods have been replaced with their overloads.

- `EditInlineMessageTextAsync` --> `EditMessageTextAsync`

> *ToDo*

`FileToSend`

New classes have replaced `FileToSend` struct.

- `InputFileStream`:
- `InputTelegramFile`:
- `InputOnlineFile`:

In many cases, you can use implicit casting to pass parameters.

```
Stream stream = System.IO.File.OpenRead("photo.png");
message = await BotClient.SendPhotoAsync("chat id", stream);

string fileId = "file_id on Telegram servers";
message = await BotClient.SendPhotoAsync("chat id", fileId);
```

> *ToDo*. implicit casts

`UpdateType` and `MessageType`

Values in these two enums are renamed e.g. `UpdateType.MessageUpdate` is now `UpdateType.Message`.

`MessageType.Service` is removed. Now each type of message has its own `MessageType` value e.g. when a chat member leaves a group, corresponding update contains a message type of `MessageType.ChatMemberLeft` value.

`VideoNote`

Properties `Width` and `Height` are removed. Vide notes are squared and `Length` property represents both width and height.

Constructor Parameters Instead of Public Setters

Many types now have the required parameters in their constructors. To avoid running into problems or getting exceptions, we recommend providing all required values in the constructor e.g.:

```
//bad way:
markup = new InlineKeyboardMarkup {
  Keyboard = buttonsArray,
  ResizeKeyboard = true
};

// better:
markup = new InlineKeyboardMarkup(buttonsArray) {
  ResizeKeyboard = true
};
```

How do I use an HTTP/Socks proxy?

Look at the wiki page: Working Behind a Proxy.

I got a '409' error. What do I do?

You are trying to receive updates multiple times at the same time. Either you are calling GetUpdates from two instances of the bot, or you are calling GetUpdates while a web hook is already set. That is not supported by the API, only receive on one instance.

How do I get the user id from a username?

There is no way to do that with the API directly. You could store a list of known usernames, mapped to ids. This is *not* recommended, because usernames can be changed.

How do I get updates in channels?

If you are using polling, you will have to subscribe to the `OnUpdate` event. Check the `UpdateType` of the `Update`. If it is `UpdateType.ChannelPost` then the `Update.ChannelPost` property will be set.

This FAQ doesn't have my question on it. Where can I get my torch and pitchfork?

Check the `Bots FAQ by Telegram` and if that doesn't pan out, feel free to let us know in the public group chat.